SKAI WORLDWIDE

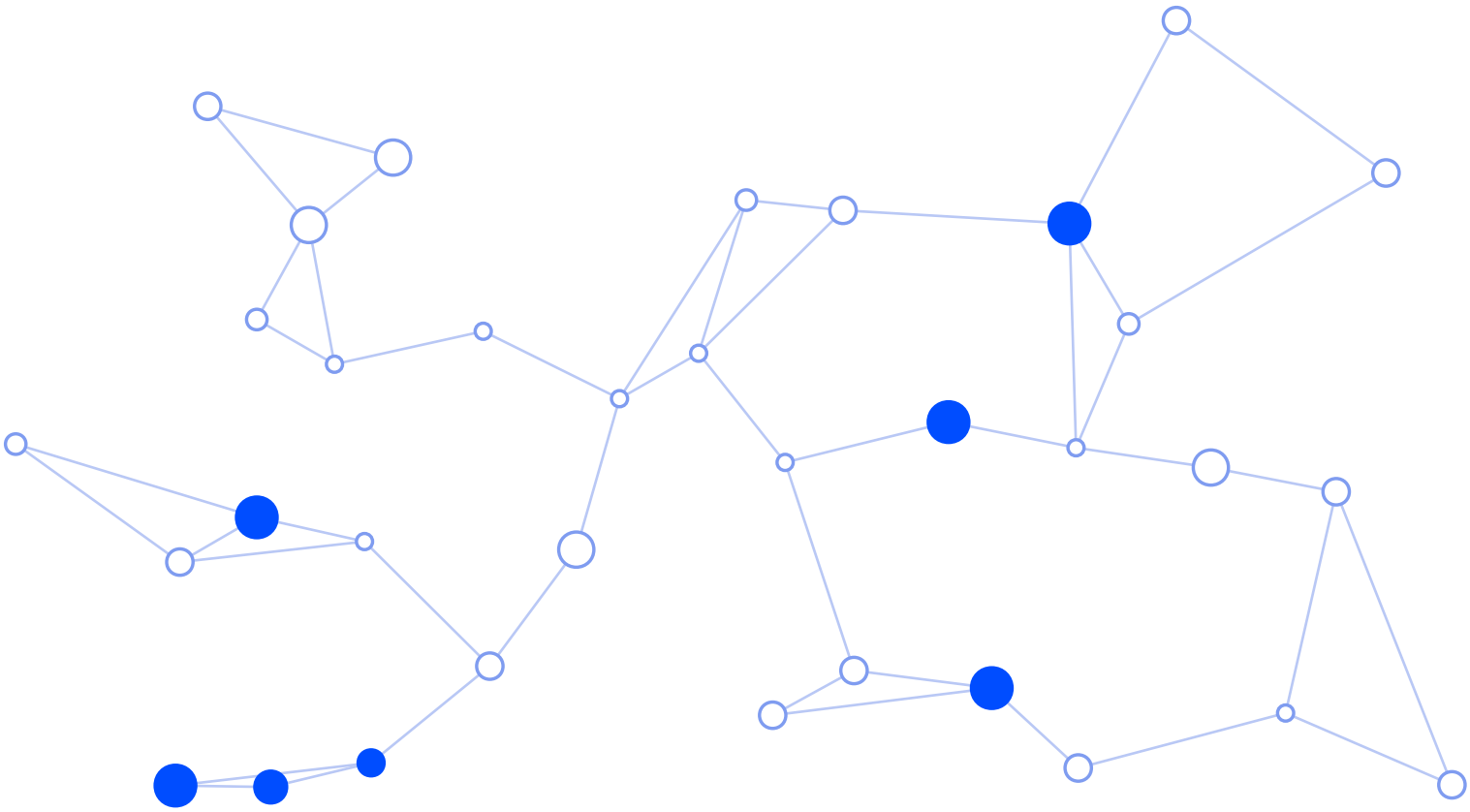
TECHNICAL DOCUMENTATION

AgensGraph 2.18

Documentation

A multi-model database on PostgreSQL: one graph, queried in Cypher, GQL or SQL.

Release 2.18.6.1



Contents

Get Started **5**
- What AgensGraph is 5
- Install AgensGraph 6
- The example dataset 9
- Your first query 15

Concepts **20**
- The property graph model 20
- How a graph is stored 23
- Label inheritance 27
- Cypher, GQL and SQL 29

Querying **32**
- Reading a graph 32
- Writing a graph 39
- Composing a query 45
- Paths 52
- Indexes and constraints 59
- Promoting a property to a column 69
- Combining SQL and Cypher 79
- Tuning a query 86

Reference **96**
- Syntax 96
- Data types 107
- Clauses 117
- Data Definition 220
- Functions 279
- Server settings 342
- The agens client 355

Guides **363**
- Loading data 363
- Vector search 367
- Full-text search 372
- Hybrid search 376
- AI and LLM integrations 380

Drivers and client libraries **382**
- The Python driver 382
- Other languages 382
- Connection and parameters 382

Operations **384**
- Deployment 384
- Configuration 386
- Security 389

© 2026 SKAI Worldwide Co., Ltd.

Backup and recovery ... 393
Server-side tools ... 395

System Catalogs ... 398
Graph catalogs ... 398
Graph views ... 401
Graph metadata ... 403

Release Notes ... 414
AgensGraph 2.18 ... 414
Upgrading to 2.18 ... 418
Upgrade walkthrough ... 423
AgensGraph 2.17.0 ... 428

Appendixes ... 430
GQL conformance ... 430
Cypher compatibility ... 440
Keywords ... 451
Error codes ... 454
Limits ... 456
Glossary ... 458

© 2026 SKAI Worldwide Co., Ltd.

AgensGraph 2.18

AgensGraph is an open-source multi-model database built on PostgreSQL, developed and maintained by **SKAI Worldwide** (formerly Bitnine) under the Apache License 2.0. It stores your data as a **property graph** and as relational tables at the same time. You query that data in **Cypher**, in **GQL** (ISO/IEC 39075), or in **SQL**, and you can combine two of them in one statement when that is the shortest way to the answer.

This manual covers AgensGraph 2.18.6.1. Release 2.18 is built on PostgreSQL 18 and adds a substantial part of the GQL standard. **AgensGraph 2.18** lists what changed, including the changes your application needs to know about.

How this manual is organized

Get Started installs the server, loads the small film graph that every example in this manual uses, and gets you to a result. Read it first.

Concepts explains the property-graph model, and how a graph is laid out in PostgreSQL. A graph is a schema and a label is a table. That is what lets one statement mix graph and relational work.

Querying teaches the language in the order you meet it: reading, writing, composing pipelines, walking paths, indexing, and crossing between SQL and Cypher.

Reference is the formal definition of the same language — the lexical rules, the types, and a page for each clause, statement and function. Look here when you know what you want and need the exact syntax.

Guides covers loading data and the search features. **Drivers and client libraries** points at the client libraries. **Operations** is for running a server. **System Catalogs** documents the catalogs that describe a graph, and **Appendixes** holds the GQL conformance tables, the compatibility notes, the keyword and error-code lists, and a map from the 2.17 page names to these.

© 2026 SKAI Worldwide Co., Ltd.

Where to go next

Get Started

Install AgensGraph, load the example graph, run your first query.

Concepts

The property-graph model, and how a graph is stored in PostgreSQL.

Querying

Reading, writing, paths, indexes, and mixing SQL with Cypher.

Reference

Clauses, data definition, functions, types, and settings.

Guides

Loading data, vector, full-text and hybrid search, and the AI integrations.

Drivers

Any PostgreSQL driver connects, and the Python driver adds the graph types.

Operations

Deploy, configure, secure and back up a server.

System Catalogs

The catalogs and views that describe a graph.

Release Notes

What changed in 2.18, and how to upgrade to it.

© 2026 SKAI Worldwide Co., Ltd.

Get Started

Install AgensGraph, load the graph the rest of this manual uses, and run your first query.

What AgensGraph is

AgensGraph is a multi-model database. One server, one database, one connection. Your data lives there as a **property graph** and as **relational tables** at the same time, and you query it with **Cypher**, **GQL** or **SQL**. You can use two of them in one statement.

It is built on PostgreSQL. Release 2.18 is PostgreSQL 18 plus the graph model, so everything PostgreSQL gives you comes with it unchanged. That is transactions, roles and row-level security, `pg_dump` and `pg_basebackup`, streaming replication, the planner and `EXPLAIN`, the extension ecosystem, and the wire protocol every PostgreSQL driver already speaks.

What makes it different

A graph is stored as PostgreSQL objects, not beside them. A graph is a schema, a label is a table, a vertex is a row. There is no separate graph store to keep in step, and no bridge to cross. A Cypher write and a SQL write in one transaction commit together, because both are writes to tables in the same database. **How a graph is stored** is the page that makes the rest of this manual make sense.

Traversal is a first-class query, not a recursive join you write yourself. "Which films share an actor with this one", "how far apart are these two people" and "what is reachable in three hops" are patterns:

```
MATCH (:person {name: 'Keanu Reeves'})-[:acted_in]->(:movie)<-[:acted_in]-(co:person)
RETURN DISTINCT co.name AS co_actor ORDER BY co_actor;
```

```
      co_actor
____________________
 "Carrie-Anne Moss"
 "Laurence Fishburne"
```

Extensions apply to graph data. A property is `jsonb` in an ordinary table. So `pgvector` will index an embedding stored on a vertex, and PostgreSQL's full-text search will index a text property. One query can then find semantically similar documents and walk the graph out from them. See **vector search**.

GQL, the ISO standard, is part of the same language. 2.18 adds a substantial part of ISO/IEC 39075, with no separate mode and no separate parser. `LET`, `NEXT`, `FILTER`, `FINISH`, `INSERT`, `FOR`, `CALL { }` and the subquery value expressions are accepted wherever Cypher is. See **GQL conformance**.

Who it is for

Teams that have relational data, have a graph-shaped question about it, and do not want to run a second database to ask it. The common shapes are recommendation and similarity, fraud and risk chains, permission and ownership hierarchies, network and dependency topologies, and knowledge graphs behind retrieval-augmented generation.

It is a good fit in three cases. When the graph and the tables belong to the same system of record. When you already operate PostgreSQL and want to keep those skills and tools. And when your queries mix traversal with aggregation, joins or window functions.

© 2026 SKAI Worldwide Co., Ltd.

It is the wrong tool if you need a distributed graph store that shards one graph across machines, or a graph-analytics engine for whole-graph algorithms such as PageRank or community detection at scale. AgensGraph scales the way PostgreSQL scales: one primary, read replicas, and vertical growth.

What is in this release

- PostgreSQL 18 as the base.
- A large step toward GQL conformance.
- Typed-column property promotion, which turns a hot `jsonb` property into a real indexable column without changing how you write queries. See **Promoting a property to a column**.
- Planner and estimation work on graph traversal.

Several changes need an application's attention before you upgrade, including new reserved words and a stricter rule about what may appear in a `WHERE`. **Release notes** lists them, and **Upgrading to 2.18** has the procedure.

Next

Install AgensGraph, then **load the example dataset** that every page in this manual uses, then **run your first query**.

Install AgensGraph

Three routes. Docker is the fastest way to a server you can query. A package is the usual choice for a workstation or a single server. Build from source if you need to change something, or if your platform has no package.

If you only want to try AgensGraph, use Docker and skip to **the example dataset**.

Docker

```
docker run --name agensgraph -e POSTGRES_PASSWORD=agensgraph -d skaiworldwide/
agensgraph:v2.18.6.1-bookworm
```

Then open a client inside the container:

```
docker exec -it agensgraph agens -U agens -d postgres
```

The image accepts the same environment variables as the official `postgres` image — `POSTGRES_USER`, `POSTGRES_DB`, `PGDATA` and the rest — so publish a port and mount a volume the usual way:

```
docker run -d --name agensgraph \
  -e POSTGRES_PASSWORD=agensgraph \
  -p 5432:5432 \
  -v agensgraph-data:/var/lib/postgresql \
  skaiworldwide/agensgraph:v2.18.6.1-bookworm
```

Tags follow the release plus a base image: `v2.18.6.1-trixie`, `v2.18.6.1-bookworm`, `v2.18.6.1-rocky10`, `v2.18.6.1-rocky9`, `v2.18.6.1-alpine3.24`, `v2.18.6.1-alpine3.23`. Pin one; `skaiworldwide/agensgraph` on its own is a moving target.

© 2026 SKAI Worldwide Co., Ltd.

WARNING

The 2.18.6.1 images follow PostgreSQL 18's layout change. `PGDATA` now defaults to `/var/lib/postgresql/18/docker`, and the declared volume moved from `/var/lib/postgresql/data` to `/var/lib/postgresql`. A host volume that an earlier tag initialised will not be found by a 2.18.6.1 container unless you point at it explicitly:

```
-e PGDATA=/var/lib/postgresql/data -v /custom/mount:/var/lib/postgresql/data
```

Read **Upgrading to 2.18** before reusing an existing data directory in any case — a 2.17 data directory cannot be started by 2.18.

Packages

Download a build for your platform from the **AgensGraph download page** and unpack it where you want it, for example `/usr/local/AgensGraph`:

```
tar xvf agensgraph-v2.18.6.1-build.tar.gz -C /usr/local
```

If you are not sure which build you need:

```
uname -sm
```

Then continue with **Set up a database**.

Build from source

Install the build dependencies

Ubuntu / Debian

```
sudo apt-get install build-essential libreadline-dev zlib1g-dev flex bison
```

Rocky / RHEL

```
sudo yum install gcc glibc glib-common readline readline-devel zlib zlib-devel flex bison perl
```

Fedora

```
sudo dnf install gcc glibc bison flex readline readline-devel zlib zlib-devel
```

macOS

```
xcode-select --install
```

Build and install

```
git clone https://github.com/skaiworldwide-oss/agensgraph.git
cd agensgraph
```


© 2026 SKAI Worldwide Co., Ltd.

```
./configure --prefix=$(pwd)
make install
```

`--prefix` decides where the binaries land. Without it, `make install` writes to `/usr/local/pgsql`. Pointing it at the source tree, as above, keeps everything in one place while you are trying things out.

To build the contrib modules as well — `pgvector` is not among them, but `file_fdw`, which the **loading guide** uses, is:

```
make install-world
```

If `configure` cannot find a header it needs, tell it where to look with `--with-includes=/path/to/headers`. `./configure --help` lists every option; they are PostgreSQL's, and **PostgreSQL's installation chapter** documents them.

Put the binaries on your path

```
export PATH=/path/to/agensgraph/bin:$PATH
export LD_LIBRARY_PATH=/path/to/agensgraph/lib:$LD_LIBRARY_PATH
```

Add those to `~/.bashrc`, or `~/.zshrc` on macOS, to make them stick.

Set up a database

The tools are PostgreSQL's, with two renamed: `agens` is the interactive client and `ag_ctl` starts and stops the server. See **The agens client**.

Create a data directory:

```
initdb -D /path/to/data
```

Start the server:

```
ag_ctl -D /path/to/data -l /path/to/data/server.log start
```

Create a database:

```
createdb mydb
```

Setting `AGDATA` lets you drop the `-D` from later commands:

```
export AGDATA=/path/to/data
```

Verify the install

Connect:

```
agens mydb
```

Ask the server what it is:

```
SELECT version();
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                        version
-------------------------------------------------------------------------------------
 PostgreSQL 18.6 (AgensGraph 2.18.6.1) on x86_64-pc-linux-gnu, compiled by gcc ...
```

Then check that the graph half is there, which is the part a plain PostgreSQL build would fail:

```
CREATE GRAPH check_install;
SET graph_path = check_install;
CREATE (:t {ok: true});
MATCH (n:t) RETURN n;
```

```
         n
---------------------
 t[3.1]{"ok": true}
```

```
DROP GRAPH check_install CASCADE;
```

`CASCADE` is needed because a graph owns the labels and sequences inside it.

Next

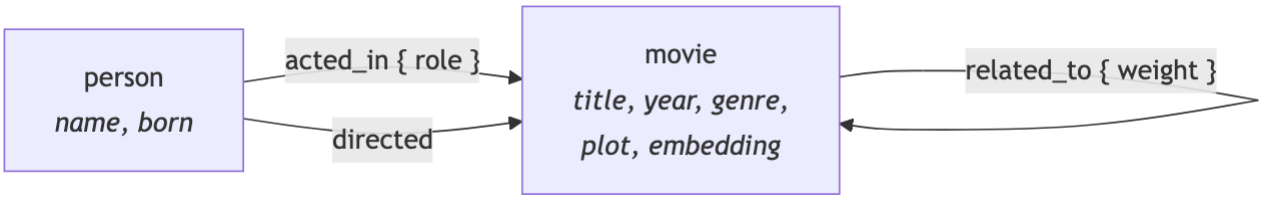
Load the example dataset, which every page in this manual uses, then **run your first query**.

For a production install — where the data directory goes, what to set in `postgresql.conf`, how to run under systemd, and how to lock it down — see **Deployment** and **Configuration**.

The example dataset

Every example in this manual runs against one small graph of films and the people who made them, in a graph named `moviekb`. Set it up once and every page will work.

The graph has two vertex labels and three edge labels:



Alongside it there is an ordinary table, `rating`, so the pages on **combining SQL and Cypher** have real relational data to join against.

Create the graph

```
CREATE GRAPH moviekb;
SET graph_path = moviekb;

CREATE VLABEL movie;
CREATE VLABEL person;
CREATE ELABEL acted_in;
```


© 2026 SKAI Worldwide Co., Ltd.

```
CREATE ELABEL directed;
CREATE ELABEL related_to;
```

`graph_path` tells the session which graph unqualified label names belong to. It is a per-session setting, so set it again in every new session, or set it for the database:

```
ALTER DATABASE mydb SET graph_path = moviekb;
```

Load the films

Each film has a `plot` and an `embedding` of that plot. Real embeddings come from a model and have hundreds of dimensions; these are reduced to four so the geometry stays readable on the page. **Vector search** and **hybrid search** use them as they are.

```
CREATE (:movie {title: 'The Matrix', year: 1999, genre: 'Action', plot: 'A computer hacker
learns about the true nature of reality and joins a rebellion to free humanity from a
simulated world controlled by machines.', embedding: [-0.07594558, 0.04081754, 0.29592122, -0.
11921061]}),
       (:movie {title: 'The Matrix Reloaded', year: 2003, genre: 'Action', plot: 'The rebels
continue their fight against the machines, uncovering deeper truths about the Matrix and the
nature of their mission.', embedding: [0.30228977, -0.22839354, 0.35070436, 0.01262819]}),
       (:movie {title: 'The Matrix Revolutions', year: 2003, genre: 'Action', plot:
'The final battle between humans and machines reaches its climax as the fate of both worlds
hangs in the balance.', embedding: [0.12240622, -0.29752459, 0.22620453, 0.24454723]}),
       (:movie {title: 'The Matrix Resurrections', year: 2021, genre: 'Action', plot: 'Neo
returns to a new version of the Matrix and must once again fight to save the people from the
control of the machines.', embedding: [0.34717246, -0.13820869, 0.29214213, 0.08090488]}),
       (:movie {title: 'Inception', year: 2010, genre: 'Sci-Fi', plot: 'A skilled thief is
given a chance at redemption if he can successfully perform an inception: planting an idea
into someone''s subconscious.', embedding: [0.03923657, 0.39284106, -0.20927092, -0.17770818]}
),
       (:movie {title: 'Interstellar', year: 2014, genre: 'Sci-Fi', plot: 'A group of
explorers travel through a wormhole in space in an attempt to ensure humanity''s survival.', e
mbedding: [-0.29302418, -0.39615033, -0.23393948, -0.09601383]}),
       (:movie {title: 'Avatar', year: 2009, genre: 'Sci-Fi', plot: 'A paraplegic Marine is
sent to the moon Pandora, where he becomes torn between following orders and protecting the
world he feels is his home.', embedding: [-0.13663386, 0.00635589, -0.03038832, -0.08252723]})
,
       (:movie {title: 'Blade Runner', year: 1982, genre: 'Sci-Fi', plot:
'A blade runner must pursue and terminate four replicants who have stolen a ship in space and
returned to Earth.', embedding: [0.27215557, -0.1479577, -0.09972772, -0.08234394]}),
       (:movie {title: 'Blade Runner 2049', year: 2017, genre: 'Sci-Fi', plot: 'A new blade
runner unearths a long-buried secret that has the potential to plunge what''s left of society
into chaos.', embedding: [0.21560573, -0.07505179, -0.01331814, 0.13403069]}),
       (:movie {title: 'Minority Report', year: 2002, genre: 'Sci-Fi', plot: 'In a future
where a special police unit can arrest murderers before they commit their crimes, a top
officer is accused of a future murder.', embedding: [0.24008012, 0.44954908, -0.30905488, 0.15
195407]}),
       (:movie {title: 'Total Recall', year: 1990, genre: 'Sci-Fi', plot: 'A construction
worker discovers that his memories have been implanted and becomes embroiled in a conspiracy
on Mars.', embedding: [-0.17471036, 0.14695261, -0.06272433, -0.21795064]}),
       (:movie {title: 'Elysium', year: 2013, genre: 'Sci-Fi', plot: 'In a future where the
rich live on a luxurious space station while the rest of humanity lives in squalor, a man
fights to bring equality.', embedding: [-0.33280967, 0.07733926, 0.11015328, 0.53382836]}),
       (:movie {title: 'Gattaca', year: 1997, genre: 'Sci-Fi', plot: 'In a future where
genetic engineering determines social class, a man defies his fate to achieve his dreams.', em
bedding: [-0.21629286, 0.31114665, 0.08303899, 0.46199759]}),
       (:movie {title: 'The Fifth Element', year: 1997, genre: 'Sci-Fi', plot: 'In a
futuristic world, a cab driver becomes the key to saving humanity from an impending cosmic
threat.', embedding: [-0.11528205, -0.0208782, -0.0735215, 0.14327449]}),
       (:movie {title: 'The Terminator', year: 1984, genre: 'Action', plot:
```


© 2026 SKAI Worldwide Co., Ltd.

```
'A cyborg assassin is sent back in time to kill the mother of the future resistance leader.',
embedding: [0.33666933, 0.18040994, -0.01075103, -0.11117851]}),
       (:movie {title: 'Terminator 2: Judgment Day', year: 1991, genre: 'Action', plot: 'A
reprogrammed Terminator is sent to protect the future leader of the human resistance from a
more advanced Terminator.', embedding: [0.34698868, 0.06439331, 0.06232323, -0.19534876]}),
       (:movie {title: 'Jurassic Park', year: 1993, genre: 'Adventure', plot: 'Scientists
clone dinosaurs to create a theme park, but things go awry when the creatures escape.', embedd
ing: [0.01794725, -0.11434246, -0.46831815, -0.01049593]}),
       (:movie {title: 'The Avengers', year: 2012, genre: 'Action', plot: 'Superheroes
assemble to face a global threat from an alien invasion led by Loki.', embedding:
[0.00546514, -0.37005171, -0.42612838, 0.07968612]});
```

Load the people

```
CREATE (:person {name: 'Keanu Reeves', born: 1964}),
       (:person {name: 'Laurence Fishburne', born: 1961}),
       (:person {name: 'Carrie-Anne Moss', born: 1967}),
       (:person {name: 'Lana Wachowski', born: 1965}),
       (:person {name: 'Lilly Wachowski', born: 1967}),
       (:person {name: 'Leonardo DiCaprio', born: 1974}),
       (:person {name: 'Matthew McConaughey', born: 1969}),
       (:person {name: 'Christopher Nolan', born: 1970}),
       (:person {name: 'Harrison Ford', born: 1942}),
       (:person {name: 'Ryan Gosling', born: 1980}),
       (:person {name: 'Ridley Scott', born: 1937}),
       (:person {name: 'Denis Villeneuve', born: 1967}),
       (:person {name: 'Tom Cruise', born: 1962}),
       (:person {name: 'Sam Neill', born: 1947}),
       (:person {name: 'Steven Spielberg', born: 1946}),
       (:person {name: 'Arnold Schwarzenegger', born: 1947}),
       (:person {name: 'Linda Hamilton', born: 1956}),
       (:person {name: 'James Cameron', born: 1954}),
       (:person {name: 'Sam Worthington', born: 1976}),
       (:person {name: 'Paul Verhoeven', born: 1938}),
       (:person {name: 'Matt Damon', born: 1970}),
       (:person {name: 'Neill Blomkamp', born: 1979}),
       (:person {name: 'Ethan Hawke', born: 1970}),
       (:person {name: 'Andrew Niccol', born: 1964}),
       (:person {name: 'Bruce Willis', born: 1955}),
       (:person {name: 'Luc Besson', born: 1959}),
       (:person {name: 'Robert Downey Jr.', born: 1965}),
       (:person {name: 'Joss Whedon', born: 1964});
```

Connect them

```
-- Cast: acted_in carries the character name as an edge property.
MATCH (p:person {name:'Keanu Reeves'}),          (m:movie {title:'The
Matrix'})                CREATE (p)-[:acted_in {role:'Neo'}]->(m);
MATCH (p:person {name:'Keanu Reeves'}),          (m:movie {title:'The Matrix
Reloaded'})        CREATE (p)-[:acted_in {role:'Neo'}]->(m);
MATCH (p:person {name:'Keanu Reeves'}),          (m:movie {title:'The Matrix
Revolutions'})     CREATE (p)-[:acted_in {role:'Neo'}]->(m);
MATCH (p:person {name:'Keanu Reeves'}),          (m:movie {title:'The Matrix
Resurrections'})   CREATE (p)-[:acted_in {role:'Neo'}]->(m);
MATCH (p:person {name:'Laurence Fishburne'}),    (m:movie {title:'The
Matrix'})                CREATE (p)-[:acted_in {role:'Morpheus'}]->(m);
MATCH (p:person {name:'Laurence Fishburne'}),    (m:movie {title:'The Matrix
Reloaded'})        CREATE (p)-[:acted_in {role:'Morpheus'}]->(m);
MATCH (p:person {name:'Carrie-Anne Moss'}),      (m:movie {title:'The
Matrix'})                CREATE (p)-[:acted_in {role:'Trinity'}]->(m);
```


© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person {name:'Carrie-Anne Moss'}),      (m:movie {title:'The Matrix
Resurrections'})   CREATE (p)-[:acted_in {role:'Trinity'}]->(m);
MATCH (p:person {name:'Leonardo DiCaprio'}),     (m:movie
{title:'Inception'})                  CREATE (p)-[:acted_in {role:'Dom Cobb'}]->(m);
MATCH (p:person {name:'Matthew McConaughey'}),   (m:movie
{title:'Interstellar'})               CREATE (p)-[:acted_in {role:'Cooper'}]->(m);
MATCH (p:person {name:'Harrison Ford'}),         (m:movie {title:'Blade
Runner'})               CREATE (p)-[:acted_in {role:'Rick Deckard'}]->(m);
MATCH (p:person {name:'Harrison Ford'}),         (m:movie {title:'Blade Runner
2049'})          CREATE (p)-[:acted_in {role:'Rick Deckard'}]->(m);
MATCH (p:person {name:'Ryan Gosling'}),          (m:movie {title:'Blade Runner
2049'})          CREATE (p)-[:acted_in {role:'K'}]->(m);
MATCH (p:person {name:'Tom Cruise'}),            (m:movie {title:'Minority
Report'})            CREATE (p)-[:acted_in {role:'John Anderton'}]->(m);
MATCH (p:person {name:'Sam Neill'}),             (m:movie {title:'Jurassic
Park'})              CREATE (p)-[:acted_in {role:'Alan Grant'}]->(m);
MATCH (p:person {name:'Arnold Schwarzenegger'}), (m:movie {title:'The
Terminator'})             CREATE (p)-[:acted_in {role:'The Terminator'}]->(m);
MATCH (p:person {name:'Arnold Schwarzenegger'}), (m:movie {title:'Terminator 2: Judgment Day'}
) CREATE (p)-[:acted_in {role:'The Terminator'}]->(m);
MATCH (p:person {name:'Arnold Schwarzenegger'}), (m:movie {title:'Total
Recall'})               CREATE (p)-[:acted_in {role:'Douglas Quaid'}]->(m);
MATCH (p:person {name:'Linda Hamilton'}),        (m:movie {title:'The
Terminator'})             CREATE (p)-[:acted_in {role:'Sarah Connor'}]->(m);
MATCH (p:person {name:'Linda Hamilton'}),        (m:movie {title:'Terminator 2: Judgment Day'}
) CREATE (p)-[:acted_in {role:'Sarah Connor'}]->(m);
MATCH (p:person {name:'Sam Worthington'}),       (m:movie
{title:'Avatar'})                    CREATE (p)-[:acted_in {role:'Jake Sully'}]->(m);
MATCH (p:person {name:'Matt Damon'}),            (m:movie
{title:'Elysium'})                   CREATE (p)-[:acted_in {role:'Max Da Costa'}]->(m);
MATCH (p:person {name:'Ethan Hawke'}),           (m:movie
{title:'Gattaca'})                   CREATE (p)-[:acted_in {role:'Vincent Freeman'}]->(m);
MATCH (p:person {name:'Bruce Willis'}),          (m:movie {title:'The Fifth
Element'})          CREATE (p)-[:acted_in {role:'Korben Dallas'}]->(m);
MATCH (p:person {name:'Robert Downey Jr.'}),     (m:movie {title:'The
Avengers'})               CREATE (p)-[:acted_in {role:'Tony Stark'}]->(m);

-- Directors.
MATCH (p:person {name:'Lana Wachowski'}),    (m:movie {title:'The Matrix'})                CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Lilly Wachowski'}),   (m:movie {title:'The Matrix'})                CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Lana Wachowski'}),    (m:movie {title:'The Matrix Reloaded'})       CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Lilly Wachowski'}),   (m:movie {title:'The Matrix Reloaded'})       CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Lana Wachowski'}),    (m:movie {title:'The Matrix Revolutions'})    CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Lilly Wachowski'}),   (m:movie {title:'The Matrix Revolutions'})    CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Lana Wachowski'}),    (m:movie {title:'The Matrix Resurrections'})  CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Christopher Nolan'}), (m:movie {title:'Inception'})                 CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Christopher Nolan'}), (m:movie {title:'Interstellar'})              CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Ridley Scott'}),      (m:movie {title:'Blade Runner'})              CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Denis Villeneuve'}),  (m:movie {title:'Blade Runner 2049'})         CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Steven Spielberg'}),  (m:movie {title:'Minority Report'})           CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Steven Spielberg'}),  (m:movie {title:'Jurassic Park'})             CR
EATE (p)-[:directed]->(m);
```


© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person {name:'James Cameron'}),     (m:movie {title:'The Terminator'})            CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'James Cameron'}),     (m:movie {title:'Terminator 2: Judgment Day'}) CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'James Cameron'}),     (m:movie {title:'Avatar'})                    CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Paul Verhoeven'}),    (m:movie {title:'Total Recall'})              CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Neill Blomkamp'}),    (m:movie {title:'Elysium'})                   CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Andrew Niccol'}),     (m:movie {title:'Gattaca'})                   CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Luc Besson'}),        (m:movie {title:'The Fifth Element'})         CR
EATE (p)-[:directed]->(m);
MATCH (p:person {name:'Joss Whedon'}),       (m:movie {title:'The Avengers'})              CR
EATE (p)-[:directed]->(m);

-- related_to keeps the search pages' edges and adds a weight for weighted-path examples.
MATCH (a:movie {title:'The Matrix'}),             (b:movie {title:'The Matrix
Reloaded'})       CREATE (a)-[:related_to {weight: 1}]->(b);
MATCH (a:movie {title:'The Matrix Reloaded'}),    (b:movie {title:'The Matrix
Revolutions'})    CREATE (a)-[:related_to {weight: 1}]->(b);
MATCH (a:movie {title:'The Matrix Revolutions'}), (b:movie {title:'The Matrix
Resurrections'})  CREATE (a)-[:related_to {weight: 1}]->(b);
MATCH (a:movie {title:'The Matrix'}),             (b:movie {title:'Total
Recall'})             CREATE (a)-[:related_to {weight: 4}]->(b);
MATCH (a:movie {title:'The Matrix'}),             (b:movie {title:'The
Terminator'})           CREATE (a)-[:related_to {weight: 3}]->(b);
MATCH (a:movie {title:'Total Recall'}),           (b:movie
{title:'Inception'})                CREATE (a)-[:related_to {weight: 2}]->(b);
MATCH (a:movie {title:'The Terminator'}),         (b:movie {title:'Terminator 2: Judgment
Day'})CREATE (a)-[:related_to {weight: 1}]->(b);
MATCH (a:movie {title:'Blade Runner'}),           (b:movie {title:'Blade Runner
2049'})         CREATE (a)-[:related_to {weight: 1}]->(b);
MATCH (a:movie {title:'Interstellar'}),           (b:movie
{title:'Avatar'})                   CREATE (a)-[:related_to {weight: 5}]->(b);
MATCH (a:movie {title:'Jurassic Park'}),          (b:movie {title:'The
Avengers'})             CREATE (a)-[:related_to {weight: 5}]->(b);
MATCH (a:movie {title:'Minority Report'}),        (b:movie
{title:'Inception'})                CREATE (a)-[:related_to {weight: 2}]->(b);
```

Add the relational table

```
CREATE TABLE rating (title text, source text, score numeric(3,1));

INSERT INTO rating VALUES
  ('The Matrix', 'critics', 8.7),                ('The Matrix', 'audience', 9.1),
  ('The Matrix Reloaded', 'critics', 7.2),       ('The Matrix Revolutions', 'critics', 6.7),
  ('The Matrix Resurrections', 'critics', 5.7),  ('Inception', 'critics', 8.8),
  ('Inception', 'audience', 9.0),                ('Interstellar', 'critics', 8.7),
  ('Interstellar', 'audience', 8.6),             ('Blade Runner', 'critics', 8.1),
  ('Blade Runner 2049', 'critics', 8.0),         ('Jurassic Park', 'critics', 8.2),
  ('The Terminator', 'critics', 8.0),            ('Terminator 2: Judgment Day', 'critics',
8.6),
  ('Avatar', 'critics', 7.9),                    ('Total Recall', 'critics', 7.5),
  ('Minority Report', 'critics', 7.6),           ('Gattaca', 'critics', 7.8),
  ('The Fifth Element', 'critics', 7.6),         ('Elysium', 'critics', 6.6),
  ('The Avengers', 'critics', 8.0);
```


© 2026 SKAI Worldwide Co., Ltd.

Check what you loaded

```
MATCH (n) RETURN label(n) AS label, count(*) AS n ORDER BY label;
```

label	n
`"movie"`	18
`"person"`	28

```
MATCH ()-[e]->() RETURN label(e) AS label, count(*) AS n ORDER BY label;
```

label	n
`"acted_in"`	25
`"directed"`	21
`"related_to"`	11

The same data as tables

A graph is a schema and a label is a table, so the graph you just created is also eight ordinary tables you can query with SQL. `\dGv` lists the vertex labels, `\dGe` the edge labels, and `\d` shows a label's shape:

```
SELECT id, properties->>'title' AS title FROM moviekb.movie ORDER BY id LIMIT 3;
```

```
 id  |         title
-----+-----------------------
 3.1 | The Matrix
 3.2 | The Matrix Reloaded
 3.3 | The Matrix Revolutions
```

An edge table carries the two endpoint ids as well:

```
SELECT id, start, "end", properties FROM moviekb.acted_in ORDER BY id LIMIT 3;
```

```
 id  | start | end |   properties
-----+-------+-----+-----------------
 5.1 | 4.1   | 3.1 | {"role": "Neo"}
 5.2 | 4.1   | 3.2 | {"role": "Neo"}
 5.3 | 4.1   | 3.3 | {"role": "Neo"}
```


© 2026 SKAI Worldwide Co., Ltd.

How a graph is stored explains what those columns are and which indexes come with them.

Start over

```
DROP GRAPH moviekb CASCADE;
```

```
DROP TABLE rating;
```

Your first query

This page walks the **example dataset** — 18 films, 28 people, and the edges between them. Load it first, then work down the page: each query builds on the one before it.

Everything here is Cypher. The equivalent GQL spellings are noted in the **reference**; **Combining SQL and Cypher** is where SQL comes back in.

Say which graph you mean

```
SET graph_path = moviekb;
```

That is the one piece of session state Cypher needs. Without it every label name is unresolvable. Now count everything in the graph:

```
MATCH (n) RETURN count(n) AS elements;
```

```
 elements
----------
       46
```

`MATCH (n)` is the emptiest possible pattern — one vertex, no label, no properties — so it matches every vertex. 46 is 18 films plus 28 people.

Find one thing

A pattern narrows in two ways: by label, and by property.

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.year AS year, m.genre AS genre;
```

```
 year |  genre
------+----------
 1999 | "Action"
```

`(m:movie {title: 'The Matrix'})` reads as: a vertex, labelled `movie`, whose `title` property is `'The Matrix'`, bound to the variable `m`. `RETURN` then projects out of `m`.

Note the quotes around `"Action"` and their absence around `1999`. A `RETURN` hands a property back as `jsonb`, so a string arrives JSON-quoted. That is worth knowing before you compare a result against a client-side string — see **Combining SQL and Cypher** for how to get a plain value.

Every vertex also has an identity, which is not a property:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.title AS title, id(m) AS id;
```

```
    title     | id
--------------+-----
 "The Matrix" | 3.1
```

`3.1` is a `graphid`: label 3, row 1. It is the vertex's identity, and `id(m)` is how you ask for it — `m.id` would look for a property called `id`, which no film has.

Follow an edge

The point of a graph is what comes next. Add an edge to the pattern:

```
MATCH (p:person)-[r:acted_in]->(:movie {title: 'The Matrix'})
RETURN p.name AS actor, r.role AS role
ORDER BY actor;
```

```
        actor         |    role
----------------------+------------
 "Carrie-Anne Moss"   | "Trinity"
 "Keanu Reeves"       | "Neo"
 "Laurence Fishburne" | "Morpheus"
```

Three things are new. `-[r:acted_in]->` is an edge, labelled `acted_in`, pointing from the person to the film. It is bound to `r`, so its properties are available: `r.role` is a property of the relationship, not of either endpoint. And `(:movie {title: 'The Matrix'})` has no variable, because the pattern needs to reach that film but the projection never mentions it.

Follow two

Extend the pattern in the other direction and you have asked a question SQL would need a self-join for:

```
MATCH (:person {name: 'Keanu Reeves'})-[:acted_in]->(:movie)<-[:acted_in]-(co:person)
RETURN DISTINCT co.name AS co_actor
ORDER BY co_actor;
```

```
      co_actor
---------------------
 "Carrie-Anne Moss"
 "Laurence Fishburne"
```

Read the pattern left to right: from Keanu Reeves, out along an `acted_in` edge to a film, then backwards along another `acted_in` edge to a second person. `DISTINCT` is needed because two people who share three films produce three rows.

Aggregate

Aggregates are PostgreSQL's, and they work the way you expect:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (d:person)-[:directed]->(m:movie)
RETURN d.name AS director, count(m) AS films
ORDER BY films DESC, director
LIMIT 5;
```

```
      director       | films
---------------------+-------
 "Lana Wachowski"    | 4
 "James Cameron"     | 3
 "Lilly Wachowski"   | 3
 "Christopher Nolan" | 2
 "Steven Spielberg"  | 2
```

Anything in `RETURN` that is not inside an aggregate becomes a grouping key, again as in SQL.

Ask for something that might not be there

`MATCH` drops a row when the pattern fails. `OPTIONAL MATCH` keeps it and fills the missing variables with null — a left join:

```
MATCH (d:person)-[:directed]->(:movie)
WITH DISTINCT d
OPTIONAL MATCH (d)-[:acted_in]->(m:movie)
RETURN d.name AS director, m.title AS also_acted_in
ORDER BY director
LIMIT 6;
```

```
      director       | also_acted_in
---------------------+---------------
 "Andrew Niccol"     |
 "Christopher Nolan" |
 "Denis Villeneuve"  |
 "James Cameron"     |
 "Joss Whedon"       |
 "Lana Wachowski"    |
```

Nobody in this dataset both directed and acted, so every `also_acted_in` is null — which is the point: with a plain `MATCH`, all six rows would have vanished.

`WITH` in the middle there is a pipeline divider. It ends one part of the query and starts the next, which is what lets `DISTINCT` apply to the directors before the optional match runs.

Walk an unknown number of hops

Ask how two people connect without knowing the route:

```
MATCH p = shortestpath((a:person {name: 'Keanu Reeves'})-[*..6]-(b:person {name: 'Arnold
Schwarzenegger'}))
RETURN length(p) AS hops;
```

```
 hops
------
 3
```


© 2026 SKAI Worldwide Co., Ltd.

`[*..6]` is a variable-length edge: up to six of them, of any label, in either direction. `p = ...` binds the whole path, and `length(p)` is its edge count. This is the query that has no reasonable SQL form.

Both endpoints of a `shortestpath` need a variable, even if you never use it. Leaving them anonymous gives `ERROR: node must be a reference to a specific node`.

Paths covers the rest — bounded ranges, `allshortestpaths`, and weighted search with `dijkstra`.

Write something

Writing looks like reading with a different verb. `CREATE` makes a vertex:

```
CREATE (p:person {name: 'Hugo Weaving', born: 1960}) RETURN p;
```

```
                        p
-------------------------------------------------
 person[4.63]{"born": 1960, "name": "Hugo Weaving"}
```

To connect it to something, find both ends first and then create the edge:

```
MATCH (p:person {name: 'Hugo Weaving'}), (m:movie {title: 'The Matrix'})
CREATE (p)-[r:acted_in {role: 'Agent Smith'}]->(m)
RETURN r;
```

```
                        r
-------------------------------------------------
 acted_in[5.60][4.63,3.1]{"role": "Agent Smith"}
```

An edge prints as `label[id][start,end]{properties}`, so you can see both endpoints in the result.

`SET` adds or replaces a property:

```
MATCH (p:person {name: 'Hugo Weaving'}) SET p.nationality = 'Australian' RETURN p;
```

```
                                      p
-------------------------------------------------------------------------------
 person[4.63]{"born": 1960, "name": "Hugo Weaving", "nationality": "Australian"}
```

`REMOVE` takes one away:

```
MATCH (p:person {name: 'Hugo Weaving'}) REMOVE p.nationality RETURN p;
```

```
                        p
-------------------------------------------------
 person[4.63]{"born": 1960, "name": "Hugo Weaving"}
```

And `DETACH DELETE` removes a vertex together with its edges. A plain `DELETE` on a vertex that still has edges is refused, because AgensGraph will not keep an edge with a missing endpoint:

```
MATCH (p:person {name: 'Hugo Weaving'}) DETACH DELETE p;
```

That puts the dataset back to 46 elements, so the rest of the manual's examples still match.

© 2026 SKAI Worldwide Co., Ltd.

Mix in SQL

The `rating` table you created with the dataset is an ordinary table, not part of the graph. Join it to a Cypher query by putting the Cypher where SQL expects a table:

```
SELECT q.film #>> '{}' AS film, r.score
FROM (MATCH (:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
      RETURN m.title AS film) q
JOIN rating r ON r.title = q.film #>> '{}'
WHERE r.source = 'critics'
ORDER BY r.score DESC;
```

```
     film     | score
--------------+-------
 Inception    |   8.8
 Interstellar |   8.7
```

`#>> '{}'` is what turns the `jsonb` title into plain `text` so it can be compared to a `text` column — and note the film names come back unquoted here, because this is a SQL projection rather than a Cypher one.

Where to go next

- **Querying** takes each of these ideas properly: reading, writing, pipelines, paths, indexes.
- **Reference** has a page per clause with the full syntax.
- **How a graph is stored** explains what the server actually did with everything you just created.
- **Vector search**, **full-text search** and **hybrid search** use the `plot` and `embedding` properties this page ignored.

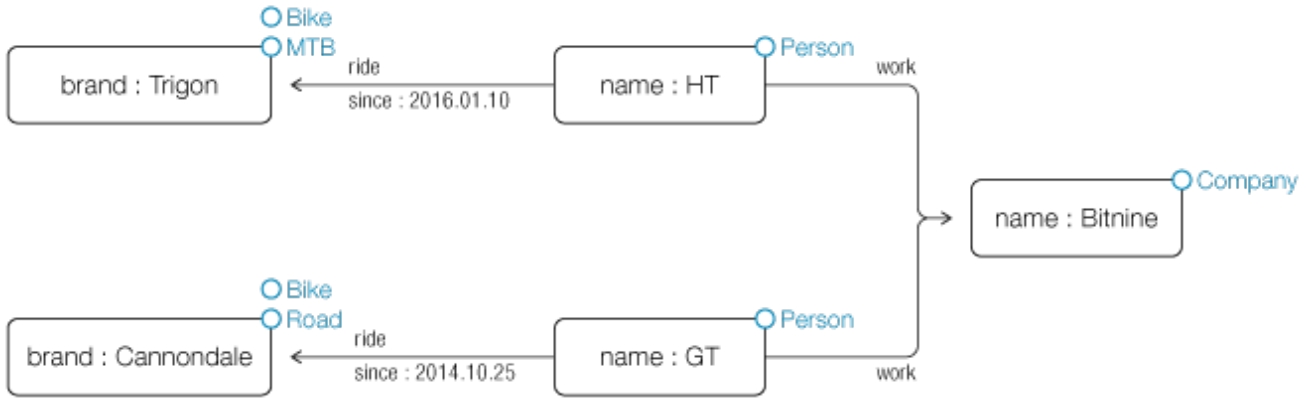
© 2026 SKAI Worldwide Co., Ltd.

Concepts

What a property graph is, how AgensGraph stores one in PostgreSQL, and how its three query languages relate to each other.

The property graph model

AgensGraph stores data in two shapes at once: as relational tables, and as a property graph. This page defines the graph half — the five things a graph is made of, and the rules each one obeys in AgensGraph specifically.



Vertices

A vertex is an entity: a person, a film, an account, a sensor. It has an identity, a label, and any number of properties.

A vertex belongs to **exactly one label**. That is the rule most likely to differ from a graph database you have used before — there is no set of labels per vertex. Where you would reach for several labels on one vertex, use **label inheritance** instead, which lets one pattern match a family of labels.

Edges

An edge connects two vertices. It has an identity, a label, any number of properties, and a direction: it always has a start vertex and an end vertex.

Three rules hold in AgensGraph:

- **An edge always has a label.** There is no unlabelled edge.
- **An edge always has both endpoints.** A dangling edge cannot exist, so deleting a vertex that still has edges is refused:

```
ERROR:  Vertex with connected edges cannot be removed
HINT:  Use DETACH DELETE or remove connected edges before deleting the vertex.
```

- **Direction is recorded, and matching it is optional.** You store an edge one way round and decide per query whether direction matters: `-[:acted_in]->` follows it, `-[:acted_in]-` ignores it. There is no reason to store the mirror image of an edge.

Properties

Properties are the data on a vertex or an edge. A property has a string name and a value.

© 2026 SKAI Worldwide Co., Ltd.

Properties are stored as a single `jsonb` document per element, so the value can be any JSON type: a string, a number, a boolean, an array, or a nested object. Two consequences:

- **A label imposes no column list.** Two vertices with the same label may carry entirely different property names. Nothing has to be declared in advance.
- **`null` is not a property value.** Assigning `null` removes the property:

```
MATCH (n:person {name: 'Tmp'}) SET n.born = NULL RETURN n;
person[4.40]{"name": "Tmp"}
```

So a property set to null and a property that was never there are the same thing. Turning on `allow_null_properties` changes that, and the null is stored:

```
person[4.40]{"born": null, "name": "Tmp"}
```

Because a read has to go into a JSON document, a property that every query filters on can be worth **promoting to a typed column**. That is a performance choice, not a modelling one — the property stays a property.

NOTE

`n.id` is not a vertex's identity. It reads a property named `id`, which usually does not exist. The identity is `id(n)`.

Labels

A label groups elements of the same kind. `movie`, `person`, `acted_in` are labels.

A label is what you attach an index to, what you grant permissions on, and what a pattern names to narrow a search. Vertex labels and edge labels are distinct kinds, created with `CREATE VLABEL` and `CREATE ELABEL`, and a name may be used once per graph.

Every graph is created with two labels you do not declare: `ag_vertex` and `ag_edge`. Every label you create inherits from one of them. That inheritance is what makes an unlabelled pattern like `MATCH (n)` mean "any vertex". A vertex created with no label at all lands in `ag_vertex`:

```
CREATE ({tmp: 1});
MATCH (n:ag_vertex ONLY) RETURN label(n) AS label, id(n) AS id, n.tmp;
```

```
    label    | id  | tmp
-------------+-----+------
 "ag_vertex" | 1.3 | 1
```

You cannot name the default label yourself:

```
CREATE (:ag_vertex {tmp: 1});
```

```
ERROR:  specifying default label is not allowed
```

Label names are folded to lower case unless you double-quote them, the same way PostgreSQL folds unquoted identifiers. A backtick is not a quoting character here — see **Syntax**.

© 2026 SKAI Worldwide Co., Ltd.

Paths

A path is a sequence of vertices joined by edges. It is a value in its own right: you can bind it to a variable, return it, and ask for its length or its members.

```
MATCH p = (a:person {name: 'Keanu Reeves'})-[:acted_in]->(m:movie)
RETURN length(p) AS hops, a.name AS actor, m.title AS film
ORDER BY film LIMIT 2;
```

```
 hops |     actor      |         film
------+----------------+----------------------
 1    | "Keanu Reeves" | "The Matrix"
 1    | "Keanu Reeves" | "The Matrix Reloaded"
```

A path's length is its number of edges, so a single vertex is a path of length 0 and an edge to itself is a path of length 1. Paths are how you express "any number of hops": see **Paths** for variable-length patterns, shortest paths and weighted search.

Graphs

A graph is the container for labels, and a database may hold several. A graph is also a PostgreSQL schema, so a graph name and a schema name cannot collide — see **How a graph is stored**.

A session works in **one** graph at a time, named by `graph_path`:

```
SET graph_path = moviekb;
```

Without it, nothing resolves:

```
ERROR:  graph_path is NULL
HINT:  Use SET graph_path
```

`graph_path` takes exactly one graph name, and a pattern cannot qualify a label with a graph name — `MATCH (n:other.work)` is a syntax error. To read two graphs in one statement, use SQL, which sees them as two schemas:

```
CREATE GRAPH other;
SET graph_path = other;
CREATE VLABEL work;
CREATE (:work {title: 'A'}), (:work {title: 'B'});
SET graph_path = moviekb;
```

```
SELECT (SELECT count(*) FROM moviekb.movie) AS movies,
       (SELECT count(*) FROM other.work)    AS works;
```

```
 movies | works
--------+-------
     18 |     2
```

```
DROP GRAPH other CASCADE;
```

Because `graph_path` is a per-session setting, set it for the database if every session wants the same graph:

© 2026 SKAI Worldwide Co., Ltd.

```
ALTER DATABASE mydb SET graph_path = moviekb;
```

How a graph is stored

AgensGraph does not bolt a graph engine onto PostgreSQL. It stores a graph as PostgreSQL objects, using the schemas, tables, indexes and inheritance that are already there. Once you know the mapping, most of what you know about running PostgreSQL applies unchanged. It also stops being surprising that one statement can mix Cypher with SQL: there is only ever one set of tables.

The mapping is three sentences long.

A graph is a schema. `CREATE GRAPH moviekb` creates a schema named `moviekb`.

A label is a table. `CREATE VLABEL movie` creates a table `moviekb.movie`.

A vertex or an edge is a row in that table.

What a label table looks like

A vertex label has two columns:

```
\d moviekb.movie
```

```
                                        Table "moviekb.movie"
   Column   |  Type   | Nullable |                                Default
------------+---------+----------
+-------------------------------------------------------------------------
 id         | graphid | not null | graphid(graph_labid('moviekb.movie'::cstring), nextval('mov
iekb.movie_id_seq'))
 properties | jsonb   | not null | jsonb_build_object()
Indexes:
    "movie_pkey" PRIMARY KEY, btree (id)
Inherits: moviekb.ag_vertex
```

An edge label has the same two columns plus the ids of its endpoints:

```
\d moviekb.acted_in
```

```
                                        Table "moviekb.acted_in"
   Column   |  Type   | Nullable |                                  Default
------------+---------+----------
+-------------------------------------------------------------------------
 id         | graphid | not null | graphid(graph_labid('moviekb.acted_in'::cstring), nextval('
moviekb.acted_in_id_seq'))
 start      | graphid | not null |
 end        | graphid | not null |
 properties | jsonb   | not null | jsonb_build_object()
Indexes:
    "acted_in_end_idx" btree ("end", start)
    "acted_in_id_idx" brin (id)
    "acted_in_start_idx" btree (start, "end")
Inherits: moviekb.ag_edge
```


© 2026 SKAI Worldwide Co., Ltd.

A label can carry more than those columns. **Promoting a property** adds a real typed column beside the bag, filled at write time. A hot property is then stored twice: once as a jsonb key, and once as a column the planner can index and compare natively. The catalog `ag_label_property` records which property key answers from which column. Cypher keeps naming the key, so the column name never appears in a query.

So an edge is a row that names two vertices, and traversing an edge is a join on `graphid`. The two btree indexes on `(start, "end")` and `("end", start)` are what make that join fast in either direction, and AgensGraph creates them for you when you create the label. The BRIN index on `id` is small and helps range scans over a label's ids.

Note which indexes each kind of label gets. A vertex label has a primary key on `id`; an edge label does not, because edges are reached through their endpoints rather than by id.

Because `properties` is a single `jsonb` column, a label imposes no column list on its rows. Two `movie` vertices may carry entirely different keys. That is what "schemaless properties" means here. It is also why reading a property means reading into a JSON document. **Promoting a property to a column** covers when that cost matters and what to do about it.

Identity: the graphid

`graphid` is a 64-bit identifier with two parts. The top 16 bits are the label id; the low 48 bits are the local id, a per-label sequence. It prints as `labid.locid`:

```
SELECT id, graphid_labid(id) AS labid, graphid_locid(id) AS locid
FROM moviekb.movie ORDER BY id LIMIT 3;
```

```
 id  | labid | locid
-----+-------+-------
 3.1 |     3 |     1
 3.2 |     3 |     2
 3.3 |     3 |     3
```

That layout has two consequences, and both matter.

First, an id carries its own label. Nothing has to be looked up to know which table a vertex came from.

Second, the label id is the high bits, so all the ids of one label form a single contiguous range of `graphid` values. A plain btree index on `id` can therefore answer "is this vertex a `movie`?" without touching the `movie` table at all.

The label ids themselves live in `ag_label`:

```
SELECT labname, labid, labkind FROM pg_catalog.ag_label
WHERE graphid = (SELECT oid FROM pg_catalog.ag_graph WHERE graphname = 'moviekb')
ORDER BY labid;
```

```
  labname   | labid | labkind
------------+-------+---------
 ag_vertex  |     1 | v
 ag_edge    |     2 | e
 movie      |     3 | v
 person     |     4 | v
 acted_in   |     5 | e
 directed   |     6 | e
 related_to |     7 | e
```

`labid` is 16 bits and 0 is not a valid label, so a database holds at most 65,535 labels. A single label holds at most $2^{48} - 1$ rows. See **Limits**.

© 2026 SKAI Worldwide Co., Ltd.

Every graph has two labels you did not create

`ag_vertex` and `ag_edge` are created with the graph, and every label you create inherits from one of them:

```
SELECT c.relname AS child, p.relname AS parent
FROM pg_inherits i
  JOIN pg_class c ON c.oid = i.inhrelid
  JOIN pg_class p ON p.oid = i.inhparent
  JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'moviekb' ORDER BY 1;
```

```
   child    |  parent
------------+-----------
 acted_in   | ag_edge
 directed   | ag_edge
 movie      | ag_vertex
 person     | ag_vertex
 related_to | ag_edge
```

That inheritance is what gives an unlabelled pattern its meaning. `MATCH (n)` does not scan a special index of everything; it scans `ag_vertex` and, through inheritance, every table below it:

```
EXPLAIN (COSTS OFF) MATCH (n) RETURN n;
```

```
              QUERY PLAN
---------------------------------------
 Result
   ->  Append
         ->  Seq Scan on ag_vertex n_1
         ->  Seq Scan on movie n_2
         ->  Seq Scan on person n_3
```

Naming a label narrows the scan to that table and its children, and the `ONLY` modifier narrows it to exactly that table. **Label inheritance** covers what you can build with that.

What a property index and a constraint become

A Cypher property index is a PostgreSQL expression index over the `properties` column:

```
CREATE PROPERTY INDEX ON movie (title);
```

```
\d moviekb.movie
```

```
Indexes:
    "movie_pkey" PRIMARY KEY, btree (id)
    "movie_title_idx" btree ((properties.'title'))
```

A uniqueness constraint becomes an exclusion constraint over the same expression:

```
CREATE CONSTRAINT movie_title_uq ON movie ASSERT title IS UNIQUE;
```


© 2026 SKAI Worldwide Co., Ltd.

```
Indexes:
    "movie_title_uq" EXCLUDE USING btree ((properties.'title') WITH =)
```

The `properties.'title'` form in those definitions is how the Cypher parser writes a property read. It is not syntax you type in SQL.

For tuning, the ordinary part is what matters. These are btree indexes over an expression. The planner picks them by the usual rules, and they only help a predicate it can match against that expression. **Indexes and constraints** covers which predicates qualify.

What this buys you

Everything PostgreSQL does to a table, it does to a label. That is not a slogan; it is the list of things you do not have to learn twice:

- **Transactions.** A Cypher write and a SQL write in one transaction commit or roll back together, because they are writes to tables in the same database.
- **Access control.** `GRANT SELECT ON moviekb.movie TO analyst` works, and so does row-level security on a label. See **Security**.
- **Backup and replication.** `pg_dump`, `pg_basebackup`, WAL archiving and streaming replication treat a graph as what it is — schemas and tables.
- **Extensions.** `pgvector` stores an embedding next to a graph because a property is `jsonb` in a table an extension can index. That is the whole basis of the **vector search** guide.
- **The planner.** `EXPLAIN` on a Cypher query shows sequential scans, index scans and joins, and you tune them the way you tune any PostgreSQL query.

And it is why SQL and Cypher compose in one statement rather than needing a bridge: **Combining SQL and Cypher**.

What is deliberately closed off

The mapping is real, but it is not an invitation to edit a graph as if it were an ordinary table. Two gates stand in the way, both off by default, because either operation can leave the graph catalog describing something that is no longer true.

Writing rows directly is refused:

```
INSERT INTO moviekb.movie (properties) VALUES ('{"title": "x"}');
```

```
ERROR:  DML query to graph objects is not allowed
```

So is reshaping a label:

```
ALTER TABLE moviekb.movie ADD COLUMN foo int;
```

```
ERROR:  cannot add a column to graph label "movie" with ALTER TABLE
HINT:  Use ALTER VLABEL or ALTER ELABEL instead, which keeps the graph catalog in step.
```

`enable_graph_dml` and `enable_graph_ddl` open them respectively, and **Server settings** says what you take on by doing so. Reading a label with SQL needs no setting at all — every `SELECT` on this page runs as it stands.

© 2026 SKAI Worldwide Co., Ltd.

Label inheritance

A label can inherit from another label. The child is a separate label with its own id sequence and its own rows, but a pattern that names the parent also matches the child's rows. That gives you a type hierarchy without giving up the ability to query the whole family in one pattern.

Inheritance is PostgreSQL table inheritance — **a label is a table**, and `INHERITS` on a label becomes `INHERITS` on that table.

Declaring a hierarchy

The examples on this page build their own small graph rather than using **the example dataset**, because inheritance is easier to see on a hierarchy than on films and people. Run `DROP GRAPH inh CASCADE;` when you are done with it.

```
CREATE GRAPH inh;
SET graph_path = inh;

CREATE VLABEL work;
CREATE VLABEL film INHERITS (work);
CREATE VLABEL series INHERITS (work);

CREATE (:film {title: 'A'}), (:series {title: 'B'}), (:work {title: 'C'});
```

Naming the parent matches every row in the family:

```
MATCH (n:work) RETURN label(n) AS label, n.title AS t ORDER BY t;
```

```
  label   |  t
----------+-----
 "film"   | "A"
 "series" | "B"
 "work"   | "C"
```

Naming a child matches only that child:

```
MATCH (n:film) RETURN label(n) AS label, n.title AS t;
```

```
 label  |  t
--------+-----
 "film" | "A"
```

Note what `label(n)` returns: the label the vertex was created with, never the parent. A vertex belongs to exactly one label. "Matching the parent" is about which tables the pattern scans, not about a vertex having two labels.

ONLY: excluding the children

`ONLY` restricts a pattern to the named label itself:

```
MATCH (n:work ONLY) RETURN label(n) AS label, n.title AS t;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 label  |  t
--------+-----
 "work" | "C"
```

It works on edges too, and is written inside the brackets:

```
MATCH ()-[e:rel ONLY]->() RETURN count(e) AS n;
```

```
 n
---
 0
```

That zero is the useful case: `rel` has a child, `sequel`, and all the edges were created as `sequel`, so `rel ONLY` matches none of them while plain `rel` matches all of them.

Edges inherit the same way

```
CREATE ELABEL rel;
CREATE ELABEL sequel INHERITS (rel);

MATCH (a:film {title: 'A'}), (b:series {title: 'B'}) CREATE (a)-[:sequel]->(b);
MATCH ()-[e:rel]->() RETURN label(e) AS label;
```

```
  label
----------
 "sequel"
```

More than one parent

A label may inherit from several labels at once:

```
CREATE VLABEL hybrid INHERITS (work, film);
```

The rows of `hybrid` are then matched by `MATCH (n:work)` and by `MATCH (n:film)`.

What every graph already inherits

Every label you create inherits from `ag_vertex` or `ag_edge`, even when you write no `INHERITS` clause. That is why an unlabelled `MATCH (n)` matches everything. See **How a graph is stored**.

What it costs

A pattern on a parent scans the parent's table and every table below it, so the plan grows an `Append` over the family:

```
EXPLAIN (COSTS OFF) MATCH (n:work) RETURN n;
```

```
            QUERY PLAN
-----------------------------------
 Result
   ->  Append
         ->  Seq Scan on work n_1
         ->  Seq Scan on film n_2
```


© 2026 SKAI Worldwide Co., Ltd.

```
         ->  Seq Scan on series n_3
         ->  Seq Scan on hybrid n_4
```

`ONLY` takes the `Append` away:

```
EXPLAIN (COSTS OFF) MATCH (n:work ONLY) RETURN n;
```

```
     QUERY PLAN
--------------------
 Seq Scan on work n
```

Two things follow. A deep hierarchy makes the cheapest possible query, "give me the rows of one label", plan over more relations than it needs. So name the child, or add `ONLY`, when you want one label's rows and no others. And an index has to exist on each table that will be scanned: an index on `work` does not index `film`'s rows. **Indexes and constraints** covers building an index across a hierarchy.

NOTE

Reach for inheritance when a query genuinely wants the whole family — a `work` that is sometimes a `film` and sometimes a `series`. If every query names the child anyway, separate labels are simpler and plan better.

Cypher, GQL and SQL

AgensGraph answers questions in three languages. They are not three products, three modes or three connections. There is one parser, one planner, one set of tables, and one wire protocol, and you choose per statement — or mix two in one statement.

Cypher is the graph language, and the most complete surface. Everything the graph can do is reachable from Cypher.

GQL is ISO/IEC 39075, the standard graph query language. In AgensGraph it is not a separate dialect. The GQL statements are accepted wherever a Cypher statement is accepted: `LET`, `NEXT`, `FILTER`, `FINISH`, `INSERT`, `FOR`, `CALL { }`, and the subquery value expressions `EXISTS { }` and `VALUE { }`. There is no switch to flip and no separate parser. **GQL conformance** records which parts of the standard are in.

SQL is PostgreSQL's, unchanged. Every table, view, function, extension, role and transaction you know from PostgreSQL is here, and a graph is visible to SQL as ordinary tables.

Choosing

Pick by the shape of the question, not by preference:

- **Following relationships** — how a person connects to a film, how far apart two films are, what is reachable in three hops — is what a graph pattern is for. A join chain in SQL says the same thing at several times the length, and cannot express "any number of hops" at all.
- **Aggregating, windowing, set operations, joining against ordinary tables, and anything administrative** is SQL's, and Cypher does not try to replace it.
- **GQL** where you want the standard's spelling, or where you are writing against more than one graph database and want the portable form. `INSERT` and `CREATE` do the same work; `FILTER` and `WHERE` do the same work.

© 2026 SKAI Worldwide Co., Ltd.

You do not have to choose once. **Combining SQL and Cypher** is a full page on the crossing points, and it is the most useful thing in this manual for a reader who already knows SQL.

If you know SQL

The vocabularies line up more closely than the syntax suggests, because **a graph is a schema and a label is a table**:

SQL	Cypher	Note
`CREATE SCHEMA s`	`CREATE GRAPH s`	The graph **is** the schema.
`CREATE TABLE t`	`CREATE VLABEL t`	A vertex label. `CREATE ELABEL` for edges.
`INSERT INTO t VALUES (…)`	`CREATE (:t {…})`	`INSERT (:t {…})` is the GQL spelling.
`SELECT … FROM t WHERE …`	`MATCH (n:t) WHERE … RETURN …`	`RETURN` is the projection, and it comes last.
`SELECT … FROM a JOIN b ON …`	`MATCH (a)-[:e]->(b)`	A pattern is a join along an edge.
`LEFT JOIN`	`OPTIONAL MATCH`	Unmatched rows come back with nulls.
`UPDATE t SET c = …`	`MATCH (n:t) SET n.c = …`	Find first, then write.
`DELETE FROM t WHERE …`	`MATCH (n:t) WHERE … DELETE n`	`DETACH DELETE` removes the edges too.
`WITH cte AS (…)`	`WITH …` / `NEXT`	Cypher's `WITH` divides a pipeline; it is not a CTE.
`ORDER BY` / `LIMIT` / `OFFSET`	`ORDER BY` / `LIMIT` / `SKIP`	`SKIP` and `OFFSET` are synonyms.
`CREATE INDEX ON t ((c))`	`CREATE PROPERTY INDEX ON t (c)`	Becomes an expression index.
`count(*)`, `sum`, `avg`	the same	Aggregates are PostgreSQL's.
`information_schema`, `pg_catalog`	`ag_graph`, `ag_label`	See **System catalogs**.

Three differences catch SQL users early, and all three are silent rather than loud:

A property read returns `jsonb`, not text. `RETURN m.title` prints `"The Matrix"` with the quotes, and casting inside the query does not change it. To get a plain scalar, unwrap it outside: `SELECT t #>> '{}' FROM (MATCH (m:movie) RETURN m.title AS t) q`.

`ORDER BY 2` does not mean "the second column". In Cypher the `2` is a constant, so every row compares equal and nothing sorts. Order by the alias or the expression.

`n.id` is not the id. `id(n)` returns the `graphid`; `n.id` looks for a property called `id` and returns nothing if there isn't one.

© 2026 SKAI Worldwide Co., Ltd.

One statement, two languages

A Cypher query can appear where SQL expects a table:

```
SELECT q.title #>> '{}' AS film, r.score
FROM (MATCH (:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
      RETURN m.title AS title) q
JOIN rating r ON r.title = q.title #>> '{}'
WHERE r.source = 'critics'
ORDER BY r.score DESC;
```

```
     film     | score
--------------+-------
 Inception    |   8.8
 Interstellar |   8.7
```

And SQL can appear where Cypher expects a value:

```
MATCH (m:movie)
WHERE m.title = (SELECT to_jsonb(title) FROM rating
                 WHERE source = 'critics' ORDER BY score DESC LIMIT 1)
RETURN m.title AS best;
```

```
    best
-------------
 "Inception"
```

Both of those run against **the example dataset**.

What “GQL support” means in this release

GQL arrived in AgensGraph as added statements on the existing grammar rather than as a second language, which has two consequences worth stating plainly.

A GQL statement can be mixed with Cypher in one query — `MATCH … LET … RETURN` is a valid statement, and so is `MATCH … FILTER … NEXT RETURN …`. Nothing marks a query as “GQL”.

And conformance is partial, by feature rather than by mode. **GQL conformance** lists it feature by feature against the subclauses of ISO/IEC 39075:2024, and each clause page in the **reference** repeats the same statement for that clause under its own Compatibility heading.

© 2026 SKAI Worldwide Co., Ltd.

Querying

How to ask questions of a graph. These pages teach the language in the order you meet it; the **Reference** gives the formal definition of every clause and function.

Reading a graph

Every read starts with a pattern. A pattern is a drawing of the shape you are looking for, written in ASCII: parentheses for vertices, brackets and dashes for edges. `MATCH` hands that drawing to the planner, which turns it into scans and joins, and returns one row for every way the drawing fits the data.

This page is about the drawing and about what the planner does with it. The formal syntax is in **`MATCH`** and **`WHERE`**; everything here runs against **the example dataset**.

One vertex

The smallest pattern is a pair of parentheses. Add a label to narrow it to one table:

```
MATCH (m:movie) RETURN m.title AS title ORDER BY title LIMIT 3;
```

```
       title
--------------------
 "Avatar"
 "Blade Runner"
 "Blade Runner 2049"
```

`m` is a variable bound to each matching vertex; `movie` is the label. Either may be left out. `(m)` matches every vertex, `(:movie)` matches every film without naming it, and `()` matches every vertex without naming it.

Naming a label includes the labels that inherit from it. `ONLY` excludes them:

```
MATCH (m:movie ONLY) RETURN count(m) AS films;
```

```
 films
-------
 18
```

Nothing inherits from `movie` here, so the count is the same either way. On a hierarchy the difference is the whole `Append`. See **Label inheritance**.

Two ways to narrow

A property map inside the pattern tests properties for equality:

```
MATCH (m:movie {genre: 'Adventure'}) RETURN m.title AS title, m.year AS year;
```


© 2026 SKAI Worldwide Co., Ltd.

```
      title      | year
-----------------+------
 "Jurassic Park" | 1993
```

`WHERE` takes any predicate, so anything that is not equality goes there:

```
MATCH (m:movie) WHERE m.year >= 2017 RETURN m.title AS title, m.year AS year ORDER BY year;
```

```
           title            | year
----------------------------+------
 "Blade Runner 2049"        | 2017
 "The Matrix Resurrections" | 2021
```

The two combine, and for a plain `MATCH` they mean exactly the same thing. The planner receives one conjunction either way:

```
MATCH (m:movie {genre: 'Action'}) WHERE m.year < 2000
RETURN m.title AS title, m.year AS year ORDER BY year;
```

```
            title             | year
------------------------------+------
 "The Terminator"             | 1984
 "Terminator 2: Judgment Day" | 1991
 "The Matrix"                 | 1999
```

Use the map for the keys that identify the thing you want and `WHERE` for the conditions on it. The one place the choice matters is `OPTIONAL MATCH`, below.

A condition has to be a boolean. `WHERE m.title` does not mean “has a title”; it raises `ERROR: cannot cast "The Matrix" (jsonb type string) to boolean`. Write the test you mean: `WHERE m.title IS NOT NULL`.

An edge

An edge is a pair of brackets between two dashes, with an arrowhead for direction:

```
MATCH (p:person)-[]->(m:movie {title: 'Blade Runner 2049'})
RETURN p.name AS person ORDER BY person;
```

```
       person
--------------------
 "Denis Villeneuve"
 "Harrison Ford"
 "Ryan Gosling"
```

Note the empty brackets. The brackets are not optional: `-->` and `--` are not patterns, because `--` begins a SQL line comment. `MATCH (p:person)-->(m:movie) RETURN count(p)` comments away everything from the first dash onwards and fails with `ERROR: Cypher query must end with RETURN, FINISH or update clause`. Write `-[]->` and `-[]-`.

Bind the edge to a variable and its label and properties become available:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person)-[r]->(m:movie {title: 'Blade Runner 2049'})
RETURN p.name AS person, label(r) AS via, r.role AS role ORDER BY person;
```

```
      person       |    via     |      role
-------------------+------------+----------------
 "Denis Villeneuve" | "directed" |
 "Harrison Ford"    | "acted_in" | "Rick Deckard"
 "Ryan Gosling"     | "acted_in" | "K"
```

`label(r)` is the edge's label and `r.role` is a property of the edge itself. `directed` edges carry no `role`, so that cell is null.

Direction

`-[]->` follows the edge as stored, `<-[]-` follows it backwards, and `-[]-` ignores direction. Direction is a property of the query, not of the data: you store an edge one way round and decide per pattern whether it matters.

```
MATCH (a:movie {title: 'The Matrix Reloaded'})-[:related_to]->(b:movie)
RETURN b.title AS neighbour ORDER BY neighbour;
```

```
       neighbour
-------------------------
 "The Matrix Revolutions"
```

```
MATCH (a:movie {title: 'The Matrix Reloaded'})-[:related_to]-(b:movie)
RETURN b.title AS neighbour ORDER BY neighbour;
```

```
       neighbour
-------------------------
 "The Matrix"
 "The Matrix Revolutions"
```

One label per edge

An edge pattern names at most one label. There is no alternation:

```
MATCH (p:person)-[r:acted_in|directed]->(m:movie) RETURN count(r);
```

```
ERROR:  multiple types for relationship not supported
```

This holds at every length and inside `shortestpath`. Leave the edge untyped and test the label with a predicate:

```
MATCH (p:person)-[r]->(m:movie {title: 'Blade Runner'})
WHERE label(r) IN ['acted_in', 'directed']
RETURN p.name AS person, label(r) AS via ORDER BY person;
```

```
    person      |    via
----------------+------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Harrison Ford" | "acted_in"
 "Ridley Scott"  | "directed"
```

That form costs something: an untyped edge scans every edge label in the graph, and the label test is applied afterwards. Two patterns joined with `UNION` keep each scan narrow when the alternation is over few labels and the graph is large.

A whole path

`p =` before a pattern binds the path itself, not just its ends:

```
MATCH p = (a:person {name: 'Ryan Gosling'})-[:acted_in]->(m:movie)
RETURN length(p) AS hops,
       [x IN nodes(p) | label(x)] AS labels,
       [r IN relationships(p) | label(r)] AS edges;
```

```
 hops |       labels        |    edges
------+---------------------+--------------
 1    | ["person", "movie"] | ["acted_in"]
```

`length(p)` is the edge count, `nodes(p)` the vertices in order and `relationships(p)` the edges. A path is a value: you can return it, order by its length, or collect it. **Paths** does the rest, including patterns whose length you do not know.

More than one pattern

`MATCH` takes a comma-separated list of patterns. A variable that appears in two of them joins them:

```
MATCH (d:person)-[:directed]->(m:movie), (a:person)-[:acted_in]->(m)
RETURN d.name AS director, a.name AS actor, m.title AS film
ORDER BY film, actor LIMIT 5;
```

```
      director      |       actor       |        film
--------------------+-------------------+--------------------
 "James Cameron"    | "Sam Worthington" | "Avatar"
 "Ridley Scott"     | "Harrison Ford"   | "Blade Runner"
 "Denis Villeneuve" | "Harrison Ford"   | "Blade Runner 2049"
 "Denis Villeneuve" | "Ryan Gosling"    | "Blade Runner 2049"
 "Neill Blomkamp"   | "Matt Damon"      | "Elysium"
```

`m` is bound by the first pattern and reused by the second. That makes this an inner join on the film rather than a product. Two patterns with no variable in common are a product, and that is legal and occasionally what you want:

```
MATCH (p:person {name: 'Keanu Reeves'}), (m:movie {genre: 'Adventure'})
RETURN p.name AS person, m.title AS film;
```

```
     person     |      film
----------------+-----------------
 "Keanu Reeves" | "Jurassic Park"
```

Writing the two patterns as two consecutive `MATCH` clauses means the same thing. The comma form and the two-clause form both produce 31 rows for the director-and-actor query above, and the same plan shape. Use whichever reads better: a comma for "these things together", a second `MATCH` for "and then".

© 2026 SKAI Worldwide Co., Ltd.

A pattern that may not fit

`MATCH` drops a row whose pattern finds nothing:

```
MATCH (p:person {name: 'Ridley Scott'})-[:acted_in]->(m:movie)
RETURN p.name AS person, m.title AS film;
```

```
 person | film
--------+------
(0 rows)
```

`OPTIONAL MATCH` keeps the row and leaves the pattern's variables null. It is a left join, and it behaves like one in every respect:

```
MATCH (p:person {name: 'Ridley Scott'})
OPTIONAL MATCH (p)-[:acted_in]->(m:movie)
RETURN p.name AS person, m.title AS film;
```

```
     person     | film
----------------+------
 "Ridley Scott" |
```

This is where the property map and `WHERE` stop agreeing. A `WHERE` written on an `OPTIONAL MATCH` belongs to the join condition, so a row that fails it survives with nulls:

```
MATCH (m:movie {title: 'The Matrix'})
OPTIONAL MATCH (m)<-[r:acted_in]-(p:person) WHERE r.role = 'Agent Smith'
RETURN m.title AS film, p.name AS actor;
```

```
     film     | actor
--------------+-------
 "The Matrix" |
```

The same predicate one clause later is a filter on the joined result, and drops the row:

```
MATCH (m:movie {title: 'The Matrix'})
OPTIONAL MATCH (m)<-[r:acted_in]-(p:person)
WITH m, p, r WHERE r.role = 'Agent Smith'
RETURN m.title AS film, p.name AS actor;
```

```
 film | actor
------+-------
(0 rows)
```

If you know SQL, you have met this before: it is the difference between a condition in `ON` and the same condition in `WHERE` after a `LEFT JOIN`. See **`OPTIONAL MATCH`**.

Asking whether a pattern exists

A pattern is not a predicate on its own. `WHERE (p)-[:acted_in]->(:movie)` is a syntax error. Put it in an `EXISTS { }` subquery:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person)
WHERE EXISTS { MATCH (p)-[:acted_in]->(:movie {title: 'The Matrix'}) }
RETURN p.name AS actor ORDER BY actor;
```

```
        actor
----------------------
 "Carrie-Anne Moss"
 "Keanu Reeves"
 "Laurence Fishburne"
```

`EXISTS`, `COUNT`, `COLLECT`, `ARRAY`, `VALUE` and `IN` all take a subquery like this. **Composing a query** covers them together.

How a pattern becomes a plan

A pattern is not executed as a walk. It is compiled into ordinary relational algebra over the label tables, and `EXPLAIN` shows exactly that:

```
EXPLAIN (COSTS OFF) MATCH (p:person)-[r:acted_in]->(m:movie) RETURN p.name, m.title;
```

```
                                                                               QUERY PLAN
----------------------------------------------------------------------------------------------
---------------------------------------------------------------------------
 Hash Join
   Hash Cond: (r.start = p.id)
   ->  Hash Join
         Hash Cond: (r."end" = m.id)
         ->  Seq Scan on acted_in r
               Filter: ((start >= '4.0'::graphid) AND (start <= '4.281474976710655'::graphid)
AND ("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710655'::graphid))
         ->  Hash
               ->  Seq Scan on movie m
   ->  Hash
         ->  Seq Scan on person p
```

Three things are worth reading out of that.

Each pattern element is a scan. `person`, `acted_in` and `movie` are three tables, and the pattern is two joins between them: `r.start = p.id` and `r."end" = m.id`. Traversing an edge **is** a join on `graphid`. The `(start, "end")` and `("end", start)` indexes that come with every edge label therefore matter a great deal.

The label test on an endpoint is answered on the edge. `p` is a `person`, so `r.start` can only be a `person`'s id. A label id lives in the high bits of a `graphid`, so all of one label's ids form a contiguous range.

The planner turns "`start` is a person" into `start >= '4.0' AND start <= '4.281474976710655'`, a range test it can push into the edge scan or an index on it, with no visit to `person` at all. See **Identity: the graphid**.

Anchoring the pattern changes the join order. Pin one end to a single vertex and the plan stops hashing whole tables:

```
EXPLAIN (COSTS OFF)
MATCH (p:person {name: 'Keanu Reeves'})-[r:acted_in]->(m:movie) RETURN m.title;
```

```


QUERY PLAN

----------------------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------------------------
----------
 Nested Loop
   ->  Nested Loop
         ->  Seq Scan on person p
               Filter: (properties.'name' = '"Keanu Reeves"'::jsonb)
         ->  Bitmap Heap Scan on acted_in r
               Recheck Cond: ((p.id = start) AND (start >= '4.0'::graphid) AND (start <= '4.28
1474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710655'::grap
hid))
               ->  Bitmap Index Scan on acted_in_start_idx
                     Index Cond: ((start = p.id) AND (start >= '4.0'::graphid) AND (start <= '
4.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <=
'3.281474976710655'::graphid))
   ->  Index Scan using movie_pkey on movie m
         Index Cond: (id = r."end")
```

Now `acted_in` is reached through `acted_in_start_idx` for the one person, and `movie` through its primary key for each edge found.

This is the shape you want for a traversal from a known starting point. It is also the reason a selective anchor, a property with an index on it, is usually worth more than any other tuning. **Indexes and constraints** covers which predicates can serve as one.

What an endpoint you do not use costs

An endpoint whose value the query never reads does not have to be fetched at all. Here `m` is bound and labelled, and nothing in the projection asks for it:

```
EXPLAIN (COSTS OFF) MATCH (p:person)-[r:acted_in]->(m:movie) RETURN p.name;
```

```
                                                                                  QUERY PLAN
------------------------------------------------------------------------------------------------
------------------------------------------------------------------------
 Hash Join
   Hash Cond: (r.start = p.id)
   ->  Seq Scan on acted_in r
         Filter: (("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710655'::graphid) AND (s
tart >= '4.0'::graphid) AND (start <= '4.281474976710655'::graphid))
   ->  Hash
         ->  Seq Scan on person p
```

There is no scan for `m`. Its only remaining contribution is the second half of that filter: `"end"` must be in the `movie` range. So the whole of `(m:movie)` has collapsed into two comparisons on a column the edge scan was reading anyway.

Switch the elision off and the vertex the query never asked about is fetched across the whole `ag_vertex` family:

```
SET enable_graph_endpoint_elision = off;
EXPLAIN (COSTS OFF) MATCH (p:person)-[r:acted_in]->(m) RETURN p.name;
```

```
                                        QUERY PLAN
------------------------------------------------------------------------------------------------
-
 Hash Join
   Hash Cond: (r."end" = m.id)
   ->  Hash Join
         Hash Cond: (r.start = p.id)
```


© 2026 SKAI Worldwide Co., Ltd.

```
        ->  Seq Scan on acted_in r
              Filter: ((start >= '4.0'::graphid) AND (start <= '4.281474976710655'::graphid))
        ->  Hash
              ->  Seq Scan on person p
  ->  Hash
        ->  Append
              ->  Seq Scan on ag_vertex m_1
              ->  Seq Scan on movie m_2
              ->  Seq Scan on person m_3
```

The setting is on by default and there is no reason to turn it off outside an experiment like this one.

The comparison shows the size of the saving, and where it stops. Add `m.title` to that `RETURN` and the `Append` comes back, because the query really does need the film. Project the endpoints you use and no more.

NOTE

Name every element of a pattern whose plan you intend to keep. An element you leave unnamed is given a synthetic alias like `"<0000000002>"`, and the counter behind it advances on every statement — so the same query prints a different plan the second time you run it.

NOTE

Anonymity is not free everywhere. Both endpoints of a `shortestpath` must be named variables, even ones you never use — an anonymous endpoint there gives `ERROR: node must be a reference to a specific node`.

Where to go next

- **Composing a query** — `WITH`, aggregation, subqueries, and the rest of the pipeline that follows a `MATCH`.
- **Paths** — variable-length edges, shortest paths and weighted search.
- **Writing a graph** — the same patterns with a write clause after them.
- **Indexes and constraints** — which predicates an index can serve.

Writing a graph

A write clause has no pattern matching of its own worth the name. It takes rows from whatever came before it and does something to each one. So every write beyond the very first `CREATE` follows the same shape: find first, then write.

That is the whole plan of this page. It walks the write clauses in the order you need them: `CREATE`, `MERGE`, `SET`, `REMOVE`, `DELETE`. Then it finishes with what a transaction does to a graph. The formal syntax for each is in the reference: **`CREATE`**, **`MERGE`**, **`SET`**, **`REMOVE`**, **`DELETE`**.

The examples run against **the example dataset** and put it back as they found it, so you can work down the page and still match the counts other pages print.

Making something new

`CREATE` with a pattern that binds nothing existing makes a vertex:

```
CREATE (p:person {name: 'Gloria Foster'}) RETURN p;
```


© 2026 SKAI Worldwide Co., Ltd.

```
                    p
------------------------------------------
 person[4.153]{"name": "Gloria Foster"}
```

The `RETURN` is optional and the label is not: a vertex created with no label lands in `ag_vertex`, and an edge must always have one. The id in that result is whatever the label's sequence was up to, so your own will differ.

To connect two things that already exist, find them and then create the edge. There is no way to write an edge to a vertex you have not matched:

```
MATCH (p:person {name: 'Gloria Foster'}), (m:movie {title: 'The Matrix'})
CREATE (p)-[r:acted_in {role: 'The Oracle'}]->(m)
RETURN r;
```

```
                        r
-------------------------------------------------
 acted_in[5.71][4.153,3.1]{"role": "The Oracle"}
```

An edge prints as `label[id][start,end]{properties}`. `(p)` and `(m)` here carry no label and no property map. They are references to variables already bound, and adding a label to one would make it a new vertex instead.

A `CREATE` pattern longer than one element makes everything in it:

```
CREATE (d:person {name: 'Marc Caro'})-[:directed]->(f:movie {title: 'The City of Lost
Children', year: 1995})
RETURN d.name AS director, f.title AS film;
```

```
  director   |            film
-------------+-----------------------------
 "Marc Caro" | "The City of Lost Children"
```

`INSERT` is the ISO/IEC 39075 spelling of the same clause and takes the same patterns: `INSERT (:person {name: 'Ada Lovelace'})`. See **`INSERT`**.

Making something only if it is not there

`MERGE` matches its pattern, and creates it only when the match finds nothing:

```
MERGE (p:person {name: 'Gloria Foster'}) RETURN p;
```

```
                    p
------------------------------------------
 person[4.153]{"name": "Gloria Foster"}
```

That found the vertex from the previous section rather than making a second one. Be careful about what it matches on: the entire pattern, including every property in the map. Add one property and the pattern no longer describes the vertex you have, so `MERGE` makes one that it does describe:

```
MERGE (p:person {name: 'Gloria Foster', born: 1933}) RETURN p;
```


© 2026 SKAI Worldwide Co., Ltd.

```
                          p
------------------------------------------------------
 person[4.155]{"born": 1933, "name": "Gloria Foster"}
```

There are now two:

```
MATCH (p:person {name: 'Gloria Foster'}) RETURN id(p) AS id, p.born AS born ORDER BY id;
```

```
  id   | born
-------+------
 4.153 |
 4.155 | 1933
```

So `MERGE` on a map of everything you know about a thing is a duplicate generator. `MERGE` on the keys that identify it, with the rest of the values applied afterwards, is idempotent. `ON CREATE SET` and `ON MATCH SET` are there for that:

```
MERGE (p:person {name: 'Joe Pantoliano'})
  ON CREATE SET p.born = 1951, p.seen = 1
  ON MATCH SET p.seen = p.seen + 1
RETURN p;
```

```
                                  p
-------------------------------------------------------------------
 person[4.156]{"born": 1951, "name": "Joe Pantoliano", "seen": 1}
```

Run it again and only the `ON MATCH` branch fires:

```
                                  p
-------------------------------------------------------------------
 person[4.156]{"born": 1951, "name": "Joe Pantoliano", "seen": 2}
```

`MERGE` works on edges the same way, and matching an edge pattern means matching its label, its direction, its property map and both endpoints. Bind the endpoints with `MATCH` first so `MERGE` decides about the edge alone. This one is already there, so nothing is created:

```
MATCH (p:person {name: 'Christopher Nolan'}), (m:movie {title: 'Inception'})
MERGE (p)-[r:directed]->(m) RETURN r;
```

```
             r
--------------------------
 directed[6.8][4.8,3.5]{}
```

Back to one Gloria Foster before going on:

```
MATCH (p:person {name: 'Gloria Foster', born: 1933}) DELETE p;
```

A uniqueness constraint on the identifying property is the durable version of this discipline, because it refuses the duplicate instead of relying on you to write the right `MERGE`. See **Indexes and constraints**.

© 2026 SKAI Worldwide Co., Ltd.

Changing properties

`SET` on one property adds it or replaces it:

```
MATCH (p:person {name: 'Gloria Foster'}) SET p.born = 1933 RETURN p;
```

```
                          p
------------------------------------------------------
 person[4.153]{"born": 1933, "name": "Gloria Foster"}
```

`SET p += { ... }` merges a map in, adding keys and overwriting the ones that collide:

```
MATCH (p:person {name: 'Gloria Foster'}) SET p += {died: 2001, city: 'Chicago'} RETURN p;
```

```
                                          p
-------------------------------------------------------------------------------------
 person[4.153]{"born": 1933, "city": "Chicago", "died": 2001, "name": "Gloria Foster"}
```

`SET p = { ... }` replaces the whole property map. Everything not in the new map is gone:

```
MATCH (p:person {name: 'Gloria Foster'}) SET p = {name: 'Gloria Foster', born: 1933} RETURN p;
```

```
                          p
------------------------------------------------------
 person[4.153]{"born": 1933, "name": "Gloria Foster"}
```

The difference between `+=` and `=` is the difference between `UPDATE ... SET one_column` and replacing the row, and it is easy to type the wrong one.

Assigning null does not store a null. It removes the property:

```
MATCH (p:person {name: 'Gloria Foster'}) SET p.born = NULL RETURN p;
```

```
                    p
----------------------------------------
 person[4.153]{"name": "Gloria Foster"}
```

So a property set to null and a property that never existed are the same state, and there is no way to tell them apart afterwards. `allow_null_properties` changes that and stores the null; it is off by default. `REMOVE` is the explicit spelling of the same thing:

```
MATCH (p:person {name: 'Joe Pantoliano'}) REMOVE p.seen RETURN p;
```

```
                           p
-----------------------------------------------------
 person[4.156]{"born": 1951, "name": "Joe Pantoliano"}
```

An edge's properties are written exactly like a vertex's:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (:person {name: 'Gloria Foster'})-[r:acted_in]->(:movie {title: 'The Matrix'})
SET r.credited = true RETURN r;
```

```
                                r
-------------------------------------------------------------------
 acted_in[5.71][4.153,3.1]{"role": "The Oracle", "credited": true}
```

NOTE

A promoted property is the one exception to "a property can hold anything". Once a key has a typed column behind it, a write whose JSON type does not fit that column aborts. See **Promoting a property to a column**.

Removing things

`DELETE` removes what you matched and hands it back, so you can see what went:

```
MATCH (p:person {name: 'Joe Pantoliano'}) DELETE p RETURN p;
```

```
                          p
-----------------------------------------------------
 person[4.156]{"born": 1951, "name": "Joe Pantoliano"}
```

A vertex that still has edges cannot be deleted, because an edge with a missing endpoint cannot exist:

```
MATCH (p:person {name: 'Gloria Foster'}) DELETE p;
```

```
ERROR:  Vertex with connected edges cannot be removed
HINT:  Use DETACH DELETE or remove connected edges before deleting the vertex.
```

Either remove the edges yourself:

```
MATCH (:person {name: 'Gloria Foster'})-[r:acted_in]->() DELETE r RETURN r;
```

```
                                r
-------------------------------------------------------------------
 acted_in[5.71][4.153,3.1]{"role": "The Oracle", "credited": true}
```

or say `DETACH DELETE`, which removes the vertex together with every edge touching it:

```
MATCH (d:person {name: 'Marc Caro'}) DETACH DELETE d;
MATCH (f:movie {title: 'The City of Lost Children'}) DETACH DELETE f;
MATCH (p:person {name: 'Gloria Foster'}) DELETE p;
```

The dataset is now back to what it was:

```
MATCH (n) RETURN count(n) AS elements;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 elements
----------
 46
```

The refusal is a guard, not an inconvenience. `DETACH DELETE` on a hub vertex can remove far more than you meant, so match the edges you intend to lose and count them before you delete anything.

Reading what you just wrote

A write clause is not the end of a statement. Its output rows carry on down the pipeline, and a read after it sees the graph as the write left it:

```
CREATE (:person {name: 'Gloria Foster'})
WITH 1 AS done
MATCH (p:person {name: 'Gloria Foster'})
RETURN count(p) AS found;
```

```
 found
-------
 1
```

The planner arranges that by materialising the write's effects before the next read runs. That is an "eager" plan, and `EXPLAIN` labels it as such. **Tuning a query** covers what that costs and the setting that forbids it.

Transactions

A graph is tables in a database, so a write to a graph is an ordinary transactional write. There is nothing to enable and no second commit to think about: a Cypher write and a SQL write in one transaction commit together or roll back together.

```
CREATE TABLE import_log (what text, at timestamptz DEFAULT now());
```

```
BEGIN;
CREATE (:person {name: 'Gloria Foster'});
INSERT INTO import_log (what) VALUES ('added Gloria Foster');
MATCH (p:person {name: 'Gloria Foster'}) RETURN count(p) AS found;
```

```
 found
-------
 1
```

Inside the transaction both writes are there. Roll back and neither is:

```
ROLLBACK;
MATCH (p:person {name: 'Gloria Foster'}) RETURN count(p) AS found;
```

```
 found
-------
 0
```


© 2026 SKAI Worldwide Co., Ltd.

```
SELECT count(*) AS logged FROM import_log;
```

```
 logged
--------
      0
```

```
DROP TABLE import_log;
```

Everything else that follows from being PostgreSQL follows here too: isolation levels, savepoints, `SELECT ... FOR UPDATE` on a label table, deadlock detection between two sessions writing the same vertex. A bulk load is one transaction if you want it to be, and `CREATE GRAPH` inside a transaction rolls back like any other DDL.

Two gates exist in the other direction. Writing rows into a label with SQL `INSERT`, `UPDATE`, `DELETE` or `MERGE` is refused unless `enable_graph_dml` is on. Reshaping a label with `ALTER TABLE` is refused unless `enable_graph_ddl` is on. Either can leave the graph catalog describing something that is no longer true. See **How a graph is stored**.

Where to go next

- **Loading data** — bulk loading, including `LOAD FROM` over an existing table.
- **Composing a query** — `WITH`, `UNWIND` and aggregation, all of which are useful in front of a write clause.
- **Security** — grants and row-level security on a label.

Composing a query

A Cypher or GQL statement is a pipeline. Each clause takes a stream of rows, does one thing to it, and hands the result to the next clause. `MATCH` produces rows, `WHERE` removes some, `WITH` reshapes them, `RETURN` ends the statement.

That is a different mental model from SQL, where a `SELECT` is a single block with named parts evaluated in a fixed order. Here the order is the order you wrote. A query can match, aggregate, filter the aggregate, match again from the survivors, and aggregate again, with no subquery, no CTE and no `HAVING`.

Everything runs against **the example dataset**.

WITH is the divider

`WITH` ends one part of a pipeline and begins the next. It takes a projection list exactly like `RETURN`, and only what it lists stays in scope:

```
MATCH (d:person)-[:directed]->(m:movie)
WITH d, count(m) AS films
WHERE films > 2
RETURN d.name AS director, films ORDER BY films DESC, director;
```

```
     director      | films
-------------------+-------
 "Lana Wachowski"  | 4
 "James Cameron"   | 3
 "Lilly Wachowski" | 3
```


© 2026 SKAI Worldwide Co., Ltd.

`m` is gone after the `WITH`; `d` and `films` are what is left. The `WHERE` after it filters the aggregate, the job SQL needs `HAVING` for.

Order inside a `WITH` is fixed and worth committing to memory: `WHERE` first, then `ORDER BY`, then `SKIP` and `LIMIT`. So this filters to five films and then takes two of them, rather than taking two films and then filtering:

```
MATCH (m:movie) WITH m.year AS y WHERE y > 2010 ORDER BY y LIMIT 2 RETURN y;
```

```
  y
------
 2012
 2013
```

A name may be passed on only once:

```
MATCH (m:movie) WITH m.year AS y, m.genre AS y RETURN y;
```

```
ERROR:  WITH passes on "y" more than once
```

How it differs from a CTE

Both `WITH` keywords divide a query, and they are not the same keyword.

A SQL CTE is named and referenced: you declare `WITH films AS (...)` and then mention `films` wherever you want it, as many times as you like, in any order.

A Cypher `WITH` is positional: it has no name, it is referenced by being next, and it is consumed exactly once. There is no way to fork a Cypher pipeline into two branches and rejoin them.

They do not mix. Cypher's `WITH` cannot introduce a named subquery: `WITH d AS (MATCH ...)` is a syntax error. But a SQL CTE can hold a Cypher query, which is the way to get the CTE behaviour when you want it:

```
WITH films AS (MATCH (m:movie) WHERE m.year > 2010 RETURN m.title AS title, m.year AS year)
SELECT f.title #>> '{}' AS title, r.score
FROM films f JOIN rating r ON r.title = f.title #>> '{}'
WHERE r.source = 'critics' ORDER BY r.score DESC;
```

```
          title           | score
--------------------------+-------
 Interstellar             |   8.7
 Blade Runner 2049        |   8.0
 The Avengers             |   8.0
 Elysium                  |   6.6
 The Matrix Resurrections |   5.7
```

The outer statement there is SQL, because a statement that begins with a SQL CTE is a SQL statement. **Combining SQL and Cypher** is the page for that boundary.

That is also why a Cypher pipeline cannot begin with `WITH`. A leading `WITH` is read as a CTE, so the parser is looking for a name and a parenthesised query:

```
WITH 1 AS x RETURN x;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  syntax error at or near "1"
LINE 1: WITH 1 AS x RETURN x;
            ^
```

`LET` has no such collision, so it is the clause to open a statement with a computed value:

```
LET x = 1 RETURN x;
```

```
 x
---
 1
```

Aggregation

Aggregates are PostgreSQL's aggregates, and grouping is implicit: anything in the projection list that is not inside an aggregate becomes a grouping key.

```
MATCH (m:movie) RETURN m.genre AS genre, count(*) AS films, min(m.year) AS earliest ORDER BY
genre;
```

```
    genre     | films | earliest
--------------+-------+----------
 "Action"     | 7     | 1984
 "Adventure"  | 1     | 1993
 "Sci-Fi"     | 10    | 1982
```

`count(*)` counts rows; `count(x)` counts rows where `x` is not null. That makes `count(x)` useful after an `OPTIONAL MATCH`. `count(DISTINCT x)` counts distinct values, and the gap between the two is usually the fan-out of your pattern:

```
MATCH (:person)-[:acted_in]->(m:movie) RETURN count(m) AS rows, count(DISTINCT m) AS films;
```

```
 rows | films
------+-------
 25   | 18
```

`collect()` is the aggregate you will reach for most, because it turns a fan-out back into one row. Over a vertex, an edge or a path it builds a native array of those elements, so the collected elements stay usable:

```
MATCH (p:person)-[:acted_in]->(m:movie {title: 'The Matrix'})
WITH collect(p) AS players
RETURN pg_typeof(players) AS type, cardinality(players) AS n, players[1].name AS first;
```

```
    type     | n |        first
-------------+---+---------------------
 "vertex[]" | 3 | "Laurence Fishburne"
```

`collect()` over a property still returns a jsonb list, so stored data keeps its shape.

© 2026 SKAI Worldwide Co., Ltd.

`DISTINCT` on the projection as a whole deduplicates rows, and is what you want after a pattern that reaches the same thing by several routes:

```
MATCH (:person {name: 'Keanu Reeves'})-[:acted_in]->(:movie)<-[:acted_in]-(co:person)
RETURN DISTINCT co.name AS co_actor ORDER BY co_actor;
```

```
      co_actor
---------------------
 "Carrie-Anne Moss"
 "Laurence Fishburne"
```

UNWIND

`UNWIND` is the opposite of `collect()`: it takes a list and produces one row per element.

```
UNWIND [1999, 2010, 2014] AS y MATCH (m:movie {year: y}) RETURN y, m.title AS title ORDER BY
y, title;
```

```
  y   |     title
------+---------------
 1999 | "The Matrix"
 2010 | "Inception"
 2014 | "Interstellar"
```

It also unpacks a list you built earlier in the same pipeline, which is how you filter on an aggregate and then get back to one row per element:

```
MATCH (m:movie)<-[:acted_in]-(p:person)
WITH m, collect(p) AS players
WHERE cardinality(players) > 2
UNWIND players AS actor
RETURN m.title AS film, actor.name AS actor ORDER BY actor;
```

```
    film     |        actor
-------------+---------------------
 "The Matrix" | "Carrie-Anne Moss"
 "The Matrix" | "Keanu Reeves"
 "The Matrix" | "Laurence Fishburne"
```

`FOR` is the GQL spelling, and it adds `WITH OFFSET` for the element's 0-based index. See **`UNWIND`**.

Ordering and paging

`ORDER BY`, `SKIP`, `OFFSET` and `LIMIT` are clauses in their own right. They can sit at the end of a `RETURN` or a `WITH`, as in SQL:

```
MATCH (m:movie) RETURN m.title AS title, m.year AS year ORDER BY year SKIP 2 LIMIT 3;
```

```
           title             | year
-----------------------------+------
 "Total Recall"              | 1990
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Terminator 2: Judgment Day" | 1991
 "Jurassic Park"              | 1993
```

or stand alone in the middle of a pipeline, which the `RETURN` form cannot do:

```
MATCH (m:movie) LET y = m.year ORDER BY y LIMIT 3 RETURN m.title AS title, y;
```

```
      title       |  y
------------------+------
 "Blade Runner"   | 1982
 "The Terminator" | 1984
 "Total Recall"   | 1990
```

They cannot end a statement: a query still has to finish with `RETURN`, `FINISH` or a write clause.

A sort key is ordered as the value it is, not as its jsonb picture, so an index on the property can serve the ordering. That is the whole reason to care: **Indexes and constraints**.

UNION

`UNION` stacks two result sets that have the same column names, and deduplicates. `UNION ALL` keeps duplicates.

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.title AS name
UNION
MATCH (p:person {name: 'Keanu Reeves'}) RETURN p.name AS name;
```

```
      name
---------------
 "Keanu Reeves"
 "The Matrix"
```

This is also the tidy way round the missing edge-label alternation: two narrow patterns unioned scan two edge tables, where one untyped pattern scans them all. See **Reading a graph**.

The GQL clauses

Four ISO/IEC 39075 clauses do jobs that Cypher needs several clauses for. They are accepted in the same statements as everything else, and there is no separate mode.

`LET` adds computed values without dropping anything in scope. Where `WITH` makes you list every variable you still want, `LET` keeps them all:

```
MATCH (m:movie) LET decade = (m.year / 10) * 10 FILTER decade = 2010
RETURN m.title AS title, m.year AS year, decade ORDER BY year;
```

```
        title         | year | decade
----------------------+------+--------
 "Inception"          | 2010 | 2010
 "The Avengers"       | 2012 | 2010
 "Elysium"            | 2013 | 2010
 "Interstellar"       | 2014 | 2010
 "Blade Runner 2049"  | 2017 | 2010
```


© 2026 SKAI Worldwide Co., Ltd.

`FILTER` filters rows and nothing else, so a filter no longer has to be attached to a `WITH`. It takes a bare predicate or the `FILTER WHERE` spelling.

`NEXT` chains one query onto the result of another, which is how you filter after a projection without a `WITH`:

```
MATCH (m:movie) RETURN m.genre AS genre, count(*) AS films
NEXT FILTER films > 3
RETURN genre, films ORDER BY genre;
```

```
  genre   | films
----------+-------
 "Action" | 7
 "Sci-Fi" | 10
```

`FINISH` ends a query that projects nothing. It is the counterpart of `RETURN` for a statement run for its effect:

```
MATCH (m:movie {title: 'The Matrix'}) FINISH;
```

```
--
(0 rows)
```

Each has a reference page: **`LET`**, **`FILTER`**, **`NEXT`**, **`FINISH`**.

Subqueries

`CALL { }` runs a subquery once per incoming row and joins its result back. Use it when the thing you want per row is a whole query rather than an expression: a per-row aggregate, a per-row top-N.

Everything about it is ordinary except one thing, and it changes the answer rather than raising an error.

> **IMPORTANT**
>
> A `CALL { }` with no import list does **not** import the outer variables. A name used inside the body that also exists outside is rebound to a fresh, unconstrained variable, so the subquery runs over the whole graph instead of over the current row. Always write the import list.

Here is the difference. `m` is bound outside, the body mentions `m`, and the answer is the same for every film:

```
MATCH (d:person)-[:directed]->(m:movie)
CALL { MATCH (a:person)-[:acted_in]->(m) RETURN count(a) AS cast_size }
RETURN m.title AS film, cast_size ORDER BY film LIMIT 3;
```

```
       film         | cast_size
--------------------+-----------
 "Avatar"           | 25
 "Blade Runner"     | 25
 "Blade Runner 2049" | 25
```

25 is every `acted_in` edge in the graph. The subquery's `m` was not the outer `m`.

Naming the import fixes it:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (d:person)-[:directed]->(m:movie)
CALL (m) { MATCH (a:person)-[:acted_in]->(m) RETURN count(a) AS cast_size }
RETURN m.title AS film, cast_size ORDER BY film LIMIT 3;
```

```
        film         | cast_size
---------------------+-----------
 "Avatar"            | 1
 "Blade Runner"      | 1
 "Blade Runner 2049" | 2
```

`CALL (*) { ... }` imports everything in scope, and `CALL { WITH m ... }` is the older spelling of the same thing. All three agree.

A subquery may not hand back a name that is already bound outside it:

```
MATCH (m:movie) CALL (m) { MATCH (p:person)-[:acted_in]->(m) RETURN m AS m } RETURN count(*);
```

```
ERROR:  variable "m" returned by the CALL subquery is already bound in the outer query
```

Write clauses are not allowed in a subquery body. `OPTIONAL CALL` is the left-join form, producing a row even when the body yields nothing. See **`CALL { }`** and **`OPTIONAL CALL`**.

Subqueries as values

Six value expressions take a subquery and reduce it to something you can put in a `WHERE` or a `RETURN`. They need no import list, because the outer variables are in scope as expressions rather than as a separate query.

`EXISTS { }` is true when the subquery returns any row:

```
MATCH (p:person) WHERE EXISTS { MATCH (p)-[:directed]->(:movie) } RETURN count(p) AS
directors;
```

```
 directors
-----------
 12
```

`COUNT { }` is how many rows it returns:

```
MATCH (m:movie) WHERE COUNT { MATCH (:person)-[:acted_in]->(m) } > 2
RETURN m.title AS title ORDER BY title;
```

```
    title
--------------
 "The Matrix"
```

`COLLECT { }` and `ARRAY { }` gather its single column into a list:

```
MATCH (m:movie {title: 'The Matrix'})
RETURN COLLECT { MATCH (p:person)-[:acted_in]->(m) RETURN p.name ORDER BY p.name } AS players;
```


© 2026 SKAI Worldwide Co., Ltd.

```
                          players
------------------------------------------------------------
 ["Carrie-Anne Moss", "Keanu Reeves", "Laurence Fishburne"]
```

`VALUE { }` takes its single value:

```
MATCH (m:movie {title: 'The Matrix'})
RETURN VALUE { MATCH (p:person)-[:directed]->(m) RETURN p.name ORDER BY p.name LIMIT 1 } AS a_
director;
```

```
    a_director
------------------
 "Lana Wachowski"
```

and `x IN { }` tests membership of its result:

```
MATCH (p:person)
WHERE p.name IN { MATCH (d:person)-[:directed]->(:movie {title: 'Inception'}) RETURN d.name }
RETURN p.name AS name;
```

```
        name
---------------------
 "Christopher Nolan"
```

`EXISTS { }` is the one you will use most, because a bare pattern is not a predicate. See **Reading a graph**. All six are defined in **Subquery expressions**.

Where to go next

- **Paths** — the patterns whose length you do not know in advance.
- **Combining SQL and Cypher** — CTEs, joins to ordinary tables, views.
- **Tuning a query** — what a pipeline looks like as a plan.

Paths

A fixed pattern asks a question whose shape you already know. `(a)-[]->(b)-[]->(c)` is two hops, always. The questions a graph is for are the ones where you do not know the shape: how are these two people connected, what is reachable from here, what is the cheapest route.

Three tools answer those, and they are different from each other. A variable-length edge enumerates every path in a range of lengths. `shortestpath` finds one path of minimum length. `dijkstra` finds one path of minimum **weight**, which is often not the same path.

Everything runs against **the example dataset**, whose `related_to` edges carry a `weight`. Two sections below also need a small four-stop graph, so create it now. It has a direct A → B leg costing 10, a three-leg detour costing 1 each, and a return leg that makes A → B → A a cycle:

```
CREATE GRAPH routelab;
SET graph_path = routelab;
CREATE VLABEL stop;
CREATE ELABEL leg;
```


© 2026 SKAI Worldwide Co., Ltd.

```
CREATE (:stop {name: 'A'}), (:stop {name: 'B'}), (:stop {name: 'C'}), (:stop {name: 'D'});
MATCH (a:stop {name:'A'}), (b:stop {name:'B'}) CREATE (a)-[:leg {cost: 10}]->(b);
MATCH (a:stop {name:'A'}), (c:stop {name:'C'}) CREATE (a)-[:leg {cost: 1}]->(c);
MATCH (c:stop {name:'C'}), (d:stop {name:'D'}) CREATE (c)-[:leg {cost: 1}]->(d);
MATCH (d:stop {name:'D'}), (b:stop {name:'B'}) CREATE (d)-[:leg {cost: 1}]->(b);
MATCH (b:stop {name:'B'}), (a:stop {name:'A'}) CREATE (b)-[:leg {cost: 1}]->(a);
```

Each block below sets `graph_path` to whichever of the two graphs it needs.

A variable number of hops

`*` after an edge's label makes its length a range. The bounds go after the asterisk:

Written	Means
`-[:related_to*2]->`	exactly two edges
`-[:related_to*1..3]->`	one, two or three edges
`-[:related_to*2..]->`	two or more
`-[:related_to*..3]->`	up to three, and at least one
`-[:related_to*]->`	one or more, no upper bound
`-[:related_to*0..1]->`	the vertex itself, or one edge away

A fixed length behaves like the pattern written out longhand:

```
SET graph_path = moviekb;
MATCH (a:movie {title: 'The Matrix'})-[:related_to*2]->(b:movie) RETURN b.title AS reached ORD
ER BY reached;
```

```
          reached
------------------------------
 "Inception"
 "Terminator 2: Judgment Day"
 "The Matrix Revolutions"
```

A range unions the lengths in it, so the shorter reaches come back too:

```
MATCH (a:movie {title: 'The Matrix'})-[:related_to*1..2]->(b:movie) RETURN b.title AS reached
ORDER BY reached;
```

```
          reached
------------------------------
 "Inception"
 "Terminator 2: Judgment Day"
 "The Matrix Reloaded"
 "The Matrix Revolutions"
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "The Terminator"
 "Total Recall"
```

A lower bound of zero includes the starting vertex, matched against nothing at all:

```
MATCH (a:movie {title: 'The Matrix'})-[:related_to*0..1]->(b:movie) RETURN b.title AS reached
ORDER BY reached;
```

```
        reached
-----------------------
 "The Matrix"
 "The Matrix Reloaded"
 "The Terminator"
 "Total Recall"
```

Leaving the upper bound off means “as far as the edges go”:

```
MATCH (a:movie {title: 'The Matrix'})-[:related_to*]->(b:movie) RETURN count(*) AS reachable;
```

```
 reachable
-----------
         7
```

It cannot loop forever

An unbounded traversal on a cyclic graph terminates, because no path may use the same edge twice. `routelab` has a cycle in it, A → B → A. So ask for paths of up to eight edges from A back to A:

```
SET graph_path = routelab;
MATCH p = (a:stop {name: 'A'})-[:leg*..8]->(a)
RETURN length(p) AS hops, [x IN nodes(p) | x.name] AS route ORDER BY hops;
```

```
 hops |           route
------+---------------------------
 2    | ["A", "B", "A"]
 4    | ["A", "C", "D", "B", "A"]
```

Two paths, not an infinite number: each is a walk that reuses no edge. A vertex may repeat, as `A` does at both ends here.

That guarantee is about correctness, not about cost. An unbounded search over a dense graph can still enumerate an enormous number of paths before it runs out of edges. Give the range an upper bound whenever you have any idea what it should be.

Binding the path

`p =` in front of the pattern binds the whole path, and three functions take it apart: `length(p)` is the edge count, `nodes(p)` the vertices in order, `relationships(p)` the edges.

```
SET graph_path = moviekb;
MATCH p = (a:movie {title: 'The Matrix'})-[:related_to*2]->(b:movie)
RETURN length(p) AS hops,
       [x IN nodes(p) | x.title] AS route,
```


© 2026 SKAI Worldwide Co., Ltd.

```
       [r IN relationships(p) | r.weight] AS weights
ORDER BY route;
```

```
 hops |                              route                                 | weights
------+--------------------------------------------------------------------+---------
 2    | ["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"] | [1, 1]
 2    | ["The Matrix", "The Terminator", "Terminator 2: Judgment Day"]  | [3, 1]
 2    | ["The Matrix", "Total Recall", "Inception"]                     | [4, 2]
```

The `[x IN list | expression]` form is a list comprehension, and it is how you render a route readably. `edges(p)` is a synonym for `relationships(p)`.

A path crossing two labels needs one comprehension per property, because a `person` has no `title` and a `movie` has no `name`:

```
MATCH p = shortestpath((a:person {name: 'Keanu Reeves'})-[e:acted_in*..6]-(b:person {name:
'Carrie-Anne Moss'}))
RETURN length(p) AS hops, [x IN nodes(p) | x.name] AS people, [x IN nodes(p) | x.title] AS fil
ms;
```

```
 hops |                  people                   |           films
------+-------------------------------------------+---------------------------
 2    | ["Keanu Reeves", null, "Carrie-Anne Moss"] | [null, "The Matrix", null]
```

Returning the path itself prints every element in full, which is useful when exploring and too much for anything else. `label(x)` is the compact way to see a route's shape.

The shortest path

`shortestpath` takes a pattern with one variable-length edge and returns one path of minimum length, or no row if there is none. It is not "the paths, sorted by length, limit 1". It is a bidirectional search that stops as soon as the two frontiers meet, so it does not enumerate the paths it rejects.

```
MATCH p = shortestpath((a:person {name: 'Keanu Reeves'})-[e*..6]-(b:person {name: 'Arnold
Schwarzenegger'}))
RETURN length(p) AS hops,
       [x IN nodes(p) | label(x)] AS labels,
       [r IN relationships(p) | label(r)] AS via;
```

```
 hops |                labels                |                  via
------+--------------------------------------+---------------------------------------
 3    | ["person", "movie", "movie", "person"] | ["acted_in", "related_to", "acted_in"]
```

Three things about that query are deliberate. The edge is untyped and undirected, because the route crosses two different edge labels, forwards along one and backwards along the other.

The upper bound of 6 is there because an unbounded shortest-path search over an unlabelled edge is the most expensive query in this manual. And both endpoints are bound to variables.

That last one is a hard requirement:

```
MATCH p = shortestpath((:person {name: 'Keanu Reeves'})-[e*..6]-(:person {name: 'Arnold
Schwarzenegger'}))
RETURN length(p);
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  node must be a reference to a specific node
```

Name both ends even when you never use the names. This is the one place where leaving a variable out is not merely wasteful but refused.

`allshortestpaths` is the same search returning every path of that minimum length rather than one of them:

```
MATCH p = allshortestpaths((a:movie {title: 'The Matrix'})-[:related_to*..4]->(b:movie
{title: 'Inception'}))
RETURN length(p) AS hops, [x IN nodes(p) | x.title] AS route;
```

```
 hops |                    route
------+-------------------------------------------
 2    | ["The Matrix", "Total Recall", "Inception"]
```

One route here, because this dataset's `related_to` edges form a tree. On a graph with several equally short routes you get one row each. You want that when the tie is the interesting part.

Neither function accepts an alternation of edge labels. `[:acted_in|directed*..4]` raises `ERROR: multiple types for relationship not supported`. Leave the edge untyped and filter afterwards; see Reading a graph.

The cheapest path

`shortestpath` counts edges. The edges are not always interchangeable: they may carry distance, latency, price or similarity. Counting them then answers the wrong question. `dijkstra` takes a weight expression and minimises the total.

The difference is easiest to see on `routelab`, which has two useful routes from A to B:

```
SET graph_path = routelab;
MATCH p = (a:stop {name: 'A'})-[:leg*..3]->(b:stop {name: 'B'})
RETURN length(p) AS hops, [x IN nodes(p) | x.name] AS route, [r IN relationships(p) | r.cost]
AS costs
ORDER BY hops;
```

```
 hops |        route         |   costs
------+----------------------+-----------
 1    | ["A", "B"]           | [10]
 3    | ["A", "C", "D", "B"] | [1, 1, 1]
```

`shortestpath` picks the one-hop route:

```
MATCH p = shortestpath((a:stop {name: 'A'})-[:leg*..5]->(b:stop {name: 'B'}))
RETURN length(p) AS hops, [x IN nodes(p) | x.name] AS route;
```

```
 hops |   route
------+------------
 1    | ["A", "B"]
```

`dijkstra` picks the three-hop one, because it costs 3 rather than 10:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (a:stop {name: 'A'}), (b:stop {name: 'B'})
MATCH (p, total) = dijkstra((a)-[e:leg]->(b), e.cost)
RETURN total AS cost, length(p) AS hops, [x IN nodes(p) | x.name] AS route;
```

```
 cost | hops |        route
------+------+----------------------
 3    | 3    | ["A", "C", "D", "B"]
```

Two things about the syntax are easy to get wrong.

The pattern holds one edge, not a variable-length one. `dijkstra` does its own expansion, so you write a single hop and it walks as far as it needs to. A `*` in there is refused with `ERROR: variable length relationship is not supported`, and more than one edge with `ERROR: only one relationship is allowed`.

The weight expression names the edge variable. `e.cost` refers to the `e` bound inside the pattern, and is evaluated per edge as the search expands.

`(p, total) = dijkstra(...)` binds two things: the path, and the total weight along it. The one-variable form `p = dijkstra(...)` gives you the path alone.

On the manual's own dataset the weights are on `related_to`:

```
SET graph_path = moviekb;
MATCH (a:movie {title: 'The Matrix'}), (b:movie {title: 'Inception'})
MATCH (p, total) = dijkstra((a)-[e:related_to]->(b), e.weight)
RETURN total AS weight, [x IN nodes(p) | x.title] AS route, [r IN relationships(p) |
r.weight] AS legs;
```

```
 weight |                    route                     |  legs
--------+----------------------------------------------+--------
 6      | ["The Matrix", "Total Recall", "Inception"] | [4, 2]
```

A third argument is a qualifier: a boolean the search applies to each candidate edge, so you can exclude edges rather than pre-filter the graph. Here nothing survives, because the only route out of The Matrix towards Inception runs over a weight-4 edge:

```
MATCH (a:movie {title: 'The Matrix'}), (b:movie {title: 'Inception'})
MATCH (p, total) = dijkstra((a)-[e:related_to]->(b), e.weight, e.weight < 3)
RETURN total AS weight, [x IN nodes(p) | x.title] AS route;
```

```
 weight | route
--------+-------
(0 rows)
```

The qualifier must be a boolean, as every condition must. `LIMIT n` inside the parentheses caps how many paths are returned. Both are described in **`dijkstra`**.

What a path costs

The plan for a variable-length edge has a node of its own, and it names the label it walks and the range it walks it over:

© 2026 SKAI Worldwide Co., Ltd.

```
EXPLAIN (COSTS OFF) MATCH (a:movie {title: 'The Matrix'})-[r:related_to*1..3]->(b:movie)
RETURN b.title;
```

```
                        QUERY PLAN
------------------------------------------------------------------
 Nested Loop
   ->  Seq Scan on movie a
         Filter: (properties.'title' = '"The Matrix"'::jsonb)
   ->  Hash Join
         Hash Cond: (r."end" = b.id)
         ->  Subquery Scan on r
               ->  Graph VLE on related_to [1..3]
                     ->  Result
         ->  Hash
               ->  Seq Scan on movie b
```

`Graph VLE on related_to [1..3]` is the thing to read. Leave the label off and it becomes `Graph VLE on ag_edge`, which walks every edge label in the graph at every hop:

```
EXPLAIN (COSTS OFF) MATCH (a:movie {title: 'The Matrix'})-[r*1..2]->(b:movie) RETURN b.title;
```

```
                        QUERY PLAN
------------------------------------------------------------------
 Nested Loop
   ->  Seq Scan on movie a
         Filter: (properties.'title' = '"The Matrix"'::jsonb)
   ->  Hash Join
         Hash Cond: (r."end" = b.id)
         ->  Subquery Scan on r
               ->  Graph VLE on ag_edge [1..2]
                     ->  Result
         ->  Hash
               ->  Seq Scan on movie b
```

`shortestpath` shows as `Shortestpath VLE [min..max]` over a pair of `Hash2Side` nodes, the two frontiers of the bidirectional search. `dijkstra` shows as a `Dijkstra` node. In all three cases the work per hop is an index lookup on the edge label's `(start, "end")` or `("end", start)` index. So the cost of a path query is the number of edges the frontier touches. Four things reduce it, in order of effect:

1. **Anchor one end** on a selective, indexed property. A traversal from one known vertex is a different query from a traversal from all of them.
2. **Name the edge label.** It is the difference between one table per hop and all of them.
3. **Bound the length.** An upper bound is a promise the executor can use.
4. `ANALYZE` **the edge labels.** Traversal costing is degree estimation, and an unanalysed edge label has no degrees to estimate from. See **Tuning a query**.

Clean up

```
DROP GRAPH routelab CASCADE;
```


© 2026 SKAI Worldwide Co., Ltd.

Where to go next

- Tuning a query — reading a plan, and the settings that reshape one.
- Reading a graph — the fixed patterns these build on.
- Vector search — using a similarity match as the anchor of a traversal.

Indexes and constraints

A property index is a PostgreSQL btree over an expression that reads one key out of a label's `jsonb` property bag. That single sentence explains almost everything about when one helps. It helps exactly when the planner can match your predicate against that expression. It does nothing at all when it cannot.

This page determines, by experiment, which predicates qualify. The answers are not symmetrical and several of them are surprising, so the tables below are the point of the page. **How a graph is stored** shows what an index looks like on disk. The full syntax is in `CREATE PROPERTY INDEX` and `CREATE CONSTRAINT`.

A graph to measure on

The example dataset has 18 films, which is too few for the planner to prefer any index over reading the table. Build a bigger label to work on, and drop it at the end of the page:

```
CREATE TABLE seed_film AS
SELECT 'Film ' || lpad(g::text, 6, '0') AS title,
       1900 + (g % 130) AS year,
       (ARRAY['Action','Sci-Fi','Drama','Comedy'])[1 + (g % 4)] AS genre
FROM generate_series(1, 50000) g;
```

```
CREATE GRAPH idxlab;
SET graph_path = idxlab;
CREATE VLABEL film;
LOAD FROM seed_film AS r CREATE (:film {title: r.title, year: r.year, genre: r.genre});
CREATE PROPERTY INDEX ON film (title);
CREATE PROPERTY INDEX ON film (year);
```

```
ANALYZE idxlab.film;
```

`ANALYZE` matters as much as the index. An index the planner has no statistics for is an index the planner will decline to use.

Every plan below is the planner's own unforced choice on 50,000 rows. Nothing is turned off to make a point.

What an index can serve

Equality is the case everything else is measured against:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.title = 'Film 000042' RETURN f.year;
```

```
                        QUERY PLAN
------------------------------------------------------------
 Index Scan using film_title_idx on film f
   Index Cond: (properties.'title' = '"Film 000042"'::jsonb)
```


© 2026 SKAI Worldwide Co., Ltd.

`Index Cond` is the word to look for. It means the predicate became a search of the index. A predicate that appears as `Filter` instead was evaluated on rows the scan had already fetched. That is the thing the index was supposed to avoid.

A property map in the pattern compiles to the same equality and gets the same plan:

```
EXPLAIN (COSTS OFF) MATCH (f:film {title: 'Film 000042'}) RETURN f.year;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using film_title_idx on film f
   Index Cond: (properties.'title' = '"Film 000042"'::jsonb)
```

A range works, because a btree is ordered:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.year >= 2025 AND f.year < 2030 RETURN f.title;
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------
------
 Bitmap Heap Scan on film f
   Recheck Cond: ((properties.'year' >= '2025'::jsonb) AND (properties.'year' < '2030'::jsonb)
)
   ->  Bitmap Index Scan on film_year_idx
         Index Cond: ((properties.'year' >= '2025'::jsonb) AND (properties.'year' < '2030'::js
onb))
```

The comparison happens in jsonb's own ordering, which orders numbers numerically, so `>= 2025` means what you want. But jsonb sorts types before values: a property holding the string `"2025"` is not between two numbers.

An `IN` list of literals becomes an array search:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.title IN ['Film 000042', 'Film 000043'] RETURN
f.year;
```

```
                                     QUERY PLAN
-------------------------------------------------------------------------------------------------
 Index Scan using film_title_idx on film f
   Index Cond: (properties.'title' = ANY ('{"\"Film 000042\"","\"Film 000043\""}'::jsonb[]))
```

What an index cannot serve

`<>` cannot use an index, for the ordinary reason that "not this one value" is most of the table:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.year <> 2020 RETURN f.title;
```

```
                   QUERY PLAN
-------------------------------------------------
 Seq Scan on film f
   Filter: (properties.'year' <> '2020'::jsonb)
```

The string predicates are the ones that catch people, because `STARTS WITH` looks like it should be a prefix range:

© 2026 SKAI Worldwide Co., Ltd.

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.title STARTS WITH 'Film 00004' RETURN f.year;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on film f
   Filter: string_starts_with(properties.'title', '"Film 00004"'::jsonb)
```

It is not. `STARTS WITH` compiles to a function call, `string_starts_with`, and a btree over `properties.'title'` has no operator class entry for that function. So there is no index path at all, not merely a rejected one. Turning `enable_seqscan` off does not produce one; the plan stays a sequential scan and is marked `Disabled: true`.

`CONTAINS`, `ENDS WITH` and `=~` behave identically, each compiling to its own function:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.title CONTAINS '00042' RETURN f.year;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on film f
   Filter: string_contains(properties.'title', '"00042"'::jsonb)
```

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.title =~ '^Film 00004' RETURN f.year;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on film f
   Filter: string_regex(properties.'title', '"^Film 00004"'::jsonb)
```

`IS NULL` and `IS NOT NULL` also fall back to a scan on an unpromoted property:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.director IS NULL RETURN f.title;
```

```
               QUERY PLAN
-----------------------------------------
 Seq Scan on film f
   Filter: (properties.'director' IS NULL)
```

The reason is worth understanding, because it also tells you why `IS NULL` behaves differently after promotion. `properties.'director'` is null for a row whose bag has no `director` key at all. An expression index over that key stores a null for every such row, so the index would have to be read almost end to end.

On a promoted column the same test is a real `IS NULL` on a real column, and the btree answers it directly.

The table

Measured on 50,000 rows with `film_title_idx` on `title` and `film_year_idx` on `year`, and with `ANALYZE` run:

© 2026 SKAI Worldwide Co., Ltd.

Predicate	Index used
`f.title = 'x'`	Yes — `Index Cond`
`(f:film {title: 'x'})` in the pattern	Yes — `Index Cond`
`f.year >= 2025 AND f.year < 2030`	Yes — `Bitmap Index Scan`
`f.title IN ['x', 'y']`	Yes — `Index Cond` with `= ANY`
`f.year <> 2020`	No — `Seq Scan`
`f.title STARTS WITH 'x'`	No — `string_starts_with` filter
`f.title CONTAINS 'x'`	No — `string_contains` filter
`f.title ENDS WITH 'x'`	No — `string_ends_with` filter
`f.title =~ '^x'`	No — `string_regex` filter
`f.director IS NULL`	No — `Seq Scan`
`f.title IS NOT NULL`	No — `Seq Scan`
`ORDER BY f.title LIMIT n`	Yes — `Index Scan`, no sort

No index type rescues the string predicates for a Cypher predicate: they are function calls, and no operator class claims them. If you need prefix or substring search, **full-text search** is the tool. You can also reach the column from SQL; see **below**.

Ordering

An index serves an `ORDER BY` on the same property. So a "top N by property" query needs no sort at all:

```
EXPLAIN (COSTS OFF) MATCH (f:film) RETURN f.title AS t ORDER BY t LIMIT 5;
```

```
                  QUERY PLAN
-----------------------------------------------
 Limit
   ->  Index Scan using film_title_idx on film f
```

This works because a sort key is ordered as the value it is rather than as a jsonb rendering of it. Ordering by an expression the index does not hold still sorts.

Composite and partial indexes

A property index takes several keys, and the leftmost-prefix rule is PostgreSQL's:

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE PROPERTY INDEX ON film (genre, year);
```

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.genre = 'Comedy' AND f.year >= 2025 RETURN f.title;
```

```
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on film f
   Recheck Cond: ((properties.'genre' = '"Comedy"'::jsonb) AND (properties.'year' >= '2025'::jsonb))
   ->  Bitmap Index Scan on film_genre_year_idx
         Index Cond: ((properties.'genre' = '"Comedy"'::jsonb) AND (properties.'year' >= '2025'::jsonb))
```

A query on `year` alone cannot use it and goes back to the single-key index on `year`. So order the keys equality-first, as you would in SQL.

A `WHERE` clause on the index makes it partial. That is how you index a small, hot subset of a large label:

```
CREATE PROPERTY INDEX film_comedy_year_idx ON film (year) WHERE genre = 'Comedy';
```

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.genre = 'Comedy' AND f.year >= 2025 RETURN f.title;
```

```
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on film f
   Recheck Cond: ((properties.'year' >= '2025'::jsonb) AND (properties.'genre' = '"Comedy"'::jsonb))
   ->  Bitmap Index Scan on film_comedy_year_idx
         Index Cond: (properties.'year' >= '2025'::jsonb)
```

The `genre` test has left the index condition, because every entry in this index is a comedy already; it survives only in the recheck. The index is a quarter of the size of the full one on `year`: 524,288 bytes against 2,056,192. The planner picked it over both alternatives.

`CREATE PROPERTY INDEX` also takes `USING btree | hash | gin | brin | gist | spgist`, `CONCURRENTLY`, `IF NOT EXISTS`, `INCLUDE` and a tablespace. All of them mean what they mean in PostgreSQL.

> **NOTE**
>
> Every bare name inside `CREATE PROPERTY INDEX (...)` is a property key, including the word `properties`. `CREATE PROPERTY INDEX ON film USING gin (properties)` indexes a property named `properties`, which no film has. To index the bag itself, use plain SQL — `CREATE INDEX film_bag_idx ON idxlab.film USING gin (properties)` — and note that only a SQL predicate such as `properties @> '{"genre": "Comedy"}'` can use it. A Cypher property read compiles to `properties.'genre'`, which a GIN index on the bag does not match.

What promotion changes

Promoting a property gives it a real typed column, and every predicate is then compiled against that column instead of against the bag. **Promoting a property to a column** is the page for why and how. What matters here is that it changes the answers in the table above, in both directions.

© 2026 SKAI Worldwide Co., Ltd.

```
ALTER VLABEL film ADD COLUMN year int GENERATED;
ALTER VLABEL film ADD COLUMN title text GENERATED;
```

Start with what promotion does to the indexes you already have. An index over `properties.'year'` is matched by nothing, because no query compiles to that expression any more:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.year >= 2025 AND f.year < 2030 RETURN f.title;
```

```
                  QUERY PLAN
-----------------------------------------------
 Seq Scan on film f
   Filter: ((year >= 2025) AND (year < 2030))
```

The index is still there, still being maintained on every write, and no longer used by anything. Promotion is therefore always two steps: promote, then rebuild the index on the column.

Rebuilding as a property index works for a bare key, which is the common case and the one below. An expression over the key needs the other form. `CREATE PROPERTY INDEX` compiles a bare key to the promoted column, but an expression over that key to the jsonb bag, which is not what the query reads. Write that one as SQL over the column, which `\dGi` still lists:

```
CREATE INDEX film_lower_idx ON idxlab.film (lower(title));
```

See **An expression property index is built over the bag.**

```
DROP PROPERTY INDEX film_year_idx;
DROP PROPERTY INDEX film_title_idx;
DROP PROPERTY INDEX film_genre_year_idx;
DROP PROPERTY INDEX film_comedy_year_idx;
CREATE PROPERTY INDEX ON film (year);
CREATE PROPERTY INDEX ON film (title);
```

The label now looks like this. The new indexes are ordinary column indexes, not expression indexes:

```
\d idxlab.film
```

```
                                                    Table "idxlab.film"
   Column    |  Type   | Collation | Nullable |                                                  Defa
ult
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------
--
 id          | graphid |           | not null | graphid(graph_labid('idxlab.film'::cstring), ne
xtval('idxlab.film_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
 year        | integer |           |          | generated always as (ag_property_text(propertie
s, 'year'::text, 'n'::"char")::integer) stored
 title       | text    |           |          | generated always as (ag_property_text(propertie
s, 'title'::text, 's'::"char")) stored
Indexes:
    "film_pkey" PRIMARY KEY, btree (id)
    "film_bag_idx" gin (properties)
    "film_title_idx" btree (title)
```


© 2026 SKAI Worldwide Co., Ltd.

```
    "film_year_idx" btree (year)
Inherits: idxlab.ag_vertex
```

Equality and range now compare native values rather than jsonb:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.title = 'Film 000042' RETURN f.year;
```

```
                 QUERY PLAN
---------------------------------------------
 Index Scan using film_title_idx on film f
   Index Cond: (title = 'Film 000042'::text)
```

`IS NULL` becomes sargable, which it was not before:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.title IS NULL RETURN f.year;
```

```
                QUERY PLAN
-------------------------------------------
 Index Scan using film_title_idx on film f
   Index Cond: (title IS NULL)
```

`ORDER BY` improves from an index scan to an index-only scan, because the value the query wants is in the index:

```
EXPLAIN (COSTS OFF) MATCH (f:film) RETURN f.title AS t ORDER BY t LIMIT 5;
```

```
                      QUERY PLAN
------------------------------------------------------
 Limit
   ->  Index Only Scan using film_title_idx on film f
```

And two predicates get worse. `IN` no longer uses the index, because the column is boxed back to jsonb to be compared against a jsonb list:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.title IN ['Film 000042', 'Film 000043'] RETURN
f.year;
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------
 Seq Scan on film f
   Filter: (cypher_to_jsonb(title) = ANY ('{"\"Film 000042\"","\"Film 000043\""}'::jsonb[]))
```

`STARTS WITH` is boxed the same way, so promotion does not rescue it either:

```
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.title STARTS WITH 'Film 00004' RETURN f.year;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on film f
   Filter: string_starts_with(cypher_to_jsonb(title), '"Film 00004"'::jsonb)
```


© 2026 SKAI Worldwide Co., Ltd.

But the column is a real `text` column, so SQL can reach it with a real `text` operator, and that is a prefix range:

```
EXPLAIN (COSTS OFF) SELECT count(*) FROM idxlab.film WHERE title LIKE 'Film 00004%';
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------------------
 Aggregate
   ->  Index Only Scan using film_title_idx on film
         Index Cond: ((title >= 'Film 00004'::text) AND (title < 'Film 00005'::text))
         Filter: (title ~~ 'Film 00004%'::text)
```

That is the practical answer for prefix search on a hot key: promote the property and write the prefix predicate in SQL, wrapping the Cypher part if you need one. See **Combining SQL and Cypher**.

The table, after promotion

Predicate	Unpromoted (jsonb)	Promoted (typed column)
`f.title = 'x'`	Yes	Yes
`f.year >= a AND f.year < b`	Yes	Yes
`f.title IN ['x', 'y']`	Yes	**No** — boxed to jsonb
`f.year <> 2020`	No	No
`f.title STARTS WITH 'x'`	No	No
`f.title CONTAINS 'x'`	No	No
`f.title =~ '^x'`	No	No
`f.title IS NULL`	No	**Yes**
`ORDER BY f.title LIMIT n`	Yes — index scan	Yes — index-only scan
SQL `title LIKE 'x%'`	Not applicable	Yes — prefix range

Across a label hierarchy

An index belongs to one table. A pattern on a parent label scans the parent and every label below it, and each of those needs its own index. Build a family to see it:

```
CREATE TABLE seed_work AS SELECT g FROM generate_series(1, 10000) g;
```

```
CREATE VLABEL work;
CREATE VLABEL short INHERITS (work);
CREATE VLABEL feature INHERITS (work);
LOAD FROM seed_work AS r CREATE (:short {code: 'S' || lpad((r.g)::text, 6, '0')});
```


© 2026 SKAI Worldwide Co., Ltd.

```
LOAD FROM seed_work AS r CREATE (:feature {code: 'F' || lpad((r.g)::text, 6, '0')});
CREATE PROPERTY INDEX ON work (code);
```

```
ANALYZE idxlab.work; ANALYZE idxlab.short; ANALYZE idxlab.feature;
```

Only `work` is indexed, and every child is read in full:

```
EXPLAIN (COSTS OFF) MATCH (w:work) WHERE w.code = 'F000042' RETURN w;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Result
   ->  Append
         ->  Seq Scan on work w_1
               Filter: (properties.'code' = '"F000042"'::jsonb)
         ->  Seq Scan on short w_2
               Filter: (properties.'code' = '"F000042"'::jsonb)
         ->  Seq Scan on feature w_3
               Filter: (properties.'code' = '"F000042"'::jsonb)
```

Index each child and every branch of the `Append` gets one:

```
CREATE PROPERTY INDEX ON short (code);
CREATE PROPERTY INDEX ON feature (code);
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Result
   ->  Append
         ->  Seq Scan on work w_1
               Filter: (properties.'code' = '"F000042"'::jsonb)
         ->  Index Scan using short_code_idx on short w_2
               Index Cond: (properties.'code' = '"F000042"'::jsonb)
         ->  Index Scan using feature_code_idx on feature w_3
               Index Cond: (properties.'code' = '"F000042"'::jsonb)
```

`work` itself keeps its sequential scan because it holds no rows of its own, and reading an empty table is cheaper than opening an index. That is the right decision. It is also the shape to watch for: a hierarchy costs you one index per label, forever, for every property you search a family by.

Constraints

Two forms exist, and they are the two PostgreSQL forms underneath.

`ASSERT <key> IS UNIQUE` builds an exclusion constraint over the key's expression:

```
CREATE VLABEL account;
CREATE CONSTRAINT account_email_uq ON account ASSERT email IS UNIQUE;
```

```
CREATE (:account {email: 'a@example.com', age: 30});
CREATE (:account {email: 'a@example.com', age: 31});
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  conflicting key value violates exclusion constraint "account_email_uq"
DETAIL:  Key ((properties.'email'))=("a@example.com") conflicts with existing key
((properties.'email'))=("a@example.com").
```

`ASSERT <predicate>` builds a check constraint:

```
CREATE CONSTRAINT account_age_ck ON account ASSERT age > 0;
CREATE (:account {email: 'b@example.com', age: -1});
```

```
ERROR:  new row for relation "account" violates check constraint "account_age_ck"
DETAIL:  Failing row contains (7.3, {"age": -1, "email": "b@example.com"}).
```

Both are visible as ordinary constraints, and the unique one is backed by a real btree over the expression, so it doubles as an index for lookups on that key:

```
SELECT conname, pg_get_constraintdef(oid) FROM pg_constraint
WHERE conrelid = 'idxlab.account'::regclass AND contype IN ('c', 'x') ORDER BY conname;
```

```
     conname      |               pg_get_constraintdef
------------------+-----------------------------------------------------
 account_age_ck   | CHECK ((properties.'age' > cypher_to_jsonb(0)))
 account_email_uq | EXCLUDE USING btree ((properties.'email') WITH =)
```

Neither constraint says anything about a vertex that lacks the property. A missing key reads as null, a null is not equal to another null, and a check constraint on an unknown value passes:

```
CREATE (:account {email: 'c@example.com'});
CREATE (:account {age: 44});
CREATE (:account {age: 45});
MATCH (a:account) RETURN a.email AS email, a.age AS age ORDER BY email NULLS LAST, age;
```

```
      email      | age
-----------------+-----
 "a@example.com" | 30
 "c@example.com" |
                 | 44
                 | 45
```

Two accounts with no email at all coexist happily. If a key must be present as well as unique, add a second constraint asserting that it is not null. A property bag will not do it for you.

`CREATE UNIQUE PROPERTY INDEX` is the other way to spell uniqueness, and it produces a unique index rather than an exclusion constraint:

```
CREATE VLABEL member;
CREATE UNIQUE PROPERTY INDEX ON member (card);
CREATE (:member {card: '0001'});
CREATE (:member {card: '0001'});
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  duplicate key value violates unique constraint "member_card_idx"
DETAIL:  Key ((properties.'card'))=("0001") already exists.
```

Prefer it for a fast unique lookup. Prefer the named constraint for a rule you want to see in `pg_constraint` and in a dump under a name you chose. Neither is inherited by a child label: a constraint on a parent constrains only the parent's own rows.

Practical order of work

The sequence that answers most "why is this slow" questions, in order:

1. `ANALYZE` the labels involved, edge labels included. An unanalysed edge label costs more than a missing index; see **Tuning a query**.
2. Index the property your pattern is **anchored** on: the one selective equality that picks the starting vertices. Everything downstream is a join through the edge indexes that already exist.
3. Check with `EXPLAIN` that the predicate arrives as `Index Cond` and not as `Filter`.
4. Only then consider promoting the property, which is the answer when the read itself is the cost rather than the search.

Clean up

```
DROP GRAPH idxlab CASCADE;
```

```
DROP TABLE seed_film;
DROP TABLE seed_work;
```

Where to go next

- **Promoting a property to a column** — the other half of the story above.
- **Tuning a query** — plan shape, statistics and the settings that change them.
- **Full-text search** — the indexable answer to substring and prefix matching.

Promoting a property to a column

A label is schemaless because a property lives in a per-element `jsonb` document. That costs you two things.

Reading `m.year` means finding the `year` key inside a JSON document, for every row the query touches. And the value the read produces is a jsonb value, so an index over it is an index over jsonb. It is never an index over an `integer` or a `vector`.

Promotion removes both costs for one chosen key. `ALTER VLABEL film ADD COLUMN year int GENERATED` gives the `year` property a real `integer` column, stored beside the bag and maintained by the server. It also teaches the Cypher compiler to answer `f.year` from that column.

Queries do not change. Nothing about the property changes either: it is still a property, the label is still schemaless in every other key, and the promotion can be undone.

This page is about when that trade is worth making and what you take on with it. The DDL syntax is in `ALTER VLABEL` and `CREATE VLABEL`.

© 2026 SKAI Worldwide Co., Ltd.

A label of your own to promote on

Promotion is a change to a label's shape, so work on a copy rather than on the manual's shared graph. `LOAD FROM` over a label table copies its property bags straight across:

```
CREATE GRAPH promolab;
SET graph_path = promolab;
CREATE VLABEL film;
LOAD FROM moviekb.movie AS r CREATE (:film = r.properties);
```

That gives you the 18 films of **the example dataset** in a graph you can reshape freely. The whole page runs there and drops it at the end.

What promotion does

Before promoting, a filter on `year` reaches into the bag. `EXPLAIN (VERBOSE)` shows the property reads as `properties.'key'`:

```
EXPLAIN (VERBOSE, COSTS OFF) MATCH (f:film) WHERE f.year > 2010 RETURN f.title;
```

```
                    QUERY PLAN
--------------------------------------------------
 Seq Scan on promolab.film f
   Output: f.properties.'title'
   Filter: (f.properties.'year' > '2010'::jsonb)
```

Promote it:

```
ALTER VLABEL film ADD COLUMN year int GENERATED;
```

```
\d promolab.film
```

```
                                                    Table "promolab.film"
   Column    |  Type   | Collation | Nullable |                                                    Defa
ult
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------
--
 id          | graphid |           | not null | graphid(graph_labid('promolab.film'::cstring),
nextval('promolab.film_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
 year        | integer |           |          | generated always as (ag_property_text(propertie
s, 'year'::text, 'n'::"char")::integer) stored
Indexes:
    "film_pkey" PRIMARY KEY, btree (id)
Inherits: promolab.ag_vertex
```

The column is `GENERATED ALWAYS AS ... STORED`: an ordinary PostgreSQL generated column whose expression pulls one key out of the bag. `ag_property_text(properties, 'year', 'n')` returns the key's text only when the value is of the kind the column can hold: `'n'` for a number, `'s'` for a string, `'b'` for a boolean.

The cast to `integer` does the rest. Nobody writes to it; the server computes it on every insert and update.

The same query now filters on the column:

© 2026 SKAI Worldwide Co., Ltd.

```
EXPLAIN (VERBOSE, COSTS OFF) MATCH (f:film) WHERE f.year > 2010 RETURN f.title;
```

```
           QUERY PLAN
--------------------------------
 Seq Scan on promolab.film f
   Output: f.properties.'title'
   Filter: (f.year > 2010)
```

`f.title` is still a bag read, because only `year` was promoted. The answers are unchanged:

```
MATCH (f:film) WHERE f.year > 2010 RETURN f.title AS title, f.year AS year ORDER BY year;
```

```
           title            | year
----------------------------+------
 "The Avengers"             | 2012
 "Elysium"                  | 2013
 "Interstellar"             | 2014
 "Blade Runner 2049"        | 2017
 "The Matrix Resurrections" | 2021
```

Existing rows are filled in as part of the `ALTER`, so the column is complete the moment it exists. And the value is in **both** places. The bag still holds it:

```
SELECT properties->>'title' AS title, year, properties->>'year' AS in_the_bag
FROM promolab.film ORDER BY year LIMIT 3;
```

```
     title      | year | in_the_bag
----------------+------+------------
 Blade Runner   | 1982 | 1982
 The Terminator | 1984 | 1984
 Total Recall   | 1990 | 1990
```

That duplication is deliberate. The property stays a property, so a client reading `properties` sees what it always saw. A dump carries the data once in the bag and regenerates the column. Dropping the column loses nothing. The price is storage: you are paying for the value twice.

The mapping from property key to column is recorded in `ag_label_property`:

```
SELECT l.labname, p.propname, a.attname, format_type(a.atttypid, a.atttypmod) AS type
FROM pg_catalog.ag_label l
  JOIN pg_catalog.ag_label_property p ON p.laboid = l.oid
  JOIN pg_attribute a ON a.attrelid = l.relid AND a.attnum = p.attnum
WHERE l.relid = 'promolab.film'::regclass;
```

```
 labname | propname | attname |  type
---------+----------+---------+---------
 film    | year     | year    | integer
```

That row makes the read resolve to the column. Without it the column would exist, but Cypher would not know about it. See The long form registers nothing.

© 2026 SKAI Worldwide Co., Ltd.

A promoted read has the column's type

A Cypher `RETURN` hands back a property read as jsonb, so `RETURN f.title` prints `"The Matrix"` with the quotes. Reading a promoted key is the exception: it comes back as the column's declared type. Here `year` is promoted and `title` is not.

```
SELECT pg_typeof(t) AS title_type, pg_typeof(y) AS year_type
FROM (MATCH (f:film) RETURN f.title AS t, f.year AS y LIMIT 1) q;
```

```
 title_type | year_type
------------+-----------
 jsonb      | integer
```

Values and orderings are the same either way; the type the client receives is not. A driver that expects every Cypher column to arrive as JSON will see an `integer` here.

This is usually a simplification, because it is the shortest route to a plain value. But it is a change in the result's shape, and worth checking against whatever consumes it.

Indexing a promoted column

`CREATE PROPERTY INDEX` on a promoted key builds a plain btree over the column rather than an expression index over the bag. The planner uses it as it would any column index. **Indexes and constraints** measures which predicates it can serve, and the two that get worse after promotion. The short version:

- Equality, ranges and `ORDER BY` are served, and `ORDER BY` improves to an index-only scan.
- `IS NULL` becomes indexable, which it was not on the bag.
- `IN` and `STARTS WITH` stop using the index, because the column is boxed back to jsonb to meet the jsonb operand.

One rule follows, and it is easy to miss: **an index over the bag is dead the moment you promote the key.** Nothing compiles to `properties.'year'` any more, so an index over that expression is never matched. It is still maintained on every write. Promoting is always two steps: promote, then rebuild the index on the column.

Declaring promoted columns with the label

At creation time the columns go in the label definition:

```
CREATE VLABEL doc (words int GENERATED, tag text GENERATED (category));
```

`words int GENERATED` promotes the property `words` to a column of the same name. The parenthesised form promotes a different key: `tag text GENERATED (category)` says "column `tag`, filled from the property `category`". Use it when the key you want is not a legal or sensible column name.

The catalog records the key, not the column name:

```
SELECT l.labname, p.propname, a.attname, format_type(a.atttypid, a.atttypmod) AS type
FROM pg_catalog.ag_label l
  JOIN pg_catalog.ag_label_property p ON p.laboid = l.oid
  JOIN pg_attribute a ON a.attrelid = l.relid AND a.attnum = p.attnum
WHERE l.relid = 'promolab.doc'::regclass ORDER BY p.propname;
```

```
 labname | propname | attname |  type
---------+----------+---------+---------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 doc     | category | tag     | text
 doc     | words    | words   | integer
```

So the Cypher read that resolves to the column is `d.category`, the key. `d.tag` is a property nobody has ever set:

```
CREATE (:doc {title: 'a note', words: 12, category: 'news'});
MATCH (d:doc) RETURN d.words AS words, d.category AS category, d.tag AS tag;
```

```
 words | category | tag
-------+----------+-----
    12 | news     |
```

`ALTER ELABEL` promotes an edge property the same way. It is worth doing for a weight you traverse on.

Extension types

The column's type can come from an extension. That is the case that makes promotion more than a micro-optimisation. An embedding stored as a property is a JSON array, so every similarity query has to cast it: `p.emb::vector(4)`, on every row, every time. Promote it and the cast happens once, at write time:

```
CREATE EXTENSION IF NOT EXISTS vector;
```

```
CREATE VLABEL page (emb vector(4) GENERATED);
```

```
\d promolab.page
```

```
                                                  Table "promolab.page"
   Column    |   Type    | Collation | Nullable |                                                  Defa
ult
-------------+-----------+-----------+----------
+--------------------------------------------------------------------------------------------------
 id          | graphid   |           | not null | graphid(graph_labid('promolab.page'::cstring)
, nextval('promolab.page_id_seq'::regclass))
 properties | jsonb      |           | not null | jsonb_build_object()
 emb        | vector(4) |           |          | generated always as (((properties ->>
'emb'::text))::vector(4)) stored
Indexes:
    "page_pkey" PRIMARY KEY, btree (id)
Inherits: promolab.ag_vertex
```

Note the generation expression. For a type with no JSON counterpart the server uses plain `->>` rather than `ag_property_text`, and leaves the type's own input function to accept or refuse the text. `vector(4)` refuses anything that is not four numbers in brackets.

The column can then carry a real vector index. The index definition needs no cast expression, because there is no cast left to make. Load enough rows for an index to be worth choosing first:

```
CREATE TABLE seed_page AS
SELECT 'page ' || g AS title, jsonb_build_array(random(), random(), random(), random()) AS emb
FROM generate_series(1, 5000) g;
```


© 2026 SKAI Worldwide Co., Ltd.

```
LOAD FROM seed_page AS r CREATE (:page {title: r.title, emb: r.emb});
CREATE PROPERTY INDEX ON page USING hnsw (emb vector_cosine_ops);
```

```
ANALYZE promolab.page;
```

and a Cypher similarity search uses it:

```
EXPLAIN (COSTS OFF)
MATCH (p:page) RETURN p.title AS title
ORDER BY p.emb <=> '[0.1,0.2,0.3,0.4]'::vector(4) LIMIT 3;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Limit
   ->  Index Scan using page_emb_idx on page p
         Order By: (emb <=> '[0.1,0.2,0.3,0.4]'::vector(4))
```

The unpromoted form of the same query has to compute a distance for every row and sort them:

```
SET enable_property_promotion = off;
EXPLAIN (COSTS OFF)
MATCH (p:page) RETURN p.title AS title
ORDER BY p.emb::vector(4) <=> '[0.1,0.2,0.3,0.4]'::vector(4) LIMIT 3;
RESET enable_property_promotion;
```

```
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Limit
   ->  Sort
         Sort Key: (((p.properties.'emb')::vector(4) <=> '[0.1,0.2,0.3,0.4]'::vector(4)))
         ->  Seq Scan on page p
```

An `::vector(4)` you leave in place on a promoted column is folded away rather than refused, so existing queries keep working and keep using the index.

Prefer promotion to a property index over a cast expression such as `((emb::vector(4)) vector_cosine_ops)`. Over a promoted column the index is an ordinary column index, which is the more durable form across a dump and restore. See Vector search.

What promotion costs you

A promoted key is no longer schemaless

This is the real trade. Every other key on the label still accepts any JSON value. The promoted one accepts only what the column can hold, and a write that does not fit aborts:

```
CREATE (:film {title: 'Bad Year', year: 'nineteen ninety nine'});
```

```
ERROR:  property "year" holds a string
DETAIL:  A column promoting this property holds numbers.
CONTEXT:  while computing the promoted typed columns of graph label "film"
```


© 2026 SKAI Worldwide Co., Ltd.

```
CREATE (:film {title: 'Fractional', year: 1999.5});
```

```
ERROR:  invalid input syntax for type integer: "1999.5"
CONTEXT:  while computing the promoted typed columns of graph label "film"
```

Two different checks are visible there. The first is the kind check: a string is not a number, whatever it says. The second is the column type's own. For an extension type only the second applies:

```
CREATE (:page {title: 'wrong dims', emb: [1, 2, 3]});
```

```
ERROR:  expected 4 dimensions, not 3
CONTEXT:  while computing the promoted typed columns of graph label "page"
```

Leaving the key out entirely is always fine. An absent property makes a null column, exactly as an absent property reads as null:

```
CREATE (:film {title: 'Undated'});
MATCH (f:film {title: 'Undated'}) RETURN f.year AS year;
```

```
 year
------

```

Promoting is therefore a decision about data quality as much as about speed. Before you promote a key on a live label, find out whether every existing value fits. The `ALTER` applies the same checks to the rows already there, and fails on the first one that does not:

```
CREATE VLABEL city;
CREATE (:city {name: 'Seoul', pop: 9700000}), (:city {name: 'Nowhere', pop: 'many'});
```

```
ALTER VLABEL city ADD COLUMN pop bigint GENERATED;
```

```
ERROR:  property "pop" holds a string
DETAIL:  A column promoting this property holds numbers.
```

That failure is the useful kind: it tells you the label's data is not what you thought before anything is committed.

The ALTER rewrites the table

Adding a stored generated column rewrites the label's heap and takes an `ACCESS EXCLUSIVE` lock while it does. On this 18-row label it is instant. On a large label, plan for it as you would plan any `ALTER TABLE` rewrite. Check the relation's file number if you want to see it happen:

```
SELECT relfilenode FROM pg_class WHERE oid = 'promolab.film'::regclass;
```

The number before the `ALTER` and the number after it are different.

© 2026 SKAI Worldwide Co., Ltd.

A promoted column cannot have a collation of its own

A property map compares strings with the database default collation. A column collated any other way would order and compare the same values differently. That would make “which of the two answers this read” a decision about results rather than about speed.

So it is refused, including when the collation arrives through a domain rather than being written down:

```
CREATE DOMAIN sortkey AS text COLLATE "C";
```

```
CREATE VLABEL note;
ALTER VLABEL note ADD COLUMN k sortkey GENERATED;
```

```
ERROR:  a promoted property cannot have a collation of its own
DETAIL:  Column "k" is collated "C", while a property map compares strings with the database d
efault.
HINT:  Leave the collation unspecified, or use a type that does not carry one.
```

A view over a Cypher query pins the column

A view is compiled once, and a view over a Cypher query that resolved a property to a column holds a dependency on that column:

```
CREATE VIEW recent_films AS
SELECT title #>> '{}' AS title, year
FROM (MATCH (f:film) WHERE f.year > 2010 RETURN f.title AS title, f.year AS year) q;
```

```
ALTER VLABEL film DROP COLUMN year;
```

```
ERROR:  cannot drop column year of table promolab.film because other objects depend on it
DETAIL:  view recent_films depends on column year of table promolab.film
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

That is correct: the view would otherwise silently start reading the bag. But it means a view is a commitment. Drop the view, drop the column, recreate the view. The dependency is visible in `pg_get_viewdef`, which shows the lowered SQL naming `f.year`: what a view over a Cypher query looks like.

```
DROP VIEW recent_films;
```

An expression property index is built over the bag

A bare key and the same key inside an expression are resolved differently by `CREATE PROPERTY INDEX`. A bare key becomes the promoted column. An expression over that key becomes an expression over the jsonb bag:

```
ALTER VLABEL film ADD COLUMN title text GENERATED;
CREATE PROPERTY INDEX film_title_idx ON film (title);
CREATE PROPERTY INDEX film_lower_idx ON film ((lower(title)));
```

```
SELECT c.relname, pg_get_indexdef(c.oid) AS real_definition
FROM pg_class c JOIN pg_index i ON i.indexrelid = c.oid
```


© 2026 SKAI Worldwide Co., Ltd.

```
WHERE i.indrelid = 'promolab.film'::regclass AND c.relname LIKE 'film_%idx'
ORDER BY 1;
```

```
    relname     |                                  real_definition
----------------
+-----------------------------------------------------------------------------------
 film_lower_idx | CREATE INDEX film_lower_idx ON promolab.film USING btree (lower(properties.'
title'))
 film_title_idx | CREATE INDEX film_title_idx ON promolab.film USING btree (title)
```

A query reads the column in both cases, so it is the bare form that gives an index the query can use.

`\dGi` reports both in Cypher form, which spells each one as `title`:

```
\dGi promolab.*
```

```
                                                 List of property indexes
  Graph   | LabelName |   IndexName    | Unique | Owner |                                  Ind
exdef
---------+-----------+----------------+--------+-------
+-----------------------------------------------------------------------------
 promolab | film      | film_lower_idx | f      | agens | CREATE PROPERTY INDEX film_lower_idx
ON film USING btree ((lower((title))))
 promolab | film      | film_title_idx | f      | agens | CREATE PROPERTY INDEX film_title_idx
ON film USING btree (title)
```

So read `pg_get_indexdef` when you want to know which of the two a label's index is over.

There are two ways to index a promoted key. Index it bare, and the query uses the column directly. Or, when the expression itself is what you need indexed, write it as SQL over the column, which matches what the query reads:

```
CREATE INDEX film_lower_col_idx ON promolab.film (lower(title));
```

Either way, drop an expression index before promoting the key it reads, and rebuild it in one of these two forms.

The long form registers nothing unless it is a single key

A generated column written the long way is an ordinary generated column. It is recognised as a promotion only when its expression is exactly one key out of the bag: `(properties ->> 'key')::type`, or the `ag_property_text` form a dump emits. Anything derived from more than one key, or computed, registers no mapping:

```
CREATE VLABEL post;
ALTER VLABEL post ADD COLUMN body_len int GENERATED ALWAYS AS (length(properties->>'body')) ST
ORED;
CREATE (:post {body: 'hello there'});
```

The column is there and SQL can read it:

```
SELECT id, body_len, properties FROM promolab.post;
```

```
 id  | body_len |       properties
-----+----------+-------------------------
 8.1 |       11 | {"body": "hello there"}
```


© 2026 SKAI Worldwide Co., Ltd.

Cypher cannot:

```
MATCH (p:post) RETURN p.body_len AS body_len;
```

```
 body_len
----------
```

`p.body_len` reads a property named `body_len`, which no post has, and there is no key such a column could be the value of. So the column looks promoted without being promoted. Use the `GENERATED` shorthand for a promotion, and the long form for a derived column you intend to read from SQL only.

Turning resolution off

`enable_property_promotion` controls whether a Cypher property read resolves to a column. It is on by default. Turned off, the same query goes back to the bag:

```
SET enable_property_promotion = off;
EXPLAIN (COSTS OFF) MATCH (f:film) WHERE f.year > 2010 RETURN f.title;
RESET enable_property_promotion;
```

```
                  QUERY PLAN
-----------------------------------------------
 Seq Scan on film f
   Filter: (properties.'year' > '2010'::jsonb)
```

and gives the same answers, because the bag still holds the value. That is the setting's purpose: to establish whether a promotion is responsible for a plan you did not expect, without dropping the column. It does not stop the column being maintained.

Undoing a promotion

```
ALTER VLABEL film DROP COLUMN year;
```

The column and its catalog row go, any index on it goes with it, and reads fall back to the bag with the same results:

```
MATCH (f:film) WHERE f.year > 2010 RETURN count(f) AS films;
```

```
 films
-------
 5
```

Nothing is lost, because nothing was ever stored only in the column. The one thing to clear first is any view that pinned it.

When to promote

Promote a key when all four are true:

- **Queries filter, join or order on it**, rather than merely projecting it. A key you only ever return gains nothing. The bag read is already the cheapest way to get it.

© 2026 SKAI Worldwide Co., Ltd.

- **Its values are uniform.** If some rows hold a number and some a string, the `ALTER` will refuse, and forcing them into one type is a data change, not a tuning change.
- **You want an index over the value's own type** — a range on a real `integer`, an index-only `ORDER BY`, an HNSW index on a `vector`.
- **The label is big enough for the read to matter.** On 18 rows nothing here is measurable.

Do not promote a key because it appears in every vertex. Uniformity is a precondition, not a reason. And check the two predicates that get worse, `IN` and `STARTS WITH`, against what your application sends.

Clean up

```
DROP GRAPH promolab CASCADE;
```

```
DROP TABLE seed_page;
DROP DOMAIN sortkey;
```

Where to go next

- Indexes and constraints — the sargability tables, before and after promotion.
- Tuning a query — where promotion sits among the other plan-shaping choices.
- Vector search — embeddings, with and without a promoted column.

Combining SQL and Cypher

There is no bridge between SQL and Cypher in AgensGraph, because there is nothing to bridge. A graph is a set of PostgreSQL tables in the same database as your ordinary tables, and both languages compile to the same plan tree.

So the two mix by nesting: a Cypher query goes where SQL expects a table, and a SQL query goes where Cypher expects a value.

This page walks the crossing points, in the order you are likely to need them, and then the one rule that surprises everybody: what a Cypher `RETURN` hands back.

Everything runs against **the example dataset**, which includes an ordinary table, `rating`, that is not part of the graph.

Reading a label with plain SQL

A label is a table, so no Cypher is needed to read one:

```
SELECT id, properties->>'title' AS title, properties->>'year' AS year
FROM moviekb.movie ORDER BY (properties->>'year')::int LIMIT 3;
```

```
  id  |     title      | year
------+----------------+------
 3.8  | Blade Runner   | 1982
 3.15 | The Terminator | 1984
 3.11 | Total Recall   | 1990
```


© 2026 SKAI Worldwide Co., Ltd.

Nothing needs enabling for that. Writing rows into a label with SQL does; see **What is deliberately closed off**. Reading is just reading.

Direct SQL is the right tool when the question is relational: counts per label, a join to a reference table on a property value, a window function over a property. It is the wrong tool for anything with a variable number of hops. The rest of this page covers those.

A Cypher query in a SQL FROM

Put a Cypher query in parentheses where a SQL `FROM` expects a table, and give it an alias. Its `RETURN` list becomes the columns:

```
SELECT * FROM (MATCH (m:movie) WHERE m.year > 2015 RETURN m.title AS title, m.year AS year) q
ORDER BY year;
```

```
          title            | year
---------------------------+------
 "Blade Runner 2049"       | 2017
 "The Matrix Resurrections" | 2021
```

The alias is required:

```
SELECT count(*) FROM (MATCH (m:movie) RETURN m.title AS title);
```

```
ERROR:  Cypher in FROM must have an alias
HINT:  For example, FROM (...) [AS] foo.
```

Two more rules. Every column needs a name, because the outer query has to be able to refer to it. So `RETURN m.title` without `AS` is refused. And the nested query must be a read: a write clause inside a `FROM` is a syntax error.

You can nest more than one, and join them like any two subqueries:

```
SELECT d.name #>> '{}' AS director, a.name #>> '{}' AS actor
FROM (MATCH (p:person)-[:directed]->(:movie {title: 'The Matrix'}) RETURN p.name AS name) d
CROSS JOIN (MATCH (p:person)-[:acted_in]->(:movie {title: 'The Matrix'}) RETURN p.name AS
name) a
ORDER BY director, actor LIMIT 4;
```

```
    director     |       actor
-----------------+--------------------
 Lana Wachowski  | Carrie-Anne Moss
 Lana Wachowski  | Keanu Reeves
 Lana Wachowski  | Laurence Fishburne
 Lilly Wachowski | Carrie-Anne Moss
```

Joining a graph to an ordinary table

This is the reason most people come to this page. Ask the graph a graph question, then join its answer to relational data:

```
SELECT q.title #>> '{}' AS film, r.score
FROM (MATCH (:person {name: 'Christopher Nolan'})-[:directed]->(m:movie) RETURN m.title AS tit
le) q
```


© 2026 SKAI Worldwide Co., Ltd.

```
JOIN rating r ON r.title = q.title #>> '{}'
WHERE r.source = 'critics'
ORDER BY r.score DESC;
```

```
     film     | score
--------------+-------
 Inception    |   8.8
 Interstellar |   8.7
```

The `#>> '{}'` on both sides is the whole trick, and **the next section** explains it. Without it you are comparing a `jsonb` string against a `text` column, which does not type-check.

A graph and a table have no foreign keys between them, so the join key is a value: a title, a code, an external id. If you control the model, store that id as a property and index it.

If the join is hot, promote it to a column: a `text` column joins to a `text` column with no casting at all. See **Promoting a property to a column**.

A Cypher query in a CTE

A `WITH` clause takes a Cypher query as easily as a `SELECT`. That gives you a name you can reference more than once:

```
WITH films AS (MATCH (m:movie) WHERE m.year > 2010 RETURN m.title AS title, m.year AS year)
SELECT f.title #>> '{}' AS title, r.score
FROM films f JOIN rating r ON r.title = f.title #>> '{}'
WHERE r.source = 'critics' ORDER BY r.score DESC;
```

```
          title           | score
--------------------------+-------
 Interstellar             |   8.7
 Blade Runner 2049        |   8.0
 The Avengers             |   8.0
 Elysium                  |   6.6
 The Matrix Resurrections |   5.7
```

Note which language the **statement** is in. A statement beginning with `WITH cte AS (...)` is SQL, and its body has to be SQL. This is not Cypher's `WITH`, which is a pipeline divider and cannot name a subquery. See **How they differ**.

The pattern generalises to aggregating a traversal relationally. Here Cypher counts each film's cast and SQL averages the scores:

```
WITH cast_size AS (
  MATCH (p:person)-[:acted_in]->(m:movie) RETURN m.title AS title, count(p) AS players
)
SELECT c.title #>> '{}' AS film, c.players #>> '{}' AS players, round(avg(r.score), 2) AS mean
_score
FROM cast_size c JOIN rating r ON r.title = c.title #>> '{}'
GROUP BY 1, 2 ORDER BY 3 DESC LIMIT 4;
```

```
           film            | players | mean_score
---------------------------+---------+------------
 The Matrix                | 3       |       8.90
 Inception                 | 1       |       8.90
 Interstellar              | 1       |       8.65
 Terminator 2: Judgment Day | 2       |       8.60
```


© 2026 SKAI Worldwide Co., Ltd.

SQL inside a Cypher query

The nesting works in the other direction too. A SQL scalar subquery is an expression, so it goes anywhere Cypher wants one: in a `WHERE`, in a `RETURN`, in a property value being written:

```
MATCH (m:movie)
WHERE m.title = (SELECT title FROM rating WHERE source = 'critics' ORDER BY score DESC LIMIT
1)
RETURN m.title AS title, m.year AS year;
```

```
    title    | year
-------------+------
 "Inception" | 2010
```

The subquery must return one row and one column, as in SQL. No cast was needed here: the subquery's `text` was compared against a `jsonb` property, and the comparison resolved. The other direction is the one that needs `#>> '{}'`: a `jsonb` property into a `text` column.

A user-defined function is an expression as well, so a SQL function can be called from a Cypher `RETURN`:

```
CREATE FUNCTION decade(y int) RETURNS int AS 'SELECT (y / 10) * 10' LANGUAGE sql IMMUTABLE;
```

```
MATCH (m:movie) RETURN decade(m.year::int) AS decade, count(*) AS films ORDER BY decade;
```

```
 decade | films
--------+-------
 1980   | 2
 1990   | 6
 2000   | 4
 2010   | 5
 2020   | 1
```

The `::int` on `m.year` turns the jsonb property into an argument the function's signature accepts. Any language works: PL/pgSQL, PL/Python, C. So does any aggregate or window function, because Cypher's aggregates are PostgreSQL's.

Getting a plain value out

Anything a Cypher `RETURN` derives from a property comes back as `jsonb`. That is not a display quirk; it is the column's type. Casting or unwrapping inside the query does not change it, because the projection re-boxes the result.

This is the single most common source of confusion at this boundary, so here it is from every angle.

Not every column is boxed, only the property-derived ones. Checking the outer types of a Cypher subquery's columns shows the rule exactly:

```
SELECT pg_typeof(raw) AS raw, pg_typeof(cast_text) AS cast_text,
       pg_typeof(unwrapped) AS unwrapped, pg_typeof(is_modern) AS boolean_expr,
       pg_typeof(gid) AS gid, pg_typeof(v) AS vertex
FROM (MATCH (m:movie {title: 'The Matrix'})
      RETURN m.title AS raw, m.title::text AS cast_text, m.title #>> '{}' AS unwrapped,
             m.year > 1990 AS is_modern, id(m) AS gid, m AS v) q;
```


© 2026 SKAI Worldwide Co., Ltd.

```
  raw   | cast_text | unwrapped | boolean_expr |   gid    | vertex
--------+-----------+-----------+--------------+----------+--------
 jsonb  | jsonb     | jsonb     | boolean      | graphid  | vertex
```

So a boolean expression stays `boolean`, `id(m)` stays `graphid`, and a graph element stays `vertex`, `edge` or `graphpath`. The three property-derived columns arrive as `jsonb`. That includes the one that was cast to `text` and the one that was already unwrapped.

A string property comes back JSON-quoted:

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.title AS t;
```

```
      t
--------------
 "The Matrix"
```

Casting inside the Cypher query does not help, because the outer projection re-boxes the result:

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.title::text AS t;
```

```
      t
--------------
 "The Matrix"
```

```
SELECT pg_typeof(t) AS type FROM (MATCH (m:movie {title: 'The Matrix'}) RETURN m.title::text
AS t) q;
```

```
 type
-------
 jsonb
```

Neither does extracting the text with `#>> '{}'` inside Cypher, for the same reason:

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.title #>> '{}' AS t;
```

```
      t
--------------
 "The Matrix"
```

The way out is to do the unwrapping in SQL, one level up. Wrap the Cypher query and apply `#>> '{}'` outside it. That operator is jsonb's "extract the value at the empty path as text":

```
SELECT t #>> '{}' AS t FROM (MATCH (m:movie {title: 'The Matrix'}) RETURN m.title AS t) q;
```

```
     t
------------
 The Matrix
```


© 2026 SKAI Worldwide Co., Ltd.

```
SELECT pg_typeof(t #>> '{}') AS type FROM (MATCH (m:movie {title: 'The Matrix'}) RETURN
m.title AS t) q;
```

```
 type
------
 text
```

From there the value behaves like any `text`: compare it to a column, pass it to a function, cast it onwards with `(q.year #>> '{}')::int` for a number.

Three notes on this rule.

It applies to the outermost projection only. A property read within a Cypher expression keeps its type all the way through. So `decade(m.year::int)` above works.

A **promoted** property is the exception. A property with a typed column behind it comes back as that column's type, not as jsonb. So `RETURN m.year` on a promoted `year` is already an `integer` and needs no unwrapping. See **A promoted read has the column's type**.

A vertex or an edge comes back as its own composite type, not as jsonb, and SQL can take it apart with field notation:

```
SELECT pg_typeof(m) AS type, (m).id AS id, (m).properties->>'title' AS title
FROM (MATCH (m:movie {title: 'The Matrix'}) RETURN m) q;
```

```
  type  | id  |   title
--------+-----+------------
 vertex | 3.1 | The Matrix
```

The parentheses around `m` are SQL's, required whenever you read a field of a composite column.

A view over a Cypher query

A view cannot begin with a Cypher clause:

```
CREATE VIEW film_rating AS MATCH (m:movie) RETURN m.title AS title;
```

```
ERROR:  syntax error at or near "MATCH"
```

Wrap it in a `SELECT` and it works. So a graph traversal can be published as a relational view for anything that speaks SQL: a BI tool, an ORM, a foreign data wrapper:

```
CREATE VIEW film_rating AS
SELECT q.title #>> '{}' AS title, (q.year #>> '{}')::int AS year, r.score
FROM (MATCH (m:movie) RETURN m.title AS title, m.year AS year) q
JOIN rating r ON r.title = q.title #>> '{}'
WHERE r.source = 'critics';
```

```
SELECT * FROM film_rating ORDER BY score DESC LIMIT 4;
```

```
          title           | year | score
--------------------------+------+-------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 Inception                 | 2010 |   8.8
 Interstellar              | 2014 |   8.7
 The Matrix                | 1999 |   8.7
 Terminator 2: Judgment Day | 1991 |   8.6
```

The view is an ordinary view, so it lands in the SQL search path rather than in the graph. `graph_path` says which graph a label name belongs to; it says nothing about where a view goes:

```
SELECT schemaname, viewname FROM pg_views WHERE viewname = 'film_rating';
```

```
 schemaname |  viewname
------------+-------------
 public     | film_rating
```

What the server stored

A view holds a parsed query tree, not your text. Ask for the definition back and you get the lowered SQL the Cypher compiled to, not the Cypher you wrote:

```
SELECT pg_get_viewdef('public.film_rating'::regclass, true);
```

```
                          pg_get_viewdef
-----------------------------------------------------------------------
  SELECT q.title #>> '{}'::text[] AS title,                           +
     (q.year #>> '{}'::text[])::integer AS year,                      +
     r.score                                                          +
    FROM ( SELECT (_agens_default_s.m).properties.'title' AS title,   +
             (_agens_default_s.m).properties.'year' AS year           +
            FROM ( SELECT ROW(m.id, m.properties, m.ctid)::vertex AS m+
                   FROM moviekb.movie m) _agens_default_s) q          +
      JOIN rating r ON r.title = (q.title #>> '{}'::text[])           +
   WHERE r.source = 'critics'::text;
```

That is worth reading once, because it is the clearest possible statement of what a Cypher query is on this engine. `MATCH (m:movie)` became a scan of `moviekb.movie` whose rows are assembled into a `vertex` value. `m.title` became `(m).properties.'title'`. The names beginning `_agens_default_s` are the generated aliases for the pipeline's stages.

Two consequences follow.

Your Cypher is not recoverable from the catalog. Keep the source of a view under version control, the way you would any other schema object. This is not specific to views. Anything built on a stored query tree records the lowered form: a rule, a materialised view.

It round-trips. `pg_dump` emits that same lowered SQL, and restoring it rebuilds a working view. The definition after a dump and restore into a fresh database is identical, character for character, and the view returns the same rows.

So a view over a Cypher query is safe to rely on operationally, even though it does not read back as you wrote it.

One thing to watch: if the Cypher resolved a property to a **promoted column**, the view holds a dependency on that column. Dropping the column is refused until the view goes. That is deliberate, because the view would otherwise silently change meaning.

© 2026 SKAI Worldwide Co., Ltd.

Clean up

```
DROP VIEW film_rating;
DROP FUNCTION decade(int);
```

Where to go next

- **How a graph is stored** — why any of this works, and the two gates on writing to a label with SQL.
- **Composing a query** — Cypher's own pipeline, and where its `WITH` differs from a CTE.
- **Loading data** — `LOAD FROM`, which reads an ordinary table into a graph.
- **Promoting a property to a column** — for a join key you use constantly.

Tuning a query

A Cypher query is planned by PostgreSQL's planner. There is no separate graph engine with its own rules, so `EXPLAIN` works, statistics matter, and the tuning you already know applies.

The AgensGraph-specific parts are a handful of plan nodes for traversal, one statistics problem that bites every new graph, and five settings that change plan shape.

This page is about plan shape only. Resource-shaped tuning covers `shared_buffers`, `work_mem`, `effective_cache_size`, parallelism and autovacuum. None of it is graph-specific, and it lives in **Configuration**.

A graph to measure on

The example dataset is 46 elements, which is not enough for the planner to make an interesting decision. Build something with a degree distribution and drop it at the end:

```
CREATE TABLE seed_account AS SELECT g AS n, 'user' || g AS handle FROM generate_series(1,
20000) g;
CREATE TABLE seed_follows AS SELECT (1 + (g % 20000)) AS src, (1 + ((g * 7919) % 20000)) AS ds
t FROM generate_series(1, 120000) g;
```

```
CREATE GRAPH perflab;
SET graph_path = perflab;
CREATE VLABEL account;
CREATE ELABEL follows;
LOAD FROM seed_account AS r CREATE (:account {handle: r.handle, n: r.n});
CREATE PROPERTY INDEX ON account (handle);
```

```
LOAD FROM seed_follows AS r MATCH (a:account {n: r.src}), (b:account {n: r.dst}) CREATE (a)-
[:follows]->(b);
```

```
ANALYZE perflab.account;
```

20,000 accounts and 120,000 `follows` edges. Only the vertex label was analysed. That omission is **the point of a later section**.

© 2026 SKAI Worldwide Co., Ltd.

Reading a Cypher EXPLAIN

```
EXPLAIN (COSTS OFF) MATCH (a:account {handle: 'user42'})-[r1:follows]->(b:account)-
[r2:follows]->(c:account) RETURN c.handle;
```

```
QUERY PLAN

------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------
----------------
 Hash Join
   Hash Cond: (r2."end" = c.id)
   ->  Nested Loop
         ->  Nested Loop
               ->  Index Scan using account_handle_idx on account a
                     Index Cond: (properties.'handle' = '"user42"'::jsonb)
               ->  Bitmap Heap Scan on follows r1
                     Recheck Cond: ((a.id = start) AND (start >= '3.0'::graphid) AND (start <=
'3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710655':
:graphid))
                     ->  Bitmap Index Scan on follows_start_idx
                           Index Cond: ((start = a.id) AND (start >= '3.0'::graphid) AND (star
t <= '3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710
655'::graphid))
         ->  Index Scan using follows_start_idx on follows r2
               Index Cond: ((start = r1."end") AND (start >= '3.0'::graphid) AND (start <= '3.
281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710655'::gr
aphid))
               Filter: (r1.id <> id)
   ->  Hash
         ->  Seq Scan on account c
```

Four things in there are AgensGraph-specific and everything else is a plan you have read before.

A property read prints as `properties.'key'`. That is the Cypher parser's spelling of a property access, and it is not SQL you can type. In an `Index Cond` it is being matched against an expression index. In a `Filter` it is being evaluated per row. Which of those you get is **the subject of the indexes page**.

A `graphid` range test is a label test. `start >= '3.0' AND start <= '3.281474976710655'` is "the vertex at this end of the edge is an `account`". A label id lives in the high bits of a `graphid`, so one label's ids form a contiguous range and the test is answerable from the edge's own columns. See **Identity: the graphid**.

`Filter: (r1.id <> id)` on the second edge is the uniqueness rule: no path may use one edge twice.

A pattern element you did not name gets a synthetic alias, printed as `"<0000000002>"`. The counter behind it advances on every statement, so a plan containing one differs from run to run. Name every element of a pattern whose plan you want to compare against a baseline.

`EXPLAIN (ANALYZE)` works and is the only way to see the estimates against reality:

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, SUMMARY OFF, BUFFERS OFF)
MATCH (a:account {handle: 'user42'})-[r1:follows]->(b:account) RETURN count(b);
```

```
QUERY PLAN

------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
-----------------
 Aggregate (actual rows=1.00 loops=1)
   ->  Nested Loop (actual rows=6.00 loops=1)
         ->  Nested Loop (actual rows=6.00 loops=1)
               ->  Index Scan using account_handle_idx on account a (actual rows=1.00 loops=1)
                     Index Cond: (properties.'handle' = '"user42"'::jsonb)
                     Index Searches: 1
               ->  Bitmap Heap Scan on follows r1 (actual rows=6.00 loops=1)
                     Recheck Cond: ((a.id = start) AND (start >= '3.0'::graphid) AND (start <=
'3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710655':
:graphid))
                     Heap Blocks: exact=1
                     ->  Bitmap Index Scan on follows_start_idx (actual rows=6.00 loops=1)
                           Index Cond: ((start = a.id) AND (start >= '3.0'::graphid) AND (star
t <= '3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710
655'::graphid))
                           Index Searches: 6
         ->  Index Only Scan using account_pkey on account b (actual rows=1.00 loops=6)
               Index Cond: (id = r1."end")
               Heap Fetches: 6
               Index Searches: 6
```

NOTE

Read a variable-length or shortest-path plan in the default text format. `Dijkstra` is the one of the three whose plan also reads correctly through `EXPLAIN (FORMAT JSON)`.

Plan nodes for a traversal

A fixed-length pattern needs no special node: it is scans and joins, as above. The nodes below appear for the things SQL has no operator for.

`Graph VLE` is a variable-length edge, and it names the edge label and the range:

```
SET graph_path = moviekb;
EXPLAIN (COSTS OFF) MATCH (a:movie {title: 'The Matrix'})-[r:related_to*1..3]->(b:movie) RETUR
N b.title;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Nested Loop
   ->  Seq Scan on movie a
         Filter: (properties.'title' = '"The Matrix"'::jsonb)
   ->  Hash Join
         Hash Cond: (r."end" = b.id)
         ->  Subquery Scan on r
               ->  Graph VLE on related_to [1..3]
                     ->  Result
         ->  Hash
               ->  Seq Scan on movie b
```

`on related_to` is the line to read. An untyped variable-length edge becomes `Graph VLE on ag_edge`, which walks every edge label in the graph at every hop. That is the single biggest avoidable cost in a path query. **Paths** has the comparison.

`Shortestpath VLE [min..max]` is a `shortestpath` or `allshortestpaths`, sitting above two `Hash2Side` nodes: the two frontiers of a bidirectional search, one growing from each end.

© 2026 SKAI Worldwide Co., Ltd.

`Dijkstra` is a weighted search. Its child is the scan it expands through, so an index on the edge label's `start` column is what it walks.

`Graph Create`, `Graph Set` and `Graph Delete` are the write nodes:

```
SET graph_path = perflab;
EXPLAIN (COSTS OFF) MATCH (a:account {handle: 'user42'}) SET a.seen = 1;
```

```
                        QUERY PLAN
------------------------------------------------------------
 Graph Set
   ->  Index Scan using account_handle_idx on account a
         Index Cond: (properties.'handle' = '"user42"'::jsonb)
```

A write node marked **`eager`** has to finish before anything above it runs, because something above it reads what the write changed:

```
EXPLAIN (COSTS OFF)
MATCH (a:account {handle: 'user42'}) SET a.seen = 1
WITH a MATCH (a)-[r:follows]->(b:account) RETURN count(b);
```

```
                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------
------------------------------------------------
 Aggregate
   ->  Nested Loop
         ->  Nested Loop
               ->  Subquery Scan on _agens_default_s
                     ->  Graph Set eager
                           ->  Index Scan using account_handle_idx on account a
                                 Index Cond: (properties.'handle' = '"user42"'::jsonb)
               ->  Index Only Scan using follows_start_idx on follows r
                     Index Cond: ((start = (_agens_default_s.a).id) AND ("end" >= '3.0'::graph
id) AND ("end" <= '3.281474976710655'::graphid))
         ->  Index Only Scan using account_pkey on account b
               Index Cond: (id = r."end")
```

Eagerness is what makes **a write visible to a later read in the same statement.** It costs a materialisation of the write's output, bounded by `eager_mem`. `MERGE` is always eager, and so is any `SET`, `REMOVE` or `DELETE` that is not the last clause in its statement.

The names beginning `_agens_default_s` are the generated aliases for a pipeline's stages, roughly one per `WITH`. They are stable within a plan and are not something to tune.

Statistics on edge labels

This is the first thing to check on any new graph, and it is worth more than any index.

The cost of a traversal is a product of degrees: how many edges leave the vertices you started from, how many leave those you reached, and so on. Statistics on an edge label's `start` and `end` columns describe those degrees. AgensGraph sets both columns' statistics target to the maximum, 1,000, when the label is created:

```
SELECT attname, attstattarget FROM pg_attribute
WHERE attrelid = 'perflab.follows'::regclass AND attnum > 0 ORDER BY attnum;
```


© 2026 SKAI Worldwide Co., Ltd.

```
  attname   | attstattarget
------------+---------------
 id         |
 start      |          1000
 end        |          1000
 properties |
```

A target is not a sample. Until something analyses the label there is nothing there:

```
SELECT count(*) AS stats_rows FROM pg_stats WHERE schemaname = 'perflab' AND tablename =
'follows';
```

```
 stats_rows
------------
          0
```

And the estimates are guesses. Here is a two-hop count on a freshly loaded graph:

```
EXPLAIN MATCH (a:account {handle: 'user42'})-[r1:follows]->(b:account)-[r2:follows]-
>(c:account) RETURN count(c);
```

```


QUERY PLAN

------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------
----------------------
 Aggregate  (cost=10088.00..10088.01 rows=1 width=32)
   ->  Hash Join  (cost=677.81..9622.10 rows=186358 width=8)
         Hash Cond: (r2."end" = c.id)
         ->  Nested Loop  (cost=20.81..8475.81 rows=186358 width=24)
               ->  Nested Loop  (cost=20.39..779.08 rows=433 width=16)
                     ->  Index Scan using account_handle_idx on account a  (cost=0.29..8.30 ro
ws=1 width=8)
                           Index Cond: (properties.'handle' = '"user42"'::jsonb)
                     ->  Bitmap Heap Scan on follows r1  (cost=20.10..766.44 rows=433
width=24)
                           Recheck Cond: ((a.id = start) AND (start >= '3.0'::graphid) AND (st
art <= '3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.2814749767
10655'::graphid))
                           ->  Bitmap Index Scan on follows_start_idx  (cost=0.00..20.00
rows=433 width=0)
                                 Index Cond: ((start = a.id) AND (start >= '3.0'::graphid) AND
(start <= '3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.2814749
76710655'::graphid))
               ->  Index Scan using follows_start_idx on follows r2  (cost=0.42..13.48 rows=43
0 width=24)
                     Index Cond: ((start = r1."end") AND (start >= '3.0'::graphid) AND (start
<= '3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.28147497671065
5'::graphid))
                     Filter: (r1.id <> id)
         ->  Hash  (cost=407.00..407.00 rows=20000 width=8)
               ->  Seq Scan on account c  (cost=0.00..407.00 rows=20000 width=8)
```

433 edges out of one account, 186,358 two-hop pairs, and a hash over all 20,000 accounts to resolve the far endpoint. Now analyse the edge label:

© 2026 SKAI Worldwide Co., Ltd.

```
ANALYZE perflab.follows;
```

```
EXPLAIN MATCH (a:account {handle: 'user42'})-[r1:follows]->(b:account)-[r2:follows]-
>(c:account) RETURN count(c);
```

```
QUERY PLAN

-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
----------------------
 Aggregate  (cost=51.17..51.18 rows=1 width=32)
   ->  Nested Loop  (cost=5.52..51.08 rows=36 width=8)
         ->  Nested Loop  (cost=5.23..39.78 rows=36 width=24)
               ->  Nested Loop  (cost=4.81..35.54 rows=6 width=16)
                     ->  Index Scan using account_handle_idx on account a  (cost=0.29..8.30 ro
ws=1 width=8)
                           Index Cond: (properties.'handle' = '"user42"'::jsonb)
                     ->  Bitmap Heap Scan on follows r1  (cost=4.52..27.18 rows=6 width=24)
                           Recheck Cond: ((a.id = start) AND (start >= '3.0'::graphid) AND (st
art <= '3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.2814749767
10655'::graphid))
                           ->  Bitmap Index Scan on follows_start_idx  (cost=0.00..4.52 rows=6
width=0)
                                 Index Cond: ((start = a.id) AND (start >= '3.0'::graphid) AND
(start <= '3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.2814749
76710655'::graphid))
               ->  Index Scan using follows_start_idx on follows r2  (cost=0.42..0.65 rows=6 w
idth=24)
                     Index Cond: ((start = r1."end") AND (start >= '3.0'::graphid) AND (start
<= '3.281474976710655'::graphid) AND ("end" >= '3.0'::graphid) AND ("end" <= '3.28147497671065
5'::graphid))
                     Filter: (r1.id <> id)
         ->  Index Only Scan using account_pkey on account c  (cost=0.29..0.31 rows=1 width=8)
               Index Cond: (id = r2."end")
```

6 edges, 36 pairs, an index-only scan for the far endpoint, and an estimated cost of 51 against 10,088. And 36 is exactly right:

```
MATCH (a:account {handle: 'user42'})-[r1:follows]->(b:account)-[r2:follows]->(c:account)
RETURN count(c) AS pairs;
```

```
 pairs
-------
    36
```

The rule that follows: **after any bulk load, `ANALYZE` the edge labels, not just the vertex labels.** Autovacuum gets there eventually, but “eventually” is after the first users have run the query. A graph whose degrees change shape, when a hub vertex arrives, is worth re-analysing explicitly rather than waiting for the row-count threshold.

The settings that change plan shape

Five settings affect how a graph query is planned. These are their real values, read from `pg_settings`:

© 2026 SKAI Worldwide Co., Ltd.

Setting	Default	Scope	What it does
`enable_eager`	`on`	session	When off, a statement that needs an eager plan is refused instead of planned.
`eager_mem`	4 MB	session	Memory an eager node may use before spilling.
`enable_graph_endpoint_elision`	`on`	session	Removes the scan for a pattern endpoint whose value the query never reads.
`enable_property_promotion`	`on`	session	Resolves a Cypher property read to a promoted typed column when one exists.
`auto_gather_graphmeta`	`off`	superuser	Maintains `ag_graphmeta` edge connectivity, which lets the planner prune unreachable labels from a scan.

NOTE

`postgresql.conf.sample` states the wrong default for `enable_eager` and `enable_multiple_update`. Both are on. Read `pg_settings`, not the sample file.

None of these is a knob to turn in production. Four are on already, and the fifth is a decision about maintaining a catalog. Know them for diagnosis: turning one off tells you whether it is responsible for a plan you did not expect.

Endpoint elision

An endpoint whose value the query never reads is not scanned at all. `b` here is bound and labelled, and never projected:

```
EXPLAIN (COSTS OFF) MATCH (a:account)-[r:follows]->(b:account) RETURN a.handle;
```

```
                                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------
---------------------------------------------------------------------
 Hash Join
   Hash Cond: (r.start = a.id)
   ->  Seq Scan on follows r
         Filter: (("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710655'::graphid) AND (s
tart >= '3.0'::graphid) AND (start <= '3.281474976710655'::graphid))
   ->  Hash
         ->  Seq Scan on account a
```

One join, and `(b:account)` survives only as a range test on the edge's `"end"` column. Switch the elision off and the second scan and join come back:

```
SET enable_graph_endpoint_elision = off;
EXPLAIN (COSTS OFF) MATCH (a:account)-[r:follows]->(b:account) RETURN a.handle;
RESET enable_graph_endpoint_elision;
```

```
                                                                               QUERY PLAN
-----------------------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
 Hash Join
   Hash Cond: (r."end" = b.id)
   ->  Hash Join
         Hash Cond: (r.start = a.id)
         ->  Seq Scan on follows r
               Filter: ((start >= '3.0'::graphid) AND (start <= '3.281474976710655'::graphid)
AND ("end" >= '3.0'::graphid) AND ("end" <= '3.281474976710655'::graphid))
         ->  Hash
               ->  Seq Scan on account a
   ->  Hash
         ->  Seq Scan on account b
```

The lesson is not about the setting. Projecting an endpoint you do not need costs a scan and a join, every time. So keep a `RETURN` to what the caller reads.

Scan pruning from ag_graphmeta

`ag_graphmeta` holds one row per start-label, edge-label, end-label triple that actually occurs in the graph, with how many edges make it up. With that, an unlabelled pattern endpoint can be narrowed to the labels the edge is known to reach. It is maintained only when `auto_gather_graphmeta` is on, so by default an unlabelled endpoint scans the whole `ag_vertex` family:

```
EXPLAIN (COSTS OFF) MATCH (n)-[r:follows]->(m) RETURN n, m;
```

```
                    QUERY PLAN
--------------------------------------------------
 Hash Join
   Hash Cond: (m.id = r."end")
   ->  Append
         ->  Seq Scan on ag_vertex m_1
         ->  Seq Scan on account m_2
   ->  Hash
         ->  Hash Join
               Hash Cond: (n.id = r.start)
               ->  Append
                     ->  Seq Scan on ag_vertex n_1
                     ->  Seq Scan on account n_2
               ->  Hash
                     ->  Seq Scan on follows r
```

Turn gathering on and both `Append`s collapse. Gathering builds a baseline for every graph in the database, so do it deliberately:

```
SET auto_gather_graphmeta = on;
EXPLAIN (COSTS OFF) MATCH (n)-[r:follows]->(m) RETURN n, m;
RESET auto_gather_graphmeta;
```

```
               QUERY PLAN
-----------------------------------------
 Hash Join
   Hash Cond: (r."end" = m.id)
   ->  Hash Join
         Hash Cond: (r.start = n.id)
         ->  Seq Scan on follows r
         ->  Hash
               ->  Seq Scan on account n
```


© 2026 SKAI Worldwide Co., Ltd.

```
   ->  Hash
         ->  Seq Scan on account m
```

`follows` only ever runs `account` → `account`, so the two unlabelled endpoints can only be accounts. On a graph with many labels this is the difference between one scan per endpoint and one per label. It costs maintenance on every edge write, plus a rescan whenever the baseline is invalidated.

Two labels are all this example has, so the saving here is small. The shape of it is the point. `pg_catalog.ag_graphmeta` holds the counts and `ag_graph.graphmeta_valid` gates their use. See **Graph metadata**.

The plainest alternative is to name the label in the pattern. `MATCH (n:account)-[r:follows]->(m:account)` needs no catalog at all.

Forbidding an eager plan

`enable_eager = off` does not produce a different plan. It refuses to produce one:

```
SET enable_eager = off;
EXPLAIN (COSTS OFF)
MATCH (a:account {handle: 'user42'}) SET a.seen = 1
WITH a MATCH (a)-[r:follows]->(b:account) RETURN count(b);
RESET enable_eager;
```

```
ERROR:  eagerness plan is not allowed.
```

Use it to find out which statement in a workload depends on eagerness. Do not run with it off. Eagerness is a correctness requirement for such a statement rather than a cost choice, so there is no cheaper plan to fall back to and the server raises an error instead.

Resolution to a promoted column

`enable_property_promotion = off` sends a property read back to the `jsonb` bag even where a typed column exists. The plan changes and the answers do not, because the value is in both places. It is the way to establish whether a promotion is behind a plan you did not expect, without dropping the column. **Turning resolution off** has the before and after.

What to do, in order

1. `ANALYZE` **every label the query touches, edge labels included.** Nothing else on this list matters until this is done.
2. **Check the anchor.** A traversal wants one selective, indexed starting point. `EXPLAIN` should show an `Index Cond`, not a `Filter`, on the property that picks it — **Indexes and constraints** says which predicates can.
3. **Name the edge label in every variable-length pattern.** `Graph VLE on ag_edge` in a plan points at the query rather than at the planner.
4. **Bound the length** of a variable-length pattern.
5. **Project only what the caller reads**, so unused endpoints elide.
6. **Consider promoting a property** you filter, join or order on constantly — **Promoting a property to a column**.
7. **Then look at resources.** Configuration.

Clean up

```
DROP GRAPH perflab CASCADE;
```


© 2026 SKAI Worldwide Co., Ltd.

```
DROP TABLE seed_account;
DROP TABLE seed_follows;
```

Where to go next

- **Indexes and constraints** — which predicates an index can serve.
- **Paths** — the traversal-specific version of this page.
- **Promoting a property to a column** — when the read itself is the cost.
- **Configuration** — memory, I/O and autovacuum.

© 2026 SKAI Worldwide Co., Ltd.

Reference

The formal definition of the language: its lexical rules, its types, every clause, every data-definition statement, every function, and every server setting AgensGraph adds. For the narrative introduction to the same material, see Querying.

Syntax

The lexical layer every statement obeys, whichever language it is written in: the notation this reference uses for syntax, and the rules for identifiers, literals, operators, comments and null.

Cypher, GQL and SQL share one parser, so these rules are the same in all three. Where a rule differs between them, the difference is called out.

Syntax conventions

Every synopsis in this reference is written the same way:

```
MATCH pattern [, pattern ] ... [ WHERE condition ]
```

Nothing else in a synopsis is notation. A comma, a parenthesis or an arrow is part of the statement, and you type it as shown.

bold	A keyword. Type it as it appears; case does not matter.
italic	A placeholder you supply. Each one is defined under Parameters on the same page.
`[ ]`	The enclosed part is optional.
`{ ... | ... }`	Braces around alternatives separated by a vertical bar: choose exactly one.
`...`	The item before it may be repeated.
`[, ... ]`	The item before it may be repeated, separated by commas.
Two or more synopsis lines	Alternative forms of the same statement. Use one of them.
`where` *placeholder* `is one of:`	Expands a placeholder that is too large to sit inline, on the lines that follow.

© 2026 SKAI Worldwide Co., Ltd.

Brackets and braces are never literal. Some statements really do contain a bracket: a list literal, a map, an edge pattern. For those, the synopsis says so in prose, or shows the bracket in a different position, as in `-[` variable `]->`.

Identifiers

An identifier names a graph, a label, a variable, a property key, a column, an index or a constraint.

Case folding

An unquoted identifier is folded to **lower case**, exactly as PostgreSQL folds an unquoted SQL identifier. So `MOVIE`, `Movie` and `movie` all name the same label:

```
MATCH (n:MOVIE {title: 'The Matrix'}) RETURN n.title AS title;
```

```
    title
--------------
 "The Matrix"
```

A double-quoted identifier is taken as written. `"MOVIE"` is a different name from `movie`. No label is called `MOVIE`, so the pattern matches nothing rather than failing:

```
MATCH (n:"MOVIE" {title: 'The Matrix'}) RETURN n.title AS title;
```

```
 title
-------
(0 rows)
```

Quoting works for variables too, and a quoted variable has to be quoted at every mention:

```
MATCH ("Film":movie {title: 'The Matrix'}) RETURN "Film".title AS title;
```

```
    title
--------------
 "The Matrix"
```

Property keys fold the same way. `{MixedKey: 1}` stores the key `mixedkey`; `{"MixedKey": 1}` stores `MixedKey`. The two are different keys and a vertex can carry both, so quote a key consistently or not at all.

Backticks are not a quoting character. `` `movie` `` is a syntax error, not a quoted identifier:

```
MATCH (n:`movie`) RETURN n.title;
```

```
ERROR:  syntax error at or near "`"
LINE 1: MATCH (n:`movie`) RETURN n.title;
                 ^
```


© 2026 SKAI Worldwide Co., Ltd.

Characters

An unquoted identifier starts with a letter or an underscore and continues with letters, digits, underscores and dollar signs. “Letter” includes non-ASCII letters, so `café` is a valid identifier. It may not start with a digit. `1abc` is read as a number with trailing junk. A double-quoted identifier may contain any character except a null byte, and a double quote inside one is written twice (`"say ""hi"""`).

The length limit is PostgreSQL’s: 63 bytes, and a longer name is truncated to that. The limit counts bytes, not characters, so a name of multi-byte letters runs out sooner.

Reserved words

Some keywords are **reserved**: the grammar reads the word as a keyword and not as a name. `both` , `let` , `finish` , `insert` , `offset` and `xor` are all reserved. Some are inherited from PostgreSQL, some added for a graph clause. Which positions each one still works in varies, so **Keywords** carries the position-by-position table. A pattern variable is a position none of them survives:

```
MATCH (both:movie {title: 'The Matrix'}) RETURN both.title;
```

```
ERROR:  syntax error at or near "both"
LINE 1: MATCH (both:movie {title: 'The Matrix'}) RETURN both.title;
               ^
```

Quoting lifts the restriction, because a quoted identifier is never read as a keyword:

```
MATCH ("both":movie {title: 'The Matrix'}) RETURN "both".title AS title;
```

```
    title
--------------
 "The Matrix"
```

A reserved word is still allowed as a `RETURN` alias and as a property key, since neither position can be confused with a keyword:

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.title AS both;
```

```
     both
--------------
 "The Matrix"
```

Literals

Strings

A string literal is enclosed in **single** quotes. Double quotes make an identifier, not a string, so `"plain"` is a variable reference and fails if no such variable is in scope.

A single quote inside a literal is written twice. Backslash escapes follow JSON’s rules, because a Cypher string becomes a `jsonb` string:

© 2026 SKAI Worldwide Co., Ltd.

Escape	Meaning
`''`	A single quote.
`\"`	A double quote.
`\\`	A backslash.
`\/`	A forward slash.
`\b` `\f` `\n` `\r` `\t`	Backspace, form feed, line feed, carriage return, tab.
`\u` followed by four hex digits	The character with that code point.

Any other backslash sequence is rejected: `'a\qb'` raises `Escape sequence "\q" is invalid`.

```
RETURN 'plain', 'it''s', 'a\tb', 'a\\tb', 'say \" hi';
```

```
 ?column? | ?column? | ?column? | ?column? |  ?column?
----------+----------+----------+----------+-------------
 "plain"  | "it's"   | "a\tb"   | "a\\tb"  | "say \" hi"
```

The output is the `jsonb` rendering of each value, which re-escapes what it prints: the third column holds three characters, `a`, a tab and `b`.

Numbers

A number literal is written as in SQL, and PostgreSQL's non-decimal and separated forms are accepted: hexadecimal `0x`, octal `0o`, binary `0b`, and underscores as digit separators.

```
RETURN 1, -1, 1.5, 1e3, 1E-3, 0x1F, 0o17, 0b101, 1_000;
```

```
 ?column? | ?column? | ?column? | ?column? | ?column? | ?column? | ?column? | ?column? | ?
column?
----------+----------+----------+----------+----------+----------+----------+----------
+----------
 1        | -1       | 1.5      | 1000     | 0.001    | 31       | 15       | 5        | 1000
```

A number written without a decimal point is a JSON integer, and dividing two of them truncates: `7 / 2` is `3`. Write `7.0 / 2` for `3.5`.

Booleans, null, lists and maps

`true`, `false` and `null` are keywords, in any case. A list is written in square brackets and a map in braces. Both nest, and both may mix value types, because both become `jsonb`.

```
RETURN true, false, null, [1, 'two', true, null], {a: 1, b: [2], c: {d: 3}};
```


© 2026 SKAI Worldwide Co., Ltd.

```
 ?column? | ?column? | ?column? |        ?column?        |             ?column?
----------+----------+----------+------------------------+-------------------------------------
 t        | f        |          | [1, "two", true, null] | {"a": 1, "b": [2], "c": {"d": 3}}
```

A map key is an identifier and folds like one. `[]` and `{}` are the empty list and the empty map. See **Data types** for how a literal relates to the stored `jsonb` value.

Operators

Precedence

Operators bind in this order, loosest first. Two operators on the same row have equal precedence and associate as the row says.

Precedence	Operators	Associativity
1 (loosest)	`UNION` `EXCEPT`	left
2	`INTERSECT`	left
3	`OR` `XOR`	left
4	`AND`	left
5	`NOT`	right
6	`IS NULL` `IS NOT NULL` `IS [NOT] UNKNOWN`	none
7	`<` `<=` `>` `>=` `=` `<>` `!=`	none
8	`IN` `BETWEEN`	none
9	any other operator, including `||` and the `jsonb` operators	left
10	`+` `-` (binary)	left
11	`*` `/` `%`	left
12	`^`	left
13	`-` (unary)	right
14	`STARTS WITH` `ENDS WITH` `CONTAINS` `=~`	none
15	`[` `]` (subscript), `(` `)` (call)	left
16	`::`	left

© 2026 SKAI Worldwide Co., Ltd.

Precedence	Operators	Associativity
17 (tightest)	`.`	left

The table lists the operators a Cypher expression can contain. SQL adds a few more at levels of its own: `AT TIME ZONE`, `COLLATE`, `LIKE`, `ILIKE` and `SIMILAR TO`. Cypher's expression grammar accepts none of them.

Three things in the table differ from a common expectation.

`^` is **left**-associative and binds looser than unary minus, which is PostgreSQL's rule rather than arithmetic's. So `2 ^ 3 ^ 2` is `(2 ^ 3) ^ 2` and `-2 ^ 2` is `(-2) ^ 2`:

```
RETURN 2 ^ 3 ^ 2 AS pow, -2 ^ 2 AS neg;
```

```
 pow | neg
-----+-----
 64  | 4
```

The string-matching operators bind **tighter** than everything except a subscript, a call, a cast and a property read. They bind tighter than concatenation and tighter than arithmetic. So the right-hand operand of `||` below is the whole `STARTS WITH` test, not just `'c'`:

```
RETURN 'ab' || 'c' STARTS WITH 'a';
```

```
 ?column?
-----------
 "abfalse"
```

Parenthesise when you mean the other reading. `('ab' || 'c') STARTS WITH 'a'` is `true`.

Comparisons do not chain. `1 < 2 < 3` is a syntax error rather than two comparisons.

Logical operators

`AND`, `OR`, `NOT` and `XOR` take booleans and return a boolean, over three values. See **Null and three-valued logic** for the truth tables. `XOR` is AgensGraph's addition to SQL's set, and it sits at `OR`'s precedence.

Comparison and predicates

`=`, `<>` (or `!=`), `<`, `<=`, `>` and `>=` compare two values. Comparing values of different `jsonb` types does not fail. It orders them by `jsonb`'s own type order.

`IN` tests membership of a list. `IS NULL` and `IS NOT NULL` test any value for null without comparing it. `IS UNKNOWN` and `IS NOT UNKNOWN` are the standard's spelling of that test on a boolean, and are accepted here:

```
RETURN 2 IN [1, 2, 3] AS present, null IS NULL AS nul, null IS UNKNOWN AS unk;
```

```
 present | nul | unk
---------+-----+-----
 t       | t   | t
```


© 2026 SKAI Worldwide Co., Ltd.

String matching

Four operators match strings. `LIKE`, `ILIKE` and `SIMILAR TO` are **SQL only**. They are not part of Cypher's expression grammar, and they are a syntax error there. PostgreSQL's regular expression operators (`~`, `~*`, `!~`, `!~*`) do work in Cypher.

Operator	Test
string `STARTS WITH` prefix	The string begins with the prefix.
string `ENDS WITH` suffix	The string ends with the suffix.
string `CONTAINS` substring	The substring occurs anywhere in the string.
string `=~` pattern	The whole string matches the POSIX regular expression.

```
RETURN 'Matrix' STARTS WITH 'Mat' AS sw, 'Matrix' ENDS WITH 'rix' AS ew,
       'Matrix' CONTAINS 'atr' AS c, 'Matrix' =~ '^M.*x$' AS re;
```

```
 sw | ew | c | re
----+----+---+----
 t  | t  | t | t
```

None of the four can be answered from an index. A predicate that has to be sargable needs `=` or a range. **Data types** covers the index-visible forms of a property.

Arithmetic

`+`, `-`, `*`, `/`, `%` and `^` work on numbers, and unary `+` and `-` negate. On a property they run against `jsonb` rather than a SQL numeric type, which gives `+` two extra jobs: it concatenates two strings, and it appends to a list.

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p.born + 1 AS plus, p.born % 7 AS mod, -p.born AS neg,
       p.name + '!' AS concat, [1, 2] + 3 AS appended;
```

```
 plus | mod |  neg  |     concat      | appended
------+-----+-------+-----------------+-----------
 1965 | 4   | -1964 | "Keanu Reeves!" | [1, 2, 3]
```

`+` on two maps is refused, with `invalid expression: {"a": 1} + {"b": 2}`. Merge maps with the `jsonb` concatenation operator instead:

```
RETURN {a: 1} || {b: 2} AS merged;
```


© 2026 SKAI Worldwide Co., Ltd.

```
      merged
------------------
 {"a": 1, "b": 2}
```

To merge into an element rather than into a value, use `SET n +=` map.

jsonb operators

A property is `jsonb`, so PostgreSQL's `jsonb` operators apply to one directly. These are the ones worth knowing:

Operator	Result
json `->` key / index	The member as `jsonb`.
json `->>` key / index	The member as `text`.
json `#>` path	The value at a text-array path, as `jsonb`.
json `#>>` path	The value at a text-array path, as `text`.
json `?` key	Whether the key is present.
json `@>` json	Whether the left value contains the right.
json `<@` json	Whether the left value is contained in the right.
json `||` json	Concatenation: two objects merge, other combinations make an array.

```
MATCH (m:movie {title: 'The Matrix'})
RETURN properties(m) -> 'title' AS as_json, properties(m) ->> 'title' AS as_text,
       properties(m) ? 'genre' AS has_genre, properties(m) @> '{"genre": "Action"}' AS is_acti
on;
```

```
   as_json    |   as_text    | has_genre | is_action
--------------+--------------+-----------+-----------
 "The Matrix" | "The Matrix" | t         | t
```

`->>` returned `text`, but a Cypher `RETURN` boxes a scalar column back to `jsonb`, so the quotes are still there. To get a plain string out, wrap the query in SQL. See **Data types**.

Two rules govern the rest of PostgreSQL's `jsonb` operator set inside Cypher, and both follow from Cypher owning the arithmetic spellings.

`-` between two `jsonb` values is subtraction, so `jsonb`'s key-deletion operator is not reachable from Cypher: `properties(m) - 'plot'` raises `invalid expression`. Delete a key with a SQL wrapper, or with `REMOVE`.

A bare string literal beside a native `jsonb` operator is parsed as a whole JSON document rather than as a JSON string. So `m.title || '!'` fails on `Token "!" is invalid`. Give the operator a typed operand, as in `m.title || to_jsonb('!'::text)`. Or use `+`, which takes the Cypher reading.

© 2026 SKAI Worldwide Co., Ltd.

Comments

A comment runs from `--` to the end of the line, or from `/*` to a matching `*/`. Block comments nest, so a commented-out region containing a comment stays balanced. `//` is not a comment and is a syntax error.

```
MATCH (m:movie {title: 'The Matrix'})   -- one film
/* a block comment
   over two lines */
RETURN m.title AS title;
```

```
    title
--------------
 "The Matrix"
```

Null and three-valued logic

`null` means unknown. It is not a value you can compare with, and it is not a property value. Assigning null to a property removes it, unless **`allow_null_properties`** is on.

Truth tables

`AND`, `OR` and `XOR` extend to three values as follows. The table is the output of running every combination:

```
SELECT a #>> '{}' AS a, b #>> '{}' AS b,
       andv::text AS "a AND b", orv::text AS "a OR b", xorv::text AS "a XOR b"
FROM (UNWIND [true, false, null] AS a
      UNWIND [true, false, null] AS b
      RETURN a, b, a AND b AS andv, a OR b AS orv, a XOR b AS xorv) q;
```

```
   a   |   b   | a AND b | a OR b | a XOR b
-------+-------+---------+--------+---------
 true  | true  | true    | true   | false
 true  | false | false   | true   | true
 true  | null  | null    | true   | null
 false | true  | false   | true   | true
 false | false | false   | false  | false
 false | null  | false   | null   | null
 null  | true  | null    | true   | null
 null  | false | false   | null   | null
 null  | null  | null    | null   | null
```

`NOT` maps true to false, false to true, and null to null:

```
SELECT a #>> '{}' AS a, notv::text AS "NOT a"
FROM (UNWIND [true, false, null] AS a RETURN a, NOT a AS notv) q;
```

```
   a   | NOT a
-------+-------
 true  | false
 false | true
 null  | null
```

So `AND` is false as soon as one side is false, and `OR` is true as soon as one side is true, whatever the other side is. Everything else involving null is null.

© 2026 SKAI Worldwide Co., Ltd.

`IN` follows from the same rule: a value not found in a list that contains null might still have been there, so the answer is unknown rather than false.

```
SELECT expr, res::text AS result FROM (
  RETURN '2 IN [1, 2, 3]' AS expr, 2 IN [1,2,3] AS res
  UNION ALL RETURN '2 IN [1, null, 3]', 2 IN [1,null,3]
  UNION ALL RETURN '2 IN [1, 2, null]', 2 IN [1,2,null]
  UNION ALL RETURN '2 IN [1]', 2 IN [1]
  UNION ALL RETURN '2 IN []', 2 IN []
  UNION ALL RETURN 'null IN [1, 2, 3]', null IN [1,2,3]
  UNION ALL RETURN 'null IN [1, null, 3]', null IN [1,null,3]
  UNION ALL RETURN 'null IN []', null IN []
) q;
```

```
          expr          | result
------------------------+--------
 "2 IN [1, 2, 3]"       | true
 "2 IN [1, null, 3]"    | null
 "2 IN [1, 2, null]"    | true
 "2 IN [1]"             | false
 "2 IN []"              | false
 "null IN [1, 2, 3]"    | null
 "null IN [1, null, 3]" | null
 "null IN []"           | false
```

An empty list is the exception: nothing is in it, so the answer is false even for null.

Expressions that yield null

These are the expressions that produce null rather than an error or a value:

Expression	Why
`n.no_such_property`	The element has no such property.
`[][0]`, `head([])`	The list has no such element.
`1 + null`	Arithmetic with an unknown operand.
`1 < null`, `null = null`	Comparison with an unknown value.
`sin(null)`, `toUpper(null)`	A function called with null.

```
MATCH (m:movie {title: 'The Matrix'})
RETURN m.no_such_property AS missing_property, [][0] AS missing_element,
       head([]) AS head_of_empty, 1 + null AS arithmetic, 1 < null AS comparison,
       sin(null) AS function, null = null AS equality;
```

```
 missing_property | missing_element | head_of_empty | arithmetic | comparison | function |
equality
------------------+-----------------+---------------+------------+------------+----------
+----------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 null             | null            | null          | null       | null       | null     | nul
l
```

`IS NULL` exists because `null = null` is null. It is the only way to test for null.

A condition takes only a boolean

`WHERE`, `FILTER`, `CASE WHEN` and every other condition accept a boolean and nothing else. A condition that evaluates to a `jsonb` string, number, list or map is an error:

```
CREATE GRAPH demo;
SET graph_path = demo;
CREATE VLABEL t;
CREATE (:t {name: 'Jack'});
MATCH (n:t) WHERE n.name RETURN n.name;
```

```
ERROR:  cannot cast "Jack" (jsonb type string) to boolean
```

That error comes from evaluating the condition, so the value it names is whichever row the executor reached first. A condition that is not `jsonb` at all is rejected earlier, when the statement is parsed:

```
MATCH (n:t) WHERE 1 RETURN n.name;
```

```
ERROR:  argument of WHERE must be type boolean, not type integer
LINE 1: MATCH (n:t) WHERE 1 RETURN n.name;
                          ^
```

Write the test you mean. `WHERE n.name IS NOT NULL` asks whether the property is there:

```
MATCH (n:t) WHERE n.name IS NOT NULL RETURN n.name AS name;
```

```
  name
--------
 "Jack"
```

`WHERE n.name <> ''` asks whether it is non-empty, and `WHERE n.active` works when the property really holds a boolean, because a `jsonb` boolean casts to one.

Null is still accepted, and behaves as it does in SQL: a row whose condition is null is not returned, and nothing is raised. So `WHERE n.no_such_property` returns no rows.

> **NOTE**
>
> This is a deliberate compatibility break with earlier releases, which accepted a non-empty value as a condition. A query written that way fails rather than filtering on something it did not mean. **Cypher compatibility** lists what to change.

© 2026 SKAI Worldwide Co., Ltd.

Compatibility

openCypher

Three lexical differences will catch a reader who knows Cypher from elsewhere.

- **Identifiers fold to lower case, and backticks are not a quoting character.** Use double quotes. `` `My Label` `` is a syntax error.
- **A string literal takes single quotes only.** Double quotes make an identifier.
- **`//` is not a comment.** Use `--` or `/* */`.

`XOR`, `STARTS WITH`, `ENDS WITH`, `CONTAINS` and `=~` are all present and mean what they mean in openCypher.

ISO/IEC 39075

The notation in this reference follows the same conventions as the standard's syntax productions, but is not the standard's BNF.

`IS [NOT] UNKNOWN` is the standard's spelling of the null test on a boolean and is accepted alongside `IS [NOT] NULL`. So is `XOR`. Both fall under the standard's `<boolean value expression>` (ISO/IEC 39075:2024 §20.20). `UNKNOWN` used as a value is a syntax error. See **GQL conformance** for the rest.

See also

- **Data types** — `graphid`, `vertex`, `edge`, `graphpath` and `jsonb` properties
- **Server settings** — including `graph_path`, which every session needs
- **Keywords** — the full keyword list and each word's category
- **Cypher, GQL and SQL** — which language to reach for
- **PostgreSQL: SQL syntax** — the SQL half of the same lexer

Data types

AgensGraph adds five types to PostgreSQL's set: `graphid`, `vertex`, `edge`, `graphpath` and `rowid`. Everything else a graph stores is `jsonb`, in the `properties` column of a label. Every PostgreSQL type is still available and behaves as PostgreSQL documents it.

The five are defined in `pg_type` with fixed OIDs, so they are the same in every database and need no extension.

Type	Storage	What it holds
`graphid`	8 bytes, by value	The identity of one vertex or edge.
`vertex`	Composite, variable	An id, a property map, and the row's physical location.
`edge`	Composite, variable	An id, two endpoint ids, a property map, and the row's location.
`graphpath`	Composite, variable	An array of vertices and an array of edges.
`rowid`	10 bytes, by reference	A table OID and a row pointer. Internal.

© 2026 SKAI Worldwide Co., Ltd.

graphid

`graphid` is the identity of a vertex or an edge. An edge stores them to name its endpoints. It is a 64-bit value in two parts: the top 16 bits are the **label id**, the low 48 bits are the **local id**, a per-label sequence. It prints as labid `.` locid.

How a graph is stored explains why that layout matters to the planner. This section is the formal detail.

Two functions take the parts back apart:

```
SELECT id, graphid_labid(id) AS labid, graphid_locid(id) AS locid
FROM moviekb.person WHERE properties->>'name' = 'Keanu Reeves';
```

```
 id  | labid | locid
-----+-------+-------
 4.1 |     4 |     1
```

`graphid(` labid `,` locid `)` builds one from its parts. It is the same function a label's `id` column calls in its default expression.

Range

The label id is 16 bits and the local id 48, so the largest `graphid` is:

```
SELECT graphid(65535, 281474976710655) AS highest;
```

```
        highest
----------------------
 65535.281474976710655
```

Either part out of range is refused rather than truncated:

```
SELECT graphid(65536, 1);
```

```
ERROR:  labid out of range: 65536
```

```
SELECT graphid(3, 281474976710656);
```

```
ERROR:  locid out of range: 281474976710656
```

0 is reserved as no label, so a database holds at most 65,535 labels and a label at most $2^{48} - 1$ elements. The reservation is a convention rather than a check, and `graphid(0, 1)` builds `0.1`. But nothing ever assigns label id 0. **Limits** collects these alongside PostgreSQL's own.

Casts

There is exactly one cast into `graphid`, from `numeric`, and it is implicit. There is none from `integer`, so a whole number written without a decimal point does not convert:

```
SELECT 1::graphid;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  cannot cast type integer to graphid
LINE 1: SELECT 1::graphid;
               ^
```

Write the two parts. A numeric literal with a decimal point casts, and a string in labid `.` locid form parses:

```
SELECT 1.1::graphid AS from_numeric, '3.1'::graphid AS from_text;
```

```
 from_numeric | from_text
--------------+-----------
 1.1          | 3.1
```

The `numeric` cast reads the number the way the type prints. So `1.1` is label 1, element 1, not the number one point one. A numeric with no fractional digits has no local id and is rejected: `1::numeric::graphid` raises `invalid input syntax for type graphid: "1"`.

There is no cast out of `graphid`. Use `graphid_labid()` and `graphid_locid()` for the parts, or let it print.

vertex and edge

A `vertex` is a composite value of three fields: `id graphid`, `properties jsonb`, and a `tid` holding the row's physical location. A write uses that `tid` to find the row again. An `edge` has the same three fields plus `start graphid` and `end graphid`.

They print in a form that shows the label, the id and the properties:

```
MATCH (p:person {name: 'Keanu Reeves'}) RETURN p;
```

```
                         p
-----------------------------------------------------
 person[4.1]{"born": 1964, "name": "Keanu Reeves"}
```

An edge prints its endpoints between its id and its properties, start first:

```
MATCH (:person {name: 'Keanu Reeves'})-[r:acted_in]->(:movie {title: 'The Matrix'}) RETURN r;
```

```
                    r
-------------------------------------------
 acted_in[5.1][4.1,3.1]{"role": "Neo"}
```

So the shapes are `label[id]{properties}` for a vertex and `label[id][start,end]{properties}` for an edge. An element with no properties prints an empty map: `directed[6.8][4.8,3.5]{}`.

The `tid` field is not part of either printed form, and it is not something to depend on. It changes when the row is updated.

What you can do with them

`vertex` and `edge` each have the six comparison operators, and nothing else:

© 2026 SKAI Worldwide Co., Ltd.

Type	Operators
`graphid`	`=` `<>` `<` `<=` `>` `>=`
`vertex`	`=` `<>` `<` `<=` `>` `>=`
`edge`	`=` `<>` `<` `<=` `>` `>=`
`rowid`	`=` `<>` `<` `<=` `>` `>=`
`graphpath`	None

Comparison is on identity alone. `vertex_eq` and its siblings compare the `id` field and ignore properties entirely. Two vertices with identical properties and different ids are not equal:

```
CREATE GRAPH twins;
SET graph_path = twins;
CREATE VLABEL pair;
CREATE (:pair {k: 1}), (:pair {k: 1});
MATCH (a:pair), (b:pair) WHERE id(a) < id(b)
RETURN a, b, a = b AS equal, properties(a) = properties(b) AS same_properties;
```

```
         a          |          b          | equal | same_properties
--------------------+---------------------+-------+-----------------
 pair[3.1]{"k": 1} | pair[3.2]{"k": 1} | f     | t
```

Ordering follows from the same rule. `a < b` compares `graphid` values, so it is label order first and then insertion order within a label. The order is total and stable, which makes it a good tie-breaker in `ORDER BY`. It is not a meaningful ordering of your data.

```
MATCH (a:person {name: 'Keanu Reeves'}), (b:person {name: 'Carrie-Anne Moss'})
RETURN a = b AS equal, a < b AS less, id(a) AS id_a, id(b) AS id_b;
```

```
 equal | less | id_a | id_b
-------+------+------+------
 f     | t    | 4.1  | 4.3
```

To compare properties instead, compare the properties: `properties(a) = properties(b)`, or name the keys you mean.

Both types cast implicitly to `jsonb`. The cast keeps the property map and drops the identity: `vertex_to_jsonb` and `edge_to_jsonb` in `pg_cast`. So `m::jsonb` is `properties(m)`. There is no cast in the other direction.

graphpath

A `graphpath` is a composite of two arrays: `vertices vertex[]` and `edges edge[]`. A path of n edges holds n + 1 vertices, so a path bound to a single vertex has one vertex and no edges:

```
MATCH p = (a:person {name: 'Keanu Reeves'}) RETURN p;
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                                  p
-------------------------------------------------------------------------
 [person[4.1]{"born": 1964, "name": "Keanu Reeves"}]
```

It prints as a bracketed, comma-separated list of its members in walk order, alternating vertex, edge, vertex. The two arrays are interleaved rather than printed as two arrays. So a one-edge path prints three members, and a path of n edges prints 2n + 1.

Reach the members with `nodes()` and `edges()`, which hand back the two arrays. `length()` is the edge count:

```
MATCH p = (:person {name: 'Keanu Reeves'})-[:acted_in]->(:movie {title: 'The Matrix'})
RETURN length(p) AS hops, nodes(p)[0] AS first_vertex, edges(p)[0] AS only_edge;
```

```
 hops |                   first_vertex                    |              only_edge
------+---------------------------------------------------+
+--------------------------------------
 1    | person[4.1]{"born": 1964, "name": "Keanu Reeves"} | acted_in[5.1][4.1,3.1]{"role": "Ne
o"}
```

`graphpath` has **no operators of its own**, not equality and not ordering. In Cypher, comparing two paths is refused. With no operator to call, the expression falls back to `jsonb`, and a path is not a `jsonb` value:

```
MATCH p1 = (:person {name:'Keanu Reeves'})-[:acted_in]->(:movie {title:'The Matrix'}),
      p2 = (:person {name:'Keanu Reeves'})-[:acted_in]->(:movie {title:'The Matrix'})
RETURN p1 = p2;
```

```
ERROR:  graph object cannot be jsonb
LINE 3: RETURN p1 = p2;
               ^
```

In SQL it does compare, through PostgreSQL's generic operators on composite values, which walk the two arrays field by field:

```
SELECT a = b AS equal, a < b AS less
FROM (MATCH p = (:person {name:'Keanu Reeves'})-[:acted_in]->(:movie {title:'The Matrix'})
      RETURN p AS a) x,
     (MATCH p = (:person {name:'Keanu Reeves'})-[:acted_in]->(:movie {title:'The Matrix
Reloaded'})
      RETURN p AS b) y;
```

```
 equal | less
-------+------
 f     | t
```

That comparison includes the `tid` fields, so it is an identity comparison of the whole walk rather than a comparison of what the path means. Compare `nodes(p)` or the ids if you want something you can reason about.

`to_jsonb()` converts a path, in Cypher and in SQL alike. The result exposes the internal structure: both arrays, and the `tid` of every member. There is no `graphpath` to `jsonb` cast in `pg_cast`. So `p::jsonb` works in Cypher, which lowers it, and fails in SQL with `cannot cast type graphpath to jsonb`.

© 2026 SKAI Worldwide Co., Ltd.

rowid

`rowid` pairs a table OID with a row pointer. The shortest-path executor uses it internally to carry edge identities. It has the six comparison operators and a btree operator class, and it has **no output function**:

```
SELECT rowid('moviekb.person'::regclass, ctid) FROM moviekb.person LIMIT 1;
```

```
ERROR:  "rowid" type does not support output function
```

`rowid_tableoid()` and `rowid_ctid()` read the two halves, which is the only way to see a value:

```
SELECT rowid_tableoid(rowid('moviekb.person'::regclass, '(0,1)')) AS tableoid,
       rowid_ctid(rowid('moviekb.person'::regclass, '(0,1)')) AS ctid;
```

```
 tableoid | ctid
----------+-------
    16482 | (0,1)
```

The OID is whatever `moviekb.person` happens to be in that database, so it differs from installation to installation.

Nothing in Cypher produces a `rowid`, and there is no reason to use one in application code. It is documented here because it appears in `pg_type` and in `\dT`.

Arrays of graph types

`vertex[]`, `edge[]` and `graphpath[]` are ordinary PostgreSQL array types with their own output functions, so they print in the same bracketed style as their elements.

`collect()` over a vertex, an edge or a path returns one of these native arrays. `collect()` over anything else returns `jsonb`:

```
MATCH (:person {name: 'Keanu Reeves'})-[:acted_in]->(m:movie)
RETURN pg_typeof(collect(m)) AS collect_vertex, pg_typeof(collect(m.title)) AS collect_propert
y;
```

```
 collect_vertex | collect_property
----------------+------------------
 "vertex[]"     | "jsonb"
```

A native array holds the elements themselves rather than a `jsonb` copy of each one, so its members keep their identity. `films[0]` is a `vertex`, and you can read a property off it or follow it in a later clause.

Several list functions carry one overload for `jsonb` and one for a native array. Which one you get follows from the argument. From Cypher the argument is usually `jsonb`, so the result is `jsonb`. Hand the same function a real array in SQL and you get a real array back. **Type conversion functions** gives the signatures.

Array index origin

Cypher indexes a list from 0 and accepts negative indexes, and it applies that rule to a native array too:

```
MATCH (:person {name: 'Keanu Reeves'})-[:acted_in]->(m:movie)
WITH collect(m) AS films
RETURN films[0].title AS cypher_index_0, films[1].title AS cypher_index_1, size(films) AS n;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 cypher_index_0 |    cypher_index_1     | n
----------------+-----------------------+---
 "The Matrix"   | "The Matrix Reloaded" | 4
```

The same array reached from SQL follows PostgreSQL's rule and starts at 1, and index 0 is out of bounds:

```
SELECT (films[1]).properties ->> 'title' AS sql_index_1,
       (films[0]).properties ->> 'title' AS sql_index_0
FROM (MATCH (:person {name: 'Keanu Reeves'})-[:acted_in]->(m:movie)
      RETURN collect(m) AS films) q;
```

```
 sql_index_1 | sql_index_0
-------------+-------------
 The Matrix  | null
```

`unnest()` turns one back into rows, which is usually what you want in SQL:

```
SELECT (v).id AS id, (v).properties ->> 'title' AS title
FROM (MATCH (:person {name: 'Keanu Reeves'})-[:acted_in]->(m:movie)
      RETURN collect(m) AS films) q,
     unnest(q.films) AS v ORDER BY id;
```

```
 id  |          title
-----+-------------------------
 3.1 | The Matrix
 3.2 | The Matrix Reloaded
 3.3 | The Matrix Revolutions
 3.4 | The Matrix Resurrections
```

Properties: jsonb

A label has one `properties` column of type `jsonb`, and every property of every vertex or edge in that label lives in it. So a property has no declared type: its type is whatever JSON type its value is.

What a property value may be

All six JSON types:

JSON type	Written in Cypher as	Notes
string	`'text'`	Single quotes. Escapes follow JSON's rules.
number	`42`, `1.5`, `1e3`	One JSON number type; no separate integer and float.
boolean	`true`, `false`	
array	`[1, 'two', null]`	May mix types and nest.
object	`{a: {b: 1}}`	Keys fold to lower case unless quoted.

© 2026 SKAI Worldwide Co., Ltd.

JSON type	Written in Cypher as	Notes
null	`null`	Assigning null **removes** the property, unless `allow_null_properties` is on.

```
MATCH (m:movie {title: 'The Matrix'})-[r:related_to]->()
RETURN jsonb_typeof(m.title) AS title, jsonb_typeof(m.embedding) AS embedding,
       jsonb_typeof(r.weight) AS weight, jsonb_typeof(m.no_such_key) AS absent
LIMIT 1;
```

```
  title   | embedding |  weight  | absent
----------+-----------+----------+--------
 "string" | "array"   | "number" | null
```

An absent property and a null property are the same thing. So `jsonb_typeof` of a missing key is null rather than `"null"`.

There is one number type, but the scale you wrote is stored, and it decides whether division truncates:

```
CREATE GRAPH demo;
SET graph_path = demo;
CREATE VLABEL t;
CREATE (:t {a: 7, b: 7.0});
MATCH (n:t) RETURN n.a AS a, n.b AS b, n.a / 2 AS a_div, n.b / 2 AS b_div;
```

```
 a |  b  | a_div |       b_div
---+-----+-------+--------------------
 7 | 7.0 | 3     | 3.5000000000000000
```

`n.a` and `n.b` are both `jsonb_typeof` `number` and they compare equal, so this is not two types. It is one type that remembers how many decimal places it was given.

Because there is no declared type, `jsonb` comparison decides how values of different types order: null, then string, then number, then boolean, then array, then object. That order is PostgreSQL's. It is the reason a property promoted to a typed column can sort differently from the same property read out of the bag.

What a RETURN hands back

A Cypher `RETURN` boxes each scalar output column to `jsonb`. A `boolean` stays a boolean, and a graph type stays itself:

```
SELECT pg_typeof(a) AS property, pg_typeof(b) AS property_cast_to_text,
       pg_typeof(c) AS string_literal, pg_typeof(d) AS number_literal,
       pg_typeof(e) AS boolean, pg_typeof(f) AS id, pg_typeof(g) AS vertex,
       pg_typeof(h) AS path
FROM (MATCH p = (:person {name: 'Keanu Reeves'})-[:acted_in]->(m:movie {title: 'The Matrix'})
      RETURN m.title AS a, m.title::text AS b, 'x' AS c, 1 AS d, true AS e,
             id(m) AS f, m AS g, p AS h) q;
```

```
 property | property_cast_to_text | string_literal | number_literal | boolean |   id    |
vertex |   path
----------+----------------------+----------------+----------------+---------+---------
+--------+-----------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 jsonb    | jsonb                | jsonb          | jsonb          | boolean | graphid | vert
ex | graphpath
```

Note the second column: a `::text` cast inside a `RETURN` is undone by the boxing, so it changes nothing. A string therefore prints with its JSON quotes:

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.title AS title;
```

```
    title
--------------
 "The Matrix"
```

To get a plain scalar, take the value out of `jsonb` in SQL. `#>> '{}'` extracts the whole document as text:

```
SELECT title #>> '{}' AS title
FROM (MATCH (m:movie {title: 'The Matrix'}) RETURN m.title AS title) q;
```

```
   title
------------
 The Matrix
```

Combining SQL and Cypher covers the wrapping in full. Boxing applies to property-derived columns only. A boolean expression stays `boolean`, `id(n)` stays `graphid`, and a graph element stays `vertex`, `edge` or `graphpath`. A property that has been promoted to a typed column is also exempt: that read is answered from the column and comes back in the column's own type.

The :: cast

Cypher's `::` accepts a smaller set of type names than SQL's does. It takes a bare type or function name, such as `text`, `date`, `uuid`, `jsonb` or `inet`. It also takes the grouped spellings the SQL grammar treats specially. Those are the numeric names (`int`, `integer`, `smallint`, `bigint`, `real`, `float`, `double precision`, `decimal`, `numeric`, `boolean`) and the character names (`char`, `varchar`, `character varying`), plus `bit`, `bit varying`, `json` and `interval`.

```
RETURN '42'::int AS i, '4.2'::numeric AS n, 'x'::text AS t, '101'::bit(3) AS b,
       '1 day'::interval AS iv, 'true'::boolean AS bo, '2020-01-01'::date AS d;
```

```
 i  |  n  |  t  |   b   |    iv    | bo |     d
----+-----+-----+-------+----------+----+--------------
 42 | 4.2 | "x" | "101" | "1 day"  | t  | "2020-01-01"
```

The reason is how a cast names its type. A Cypher cast takes an ordinary type name. It also takes the numeric, character, bit, json and interval spellings, which the language treats specially. `TIMESTAMP` and `TIME` are special spellings too, and they are the two that were left out. So both are refused:

```
RETURN '2020-01-01 10:00'::timestamp;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  syntax error at or near "timestamp"
LINE 1: RETURN '2020-01-01 10:00'::timestamp;
                                   ^
```

```
RETURN '10:00'::time;
```

```
ERROR:  syntax error at or near "time"
LINE 1: RETURN '10:00'::time;
                        ^
```

Quote the type name. A quoted name is an identifier rather than a keyword, so it goes through the rule that does work. It names the same type:

```
RETURN '2020-01-01 10:00'::"timestamp" AS ts, '10:00'::"time" AS t;
```

```
          ts           |     t
-----------------------+------------
 "2020-01-01T10:00:00" | "10:00:00"
```

Everything else temporal is an ordinary name and needs no quoting, `date` and `timestamptz` included:

```
RETURN pg_typeof('2020-01-01'::date) AS d, pg_typeof('2020-01-01'::timestamptz) AS tz;
```

```
   d    |             tz
--------+----------------------------
 "date" | "timestamp with time zone"
```

The other route is to cast outside the Cypher, where the whole SQL grammar is available:

```
SELECT (t #>> '{}')::timestamp AS released FROM (RETURN '2020-01-01 10:00' AS t) q;
```

```
      released
---------------------
 2020-01-01 10:00:00
```

Note the difference in the two results. Inside Cypher the value is boxed back to `jsonb`, so it prints in JSON's ISO form with quotes. The SQL cast gives a real `timestamp`, printed by that type's output function.

One more asymmetry to know. `'…'::jsonb` on a string literal produces a `jsonb` **string**, not a parsed document, because the literal is already a JSON string. To build a `jsonb` object in Cypher, write a map literal:

```
RETURN '{"a": 1}'::jsonb AS as_string, {a: 1} AS as_map;
```

```
  as_string   |  as_map
--------------+----------
 "{\"a\": 1}" | {"a": 1}
```


© 2026 SKAI Worldwide Co., Ltd.

Which PostgreSQL types you can use

All of them. A graph lives in the same database as your tables, so every type in **PostgreSQL's data-type chapter** is available unchanged. That covers numeric, character, date/time, boolean, enumerated, geometric, network address, bit string, text search, UUID, XML, JSON, array, composite, range, domain and object identifier. Any type an extension adds is available too.

Three places a PostgreSQL type meets a graph:

- **An ordinary table** beside the graph, joined to it in one statement. This is where a `timestamptz`, a `numeric(10,2)` or a `tsvector` belongs when you need its semantics and its operators.
- **A promoted column** on a label, which materialises one property as a real column of a real type, so that property gets that type's ordering, comparison and index support.
- **An extension type** stored in a property or a promoted column. `vector` from `pgvector` is the case **vector search** builds on.

A property itself cannot be any of those types. A property is `jsonb`, and a value put into one is converted to a JSON type on the way in.

Compatibility

openCypher

`vertex`, `edge` and `graphpath` are the node, relationship and path of openCypher's type system. The scalar types are not separate here: string, number, boolean, list and map are all `jsonb`. So a number's integer-or-decimal character comes from the value rather than from a declared type.

Temporal values are not graph types. `date`, `timestamptz` and `interval` are PostgreSQL's, reached through a cast, an ordinary column or a promoted one. There is no graph-level `datetime` or `duration`. **Cypher compatibility** is the full list of differences.

ISO/IEC 39075

The standard's property graph types line up with `vertex`, `edge` and `graphpath` in the same way. A promoted column provides its typed-value model, in which a property carries a declared type. See **GQL conformance**.

See also

- **Syntax** — literals, operators and how a value is written
- **Server settings** — `allow_null_properties` and `enable_property_promotion`
- **How a graph is stored** — what a label table is made of
- **Combining SQL and Cypher** — getting values across the boundary
- **System catalogs** — `ag_graph`, `ag_label` and the rest
- **PostgreSQL: data types** — every other type you can use

Clauses

A Cypher or GQL statement is a pipeline of clauses. These pages are grouped by the job a clause does in that pipeline rather than alphabetically; every page has the same shape.

Clauses marked **GQL** come from ISO/IEC 39075. They are accepted wherever a Cypher statement is accepted — there is no separate mode. See **GQL conformance** for which parts of the standard are supported.

© 2026 SKAI Worldwide Co., Ltd.

Reading

MATCH

GQL clause

Finds every place in the graph where a pattern fits, and binds one row of variables per match.

Synopsis

```
[ OPTIONAL ] MATCH pattern [, pattern ] ... [ WHERE condition ]
```

where pattern is a chain of node and edge patterns, optionally named:

```
[ path_variable = ] node [ edge node ] ...
```

node is written in parentheses:

```
( [ variable ] [ : label [ ONLY ] ] [ { property_map } ] )
```

edge is written in square brackets between two dashes, with edge_body:

```
[ variable ] [ : label [ ONLY ] ] [ * [ min ] [ .. [ max ] ] ] [ { property_map } ]
```

The dashes and arrowheads around it give the direction. There are three:

```
-[ edge_body ]->     left to right
<-[ edge_body ]-     right to left
-[ edge_body ]-      either direction
```

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`MATCH` reads the graph. It emits one row for every way its pattern can be laid over the stored vertices and edges. Each named element of the pattern is bound to the vertex, edge or path it matched. Later clauses in the pipeline see those bindings. A `MATCH` in the middle of a pipeline runs once per incoming row and joins its matches to that row, the way a join adds rows to a relational query.

- A pattern element with **no variable** is still matched. It just cannot be referred to later. Leave the variable out when you only need the element to constrain the shape.
- A **label** restricts an element to that label and its child labels; `ONLY` restricts it to the label itself. Without a label, an element matches every vertex or every edge in the graph.
- A **property map** in the pattern is an equality test on each key it lists. It is shorthand: `(m:movie {year: 1999})` and `(m:movie) WHERE m.year = 1999` match the same vertices.
- **Commas** separate independent patterns inside one `MATCH`. Patterns that share a variable are joined on it; patterns that share nothing produce their Cartesian product.

© 2026 SKAI Worldwide Co., Ltd.

- Within one pattern match **no edge is used twice**. Node bindings may repeat, so a pattern can close a triangle, but a two-hop pattern never doubles back along the edge it arrived on.
- An **edge variable** may be used once per pattern; reusing it raises `duplicate variable`. A **node variable** may be repeated, which constrains both positions to the same vertex.
- `MATCH` may lead a query or sit anywhere after the first clause. It never ends one: a query has to finish with `RETURN`, `FINISH` or a write clause.

`OPTIONAL MATCH` keeps input rows for which the pattern finds nothing, binding the pattern's variables to null instead of dropping the row. It has its own page, **`OPTIONAL MATCH`**.

Parameters

pattern	One path pattern, optionally preceded by `path_variable =`. Several patterns may be listed, separated by commas.
path_variable	Name bound to the whole matched path, as a value carrying its vertices and edges in order. Read it with the path functions `length()`, `nodes()` and `edges()`. The name must be an identifier: an unreserved keyword such as `path` is rejected here unless you quote it (`"path"`).
variable	Name bound to the matched vertex or edge. Optional. Visible to every clause after this one.
label	A vertex label on a node pattern, an edge label on an edge pattern. One label per element.
ONLY	Excludes child labels, so the element matches rows of that label's own table and nothing that inherits from it. Requires a label.
property_map	A map literal, or a query parameter holding one. Each key is compared for equality against the element's property of that name.
min, max	Bounds on the number of edges a variable-length edge pattern traverses. `*n` is exactly n hops; `*m..n` is between m and n; `*m..` is m or more; `*..n` is up to n; a bare `*` is one or more.
condition	A boolean expression filtering the matches, evaluated with every variable in the pattern bound. See **`WHERE`**.

Examples

Every example runs against the example dataset.

© 2026 SKAI Worldwide Co., Ltd.

Find one vertex by a property

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p;
```

```
                        p
-------------------------------------------------
 person[4.1]{"born": 1964, "name": "Keanu Reeves"}
(1 row)
```

A vertex prints as `label[id]{properties}`. `4.1` is its graph id, which you read with `id(p)`. `p.id` would read a property named `id`, and there is none.

Follow an edge in one direction

```
MATCH (p:person)-[r:acted_in]->(m:movie {title: 'The Matrix'})
RETURN p.name AS actor, r.role AS role
ORDER BY actor;
```

```
        actor         |    role
----------------------+------------
 "Carrie-Anne Moss"   | "Trinity"
 "Keanu Reeves"       | "Neo"
 "Laurence Fishburne" | "Morpheus"
(3 rows)
```

Both endpoints and the edge carry a label, and the edge is bound to `r` so its `role` property can be projected. String values come back quoted because a Cypher `RETURN` produces jsonb.

Leave out the elements you do not need

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m)
RETURN m.title AS film
ORDER BY film;
```

```
      film
---------------
 "Inception"
 "Interstellar"
(2 rows)
```

The edge has no variable and the far endpoint has no label. Both still constrain the match; neither costs a binding.

Match an edge in either direction

An edge pattern with no arrowhead matches the edge whichever way it points:

```
MATCH (m:movie {title: 'Total Recall'})-[:related_to]-(n:movie)
RETURN n.title AS neighbour
ORDER BY neighbour;
```

```
  neighbour
--------------
 "Inception"
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "The Matrix"
(2 rows)
```

`Total Recall` has one outgoing `related_to` edge and one incoming one. Writing arrowheads at both ends, `<-[:related_to]->`, means the same thing.

Join two patterns on a shared variable

```
MATCH (a:person)-[:acted_in]->(m:movie), (d:person)-[:directed]->(m)
WHERE m.title = 'The Matrix'
RETURN a.name AS actor, d.name AS director
ORDER BY actor, director;
```

```
        actor         |     director
----------------------+------------------
 "Carrie-Anne Moss"   | "Lana Wachowski"
 "Carrie-Anne Moss"   | "Lilly Wachowski"
 "Keanu Reeves"       | "Lana Wachowski"
 "Keanu Reeves"       | "Lilly Wachowski"
 "Laurence Fishburne" | "Lana Wachowski"
 "Laurence Fishburne" | "Lilly Wachowski"
(6 rows)
```

Both patterns mention `m`, so they are joined on it. Two consecutive `MATCH` clauses would give the same result; so would one pattern that meets in the middle, `(a:person)-[:acted_in]->(m:movie)<-[:directed]-(d:person)`.

Name the whole path

Put `variable =` in front of a pattern to bind the path itself:

```
MATCH p = (a:movie {title: 'The Matrix'})-[:related_to*1..3]->(b:movie)
RETURN length(p) AS hops, [x IN nodes(p) | x.title] AS route
ORDER BY hops, route;
```

```
 hops |                                                route
------
+-----------------------------------------------------------------------------------------------
 1    | ["The Matrix", "The Matrix Reloaded"]
 1    | ["The Matrix", "The Terminator"]
 1    | ["The Matrix", "Total Recall"]
 2    | ["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
 2    | ["The Matrix", "The Terminator", "Terminator 2: Judgment Day"]
 2    | ["The Matrix", "Total Recall", "Inception"]
 3    | ["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions", "The Matrix
Resurrections"]
(7 rows)
```

Walk a variable number of edges

`*` on an edge pattern repeats it. Each bound is optional:

```
MATCH (m:movie {title: 'The Matrix'})-[:related_to*2]->(n:movie)
RETURN n.title AS reached
ORDER BY reached;
```

```
          reached
-------------------------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Inception"
 "Terminator 2: Judgment Day"
 "The Matrix Revolutions"
(3 rows)
```

A lower bound of `0` includes the zero-length path, so the starting vertex is one of the answers:

```
MATCH (m:movie {title: 'The Matrix'})-[:related_to*0..1]->(n:movie)
RETURN n.title AS reached
ORDER BY reached;
```

```
        reached
----------------------
 "The Matrix"
 "The Matrix Reloaded"
 "The Terminator"
 "Total Recall"
(4 rows)
```

Exclude child labels with ONLY

A label matches its child labels too, because a label is a table and a child label inherits from it. `ONLY` narrows the match to the one label. Give `person` a child label to see the difference:

```
CREATE VLABEL crew INHERITS (person);
CREATE (:crew {name: 'Bill Pope', born: 1952});
```

```
CREATE VLABEL
UPDATE 1
```

```
MATCH (p:person) RETURN count(*) AS people;
```

```
 people
--------
 29
(1 row)
```

```
MATCH (p:person ONLY) RETURN count(*) AS people;
```

```
 people
--------
 28
(1 row)
```

`ONLY` goes after the label and before the property map, as in `(p:person ONLY {name: 'Keanu Reeves'})`. It works the same on an edge pattern, `-[r:acted_in ONLY]->`. Undo the setup with:

```
MATCH (p:crew) DETACH DELETE p;
DROP VLABEL crew;
```


© 2026 SKAI Worldwide Co., Ltd.

```
UPDATE 1
DROP VLABEL
```

Filter the matches with WHERE

`WHERE` attached to a `MATCH` is part of the match, not a later step:

```
MATCH (m:movie) WHERE m.year > 2015
RETURN m.title AS film, m.year AS year
ORDER BY year, film;
```

```
            film            | year
----------------------------+------
 "Blade Runner 2049"        | 2017
 "The Matrix Resurrections" | 2021
(2 rows)
```

Match edges of several labels

An edge pattern carries at most one label. Listing two raises an error:

```
MATCH (p:person)-[r:acted_in|directed]->(m:movie) RETURN count(r);
```

```
ERROR:  multiple types for relationship not supported
```

Match an unlabelled edge instead and test its label in `WHERE`:

```
MATCH (p:person)-[r]->(m:movie)
WHERE label(r) IN ['acted_in', 'directed']
RETURN count(r) AS edges;
```

```
 edges
-------
 46
(1 row)
```

`label()` returns the element's label as a string. `IN` over a list of names is therefore the general form, and the only way to match a set of edge labels.

Notes

A label test belongs in `label(n) = '...'`, not in a colon in `WHERE`. Writing `WHERE n:movie` is a syntax error, and so is a bare pattern used as a predicate: `WHERE (m)-[:related_to]->(:movie)` does not parse. Use `WHERE EXISTS { MATCH (m)-[:related_to]->(:movie) }` to test for a pattern's existence.

A property map compares jsonb values, and so does a comparison against a property. That matters when the literal's type does not match the stored one. See **`WHERE`**.

> **NOTE**
>
> The named-path variable is parsed more narrowly than an element variable. `(path)` is an acceptable node variable, but `MATCH path = ...` is a syntax error because `path` is a keyword; write `MATCH p = ...` or quote the name.

© 2026 SKAI Worldwide Co., Ltd.

Compatibility

Label expressions (ISO/IEC 39075 §16.8)

The standard's `<label expression>` combines label names with `|` for disjunction, `&` for conjunction and `!` for negation. AgensGraph accepts a single label name per pattern element, so none of the three is available. `-[r:acted_in|directed]->` raises `multiple types for relationship not supported`, and `-[r:acted_in|:directed]->` is a syntax error. openCypher spells the disjunction with a pipe too, and it is refused here on a fixed-length edge, on a variable-length one and inside `shortestpath` alike. Use `WHERE label(r) IN [...]` as shown above.

Labeled predicate (ISO/IEC 39075 §19.9)

The standard writes a label test as `n IS LABELED movie` or `n:movie`. AgensGraph accepts neither spelling; use `label(n) = 'movie'`.

The ONLY modifier (ISO/IEC 39075 §16.7)

The standard's `<element pattern filler>` holds a variable declaration, a label expression and a predicate, and nothing else. There is no `ONLY`. AgensGraph adds it because a label here is a table that child labels inherit from, so a label match is a match over an inheritance tree. See **label inheritance**.

See also

- **`OPTIONAL MATCH`** — keep the rows a pattern does not match
- **`WHERE`** — the conditions a match can carry
- **`CREATE`** and **`MERGE`** — the same pattern syntax, used to write
- **`LOAD FROM`** — read a table into the same pipeline
- Functions — `length()`, `nodes()`, `edges()` and the other path functions
- Querying — path queries, including the `shortestpath`, `allshortestpaths` and `dijkstra` forms

OPTIONAL MATCH

GQL clause

Matches a pattern the way `MATCH` does, but keeps an input row whose pattern finds nothing, binding the pattern's variables to null for it.

Synopsis

OPTIONAL MATCH *pattern* [, *pattern*] ... [**WHERE** *condition*]

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`OPTIONAL MATCH` takes the same patterns as **`MATCH`** and matches them the same way. The difference is what happens when a pattern finds no match. A plain `MATCH` drops the row. `OPTIONAL MATCH` keeps it and binds every variable the pattern introduces to null. It is the graph counterpart of a `LEFT JOIN`.

- Variables already in scope keep their values. Only the variables the optional pattern introduces can become null.
- A `WHERE` attached to an `OPTIONAL MATCH` is **part of the optional match**. A row whose pattern matched but whose condition failed is kept with nulls, not dropped. To drop it, filter after the clause with **`WITH`** or **`FILTER`**.
- Leading a query with `OPTIONAL MATCH` yields one all-null row when the pattern matches nothing, rather than no rows.

© 2026 SKAI Worldwide Co., Ltd.

- Several comma-separated patterns in one clause are optional together: they either all match or the whole clause contributes nulls.

Parameters

pattern	A path pattern, in the same syntax `MATCH` uses. See **`MATCH`** for the node, edge, path and `ONLY` forms.
condition	A boolean expression evaluated as part of the match. A row that fails it is kept, with the pattern's variables null.

Examples

Every example runs against the example dataset.

List people whether or not they directed anything

```
MATCH (p:person) WHERE p.born > 1974
OPTIONAL MATCH (p)-[:directed]->(m:movie)
RETURN p.name AS person, m.title AS directed
ORDER BY person;
```

```
      person       | directed
-------------------+-----------
 "Neill Blomkamp"  | "Elysium"
 "Ryan Gosling"    |
 "Sam Worthington" |
(3 rows)
```

Two of the three directed nothing, so `m` is null for them, and therefore `m.title` is null too. The same query with a plain `MATCH` on the second line returns one row:

```
MATCH (p:person) WHERE p.born > 1974
MATCH (p)-[:directed]->(m:movie)
RETURN p.name AS person, m.title AS directed
ORDER BY person;
```

```
      person      | directed
------------------+-----------
 "Neill Blomkamp" | "Elysium"
(1 row)
```

Keep only the rows that failed to match

Test the optional variable for null after the clause. This finds the science fiction films with no outgoing `related_to` edge:

```
MATCH (m:movie) WHERE m.genre = 'Sci-Fi'
OPTIONAL MATCH (m)-[:related_to]->(n:movie)
WITH m, n WHERE n IS NULL
RETURN m.title AS film
ORDER BY film;
```


© 2026 SKAI Worldwide Co., Ltd.

```
        film
---------------------
 "Avatar"
 "Blade Runner 2049"
 "Elysium"
 "Gattaca"
 "Inception"
 "The Fifth Element"
(6 rows)
```

See where the WHERE goes

`Jurassic Park` has one actor, Sam Neill, born 1947. With the condition inside the optional match, the film survives with a null actor:

```
MATCH (m:movie) WHERE m.genre = 'Adventure'
OPTIONAL MATCH (p:person)-[:acted_in]->(m) WHERE p.born > 1950
RETURN m.title AS film, p.name AS actor
ORDER BY film, actor;
```

```
      film       | actor
-----------------+-------
 "Jurassic Park" |
(1 row)
```

With the same condition applied after the clause, the film is dropped:

```
MATCH (m:movie) WHERE m.genre = 'Adventure'
OPTIONAL MATCH (p:person)-[:acted_in]->(m)
WITH m, p WHERE p.born > 1950
RETURN m.title AS film, p.name AS actor
ORDER BY film, actor;
```

```
 film | actor
------+-------
(0 rows)
```

Lead a query with an optional match

```
OPTIONAL MATCH (m:movie {title: 'Nope'})
RETURN m;
```

```
 m
---

(1 row)
```

One row, with `m` null. A plain `MATCH` here returns no rows at all.

Notes

Reading a property of a null vertex gives null rather than an error, so `m.title` is safe on a row the pattern did not match. Testing the variable itself, `WHERE n IS NULL`, is the reliable way to detect that row. A matched vertex may legitimately be missing the property you would otherwise test.

© 2026 SKAI Worldwide Co., Ltd.

Compatibility

Optional blocks (ISO/IEC 39075 §14.4)

The standard's `<optional match statement>` also allows a braced or parenthesized block of match statements, `OPTIONAL { MATCH ... MATCH ... }`. AgensGraph accepts `OPTIONAL` only directly in front of `MATCH`; the block form is a syntax error. `OPTIONAL` also prefixes `CALL`.

See also

- **`MATCH`** — the pattern syntax, and the non-optional form
- **`WHERE`** — conditions, and where they take effect
- **`WITH`** — the clause that filters after an optional match
- **`CALL { subquery }`** — `OPTIONAL` prefixes this clause too

WHERE

GQL clause

Constrains a pattern or filters a projection: rows for which the condition is not true are dropped.

Synopsis

```
WHERE condition
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`WHERE` is not a clause of its own. It attaches to one of three clauses and takes its meaning from the one it attaches to:

- On **`MATCH`** it is part of the match. Every variable in the pattern is bound when the condition runs, and a match that fails it is not produced.
- On **`OPTIONAL MATCH`** it is part of the optional match. A row that fails the condition is **kept**, with the pattern's variables null.
- On **`WITH`** it filters the projected rows, after any aggregation. This is where a condition over an aggregate goes.

The condition must be a boolean expression. A value of another type is not coerced: a jsonb string or number in a condition raises an error rather than being treated as true. Aggregates are not allowed in a `WHERE`. Put them in a `WITH` and filter the result.

`WHERE` on its own is a syntax error, and a query cannot end with it. The standalone, chainable filter is **`FILTER`**, which takes both `FILTER condition` and `FILTER WHERE condition`.

Parameters

condition	An expression of type boolean. Combine tests with `AND`, `OR`, `XOR` and `NOT`.

Examples

Every example runs against **the example dataset**.

© 2026 SKAI Worldwide Co., Ltd.

Compare a property

```
MATCH (m:movie) WHERE m.year > 2015
RETURN m.title AS film, m.year AS year
ORDER BY year, film;
```

```
           film            | year
---------------------------+------
 "Blade Runner 2049"       | 2017
 "The Matrix Resurrections" | 2021
(2 rows)
```

Combine conditions

```
MATCH (m:movie) WHERE m.genre = 'Action' AND m.year < 1995
RETURN m.title AS film
ORDER BY film;
```

```
            film
-------------------------------
 "Terminator 2: Judgment Day"
 "The Terminator"
(2 rows)
```

`XOR` is available alongside `AND`, `OR` and `NOT`:

```
MATCH (m:movie) WHERE m.genre = 'Action' XOR m.year < 1995
RETURN count(*) AS n;
```

```
 n
---
 8
(1 row)
```

Test a label

`label()` returns an element's label as a string, for vertices and edges alike:

```
MATCH (n) WHERE label(n) = 'person'
RETURN count(*) AS n;
```

```
 n
----
 28
(1 row)
```

`IN` over a list of names is how a pattern matches several edge labels, which an edge pattern cannot express:

```
MATCH (p:person)-[r]->(m:movie)
WHERE label(r) IN ['acted_in', 'directed']
RETURN count(r) AS edges;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 edges
-------
 46
(1 row)
```

Match on a substring or a pattern

`STARTS WITH`, `ENDS WITH` and `CONTAINS` test a string property:

```
MATCH (m:movie) WHERE m.title STARTS WITH 'The Matrix'
RETURN count(*) AS n;
```

```
 n
---
 4
(1 row)
```

```
MATCH (m:movie) WHERE m.title ENDS WITH 'Runner'
RETURN m.title AS film;
```

```
      film
---------------
 "Blade Runner"
(1 row)
```

`=~` matches a POSIX regular expression:

```
MATCH (m:movie) WHERE m.title =~ '^Blade.*'
RETURN m.title AS film
ORDER BY film;
```

```
        film
--------------------
 "Blade Runner"
 "Blade Runner 2049"
(2 rows)
```

Negate with `NOT`; there is no `!~` for a property and no `NOT LIKE`:

```
MATCH (m:movie) WHERE NOT m.title STARTS WITH 'The'
RETURN count(*) AS n;
```

```
 n
----
 11
(1 row)
```

Test for a missing property

A property that was never set reads as null, so `IS NULL` is the test for its absence:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie) WHERE m.runtime IS NULL
RETURN count(*) AS n;
```

```
 n
----
 18
(1 row)
```

Test for a pattern

A subquery expression tests whether a pattern exists without binding anything from it:

```
MATCH (p:person) WHERE EXISTS { MATCH (p)-[:directed]->() }
RETURN p.name AS director
ORDER BY director
LIMIT 4;
```

```
      director
--------------------
 "Andrew Niccol"
 "Christopher Nolan"
 "Denis Villeneuve"
 "James Cameron"
(4 rows)
```

Filter an aggregate with WITH

An aggregate in a `WHERE` is refused:

```
MATCH (p:person)-[:acted_in]->(m:movie) WHERE count(m) > 1
RETURN p.name;
```

```
ERROR:  aggregate functions are not allowed in WHERE
```

Aggregate in a `WITH`, then filter the result. The `WITH` clause's own `WHERE` runs after the grouping:

```
MATCH (p:person)-[:acted_in]->(m:movie)
WITH p, count(m) AS films WHERE films >= 3
RETURN p.name AS actor, films
ORDER BY actor;
```

```
          actor          | films
-------------------------+-------
 "Arnold Schwarzenegger" | 3
 "Keanu Reeves"          | 4
(2 rows)
```

Notes

A property holds a jsonb value, and comparing it to a literal compares jsonb values. In jsonb ordering a number sorts above every string, so a quoted number compared against a numeric property is not the comparison you meant:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie {title: 'The Matrix'})
RETURN m.year > '9999' AS num_gt_str, m.year > 1990 AS num_gt_num;
```

```
 num_gt_str | num_gt_num
------------+------------
 t          | t
(1 row)
```

Write the literal in the property's own type, or cast the property with `m.year::int > 1990`, when the stored type is not the one you expect.

> NOTE
>
> `WHERE` takes a boolean and nothing else. `WHERE m.title` raises `cannot cast "The Matrix" (jsonb type string) to boolean`, where a language with truthiness would treat a non-empty string as true. Compare explicitly, or use `IS NOT NULL`.

Compatibility

Labeled predicate (ISO/IEC 39075 §19.9)

The standard writes a label test as `n IS LABELED movie`, with `n:movie` as a shorthand. AgensGraph accepts neither; use `label(n) = 'movie'`.

Pattern predicates

A bare pattern is not an expression, so it cannot be used as a predicate: `WHERE (m)-[:related_to]->(:movie)` is a syntax error, and so is the unlabelled `WHERE (m)-[]->()`. Wrap it in `EXISTS { MATCH ... }`, which the standard defines in §19.4 and AgensGraph supports.

SQL string operators

`LIKE` and `NOT LIKE` are not part of a Cypher condition. `WHERE m.title LIKE 'Blade%'` is a syntax error, with or without a cast. Use `STARTS WITH`, `ENDS WITH`, `CONTAINS` or `=~`. To use the SQL operators, wrap the Cypher in SQL; see **combining SQL and Cypher**.

See also

- **`MATCH`** — the clause `WHERE` most often attaches to
- **`OPTIONAL MATCH`** — where a failed condition keeps the row
- **`WITH`** — filtering after a projection or an aggregate
- **`FILTER`** — the standalone filter clause
- **Subquery expressions** — `EXISTS`, `COUNT`, `COLLECT` and the rest

LOAD FROM

Reads an ordinary table or view into a Cypher pipeline, one working row per table row.

Synopsis

LOAD FROM *table_name* **AS** *alias*

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See **Syntax conventions**.

© 2026 SKAI Worldwide Co., Ltd.

Description

`LOAD FROM` is the relational source clause. It scans a table and binds each row to **alias**, whose columns are then readable as `alias.column` anywhere later in the pipeline. Because a graph and a table live in the same database, this needs no import step and no foreign-data wrapper.

- The alias's columns keep their **SQL types**. `r.score` on a `numeric(3,1)` column is a numeric, so `r.score > 8.5` is a numeric comparison, not a jsonb one.
- The alias itself is a **row value**. Projecting it gives the row as a jsonb object, so `CREATE (:label = to_jsonb(alias))` is a whole-row load.
- `LOAD FROM` may **lead** a query or follow another clause, and several may appear in one query. It never ends one.
- The clause has **no `WHERE` of its own**. Filter with a following `MATCH` `WHERE`, a `WITH` `WHERE`, or a `FILTER`.
- Anything a `SELECT` can read works: a table, a view, a schema-qualified name.

Parameters

table_name	Name of the table or view to read, optionally schema-qualified. It must exist and be readable; an unknown name raises `relation "..." does not exist`.
alias	Name bound to each row. Read a column as `alias.column`, and the whole row as `alias`.

Examples

Every example runs against **the example dataset**, which includes the ordinary table `rating`.

Read a table

```
LOAD FROM rating AS r
RETURN r.title AS title, r.source AS source, r.score AS score
ORDER BY title, source
LIMIT 4;
```

```
        title        |  source   | score
---------------------+-----------+-------
 "Avatar"            | "critics" | 7.9
 "Blade Runner"      | "critics" | 8.1
 "Blade Runner 2049" | "critics" | 8.0
 "Elysium"           | "critics" | 6.6
(4 rows)
```

The values are quoted because a Cypher `RETURN` boxes its output columns to jsonb, not because the columns are jsonb. Their SQL types are unchanged:

```
LOAD FROM rating AS r RETURN pg_typeof(r.score) AS t LIMIT 1;
```


© 2026 SKAI Worldwide Co., Ltd.

```
     t
-----------
 "numeric"
(1 row)
```

Join a table to the graph

Load the table, then match against it. The relational column and the graph property are compared directly:

```
LOAD FROM rating AS r
MATCH (m:movie {title: r.title})
WHERE r.source = 'critics' AND r.score > 8.5
RETURN m.title AS film, m.year AS year, r.score AS score
ORDER BY score DESC;
```

```
             film             | year | score
------------------------------+------+-------
 "Inception"                  | 2010 | 8.8
 "The Matrix"                 | 1999 | 8.7
 "Interstellar"               | 2014 | 8.7
 "Terminator 2: Judgment Day" | 1991 | 8.6
(4 rows)
```

Read a whole row

Project the alias by itself to get the row as a map:

```
LOAD FROM rating AS r RETURN r LIMIT 2;
```

```
                             r
-------------------------------------------------------------
 {"score": 8.7, "title": "The Matrix", "source": "critics"}
 {"score": 9.1, "title": "The Matrix", "source": "audience"}
(2 rows)
```

Read a view

Any relation a `SELECT` can read is a valid source:

```
CREATE VIEW critic AS SELECT title, score FROM rating WHERE source = 'critics';
```

```
CREATE VIEW
```

```
LOAD FROM critic AS c
RETURN c.title AS film, c.score AS score
ORDER BY score DESC
LIMIT 3;
```

```
      film      | score
----------------+-------
 "Inception"    | 8.8
 "Interstellar" | 8.7
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "The Matrix"    | 8.7
(3 rows)
```

```
DROP VIEW critic;
```

```
DROP VIEW
```

Build vertices from table rows

Name the properties you want:

```
CREATE TABLE tmp_people (name text, born int);
INSERT INTO tmp_people VALUES ('Ana Lily Amirpour', 1980), ('Rian Johnson', 1973);
```

```
CREATE TABLE
INSERT 0 2
```

```
LOAD FROM tmp_people AS t
CREATE (:person {name: t.name, born: t.born});
```

```
UPDATE 2
```

Or take the whole row as the property map, using `= expression` in place of a map literal:

```
LOAD FROM tmp_people AS t
CREATE (:person = to_jsonb(t));
```

```
UPDATE 2
```

Undo both loads and drop the table:

```
MATCH (p:person) WHERE p.name IN ['Ana Lily Amirpour', 'Rian Johnson'] DETACH DELETE p;
```

```
UPDATE 4
```

```
DROP TABLE tmp_people;
```

```
DROP TABLE
```

Notes

`LOAD FROM` reads. It has no bulk-load machinery of its own, no parallel writer and no error routing. The `CREATE` that follows it inserts row by row through the graph write path. For loading a large table, and for the index and logging settings that make it faster, see **loading data**.

© 2026 SKAI Worldwide Co., Ltd.

See also

- `CREATE` — the clause that turns loaded rows into vertices and edges
- `MATCH` — joining loaded rows to the graph
- `WITH` — filtering and aggregating loaded rows
- Combining SQL and Cypher — the other direction, a Cypher query inside SQL
- Loading data — the practical loading guide

Projecting

RETURN

GQL clause

Ends a query and names the columns of its result.

Synopsis

RETURN [**DISTINCT** | **ALL**] { * | expression [**AS** alias] } [, ...]

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`RETURN` is the terminal projection. It evaluates its expressions once per working row and emits them as the query's result columns. A read query must end with `RETURN` or `FINISH`; a write query may end with either of those or with the write clause itself.

- **A property-derived column comes back as jsonb.** `RETURN m.title` prints `"The Matrix"`, with the quotes, and so does `RETURN m.title::text`. To get a plain scalar, wrap the query in SQL. See Notes.
- `*` projects every **named** binding in scope. Pattern elements you left unnamed are not projected.
- An **aggregate** in the list groups implicitly: the non-aggregated expressions become the grouping key, as they would in SQL with a `GROUP BY`.
- `DISTINCT` removes duplicate result rows. `ALL` is the explicit spelling of the default and keeps them.
- `ORDER BY`, `SKIP` / `OFFSET` and `LIMIT` are **separate clauses** that follow a `RETURN`, not parts of it. They have their own pages, `ORDER BY` and `SKIP` and `LIMIT`.
- Unlike `WITH`, `RETURN` allows the same output name twice, because nothing downstream has to resolve it.

Parameters

DISTINCT	Removes duplicate rows from the result.
ALL	Keeps duplicate rows. This is the default.
*	Projects every named binding currently in scope.

© 2026 SKAI Worldwide Co., Ltd.

expression	Any Cypher expression over the bindings in scope. That includes a vertex or edge variable, a property reference, a path variable, a function call, an aggregate, an arithmetic or `CASE` expression, a list or map constructor, and a constant.
alias	The result column's name. Without double quotes it folds to lower case; quote it to keep the case you wrote. With no alias, a property reference takes the property's name and anything else becomes `?column?`.

Examples

Every example runs against the example dataset.

Return a whole vertex

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p;
```

```
                         p
-----------------------------------------------
 person[4.1]{"born": 1964, "name": "Keanu Reeves"}
(1 row)
```

A vertex prints as `label[id]{properties}`. An edge adds its endpoints:

```
MATCH (:person {name: 'Keanu Reeves'})-[r:acted_in]->(:movie {title: 'The Matrix'})
RETURN r;
```

```
                    r
-------------------------------------
 acted_in[5.1][4.1,3.1]{"role": "Neo"}
(1 row)
```

Return properties, with and without an alias

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p.born;
```

```
 born
------
 1964
(1 row)
```

An unaliased property reference takes the property's name; a computed expression has no name to take:

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p.born + 1;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 ?column?
----------
 1965
(1 row)
```

An unquoted alias folds to lower case, a quoted one does not:

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p.born AS BornIn;
```

```
 bornin
--------
 1964
(1 row)
```

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p.born AS "BornIn";
```

```
 BornIn
--------
 1964
(1 row)
```

Return everything that has a name

```
MATCH (p:person {name: 'Ryan Gosling'})-[r:acted_in]->(m:movie)
RETURN *;
```

```
                        p                         |                 r                  |
m
--------------------------------------------------+------------------------------------
+-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
-------------------------------------------------------------------------
 person[4.10]{"born": 1980, "name": "Ryan Gosling"} | acted_in[5.13][4.10,3.9]{"role": "K"} |
movie[3.9]{"plot":
"A new blade runner unearths a long-buried secret that has the potential to plunge what's left
of society into chaos.", "year": 2017, "genre": "Sci-Fi", "title": "Blade Runner 2049", "embed
ding": [0.21560573, -0.07505179, -0.01331814, 0.13403069]}
(1 row)
```

Three columns, one per named element. Had the pattern been written `(p:person {name: 'Ryan Gosling'})-[:acted_in]->(m:movie)`, `RETURN *` would give two.

Remove duplicate rows

Twenty-five `acted_in` edges point at films of three genres:

```
MATCH (p:person)-[:acted_in]->(m:movie)
RETURN DISTINCT m.genre AS genre
ORDER BY genre;
```


© 2026 SKAI Worldwide Co., Ltd.

```
    genre
-------------
 "Action"
 "Adventure"
 "Sci-Fi"
(3 rows)
```

Without `DISTINCT` the same query returns 25 rows. `ALL` is the explicit form of that default.

Aggregate

An aggregate with no other expression collapses the whole input to one row:

```
MATCH (m:movie)
RETURN count(*) AS films, min(m.year) AS earliest, max(m.year) AS latest;
```

```
 films | earliest | latest
-------+----------+--------
 18    | 1982     | 2021
(1 row)
```

List any other expression alongside it and that expression becomes the grouping key:

```
MATCH (p:person)-[:acted_in]->(m:movie)
RETURN m.genre AS genre, count(*) AS roles
ORDER BY genre;
```

```
    genre    | roles
-------------+-------
 "Action"    | 13
 "Adventure" | 1
 "Sci-Fi"    | 11
(3 rows)
```

`collect()` gathers a group into a list:

```
MATCH (p:person)-[:acted_in]->(m:movie {title: 'The Matrix'})
RETURN collect(p.name) AS cast;
```

```
                          cast
-------------------------------------------------------
 ["Keanu Reeves", "Laurence Fishburne", "Carrie-Anne Moss"]
(1 row)
```

Return a constant

`RETURN` needs no pattern in front of it:

```
RETURN 1 + 2 AS sum, 'Hello' AS greeting;
```

```
 sum | greeting
-----+----------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 3   | "Hello"
(1 row)
```

Notes

Because a property-derived column comes back as jsonb, a `RETURN` is a poor interface for an application that wants native types. Put the Cypher in a `SELECT` subquery and unbox there. The jsonb operators do it: `->>` reads a key as text, and `#>> '{}'` extracts a jsonb scalar column as text.

```
SELECT d.props ->> 'title' AS film, r.score
FROM (MATCH (p:person)-[:directed]->(m:movie)
      WHERE p.name = 'Christopher Nolan'
      RETURN properties(m) AS props) d
JOIN rating r ON r.title = d.props ->> 'title'
WHERE r.source = 'critics'
ORDER BY r.score DESC;
```

```
     film     | score
--------------+-------
 Inception    |   8.8
 Interstellar |   8.7
(2 rows)
```

The film titles are unquoted text here, and `score` is the table's own numeric. **Combining SQL and Cypher** covers the pattern in full.

Compatibility

Column types

ISO/IEC 39075 §14.11 `<return statement>` gives a result column the declared type of its expression. AgensGraph returns every Cypher result column as jsonb regardless of the expression's type, including after an explicit cast. Unbox in SQL, as above.

DISTINCT and ORDER BY

An `ORDER BY` after `RETURN DISTINCT` may only sort by expressions that appear in the projection:

```
MATCH (m:movie) RETURN DISTINCT m.genre AS genre ORDER BY m.year;
```

```
ERROR:  for SELECT DISTINCT, ORDER BY expressions must appear in select list
```

This matches SQL's rule for `SELECT DISTINCT`. Project the sort key, or drop `DISTINCT` and deduplicate with a grouping aggregate instead.

See also

- **`WITH`** — the same projection, in the middle of a pipeline
- **`ORDER BY`**, **`SKIP`** and **`LIMIT`** — the clauses that follow a `RETURN`
- **`FINISH`** — end a query without returning rows
- **`UNION`** — combine the results of several queries
- **Combining SQL and Cypher** — getting native types out

© 2026 SKAI Worldwide Co., Ltd.

WITH

Projects the working rows into a new set of columns and passes them to the next clause. It is a `RETURN` in the middle of a pipeline.

Synopsis

```
WITH [ DISTINCT | ALL ] { * | expression [ AS alias ] } [, ...] [ WHERE condition ]
```

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`WITH` closes one section of a pipeline and opens the next. It evaluates its expressions per row exactly as `RETURN` does, then hands the resulting columns to the clause that follows. Three jobs need it:

- **Aggregating, then filtering or matching further.** An aggregate cannot appear in a `WHERE`, so aggregate in a `WITH` and filter its output.
- **Computing a value once** and using it in several later clauses.
- **Cutting the pipeline down** with `DISTINCT`, `ORDER BY` and `LIMIT` before the expensive part of a query runs.

Rules that follow from it being a projection:

- `WITH` **replaces** the scope. Only what it projects survives; a variable it omits is gone, and mentioning it later raises `variable does not exist`. Write `WITH *, expr AS v`, or use `LET`, to add without dropping.
- Each output name must be **unique within the clause**. Projecting the same name twice raises `WITH passes on "..." more than once`. `RETURN` has no such rule.
- Its `WHERE` runs **after** the projection and after any aggregation, so it can test an aggregate's alias.
- `ORDER BY`, `SKIP` / `OFFSET` and `LIMIT` are separate clauses that follow the `WITH`, and therefore follow its `WHERE` too. The filter runs first, and the paging applies to what survives.
- `WITH` cannot **lead** a query and cannot end one. At the start of a statement `WITH` is SQL's common table expression, which is a different clause taking a name and a parenthesised query. `LET` has no such collision and may lead.

Parameters

DISTINCT	Removes duplicate rows from the projection before passing them on.
ALL	Keeps duplicate rows. This is the default.
*	Carries every named binding in scope through unchanged. Combine it with further items, as in `WITH *, expr AS v`, to add columns without dropping any.
expression	Any Cypher expression over the bindings in scope, including an aggregate.

© 2026 SKAI Worldwide Co., Ltd.

alias	The name the next clause uses. Required for anything that is not a plain variable reference, because the following clauses have to be able to name it.
condition	A boolean expression filtering the projected rows. Evaluated after the projection, so it can reference an alias defined in this clause.

Examples

Every example runs against the example dataset.

Filter on a count

```
MATCH (p:person)-[:acted_in]->(m:movie)
WITH p.name AS actor, count(m) AS films
WHERE films > 1
RETURN actor, films
ORDER BY films DESC, actor;
```

```
          actor          | films
-------------------------+-------
 "Keanu Reeves"          | 4
 "Arnold Schwarzenegger" | 3
 "Carrie-Anne Moss"      | 2
 "Harrison Ford"         | 2
 "Laurence Fishburne"    | 2
 "Linda Hamilton"        | 2
(6 rows)
```

`count(m)` groups by `p.name`, and the `WHERE` tests the resulting `films`. The same condition in a `MATCH ... WHERE` would be refused with `aggregate functions are not allowed in WHERE`.

Add a column without losing the others

```
MATCH (p:person)-[:directed]->(m:movie)
WITH *, m.year - p.born AS age_at_release
RETURN p.name AS director, m.title AS film, age_at_release
ORDER BY age_at_release
LIMIT 3;
```

```
     director      |       film       | age_at_release
-------------------+------------------+----------------
 "James Cameron"   | "The Terminator" | 30
 "Lilly Wachowski" | "The Matrix"     | 32
 "Andrew Niccol"   | "Gattaca"        | 33
(3 rows)
```

`WITH *` keeps `p` and `m` in scope so `RETURN` can still read their properties. **`LET`** does the same thing without the `*`.

Watch what a WITH drops

Anything the projection omits is out of scope afterwards:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person)-[:acted_in]->(m:movie)
WITH m
RETURN p.name;
```

```
ERROR:  variable does not exist
LINE 1: ...ATCH (p:person)-[:acted_in]->(m:movie) WITH m RETURN p.name;
                                                               ^
```

Deduplicate mid-pipeline

```
MATCH (p:person)-[:acted_in]->(m:movie)
WITH DISTINCT m.genre AS genre
RETURN genre
ORDER BY genre;
```

```
    genre
-------------
 "Action"
 "Adventure"
 "Sci-Fi"
(3 rows)
```

Cut the pipeline down before matching further

`ORDER BY` and `LIMIT` after a `WITH` reduce the rows the next clause has to work on:

```
MATCH (m:movie)
WITH m ORDER BY m.year DESC LIMIT 2
MATCH (p:person)-[:directed]->(m)
RETURN m.title AS film, p.name AS director
ORDER BY film;
```

```
           film            |      director
---------------------------+--------------------
 "Blade Runner 2049"       | "Denis Villeneuve"
 "The Matrix Resurrections" | "Lana Wachowski"
(2 rows)
```

Only the two newest films reach the second `MATCH`.

See that paging follows the filter

`WHERE` belongs to the `WITH`; `ORDER BY` and `LIMIT` are the clauses after it. So the `LIMIT` counts rows that passed the filter:

```
MATCH (m:movie)
WITH m.title AS film, m.genre AS genre WHERE genre = 'Action'
ORDER BY film LIMIT 3
RETURN film, genre;
```

```
            film             |  genre
-----------------------------+----------
 "Terminator 2: Judgment Day" | "Action"
 "The Avengers"               | "Action"
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "The Matrix"                | "Action"
(3 rows)
```

Drop the filter and the same paging picks entirely different rows, because now the first three by title are the ones it keeps:

```
MATCH (m:movie)
WITH m.title AS film, m.genre AS genre
ORDER BY film LIMIT 3
RETURN film, genre;
```

```
        film         |  genre
---------------------+----------
 "Avatar"            | "Sci-Fi"
 "Blade Runner"      | "Sci-Fi"
 "Blade Runner 2049" | "Sci-Fi"
(3 rows)
```

Reject a repeated output name

```
MATCH (m:movie)
WITH m.title AS film, m.genre AS film
RETURN film;
```

```
ERROR:  WITH passes on "film" more than once
LINE 1: MATCH (m:movie) WITH m.title AS film, m.genre AS film RETURN...
                                              ^
```

Carrying a variable and then aliasing something else to its name collides the same way: `WITH m, m.title AS m` raises `WITH passes on "m" more than once`.

Notes

`WHERE` has to be written directly after the projection list, before any `ORDER BY`, `SKIP` or `LIMIT`. Putting it later is a syntax error: `WITH ... ORDER BY film WHERE genre = 'Action'` does not parse.

See also

- **`RETURN`** — the same projection, ending the query
- **`LET`** GQL — add bindings without dropping any
- **`WHERE`** — the condition a `WITH` can carry
- **`ORDER BY`**, **`SKIP`** and **`LIMIT`** — the clauses that follow a `WITH`
- **`UNWIND`** — the usual partner for undoing a `collect()`

LET

GQL clause | Added in 2.18

Adds computed values to the working record without dropping columns already in scope. It is the GQL-standard counterpart to Cypher's `WITH *, expr AS v`.

© 2026 SKAI Worldwide Co., Ltd.

Synopsis

```
LET variable = expression [, variable = expression ] ...
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See **Syntax conventions**.

Description

`LET` introduces one or more new variables into the current row, computed from expressions over variables already in scope. Unlike `WITH`, it is **additive**: every binding in scope is carried forward automatically, so `LET` can never accidentally drop a variable from the pipeline.

- Each variable is **introduce-only**. A name already in scope cannot be rebound, inside the body of a subquery expression or of an importing `CALL { }` as much as at the top level.
- Expressions are evaluated **row-by-row**; aggregates are not allowed.
- `LET` may **lead** a query or sit in the middle, but cannot terminate one. It is the clause to open a statement with, because `WITH` in that position is SQL's common table expression.

Parameters

variable	Name of the new binding. Must not already be in scope. Visible to every clause that follows.
expression	Any row-wise value expression: property access, arithmetic, function calls, `CASE`, list/map constructors. Evaluated once per input row.

Examples

These examples run against **the example dataset**.

Add a derived value and keep everything in scope

The GQL and Cypher forms are equivalent. Switch tabs to compare:

GQL — LET

```
MATCH (p:person)
LET age_in_2026 = 2026 - p.born
RETURN p.name AS name, age_in_2026
ORDER BY age_in_2026 LIMIT 3;
```

Cypher — WITH

```
MATCH (p:person)
WITH *, 2026 - p.born AS age_in_2026
RETURN p.name AS name, age_in_2026
ORDER BY age_in_2026 LIMIT 3;
```

```
       name        | age_in_2026
-------------------+-------------
 "Ryan Gosling"    | 46
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Neill Blomkamp"  | 47
 "Sam Worthington" | 50
(3 rows)
```

The difference is what happens to `p`. `LET` carries it forward automatically. The `WITH` form has to ask for it with `*`, and a `WITH` that forgets the `*` drops `p`, so the `RETURN` then fails.

Lead a query with computed values

Because `LET` is additive it can open a statement, seeding values with nothing yet in scope:

```
LET radius = 7, area = 3.14159 * 7 * 7
RETURN radius, area;
```

```
 radius |   area
--------+-----------
 7      | 153.93791
(1 row)
```

Introduce-only

A name already in scope cannot be rebound:

```
MATCH (p:person) LET p = 1 RETURN p;
```

```
ERROR:  variable "p" already exists
HINT:  LET cannot redefine an existing variable; use a different name.
```

GQL CONFORMANCE

Implements the GQL LET statement (ISO/IEC 39075:2024 §14.7).

- **Supported** — variable definition, multiple bindings, leading position, and row-wise expressions.
- **Not supported** — `LET` over aggregate expressions. Track status in the **conformance matrix**.

NOTE

`LET` became a **reserved keyword** in 2.18. If you used `let` as a label or variable name in an earlier version, quote it (`"let"` — double quotes, not backticks) or rename it. Property keys and `RETURN` aliases are unaffected.

See also

- `WITH` — the Cypher analogue
- `NEXT` GQL
- `RETURN`
- GQL conformance matrix

ORDER BY

GQL clause

Sorts the working rows. It is a clause of its own, so it can sort anywhere in a pipeline, not only at the end.

© 2026 SKAI Worldwide Co., Ltd.

Synopsis

```
ORDER BY sort_key [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...]
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`ORDER BY` sorts the rows currently in the pipeline. It is written after the clause whose rows it sorts, and it applies to that clause's output. That clause is a `RETURN`, a `WITH` or a `MATCH`.

- Rows are compared by the first sort key; ties go to the next, and so on.
- Each key sorts ascending unless you write `DESC`. `ASC` is the explicit default.
- Nulls sort **last** under `ASC` and **first** under `DESC`, unless `NULLS FIRST` or `NULLS LAST` says otherwise.
- A sort key is an expression, not a column number. `ORDER BY 2` sorts by the constant 2, which orders nothing. See Compatibility.
- `ORDER BY` cannot **lead** a query and cannot **end** one.
- After `RETURN DISTINCT`, every sort key has to appear in the projection.

It is a standalone clause, so it can also sort in the middle of a pipeline. A `LIMIT` usually follows, to cut the rows a later clause has to process.

Parameters

sort_key	An expression over the bindings in scope. After a `RETURN` or `WITH` this includes that clause's aliases; it may also be an expression that was not projected at all.
ASC, DESC	Ascending or descending. `ASC` is the default.
NULLS FIRST, NULLS LAST	Where null values go. The default is `NULLS LAST` for an ascending key and `NULLS FIRST` for a descending one.

Examples

Every example runs against the example dataset.

Sort by a projected column

```
MATCH (m:movie) WHERE m.year > 2015
RETURN m.title AS film, m.year AS year
ORDER BY year;
```

```
           film            | year
---------------------------+------
 "Blade Runner 2049"       | 2017
 "The Matrix Resurrections" | 2021
(2 rows)
```


© 2026 SKAI Worldwide Co., Ltd.

Sort by something you did not project

The sort key is an ordinary expression, so it need not be a result column:

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY m.year
LIMIT 3;
```

```
       film
-----------------
 "Blade Runner"
 "The Terminator"
 "Total Recall"
(3 rows)
```

```
MATCH (p:person)
RETURN p.name AS person
ORDER BY length(p.name) DESC
LIMIT 3;
```

```
         person
------------------------
 "Arnold Schwarzenegger"
 "Matthew McConaughey"
 "Laurence Fishburne"
(3 rows)
```

Sort by several keys, in different directions

```
MATCH (m:movie)
RETURN m.genre AS genre, m.year AS year, m.title AS film
ORDER BY genre, year DESC
LIMIT 5;
```

```
  genre   | year |            film
----------+------+----------------------------
 "Action" | 2021 | "The Matrix Resurrections"
 "Action" | 2012 | "The Avengers"
 "Action" | 2003 | "The Matrix Reloaded"
 "Action" | 2003 | "The Matrix Revolutions"
 "Action" | 1999 | "The Matrix"
(5 rows)
```

`genre` ascending, `year` descending within each genre.

Place the nulls

An `OPTIONAL MATCH` leaves nulls behind, which makes the default visible. Ascending puts them last:

```
MATCH (p:person) WHERE p.born > 1974
OPTIONAL MATCH (p)-[:directed]->(m:movie)
RETURN p.name AS person, m.title AS directed
ORDER BY directed;
```


© 2026 SKAI Worldwide Co., Ltd.

```
      person       | directed
-------------------+-----------
 "Neill Blomkamp"  | "Elysium"
 "Ryan Gosling"    |
 "Sam Worthington" |
(3 rows)
```

Descending puts them first:

```
MATCH (p:person) WHERE p.born > 1974
OPTIONAL MATCH (p)-[:directed]->(m:movie)
RETURN p.name AS person, m.title AS directed
ORDER BY directed DESC;
```

```
      person       | directed
-------------------+-----------
 "Ryan Gosling"    |
 "Sam Worthington" |
 "Neill Blomkamp"  | "Elysium"
(3 rows)
```

`NULLS FIRST` asks for that placement without reversing the sort:

```
MATCH (p:person) WHERE p.born > 1974
OPTIONAL MATCH (p)-[:directed]->(m:movie)
RETURN p.name AS person, m.title AS directed
ORDER BY directed NULLS FIRST;
```

```
      person       | directed
-------------------+-----------
 "Ryan Gosling"    |
 "Sam Worthington" |
 "Neill Blomkamp"  | "Elysium"
(3 rows)
```

Sort in the middle of a pipeline

`ORDER BY` after a `MATCH`, with a `LIMIT`, narrows what the rest of the query sees:

```
MATCH (m:movie)
ORDER BY m.year DESC LIMIT 2
MATCH (p:person)-[:directed]->(m)
RETURN m.title AS film, p.name AS director
ORDER BY film;
```

```
            film            |      director
----------------------------+--------------------
 "Blade Runner 2049"        | "Denis Villeneuve"
 "The Matrix Resurrections" | "Lana Wachowski"
(2 rows)
```

The second `MATCH` runs against two rows, not eighteen.

Where ORDER BY cannot go

It cannot open a query:

© 2026 SKAI Worldwide Co., Ltd.

```
ORDER BY 1;
```

```
ERROR:  syntax error at or near "ORDER"
LINE 1: ORDER BY 1;
        ^
```

And it cannot close one, because a sort produces no result on its own:

```
MATCH (m:movie) ORDER BY m.title;
```

```
ERROR:  Cypher query must end with RETURN, FINISH or update clause
```

Notes

Sorting a property sorts a jsonb value. jsonb orders by type before value: a number sorts above every string, and both above null. So a key whose values have mixed types groups by type first. Cast the key when the stored types are mixed and you want one ordering across them.

Compatibility

A number is a sort key, not a column reference

In SQL, `ORDER BY 2` sorts by the second output column. Here the sort key is a value expression, so the integer is a constant: every row compares equal, nothing is reordered, and no error is raised.

```
MATCH (m:movie) WHERE m.year > 2015
RETURN m.title AS film, m.year AS year
ORDER BY 2;
```

```
            film            | year
----------------------------+------
 "The Matrix Resurrections" | 2021
 "Blade Runner 2049"        | 2017
(2 rows)
```

The rows come back in scan order, not by year. Naming the key sorts correctly, as the first example on this page shows. This follows ISO/IEC 39075 §16.17, whose `<sort key>` is an `<aggregating value expression>` with no ordinal form; the deviation is from SQL. Always name the sort key.

ASCENDING and DESCENDING (ISO/IEC 39075 §16.17)

The standard accepts `ASCENDING` and `DESCENDING` alongside `ASC` and `DESC`. AgensGraph accepts only the short spellings; `ORDER BY film ASCENDING` is a syntax error.

See also

- **`SKIP`** and **`LIMIT`** — the paging clauses that usually follow a sort
- **`RETURN`**, **`WITH`** — the projections a sort applies to
- **`MATCH`** — sorting mid-pipeline to cut work

© 2026 SKAI Worldwide Co., Ltd.

SKIP and LIMIT

GQL clause

Pages the working rows: `SKIP` discards rows from the front, `LIMIT` caps how many are kept. `OFFSET` is a synonym for `SKIP`.

Synopsis

```
{ SKIP | OFFSET } count
```

```
LIMIT count
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`SKIP` and `LIMIT` are clauses of their own, so either can page the rows anywhere in a pipeline. Write one after a `RETURN` or a `WITH`, or straight after a `MATCH`. Each may be used without the other.

- `SKIP` **and** `OFFSET` are the same clause. Use whichever spelling you prefer; `OFFSET` is the one ISO/IEC 39075 lists first.
- *count* is an expression, not just a literal, and is evaluated once.
- A negative count is an error. A count larger than the number of rows is not: `SKIP` past the end gives no rows, and `LIMIT` beyond the end gives all of them.
- A fractional count is rounded to the nearest integer.
- Neither clause can **lead** a query, and neither can end one on its own. A query still has to finish with `RETURN`, `FINISH` or a write clause.
- Without an `ORDER BY`, which rows you get is not defined. Sort first whenever the page has to be stable.
- Written after a projection, the modifiers have to be in the order `ORDER BY`, `SKIP` / `OFFSET`, `LIMIT`. See Notes.

Parameters

count	A non-negative integer expression. `SKIP` discards that many leading rows; `LIMIT` keeps at most that many.

Examples

Every example runs against the example dataset.

Take the first few rows

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
LIMIT 2;
```

```
      film
----------------
 "Avatar"
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Blade Runner"
(2 rows)
```

Take a page from the middle

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
SKIP 2
LIMIT 3;
```

```
        film
--------------------
 "Blade Runner 2049"
 "Elysium"
 "Gattaca"
(3 rows)
```

`OFFSET` reads the same:

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
OFFSET 2
LIMIT 3;
```

```
        film
--------------------
 "Blade Runner 2049"
 "Elysium"
 "Gattaca"
(3 rows)
```

Compute the count

Any expression will do:

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
SKIP 1+1
LIMIT 2;
```

```
        film
--------------------
 "Blade Runner 2049"
 "Elysium"
(2 rows)
```

A fraction rounds. `LIMIT 1.5` keeps two rows:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
LIMIT 1.5;
```

```
      film
----------------
 "Avatar"
 "Blade Runner"
(2 rows)
```

Page past the end

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
SKIP 20;
```

```
 film
------
(0 rows)
```

There are 18 films, so nothing is left. A `LIMIT 100` on the same query returns all 18 rather than failing.

Cut the pipeline down mid-query

`LIMIT` after a `MATCH` reduces the rows a later clause has to join:

```
MATCH (m:movie)
ORDER BY m.year DESC LIMIT 2
MATCH (p:person)-[:directed]->(m)
RETURN m.title AS film, p.name AS director
ORDER BY film;
```

```
            film            |      director
----------------------------+--------------------
 "Blade Runner 2049"        | "Denis Villeneuve"
 "The Matrix Resurrections" | "Lana Wachowski"
(2 rows)
```

Reject a negative count

```
MATCH (m:movie) RETURN m.title AS film ORDER BY film LIMIT -1;
```

```
ERROR:  LIMIT must not be negative
```

```
MATCH (m:movie) RETURN m.title AS film ORDER BY film SKIP -1;
```

```
ERROR:  OFFSET must not be negative
```


© 2026 SKAI Worldwide Co., Ltd.

`SKIP` reports itself as `OFFSET` because the two are one clause.

Where they cannot go

Neither clause can open a query:

```
LIMIT 3;
```

```
ERROR:  syntax error at or near "LIMIT"
LINE 1: LIMIT 3;
        ^
```

And neither closes one:

```
MATCH (m:movie) LIMIT 3;
```

```
ERROR:  Cypher query must end with RETURN, FINISH or update clause
```

Notes

After a projection, write the modifiers in the order `ORDER BY`, then `SKIP` / `OFFSET`, then `LIMIT`. Only that order folds into the projection. Put `LIMIT` before `SKIP` and the trailing `SKIP` is read as a clause of its own, so the query no longer ends on a projection. That is what the error reports:

```
MATCH (m:movie) RETURN m.title AS film ORDER BY film LIMIT 3 SKIP 2;
```

```
ERROR:  Cypher query must end with RETURN, FINISH or update clause
```

Mid-pipeline both orders parse, and they mean different things. Each clause applies to the rows the one before it produced. Skipping first takes rows three to six:

```
MATCH (m:movie) ORDER BY m.title SKIP 2 LIMIT 4 RETURN m.title AS film;
```

```
        film
--------------------
 "Blade Runner 2049"
 "Elysium"
 "Gattaca"
 "Inception"
(4 rows)
```

Limiting first cuts to four rows and then skips two of them, leaving two:

```
MATCH (m:movie) ORDER BY m.title LIMIT 4 SKIP 2 RETURN m.title AS film;
```

```
        film
---------------------
 "Blade Runner 2049"
 "Elysium"
(2 rows)
```


© 2026 SKAI Worldwide Co., Ltd.

Compatibility

SKIP and OFFSET (ISO/IEC 39075 §16.19)

The standard's `<offset synonym>` is `OFFSET | SKIP`, so both spellings conform. openCypher has only `SKIP`.

See also

- **`ORDER BY`** — sort first, so a page is reproducible
- **`RETURN`**, **`WITH`** — the projections these clauses page
- **`MATCH`** — paging mid-pipeline to cut work

Writing

CREATE

Adds vertices and edges to the graph, one for every element of its pattern.

Synopsis

CREATE *pattern* [, *pattern*] ...

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`CREATE` writes the pattern you give it into the graph. It takes the same pattern syntax **`MATCH`** reads with, but every element is built rather than looked for.

- `CREATE` **always creates**. It does not check whether an identical vertex or edge is already there. To create only what is missing, use **`MERGE`**.
- An element that names a **variable already in scope** is reused, not rebuilt. That is how an edge is attached to vertices a `MATCH` found: the endpoints are bound variables, so only the edge is new.
- An element that names a **new variable, or none**, is created. A pattern can therefore create a vertex and its edge in one step.
- Every element needs enough information to be built. An edge needs a label and a direction; a vertex without a label becomes a bare `ag_vertex`.
- A property map may be a **map literal**, a query parameter, or `= expression` where the expression yields a jsonb object. That last form is how a whole table row becomes a property bag. See **`LOAD FROM`**.
- `CREATE` runs once per incoming row. With no clause in front of it, it runs once.
- `CREATE` may lead a query or follow another clause, and it may end one. Add a **`RETURN`** to see what it built.

`INSERT` is an exact synonym, the spelling ISO/IEC 39075 uses.

Parameters

pattern	A path pattern in **`MATCH`** syntax. Several may be listed, separated by commas; they are created in order and may share variables.

© 2026 SKAI Worldwide Co., Ltd.

Examples

Every example runs against **the example dataset**. The last example undoes them.

Create one vertex

```
CREATE (:movie {title: 'Tenet', year: 2020, genre: 'Action'});
```

```
UPDATE 1
```

A graph write reports the number of graph elements it touched under the `UPDATE` tag, whatever the clause was.

Create a vertex and see what you made

```
CREATE (p:person {name: 'John David Washington', born: 1984})
RETURN p.name AS person, p.born AS born;
```

```
         person          | born
-------------------------+------
 "John David Washington" | 1984
(1 row)
```

Create an edge between two existing vertices

Find both endpoints, then create only the edge:

```
MATCH (d:person {name: 'Christopher Nolan'}), (m:movie {title: 'Tenet'})
CREATE (d)-[r:directed]->(m)
RETURN d.name AS director, label(r) AS edge, m.title AS film;
```

```
      director       |    edge    |  film
---------------------+------------+---------
 "Christopher Nolan" | "directed" | "Tenet"
(1 row)
```

`d` and `m` are already bound, so they are reused. Had the pattern written `(d:person {name: 'Christopher Nolan'})` again inside the `CREATE`, a second Christopher Nolan would have been created.

Create a vertex and its edge together

Anything in the pattern that is not already bound is built:

```
MATCH (m:movie {title: 'Tenet'})
CREATE (a:person {name: 'Elizabeth Debicki', born: 1990})-[r:acted_in {role: 'Kat'}]->(m)
RETURN a.name AS actor, r.role AS role, m.title AS film;
```

```
        actor         | role  |  film
----------------------+-------+---------
 "Elizabeth Debicki" | "Kat" | "Tenet"
(1 row)
```

Create several patterns at once

Comma-separated patterns are built in one clause and can share variables:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person {name: 'John David Washington'}), (m:movie {title: 'Tenet'})
CREATE (p)-[:acted_in {role: 'The Protagonist'}]->(m),
       (m)-[:related_to {weight: 2}]->(:movie {title: 'Memento', year: 2000, genre:
'Thriller'});
```

```
UPDATE 3
```

Three elements: one `acted_in` edge, one `related_to` edge, and the `Memento` vertex. `Tenet` now has four edges:

```
MATCH (m:movie {title: 'Tenet'})-[r]-(x)
RETURN label(r) AS edge, label(x) AS other, coalesce(x.name, x.title) AS name
ORDER BY edge, name;
```

```
     edge     |  other   |          name
--------------+----------+-------------------------
 "acted_in"   | "person" | "Elizabeth Debicki"
 "acted_in"   | "person" | "John David Washington"
 "directed"   | "person" | "Christopher Nolan"
 "related_to" | "movie"  | "Memento"
(4 rows)
```

Watch CREATE duplicate

Nothing about `CREATE` is idempotent:

```
CREATE (:person {name: 'Duplicate', born: 1}), (:person {name: 'Duplicate', born: 1});
```

```
UPDATE 2
```

```
MATCH (p:person {name: 'Duplicate'}) RETURN count(*) AS rows;
```

```
 rows
------
 2
(1 row)
```

Running the first statement again would make four. Use `MERGE`, or a unique constraint, when a key has to stay unique.

Undo the examples

```
MATCH (p:person {name: 'Duplicate'}) DELETE p;
MATCH (m:movie) WHERE m.title IN ['Tenet', 'Memento'] DETACH DELETE m;
MATCH (p:person) WHERE p.name IN ['John David Washington', 'Elizabeth Debicki'] DETACH DELETE
p;
```


© 2026 SKAI Worldwide Co., Ltd.

```
UPDATE 2
UPDATE 6
UPDATE 2
```

Notes

A vertex's id is assigned from the label's own sequence, so the number a new vertex gets depends on what has been created in that label before. Read it with `id(v)`; do not depend on a particular value.

> NOTE
>
> A `CREATE` with no label — `CREATE ()` — is legal and produces a vertex of the graph's base label `ag_vertex`. Such a vertex is invisible to every `MATCH (n:label)` and easy to lose track of. Always give a label.

Compatibility

The clause name (ISO/IEC 39075 §13.2)

The standard's data-inserting statement is `INSERT`, not `CREATE`. AgensGraph accepts both spellings for the same clause; see **`INSERT`**. SQL's `CREATE` statements, such as `CREATE GRAPH` and `CREATE VLABEL`, are separate statements and are unaffected.

See also

- **`INSERT`** GQL — the standard spelling of this clause
- **`MERGE`** — create only what is not already there
- **`MATCH`** — the pattern syntax, and how to bind endpoints first
- **`SET`** — change properties after creating
- **`DELETE`** — remove what you created
- **`LOAD FROM`** — create from the rows of a table

INSERT

GQL clause | Added in 2.18

Creates the vertices and edges a pattern describes. It is the GQL name for the clause Cypher calls `CREATE`.

Synopsis

INSERT [*path_variable* =] *pattern* [, [*path_variable* =] *pattern*] ...

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`INSERT` and **`CREATE`** are the same clause under two names. The parser builds an identical tree for both, so they accept the same patterns, raise the same errors and produce the same plan.

- Every element in the pattern is created, except one already bound to a variable in scope. That element is reused as an endpoint and not duplicated.
- An `INSERT` runs once per input row. With no preceding clause it runs once.
- Variables the pattern binds are visible to the clauses that follow, and to later patterns in the same `INSERT`.

© 2026 SKAI Worldwide Co., Ltd.

- Every edge needs exactly one label and a direction. Labels must already exist unless `enable_graph_ddl` is on to create them implicitly.

Parameters

path_variable	Optional name bound to the whole path the pattern created. Returning it yields a `graphpath` value listing the new vertices and edges in order.
pattern	A vertex or path pattern, as described in **Syntax**. Property maps may be written inline (`{title: 'Tenet'}`) or supplied whole with `=` from an expression, including a SQL subquery.

Examples

Every example runs against **the example dataset**.

Add one film. `INSERT` and `CREATE` are interchangeable. Switch tabs to compare:

GQL — INSERT

```
INSERT (:movie {title: 'Tenet', year: 2020, genre: 'Action'});
```

Cypher — CREATE

```
CREATE (:movie {title: 'Tenet', year: 2020, genre: 'Action'});
```

```
UPDATE 1
```

Both plan identically:

```
EXPLAIN (COSTS OFF) INSERT (:movie {title: 'Tenet'});
```

```
  QUERY PLAN
--------------
 Graph Create
   ->  Result
(2 rows)
```

Return what you created

`INSERT` binds its variables like any other clause, so a following `RETURN` can report them:

```
INSERT (p:person {name: 'John David Washington', born: 1984}) RETURN p.name AS person;
```


© 2026 SKAI Worldwide Co., Ltd.

```
         person
------------------------
 "John David Washington"
(1 row)
```

Attach a new element to one already in the graph

A variable bound by a preceding `MATCH` is an endpoint, not a new vertex, so only the edge is created:

```
MATCH (p:person {name: 'Christopher Nolan'}), (m:movie {title: 'Tenet'})
INSERT (p)-[:directed]->(m)
RETURN count(*) AS edges;
```

```
 edges
-------
 1
(1 row)
```

Build several patterns in one clause

Patterns are separated by commas and may refer to variables earlier patterns bound:

```
MATCH (m:movie {title: 'Tenet'})
INSERT (a:person {name: 'Robert Pattinson', born: 1986}),
       (b:person {name: 'Elizabeth Debicki', born: 1990}),
       (a)-[:acted_in {role: 'Neil'}]->(m),
       (b)-[:acted_in {role: 'Kat'}]->(m);
```

```
UPDATE 4
```

The tag counts the four elements the one clause created:

```
MATCH (p:person)-[r:acted_in]->(:movie {title: 'Tenet'})
RETURN p.name AS actor, r.role AS role ORDER BY actor;
```

```
        actor         |  role
----------------------+--------
 "Elizabeth Debicki" | "Kat"
 "Robert Pattinson"  | "Neil"
(2 rows)
```

Take a property map from a SQL query

The `= expression` form supplies the whole map at once. Use it to load properties out of a table:

```
INSERT (:movie = (SELECT jsonb_build_object('title', 'Dunkirk', 'year', 2017)));
```

```
UPDATE 1
```

```
MATCH (m:movie {title: 'Dunkirk'}) RETURN m.year AS year;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 year
------
 2017
(1 row)
```

Notes

The clause reports its ordinary graph-write command tag, `UPDATE n`, rather than a `CREATE` tag. Pair it with `FINISH` when you want a write and no projection.

Refusals name `CREATE` even when you wrote `INSERT`, because both spellings build the same clause. The ones worth knowing:

Pattern	Error
`INSERT ()-[]->()`	`only one relationship type is allowed for CREATE`
`INSERT ()-[:acted_in|directed]->()`	`only one relationship type is allowed for CREATE`
`INSERT ()-[:directed]-()`	`only directed relationships are allowed in CREATE`
`INSERT (:ag_vertex)`	`specifying default label is not allowed`
`INSERT ()-[:ag_edge]->()`	`cannot create edge on default label`
`INSERT (:directed)`	`label "directed" is edge label`
`INSERT ()-[:movie]->()`	`label "movie" is vertex label`
`INSERT (m)` where `m` is already bound	`there must be at least one relationship`

NOTE

`insert` became a reserved keyword in 2.18. It is still accepted unquoted wherever that leaves the grammar unambiguous — as a variable name, `MATCH (insert:person)`, and as a label name in a pattern, `MATCH (n:insert)`. It is not accepted as a `CREATE VLABEL` name, as an `ORDER BY` key naming a `RETURN` alias, or as a SQL table or column name; write `"insert"` there, or rename.

Compatibility

GQL CONFORMANCE

Implements the GQL INSERT statement (ISO/IEC 39075:2024 §13.2) over the insert graph pattern of §16.5.

- **Supported** — inserting nodes and edges, several patterns per statement, reusing a bound element as an endpoint, inline property maps, and a path variable over the inserted pattern.
- **Not supported** — the standard's typed-graph checks (§18), which require a graph type; AgensGraph graphs are untyped, so an insert is validated against the labels that exist rather than against a declared node or edge type.

© 2026 SKAI Worldwide Co., Ltd.

Track status in the **GQL conformance matrix**.

openCypher

openCypher spells this clause `CREATE`, which AgensGraph accepts as a synonym. The `INSERT` name comes from GQL and from SQL, where it does the same job on a table.

See also

- **`CREATE`** — the Cypher synonym, same clause
- **`MERGE`** — create only what is not already there
- **`FOR`** GQL — drive an `INSERT` once per list element
- **`FINISH`** GQL — write without projecting
- **Syntax** — the pattern grammar `INSERT` takes
- **GQL conformance matrix**

MERGE

Matches a pattern if it already exists and creates it if it does not, then binds its variables either way.

Synopsis

```
MERGE pattern
  [ ON { CREATE | MATCH } SET set_item [, set_item ] ... ] ...
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See **Syntax conventions**.

Description

`MERGE` is one clause doing the work of a **`MATCH`** and a **`CREATE`**. It looks for its pattern; if the pattern is there it behaves like a `MATCH`, and if it is not there it behaves like a `CREATE`. Either way the pattern's variables end up bound, so the rest of the query does not have to know which happened.

One rule decides everything about `MERGE`. The clause considers **the whole pattern**:

- If the entire pattern matches, nothing is written.
- If any part of it is missing, the **whole pattern is created**, including parts that already exist elsewhere in the graph. Merging `(a)-[:e]->(b)` when `a` and `b` exist but the edge does not creates a second `a` and a second `b` as well as the edge.
- So merge **one element at a time** when you are not sure what is already there. Use a `MERGE` per vertex, then a `MERGE` for the edge between the variables they bound. Written that way, `MERGE` is idempotent.

Other rules:

- `MERGE` takes **exactly one pattern**. Comma-separated patterns are a syntax error; use several `MERGE` clauses.
- `ON CREATE SET` runs only when the pattern was created; `ON MATCH SET` runs only when it was matched. Both may be given, in either order, and each takes a **`SET`** item list.
- Properties in the pattern are part of what is matched. `MERGE (m:movie {title: 'Tenet'})` matches only a `movie` whose `title` is exactly `Tenet`; it does not match one that merely has a `title`.
- `MERGE` runs once per incoming row. It may lead a query, follow another clause, or end one.

© 2026 SKAI Worldwide Co., Ltd.

Parameters

pattern	One path pattern in `MATCH` syntax. Every property in it is both a match condition and, on create, a property to set.
ON CREATE SET	Assignments applied only when the pattern was created.
ON MATCH SET	Assignments applied only when the pattern was found.
set_item	An assignment in `SET` syntax: `variable.property = expression`, `variable = map` or `variable += map`.

Examples

Every example runs against **the example dataset**. The last example undoes them.

Create the vertex, or find it

The first run creates:

```
MERGE (m:movie {title: 'Tenet'}) RETURN m.title AS film, m.year AS year;
```

```
  film   | year
---------+------
 "Tenet" |
(1 row)
```

Only `title` was in the pattern, so `year` is unset. The second run finds the same vertex and writes nothing:

```
MERGE (m:movie {title: 'Tenet'}) RETURN m.title AS film, m.year AS year;
```

```
  film   | year
---------+------
 "Tenet" |
(1 row)
```

```
MATCH (m:movie {title: 'Tenet'}) RETURN count(*) AS rows;
```

```
 rows
------
 1
(1 row)
```

Set different properties on create and on match

Start from nothing:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
UPDATE 1
```

```
MERGE (m:movie {title: 'Tenet'})
ON CREATE SET m.year = 2020, m.source = 'created'
ON MATCH SET m.source = 'matched'
RETURN m.title AS film, m.year AS year, m.source AS source;
```

```
  film   | year |  source
---------+------+-----------
 "Tenet" | 2020 | "created"
(1 row)
```

The identical statement run a second time takes the other branch:

```
MERGE (m:movie {title: 'Tenet'})
ON CREATE SET m.year = 2020, m.source = 'created'
ON MATCH SET m.source = 'matched'
RETURN m.title AS film, m.year AS year, m.source AS source;
```

```
  film   | year |  source
---------+------+-----------
 "Tenet" | 2020 | "matched"
(1 row)
```

`year` keeps the value the create branch gave it, because `ON MATCH SET` did not touch it.

See a whole-pattern MERGE duplicate a vertex

There is one Christopher Nolan, and he has not directed `Tenet`:

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
MATCH (p:person {name: 'Christopher Nolan'}) RETURN count(*) AS nolans;
```

```
UPDATE 1
 nolans
--------
 1
(1 row)
```

Merging the whole path finds no match, because the edge is missing. So it creates the whole path:

```
MERGE (d:person {name: 'Christopher Nolan'})-[:directed]->(m:movie {title: 'Tenet'})
RETURN d.name AS director, m.title AS film;
```

```
      director       |  film
---------------------+---------
 "Christopher Nolan" | "Tenet"
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person {name: 'Christopher Nolan'}) RETURN count(*) AS nolans;
```

```
 nolans
--------
 2
(1 row)
```

A second Christopher Nolan, with no `born` property and none of the original's edges. This is the mistake `MERGE` invites.

Merge one element at a time instead

Undo that, then merge each element separately:

```
MATCH (p:person {name: 'Christopher Nolan'})
WHERE NOT EXISTS { MATCH (p)-[:directed]->(:movie {title: 'Inception'}) }
DETACH DELETE p;
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
UPDATE 2
UPDATE 1
```

```
MERGE (d:person {name: 'Christopher Nolan'})
MERGE (m:movie {title: 'Tenet'})
MERGE (d)-[:directed]->(m)
RETURN d.name AS director, m.title AS film;
```

```
      director       |  film
---------------------+---------
 "Christopher Nolan" | "Tenet"
(1 row)
```

```
MATCH (p:person {name: 'Christopher Nolan'}) RETURN count(*) AS nolans;
```

```
 nolans
--------
 1
(1 row)
```

The first `MERGE` matched the existing person, the second created the film, the third created just the edge. Running the same three clauses again changes nothing:

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
RETURN count(*) AS edges;
```

```
 edges
-------
 3
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

Two from the original dataset plus the one merged, however many times the statement runs.

Reject more than one pattern

```
MERGE (a:movie {title: 'A'}), (b:movie {title: 'B'});
```

```
ERROR:  syntax error at or near ","
LINE 1: MERGE (a:movie {title: 'A'}), (b:movie {title: 'B'});
                                    ^
```

Write two `MERGE` clauses.

Undo the examples

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
UPDATE 2
```

Notes

A unique constraint turns the duplication risk into a clean failure rather than silent data corruption. With `name` unique on `person`, the whole-pattern merge above cannot create its second Nolan:

```
CREATE CONSTRAINT unique_person_name ON person ASSERT name IS UNIQUE;
MERGE (d:person {name: 'Christopher Nolan'})-[:directed]->(m:movie {title: 'Tenet'})
RETURN d.name AS director, m.title AS film;
```

```
CREATE CONSTRAINT
ERROR:  conflicting key value violates exclusion constraint "unique_person_name"
DETAIL:  Key ((properties.'name'))=("Christopher Nolan") conflicts with existing key ((propert
ies.'name'))=("Christopher Nolan").
```

Nothing is created, and the transaction reports the reason. Declare a constraint on whatever key your merges use.

Compatibility

MERGE is not a GQL clause

ISO/IEC 39075 has no `MERGE`: its data-modifying statements are `<insert statement>` (§13.2), `<set statement>` (§13.3), `<remove statement>` (§13.4) and `<delete statement>` (§13.5). `MERGE` comes from Cypher, and AgensGraph keeps it. A standard-only equivalent is a `CALL { subquery }` or an `EXISTS` test followed by `INSERT`.

See also

- **`CREATE`** — write unconditionally
- **`MATCH`** — the pattern syntax
- **`SET`** — the assignment syntax `ON CREATE SET` and `ON MATCH SET` take
- **`DELETE`** — remove what a merge created

SET

GQL clause

Changes the properties of vertices and edges the query has bound.

© 2026 SKAI Worldwide Co., Ltd.

Synopsis

```
SET { target = expression | target += expression } [, ...]
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See **Syntax conventions**.

Description

`SET` writes properties onto elements already bound by an earlier clause. The effect depends on what is on the left of the operator and which operator you use:

Written as	Effect
`n.key = expression`	Sets one property, adding the key if it is not there.
`n.key = NULL`	Removes that key.
`n = {…}`	**Replaces** the whole property bag with the map. Keys not in the map are lost.
`n = other`	Replaces `n`'s bag with `other`'s.
`n += {…}`	**Merges** the map into the bag: listed keys are set, the rest are kept.

- Assignments in one `SET` are separated by commas and applied left to right.
- `SET` runs once per incoming row, so a `MATCH` that finds 10 vertices updates 10 of them.
- `SET` needs a bound variable. It cannot introduce one, and it cannot lead a query. It always follows a `MATCH`, a `MERGE` or another clause that binds something.
- `SET` may end a query. Add a `RETURN` to see the result.
- `SET` writes properties only. It cannot add or change a label.
- `ON CREATE SET` and `ON MATCH SET` on a `MERGE` take this same assignment syntax.

Parameters

target

A property reference, `variable.key`, optionally nested as `variable.key.subkey`; or a bare element variable when you are replacing or merging the whole bag.

expression

The value to store. Any Cypher expression: a literal, a property of another element, an arithmetic or `CASE` expression, a list, a map. For `+=`, and for `=` against a bare variable, it must be a map or another graph element.

Examples

Every example runs against **the example dataset**. The last example undoes them.

© 2026 SKAI Worldwide Co., Ltd.

Set one property

```
CREATE (:movie {title: 'Tenet', year: 2020, genre: 'Action'});
```

```
UPDATE 1
```

```
MATCH (m:movie {title: 'Tenet'})
SET m.runtime = 150
RETURN properties(m) AS props;
```

```
                                props
----------------------------------------------------------------------
 {"year": 2020, "genre": "Action", "title": "Tenet", "runtime": 150}
(1 row)
```

`properties()` returns the whole bag, which is the easiest way to see what a `SET` did.

Set several properties in one clause

```
MATCH (m:movie {title: 'Tenet'})
SET m.genre = 'Sci-Fi', m.runtime = 151
RETURN properties(m) AS props;
```

```
                                props
----------------------------------------------------------------------
 {"year": 2020, "genre": "Sci-Fi", "title": "Tenet", "runtime": 151}
(1 row)
```

Remove a property with NULL

Assigning null deletes the key rather than storing a null:

```
MATCH (m:movie {title: 'Tenet'})
SET m.runtime = NULL
RETURN properties(m) AS props;
```

```
                        props
-----------------------------------------------------
 {"year": 2020, "genre": "Sci-Fi", "title": "Tenet"}
(1 row)
```

`REMOVE` says the same thing more plainly.

Merge a map with +=

`+=` sets the keys the map lists and leaves the others alone:

```
MATCH (m:movie {title: 'Tenet'})
SET m += {runtime: 150, imax: true}
RETURN properties(m) AS props;
```

```
                                      props
-------------------------------------------------------------------------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 {"imax": true, "year": 2020, "genre": "Sci-Fi", "title": "Tenet", "runtime": 150}
(1 row)
```

Replace the whole bag with =

`=` against a bare variable throws away everything not in the map. `genre`, `runtime` and `imax` are gone:

```
MATCH (m:movie {title: 'Tenet'})
SET m = {title: 'Tenet', year: 2020}
RETURN properties(m) AS props;
```

```
              props
-------------------------------
 {"year": 2020, "title": "Tenet"}
(1 row)
```

This is the difference between the two operators, and the one to be careful about: `=` is a replacement, `+=` is an update.

Copy one element's properties to another

The right-hand side can be another element, which copies its whole bag:

```
CREATE (:person {name: 'Stand-in'});
```

```
UPDATE 1
```

```
MATCH (p:person {name: 'Stand-in'}), (k:person {name: 'Keanu Reeves'})
SET p = k
RETURN properties(p) AS props;
```

```
                 props
---------------------------------------
 {"born": 1964, "name": "Keanu Reeves"}
(1 row)
```

There are now two vertices whose `name` is `Keanu Reeves`:

```
MATCH (p:person {name: 'Keanu Reeves'}) RETURN count(*) AS keanus;
```

```
 keanus
--------
 2
(1 row)
```

Copying a bag copies the identifying properties too. It does not copy edges, and it does not merge the two vertices.

Set a property on an edge

An edge takes properties exactly as a vertex does:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (:person {name: 'Keanu Reeves'})-[r:acted_in]->(:movie {title: 'The Matrix'})
SET r.credited = true
RETURN properties(r) AS props;
```

```
                 props
-----------------------------------
 {"role": "Neo", "credited": true}
(1 row)
```

```
MATCH (:person {name: 'Keanu Reeves'})-[r:acted_in]->(:movie {title: 'The Matrix'})
SET r.credited = NULL
RETURN properties(r) AS props;
```

```
      props
-----------------
 {"role": "Neo"}
(1 row)
```

The `MATCH` finds the one edge that belongs to the original Keanu Reeves, because the copy created above has no edges.

Empty a bag

An empty map removes every property, leaving the element in place:

```
MATCH (m:movie {title: 'Tenet'})
SET m = {}
RETURN properties(m) AS props;
```

```
 props
-------
 {}
(1 row)
```

The vertex is now unreachable by any property. The cleanup below therefore looks for a missing `title` rather than for `Tenet`.

Undo the examples

```
MATCH (m:movie) WHERE m.title IS NULL DETACH DELETE m;
MATCH (p:person) WHERE p.name = 'Keanu Reeves' AND NOT EXISTS { MATCH (p)-[]->() } DETACH DELE
TE p;
```

```
UPDATE 1
UPDATE 1
```

Notes

The default setting `allow_null_properties = off` makes `SET n.key = NULL` a deletion. Turn it on and the key is kept with a JSON null value instead:

© 2026 SKAI Worldwide Co., Ltd.

```
SET allow_null_properties = true;
CREATE (m:movie {title: 'Tenet', year: 2020})
SET m.runtime = NULL
RETURN properties(m) AS props;
```

```
SET
                      props
-----------------------------------------------
 {"year": 2020, "title": "Tenet", "runtime": null}
(1 row)
```

```
SET allow_null_properties = false;
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
SET
UPDATE 1
```

A stored null is not the same as an absent key: `m.runtime IS NULL` is true either way, but the key shows up in `properties()` and in a `jsonb` key test.

Compatibility

Setting a label (ISO/IEC 39075 §13.3)

The standard's `<set label item>` adds a label to an element, written `SET n:blockbuster` or `SET n IS blockbuster`. AgensGraph accepts neither:

```
MATCH (m:movie {title: 'Inception'}) SET m:blockbuster RETURN m.title;
```

```
ERROR:  syntax error at or near ":"
LINE 1: MATCH (m:movie {title: 'Inception'}) SET m:blockbuster RETUR...
                                                ^
```

An element's label is the table it lives in, so it is fixed when the element is created. To move a vertex to another label, create it there and delete the original.

See also

- **`REMOVE`** — remove properties by name
- **`MERGE`** — `ON CREATE SET` and `ON MATCH SET`
- **`MATCH`** — binding the elements a `SET` writes to
- **`DELETE`** — remove elements rather than properties

REMOVE

GQL clause

Deletes properties from vertices and edges the query has bound.

© 2026 SKAI Worldwide Co., Ltd.

Synopsis

REMOVE property_reference [, property_reference] ...

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`REMOVE` takes property references and deletes those keys. It is the plain spelling of `SET n.key = NULL`, and does exactly the same thing.

- Removing a key that is **not there** is not an error. The element is left as it was.
- Several references may be listed, separated by commas.
- A reference may be **nested**. `n.key.subkey` removes the inner key and leaves the outer map.
- `REMOVE` runs once per incoming row, so a `MATCH` that finds many elements strips them all. There is no `WHERE` on the clause itself; filter in the `MATCH`.
- `REMOVE` needs a bound variable and cannot lead a query. It may end one; add a `RETURN` to see the result.
- `REMOVE` removes properties only. It cannot remove a label, and it cannot remove an element. Removing an element is `DELETE`.

Parameters

property_reference	A property of a bound element, `variable.key`, optionally nested as `variable.key.subkey`. A bare element variable is rejected.

Examples

Every example runs against **the example dataset**. The last example undoes them.

Remove one property

```
CREATE (:movie {title: 'Tenet', year: 2020, genre: 'Action', runtime: 150, imax: true});
```

```
UPDATE 1
```

```
MATCH (m:movie {title: 'Tenet'})
REMOVE m.runtime
RETURN properties(m) AS props;
```

```
                            props
-----------------------------------------------------------------
 {"imax": true, "year": 2020, "genre": "Action", "title": "Tenet"}
(1 row)
```

Remove several at once

```
MATCH (m:movie {title: 'Tenet'})
REMOVE m.imax, m.genre
RETURN properties(m) AS props;
```


© 2026 SKAI Worldwide Co., Ltd.

```
                props
-----------------------------------
 {"year": 2020, "title": "Tenet"}
(1 row)
```

Remove a key that is not there

Nothing happens, and nothing complains:

```
MATCH (m:movie {title: 'Tenet'})
REMOVE m.nosuchkey
RETURN properties(m) AS props;
```

```
                props
-----------------------------------
 {"year": 2020, "title": "Tenet"}
(1 row)
```

Remove a key inside a nested map

A property may itself be a map:

```
MATCH (m:movie {title: 'Tenet'})
SET m.crew = {dop: 'Hoyte van Hoytema', editor: 'Jennifer Lame'}
RETURN properties(m) AS props;
```

```
                                                 props
-------------------------------------------------------------------------------------------------
-----
 {"crew": {"dop": "Hoyte van Hoytema", "editor": "Jennifer Lame"}, "year": 2020, "title": "Ten
et"}
(1 row)
```

A dotted reference reaches into it, leaving the rest of the map alone:

```
MATCH (m:movie {title: 'Tenet'})
REMOVE m.crew.dop
RETURN properties(m) AS props;
```

```
                                  props
-----------------------------------------------------------------------
 {"crew": {"editor": "Jennifer Lame"}, "year": 2020, "title": "Tenet"}
(1 row)
```

Naming the outer key removes the whole map:

```
MATCH (m:movie {title: 'Tenet'})
REMOVE m.crew
RETURN properties(m) AS props;
```

```
                props
-----------------------------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 {"year": 2020, "title": "Tenet"}
(1 row)
```

Remove a property from an edge

```
MATCH (:person {name: 'Keanu Reeves'})-[r:acted_in]->(:movie {title: 'The Matrix'})
SET r.credited = true
RETURN properties(r) AS props;
```

```
              props
-----------------------------------
 {"role": "Neo", "credited": true}
(1 row)
```

```
MATCH (:person {name: 'Keanu Reeves'})-[r:acted_in]->(:movie {title: 'The Matrix'})
REMOVE r.credited
RETURN properties(r) AS props;
```

```
      props
----------------
 {"role": "Neo"}
(1 row)
```

Undo the examples

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
UPDATE 1
```

Notes

`REMOVE` will not take a bare element variable. Asking it to strip everything that way is refused, with a hint pointing at the `SET` form instead:

```
MATCH (m:movie {title: 'Tenet'}) REMOVE m RETURN m.title;
```

```
ERROR:  cannot set property map to NULL
HINT:  use {} instead of NULL to remove all properties
```

`SET m = {}` empties the bag and keeps the element; `DETACH DELETE m` removes the element altogether.

Compatibility

Removing a label (ISO/IEC 39075 §13.4)

The standard's `<remove label item>` takes a label off an element, written `REMOVE n:blockbuster` or `REMOVE n IS blockbuster`. AgensGraph accepts neither spelling:

```
MATCH (m:movie {title: 'Tenet'}) REMOVE m:movie RETURN m.title;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  syntax error at or near ":"
LINE 1: MATCH (m:movie {title: 'Tenet'}) REMOVE m:movie RETURN m.tit...
                                                ^
```

A label is the table an element lives in, so it is fixed at creation. See `SET` for the same restriction on adding one.

See also

- `SET` — the assignment form, including `= NULL` and `= {}`
- `DELETE` — remove elements instead of properties
- `MATCH` — binding the elements to strip

DELETE

GQL clause

Removes vertices and edges from the graph. `DETACH DELETE` removes a vertex's edges along with it.

Synopsis

[**DETACH**] **DELETE** expression [, expression] ...

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`DELETE` removes the graph elements its expressions evaluate to. It needs them bound first, so it always follows a clause that found them.

- A **plain** `DELETE` **refuses a vertex that still has edges**. Deleting it would leave edges pointing at nothing, so the clause raises an error and the transaction rolls back.
- `DETACH DELETE` deletes the vertex's edges first and then the vertex. Use it whenever the vertex might be connected.
- Deleting an **edge** never needs `DETACH`, and never touches its endpoints.
- Several expressions may be listed. They are deleted in the order given, so listing an edge before its endpoints works without `DETACH`.
- `DELETE` runs once per incoming row. An element already deleted by an earlier row is not an error.
- Each expression must be a vertex, an edge or a path. A property reference is not accepted: to remove a property, use `REMOVE`.
- `DELETE` may end a query. Add a `RETURN` to see what went.

Parameters

DETACH	Delete each vertex's connected edges along with it.
expression	An expression yielding a vertex, an edge or a path, usually a variable bound by a `MATCH`.

© 2026 SKAI Worldwide Co., Ltd.

Examples

Every example runs against **the example dataset**. The first example builds the subgraph the rest remove, so the dataset ends as it began.

Build something to delete

```
CREATE (:movie {title: 'Tenet', year: 2020, genre: 'Action'});
MATCH (d:person {name: 'Christopher Nolan'}), (m:movie {title: 'Tenet'})
CREATE (d)-[:directed]->(m);
MATCH (m:movie {title: 'Tenet'})
CREATE (a:person {name: 'Elizabeth Debicki', born: 1990})-[:acted_in {role: 'Kat'}]->(m);
```

```
UPDATE 1
UPDATE 1
UPDATE 2
```

`Tenet` now has two edges:

```
MATCH (m:movie {title: 'Tenet'})-[r]-(x)
RETURN label(r) AS edge, count(*) AS n
ORDER BY edge;
```

```
    edge    | n
------------+---
 "acted_in" | 1
 "directed" | 1
(2 rows)
```

See a plain DELETE refuse a connected vertex

```
MATCH (m:movie {title: 'Tenet'}) DELETE m;
```

```
ERROR:  Vertex with connected edges cannot be removed
HINT:  Use DETACH DELETE or remove connected edges before deleting the vertex.
```

Nothing is deleted. The graph never holds an edge whose endpoint is gone.

Delete an edge

An edge needs no `DETACH`, and its endpoints survive:

```
MATCH (:person {name: 'Elizabeth Debicki'})-[r:acted_in]->(:movie {title: 'Tenet'})
DELETE r;
```

```
UPDATE 1
```

```
MATCH (p:person {name: 'Elizabeth Debicki'})-[r]-() RETURN count(*) AS edges;
```

```
 edges
-------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 0
(1 row)
```

Delete an unconnected vertex, and see what went

With no edges left, a plain `DELETE` succeeds. A `RETURN` after it reports the element that was removed:

```
MATCH (p:person {name: 'Elizabeth Debicki'})
DELETE p
RETURN properties(p) AS removed;
```

```
                  removed
------------------------------------------
 {"born": 1990, "name": "Elizabeth Debicki"}
(1 row)
```

Delete a vertex and its edges together

`Tenet` still has the `directed` edge. `DETACH DELETE` takes both:

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
UPDATE 2
```

Two elements: the edge and the vertex. Christopher Nolan is untouched, and keeps his other edges:

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
RETURN m.title AS film
ORDER BY film;
```

```
      film
----------------
 "Inception"
 "Interstellar"
(2 rows)
```

Delete several elements in one clause

List the edge before the vertices it joins and no `DETACH` is needed, because by the time the vertices are deleted the edge is gone:

```
CREATE (:person {name: 'Extra Two'});
MATCH (a:person {name: 'Extra Two'}) CREATE (a)-[:acted_in]->(:movie {title: 'Scratch'});
```

```
UPDATE 1
UPDATE 2
```

```
MATCH (a:person {name: 'Extra Two'})-[r:acted_in]->(m:movie {title: 'Scratch'})
DELETE r, a, m;
```


© 2026 SKAI Worldwide Co., Ltd.

```
UPDATE 3
```

The dataset is back to its original size:

```
MATCH (n) RETURN label(n) AS label, count(*) AS n ORDER BY label;
```

```
  label   | n
----------+----
 "movie"  | 18
 "person" | 28
(2 rows)
```

Notes

`DELETE` takes elements, not properties. Naming a property is refused rather than quietly ignored:

```
MATCH (m:movie {title: 'The Matrix'}) DELETE m.title;
```

```
ERROR:  node, relationship, or path is expected
LINE 1: MATCH (m:movie {title: 'The Matrix'}) DELETE m.title;
                                                     ^
```

Use **`REMOVE`** for a property, or `SET m.title = NULL`.

> NOTE
>
> `DETACH DELETE` on a busy vertex can delete a great many edges in one statement, and all of it is one transaction. Count first — `MATCH (v)-[r]-() RETURN count(r)` — when you are not sure how large the blast radius is.

Compatibility

NODETACH (ISO/IEC 39075 §13.5)

The standard's `<delete statement>` is `[ DETACH | NODETACH ] DELETE`, with `NODETACH` the implicit default. AgensGraph implements the default but not the keyword: `NODETACH DELETE m` is a syntax error. Write a plain `DELETE`, which means the same thing.

See also

- **`REMOVE`** — remove properties instead of elements
- **`MATCH`** — binding the elements to delete
- **`CREATE`**, **`MERGE`** — the clauses that put them there
- Data definition — dropping a whole label or graph

Composing

UNWIND

Turns a list into rows: one working row per element.

© 2026 SKAI Worldwide Co., Ltd.

Synopsis

UNWIND list_expression **AS** variable

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See **Syntax conventions**.

Description

`UNWIND` expands a list into the pipeline. For each element it emits one row, binding variable to that element; everything else already in scope is carried along unchanged. It is the inverse of `collect()`, and the way to get from one row holding a list to many rows holding its parts.

- `AS variable` is **required**. There is no unnamed form.
- The list may be a **literal**, a **list-valued property**, a variable a `collect()` produced, or any expression that yields a jsonb array or a PostgreSQL array. A scalar is refused.
- An **empty list** produces no rows, so the whole input row disappears.
- **Nulls inside** the list are elements like any other and produce rows.
- `UNWIND` runs once per incoming row, and the rows it produces multiply out: three input rows over a two-element list give six.
- `UNWIND` may lead a query or follow another clause. It cannot end one.
- Nest two `UNWIND` clauses to flatten a list of lists.

`FOR` is the GQL counterpart and does the same job, with an optional ordinal.

Parameters

list_expression	An expression yielding a jsonb array or a PostgreSQL array.
variable	Name bound to each element. Visible to every clause after this one.

Examples

Every example runs against **the example dataset**.

Expand a list literal

```
UNWIND [1, 91, 203, 1, 7, 91] AS n
RETURN n;
```

```
  n
-----
 1
 91
 203
 1
 7
 91
(6 rows)
```

Six rows, in list order, duplicates included.

© 2026 SKAI Worldwide Co., Ltd.

Deduplicate and sort a list

There is no way to apply `DISTINCT` to a list in place. Unwind it, deduplicate the rows, and collect them back:

```
UNWIND [1, 91, 203, 1, 7, 91] AS n
WITH DISTINCT n
ORDER BY n
RETURN collect(n) AS uniq;
```

```
      uniq
----------------
 [1, 7, 91, 203]
(1 row)
```

Drive a MATCH from a list

One row per element, each matched in turn:

```
UNWIND ['The Matrix', 'Inception'] AS t
MATCH (m:movie {title: t})
RETURN m.title AS film, m.year AS year;
```

```
     film     | year
--------------+------
 "The Matrix" | 1999
 "Inception"  | 2010
(2 rows)
```

This is the usual way to run a query over a client-supplied list of keys, with the list passed as one parameter.

Expand a list-valued property

A property may hold a list, and `UNWIND` opens it:

```
MATCH (m:movie {title: 'The Matrix'})
UNWIND m.embedding AS component
RETURN component;
```

```
  component
-------------
 -0.07594558
 0.04081754
 0.29592122
 -0.11921061
(4 rows)
```

Undo a collect

`collect()` folds rows into a list; `UNWIND` unfolds it again. The pair is how you aggregate, then go back to rows:

```
MATCH (p:person)-[:acted_in]->(m:movie)
WITH collect(DISTINCT m.genre) AS genres
UNWIND genres AS genre
RETURN genre
ORDER BY genre;
```


© 2026 SKAI Worldwide Co., Ltd.

```
    genre
-------------
 "Action"
 "Adventure"
 "Sci-Fi"
(3 rows)
```

Flatten a list of lists

Two `UNWIND` clauses, the second over the variable the first bound:

```
UNWIND [[1, 2], [3]] AS pair
UNWIND pair AS x
RETURN pair, x;
```

```
  pair  | x
--------+---
 [1, 2] | 1
 [1, 2] | 2
 [3]    | 3
(3 rows)
```

Expand an empty list

No rows come out, and the input row goes with them:

```
UNWIND [] AS x RETURN x;
```

```
 x
---
(0 rows)
```

Use `OPTIONAL MATCH` or a `CASE` guard when a row must survive an empty list.

Keep the nulls

A null element is still an element:

```
UNWIND [1, NULL, 3] AS x RETURN x;
```

```
  x
------
 1
 null
 3
(3 rows)
```

The value prints as `null` because the output column is jsonb, and a jsonb null is a value rather than a missing one.

Notes

`UNWIND` requires a list. Neither a scalar nor a bare `NULL` is treated as a one-element or zero-element list:

```
UNWIND 3 AS x RETURN x;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  expression must be jsonb or array, but integer
LINE 1: UNWIND 3 AS x RETURN x;
               ^
```

```
UNWIND NULL AS x RETURN x;
```

```
ERROR:  expression must be jsonb or array, but unknown
LINE 1: UNWIND NULL AS x RETURN x;
               ^
```

Wrap a value that might not be a list, as in `UNWIND coalesce(m.tags, []) AS tag`, when the property is not guaranteed to be there.

Compatibility

UNWIND is not a GQL clause

ISO/IEC 39075 has no `UNWIND`. Its list-expanding statement is `<for statement>` (§14.8), which AgensGraph also implements as **`FOR`**. `FOR` adds a `WITH OFFSET` clause that gives each element its 0-based position, something `UNWIND` cannot do.

See also

- **`FOR`** GQL — the standard clause, with an ordinal
- **`WITH`** — where `collect()` goes, and how to deduplicate the rows
- **`MATCH`** — driving a match from the expanded rows
- Functions — `collect()` and the other list functions

FOR

GQL clause · Added in 2.18

Expands a list into rows, one row per element, optionally binding each element's position. It is the GQL counterpart of Cypher's `UNWIND`.

Synopsis

FOR variable **IN** list [**WITH OFFSET** [**AS** offset_variable]]

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`FOR` joins each input row with the elements of a list, so an input row that meets an n-element list becomes n output rows. When the list depends on the row, the expansion runs per row.

- The element variable must be a **new** name. Reusing one in scope raises `duplicate variable "<name>"`.
- `WITH OFFSET` binds each element's position. The offset is **0-based**.
- An empty list, a null list and a missing property all produce **zero rows**, so a `FOR` over nothing drops its input row.
- A scalar is not a list and is refused, not wrapped in one.
- `FOR` may open a query, sit in the middle of one, or drive a write clause. It cannot end one.

© 2026 SKAI Worldwide Co., Ltd.

Parameters

variable	Name bound to one element per output row. Must not already be in scope.
list	Any expression producing a list: a literal, `range()`, a list-valued property, a list built by an earlier `collect()`, or two lists concatenated with `+`.
offset_variable	Name bound to the element's 0-based position. Omit the `AS` clause and the position is bound to `offset`. That is a reserved word, so it has to be written `"offset"` everywhere afterwards. Prefer the `AS` form.

Examples

Every example runs against **the example dataset**.

Expand a list of titles and look each one up. `FOR` and Cypher's `UNWIND` expand the same list. Switch tabs to compare:

GQL — FOR

```
FOR t IN ['The Matrix', 'Inception', 'Gattaca']
MATCH (m:movie {title: t})
RETURN t AS title, m.year AS year ORDER BY year;
```

Cypher — UNWIND

```
UNWIND ['The Matrix', 'Inception', 'Gattaca'] AS t
MATCH (m:movie {title: t})
RETURN t AS title, m.year AS year ORDER BY year;
```

```
    title     | year
--------------+------
 "Gattaca"    | 1997
 "The Matrix" | 1999
 "Inception"  | 2010
(3 rows)
```

Number the elements as they expand

`WITH OFFSET AS` names the position. You want that when the list order is the answer, as in a ranked list of titles:

```
FOR t IN ['The Matrix', 'Inception', 'Gattaca'] WITH OFFSET AS i
MATCH (m:movie {title: t})
RETURN i, t AS title ORDER BY i;
```

```
 i |    title
---+--------------
 0 | "The Matrix"
 1 | "Inception"
```


© 2026 SKAI Worldwide Co., Ltd.

```
 2 | "Gattaca"
(3 rows)
```

The first element is at 0, not 1.

Reach the offset without naming it

Leaving off the `AS` clause still binds the position, but under the name `offset`, which is a reserved keyword. Every later reference has to be quoted:

```
FOR t IN ['The Matrix', 'Inception'] WITH OFFSET RETURN "offset", t;
```

```
 offset |      t
--------+--------------
 0      | "The Matrix"
 1      | "Inception"
(2 rows)
```

Drop the quotes and the query does not parse:

```
FOR t IN ['The Matrix'] WITH OFFSET RETURN offset, t;
```

```
ERROR:  syntax error at or near "offset"
LINE 1: FOR t IN ['The Matrix'] WITH OFFSET RETURN offset, t;
                                                   ^
```

Expand a list-valued property

The list can come from the matched row, in which case the expansion runs per row:

```
MATCH (m:movie {title: 'The Matrix'})
FOR e IN m.embedding
RETURN e ORDER BY e;
```

```
      e
-------------
 -0.11921061
 -0.07594558
 0.04081754
 0.29592122
(4 rows)
```

Generate the rows a query needs

`range()` is a list like any other, so `FOR` can drive a lookup over generated keys:

```
FOR y IN range(1999, 2001)
MATCH (m:movie {year: y})
RETURN y, m.title AS film ORDER BY y, film;
```

```
  y   |     film
------+--------------
 1999 | "The Matrix"
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

Notes

`FOR` compiles to a lateral `unnest()`, while `UNWIND` compiles to a `ProjectSet`. The two are interchangeable in meaning, but they are not the same plan:

```
EXPLAIN (COSTS OFF) FOR x IN [1, 2, 3] RETURN x;
```

```
                QUERY PLAN
------------------------------------------
 Function Scan on unnest _agens_default_for
(1 row)
```

Because the expansion goes through `unnest()`, a scalar source is a type error rather than a one-element list. An integer raises `function unnest(integer) does not exist` and a scalar property raises `cannot extract elements from a scalar`. A null, by contrast, is treated as an empty list and yields no rows.

> **NOTE**
>
> `FOR` is a reserved keyword in PostgreSQL, so the clause puts no new restriction on identifier names. `offset` is reserved as well, which is why the bare `WITH OFFSET` form has to be quoted where it is read back.

Compatibility

> **GQL CONFORMANCE**
>
> Implements the GQL FOR statement (ISO/IEC 39075:2024 §14.8), Feature GQ10, "FOR statement: list value support", and Feature GQ24, "FOR statement: WITH OFFSET".
>
> - **Supported** — expansion of a list value expression, per-input-row evaluation, a null source treated as an empty list, and `WITH OFFSET` with the 0-based position the standard specifies (`j−1` in §14.8, General Rule 2).
> - **Not supported** — `WITH ORDINALITY` (Feature GQ11), the 1-based counterpart; add one to the offset instead. Also not supported is a `<binding table reference value expression>` as the source (Feature GQ23), since AgensGraph has no named binding tables.
> - **Deviation** — the standard writes the position variable without `AS` (`WITH OFFSET i`). AgensGraph requires `WITH OFFSET AS i`, and a bare `WITH OFFSET` defaults the name to `offset`.
>
> Track status in the GQL conformance matrix.

openCypher

openCypher spells this clause **`UNWIND … AS x`**, which AgensGraph accepts unchanged and which has no offset form.

See also

- **`UNWIND`** — the Cypher analogue, without an offset
- **`INSERT`** GQL — a write `FOR` can drive
- **`LET`** GQL
- **`FILTER`** GQL — narrow the expanded rows
- **`ORDER BY`** — sort the rows a `FOR` produced
- GQL conformance matrix

© 2026 SKAI Worldwide Co., Ltd.

FILTER

GQL clause | Added in 2.18

Keeps only the rows that satisfy a condition. It is a standalone filter that needs no projection, unlike Cypher's `WITH … WHERE`.

Synopsis

```
FILTER [ WHERE ] condition
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`FILTER` discards every row whose condition is not true, and changes nothing else: the columns in scope before it are exactly the columns in scope after it. The optional `WHERE` is noise words. `FILTER x > 1` and `FILTER WHERE x > 1` parse to the same clause.

- `FILTER` is a **middle** clause. It cannot open a query and it cannot end one.
- It applies to whatever the preceding clause produced, including the output of an aggregating `WITH`. It is therefore the `HAVING` of a Cypher pipeline.
- Chained `FILTER`s apply in order, each narrowing the rows the last one kept.
- A row whose condition evaluates to null is discarded, as in SQL.

Parameters

condition	A predicate over the variables in scope. Any boolean expression `WHERE` accepts, including a **subquery value expression** such as `EXISTS { }` or `COUNT { }`. The condition must be a boolean: a bare number is rejected with `argument of FILTER must be type boolean, not type integer`, and a bare property with `cannot cast 2003 (jsonb type numeric) to boolean`.

Examples

Every example runs against **the example dataset**.

Keep the films from 2014 onwards. `FILTER` after a pattern and Cypher's `WHERE` attached to it are the same test. Switch tabs to compare:

GQL — FILTER

```
MATCH (m:movie)
FILTER m.year >= 2014
RETURN m.title AS film, m.year AS year ORDER BY year;
```


© 2026 SKAI Worldwide Co., Ltd.

Cypher — WHERE

```
MATCH (m:movie)
WHERE m.year >= 2014
RETURN m.title AS film, m.year AS year ORDER BY year;
```

```
           film            | year
---------------------------+------
 "Interstellar"            | 2014
 "Blade Runner 2049"       | 2017
 "The Matrix Resurrections" | 2021
(3 rows)
```

Both forms plan identically, down to the same `Filter: (properties.'year' >= '2014'::jsonb)` on the scan of `movie`.

Filter an aggregate

`WHERE` cannot follow an aggregating `RETURN`, and attaching it to `WITH` forces you to restate the projection. `FILTER` tests the aggregate directly:

```
MATCH (p:person)-[:acted_in]->(m:movie)
WITH p.name AS actor, count(m) AS films
FILTER films >= 3
RETURN actor, films ORDER BY actor;
```

```
         actor          | films
------------------------+-------
 "Arnold Schwarzenegger" | 3
 "Keanu Reeves"          | 4
(2 rows)
```

Narrow rows in two stages

Each `FILTER` sees only the rows the one before it kept:

```
MATCH (m:movie)
FILTER m.genre = 'Sci-Fi'
FILTER m.year < 1998
RETURN m.title AS film ORDER BY film;
```

```
       film
--------------------
 "Blade Runner"
 "Gattaca"
 "The Fifth Element"
 "Total Recall"
(4 rows)
```

Filter the rows a write clause produced

A `FILTER` after a write runs the write first, then restricts the rows that flow on. Here `SET` touches the film and `FILTER` decides whether the row is reported:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie {title: 'The Matrix'})
SET m.reviewed = true
FILTER m.year = 1999
RETURN m.title AS film, m.reviewed AS reviewed;
```

```
     film     | reviewed
--------------+----------
 "The Matrix" | true
(1 row)
```

Notes

Because `FILTER` cannot open a query, a statement that begins with it is a syntax error rather than a clause-placement error:

```
FILTER true;
```

```
ERROR:  syntax error at or near "FILTER"
LINE 1: FILTER true;
        ^
```

The one place `FILTER` does come first is after **`NEXT`**, where it operates on the table carried across the boundary. Ending a query on `FILTER` raises `Cypher query must end with RETURN, FINISH or update clause`.

`filter` is an unreserved keyword, so it remains usable as a variable, label or property name.

Compatibility

GQL CONFORMANCE

Implements the GQL FILTER statement (ISO/IEC 39075:2024 §14.6), Feature GQ08, "FILTER statement".

- **Supported** — both spellings the standard defines, `FILTER <search condition>` and `FILTER WHERE <search condition>`, in every non-leading, non-terminal position.
- **Not supported** — nothing in the subclause. `FILTER` is complete.

Track status in the GQL conformance matrix.

openCypher

openCypher has no `FILTER` statement; the equivalent is a `WHERE` attached to `MATCH` or to `WITH`, both of which AgensGraph accepts unchanged. The unrelated openCypher list function `filter(x IN list WHERE predicate)` is not accepted either. The `WHERE` inside the parentheses is a syntax error. Write a list comprehension instead: `[x IN m.embedding WHERE x > 0]`.

See also

- **`WHERE`** — the Cypher analogue, attached to a clause
- **`WITH`** — project and filter in one clause
- **`LET`** GQL — add columns without dropping any
- **`NEXT`** GQL — the one place `FILTER` may come first
- Subquery value expressions GQL — predicates a `FILTER` can use
- GQL conformance matrix

© 2026 SKAI Worldwide Co., Ltd.

NEXT

GQL clause | Added in 2.18

Chains two queries end to end: the rows the first query returns become the input rows of the second.

Synopsis

```
query NEXT query [ NEXT query ] ...
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`NEXT` turns the result table of the query on its left into the driving table of the query on its right. It is the composition operator `WITH` performs inside a single query, promoted to a boundary between two whole queries.

- The query left of `NEXT` must end with `RETURN`.
- The boundary **resets scope**: only the columns that `RETURN` produced are visible on the right. Every other variable is gone.
- Columns cross the boundary with their own types. A vertex is still a vertex, an edge still an edge, a path still a path.
- `NEXT` is left-associative, so `A NEXT B NEXT C` composes as `(A NEXT B) NEXT C`.
- A `UNION`, `INTERSECT` or `EXCEPT` composite may stand on either side.
- `NEXT` is accepted at the start of a statement and inside `EXPLAIN` and `PREPARE`. It is not accepted inside a subquery, in either the `CALL { }` body or a subquery value expression.

Parameters

query	A complete query. The one on the left of `NEXT` must end with `RETURN`, and every column it returns must be named. An unaliased expression raises `expression in WITH must be aliased (use AS)`. The query on the right is a fresh one. It begins with a clause that can open a query (`MATCH`, `OPTIONAL MATCH`, `CREATE`, `INSERT`, `MERGE`, `LOAD FROM`, `UNWIND`, `FOR`, `CALL`, `LET`, `RETURN` or `FINISH`). It can also begin with a clause that operates on the carried table (`WITH`, `FILTER`, `ORDER BY`, `SKIP`, `OFFSET` or `LIMIT`). It ends the way any query ends: with `RETURN`, `FINISH` or a write clause.

Examples

Every example runs against the example dataset.

Carry the films one director made into a second query that finds their casts. `NEXT` and an intermediate `WITH` compose the same two steps. Switch tabs to compare:

GQL — NEXT

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
RETURN m
NEXT
```


© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (a:person)-[:acted_in]->(m)
RETURN m.title AS film, a.name AS actor ORDER BY film;
```

Cypher — WITH

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
WITH m
MATCH (a:person)-[:acted_in]->(m)
RETURN m.title AS film, a.name AS actor ORDER BY film;
```

```
      film      |         actor
----------------+-----------------------
 "Inception"    | "Leonardo DiCaprio"
 "Interstellar" | "Matthew McConaughey"
(2 rows)
```

`m` crosses the boundary as a vertex, so the second query re-traverses from it.

Filter an aggregate computed by the first query

`FILTER` cannot open a standalone query, but it can open the query after `NEXT`. That gives an aggregate a `HAVING`-style filter without an extra `WITH`:

```
MATCH (p:person)-[:acted_in]->(m:movie)
RETURN p.name AS actor, count(m) AS films
NEXT
FILTER films > 2
RETURN actor, films ORDER BY actor;
```

```
          actor          | films
-------------------------+-------
 "Arnold Schwarzenegger" | 3
 "Keanu Reeves"          | 4
(2 rows)
```

Chain three queries

Each `NEXT` is another boundary, evaluated left to right:

```
MATCH (m:movie) RETURN m.genre AS genre
NEXT
RETURN DISTINCT genre AS g
NEXT
RETURN count(g) AS genres;
```

```
 genres
--------
 3
(1 row)
```

Reach a variable the first query did not return

Only returned columns survive the boundary, so a reference to anything else fails at parse time:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
RETURN m
NEXT
RETURN p.name;
```

```
ERROR:  variable does not exist
LINE 4: RETURN p.name;
               ^
```

Notes

`NEXT` costs nothing at run time. The boundary is built by rewriting the left query's `RETURN` into the `WITH` the clause pipeline already places between clause groups. The two tab-set queries above therefore plan identically.

Two guards report the misuse directly rather than as a syntax error. A left query that does not end with `RETURN` raises `a query statement before NEXT must end with RETURN`. A left query ending in `FINISH` raises `FINISH cannot be followed by NEXT`.

The last query in a chain still has to end the statement properly. `ORDER BY`, `SKIP`, `OFFSET` and `LIMIT` are clauses in their own right, but none of them can terminate a query. A chain that stops on one raises `Cypher query must end with RETURN, FINISH or update clause`. Follow it with `RETURN` or `FINISH`.

`next` is an unreserved keyword, so it remains usable as a variable, label or property name.

Compatibility

GQL CONFORMANCE

Implements the GQL `<next statement>` of the procedure body (ISO/IEC 39075:2024 §9.2), Feature GQ20, "Advanced linear composition with NEXT".

- **Supported** — chaining query statements, left-associative multi-chaining, the scope reset at the boundary, and a set-operation composite on either side.
- **Not supported** — the optional `YIELD` clause the standard allows between `NEXT` and the following statement (`NEXT YIELD a, b …`), which raises a syntax error. Project and rename in the preceding `RETURN` instead.

Track status in the GQL conformance matrix.

openCypher

openCypher has no `NEXT`. Composition there is always the `WITH` form shown in the tab set above, which AgensGraph accepts unchanged.

See also

- **`WITH`** — the Cypher analogue, inside one query
- **`LET`** GQL — add columns without a boundary
- **`FILTER`** GQL — the clause `NEXT` makes available first
- **`FINISH`** GQL
- **`RETURN`** — what the left query has to end with
- GQL conformance matrix

© 2026 SKAI Worldwide Co., Ltd.

FINISH

GQL clause | Added in 2.18

Ends a query without projecting anything: the work runs, and the query returns an empty result.

Synopsis

```
FINISH
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

Every query has to end with `RETURN`, `FINISH` or a write clause. `FINISH` is the ending that says "produce no result": it takes no arguments, projects no columns and closes the statement.

- `FINISH` must be the **last** clause. Anything after it raises `FINISH must be at the end of the query`.
- After a read, `FINISH` discards the rows. The planner applies it as `LIMIT 0`, so the scan is opened but no row is fetched.
- After a write, the write still happens. `FINISH` only suppresses the projection.
- A query ending in `FINISH` returns an **empty result**: no columns and no rows. `psql` prints it as a `--` header followed by `(0 rows)`, and `\gdesc` reports `The command has no result, or the result has no columns.`
- A write that ends in `FINISH` reports its ordinary command tag instead of a result, such as `UPDATE 1` for one created vertex.

Examples

Every example runs against **the example dataset**.

Run a read for its plan and nothing else. There is no Cypher spelling of this. A Cypher query with no `RETURN` and no write clause is rejected:

```
MATCH (m:movie) FINISH;
```

```
--
(0 rows)
```

Perform a write without reading the result back

`CREATE … FINISH` is the form to reach for when you do not want the created elements shipped to the client:

```
CREATE (:movie {title: 'Tenet', year: 2020, genre: 'Action'}) FINISH;
```

```
UPDATE 1
```

The vertex is there:

```
MATCH (m:movie {title: 'Tenet'}) RETURN m.title AS film, m.year AS year;
```

```
  film   | year
---------+------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Tenet" | 2020
(1 row)
```

Confirm that a read costs nothing after FINISH

The plan shows the `LIMIT 0` the clause compiles to, above the scan it never draws from:

```
EXPLAIN (COSTS OFF) MATCH (m:movie) FINISH;
```

```
        QUERY PLAN
---------------------------
 Limit
   ->  Seq Scan on movie m
(2 rows)
```

A write plan keeps its write node, because the write is not optional:

```
EXPLAIN (COSTS OFF) CREATE (:movie {title: 'Tenet'}) FINISH;
```

```
  QUERY PLAN
--------------
 Graph Create
   ->  Result
(2 rows)
```

End a sorted query that projects nothing

`ORDER BY`, `SKIP`, `OFFSET` and `LIMIT` are clauses in their own right, but none of them can end a query. `FINISH` closes a pipeline that ends on one:

```
MATCH (m:movie) ORDER BY m.year LIMIT 3 FINISH;
```

```
--
(0 rows)
```

Without the `FINISH` the same query raises `Cypher query must end with RETURN, FINISH or update clause`.

Notes

`FINISH` is not accepted where a result is required. A `CALL { }` body must project something, so a body ending in `FINISH` raises `CALL subquery must RETURN at least one variable`. A query ending in `FINISH` cannot be the left operand of `NEXT` either. That raises `FINISH cannot be followed by NEXT`.

> **NOTE**
>
> `finish` became a reserved keyword in 2.18. It is still accepted unquoted as a label name in a pattern, `MATCH (n:finish)`, which is what an upgraded database needs. It is not accepted as a variable name, a `CREATE VLABEL` name or a SQL identifier; write `"finish"` there, or rename. Property keys and `RETURN` aliases are unaffected.

Compatibility

> **GQL CONFORMANCE**

Implements `FINISH` of the GQL primitive result statement (ISO/IEC 39075:2024 §14.10). The standard specifies that `FINISH` has "the empty type" and sets the execution outcome to "a successful outcome with an omitted result", which is what the empty, column-less result above represents.

- **Supported** — `FINISH` as the terminal statement of a read query and of a data-modifying query, in which case the standard makes it implicit anyway (§13.1).
- **Not supported** — `FINISH` as a branch of a set operation, which the standard also forbids (§14.2, Syntax Rule 5). AgensGraph rejects it as a column-count mismatch: `each UNION query must have the same number of columns`.

Track status in the **GQL conformance matrix**.

openCypher

openCypher has no `FINISH`. There, a read query must end with `RETURN`, and a write query may end with its write clause. AgensGraph accepts that second form too, so `FINISH` adds the read case rather than replacing anything.

See also

- **`RETURN`** — the other way to end a query
- **`ORDER BY`** and **`SKIP`** and **`LIMIT`** — clauses that need a terminator
- **`NEXT`** GQL — the one clause `FINISH` may not precede
- **`INSERT`** GQL — a write worth ending with `FINISH`
- GQL conformance matrix

UNION

GQL clause

Combines the results of two or more queries into one. `INTERSECT` and `EXCEPT` are the same construct with the other two set operations.

Synopsis

```
query { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] query [ ... ]
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

A set operator joins two complete queries, each with its own `RETURN`, into a single result:

Operator	Result
`UNION`	Rows in either query.
`INTERSECT`	Rows in both queries.
`EXCEPT`	Rows in the first query and not in the second.

© 2026 SKAI Worldwide Co., Ltd.

Rules that apply to all three:

- Every query must return the **same number of columns**, and corresponding columns must be comparable. Their names need not match; the result takes the names from the first query.
- `DISTINCT` is the default: duplicate rows are removed. `ALL` keeps them. `UNION` and `UNION DISTINCT` mean the same thing.
- Operators **chain**, and a chain is evaluated left to right.
- Row order is not defined. An `ORDER BY` written after a set operation does **not** sort the combined result. See **Notes** for what does.
- Each query must be a complete query ending in `RETURN`. An arm cannot end in a write clause or a bare `MATCH`.

Parameters

query	A complete query ending in a `RETURN` with the same number of columns as every other query in the chain.
ALL	Keep duplicate rows.
DISTINCT	Remove duplicate rows. This is the default.

Examples

Every example runs against **the example dataset**.

Combine two result sets

Everyone connected to `Inception`, whichever way they are connected:

```
MATCH (p:person)-[:directed]->(:movie {title: 'Inception'}) RETURN p.name AS name
UNION
MATCH (p:person)-[:acted_in]->(:movie {title: 'Inception'}) RETURN p.name AS name;
```

```
        name
--------------------
 "Christopher Nolan"
 "Leonardo DiCaprio"
(2 rows)
```

Keep or drop the duplicates

Both arms here return the same 18 genres, one per film. `UNION` reduces them to three:

```
MATCH (m:movie) RETURN m.genre AS genre
UNION
MATCH (m:movie) RETURN m.genre AS genre;
```

```
    genre
-------------
 "Action"
 "Sci-Fi"
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Adventure"
(3 rows)
```

`UNION DISTINCT` is the same statement written out in full. `UNION ALL` keeps every row from both arms:

```
MATCH (m:movie {genre: 'Adventure'}) RETURN m.title AS film
UNION ALL
MATCH (m:movie) WHERE m.year = 2021 RETURN m.title AS film;
```

```
            film
-----------------------------
 "Jurassic Park"
 "The Matrix Resurrections"
(2 rows)
```

Those two rows are distinct anyway, so `ALL` shows only in its cost. It skips the deduplication.

Take the rows in both queries

```
MATCH (m:movie {genre: 'Action'}) RETURN m.title AS film
INTERSECT
MATCH (m:movie) WHERE m.year < 1995 RETURN m.title AS film;
```

```
             film
-------------------------------
 "The Terminator"
 "Terminator 2: Judgment Day"
(2 rows)
```

Subtract one query from another

The people who directed something but never acted:

```
MATCH (p:person)-[:directed]->(m:movie) RETURN p.name AS name
EXCEPT
MATCH (p:person)-[:acted_in]->(m:movie) RETURN p.name AS name;
```

```
        name
--------------------
 "Andrew Niccol"
 "Neill Blomkamp"
 "Lana Wachowski"
 "Lilly Wachowski"
 "Joss Whedon"
 "James Cameron"
 "Luc Besson"
 "Denis Villeneuve"
 "Christopher Nolan"
 "Paul Verhoeven"
 "Ridley Scott"
 "Steven Spielberg"
(12 rows)
```

The rows come back in no particular order.

© 2026 SKAI Worldwide Co., Ltd.

Sort and page the combined result

Put the set operation in a `NEXT` chain and sort the second half of the chain, which receives the combined rows:

```
MATCH (m:movie) WHERE m.genre = 'Adventure' RETURN m.title AS film
UNION
MATCH (m:movie) WHERE m.genre = 'Sci-Fi' RETURN m.title AS film
NEXT ORDER BY film LIMIT 3 RETURN film;
```

```
       film
--------------------
 "Avatar"
 "Blade Runner"
 "Blade Runner 2049"
(3 rows)
```

A `CALL { subquery }` wrapper does the same job:

```
CALL { MATCH (m:movie) WHERE m.genre = 'Adventure' RETURN m.title AS film
       UNION
       MATCH (m:movie) WHERE m.genre = 'Sci-Fi' RETURN m.title AS film }
RETURN film ORDER BY film LIMIT 3;
```

```
       film
--------------------
 "Avatar"
 "Blade Runner"
 "Blade Runner 2049"
(3 rows)
```

Reject a mismatched column count

```
MATCH (m:movie) RETURN m.title AS t
UNION
MATCH (p:person) RETURN p.name AS n, p.born AS b;
```

```
ERROR:  each UNION query must have the same number of columns
```

The message names `UNION` for all three operators.

Notes

An `ORDER BY`, `SKIP` or `LIMIT` written after a set operation attaches to the **last query in the chain**, not to the combined result. It therefore sorts or pages that one arm before the sets are combined. `LIMIT 1` here does not produce one row:

```
MATCH (m:movie) WHERE m.genre = 'Adventure' RETURN m.title AS film
UNION
MATCH (m:movie) WHERE m.genre = 'Sci-Fi' RETURN m.title AS film
LIMIT 1;
```

```
     film
----------------
 "Inception"
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Jurassic Park"
(2 rows)
```

One row survived the `LIMIT` in the Sci-Fi arm, which returns 10 films without it. The single Adventure film joined that row afterwards. Alias resolution follows the same rule, so a sort key must name a column of the last arm:

```
MATCH (m:movie) RETURN m.title AS first_col
UNION
MATCH (p:person) RETURN p.name AS second_col
ORDER BY first_col LIMIT 2;
```

```
ERROR:  variable does not exist
LINE 4: ORDER BY first_col LIMIT 2;
                ^
```

Use the `NEXT` or `CALL` form shown above whenever you need to sort or page the combined rows. In SQL, a `SELECT` over the set operation does it too:

```
SELECT * FROM (
  MATCH (m:movie) WHERE m.genre = 'Adventure' RETURN m.title AS film
  UNION
  MATCH (m:movie) WHERE m.genre = 'Sci-Fi' RETURN m.title AS film
) q ORDER BY film LIMIT 3;
```

```
        film
--------------------
 "Avatar"
 "Blade Runner"
 "Blade Runner 2049"
(3 rows)
```

Compatibility

Set operators (ISO/IEC 39075 §14.2)

The standard's `<set operator>` is `UNION [ <set quantifier> ] | EXCEPT [ <set quantifier> ] | INTERSECT [ <set quantifier> ]`, with `DISTINCT` implicit when no quantifier is given. AgensGraph implements all three operators and both quantifiers, on exactly those rules. openCypher has only `UNION` and `UNION ALL`, so `INTERSECT` and `EXCEPT` are available here and not there.

The standard's `OTHERWISE` conjunction, which takes the second query's rows only when the first returns none, is not implemented.

Sorting a combined result (ISO/IEC 39075 §14.9)

The standard applies an `<order by and page statement>` to the current working table, which after a composite query is the combined result. AgensGraph attaches it to the last query of the chain instead, as shown in **Notes**. Wrap the set operation in `NEXT`, in a `CALL { }`, or in a SQL `SELECT` to sort or page what the operation produced.

See also

- **`RETURN`** — every arm of a set operation ends with one
- **`NEXT`** `GQL` — chain a further query onto the combined rows
- **`CALL { subquery }`** — the other way to wrap a set operation
- **`ORDER BY`**, **`SKIP`** and **`LIMIT`** — what they attach to here

© 2026 SKAI Worldwide Co., Ltd.

- Combining SQL and Cypher — sorting a set operation from SQL

Subqueries

CALL

GQL clause | Added in 2.18

Runs a routine from inside a query pipeline. The routine is either a named function whose rows join onto the working table, or a subquery written inline.

Synopsis

[**OPTIONAL**] **CALL** function([argument [, argument] ...]) **YIELD** yield_list
[**OPTIONAL**] **CALL** [variable_scope] { subquery }

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`CALL` has two forms, and which one you get is decided by the token after the keyword:

- `CALL` followed by a **name** is the named-routine form. It invokes a function and adds the columns named in the mandatory `YIELD` list to the working row. See **`CALL … YIELD`**.
- `CALL` followed by `(` or `{` is the inline form. It runs the subquery between the braces, optionally once per input row. See **`CALL { subquery }`**.

`OPTIONAL` may prefix either form. It keeps an input row whose routine or subquery produced no rows, binding everything the call would have returned to null. See **`OPTIONAL CALL`**.

What CALL invokes

In GQL the thing `CALL` invokes is a procedure. AgensGraph has no procedure catalog of its own. `CALL … YIELD` resolves the name against PostgreSQL's ordinary functions, and a stock installation registers no procedures at all.

```
SELECT count(*) AS procedures FROM pg_proc WHERE prokind = 'p';
```

```
 procedures
------------
          0
(1 row)
```

So "call a procedure" means "call a function" here. Any function works, and its shape decides what `YIELD` can name:

Function shape	What `YIELD` sees
Scalar, e.g. `length(text)`	One column named after the function
Set-returning, e.g. `generate_series(int, int)`	One column named after the function, one row per element

© 2026 SKAI Worldwide Co., Ltd.

Function shape	What `YIELD` sees
Composite or `RETURNS TABLE`, e.g. `jsonb_each_text(jsonb)`	One column per output field, under the field names
Any of these, under the function's own name	The whole row, when no output column has that name; a function `RETURNS SETOF vertex` yields vertices this way

A yield item is one of these and nothing else. A name the function does not return is refused, even when a variable of that name is bound earlier in the query.

PostgreSQL's own `CALL` statement is a different thing: it is a top-level SQL statement, it requires a real procedure, and it is not a Cypher clause. The two do not overlap, and each rejects the other's target.

Parameters

function	Name of a function to invoke. Schema-qualify it if `search_path` does not reach it. A procedure is refused: `<name>() is a procedure`, with the hint `To call a procedure, use CALL`.
argument	A value expression, evaluated per input row. Arguments may reference variables bound earlier, which makes the call correlated. A property access produces `jsonb`, so cast it when the parameter is another type.
variable_scope	Which outer variables the inline subquery may see: `(v, w)` for those named, `(*)` for all of them, `()` for none. **Omitting it imports nothing**. See **`CALL { subquery }`**, which leads with that rule.
yield_list	`*`, or one or more of the function's output columns, each optionally renamed with `AS`. Mandatory. See **`CALL … YIELD`**.
subquery	A read-only query ending in `RETURN`. Write clauses are rejected.

Examples

Every example runs against **the example dataset**.

Call a set-returning function and join its rows onto the matched film:

```
MATCH (m:movie {title: 'The Matrix'})
CALL jsonb_array_elements_text(m.embedding) YIELD jsonb_array_elements_text AS d
RETURN d ORDER BY d;
```

```
       d
---------------
 "-0.07594558"
 "-0.11921061"
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "0.04081754"
 "0.29592122"
(4 rows)
```

Compute a value per row with a subquery

The inline form answers a question per input row, keeping the body out of the main pattern:

```
MATCH (m:movie {genre: 'Adventure'})
CALL (m) { MATCH (p:person)-[:acted_in]->(m) RETURN count(p) AS cast_size }
RETURN m.title AS film, cast_size;
```

```
      film       | cast_size
-----------------+-----------
 "Jurassic Park" | 1
(1 row)
```

Try to call a procedure

A procedure is the one routine kind `CALL … YIELD` will not take:

```
CREATE PROCEDURE noop() LANGUAGE sql AS 'SELECT 1';
```

```
CREATE PROCEDURE
```

```
MATCH (m:movie) CALL noop() YIELD * RETURN count(*);
```

```
ERROR:  noop() is a procedure
LINE 1: MATCH (m:movie) CALL noop() YIELD * RETURN count(*);
                            ^
HINT:  To call a procedure, use CALL.
```

And a function is the one routine kind the SQL `CALL` statement will not take:

```
CALL generate_series(1, 3);
```

```
ERROR:  generate_series(integer, integer) is not a procedure
LINE 1: CALL generate_series(1, 3);
             ^
HINT:  To call a function, use SELECT.
```

Notes

The lexer decides between the two forms by looking one token ahead. That keeps `call` usable as an ordinary identifier: `MATCH (call:movie) RETURN count(call)` still works, and so does the SQL statement `CALL myproc()`.

The named-routine form is a **middle** clause: it cannot open a query and cannot end one. The inline form may open a query, but only without a scope clause. A scope clause with nothing before it raises `CALL has no preceding clause to import variables from`.

Compatibility

© 2026 SKAI Worldwide Co., Ltd.

GQL CONFORMANCE

Implements the GQL **call procedure statement** and **procedure call** (ISO/IEC 39075:2024 §15.1) in the query position of §14.5, **call query statement**.

- **Supported** — both procedure-call forms the standard defines, the inline call of §15.2 and the named call of §15.3, each with the optional `OPTIONAL` prefix.
- **Not supported** — the **call data-modifying procedure statement** (§13.6): a called routine or subquery may not write to the graph. Also not supported is a catalog of GQL procedures — §17.5 procedure references resolve to PostgreSQL functions, and the standard's `QUERY PROCEDURE` indication has no counterpart.
- **Deviation** — the standard makes the `YIELD` clause optional on a named call (§15.3). AgensGraph requires it.

Track status in the **GQL conformance matrix**.

openCypher

openCypher's `CALL db.labels()` style of procedure call maps onto the named form, but the names differ: there is no `db.*` or `dbms.*` namespace here. Use the PostgreSQL catalogs and the `\dGv` / `\dGe` meta-commands instead, or read **ag_label** directly.

Drop the routines the examples created:

```
DROP PROCEDURE noop();
```

See also

- **`CALL … YIELD`** GQL — the named-routine form in detail
- **`CALL { subquery }`** GQL — the inline form, and its import list
- **`OPTIONAL CALL`** GQL — the input-preserving form
- **Subquery value expressions** GQL — a subquery in an expression rather than a clause
- GQL conformance matrix

CALL … YIELD

GQL clause | Added in 2.18

Invokes a named function and adds the columns it produces to the working row, joining its rows onto the pipeline.

Synopsis

```
[ OPTIONAL ] CALL function( [ argument [, argument ] ... ] ) YIELD *
[ OPTIONAL ] CALL function( [ argument [, argument ] ... ] ) YIELD column [ AS alias ] [, column ] ...
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See **Syntax conventions**.

Description

The clause becomes a function scan joined onto the preceding clause's rows, like a lateral `SELECT * FROM function(arguments)`, and the `YIELD` list picks from that result. A yield item is resolved against the function's result and nothing else, so a name the function does not return is an error even when the query binds a variable of that name.

- `YIELD` is **mandatory**. `CALL f(x)` with no `YIELD` is a syntax error.

© 2026 SKAI Worldwide Co., Ltd.

- The clause can neither open a query nor end one. `CALL` at the start of a statement is read as PostgreSQL's SQL `CALL`, so a leading `CALL f() YIELD …` is a syntax error.
- If the arguments reference variables bound earlier, the call is **correlated** and runs once per input row. With constant arguments the function runs once and its rows cross-join onto the pipeline.
- Yielded columns are ordinary bindings afterwards: filter them, aggregate them, order by them, or feed them to a following `MATCH`.
- An input row for which the function yields no rows is **dropped**, exactly as an inner join drops it. Use `OPTIONAL CALL` to keep it.

Parameters

function	Name of the function to invoke, schema-qualified if `search_path` does not reach it. The parentheses are required even with no arguments. A procedure is refused with `<name>() is a procedure`.
argument	A value expression evaluated per input row. A property access produces `jsonb`, so a function whose parameter is another type needs an explicit cast.
column	An output column of the function. A scalar or single-column set-returning function has one, named after the function itself; a composite or `RETURNS TABLE` function has one per output field, under the field names. `YIELD *` takes all of them. The function's own name, when no column has it, stands for the whole row, which is how a function declared `RETURNS SETOF vertex` or `RETURNS SETOF edge` is yielded as vertices or edges. Any other name is refused with `function f has no output column "x"`.
alias	New name for the yielded column. Required when the column's own name is already bound in the query.

Examples

Every example runs against **the example dataset**.

Expand one film's properties into rows. `jsonb_each_text` returns two columns, and `YIELD` names both:

```
MATCH (m:movie {title: 'The Matrix'})
CALL jsonb_each_text(jsonb_build_object('title', m.title, 'year', m.year, 'genre', m.genre))
     YIELD key, value
RETURN key AS property, value ORDER BY property;
```

```
 property |    value
----------+--------------
 "genre"  | "Action"
 "title"  | "The Matrix"
 "year"   | "1999"
(3 rows)
```


© 2026 SKAI Worldwide Co., Ltd.

Rename a yielded column

`AS` renames it for every clause that follows:

```
MATCH (m:movie {title: 'The Matrix'})
CALL jsonb_each_text(properties(m)) YIELD key AS property
RETURN property ORDER BY property;
```

```
  property
-------------
 "embedding"
 "genre"
 "plot"
 "title"
 "year"
(5 rows)
```

Yielding a subset is allowed. `value` is not bound here.

Take every column with YIELD *

A single-column set-returning function names its column after itself:

```
MATCH (m:movie) WHERE m.genre = 'Adventure'
CALL generate_series(1, 2) YIELD *
RETURN m.title AS film, generate_series AS n ORDER BY n;
```

```
      film       | n
-----------------+---
 "Jurassic Park" | 1
 "Jurassic Park" | 2
(2 rows)
```

The arguments are constants, so `generate_series` runs once and its two rows cross-join onto the film.

Correlate the call with the matched row

Pass a value from the row and the function runs per row. `m.year` is `jsonb`, so it needs a cast:

```
MATCH (m:movie {title: 'The Matrix'})
CALL generate_series(1, (m.year)::int) YIELD generate_series AS n
RETURN count(n) AS years;
```

```
 years
-------
 1999
(1 row)
```

Without the cast the function cannot be resolved:

```
MATCH (m:movie {title: 'The Matrix'})
CALL generate_series(1, m.year) YIELD generate_series AS n
RETURN count(n);
```

```
ERROR:  function generate_series(integer, jsonb) does not exist
LINE 2: CALL generate_series(1, m.year) YIELD generate_series AS n
```


© 2026 SKAI Worldwide Co., Ltd.

```
       ^
HINT:  No function matches the given name and argument types. You might need to add explicit t
ype casts.
```

Chain two calls

The second call may correlate on the first one's yielded column:

```
MATCH (m:movie {title: 'The Matrix'})
CALL generate_series(1, 2) YIELD generate_series AS n
CALL generate_series(1, n) YIELD generate_series AS k
RETURN n, k ORDER BY n, k;
```

```
 n | k
---+---
 1 | 1
 2 | 1
 2 | 2
(3 rows)
```

Yield the whole row

When no output column carries the function's own name, that name stands for the whole row. `jsonb_each_text` returns `(key, value)` rows:

```
MATCH (m:movie {title: 'The Matrix'})
CALL jsonb_each_text(jsonb_build_object('title', m.title, 'year', m.year))
     YIELD jsonb_each_text AS pair
RETURN pair ORDER BY pair;
```

```
                  pair
-----------------------------------------
 {"key": "title", "value": "The Matrix"}
 {"key": "year", "value": "1999"}
(2 rows)
```

A function declared `RETURNS SETOF vertex` yields its vertices the same way, under its own name.

Notes

The plan shows which of the two execution shapes you got. A correlated call is a `Nested Loop` whose inner side is the function scan. An uncorrelated call puts the function scan on the outer side and materializes the graph scan:

```
EXPLAIN (COSTS OFF)
MATCH (m:movie) CALL generate_series(1, (m.year)::int) YIELD generate_series AS n
RETURN m.title, n;
```

```
              QUERY PLAN
---------------------------------------
 Nested Loop
   ->  Seq Scan on movie m
   ->  Function Scan on generate_series
(3 rows)
```

A yielded name that collides with a binding already in scope is rejected rather than shadowing it. The error is `variable "m" yielded by CALL is already bound in the outer query`. Rename it with `AS`.

© 2026 SKAI Worldwide Co., Ltd.

The reverse is rejected too: a yield item that is not an output column of the function is refused even when a variable of that name is bound earlier. The earlier binding is never what `YIELD` means.

```
MATCH (m:movie {title: 'The Matrix'})
CALL generate_series(1, 2) YIELD m AS n
RETURN n;
```

```
ERROR:  function generate_series has no output column "m"
LINE 2: CALL generate_series(1, 2) YIELD m AS n
                                         ^
```

NOTE

A function returning the pseudo-type `record` cannot be called: the clause has nowhere to put the column definition list PostgreSQL requires, so `CALL jsonb_to_recordset(…) YIELD *` raises `a column definition list is required for functions returning "record"`. Wrap such a call in a SQL subquery instead.

Compatibility

GQL CONFORMANCE

Implements the GQL named procedure call (ISO/IEC 39075:2024 §15.3) with the yield clause of §16.14, under the call procedure statement of §15.1.

- **Supported** — argument lists, correlated arguments, the `YIELD *` and `YIELD column [AS alias]` forms, the rule of §16.14 Syntax Rule 7 that a yield item names a column of the procedure's result, the duplicate-alias rejection of Syntax Rule 6, and the `OPTIONAL` prefix.
- **Not supported** — calling a GQL procedure. §15.3 resolves a procedure reference against the catalog described in §17.5, which AgensGraph does not have; names resolve to PostgreSQL functions instead.
- **Deviation** — §15.3 makes the `YIELD` clause optional. AgensGraph requires it, because the clause has to contribute columns to the working table.

Track status in the GQL conformance matrix.

openCypher

openCypher writes the same clause but resolves the name against a procedure registry (`db.labels()`, `dbms.components()`), and allows `CALL proc()` with no `YIELD` as a standalone statement. Neither the registry nor the standalone form exists here: names resolve to PostgreSQL functions, and `YIELD` is required.

See also

- **`CALL`** GQL — both call forms, and what a routine is here
- **`CALL { subquery }`** GQL — the inline form
- **`OPTIONAL CALL`** GQL — keep a row the function yielded nothing for
- GQL conformance matrix

© 2026 SKAI Worldwide Co., Ltd.

CALL { subquery }

GQL clause | Added in 2.18

Runs a read-only subquery as a clause in the pipeline, once per input row when the subquery imports a variable from the enclosing query.

> IMPORTANT
>
> **A bare `CALL { … }` imports nothing.** A pattern inside the braces declares its variable names, so an outer variable's name written in the body binds a fresh, unconstrained variable of the same name rather than referring to the outer one. Name the variables you need: `CALL (v) { … }`, `CALL (*) { … }` or `CALL { WITH v … }`.

Synopsis

```
[ OPTIONAL ] CALL { subquery }
[ OPTIONAL ] CALL ( [ variable [, variable ] ... ] ) { subquery }
[ OPTIONAL ] CALL ( * ) { subquery }
[ OPTIONAL ] CALL { WITH variable [, variable ] ... subquery }
```

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`CALL { … }` evaluates the subquery and joins its result onto the working table. The import list decides entirely what the body can see.

- With an import list, the body runs **per input row** with those variables bound, and the rows it returns are joined to that row.
- With no imports the body is **uncorrelated**: it runs once and its result cross-joins onto every input row.
- The body is read-only and must end with `RETURN`. Write clauses are rejected at parse time, and a body ending in `FINISH` raises `CALL subquery must RETURN at least one variable`.
- The names the body returns become bindings in the enclosing query. A name already bound outside is rejected rather than shadowed. Inside the body the imported variables keep their names as well: a `LET`, `UNWIND`, `FOR` or renaming `WITH` that would bind one of them again is refused.
- An input row for which the body returns nothing is dropped. Use `OPTIONAL CALL` to keep it.

The forms that import

Five spellings are accepted. The first two import nothing. The difference between them and the last three is the whole point of the warning above:

Form	Body sees
`CALL { … }`	Nothing
`CALL () { … }`	Nothing
`CALL (v, w) { … }`	`v` and `w`
`CALL (*) { … }`	Every variable in scope

© 2026 SKAI Worldwide Co., Ltd.

Form	Body sees
`CALL { WITH v, w … }`	`v` and `w`

A scope clause and an importing `WITH` may be combined. `CALL (v) { WITH w … }` imports both. Three near-misses are syntax errors rather than alternatives: `CALL (*) { WITH v … }`, `CALL (*, v) { … }`, and any importing `WITH` that is not a plain list of names (`WITH *` and `WITH v AS x` are both refused).

Parameters

variable	A variable bound in the enclosing query, to be made visible inside the braces. One that does not exist raises `variable "<name>" to import into CALL does not exist`. A scope clause with no preceding clause raises `CALL has no preceding clause to import variables from`.
subquery	A read-only query ending in `RETURN`. `MATCH`, `OPTIONAL MATCH`, `WITH`, `LET`, `UNWIND`, `FOR`, `LOAD FROM`, a nested `CALL`, `DISTINCT`, `ORDER BY`, `SKIP` / `LIMIT` and the set operators are all allowed. **`NEXT`** is not.

Examples

Every example runs against **the example dataset**.

Why the import list matters

The subquery form is made for a question like this one: for each film that somebody directed, how large was the cast? Written without an import list, it answers a different question:

```
MATCH (d:person)-[:directed]->(m:movie)
CALL { MATCH (a:person)-[:acted_in]->(m) RETURN count(a) AS cast_size }
RETURN m.title AS film, cast_size ORDER BY film LIMIT 4;
```

```
        film         | cast_size
---------------------+-----------
 "Avatar"            | 25
 "Blade Runner"      | 25
 "Blade Runner 2049" | 25
 "Elysium"           | 25
(4 rows)
```

Every film has the same cast size, and 25 is the number of `acted_in` edges in the whole graph:

```
MATCH ()-[r:acted_in]->() RETURN count(r) AS acted_in_edges;
```

```
 acted_in_edges
----------------
 25
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

The `m` inside the braces is not the `m` matched outside. It is a brand-new variable that the body's own pattern declares, constrained only by the `acted_in` edge. So the body counts every `acted_in` edge in the graph, once and uncorrelated, and then cross-joins that one number onto every row.

Naming the import asks the question that was intended:

```
MATCH (d:person)-[:directed]->(m:movie)
CALL (m) { MATCH (a:person)-[:acted_in]->(m) RETURN count(a) AS cast_size }
RETURN m.title AS film, cast_size ORDER BY film LIMIT 4;
```

```
        film         | cast_size
---------------------+-----------
 "Avatar"            | 1
 "Blade Runner"      | 1
 "Blade Runner 2049" | 2
 "Elysium"           | 1
(4 rows)
```

`CALL (*) { … }` and `CALL { WITH m … }` give the same numbers.

Nothing is reported because only expressions reference a variable, while patterns declare one. Move the same mistake into a `WHERE` and it is caught:

```
MATCH (d:person)-[:directed]->(m:movie)
CALL { MATCH (a:person) WHERE a.name = m.title RETURN count(a) AS n }
RETURN n;
```

```
ERROR:  variable does not exist
LINE 2: CALL { MATCH (a:person) WHERE a.name = m.title RETURN count(...
                                               ^
```

So the silence is specific to a name reused in a pattern. That is the common case, and the reason to write the import list every time.

Aggregate per row without collapsing the pipeline

An aggregating body returns exactly one row for every input row, even when the value it computes is zero, so the outer row count is unchanged:

```
MATCH (p:person) WHERE p.name IN ['Keanu Reeves', 'Christopher Nolan']
CALL (p) { MATCH (p)-[:acted_in]->(m:movie) RETURN count(m) AS films }
RETURN p.name AS person, films ORDER BY person;
```

```
       person        | films
---------------------+-------
 "Christopher Nolan" | 0
 "Keanu Reeves"      | 4
(2 rows)
```

Compute one number once

An empty scope clause states plainly that the body is uncorrelated. Write it when the body is a constant of the query:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person {name: 'Keanu Reeves'})
CALL () { MATCH (m:movie) RETURN count(m) AS movies }
RETURN p.name AS person, movies;
```

```
     person     | movies
----------------+--------
 "Keanu Reeves" | 18
(1 row)
```

Import several variables

Every named variable is visible, and the body may use only some of them:

```
MATCH (p:person {name: 'Keanu Reeves'}), (q:person {name: 'Carrie-Anne Moss'})
CALL (p, q) { MATCH (p)-[:acted_in]->(m:movie)<-[:acted_in]-(q) RETURN count(m) AS shared }
RETURN shared;
```

```
 shared
--------
 2
(1 row)
```

Notes

A correlated body is not a per-row subquery in the plan. The planner pulls it up, so the query above plans as the same join an inline pattern would produce. That is a `Hash Join` over `person`, `acted_in` and `movie`, identical to `MATCH (p:person)-[:acted_in]->(m:movie)`. Only a body the planner cannot flatten, such as an aggregating one, stays a per-row lateral join with an `Aggregate` under a `Nested Loop`.

Write clauses are refused as a syntax error at the write keyword, one for each kind: `CREATE`, `INSERT`, `MERGE`, `SET`, `REMOVE`, `DELETE` and `DETACH DELETE`.

A body that returns a name the outer query already binds raises `variable "<name>" returned by the CALL subquery is already bound in the outer query`. It would shadow the outer binding for everything that follows. Alias it in the body's `RETURN`.

Compatibility

GQL CONFORMANCE

Implements the GQL inline procedure call (ISO/IEC 39075:2024 §15.2) under the call procedure statement of §15.1, in the query position of §14.5.

- **Supported** — the variable scope clause with a binding-variable list and in its empty `()` form, nesting, and the `OPTIONAL` prefix.
- **Not supported** — a data-modifying body (§13.6, call data-modifying procedure statement).
- **Deviation, and the one to know** — §15.2 Syntax Rule 3 says that when the variable scope clause is absent, the call is rewritten with a scope clause listing **every** field of the incoming working record. Under the standard a bare `CALL { … }` therefore imports everything, exactly like `CALL (*) { … }`. AgensGraph treats an absent scope clause as importing nothing, like `CALL () { … }`.
- **Extension** — `(*)` is not in the standard's binding variable reference list, and neither is the importing `WITH`. Both come from openCypher.

Track status in the GQL conformance matrix.

© 2026 SKAI Worldwide Co., Ltd.

openCypher

openCypher requires the same import list, but reports its absence: referring to an undeclared outer variable inside `CALL { }` is an error there, not a silent rebinding. Code moved from openCypher keeps working, since `CALL (v) { }`, `CALL (*) { }` and `CALL { WITH v … }` all mean what they do there. A bare `CALL { }` that openCypher rejected will run here and answer differently.

See also

- **`CALL`** GQL — both call forms
- **`OPTIONAL CALL`** GQL — keep a row the body returned nothing for
- **`CALL … YIELD`** GQL — call a named function instead
- Subquery value expressions GQL — a subquery in an expression
- GQL conformance matrix

OPTIONAL CALL

GQL clause | Added in 2.18

Keeps an input row whose call produced no rows, binding everything the call would have returned to null. It is the `CALL` counterpart of `OPTIONAL MATCH`.

Synopsis

OPTIONAL CALL [variable_scope] { subquery }
OPTIONAL CALL function([argument [, argument] ...]) **YIELD** yield_list

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`OPTIONAL` applies to both forms of **`CALL`**: the inline subquery and the named function. A plain `CALL` joins the call's rows onto the pipeline, so an input row that produced none drops out. `OPTIONAL CALL` lowers to a **left join** instead, which preserves that row.

- Preservation is decided **per input row**. A correlated call that finds nothing for one row nulls only that row and leaves the others alone.
- Every variable the call would have bound is nulled together, whatever its type: vertex, edge, path, list, map or scalar.
- The binding is a real SQL null, not a jsonb `null`, so `IS NULL` is the way to test it.
- `OPTIONAL` preserves rows; it does not invent them. An empty input still produces no output.
- A preserved row is an ordinary row afterwards, so a later clause multiplies it like any other.

Parameters

variable_scope	Which outer variables the subquery may see, exactly as for **`CALL { subquery }`**: `(v, w)`, `(*)`, `()`, or an importing `WITH` inside the braces. The import rule applies unchanged. A bare `OPTIONAL CALL { … }` imports nothing.

© 2026 SKAI Worldwide Co., Ltd.

subquery	A read-only query ending in `RETURN`.
function, argument, yield_list	As for **`CALL … YIELD`**. `YIELD` is still mandatory.

Examples

Every example runs against **the example dataset**.

Three people: Keanu Reeves has four acting credits, Ryan Gosling one, Christopher Nolan none. `OPTIONAL CALL` keeps the director:

```
MATCH (p:person) WHERE p.name IN ['Keanu Reeves', 'Ryan Gosling', 'Christopher Nolan']
OPTIONAL CALL (p) { MATCH (p)-[:acted_in]->(m:movie) RETURN m.title AS film }
RETURN p.name AS person, film ORDER BY person, film;
```

```
       person        |            film
---------------------+-----------------------------
 "Christopher Nolan" |
 "Keanu Reeves"      | "The Matrix"
 "Keanu Reeves"      | "The Matrix Reloaded"
 "Keanu Reeves"      | "The Matrix Resurrections"
 "Keanu Reeves"      | "The Matrix Revolutions"
 "Ryan Gosling"      | "Blade Runner 2049"
(6 rows)
```

Drop the `OPTIONAL` and the same body loses that row entirely, while the other two people are unaffected:

```
MATCH (p:person) WHERE p.name IN ['Keanu Reeves', 'Ryan Gosling', 'Christopher Nolan']
CALL (p) { MATCH (p)-[:acted_in]->(m:movie) RETURN m.title AS film }
RETURN p.name AS person, film ORDER BY person, film;
```

```
     person     |            film
----------------+-----------------------------
 "Keanu Reeves" | "The Matrix"
 "Keanu Reeves" | "The Matrix Reloaded"
 "Keanu Reeves" | "The Matrix Resurrections"
 "Keanu Reeves" | "The Matrix Revolutions"
 "Ryan Gosling" | "Blade Runner 2049"
(5 rows)
```

Four rows, one row and one preserved row: the multiplicity is per input row, not a property of the query.

Preserve a row a function yielded nothing for

`OPTIONAL` sits in the same place on the named form, where it covers a function that returns an empty set for some rows:

```
MATCH (m:movie) WHERE m.title IN ['The Matrix', 'Avatar']
OPTIONAL CALL generate_series(1, CASE WHEN m.title = 'Avatar' THEN 0 ELSE 2 END)
         YIELD generate_series AS n
RETURN m.title AS film, n ORDER BY film, n;
```


© 2026 SKAI Worldwide Co., Ltd.

```
     film     | n
--------------+---
 "Avatar"     |
 "The Matrix" | 1
 "The Matrix" | 2
(3 rows)
```

`generate_series(1, 0)` is empty, so Avatar's row survives with `n` null. Without `OPTIONAL` that row is gone and only The Matrix's two rows remain.

Test a preserved binding

Every variable the body would have returned is null on a preserved row, and it is a SQL null:

```
MATCH (p:person {name: 'Christopher Nolan'})
OPTIONAL CALL (p) { MATCH (p)-[:acted_in]->(m:movie) RETURN m.title AS film, m.year AS year }
RETURN film IS NULL AS film_is_null, year IS NULL AS year_is_null;
```

```
 film_is_null | year_is_null
--------------+--------------
 t            | t
(1 row)
```

Confirm that no row is invented

`OPTIONAL` preserves input rows. With no input row there is nothing to preserve:

```
MATCH (p:person {name: 'Nobody'})
OPTIONAL CALL (p) { MATCH (p)-[:acted_in]->(m:movie) RETURN m.title AS film }
RETURN p.name AS person, film;
```

```
 person | film
--------+------
(0 rows)
```

Notes

The plan states the mechanism outright. The `generate_series` scan is the nullable side of a `Nested Loop Left Join`. That join preserves Avatar's row:

```
EXPLAIN (COSTS OFF)
MATCH (m:movie) WHERE m.title IN ['The Matrix', 'Avatar']
OPTIONAL CALL generate_series(1, CASE WHEN m.title = 'Avatar' THEN 0 ELSE 2 END)
         YIELD generate_series AS n
RETURN m.title AS film, n;
```

```
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Nested Loop Left Join
   ->  Seq Scan on movie m
         Filter: (properties.'title' = ANY ('{"\"The Matrix\"","\"Avatar\""}'::jsonb[]))
   ->  Function Scan on generate_series
(4 rows)
```


© 2026 SKAI Worldwide Co., Ltd.

Some bodies always yield at least one row, and for those `OPTIONAL` changes nothing. An ungrouped aggregate is the common case: it always produces one row, so a person with no credits gets `0`, not null. Compare the `films` column in **`CALL { subquery }`**.

`OPTIONAL` relaxes nothing else. A write clause in the body is still a syntax error, a body that returns an outer name is still rejected, and a bare `OPTIONAL CALL { … }` still imports nothing.

Compatibility

GQL CONFORMANCE

Implements the `OPTIONAL` prefix of the GQL call procedure statement (ISO/IEC 39075:2024 §15.1), which covers both the inline procedure call of §15.2 and the named procedure call of §15.3.

- **Supported** — `OPTIONAL` on both call forms, with per-input-row preservation and null binding of every returned variable.
- **Not supported** — `OPTIONAL` on a data-modifying call (§13.6), since a call cannot write to the graph at all.

Track status in the GQL conformance matrix.

openCypher

`OPTIONAL CALL` is a GQL construct; openCypher has only `OPTIONAL MATCH`. The two agree wherever both can express the same thing, because the standard defines `OPTIONAL MATCH p` as `OPTIONAL CALL (imports) { MATCH p RETURN vars }`. So `OPTIONAL CALL` is the more general primitive, not a second mechanism.

See also

- **`OPTIONAL MATCH`** — the same preservation, on a pattern
- **`CALL`** GQL — both call forms
- **`CALL { subquery }`** GQL — the inline form and its import list
- **`CALL … YIELD`** GQL — the named-function form
- GQL conformance matrix

Subquery value expressions

GQL clause · Added in 2.18

Six expressions that answer a question with a subquery instead of a clause, usable anywhere a value is: `RETURN`, `WHERE`, `FILTER`, `WITH` and SQL.

Synopsis

```
EXISTS { body }
COUNT { body }
COLLECT { body }
ARRAY { body }
VALUE { body }
expression [ NOT ] IN { body }
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

© 2026 SKAI Worldwide Co., Ltd.

Description

Each expression runs its body and reduces the result to one value. The body is correlated: it sees the variables in scope where the expression is written, with no import list to declare, unlike **`CALL { subquery }`**. Those variables stay in scope through the whole body, so a clause inside it cannot bind one of their names again: a `LET`, `UNWIND` or `FOR` on such a name, a `WITH` that renames a value to it, or a `CALL { }` that returns it is refused. `MATCH (n)` on the name refers to the outer node, and `WITH n` passes it on unchanged.

This table says what each one returns and what it does when the body is empty. Five of the six answer an empty body with a value of their own type rather than with null: a false, a zero, an empty list. `VALUE { }` has no such value to fall back on, so it is the one that gives null.

Expression	Returns	Body empty
`EXISTS { }`	Boolean	`false`
`COUNT { }`	Row count, a `bigint`	`0`
`COLLECT { }`	List of the body's single column	`[]`
`ARRAY { }`	Same as `COLLECT`	`[]`
`VALUE { }`	The body's single value, in its own type	`null`
`x IN { }`	Boolean	`false`

What a body may be

Four kinds of body are accepted, and which ones are allowed depends on the expression:

Body	`EXISTS` / `COUNT`	`COLLECT` / `ARRAY` / `VALUE` / `IN`
A bare pattern, with an optional `WHERE`	Yes	No
A `MATCH` statement, `OPTIONAL MATCH` included	Yes	No
A GQL read statement ending in `RETURN`	Yes	Yes
A SQL `SELECT`	Yes	Yes

- **Write clauses are rejected.** `CREATE`, `INSERT`, `MERGE`, `SET`, `REMOVE`, `DELETE` and `DETACH DELETE` each raise a syntax error at the keyword.
- A read-statement body may use `DISTINCT`, `ORDER BY`, `SKIP`, `LIMIT`, the set operators and a nested `CALL { }`. It may not use **`NEXT`**.
- A bare-pattern body may not name the path (`EXISTS { p = (a)-[]->(b) }` is a syntax error), but `shortestpath()` and a variable-length edge are both fine.

© 2026 SKAI Worldwide Co., Ltd.

- A **SQL `SELECT` body is uncorrelated**: it cannot see the enclosing query's variables. Referring to one raises `missing FROM-clause entry for table "<name>"`.

Parameters

body	The subquery, in one of the four shapes above. `COLLECT`, `ARRAY`, `VALUE` and `IN` need it to produce exactly one column.
expression	For `x IN { }`, the value to look for. A node, relationship or graphid is matched by identity, as in `x IN [list]`, and an element on the left against a body of plain values is a type error. Any other value and a Cypher body's column meet as `jsonb`. A SQL body keeps its own column type, so unwrap the property first: `m.title #>> '{}'` yields `text`.

Examples

Every example runs against **the example dataset**.

Test whether a pattern exists

`EXISTS { }` short-circuits: the planner turns it into a semi-join or an `EXISTS` subplan rather than counting matches. A bare pattern is the shortest body, and it takes a `WHERE`:

```
MATCH (p:person)
WHERE EXISTS { (p)-[:directed]->(m:movie) WHERE m.year > 2015 }
RETURN p.name AS director ORDER BY director;
```

```
      director
--------------------
 "Denis Villeneuve"
 "Lana Wachowski"
(2 rows)
```

In `RETURN` it is a boolean column, `false` rather than null when nothing matches:

```
MATCH (p:person {name: 'Keanu Reeves'}) RETURN EXISTS { (p)-[:directed]->(:movie) } AS
directs;
```

```
 directs
---------
 f
(1 row)
```

Count matches

`COUNT { }` over a pattern counts the pattern's matches:

```
MATCH (m:movie {title: 'The Matrix'}) RETURN COUNT { (:person)-[:acted_in]->(m) } AS
cast_size;
```

```
 cast_size
-----------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 3
(1 row)
```

A read-statement body keeps its own `DISTINCT`, `LIMIT` and grouping, so it can count something a pattern cannot:

```
MATCH (m:movie {title: 'The Matrix'})
RETURN COUNT { MATCH (p:person)-[:acted_in]->(m) RETURN DISTINCT p } AS distinct_cast;
```

```
 distinct_cast
---------------
 3
(1 row)
```

Gather a column into a list

`COLLECT { }` returns a list, preserving the body's order and its nulls. `ARRAY { }` is the same expression under the standard's name:

GQL — ARRAY

```
MATCH (m:movie {title: 'The Matrix'})
RETURN ARRAY { MATCH (p:person)-[:acted_in]->(m) RETURN p.name ORDER BY p.name } AS cast;
```

Cypher — COLLECT

```
MATCH (m:movie {title: 'The Matrix'})
RETURN COLLECT { MATCH (p:person)-[:acted_in]->(m) RETURN p.name ORDER BY p.name } AS cast;
```

```
                          cast
------------------------------------------------------------
 ["Carrie-Anne Moss", "Keanu Reeves", "Laurence Fishburne"]
(1 row)
```

An empty body gives an empty list, not null:

```
MATCH (m:movie {title: 'The Matrix'})
RETURN COLLECT { MATCH (p:person)-[:acted_in]->(m) WHERE p.born > 2000 RETURN p.name } AS nobo
dy;
```

```
 nobody
--------
 []
(1 row)
```

Take a single value

`VALUE { }` is the scalar subquery. Unlike everything else on this page it keeps the column's own type, so a graph element survives instead of being boxed:

```
MATCH (m:movie {title: 'Inception'})
RETURN VALUE { MATCH (p:person)-[:directed]->(m) RETURN p.name } AS director;
```


© 2026 SKAI Worldwide Co., Ltd.

```
      director
--------------------
 "Christopher Nolan"
(1 row)
```

```
MATCH (m:movie {title: 'Inception'})
RETURN label(VALUE { MATCH (p:person)-[:directed]->(m) RETURN p }) AS yielded_type;
```

```
 yielded_type
--------------
 "person"
(1 row)
```

The body must produce at most one row. More than one is a run-time error, so add an `ORDER BY … LIMIT 1` or an aggregate when the body is not already single-valued:

```
MATCH (m:movie {title: 'The Matrix'})
RETURN VALUE { MATCH (p:person)-[:acted_in]->(m) RETURN p.name } AS one_of_three;
```

```
ERROR:  more than one row returned by a subquery used as an expression
```

Test membership against a subquery

`x IN { }` is a semi-join, not a materialized list, so it stays cheap on a large body. A value and a Cypher body's column meet as `jsonb`:

```
MATCH (m:movie)
WHERE m.year IN { MATCH (x:movie) WHERE x.genre = 'Adventure' RETURN x.year }
RETURN m.title AS film ORDER BY film;
```

```
      film
-----------------
 "Jurassic Park"
(1 row)
```

A node is matched by identity, so two nodes with the same properties are still two nodes. The semi-join then runs on the id:

```
MATCH (p:person)
WHERE p IN { MATCH (d:person)-[:directed]->(:movie {title: 'Inception'}) RETURN d }
RETURN p.name AS director;
```

```
      director
--------------------
 "Christopher Nolan"
(1 row)
```

A SQL body keeps its own column type. `rating.title` is `text`, so unwrap the property with `#>> '{}'` rather than comparing it as `jsonb`:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie)
WHERE m.title #>> '{}' IN { SELECT title FROM rating WHERE source = 'audience' }
RETURN m.title AS film ORDER BY film;
```

```
      film
----------------
 "Inception"
 "Interstellar"
 "The Matrix"
(3 rows)
```

Without the unwrap the comparison has no operator: `operator does not exist: jsonb = text`.

Use one in a projection or from SQL

A subquery expression is a value, so it can be projected by `WITH` and then tested:

```
MATCH (m:movie)
WITH m.title AS film, COUNT { (:person)-[:acted_in]->(m) } AS cast_size
FILTER cast_size > 2
RETURN film, cast_size;
```

```
     film     | cast_size
--------------+-----------
 "The Matrix" | 3
(1 row)
```

It also works from SQL, wrapped in the Cypher query that supplies its variables:

```
SELECT (MATCH (m:movie {title: 'The Matrix'}) RETURN COUNT { (:person)-[:acted_in]->(m) }) AS
cast_size;
```

```
 cast_size
-----------
 3
(1 row)
```

Notes

A write clause anywhere in a body is refused at the keyword, whichever expression wraps it:

```
MATCH (m:movie) RETURN COUNT { CREATE (:tmp) RETURN 1 AS one } AS n;
```

```
ERROR:  syntax error at or near "CREATE"
LINE 1: MATCH (m:movie) RETURN COUNT { CREATE (:tmp) RETURN 1 AS one...
                                       ^
```

So is a binding that would hide a variable of the enclosing query, with the same error the clause raises at the top level:

```
MATCH (m:movie) RETURN COUNT { LET m = 1 RETURN m } AS n;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  variable "m" already exists
LINE 1: MATCH (m:movie) RETURN COUNT { LET m = 1 RETURN m } AS n;
                                       ^
HINT:  LET cannot redefine an existing variable; use a different name.
```

NOTE

`ARRAY` is a reserved keyword, as it is in PostgreSQL, and so is `IN`. `COLLECT`, `COUNT`, `EXISTS` and `VALUE` are not reserved, so they remain usable as identifiers — a `RETURN … AS value` still works.

Compatibility

GQL CONFORMANCE

Two of the six are in the standard:

- `EXISTS { }` implements the GQL **exists predicate** (ISO/IEC 39075:2024 §19.4), including the bare `<graph pattern>` and `<match statement block>` bodies of that subclause and the nested query specification. **Supported** in full.
- `VALUE { }` implements the GQL **value query expression** (§20.6), Feature GQ18, "Scalar subqueries", with the empty-body-is-null rule of General Rule 2. **Supported**, except that §20.6 Syntax Rule 10 requires the body to carry `LIMIT 1` or an aggregate; AgensGraph accepts any body and raises a run-time error if it returns more than one row.

`COUNT { }`, `COLLECT { }`, `ARRAY { }` and `x IN { }` are **extensions**. ISO/IEC 39075:2024 has no subquery form of `COUNT` — §20.9 defines only `COUNT(*)` and `COUNT(<value expression>)` — and its list aggregate is `COLLECT_LIST(…)`, also an aggregate over rows rather than a subquery. `COUNT { }` and `COLLECT { }` come from openCypher; `ARRAY { }` follows SQL's `ARRAY(<subquery>)`; `x IN { }` follows SQL's `IN (<subquery>)`.

Track status in the **GQL conformance matrix**.

openCypher

openCypher has `EXISTS { }`, `COUNT { }` and `COLLECT { }` with the same meanings. It has no `ARRAY { }`, no `VALUE { }` and no `x IN { }` over a subquery. For a scalar it uses `head(COLLECT { … })`, and for membership `x IN COLLECT { … }`. Both of those work here too, but they materialize the whole list, so prefer `VALUE { }` and `x IN { }`.

See also

- **`CALL { subquery }`** GQL — a subquery as a clause, with an explicit import list
- **`FILTER`** GQL — a clause these predicates suit
- **`WHERE`** — the other place a predicate goes
- **Combining SQL and Cypher** — how a SQL body fits in
- **GQL conformance matrix**

© 2026 SKAI Worldwide Co., Ltd.

Data Definition

The statements that create and alter graphs, labels, property indexes and constraints. They are Cypher statements, so they run wherever Cypher runs.

CREATE GRAPH

Creates a graph — a schema, the two base labels that every label inherits from, and the sequence that hands out label ids.

Synopsis

CREATE GRAPH [**IF NOT EXISTS**] graph_name [**AUTHORIZATION** role_specification]
CREATE GRAPH ONLY graph_name

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

A graph is a schema. `CREATE GRAPH` creates that schema and puts three objects in it:

- `ag_vertex`, the base vertex label,
- `ag_edge`, the base edge label,
- `ag_label_seq`, the sequence that assigns label ids.

Every label you create afterwards inherits from `ag_vertex` or `ag_edge`. That inheritance gives an unlabelled pattern like `MATCH (n)` its meaning. **How a graph is stored** describes the objects and the identifier scheme in full.

- The name has to be free as a graph name and as a schema name, because the graph owns a schema of that name.
- If the session's `graph_path` is empty, `CREATE GRAPH` sets it to the new graph. A `graph_path` that is already set is left alone.
- `CREATE GRAPH ONLY` creates the schema and the catalog row and nothing else. It is the form `pg_dump` emits, so that a restore can recreate the base labels itself with their original label ids. A graph created this way cannot hold a label until those base labels exist.

Parameters

IF NOT EXISTS	Issue a notice instead of an error when a graph of that name already exists. Nothing is created and nothing existing is changed.
graph_name	Name of the graph to create.
AUTHORIZATION role_specification	Role that owns the graph and the objects created with it. Write a role name, `CURRENT_ROLE`, `CURRENT_USER` or `SESSION_USER`. Without this clause the graph belongs to the role that runs the statement.

© 2026 SKAI Worldwide Co., Ltd.

ONLY	Create the schema and the catalog row without `ag_vertex`, `ag_edge` or `ag_label_seq`. Unlike the plain form, it does not touch `graph_path`.

Examples

The examples build their own graph, `boxoffice`, rather than using **the example dataset**, and drop it at the end.

Create a graph

```
CREATE GRAPH boxoffice;
```

```
CREATE GRAPH
```

The statement claimed an empty `graph_path`:

```
SHOW graph_path;
```

```
 graph_path
------------
 boxoffice
(1 row)
```

The graph is listed by `\dG`:

```
\dG boxoffice
```

```
  List of graphs
   Name    | Owner
-----------+-------
 boxoffice | agens
(1 row)
```

And it already holds the two base labels:

```
\dGl boxoffice.*
```

```
            List of labels
   Graph    |   Name    |  Type  | Owner
------------+-----------+--------+-------
 boxoffice | ag_edge   | edge   | agens
 boxoffice | ag_vertex | vertex | agens
(2 rows)
```

Create a graph only if it is not there

```
CREATE GRAPH IF NOT EXISTS boxoffice;
```


© 2026 SKAI Worldwide Co., Ltd.

```
NOTICE:  graph "boxoffice" already exists, skipping
CREATE GRAPH
```

The existing graph is untouched. This is not a "create or replace".

Create a graph owned by another role

```
CREATE ROLE curator;
```

```
CREATE GRAPH festival AUTHORIZATION curator;
```

```
CREATE GRAPH
```

```
\dG festival
```

```
   List of graphs
   Name    |  Owner
-----------+---------
 festival | curator
(1 row)
```

Create the schema without the base labels

```
CREATE GRAPH ONLY skeleton;
```

```
CREATE GRAPH
```

The graph exists but holds nothing:

```
\dGl skeleton.*
```

```
No matching labels found.
```

Because `ag_vertex` is missing, a label cannot be created in it:

```
SET graph_path = skeleton;
CREATE VLABEL film;
```

```
ERROR:  relation "skeleton.ag_vertex" does not exist
```

Clean up the three graphs and the role:

```
DROP GRAPH boxoffice CASCADE;
DROP GRAPH festival CASCADE;
DROP GRAPH skeleton;
```


© 2026 SKAI Worldwide Co., Ltd.

```
DROP ROLE curator;
```

`skeleton` needs no `CASCADE` because nothing was created inside it. A graph made by the plain form always does. See **`DROP GRAPH`**.

Notes

`graph_path` names the graph that unqualified label names belong to. It is a per-session setting; set it in each new session, or attach it to a database or a role:

```
ALTER DATABASE <database> SET graph_path = boxoffice;
```

Two more statements act on a graph as a whole. Neither has an `IF EXISTS` form:

```
CREATE GRAPH receipts;
ALTER GRAPH receipts RENAME TO takings;
ALTER GRAPH takings OWNER TO CURRENT_USER;
SHOW graph_path;
```

```
 graph_path
------------
 takings
(1 row)
```

A rename moves the underlying schema with the graph, and the session's `graph_path` follows it. So it reads `takings` above rather than `receipts`.

```
DROP GRAPH takings CASCADE;
```

Compatibility

ISO/IEC 39075 (GQL)

GQL defines `<create graph statement>` (§12.4) as `CREATE [PROPERTY] GRAPH <name> {<open graph type> | <of graph type>}`. A graph is created either as `ANY`, which accepts any content, or against a graph type that constrains its node types, edge types and properties. AgensGraph's graphs are always open, so no graph type is written and none can be. The `PROPERTY` spelling is not accepted: `CREATE PROPERTY GRAPH boxoffice` is a syntax error. Graph types are tracked as planned in the **GQL conformance matrix**.

`AUTHORIZATION` and `ONLY` are AgensGraph's own; GQL has neither. GQL's `OR REPLACE` form is not accepted either.

PostgreSQL

`CREATE GRAPH` corresponds to `CREATE SCHEMA`, and `IF NOT EXISTS` and `AUTHORIZATION` mean what they mean there. The difference is the contents. `CREATE SCHEMA` creates an empty schema. `CREATE GRAPH` also creates the base labels and the label-id sequence, and registers the schema in `ag_graph` so that Cypher can find it. `CREATE SCHEMA` on its own does not produce a graph.

See also

- **`DROP GRAPH`**
- **`CREATE VLABEL`**
- **`CREATE ELABEL`**

© 2026 SKAI Worldwide Co., Ltd.

- How a graph is stored

DROP GRAPH

Removes a graph, and with `CASCADE` every label, vertex and edge in it.

Synopsis

DROP GRAPH [**IF EXISTS**] graph_name [, ...] [**CASCADE** | **RESTRICT**]

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`DROP GRAPH` drops the graph's catalog row and its schema. Everything the graph holds depends on that schema: labels, the tables behind them, their rows, indexes, constraints and sequences. So `RESTRICT`, which is the default, refuses while any of it is there.

A graph made by the plain `CREATE GRAPH` is never empty: `ag_vertex`, `ag_edge` and `ag_label_seq` are created with it. So every such graph needs `CASCADE`. Only a graph made with `CREATE GRAPH ONLY`, which creates none of those, can be dropped without it.

- Dropping a graph does not require its labels to be empty. `CASCADE` takes the rows with the tables.
- If the dropped graph is the session's `graph_path`, the setting is cleared.
- The graph's schema cannot be dropped on its own: `DROP SCHEMA` on it is refused and points you here.

Parameters

IF EXISTS	Issue a notice instead of an error for a graph that does not exist. Other names in the same statement are still dropped.
graph_name	Name of the graph to drop. Several names may be given, separated by commas.
CASCADE	Drop the objects that depend on the graph: its labels, their tables and rows, and the label-id sequence.
RESTRICT	Refuse if anything depends on the graph. This is the default.

Examples

Drop a graph and everything in it

```
CREATE GRAPH clearance;
SET graph_path = clearance;
CREATE VLABEL film;
CREATE (:film {title: 'Arrival'});
```


© 2026 SKAI Worldwide Co., Ltd.

Without `CASCADE` the drop is refused, and the message lists what stands in the way:

```
DROP GRAPH clearance;
```

```
ERROR:  cannot drop graph clearance because other objects depend on it
DETAIL:  sequence clearance.ag_label_seq depends on schema clearance
vlabel ag_vertex depends on schema clearance
elabel ag_edge depends on schema clearance
vlabel film depends on schema clearance
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

`CASCADE` takes all four, and the vertex in `film` with them:

```
DROP GRAPH clearance CASCADE;
```

```
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to sequence clearance.ag_label_seq
drop cascades to vlabel ag_vertex
drop cascades to elabel ag_edge
drop cascades to vlabel film
DROP GRAPH
```

`clearance` was the session's `graph_path`, so the setting is now empty:

```
SHOW graph_path;
```

```
 graph_path
------------

(1 row)
```

Drop several graphs, skipping the ones that are gone

```
CREATE GRAPH archive_2024;
CREATE GRAPH archive_2025;
DROP GRAPH IF EXISTS archive_2024, archive_2025, archive_2026 CASCADE;
```

```
NOTICE:  graph "archive_2026" does not exist, skipping
NOTICE:  drop cascades to 6 other objects
DETAIL:  drop cascades to sequence archive_2025.ag_label_seq
drop cascades to vlabel ag_vertex
drop cascades to elabel ag_edge
drop cascades to sequence archive_2024.ag_label_seq
drop cascades to vlabel ag_vertex
drop cascades to elabel ag_edge
DROP GRAPH
```

The missing name is reported and the two that exist are dropped:

```
\dG archive*
```


© 2026 SKAI Worldwide Co., Ltd.

```
No matching graphs found.
```

Notes

To empty a graph without dropping it, run `MATCH (n) DETACH DELETE n` and then drop the labels you no longer want. The schema and the base labels stay in place. A label has to be empty before `DROP VLABEL` will remove it, so that delete is the usual first step. `DROP GRAPH ... CASCADE` skips the requirement altogether, because it drops the tables rather than the labels one at a time.

Compatibility

ISO/IEC 39075 (GQL)

GQL's `<drop graph statement>` (§12.5) is `DROP [PROPERTY] GRAPH [IF EXISTS] <name>`. AgensGraph accepts the `DROP GRAPH` and `IF EXISTS` parts of that. It adds a name list and the `CASCADE` / `RESTRICT` choice, which GQL does not have. The `PROPERTY` spelling is not accepted: `DROP PROPERTY GRAPH clearance` is a syntax error.

PostgreSQL

`DROP GRAPH` corresponds to `DROP SCHEMA`, and `IF EXISTS`, the name list and `CASCADE` / `RESTRICT` behave as they do there. It differs in refusing to leave a half-dropped graph behind. `DROP SCHEMA` on a graph's schema fails with `cannot drop schema clearance because graph clearance requires it`, so the catalog row and the schema always go together.

See also

- `CREATE GRAPH`
- `DROP VLABEL` and `DROP ELABEL`
- How a graph is stored

CREATE VLABEL

Creates a vertex label — a table of `(id, properties)` that inherits the graph's base vertex label, optionally with properties promoted to typed columns.

Synopsis

CREATE [**UNLOGGED**] **VLABEL** [**IF NOT EXISTS**] label_name [(promoted_property [, ...])]
  [**DISABLE INDEX**]
  [**INHERITS** (parent_label [, ...])]
  [**WITH** (storage_parameter [= value] [, ...])]
  [**TABLESPACE** tablespace_name]

CREATE [**UNLOGGED**] **VLABEL ONLY** label_name (label_id) [option ...]

where promoted_property is one of:

  column_name data_type **GENERATED**
  column_name data_type **GENERATED** (property_key)
  column_name data_type **GENERATED ALWAYS AS** (expression) [**STORED**]

© 2026 SKAI Worldwide Co., Ltd.

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

A vertex label is a table with an `id` of type `graphid` and a `properties` column of type `jsonb`. It inherits the graph's `ag_vertex` unless you name a parent. Creating one also creates the sequence that numbers its ids and the primary key on `id`. **How a graph is stored** covers that layout.

The label is created in the graph named by `graph_path`.

- **The options are position-sensitive.** They have to appear in the order shown: `DISABLE INDEX`, then `INHERITS`, then `WITH`, then `TABLESPACE`. Any other order is a syntax error.
- A parent named in `INHERITS` must be in the same graph and must be a vertex label.
- `IF NOT EXISTS` and `ONLY` belong to separate forms and cannot be combined.
- Cypher creates a label on demand: `CREATE (:film)` against a graph with no `film` label creates one. Use `CREATE VLABEL` when you want options, promoted columns or a parent.

Promoted properties

A promoted property is a real typed column beside the `properties` bag, computed from the bag and stored. Reads of that property go to the column instead of walking the JSON document. The three forms differ in where the value comes from:

- `column_name data_type GENERATED` mirrors the property whose key is the column name.
- `column_name data_type GENERATED ( property_key )` mirrors the named key, so the column can be called something else. The key is an identifier and is down-cased, matching the way Cypher stores map keys.
- `column_name data_type GENERATED ALWAYS AS ( expression ) STORED` computes the column from a SQL expression over the label's own columns.

Whether a column answers for a property depends on the expression, not on which form wrote it. A plain read of one key out of the bag, such as `properties ->> 'year'` with or without a cast, is recorded in `ag_label_property`. That is the catalog a Cypher property read consults. Wrap the read in anything else and the column is an ordinary generated column: SQL reads it, and `n.<key>` does not see it. **Promoting a property to a column** explains when promotion is worth it.

Four things a promoted property cannot be:

- **Not generated.** A bare `column_name data_type` is refused with `a non-generated promoted column is not supported yet`.
- **`VIRTUAL`.** A promoted column has to be materialized to carry an index, so `VIRTUAL` is refused with `promoted generated properties must be STORED`. `STORED` is the default and may be left out.
- **`GENERATED BY DEFAULT`.** Only `GENERATED ALWAYS` is accepted.
- **Named `id`, `properties`, `start` or `end`.** Those are the structural columns of a graph element, and the name is refused as reserved.

The data type has to be one whose text input function is immutable, and it cannot fix a length. `date`, `timestamp`, `timestamptz` and array types are refused with `generation expression is not immutable`. `varchar(n)` and `char(n)` are refused because fitting a value to a length would let the column and the property hold different values. `smallint`, `int`, `bigint`, `numeric`, `real`, `double precision`, `text`, `boolean`, `uuid` and `jsonb` are accepted.

© 2026 SKAI Worldwide Co., Ltd.

Parameters

UNLOGGED	Create the label's table as unlogged: its contents are not written to the write-ahead log and are lost after a crash.
IF NOT EXISTS	Issue a notice instead of an error when a label of that name already exists. Nothing is created; the existing label's options are not changed.
label_name	Name of the label to create, in the graph named by `graph_path`.
promoted_property	A property to mirror into a typed column, in one of the three forms above. Columns are added after `properties`, in the order written.
DISABLE INDEX	Create the label's indexes but leave them invalid, so that loading rows does not maintain them. Reads and writes still work; the planner ignores an invalid index. Rebuild with **`REINDEX VLABEL`**. There is no `ENABLE INDEX`.
INHERITS (parent_label [, ...])	Parent labels, replacing the default parent `ag_vertex`. Each must be a vertex label in the same graph. See **Label inheritance**.
WITH (storage_parameter [= value] [, ...])	Storage parameters for the label's table, as for `CREATE TABLE`.
TABLESPACE tablespace_name	Tablespace for the label's table. Without it, `default_tablespace` decides.
ONLY label_name (label_id)	Create the label with the given label id, and without the default parent, the id default or the primary key. The label id is the high 16 bits of every `graphid` the label hands out. This is the form `pg_dump` emits so that a restore preserves the ids in a dumped graph. A label created this way is not usable from Cypher until those are put back, so write it only when reproducing a dump.

Examples

The examples build their own graph, `filmstore`, rather than using **the example dataset**, and drop it at the end.

Create a vertex label

```
CREATE GRAPH filmstore;
SET graph_path = filmstore;
CREATE VLABEL film;
```


© 2026 SKAI Worldwide Co., Ltd.

```
CREATE VLABEL
```

```
\d filmstore.film
```

```
                                                    Table "filmstore.film"
   Column   |  Type   | Collation | Nullable |                                      Defau
lt
------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------
 id         | graphid |           | not null | graphid(graph_labid('filmstore.film'::cstring),
nextval('filmstore.film_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "film_pkey" PRIMARY KEY, btree (id)
Inherits: filmstore.ag_vertex
```

Create a label only if it is not there

```
CREATE VLABEL IF NOT EXISTS film;
```

```
NOTICE:  label "film" already exists, skipping
CREATE VLABEL
```

Create a label under a parent

```
CREATE VLABEL work;
CREATE VLABEL documentary INHERITS (work);
```

```
\dGv filmstore.documentary
```

```
             List of labels
   Graph   |    Name     |  Type  | Owner
-----------+-------------+--------+-------
 filmstore | documentary | vertex | agens
(1 row)

Vertex label "filmstore.documentary"
--
Inherits: filmstore.work
```

`documentary` inherits `work` instead of `ag_vertex`, so `MATCH (n:work)` also matches documentaries. Label inheritance covers the rest.

Load a label before building its indexes

```
CREATE VLABEL bulk DISABLE INDEX;
```

```
\d filmstore.bulk
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                                  Table "filmstore.bulk"
   Column    |  Type   | Collation | Nullable |                                    Defau
lt
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------
 id          | graphid |           | not null | graphid(graph_labid('filmstore.bulk'::cstring),
nextval('filmstore.bulk_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "bulk_pkey" PRIMARY KEY, btree (id) INVALID
Inherits: filmstore.ag_vertex
```

The primary key exists but is marked `INVALID`. Load the rows, then run `REINDEX VLABEL bulk` to build it.

Set storage parameters and a tablespace

```
CREATE UNLOGGED VLABEL staging WITH (fillfactor = 70) TABLESPACE pg_default;
```

```
SELECT relpersistence, reloptions FROM pg_class WHERE oid = 'filmstore.staging'::regclass;
```

```
 relpersistence |   reloptions
----------------+-----------------
 u              | {fillfactor=70}
(1 row)
```

Promote two properties to typed columns

```
CREATE VLABEL title (year int GENERATED, name text GENERATED);
```

```
\d filmstore.title
```

```
                                                  Table "filmstore.title"
   Column    |  Type   | Collation | Nullable |                                    Defa
ult
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------
--
 id          | graphid |           | not null | graphid(graph_labid('filmstore.title'::cstring)
, nextval('filmstore.title_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
 year        | integer |           |          | generated always as (ag_property_text(propertie
s, 'year'::text, 'n'::"char")::integer) stored
 name        | text    |           |          | generated always as (ag_property_text(propertie
s, 'name'::text, 's'::"char")) stored
Indexes:
    "title_pkey" PRIMARY KEY, btree (id)
Inherits: filmstore.ag_vertex
```

Nothing about writing changes. The property still goes into the bag, and the column follows:

```
CREATE (:title {name: 'Arrival', year: 2016});
MATCH (t:title) RETURN t.name AS name, t.year AS year;
```


© 2026 SKAI Worldwide Co., Ltd.

```
  name   | year
---------+------
 Arrival | 2016
(1 row)
```

The same values are in the columns, reachable from SQL:

```
SELECT name, year FROM filmstore.title;
```

```
  name   | year
---------+------
 Arrival | 2016
(1 row)
```

A property whose value does not match the column's kind is refused at write time, naming both sides:

```
CREATE (:title {name: 42, year: 2016});
```

```
ERROR:  property "name" holds a number
DETAIL:  A column promoting this property holds strings.
CONTEXT:  while computing the promoted typed columns of graph label "title"
```

Promote a property whose key differs from the column name

```
CREATE VLABEL review (released text GENERATED (releaseDate));
```

```
\d filmstore.review
```

```
                                                    Table "filmstore.review"
   Column    |  Type   | Collation | Nullable |                                                  Def
ault
-------------+---------+-----------+----------
+--------------------------------------------------------------------------------------------------
----
 id          | graphid |           | not null |
graphid(graph_labid('filmstore.review'::cstring), nextval('filmstore.review_id_seq'::regclass)
)
 properties | jsonb   |           | not null | jsonb_build_object()
 released   | text    |           |          | generated always as (ag_property_text(propertie
s, 'releasedate'::text, 's'::"char")) stored
Indexes:
    "review_pkey" PRIMARY KEY, btree (id)
Inherits: filmstore.ag_vertex
```

The key was written `releaseDate` and stored as `releasedate`, because a property key is an identifier here and identifiers are down-cased. Cypher reads this property as `n.releasedate`; SQL reads the column as `released`.

Compute a column from the property bag

The long form takes a SQL expression over the label's own columns rather than a property key:

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE VLABEL slug (handle text GENERATED ALWAYS AS (lower(properties ->> 'name')) STORED);
```

```
\d filmstore.slug
```

```
                                                    Table "filmstore.slug"
   Column   |  Type   | Collation | Nullable |                                                 Defau
lt
------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------
 id         | graphid |           | not null | graphid(graph_labid('filmstore.slug'::cstring),
nextval('filmstore.slug_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
 handle     | text    |           |          | generated always as (lower(properties ->> 'name
'::text)) stored
Indexes:
    "slug_pkey" PRIMARY KEY, btree (id)
Inherits: filmstore.ag_vertex
```

```
CREATE (:slug {name: 'Arrival'});
```

```
UPDATE 1
```

A Cypher write reports its row count with the tag `UPDATE`, whatever the write was, and the column is computed and stored:

```
SELECT handle FROM filmstore.slug;
```

```
 handle
---------
 arrival
(1 row)
```

But it answers to no property key, so Cypher does not see it:

```
MATCH (s:slug) RETURN s.handle AS handle;
```

```
 handle
--------

(1 row)
```

`s.handle` reads the key `handle` out of the bag, which nothing ever wrote.

A long form whose expression is a plain read of one key is recorded like the shorthand, and Cypher does see it:

```
CREATE VLABEL runtime (minutes int GENERATED ALWAYS AS ((properties ->> 'minutes')::int)
STORED);
CREATE (:runtime {minutes: 116});
MATCH (r:runtime) RETURN r.minutes AS minutes;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 minutes
---------
     116
(1 row)
```

So reach for one key when the column should answer for that property, and compute from it when the column is for SQL alone.

Refusals worth recognizing

Each of these is rejected outright:

```
CREATE VLABEL bad1 (name varchar(40) GENERATED);
```

```
ERROR:  a promoted property cannot have a length-limited string type
DETAIL:  Type character varying fits a value to its length by truncating or padding it.
HINT:  Use text, which stores the property as it is.
```

```
CREATE VLABEL bad2 (name text);
```

```
ERROR:  a non-generated promoted column is not supported yet
HINT:  Use GENERATED to mirror the property from the jsonb bag.
```

```
CREATE VLABEL bad3 (released date GENERATED);
```

```
ERROR:  generation expression is not immutable
```

```
CREATE VLABEL bad4 (id int GENERATED);
```

```
ERROR:  promoted property name "id" is reserved
```

```
CREATE VLABEL bad5 (name text GENERATED ALWAYS AS (properties ->> 'name') VIRTUAL);
```

```
ERROR:  promoted generated properties must be STORED
LINE 1: ...e text GENERATED ALWAYS AS (properties ->> 'name') VIRTUAL);
                                                              ^
HINT:  A VIRTUAL generated column cannot carry an index.
```

A property key has to lead to one column all the way down a hierarchy. So promoting a key a parent already promotes is refused unless the column name matches:

```
CREATE VLABEL feature (year int GENERATED) INHERITS (work);
CREATE VLABEL short (released int GENERATED (year)) INHERITS (feature);
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  property "year" is already promoted on graph label "feature"
DETAIL:  It answers from column "year" there and would answer from column "released" here.
HINT:  Promote it to the same column name, or leave it to the label that has it.
```

The mirror image is refused too: promoting a different key to a column name an ancestor already uses would make two keys read one inherited column.

Recreate a label with a given label id

Label ids come from the graph's `ag_label_seq` in creation order. `ONLY` sets one explicitly, which is how a restore reproduces the ids a dump captured:

```
CREATE VLABEL ONLY restored (300);
```

```
SELECT l.labname, l.labid FROM pg_catalog.ag_label l
  JOIN pg_catalog.ag_graph g ON g.oid = l.graphid
 WHERE g.graphname = 'filmstore' ORDER BY l.labid;
```

```
   labname    | labid
--------------+-------
 ag_vertex    |     1
 ag_edge      |     2
 film         |     3
 work         |     4
 documentary  |     5
 bulk         |     6
 staging      |     7
 title        |     8
 review       |     9
 slug         |    10
 runtime      |    11
 feature      |    12
 restored     |   300
(13 rows)
```

The label has no parent, no `id` default and no primary key. So it is not usable from Cypher as it stands:

```
\d filmstore.restored
```

```
              Table "filmstore.restored"
   Column    |  Type   | Collation | Nullable | Default
-------------+---------+-----------+----------+---------
 id          | graphid |           | not null |
 properties | jsonb   |           | not null |
```

Reusing an id the graph has already given out fails on the unique index over `(graphid, labid)`.

Finally, drop the graph the examples built:

```
DROP GRAPH filmstore CASCADE;
```


© 2026 SKAI Worldwide Co., Ltd.

Notes

The label name cannot be qualified. `CREATE VLABEL filmstore.film` is a syntax error, and the graph always comes from `graph_path`. With none set, the statement stops with `graph_path is NULL`.

`\dGv` lists vertex labels and describes each one it matches; `\dGl` lists labels of both kinds without the detail.

> NOTE
>
> `enable_property_promotion` decides whether a Cypher property read is answered from a promoted column. It defaults to on, and turning it off can change results as well as plans: a column compares and orders values with its own type's rules, while the `jsonb` bag uses `jsonb` ordering.

Compatibility

ISO/IEC 39075 (GQL)

GQL has no label DDL. Its schema mechanism is the graph type, `<create graph type statement>` (§12.6). A graph type declares node types and edge types for a whole graph at once, and constrains a graph that references it. AgensGraph declares labels one at a time, and its graphs are open: a label puts no restriction on which properties its vertices carry. Graph types are tracked as planned in the **GQL conformance matrix**.

PostgreSQL

`CREATE VLABEL` corresponds to `CREATE TABLE`. `UNLOGGED`, `IF NOT EXISTS`, `INHERITS`, `WITH` and `TABLESPACE` mean what they mean there, and the promoted-property forms are `GENERATED ALWAYS AS ... STORED` generated columns. `CREATE TABLE` has none of the rest: a fixed column list you cannot choose, a registration in `ag_label` and `ag_label_property`, an automatic parent, and the position-sensitive option order. Going the other way, a label's table cannot be reshaped with `ALTER TABLE` unless `enable_graph_ddl` is on. See **`ALTER VLABEL`** and **`ALTER ELABEL`**.

See also

- **`CREATE ELABEL`**
- **`ALTER VLABEL`** and **`ALTER ELABEL`**
- **`DROP VLABEL`** and **`DROP ELABEL`**
- **`CREATE PROPERTY INDEX`**
- Promoting a property to a column
- Label inheritance

CREATE ELABEL

Creates an edge label — a table of `(id, start, end, properties)` that inherits the graph's base edge label, optionally with properties promoted to typed columns.

Synopsis

```
CREATE [ UNLOGGED ] ELABEL [ IF NOT EXISTS ] label_name [ ( promoted_property [, ... ] ) ]
  [ DISABLE INDEX ]
  [ INHERITS ( parent_label [, ... ] ) ]
  [ WITH ( storage_parameter [= value] [, ... ] ) ]
  [ TABLESPACE tablespace_name ]
```


© 2026 SKAI Worldwide Co., Ltd.

```
CREATE [ UNLOGGED ] ELABEL ONLY label_name ( label_id ) [ option ... ]

where promoted_property is one of:

  column_name data_type GENERATED
  column_name data_type GENERATED ( property_key )
  column_name data_type GENERATED ALWAYS AS ( expression ) [ STORED ]
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

An edge label is a table with four columns: an `id` of type `graphid`, the `graphid` of the edge's `start` and `end` vertices, and a `properties` column of type `jsonb`. It inherits the graph's `ag_edge` unless you name a parent.

Creating one also creates the sequence that numbers its ids and three indexes. Two are btree indexes, on `(start, "end")` and `("end", start)`, which make a traversal fast in either direction. The third is a BRIN index on `id`. An edge label gets no primary key on `id`, because edges are reached through their endpoints. **How a graph is stored** covers that layout.

The label is created in the graph named by `graph_path`.

- **The options are position-sensitive.** They have to appear in the order shown: `DISABLE INDEX`, then `INHERITS`, then `WITH`, then `TABLESPACE`. Any other order is a syntax error.
- A parent named in `INHERITS` must be in the same graph and must be an edge label. Naming a vertex label is refused.
- `IF NOT EXISTS` and `ONLY` belong to separate forms and cannot be combined.
- An edge label puts no restriction on which labels its endpoints have. `start` and `end` are plain `graphid` columns, not foreign keys.
- Cypher creates a label on demand: `CREATE (a)-[:directed]->(b)` against a graph with no `directed` label creates one. Use `CREATE ELABEL` when you want options, promoted columns or a parent.

Promoted properties

A promoted property is a typed column beside the `properties` bag, computed from the bag and stored. Reads of that property go to the column rather than into the JSON document. The three forms and their restrictions are the same as for a vertex label, and `CREATE VLABEL` sets them out. On an edge label the reserved column names are `id`, `properties`, `start` and `end`.

Parameters

UNLOGGED	Create the label's table as unlogged: its contents are not written to the write-ahead log and are lost after a crash.
IF NOT EXISTS	Issue a notice instead of an error when a label of that name already exists. Nothing is created; the existing label's options are not changed.
label_name	Name of the label to create, in the graph named by `graph_path`.

© 2026 SKAI Worldwide Co., Ltd.

promoted_property	A property to mirror into a typed column. Columns are added after `properties`, in the order written.
DISABLE INDEX	Create the label's three indexes but leave them invalid, so that loading rows does not maintain them. Reads and writes still work; the planner ignores an invalid index. Rebuild with **`REINDEX ELABEL`**. There is no `ENABLE INDEX`.
INHERITS (parent_label [, ...])	Parent labels, replacing the default parent `ag_edge`. Each must be an edge label in the same graph. See **Label inheritance**.
WITH (storage_parameter [= value] [, ...])	Storage parameters for the label's table, as for `CREATE TABLE`.
TABLESPACE tablespace_name	Tablespace for the label's table. Without it, `default_tablespace` decides.
ONLY label_name (label_id)	Create the label with the given label id and without the default parent, the id default or the endpoint indexes. This is the form `pg_dump` emits so that a restore preserves the ids in a dumped graph. Write it only when reproducing a dump.

Examples

The examples build their own graph, `castlists`, rather than using **the example dataset**, and drop it at the end.

Create an edge label

```
CREATE GRAPH castlists;
SET graph_path = castlists;
CREATE VLABEL person;
CREATE VLABEL film;
CREATE ELABEL worked_on;
```

```
CREATE ELABEL
```

```
\d castlists.worked_on
```

```
                                                  Table "castlists.worked_on"
   Column    |  Type    | Collation | Nullable |
Default
------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------
----------
 id         | graphid |           | not null | graphid(graph_labid('castlists.worked_on'::cstr
ing), nextval('castlists.worked_on_id_seq'::regclass))
```


© 2026 SKAI Worldwide Co., Ltd.

```
 start      | graphid |           | not null |
 end        | graphid |           | not null |
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "worked_on_end_idx" btree ("end", start)
    "worked_on_id_idx" brin (id)
    "worked_on_start_idx" btree (start, "end")
Inherits: castlists.ag_edge
```

Creating the label also set the statistics target of `start` and `end` to 1000, because a traversal's row estimate depends on how those two columns are distributed:

```
SELECT attname, attstattarget FROM pg_attribute
 WHERE attrelid = 'castlists.worked_on'::regclass AND attnum > 0;
```

```
  attname   | attstattarget
------------+---------------
 id         |
 start      |          1000
 end        |          1000
 properties |
(4 rows)
```

Group several edge types under one parent

```
CREATE ELABEL acted_in INHERITS (worked_on);
CREATE ELABEL directed INHERITS (worked_on);
```

```
\dGe castlists.acted_in
```

```
           List of labels
   Graph    |   Name    | Type | Owner
------------+-----------+------+-------
 castlists | acted_in | edge | agens
(1 row)

Edge label "castlists.acted_in"
--
Inherits: castlists.worked_on
```

A pattern on the parent now matches either kind of edge, and `label()` still reports the one the edge was created with:

```
CREATE (:person {name: 'Denis Villeneuve'})-[:directed {credited: true}]->(:film {title:
'Arrival'});
MATCH (p)-[e:worked_on]->(f) RETURN label(e) AS type, e.credited AS credited;
```

```
    type     | credited
-------------+----------
 "directed" | true
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

Create a label only if it is not there

```
CREATE ELABEL IF NOT EXISTS acted_in;
```

```
NOTICE:  label "acted_in" already exists, skipping
CREATE ELABEL
```

Load an edge label before building its indexes

All three indexes are created invalid, and the table itself is unlogged:

```
CREATE UNLOGGED ELABEL staged DISABLE INDEX WITH (fillfactor = 70) TABLESPACE pg_default;
```

```
\d castlists.staged
```

```
                                                  Unlogged table "castlists.staged"
   Column    |  Type   | Collation | Nullable |                                                  Def
ault
------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------
----
 id         | graphid |           | not null |
graphid(graph_labid('castlists.staged'::cstring), nextval('castlists.staged_id_seq'::regclass)
)
 start      | graphid |           | not null |
 end        | graphid |           | not null |
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "staged_end_idx" btree ("end", start) INVALID
    "staged_id_idx" brin (id) INVALID
    "staged_start_idx" btree (start, "end") INVALID
Inherits: castlists.ag_edge
```

Load the edges, then run `REINDEX ELABEL staged` to build the three indexes.

Promote an edge property to a typed column

```
CREATE ELABEL scored (rating numeric GENERATED, source text GENERATED (site));
```

```
\d castlists.scored
```

```
                                                    Table "castlists.scored"
   Column    |  Type   | Collation | Nullable |                                                  Def
ault
------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------
----
 id         | graphid |           | not null |
graphid(graph_labid('castlists.scored'::cstring), nextval('castlists.scored_id_seq'::regclass)
)
 start      | graphid |           | not null |
 end        | graphid |           | not null |
 properties | jsonb   |           | not null | jsonb_build_object()
 rating     | numeric |           |          | generated always as (ag_property_text(propertie
```


© 2026 SKAI Worldwide Co., Ltd.

```
s, 'rating'::text, 'n'::"char")::numeric) stored
 source     | text    |           |          | generated always as (ag_property_text(propertie
s, 'site'::text, 's'::"char")) stored
Indexes:
    "scored_end_idx" btree ("end", start)
    "scored_id_idx" brin (id)
    "scored_start_idx" btree (start, "end")
Inherits: castlists.ag_edge
```

Writing the edge fills both columns:

```
MATCH (p:person), (f:film) CREATE (p)-[:scored {rating: 8.5, site: 'critics'}]->(f);
```

```
UPDATE 1
```

SQL reads them under the column names:

```
SELECT rating, source FROM castlists.scored;
```

```
 rating | source
--------+---------
    8.5 | critics
(1 row)
```

Cypher reads them under the property keys, so the second one is `e.site`, not `e.source`:

```
MATCH ()-[e:scored]->() RETURN e.rating AS rating, e.site AS site;
```

```
 rating |  site
--------+---------
    8.5 | critics
(1 row)
```

Refusals worth recognizing

An edge label can only inherit an edge label:

```
CREATE ELABEL wrong INHERITS (person);
```

```
ERROR:  parent graph label "person" is not edge label.
```

Finally, drop the graph the examples built:

```
DROP GRAPH castlists CASCADE;
```

Notes

The label name cannot be qualified. `CREATE ELABEL castlists.directed` is a syntax error, and the graph always comes from `graph_path`. With none set, the statement stops with `graph_path is NULL`.

`\dGe` lists edge labels and describes each one it matches; `\dGl` lists labels of both kinds without the detail.

© 2026 SKAI Worldwide Co., Ltd.

Compatibility

ISO/IEC 39075 (GQL)

GQL has no label DDL. An edge is described by an edge type inside a graph type, `<create graph type statement>` (§12.6). An edge type names the edge's endpoint node types and its properties, and constrains a graph that references it. AgensGraph declares edge labels one at a time, and an edge label constrains neither its endpoints nor its properties. Graph types are tracked as planned in the GQL conformance matrix.

PostgreSQL

`CREATE ELABEL` corresponds to `CREATE TABLE`. `UNLOGGED`, `IF NOT EXISTS`, `INHERITS`, `WITH` and `TABLESPACE` mean what they mean there, and the promoted-property forms become `GENERATED ALWAYS AS ... STORED` generated columns. The differences are the fixed four-column shape, the three indexes created for you, the registration in `ag_label`, the automatic parent, and the position-sensitive option order. `start` and `end` are not declared as foreign keys, so nothing at the table level stops an edge from pointing at a vertex that has been deleted. `DROP VLABEL` handles that.

See also

- `CREATE VLABEL`
- `ALTER VLABEL` and `ALTER ELABEL`
- `DROP VLABEL` and `DROP ELABEL`
- `REINDEX VLABEL` and `REINDEX ELABEL`
- Label inheritance

ALTER VLABEL and ALTER ELABEL

Renames a label, changes its storage, attaches or detaches a parent, turns its indexes off, or adds and removes promoted typed columns.

Synopsis

ALTER VLABEL [**IF EXISTS**] label_name action [, ...]
ALTER VLABEL [**IF EXISTS**] label_name **RENAME TO** new_label_name

ALTER ELABEL [**IF EXISTS**] label_name action [, ...]
ALTER ELABEL [**IF EXISTS**] label_name **RENAME TO** new_label_name

where action is one of:

ADD [**COLUMN**] promoted_property
DROP [**COLUMN**] column_name [**CASCADE** | **RESTRICT**]
DISABLE INDEX
SET STORAGE { **PLAIN** | **EXTERNAL** | **EXTENDED** | **MAIN** }
SET TABLESPACE tablespace_name
SET { **LOGGED** | **UNLOGGED** }
CLUSTER ON index_name
SET WITHOUT CLUSTER
INHERIT parent_label

© 2026 SKAI Worldwide Co., Ltd.

NO INHERIT parent_label
REPLICA IDENTITY { **DEFAULT** | **USING INDEX** index_name | **FULL** | **NOTHING** }
OWNER TO role_specification

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`ALTER VLABEL` acts on a vertex label and `ALTER ELABEL` on an edge label. Apart from the keyword, the two statements are the same: **every action listed above is accepted on both kinds.** The difference is what each kind of label has for an action to act on.

- The indexes a label is created with differ: a vertex label gets a primary key on `id`, an edge label gets btree indexes on `(start, "end")` and `("end", start)` plus a BRIN index on `id`. `DISABLE INDEX` invalidates all of a label's indexes either way, so it has one to invalidate on a new vertex label and three on a new edge label. `CLUSTER ON` also has different names to choose from.
- The structural columns that `DROP COLUMN` refuses are `id` and `properties` on a vertex label, plus `start` and `end` on an edge label.
- `INHERIT` and `NO INHERIT` require a parent of the same kind as the label, and in the same graph.

The label is looked up in the graph named by `graph_path`; the name cannot be qualified.

- `RENAME TO` is its own form. It cannot be combined with any other action, and a list of several actions cannot contain it.
- Several actions in one statement are separated by commas and applied in order.
- The base labels `ag_vertex` and `ag_edge` cannot be altered at all.
- `ALTER VLABEL` refuses an edge label and `ALTER ELABEL` refuses a vertex label.

Reshaping a label with `ALTER TABLE` instead is refused unless `enable_graph_ddl` is on. The graph catalog describes a label's columns, and `ALTER TABLE` does not update it. **How a graph is stored** covers that gate.

Parameters

IF EXISTS	Issue a notice instead of an error when the label does not exist.
label_name	Label to alter, in the graph named by `graph_path`.
RENAME TO new_label_name	Give the label a new name. Vertices and edges keep their ids, so `label()` reports the new name for rows created before the rename. The table's sequence and indexes keep the names they were given, so check those before you write `CLUSTER ON`.
ADD [COLUMN] promoted_property	Add a promoted typed column and backfill it from the existing rows. `promoted_property` takes the same three forms as in `CREATE VLABEL`, under the same restrictions. A property already promoted on the label is refused, as is promoting the same key twice in one statement.

© 2026 SKAI Worldwide Co., Ltd.

DROP [COLUMN] column_name [CASCADE | RESTRICT]	Drop a column and, for a promoted property, forget its mapping. A structural column of the label is refused. Indexes built on the dropped column go with it.
DISABLE INDEX	Mark every index on the label invalid, so that a bulk load does not maintain them. Reads and writes keep working and the planner ignores an invalid index. There is no `ENABLE INDEX`; the way back is **`REINDEX VLABEL`**.
SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }	Set the storage mode of the label's `properties` column. The mode decides whether a large property map is held inline or in the table's TOAST relation, and whether it is compressed. A label starts with `EXTENDED`, which is out of line and compressed. The statement changes the strategy for future writes; it does not rewrite what is already stored.
SET TABLESPACE tablespace_name	Move the label's table to another tablespace. Its indexes stay where they are.
SET { LOGGED | UNLOGGED }	Turn write-ahead logging for the label's table on or off. The label's id sequence changes persistence with it.
CLUSTER ON index_name	Record an index as the one a later `CLUSTER` should use. It does not reorder anything now.
SET WITHOUT CLUSTER	Forget that record.
INHERIT parent_label	Add the label as a child of another label of the same kind in the same graph, so that a pattern on the parent starts matching this label's rows.
NO INHERIT parent_label	Remove that relationship.
REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }	Choose which old values a write writes to the WAL to identify a row, for logical replication.
OWNER TO role_specification	Give the label to another role. Write a role name, `CURRENT_ROLE`, `CURRENT_USER` or `SESSION_USER`.

Examples

The examples build their own graph, `catalogue`, rather than using **the example dataset**, and drop it at the end.

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE GRAPH catalogue;
SET graph_path = catalogue;
CREATE VLABEL film;
CREATE (:film {title: 'Arrival', year: 2016}), (:film {title: 'Dune', year: 2021});
```

Rename a label

```
ALTER VLABEL film RENAME TO movie;
```

```
\dGl catalogue.mov*
```

```
           List of labels
   Graph    | Name  |  Type  | Owner
------------+-------+--------+-------
 catalogue | movie | vertex | agens
(1 row)
```

The rows came with it:

```
MATCH (m:movie) RETURN count(m) AS movies;
```

```
 movies
--------
 2
(1 row)
```

The sequence and the primary key still carry the old name:

```
\d catalogue.movie
```

```
                                                  Table "catalogue.movie"
   Column    |  Type   | Collation | Nullable |                                                 Defau
lt
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------
-
 id          | graphid |           | not null | graphid(graph_labid('catalogue.movie'::cstring)
, nextval('catalogue.film_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "film_pkey" PRIMARY KEY, btree (id)
Inherits: catalogue.ag_vertex
```

So `ALTER VLABEL movie CLUSTER ON movie_pkey` fails with `index "movie_pkey" for table "movie" does not exist`, and the name to write is `film_pkey`.

Turn a label's indexes off for a bulk load

```
ALTER VLABEL movie DISABLE INDEX;
```


© 2026 SKAI Worldwide Co., Ltd.

```
\d catalogue.movie
```

```
                                                    Table "catalogue.movie"
   Column    |  Type    | Collation | Nullable |                                      Defau
lt
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------
-
 id          | graphid |           | not null | graphid(graph_labid('catalogue.movie'::cstring)
, nextval('catalogue.film_id_seq'::regclass))
 properties | jsonb    |           | not null | jsonb_build_object()
Indexes:
    "film_pkey" PRIMARY KEY, btree (id) INVALID
Inherits: catalogue.ag_vertex
```

Load the rows, then build the indexes again:

```
REINDEX VLABEL movie;
```

```
REINDEX
```

Promote a property on a label that already holds rows

```
ALTER VLABEL movie ADD COLUMN year int GENERATED;
```

```
\d catalogue.movie
```

```
                                                      Table "catalogue.movie"
   Column    |  Type    | Collation | Nullable |                                        Defa
ult
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------
--
 id          | graphid |           | not null | graphid(graph_labid('catalogue.movie'::cstring)
, nextval('catalogue.film_id_seq'::regclass))
 properties | jsonb    |           | not null | jsonb_build_object()
 year        | integer |           |          | generated always as (ag_property_text(propertie
s, 'year'::text, 'n'::"char")::integer) stored
Indexes:
    "film_pkey" PRIMARY KEY, btree (id)
Inherits: catalogue.ag_vertex
```

The existing rows are backfilled, so the column is complete the moment the statement returns:

```
SELECT year FROM catalogue.movie ORDER BY year;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 year
------
 2016
 2021
(2 rows)
```

Look up what a label promotes

`ag_label_property` maps a property key to the column that answers for it:

```
ALTER VLABEL movie ADD COLUMN released text GENERATED (releaseDate);
```

```
SELECT p.propname, p.attnum FROM pg_catalog.ag_label_property p
  JOIN pg_catalog.ag_label l ON l.oid = p.laboid
  JOIN pg_catalog.ag_graph g ON g.oid = l.graphid
 WHERE g.graphname = 'catalogue' AND l.labname = 'movie' ORDER BY p.attnum;
```

```
  propname   | attnum
-------------+--------
 year        |      3
 releasedate |      4
(2 rows)
```

`attnum` is the column's position in the table. Join the query to `pg_attribute` on it when you want the column name as well. The two differ whenever the explicit-key form was used.

Drop a promoted column

```
ALTER VLABEL movie DROP COLUMN released;
```

The mapping goes with the column:

```
SELECT p.propname, p.attnum FROM pg_catalog.ag_label_property p
  JOIN pg_catalog.ag_label l ON l.oid = p.laboid
  JOIN pg_catalog.ag_graph g ON g.oid = l.graphid
 WHERE g.graphname = 'catalogue' AND l.labname = 'movie' ORDER BY p.attnum;
```

```
 propname | attnum
----------+--------
 year     |      3
(1 row)
```

Reads of `releasedate` go back to the property bag; nothing was deleted from it.

Change how the property map is stored

```
ALTER VLABEL movie SET STORAGE external;
```

The mode is set on `properties`, whichever column you might have expected:

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT attname, attstorage FROM pg_attribute
 WHERE attrelid = 'catalogue.movie'::regclass AND attnum > 0 AND NOT attisdropped;
```

```
  attname   | attstorage
------------+------------
 id         | p
 properties | e
 year       | p
(3 rows)
```

Attach and detach a parent

```
CREATE VLABEL work;
ALTER VLABEL movie INHERIT work;
```

```
\dGv catalogue.movie
```

```
           List of labels
   Graph   | Name  |  Type  | Owner
-----------+-------+--------+-------
 catalogue | movie | vertex | agens
(1 row)

Vertex label "catalogue.movie"
--
Inherits: catalogue.ag_vertex,
          catalogue.work
```

`movie` now has two parents: the base label it started with, and `work`. Detaching leaves the base label alone:

```
ALTER VLABEL movie NO INHERIT work;
```

```
ALTER VLABEL
```

Refusals worth recognizing

A structural column cannot be dropped:

```
ALTER VLABEL movie DROP COLUMN properties;
```

```
ERROR:  cannot drop column "properties" of graph label "movie"
HINT:  It is a structural column of the graph label.
```

An added column has to be generated from the bag:

```
ALTER VLABEL movie ADD COLUMN rank int;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  a non-generated promoted column is not supported yet
HINT:  Use GENERATED to mirror the property from the jsonb bag.
```

A property already promoted cannot be promoted a second time:

```
ALTER VLABEL movie ADD COLUMN yr2 int GENERATED (year);
```

```
ERROR:  property "year" is already promoted on graph label "movie"
HINT:  Drop the existing promoted column first, or promote the property to a different column
name.
```

The base labels are off limits, as parents and as targets:

```
ALTER VLABEL movie INHERIT ag_vertex;
```

```
ERROR:  cannot ALTER inheritance with base label
```

```
ALTER VLABEL ag_vertex SET STORAGE main;
```

```
ERROR:  ALTER base vertex label is prohibited
```

Each statement acts on one kind of label:

```
CREATE ELABEL directed;
ALTER VLABEL directed RENAME TO helmed;
```

```
ERROR:  RENAME VLABEL cannot RENAME edge label
```

And `RENAME TO` stands alone:

```
ALTER VLABEL movie RENAME TO film, SET STORAGE main;
```

```
ERROR:  syntax error at or near ","
LINE 1: ALTER VLABEL movie RENAME TO film, SET STORAGE main;
                                          ^
```

Reshaping the label's table with SQL is refused while `enable_graph_ddl` is off:

```
ALTER TABLE catalogue.movie ADD COLUMN rank int;
```

```
ERROR:  cannot add a column to graph label "movie" with ALTER TABLE
HINT:  Use ALTER VLABEL or ALTER ELABEL instead, which keeps the graph catalog in step.
```

Finally, drop the graph the examples built:

© 2026 SKAI Worldwide Co., Ltd.

```
DROP GRAPH catalogue CASCADE;
```

Notes

`SET STORAGE` takes four keywords, not five. `DEFAULT` is reserved and is a syntax error here. To go back to the mode a label starts with, write `SET STORAGE extended`. An unrecognized word is reported as `invalid storage type "..."`.

The `IF EXISTS` notice differs between the two forms: `label "nosuch" does not exist, skipping` from `RENAME TO`, and `graph label "nosuch" does not exist, skipping` from an action list. Both then report success and do nothing.

Compatibility

ISO/IEC 39075 (GQL)

GQL has no statement that alters a label, because it has no labels to alter. An element type belongs to a graph type, and a graph type is created and dropped whole (`<create graph type statement>`, §12.6). Nothing in GQL corresponds to renaming a label, promoting a property, or turning an index off. Graph types are tracked as planned in the GQL conformance matrix.

PostgreSQL

Every action here is an `ALTER TABLE` action, and each one means what it means there, with two adjustments. `SET STORAGE` names no column. The only column whose storage is worth choosing is `properties`, and the mode is applied to that one. `DISABLE INDEX` has no `ALTER TABLE` counterpart at all: it marks the label's indexes invalid, which SQL can otherwise only do by interrupting `CREATE INDEX CONCURRENTLY`.

Going the other way, `ALTER TABLE` on a label's table is refused for the actions that would reshape it, unless `enable_graph_ddl` is on. Adding, dropping or retyping a column is such an action. The example above shows the refusal. The actions on this page never need that setting, because they update the graph catalog themselves.

See also

- `CREATE VLABEL`
- `CREATE ELABEL`
- `DROP VLABEL` and `DROP ELABEL`
- `REINDEX VLABEL` and `REINDEX ELABEL`
- Promoting a property to a column
- Label inheritance

DROP VLABEL and DROP ELABEL

Removes a vertex label or an edge label, and with `CASCADE` the rows in it.

Synopsis

DROP VLABEL [**IF EXISTS**] label_name [, ...] [**CASCADE** | **RESTRICT**]
DROP ELABEL [**IF EXISTS**] label_name [, ...] [**CASCADE** | **RESTRICT**]

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

© 2026 SKAI Worldwide Co., Ltd.

Description

`DROP VLABEL` drops a vertex label and `DROP ELABEL` an edge label; each refuses the other kind. Dropping a label drops the table behind it, along with its id sequence, its indexes and its constraints.

`RESTRICT`, which is the default, has two things to object to:

- **Rows.** A label that still holds vertices or edges is refused with `cannot drop <label> because it is not empty.` This is stricter than the dependency check PostgreSQL applies to a table, and it catches people out.
- **Children.** A label with a child label is refused, and the message names the child.

`CASCADE` overrides both. It drops a label whatever it holds, and:

- dropping a **vertex** label deletes the edges attached to its vertices, whatever labels those edges have. The edge labels themselves survive.
- dropping an **edge** label deletes only its own edges. The vertices they connected survive.

The label is looked up in the graph named by `graph_path`; the name cannot be qualified.

- The base labels `ag_vertex` and `ag_edge` cannot be dropped.
- The table behind a label cannot be dropped on its own: `DROP TABLE` on it is refused and points you here.
- To empty a graph without dropping its labels, use `MATCH (n) DETACH DELETE n` instead.

Parameters

IF EXISTS	Issue a notice instead of an error for a label that does not exist. Other names in the same statement are still dropped.
label_name	Name of the label to drop. Several names may be given, separated by commas.
CASCADE	Drop the label along with its rows and its child labels. For a vertex label, also drop the edges attached to its vertices.
RESTRICT	Refuse if the label holds any rows or has a child label. This is the default.

Examples

The examples build their own graph, `tidygraph`, rather than using **the example dataset**, and drop it at the end.

```
CREATE GRAPH tidygraph;
SET graph_path = tidygraph;
CREATE VLABEL film;
CREATE VLABEL person;
CREATE ELABEL directed;
CREATE (:person {name: 'Denis Villeneuve'})-[:directed]->(:film {title: 'Arrival'});
CREATE VLABEL draft;
```


© 2026 SKAI Worldwide Co., Ltd.

Drop an empty label

```
DROP VLABEL draft;
```

```
DROP VLABEL
```

Nothing else is reported, because nothing else was affected.

Drop a label that still holds vertices

`film` has a vertex in it, so the default refuses:

```
DROP VLABEL film;
```

```
ERROR:  cannot drop film because it is not empty.
```

`CASCADE` takes the label, its vertex, and the `directed` edge that pointed at it:

```
DROP VLABEL film CASCADE;
```

```
DROP VLABEL
```

```
MATCH ()-[e:directed]->() RETURN count(e) AS directed_edges;
```

```
 directed_edges
----------------
 0
(1 row)
```

The edge is gone but its label is not. `directed` is still there, empty:

```
\dGl tidygraph.dir*
```

```
           List of labels
   Graph   |   Name   | Type | Owner
-----------+----------+------+-------
 tidygraph | directed | edge | agens
(1 row)
```

Drop a parent label

```
CREATE VLABEL work;
CREATE VLABEL documentary INHERITS (work);
DROP VLABEL work;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  cannot drop vlabel work because other objects depend on it
DETAIL:  vlabel documentary depends on vlabel work
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

`CASCADE` names the child it takes with it:

```
DROP VLABEL work CASCADE;
```

```
NOTICE:  drop cascades to vlabel documentary
DROP VLABEL
```

Drop several labels, skipping the ones that are gone

```
CREATE VLABEL a;
CREATE VLABEL b;
DROP VLABEL IF EXISTS a, b, c;
```

```
NOTICE:  vlabel "c" does not exist, skipping
DROP VLABEL
```

Refusals worth recognizing

Each statement drops one kind of label:

```
DROP ELABEL person;
```

```
ERROR:  DROP ELABEL cannot DROP vertex label
```

The base labels stay:

```
DROP VLABEL ag_vertex;
```

```
ERROR:  DROP base vertex label is prohibited
```

And the table cannot be dropped out from under the label:

```
DROP TABLE tidygraph.person;
```

```
ERROR:  cannot drop table tidygraph.person because vlabel person requires it
HINT:  You can drop vlabel person instead.
```

Finally, drop the graph the examples built:

```
DROP GRAPH tidygraph CASCADE;
```


© 2026 SKAI Worldwide Co., Ltd.

Notes

An edge whose endpoint vertex is deleted is deleted with it, both here and in `DETACH DELETE`. `start` and `end` are plain `graphid` columns rather than foreign keys. The graph write path does this, not a referential action on the table.

> NOTE
>
> `cypher_allow_unsafe_ddl` switches off the kind check and the base-label check, so with it on `DROP ELABEL` will drop a vertex label and `ag_vertex` can be dropped. It exists to repair a graph whose catalog is already wrong. Leave it off otherwise.

Compatibility

ISO/IEC 39075 (GQL)

GQL has no statement that drops a label. Element types belong to a graph type, which is dropped whole by `<drop graph type statement>` (§12.7), and dropping one does not delete any data. Graph types are tracked as planned in the GQL conformance matrix.

PostgreSQL

`DROP VLABEL` and `DROP ELABEL` correspond to `DROP TABLE`, and `IF EXISTS`, the name list and `CASCADE` / `RESTRICT` behave as they do there. One addition has no `DROP TABLE` counterpart. `DROP TABLE` does not care how many rows a table has; these statements refuse a label that is not empty unless you write `CASCADE`. And for a vertex label, `CASCADE` also deletes edges in other labels. Read `CASCADE` here as "and the graph data that depends on it", not only "and the dependent catalog objects".

See also

- `CREATE VLABEL`
- `CREATE ELABEL`
- `ALTER VLABEL` and `ALTER ELABEL`
- `DROP GRAPH`
- Label inheritance

CREATE PROPERTY INDEX

Builds an index on one or more properties of a label.

Synopsis

```
CREATE [ UNIQUE ] PROPERTY INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] index_name ]
  ON [ ONLY ] label_name [ USING method ]
  ( { property | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
[, ... ] )
  [ NULLS [ NOT ] DISTINCT ]
  [ WITH ( storage_parameter [= value] [, ... ] ) ]
  [ TABLESPACE tablespace_name ]
  [ WHERE condition ]
```

bold keywords · *italic* placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

© 2026 SKAI Worldwide Co., Ltd.

Description

There is no separate kind of index behind this statement. A property index is an ordinary PostgreSQL index over an expression that reads the property out of the label's `jsonb` bag. `CREATE PROPERTY INDEX ON movie (title)` builds a btree whose key is `(properties.'title')`. That is how `\d` reports it, and the planner must be able to see the predicate in that form to use the index.

That expression is the whole story of the statement's limits:

- **The indexed value is of type `jsonb`,** so the access method has to have a default operator class for `jsonb`. `btree` (the default), `hash` and `gin` do. `gist`, `spgist` and `brin` do not, and asking for one of them fails with `data type jsonb has no default operator class for access method "gist"`.
- **A collation cannot be applied to it.** `jsonb` carries no collation, so `COLLATE` on a bag read fails with `collations are not supported by type jsonb`.
- **The value can never be an included column.** `INCLUDE` is refused with `expressions are not supported in included columns`, and a property read is always an expression, so `INCLUDE` does not apply to one.

An index on a **promoted** property is different, because the value is then the column, not the expression. `\d` reports `btree (year)` rather than `btree ((properties.'year'))`. The access method, the operator class and the collation are limited only by the column's own type, so `brin (year)` on an `int` column and `btree (name COLLATE "C")` on a `text` column both work. **Promoting a property to a column** covers when to promote in the first place.

Three more rules:

- The label is looked up in the graph named by `graph_path`; its name cannot be qualified.
- An index on a parent label covers that label's own rows only. Child labels are not indexed, with or without `ONLY`. See **Label inheritance**.
- `IF NOT EXISTS` needs a name to check, so it cannot be combined with an unnamed index.

Parameters

UNIQUE	Reject a row whose indexed value duplicates an existing one, checking the rows already there when the index is built. For a uniqueness rule expressed in Cypher's own terms, see `CREATE CONSTRAINT`.
CONCURRENTLY	Build the index without locking out writes to the label, at the cost of a longer build and two table scans.
IF NOT EXISTS index_name	Issue a notice instead of an error when a relation of that name already exists. Nothing is built, and nothing checks that the existing index is the one you asked for.
index_name	Name for the index. Left out, a name is derived from the label and the indexed properties: `movie` and `year` give `movie_year_idx`.

© 2026 SKAI Worldwide Co., Ltd.

ONLY	Accepted, and without effect on an ordinary label. It suppresses recursion into partitions, and a label is an inheritance parent rather than a partitioned table.
label_name	Label whose rows are indexed, in the graph named by `graph_path`.
USING method	Index access method. Over a property read from the bag the choice is `btree` (the default), `hash` or `gin`. Over a promoted column, any method with a default operator class for that column's type.
property	A property of the label, written the way Cypher writes it: `title`, or `name.first` for a key inside a nested map.
(expression)	A parenthesised Cypher expression over the label's properties, for example `(year / 10)`.
COLLATE collation	Collation for the indexed value. Usable only where that value is a promoted column of a collatable type.
opclass	Operator class for the indexed value, for example `jsonb_path_ops` for a `gin` index. Operator class parameters are not accepted.
ASC | DESC | NULLS FIRST | NULLS LAST	Sort order stored in the index. `ASC` and `NULLS LAST` are the defaults, and `NULLS FIRST` is the default under `DESC`.
NULLS [NOT] DISTINCT	For a unique index, whether two nulls count as different values. `NULLS DISTINCT`, the default, lets an index hold many nulls.
WITH (storage_parameter [= value] [, ...])	Storage parameters for the index, such as `fillfactor` for btree or `fastupdate` for gin.
TABLESPACE tablespace_name	Tablespace for the index. Without it, `default_tablespace` decides.
WHERE condition	Build a partial index, covering only the rows the condition accepts.

Examples

The examples build their own graph, `indexdemo`, rather than using **the example dataset**, and drop it at the end.

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE GRAPH indexdemo;
SET graph_path = indexdemo;
CREATE VLABEL movie;
CREATE VLABEL person;
CREATE (:movie {title: 'Arrival', year: 2016, genre: 'Sci-Fi'}),
       (:movie {title: 'Dune', year: 2021, genre: 'Sci-Fi'}),
       (:movie {title: 'Gattaca', year: 1997, genre: 'Sci-Fi'}),
       (:movie {title: 'Sicario', year: 2015, genre: 'Thriller'}),
       (:movie {title: 'Prisoners', year: 2013, genre: 'Thriller'}),
       (:movie {title: 'Enemy', year: 2013, genre: 'Thriller'});
CREATE (:person {name: 'Denis Villeneuve'}), (:person {name: 'Roger Deakins'});
```

Index a property

```
CREATE PROPERTY INDEX movie_title_idx ON movie (title);
```

```
CREATE PROPERTY INDEX
```

That produced a btree over an expression:

```
\d indexdemo.movie
```

```
                                                    Table "indexdemo.movie"
   Column    |  Type    | Collation | Nullable |                                                  Defa
ult
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------
--
 id          | graphid |           | not null | graphid(graph_labid('indexdemo.movie'::cstring)
, nextval('indexdemo.movie_id_seq'::regclass))
 properties | jsonb    |           | not null | jsonb_build_object()
Indexes:
    "movie_pkey" PRIMARY KEY, btree (id)
    "movie_title_idx" btree ((properties.'title'))
Inherits: indexdemo.ag_vertex
```

`\dGi` lists the same index as the statement that would recreate it:

```
\dGi indexdemo.*
```

```
                                              List of property indexes
   Graph    | LabelName |    IndexName    | Unique | Owner |                                 Index
def
-----------+-----------+-----------------+--------+-------
+-----------------------------------------------------------------
 indexdemo | movie     | movie_title_idx | f      | agens | CREATE PROPERTY INDEX movie_title_
idx ON movie USING btree (title)
(1 row)
```

The planner matches an equality on `title` to that expression. Six vertices are far too few for it to prefer an index scan, so the setting below makes the plan visible:

© 2026 SKAI Worldwide Co., Ltd.

```
SET enable_seqscan = off;
EXPLAIN (COSTS OFF) MATCH (m:movie {title: 'Dune'}) RETURN m.year;
```

```
                      QUERY PLAN
------------------------------------------------------
 Index Scan using movie_title_idx on movie m
   Index Cond: (properties.'title' = '"Dune"'::jsonb)
(2 rows)
```

```
RESET enable_seqscan;
```

Let the index name itself

```
CREATE PROPERTY INDEX ON movie (year);
```

```
\dGi indexdemo.movie_year_idx
```

```
                                          List of property indexes
   Graph    | LabelName |   IndexName    | Unique | Owner |                              Indexde
f
-----------+-----------+----------------+--------+-------
+-------------------------------------------------------------------
 indexdemo | movie     | movie_year_idx | f      | agens | CREATE PROPERTY INDEX movie_year_id
x ON movie USING btree (year)
(1 row)
```

Choose an access method

`gin` and `hash` are the alternatives to btree over a property read:

```
CREATE PROPERTY INDEX movie_genre_gin ON movie USING gin (genre);
CREATE PROPERTY INDEX movie_genre_hash ON movie USING hash (genre);
```

```
CREATE PROPERTY INDEX
CREATE PROPERTY INDEX
```

The other three access methods have nothing to do with AgensGraph. They have no default operator class for `jsonb`:

```
CREATE PROPERTY INDEX movie_genre_gist ON movie USING gist (genre);
```

```
ERROR:  data type jsonb has no default operator class for access method "gist"
HINT:  You must specify an operator class for the index or define a default operator class for
the data type.
```

`spgist` and `brin` fail the same way.

© 2026 SKAI Worldwide Co., Ltd.

Reject duplicate values

```
CREATE UNIQUE PROPERTY INDEX movie_title_uniq ON movie (title);
CREATE (:movie {title: 'Dune', year: 1984});
```

```
ERROR:  duplicate key value violates unique constraint "movie_title_uniq"
DETAIL:  Key ((properties.'title'))=("Dune") already exists.
```

The `Key` in that message is the index expression, which is how you tell a unique property index apart from a unique constraint on a column.

Index part of a label

```
CREATE PROPERTY INDEX movie_recent_idx ON movie (title) WHERE (year > 2010);
```

```
CREATE PROPERTY INDEX
```

Index several properties, and an expression

```
CREATE PROPERTY INDEX movie_genre_year_idx ON movie (genre, year DESC NULLS LAST);
CREATE PROPERTY INDEX movie_decade_idx ON movie ((year / 10));
CREATE PROPERTY INDEX movie_title_ops ON movie USING gin (title jsonb_path_ops);
```

Every one of them is an expression index, and `\d` shows exactly what was built, including the Cypher operator that the arithmetic index carries:

```
\d indexdemo.movie
```

```
                                                    Table "indexdemo.movie"
   Column    |  Type   | Collation | Nullable |
Default
-------------+---------+-----------+----------
+-----------------------------------------------------------------------------------------------
--
 id          | graphid |           | not null |
graphid(graph_labid('indexdemo.movie'::cstring), nextval('indexdemo.movie_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "movie_pkey" PRIMARY KEY, btree (id)
    "movie_decade_idx" btree (((properties.'year') `/` cypher_to_jsonb(10)))
    "movie_genre_gin" gin ((properties.'genre'))
    "movie_genre_hash" hash ((properties.'genre'))
    "movie_genre_year_idx" btree ((properties.'genre'), (properties.'year') DESC NULLS LAST)
    "movie_recent_idx" btree ((properties.'title')) WHERE (properties.'year') >
cypher_to_jsonb(2010)
    "movie_title_idx" btree ((properties.'title'))
    "movie_title_ops" gin ((properties.'title') jsonb_path_ops)
    "movie_title_uniq" UNIQUE, btree ((properties.'title'))
    "movie_year_idx" btree ((properties.'year'))
Inherits: indexdemo.ag_vertex
```

Index a promoted property

A label whose property is promoted to a column indexes the column instead:

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE VLABEL title (year int GENERATED, name text GENERATED);
CREATE (:title {name: 'Arrival', year: 2016}), (:title {name: 'Dune', year: 2021}), (:title {n
ame: 'Gattaca', year: 1997});
CREATE PROPERTY INDEX title_year_idx ON title (year);
```

```
\d indexdemo.title
```

```
                                                    Table "indexdemo.title"
   Column    |  Type   | Collation | Nullable |                                                  Defa
ult
-------------+---------+-----------+----------
+-----------------------------------------------------------------------------------------------
--
 id          | graphid |           | not null | graphid(graph_labid('indexdemo.title'::cstring)
, nextval('indexdemo.title_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
 year        | integer |           |          | generated always as (ag_property_text(propertie
s, 'year'::text, 'n'::"char")::integer) stored
 name        | text    |           |          | generated always as (ag_property_text(propertie
s, 'name'::text, 's'::"char")) stored
Indexes:
    "title_pkey" PRIMARY KEY, btree (id)
    "title_year_idx" btree (year)
Inherits: indexdemo.ag_vertex
```

`btree (year)`, not `btree ((properties.'year'))`. The plan reads the same way, comparing integers rather than `jsonb`:

```
SET enable_seqscan = off;
EXPLAIN (COSTS OFF) MATCH (t:title) WHERE t.year > 2010 RETURN t.name;
```

```
                QUERY PLAN
------------------------------------------
 Index Scan using title_year_idx on title t
   Index Cond: (year > 2010)
(2 rows)
```

```
RESET enable_seqscan;
```

Because the indexed value is the column, a collation, an operator class and an access method the column's type supports all become available:

```
CREATE PROPERTY INDEX title_name_coll ON title (name COLLATE "C");
CREATE PROPERTY INDEX title_name_pat ON title (name text_pattern_ops);
CREATE PROPERTY INDEX title_year_brin ON title USING brin (year);
```

```
\d indexdemo.title
```

```
                                                    Table "indexdemo.title"
   Column    |  Type   | Collation | Nullable |                                                  Defa
ult
```


© 2026 SKAI Worldwide Co., Ltd.

```
------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------
--
 id         | graphid |           | not null | graphid(graph_labid('indexdemo.title'::cstring)
, nextval('indexdemo.title_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
 year       | integer |           |          | generated always as (ag_property_text(propertie
s, 'year'::text, 'n'::"char")::integer) stored
 name       | text    |           |          | generated always as (ag_property_text(propertie
s, 'name'::text, 's'::"char")) stored
Indexes:
    "title_pkey" PRIMARY KEY, btree (id)
    "title_name_coll" btree (name COLLATE "C")
    "title_name_pat" btree (name text_pattern_ops)
    "title_year_brin" brin (year)
    "title_year_idx" btree (year)
Inherits: indexdemo.ag_vertex
```

None of those three is accepted over a property read from the bag.

Refusals worth recognizing

`INCLUDE` cannot be used at all, promoted column or not, because a property is always written as an expression:

```
CREATE PROPERTY INDEX title_name_incl ON title (name) INCLUDE (year);
```

```
ERROR:  expressions are not supported in included columns
```

Operator class parameters are not part of the statement:

```
CREATE PROPERTY INDEX movie_title_op ON movie USING gin (title jsonb_ops (fastupdate = off));
```

```
ERROR:  syntax error at or near "("
LINE 1: ...ovie_title_op ON movie USING gin (title jsonb_ops (fastupdat...
                                                             ^
```

A nested path works only where the property holds a map. `person.name` is a string here, so reaching into it fails while the index is being built:

```
CREATE PROPERTY INDEX person_first_idx ON person (name.first);
```

```
ERROR:  map or list is expected but scalar value
```

And a name already taken is reported, not replaced:

```
CREATE PROPERTY INDEX IF NOT EXISTS movie_title_idx ON movie (title);
```

```
NOTICE:  relation "movie_title_idx" already exists, skipping
CREATE PROPERTY INDEX
```

Finally, drop the graph the examples built:

© 2026 SKAI Worldwide Co., Ltd.

```
DROP GRAPH indexdemo CASCADE;
```

Notes

`CONCURRENTLY` is accepted and does what it does for `CREATE INDEX`: it builds without locking writes out. `ONLY` is accepted too and changes nothing on an ordinary label. An index on a parent covers the parent's rows only either way, so a hierarchy needs one index per label that will be scanned.

How a promoted key resolves here

On a label with **promoted properties**, the two forms of `( ... )` resolve differently:

What you write	What is indexed
`( year )`, a bare key	The promoted column `year`
`( (lower(title)) )`, an expression	`lower(properties.'title')`, over the jsonb bag

A query over a promoted key reads the column in both cases, so the bare form is the one that gives the query an index to use. To index an expression over a promoted key, write it as SQL over the column — `CREATE INDEX ON g.film (lower(title))` — which `\dGi` still lists.

`\dGi` renders its `Indexdef` column in Cypher form, so both spell the key the same way. Read `pg_get_indexdef`, or `\d` on the label's table, to see which of the two an index is over: there a bare promoted key reads `btree (year)` and the property form reads `btree ((properties.'year'))`.

Compatibility

ISO/IEC 39075 (GQL)

GQL has no index DDL. Its catalog-modifying statements are limited to schemas, graphs and graph types (§12). An index has no standard spelling there, so `CREATE PROPERTY INDEX` is AgensGraph's own.

PostgreSQL

`CREATE PROPERTY INDEX` corresponds to `CREATE INDEX` on an expression, and `UNIQUE`, `CONCURRENTLY`, `USING`, `COLLATE`, the operator class, the sort order, `NULLS [NOT] DISTINCT`, `WITH`, `TABLESPACE` and `WHERE` all mean what they mean there. Three differences matter. The index target is written as a Cypher property rather than a column or a parenthesised SQL expression, so the parser builds the `properties` read for you. `INCLUDE` and operator class parameters, which `CREATE INDEX` accepts, cannot be used. And **`DROP PROPERTY INDEX`** takes only a property index, while `DROP INDEX` takes any index of a label, a property index included.

See also

- **`DROP PROPERTY INDEX`**
- **`REINDEX VLABEL`** and **`REINDEX ELABEL`**
- **`CREATE CONSTRAINT`**
- Promoting a property to a column
- How a graph is stored

© 2026 SKAI Worldwide Co., Ltd.

DROP PROPERTY INDEX

Removes a property index.

Synopsis

DROP PROPERTY INDEX [**IF EXISTS**] index_name [, ...] [**CASCADE** | **RESTRICT**]

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`DROP PROPERTY INDEX` drops one or more indexes created by `CREATE PROPERTY INDEX`, and nothing else.

- The name is resolved in the graph named by `graph_path`, and **cannot be qualified**. A name with a schema on it is rejected with `improper property index name`. To drop an index in another graph, change `graph_path` first.
- Only a property index is accepted. Any other index on the label is refused with `"<name>" is not property index`: its primary key, an edge label's endpoint indexes, an index created with SQL.
- `DROP INDEX`, by contrast, takes any index of a label, a property index included, so an index made by either statement can be dropped by either.
- An index that belongs to a constraint is dropped with the constraint, by `DROP CONSTRAINT`.

Parameters

IF EXISTS	Issue a notice instead of an error for an index that does not exist. Other names in the same statement are still dropped.
index_name	Unqualified name of the property index to drop. Several names may be given, separated by commas, and they need not be on the same label.
CASCADE	Drop the objects that depend on the index.
RESTRICT	Refuse if anything depends on the index. This is the default. An index rarely has dependents, so the two behave alike most of the time.

Examples

The examples build their own graph, `unindexed`, rather than using **the example dataset**, and drop it at the end.

```
CREATE GRAPH unindexed;
SET graph_path = unindexed;
CREATE VLABEL movie;
CREATE VLABEL person;
CREATE (:movie {title: 'Arrival', year: 2016}), (:person {name: 'Denis Villeneuve'});
CREATE PROPERTY INDEX movie_title_idx ON movie (title);
CREATE PROPERTY INDEX movie_year_idx ON movie (year);
CREATE PROPERTY INDEX person_name_idx ON person (name);
```


© 2026 SKAI Worldwide Co., Ltd.

```
\dGi unindexed.*
```

```
                                                 List of property indexes
   Graph   | LabelName |    IndexName    | Unique | Owner |                              Index
def
-----------+-----------+-----------------+--------+-------
+-------------------------------------------------------------------------
 unindexed | movie     | movie_title_idx | f      | agens | CREATE PROPERTY INDEX movie_title_
idx ON movie USING btree (title)
 unindexed | movie     | movie_year_idx  | f      | agens | CREATE PROPERTY INDEX movie_year_i
dx ON movie USING btree (year)
 unindexed | person    | person_name_idx | f      | agens | CREATE PROPERTY INDEX person_name_
idx ON person USING btree (name)
(3 rows)
```

Drop one index

```
DROP PROPERTY INDEX movie_title_idx;
```

```
DROP PROPERTY INDEX
```

A second attempt is an error, because the index is gone:

```
DROP PROPERTY INDEX movie_title_idx;
```

```
ERROR:  index "movie_title_idx" does not exist
```

`IF EXISTS` turns that into a notice:

```
DROP PROPERTY INDEX IF EXISTS movie_title_idx;
```

```
NOTICE:  index "movie_title_idx" does not exist, skipping
DROP PROPERTY INDEX
```

Drop several indexes at once

The names need not belong to the same label:

```
DROP PROPERTY INDEX movie_year_idx, person_name_idx;
```

```
DROP PROPERTY INDEX
```

```
\dGi unindexed.*
```

```
No matching property indexes found.
```


© 2026 SKAI Worldwide Co., Ltd.

Refusals worth recognizing

The name cannot carry the graph:

```
CREATE PROPERTY INDEX movie_title_idx ON movie (title);
DROP PROPERTY INDEX unindexed.movie_title_idx;
```

```
ERROR:  improper property index name
```

An index the statement does not own is refused. A label's primary key is one:

```
DROP PROPERTY INDEX movie_pkey;
```

```
ERROR:  "movie_pkey" is not property index
```

`DROP INDEX` takes a property index as well, under its schema-qualified name:

```
DROP INDEX unindexed.movie_title_idx;
```

```
DROP INDEX
```

Finally, drop the graph the examples built:

```
DROP GRAPH unindexed CASCADE;
```

Notes

Because the name is resolved in `graph_path`, an index that exists in another graph reads as missing rather than as belonging to someone else:

```
CREATE GRAPH oneplace;
SET graph_path = oneplace;
CREATE VLABEL movie;
CREATE PROPERTY INDEX movie_title_idx ON movie (title);
CREATE GRAPH otherplace;
SET graph_path = otherplace;
DROP PROPERTY INDEX movie_title_idx;
```

```
ERROR:  index "movie_title_idx" does not exist
```

Two graphs may hold indexes of the same name, so check `\dGi` before assuming which one a bare name will reach.

```
DROP GRAPH oneplace CASCADE;
DROP GRAPH otherplace CASCADE;
```

To turn a label's indexes off without dropping them, for a bulk load say, use `ALTER VLABEL ... DISABLE INDEX` and `REINDEX VLABEL` instead. That keeps the definitions and only invalidates them.

© 2026 SKAI Worldwide Co., Ltd.

Compatibility

ISO/IEC 39075 (GQL)

GQL has no index DDL. Its catalog-modifying statements are limited to schemas, graphs and graph types (§12), so `DROP PROPERTY INDEX` is AgensGraph's own.

PostgreSQL

`DROP PROPERTY INDEX` corresponds to `DROP INDEX`, and `IF EXISTS`, the name list and `CASCADE` / `RESTRICT` behave as they do there. It differs in three ways. The name is resolved in `graph_path` rather than `search_path` and may not be schema-qualified. `CONCURRENTLY` is not accepted. And the two statements are strictly partitioned: each refuses the other's indexes, so a property index can only be dropped by this statement.

See also

- `CREATE PROPERTY INDEX`
- `REINDEX VLABEL` and `REINDEX ELABEL`
- `DROP CONSTRAINT`
- How a graph is stored

REINDEX VLABEL and REINDEX ELABEL

Rebuilds every index on a label, which is also how a label whose indexes were disabled gets them back.

Synopsis

REINDEX [(option [, ...])] **VLABEL** label_name
REINDEX [(option [, ...])] **ELABEL** label_name

where option is one of:

CONCURRENTLY [boolean]
TABLESPACE new_tablespace
VERBOSE [boolean]

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`REINDEX VLABEL` rebuilds every index on a vertex label and `REINDEX ELABEL` every index on an edge label. That covers the label's own indexes, every property index on it, and the index on its TOAST relation. Rebuilding marks an invalid index valid again, which makes this statement the counterpart to `DISABLE INDEX`.

There is no `ENABLE INDEX`. The load-then-build sequence is `ALTER VLABEL <label> DISABLE INDEX`, then the load, then `REINDEX VLABEL <label>`.

- The label is looked up in the graph named by `graph_path`. The name **cannot be qualified**: `REINDEX VLABEL rebuildme.movie` is a syntax error.
- Each statement takes one kind of label. `REINDEX VLABEL` on an edge label fails with `REINDEX VLABEL cannot REINDEX edge label`, and the reverse with the matching message.
- Options go in parentheses before the label keyword. `REINDEX VLABEL CONCURRENTLY movie` is a syntax error here, though `REINDEX TABLE` accepts that bare-keyword spelling.

© 2026 SKAI Worldwide Co., Ltd.

- The old spelling `REINDEX LABEL` does not exist. It is a syntax error at `LABEL`.

Parameters

option	`CONCURRENTLY`, `TABLESPACE` or `VERBOSE`, in parentheses and separated by commas. They mean what they mean for `REINDEX`: build without locking writes out, move the rebuilt indexes to another tablespace, and report each index as it is finished. `CONCURRENTLY` and `VERBOSE` also accept an explicit boolean, so `(VERBOSE false)` is the same as leaving `VERBOSE` out.
label_name	Label whose indexes are rebuilt, in the graph named by `graph_path`.

Examples

The examples build their own graph, `rebuildme`, rather than using **the example dataset**, and drop it at the end.

```
CREATE GRAPH rebuildme;
SET graph_path = rebuildme;
CREATE VLABEL movie;
CREATE ELABEL directed;
CREATE PROPERTY INDEX movie_title_idx ON movie (title);
CREATE (:movie {title: 'Arrival'});
```

Build a label's indexes after a bulk load

Turning the indexes off leaves them in place but invalid, so the load does not maintain them:

```
ALTER VLABEL movie DISABLE INDEX;
```

```
\d rebuildme.movie
```

```
                                                    Table "rebuildme.movie"
   Column    |  Type    | Collation | Nullable |                                                Defa
ult
-------------+----------+-----------+----------
+-----------------------------------------------------------------------------------------------
--
 id          | graphid |           | not null | graphid(graph_labid('rebuildme.movie'::cstring)
, nextval('rebuildme.movie_id_seq'::regclass))
 properties | jsonb    |           | not null | jsonb_build_object()
Indexes:
    "movie_pkey" PRIMARY KEY, btree (id) INVALID
    "movie_title_idx" btree ((properties.'title')) INVALID
Inherits: rebuildme.ag_vertex
```

Both indexes are marked `INVALID`: the label's primary key and the property index alike. One statement builds them:

```
REINDEX VLABEL movie;
```


© 2026 SKAI Worldwide Co., Ltd.

```
REINDEX
```

```
\d rebuildme.movie
```

```
                                                    Table "rebuildme.movie"
   Column    |  Type   | Collation | Nullable |                                                Defa
ult
-------------+---------+-----------+----------
+-----------------------------------------------------------------------------------------------------
--
 id          | graphid |           | not null | graphid(graph_labid('rebuildme.movie'::cstring)
, nextval('rebuildme.movie_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "movie_pkey" PRIMARY KEY, btree (id)
    "movie_title_idx" btree ((properties.'title'))
Inherits: rebuildme.ag_vertex
```

Watch each index as it is rebuilt

```
ALTER ELABEL directed DISABLE INDEX;
REINDEX (VERBOSE) ELABEL directed;
```

```
INFO:  index "pg_toast_60285_index" was reindexed
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  index "directed_id_idx" was reindexed
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  index "directed_start_idx" was reindexed
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  index "directed_end_idx" was reindexed
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
REINDEX
```

An edge label's three indexes are all rebuilt, and so is the index on the TOAST relation that holds oversized property maps. The TOAST index name carries the table's OID, so it differs from run to run.

Rebuild without locking writes out

```
REINDEX (CONCURRENTLY) VLABEL movie;
```

```
REINDEX
```

```
REINDEX (TABLESPACE pg_default) VLABEL movie;
```

```
REINDEX
```

Refusals worth recognizing

Each statement takes one kind of label:

© 2026 SKAI Worldwide Co., Ltd.

```
REINDEX VLABEL directed;
```

```
ERROR:  REINDEX VLABEL cannot REINDEX edge label
```

```
REINDEX ELABEL movie;
```

```
ERROR:  REINDEX ELABEL cannot REINDEX vertex label
```

There is no `REINDEX LABEL`:

```
REINDEX LABEL movie;
```

```
ERROR:  syntax error at or near "LABEL"
LINE 1: REINDEX LABEL movie;
                ^
```

The name cannot carry the graph:

```
REINDEX VLABEL rebuildme.movie;
```

```
ERROR:  syntax error at or near "."
LINE 1: REINDEX VLABEL rebuildme.movie;
                                 ^
```

And `CONCURRENTLY` belongs inside the parentheses:

```
REINDEX VLABEL CONCURRENTLY movie;
```

```
ERROR:  syntax error at or near "CONCURRENTLY"
LINE 1: REINDEX VLABEL CONCURRENTLY movie;
                       ^
```

A label that is not there is reported by its qualified name, because `graph_path` has already been applied to it:

```
REINDEX VLABEL nosuch;
```

```
ERROR:  relation "rebuildme.nosuch" does not exist
```

Finally, drop the graph the examples built:

```
DROP GRAPH rebuildme CASCADE;
```


© 2026 SKAI Worldwide Co., Ltd.

Notes

`REINDEX TABLE` on a label's table does the same job and needs no setting to do it, so `REINDEX TABLE rebuildme.movie` also brings a disabled label's indexes back. Prefer the label spellings anyway. They check that the label is of the kind you named, and they take a name resolved in `graph_path` rather than a schema-qualified one.

`REINDEX SCHEMA` on a graph's schema rebuilds every index in the graph, and `REINDEX INDEX` rebuilds one by name. Both are the ordinary SQL statements, with no label-aware form.

Compatibility

ISO/IEC 39075 (GQL)

GQL has no index DDL and nothing that rebuilds one. Its catalog-modifying statements are limited to schemas, graphs and graph types (§12), so these two spellings are AgensGraph's own.

PostgreSQL

`REINDEX VLABEL` and `REINDEX ELABEL` are `REINDEX TABLE` with a label-aware target. The options are `REINDEX`'s own options and mean the same things. Two spelling differences matter. The target name is resolved in `graph_path` and cannot be schema-qualified. And `CONCURRENTLY` is only accepted as a parenthesised option, never as a bare keyword after the label keyword.

See also

- `ALTER VLABEL` and `ALTER ELABEL`
- `CREATE PROPERTY INDEX`
- `DROP PROPERTY INDEX`
- `CREATE VLABEL`

CREATE CONSTRAINT

Requires a property of a label to be unique, or requires every element of the label to satisfy a condition.

Synopsis

CREATE CONSTRAINT [constraint_name] **ON** label_name **ASSERT** property **IS UNIQUE**
CREATE CONSTRAINT [constraint_name] **ON** label_name **ASSERT** condition

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

There are two forms, and they become two different things on disk.

`ASSERT property IS UNIQUE` becomes an exclusion constraint over the property read: `EXCLUDE USING btree ((properties.'title') WITH =)`. A write whose value already exists is rejected as a conflict.

`ASSERT condition` becomes a check constraint over the same kind of read: `CHECK ((properties.'age')::integer > 9)`. A write whose row fails the condition is rejected.

The name is optional in both forms. Left out, one is derived: `<label>_unique_constraint` for the unique form and `<label>_properties_check` for the check form. A numeric suffix is added if that name is taken. A name you supply has to be free in the graph's schema, because the unique form creates an index of that name.

Other rules:

- The label is looked up in the graph named by `graph_path`; its name cannot be qualified.

© 2026 SKAI Worldwide Co., Ltd.

- Both forms work on a vertex label and on an edge label.
- `CREATE OR REPLACE CONSTRAINT` is parsed and then refused with `CREATE OR REPLACE CONSTRAINT is not supported`. To change a constraint, drop it and create it again.
- A constraint is created over the rows already in the label, so an existing row that violates it makes the statement fail.
- A check constraint is a SQL check constraint, so **a missing property satisfies it.** The condition evaluates to unknown for an absent property, and a check constraint accepts unknown. Use the constraint to restrict values, not to require a key.

How a graph is stored covers the on-disk mapping in one screen.

Parameters

constraint_name	Name for the constraint. Optional; a name is derived from the label and the form when you leave it out. For the unique form it also names the index, so it must not collide with an existing relation in the graph.
label_name	Label the constraint applies to, in the graph named by `graph_path`.
property	The property that must be unique, written the way Cypher writes it: `title`, or `name.first` for a key inside a nested map.
condition	A Cypher condition over the label's properties. It is evaluated once per written element.

Examples

The examples build their own graph, `constrain`, rather than using the example dataset, and drop it at the end.

```
CREATE GRAPH constrain;
SET graph_path = constrain;
CREATE VLABEL film;
CREATE ELABEL rated;
```

Require a property to be unique

```
CREATE CONSTRAINT film_title_uniq ON film ASSERT title IS UNIQUE;
```

```
CREATE CONSTRAINT
```

That produced an exclusion constraint over the property read:

```
\d constrain.film
```

```
                                                Table "constrain.film"
  Column   |  Type   | Collation | Nullable |                                      Defau
```


© 2026 SKAI Worldwide Co., Ltd.

```
lt
------------+---------+-----------+----------
+-----------------------------------------------------------------------------------------------
 id         | graphid |           | not null | graphid(graph_labid('constrain.film'::cstring),
nextval('constrain.film_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "film_pkey" PRIMARY KEY, btree (id)
    "film_title_uniq" EXCLUDE USING btree ((properties.'title') WITH =)
Inherits: constrain.ag_vertex
```

The first write goes in:

```
CREATE (:film {title: 'Arrival', year: 2016});
```

```
UPDATE 1
```

The second is rejected, and the message names the constraint and the value:

```
CREATE (:film {title: 'Arrival', year: 2021});
```

```
ERROR:  conflicting key value violates exclusion constraint "film_title_uniq"
DETAIL:  Key ((properties.'title'))=("Arrival") conflicts with existing key ((properties.'titl
e'))=("Arrival").
```

Leave the constraint unnamed

```
CREATE CONSTRAINT ON film ASSERT year IS UNIQUE;
```

`\dGv` shows both constraints in the terms they were written in, and the derived name the second one was given:

```
\dGv constrain.film
```

```
          List of labels
   Graph    | Name |  Type  | Owner
------------+------+--------+-------
 constrain | film | vertex | agens
(1 row)

Vertex label "constrain.film"
--
Constraints:
    "film_title_uniq" ASSERT (title) IS UNIQUE
    "film_unique_constraint" ASSERT (year) IS UNIQUE
Inherits: constrain.ag_vertex
```

Restrict what a property may hold

```
CREATE VLABEL teen;
CREATE CONSTRAINT teen_age ON teen ASSERT (10 <= age::int AND age::int <= 19);
```


© 2026 SKAI Worldwide Co., Ltd.

```
\d constrain.teen
```

```
                                                    Table "constrain.teen"
   Column    |  Type    | Collation | Nullable |                                      Defau
lt
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------
 id          | graphid |           | not null | graphid(graph_labid('constrain.teen'::cstring),
nextval('constrain.teen_id_seq'::regclass))
 properties | jsonb    |           | not null | jsonb_build_object()
Indexes:
    "teen_pkey" PRIMARY KEY, btree (id)
Check constraints:
    "teen_age" CHECK (10 <= ((properties.'age')::integer) AND ((properties.'age')::integer) <=
19)
Inherits: constrain.ag_vertex
```

A row inside the range is written:

```
CREATE (:teen {name: 'Kim', age: 15});
```

```
UPDATE 1
```

One outside it is not:

```
CREATE (:teen {name: 'Lee', age: 20});
```

```
ERROR:  new row for relation "teen" violates check constraint "teen_age"
DETAIL:  Failing row contains (5.2, {"age": 20, "name": "Lee"}).
```

A vertex with **no** `age` at all is accepted, because the condition is unknown rather than false:

```
CREATE (:teen {name: 'Bo'});
MATCH (t:teen) RETURN t.name AS name, t.age AS age ORDER BY name;
```

```
 name  | age
-------+-----
 "Bo"  |
 "Kim" | 15
(2 rows)
```

Compare without a cast

Writing the comparison against the raw property compares `jsonb` values instead of integers:

```
CREATE CONSTRAINT ON teen ASSERT age > 9;
```

```
\dGv constrain.teen
```


© 2026 SKAI Worldwide Co., Ltd.

```
          List of labels
   Graph    | Name |  Type  | Owner
------------+------+--------+-------
 constrain | teen | vertex | agens
(1 row)

Vertex label "constrain.teen"
--
Constraints:
    "teen_age" ASSERT (10 <= ((age)::integer) AND ((age)::integer) <= 19)
    "teen_properties_check" ASSERT ((age) > cypher_to_jsonb(9))
Inherits: constrain.ag_vertex
```

Both forms reject `{age: 'twelve'}`, but they report it differently. The cast form raises `invalid input syntax for type integer: "twelve"`. The uncast form reports a violated check constraint. Cast when you want the property's type enforced as well as its range.

Constrain an edge label

```
CREATE CONSTRAINT rated_pair ON rated ASSERT source IS UNIQUE;
```

```
\d constrain.rated
```

```
                                                    Table "constrain.rated"
   Column    |  Type   | Collation | Nullable |                                                    Defa
ult
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------------
--
 id         | graphid |           | not null | graphid(graph_labid('constrain.rated'::cstring)
, nextval('constrain.rated_id_seq'::regclass))
 start      | graphid |           | not null |
 end        | graphid |           | not null |
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "rated_end_idx" btree ("end", start)
    "rated_id_idx" brin (id)
    "rated_pair" EXCLUDE USING btree ((properties.'source') WITH =)
    "rated_start_idx" btree (start, "end")
Inherits: constrain.ag_edge
```

The constraint sits beside the label's three ordinary indexes.

Add a constraint to a label that already has rows

The constraint is checked against what is there, so it cannot be created over rows that break it. The two forms report that differently:

```
CREATE VLABEL rerun;
CREATE (:rerun {title: 'Arrival'}), (:rerun {title: 'Arrival'});
CREATE CONSTRAINT rerun_t ON rerun ASSERT title IS UNIQUE;
```

```
ERROR:  could not create exclusion constraint "rerun_t"
DETAIL:  Key ((properties.'title'))=("Arrival") conflicts with key ((properties.'title'))=("Ar
rival").
```


© 2026 SKAI Worldwide Co., Ltd.

```
CREATE VLABEL adult;
CREATE (:adult {age: 40});
CREATE CONSTRAINT adult_a ON adult ASSERT (age::int < 20);
```

```
ERROR:  check constraint "adult_a" of relation "adult" is violated by some row
```

Refusals worth recognizing

There is no replace:

```
CREATE OR REPLACE CONSTRAINT film_title_uniq ON film ASSERT title IS UNIQUE;
```

```
ERROR:  CREATE OR REPLACE CONSTRAINT is not supported
```

A unique constraint's name is an index name, so it collides with one:

```
CREATE CONSTRAINT film_title_uniq ON film ASSERT year IS UNIQUE;
```

```
ERROR:  relation "film_title_uniq" already exists
```

And the label has to exist. Cypher does not create one here the way `CREATE (:film)` does:

```
CREATE CONSTRAINT c ON nosuch ASSERT a IS UNIQUE;
```

```
ERROR:  label "nosuch" does not exist
```

Finally, drop the graph the examples built:

```
DROP GRAPH constrain CASCADE;
```

Notes

A unique constraint is an exclusion constraint, not a unique index, so it is not interchangeable with `CREATE UNIQUE PROPERTY INDEX`. Both reject a duplicate. The error differs: `conflicting key value violates exclusion constraint` against `duplicate key value violates unique constraint`. What `\dGv` reports differs. And so does the statement that removes it: a constraint is dropped by **`DROP CONSTRAINT`**, a unique index by **`DROP PROPERTY INDEX`**.

The two forms differ over **label inheritance**, and the difference catches people out. A check constraint is inherited: a child label created after it carries the constraint and is checked by it. A unique constraint is not. A child label gets no exclusion constraint, so the same value can be written once through the parent and once through the child. `MATCH` on the parent then returns both.

> NOTE
>
> To make a property unique across a hierarchy, create the constraint on each label in it. There is nothing that makes one value unique across several labels; a constraint, like the index behind it, covers one table.

© 2026 SKAI Worldwide Co., Ltd.

Compatibility

ISO/IEC 39075 (GQL)

GQL has no constraint DDL. It constrains a graph with a graph type (`<create graph type statement>`, §12.6), which declares which labels and properties a graph's elements may have. A graph type has no per-property uniqueness rule and no check condition. And a graph is only constrained at all if it references a graph type. AgensGraph's graphs are open, so `CREATE CONSTRAINT` is its own. Graph types are tracked as planned in the GQL conformance matrix.

PostgreSQL

The check form corresponds to `ALTER TABLE ... ADD CONSTRAINT ... CHECK`, and the unique form to `ALTER TABLE ... ADD CONSTRAINT ... EXCLUDE USING btree (... WITH =)`. It does not correspond to `UNIQUE`, which cannot be written over an expression. The condition is a Cypher expression rather than a SQL one, so `age` means the property, not a column. `NOT NULL`, `PRIMARY KEY`, `FOREIGN KEY`, `DEFERRABLE` and `NOT VALID`, which `ADD CONSTRAINT` offers, have no spelling here.

See also

- **`DROP CONSTRAINT`**
- **`CREATE PROPERTY INDEX`**
- **`CREATE VLABEL`**
- How a graph is stored

DROP CONSTRAINT

Removes a constraint from a label.

Synopsis

DROP CONSTRAINT constraint_name **ON** label_name

bold keywords · italic placeholders you supply · `[ ]` optional · `...` repeatable. See Syntax conventions.

Description

`DROP CONSTRAINT` removes one constraint from one label. It takes both forms that **`CREATE CONSTRAINT`** creates: the exclusion constraint behind `IS UNIQUE`, and the check constraint behind a condition. It also drops the index a unique constraint owns.

The statement is deliberately small. These are the things it does not have:

- **The name is required.** There is no way to say "whatever constraint is on this property".
- **There is no `IF EXISTS`.** `DROP CONSTRAINT IF EXISTS ...` is a syntax error, so a constraint that is not there is always an error.
- **There is no `CASCADE` or `RESTRICT`,** and no name list.
- The name cannot be qualified, and neither can the label's. The label is looked up in the graph named by `graph_path`.
- It reaches any named constraint on the label's table, not only the ones Cypher created. The label's primary key is one of them. Read the name before you run it.
- A property index is not a constraint. Drop one with **`DROP PROPERTY INDEX`**.

© 2026 SKAI Worldwide Co., Ltd.

Parameters

constraint_name	Name of the constraint to drop. Unqualified. `\dGv`, `\dGe` and `\d` all report it; so does `pg_constraint`.
label_name	Label the constraint is on, in the graph named by `graph_path`.

Examples

The examples build their own graph, `rulebooks`, rather than using **the example dataset**, and drop it at the end.

```
CREATE GRAPH rulebooks;
SET graph_path = rulebooks;
CREATE VLABEL film;
CREATE CONSTRAINT film_title_uniq ON film ASSERT title IS UNIQUE;
CREATE CONSTRAINT ON film ASSERT year IS UNIQUE;
CREATE VLABEL teen;
CREATE CONSTRAINT teen_age ON teen ASSERT (10 <= age::int AND age::int <= 19);
CREATE CONSTRAINT ON teen ASSERT age > 9;
```

Drop a unique constraint

```
DROP CONSTRAINT film_title_uniq ON film;
```

```
DROP CONSTRAINT
```

The constraint and its index are both gone; the unnamed one created afterwards is not:

```
\d rulebooks.film
```

```
                                                    Table "rulebooks.film"
   Column    |  Type    | Collation | Nullable |                                                 Defau
lt
-------------+---------+-----------+----------
+-----------------------------------------------------------------------------------------------
 id          | graphid |           | not null | graphid(graph_labid('rulebooks.film'::cstring),
nextval('rulebooks.film_id_seq'::regclass))
 properties | jsonb    |           | not null | jsonb_build_object()
Indexes:
    "film_pkey" PRIMARY KEY, btree (id)
    "film_unique_constraint" EXCLUDE USING btree ((properties.'year') WITH =)
Inherits: rulebooks.ag_vertex
```

Running it a second time is an error, because there is no `IF EXISTS` to soften it:

```
DROP CONSTRAINT film_title_uniq ON film;
```

```
ERROR:  constraint "film_title_uniq" of relation "film" does not exist
```


© 2026 SKAI Worldwide Co., Ltd.

```
DROP CONSTRAINT IF EXISTS film_title_uniq ON film;
```

```
ERROR:  syntax error at or near "EXISTS"
LINE 1: DROP CONSTRAINT IF EXISTS film_title_uniq ON film;
                           ^
```

Find the name of a constraint you did not name

A constraint created without a name has a derived one. `pg_constraint` lists every constraint on the label's table:

```
SELECT conname FROM pg_constraint WHERE conrelid = 'rulebooks.teen'::regclass ORDER BY
conname;
```

```
         conname
-------------------------
 teen_age
 teen_id_not_null
 teen_pkey
 teen_properties_check
 teen_properties_not_null
(5 rows)
```

`teen_age` is the named check constraint and `teen_properties_check` the derived one. The other three belong to the label's own shape.

Drop a check constraint

```
DROP CONSTRAINT teen_properties_check ON teen;
DROP CONSTRAINT teen_age ON teen;
```

Both are gone, and `\dGv` says so by having nothing to report:

```
\dGv rulebooks.teen
```

```
          List of labels
   Graph    | Name |  Type  | Owner
------------+------+--------+-------
 rulebooks | teen | vertex | agens
(1 row)

Vertex label "rulebooks.teen"
--
Inherits: rulebooks.ag_vertex
```

Refusals worth recognizing

A property index is not a constraint, whatever it is named:

```
CREATE PROPERTY INDEX film_year_idx ON film (year);
DROP CONSTRAINT film_year_idx ON film;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  constraint "film_year_idx" of relation "film" does not exist
```

And the name cannot carry the graph:

```
DROP CONSTRAINT rulebooks.film_unique_constraint ON film;
```

```
ERROR:  syntax error at or near "."
LINE 1: DROP CONSTRAINT rulebooks.film_unique_constraint ON film;
                                 ^
```

Finally, drop the graph the examples built:

```
DROP GRAPH rulebooks CASCADE;
```

Notes

> **WARNING**
>
> `DROP CONSTRAINT` takes any named constraint on the label's table, and a label's primary key is one. `DROP CONSTRAINT film_pkey ON film` succeeds and leaves the label without its primary key on `id`, and no statement puts that back. Check the name against `\dGv` or `pg_constraint` first, and keep to the names `CREATE CONSTRAINT` produced.

To change a constraint, drop it and create it again; `CREATE OR REPLACE CONSTRAINT` is refused.

A check constraint that a child label inherited can only be dropped from the label that declares it:

```
CREATE GRAPH hierarchy;
SET graph_path = hierarchy;
CREATE VLABEL work;
CREATE CONSTRAINT work_y ON work ASSERT (year::int > 1900);
CREATE VLABEL feature INHERITS (work);
```

Naming the child is refused:

```
DROP CONSTRAINT work_y ON feature;
```

```
ERROR:  cannot drop inherited constraint "work_y" of relation "feature"
```

Dropping it from the parent removes it from the whole family:

```
DROP CONSTRAINT work_y ON work;
DROP GRAPH hierarchy CASCADE;
```


© 2026 SKAI Worldwide Co., Ltd.

Compatibility

ISO/IEC 39075 (GQL)

GQL has no constraint DDL, so nothing corresponds to this statement. Its way of constraining a graph is a graph type, dropped whole by `<drop graph type statement>` (§12.7). Graph types are tracked as planned in the GQL conformance matrix.

PostgreSQL

`DROP CONSTRAINT ... ON <label>` corresponds to `ALTER TABLE <table> DROP CONSTRAINT <name>`, with the label name after the constraint name instead of before it. It offers less: no `IF EXISTS`, no `CASCADE` / `RESTRICT`, and no way to drop several constraints in one statement. It reaches exactly as far, though: any named constraint on the label's table, the same set `ALTER TABLE` would reach.

See also

- `CREATE CONSTRAINT`
- `DROP PROPERTY INDEX`
- `ALTER VLABEL` and `ALTER ELABEL`
- How a graph is stored

Functions

All functions

This page lists every function documented in this reference, 126 entries in all, in one alphabetical table. A function can be found by name without knowing which category it belongs to. The **Function** column links to the full entry, which carries the considerations and a worked example. The **Signature** column shows every overload the catalog holds.

Where two entries share a name, they are two functions with two OIDs and two meanings. `length` over a string is not `length` over a path, and the signature column tells them apart.

Function	Signature	Description	Availability
`abs`	`abs ( x jsonb ) -> jsonb` `abs ( x numeric ) -> numeric`	Returns the absolute value of a number.	2.17 and earlier
`acos`	`acos ( x jsonb ) -> jsonb` `acos ( x double precision ) -> double precision`	Returns the arc cosine of `x`, in radians.	2.17 and earlier
`ag_get_graphconstraintdef`	`ag_get_graphconstraintdef ( constraint oid ) -> text`	Reconstructs the `CREATE CONSTRAINT` command for a graph constraint, given its OID.	2.17 and earlier
`ag_get_propindexdef`	`ag_get_propindexdef ( index oid ) -> text`	Reconstructs the `CREATE PROPERTY INDEX` command for a property index, given its OID.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability
ag_property_text	`ag_property_text ( properties jsonb, key text, expected_kind “char” ) -> text`	Returns a property’s value as `text`, but only when the stored value is of the JSON kind asked for.	Added in 2.18
agversion	`agversion ( ) -> text`	The AgensGraph version of the running server.	2.17 and earlier
all	`all ( variable IN list WHERE predicate ) -> boolean`	Returns true when the predicate holds for every element of the list.	2.17 and earlier
allshortestpaths	`allshortestpaths ( pattern ) -> graphpath`	Every shortest path between two vertices.	2.17 and earlier
any	`any ( variable IN list WHERE predicate ) -> boolean`	Returns true when the predicate holds for at least one element of the list.	2.17 and earlier
array_size	`array_size ( list jsonb ) -> integer` `array_size ( list json ) -> integer` `array_size ( list anyarray ) -> integer`	Returns the number of elements in an array.	2.17 and earlier
asin	`asin ( x jsonb ) -> jsonb` `asin ( x double precision ) -> double precision`	Returns the arc sine of `x`, in radians.	2.17 and earlier
atan	`atan ( x jsonb ) -> jsonb` `atan ( x double precision ) -> double precision`	Returns the arc tangent of `x`, in radians.	2.17 and earlier
atan2	`atan2 ( y jsonb, x jsonb ) -> jsonb` `atan2 ( y double precision, x double precision ) -> double precision`	Arc tangent of `y / x` in radians, using both signs to fix the quadrant.	2.17 and earlier
avg	`avg ( expression numeric ) -> numeric`	Returns the arithmetic mean of the non-null values in the group.	2.17 and earlier
cardinality	`cardinality ( value jsonb ) -> jsonb` `cardinality ( p graphpath ) -> jsonb`	Returns the number of elements in a list, entries in a map, or vertices and edges in a path.	Added in 2.18
ceil	`ceil ( x jsonb ) -> jsonb` `ceil ( x numeric ) -> numeric`	Returns the smallest integer that is not less than `x`.	2.17 and earlier
coalesce	`coalesce ( value [, value ] … ) -> jsonb`	Returns the first argument that is not null.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability

collect	`collect ( expression anyelement ) -> jsonb`
	`collect ( v vertex ) -> vertex[]`
	`collect ( e edge ) -> edge[]`
	`collect ( p graphpath ) -> graphpath[]`
		Gathers the group's values into a list; a graph element gives a native array.	2.17 and earlier

cos	`cos ( x jsonb ) -> jsonb` `cos ( x double precision ) -> double precision`	Returns the cosine of an angle given in radians.	2.17 and earlier
cosd	`cosd ( x double precision ) -> double precision`	Returns the cosine of an angle given in degrees.	2.17 and earlier
cot	`cot ( x jsonb ) -> jsonb` `cot ( x double precision ) -> double precision`	Returns the cotangent of an angle given in radians — the reciprocal of the tangent.	2.17 and earlier
count	`count ( * ) -> bigint` `count ( expression ) -> bigint`	Counts rows in the group.	2.17 and earlier
cypher_to_jsonb	`cypher_to_jsonb ( value anyelement ) -> jsonb`	Boxes a value to `jsonb` using Cypher's rules; refuses a graph type.	2.17 and earlier

datetime	`datetime ( ) -> timestamp with time zone`
	`datetime ( timezone "any" ) -> timestamp without time zone`
	`datetime ( year integer, month integer, mday integer, hour integer, min integer, sec integer, millisec integer, nanosec integer, timezone "any" ) -> timestamp without time zone`
		The current transaction timestamp, the time in a named zone, or a timestamp built from parts.	2.17 and earlier

degrees	`degrees ( x jsonb ) -> jsonb` `degrees ( x double precision ) -> double precision`	Converts radians to degrees.	2.17 and earlier

dijkstra	`dijkstra ( pattern, weight ) -> graphpath`
	`dijkstra ( pattern, weight, condition ) -> graphpath`
	`dijkstra ( pattern, weight, LIMIT count ) -> graphpath`
	`dijkstra ( pattern, weight, condition, LIMIT count ) -> graphpath`
		Finds the cheapest path between two vertices, where each edge costs what `weight` evaluates to.	2.17 and earlier

e	`e ( ) -> numeric`	Returns Euler's number as a `numeric`.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability
`edge_to_jsonb`	`edge_to_jsonb ( e edge ) -> jsonb`	Returns an edge's property bag.	2.17 and earlier
`edges`	`edges ( p graphpath ) -> edge[]`	Returns the edges of a path, in order.	2.17 and earlier
`end_id`	`end_id ( e edge ) -> graphid`	Returns the `graphid` of the vertex an edge points at, without reading that vertex.	2.17 and earlier
`end_vertex`	`end_vertex ( e edge ) -> vertex`	Returns the vertex an edge points at.	2.17 and earlier
`endNode`	`endNode ( e edge ) -> vertex`	Returns the vertex an edge points at.	2.17 and earlier
`exists`	`exists ( pattern ) -> boolean` `exists ( property_reference ) -> boolean`	True when a pattern matches at least once, or when a property is present and not null.	2.17 and earlier
`exp`	`exp ( x jsonb ) -> jsonb` `exp ( x numeric ) -> numeric`	Returns e raised to the power `x`.	2.17 and earlier
`floor`	`floor ( x jsonb ) -> jsonb` `floor ( x numeric ) -> numeric`	Returns the largest integer that is not greater than `x`.	2.17 and earlier
`get_last_graph_write_stats`	`get_last_graph_write_stats ( ) -> record`	What the most recent graph write in this session changed, as five counters.	2.17 and earlier
`get_time`	`get_time ( ) -> time without time zone` `get_time ( timezone "any" ) -> time without time zone`	Returns the current time of day, either locally or in a named zone.	2.17 and earlier
`graph_labid`	`graph_labid ( qualified_label cstring ) -> integer`	Returns the label id of a label named as `graph.label`.	2.17 and earlier
`graphid`	`graphid ( labid integer, locid bigint ) -> graphid` `graphid ( id numeric ) -> graphid`	Builds a `graphid` from a label id and a local id, or from its printed form.	2.17 and earlier
`graphid_in_jsonb_array`	`graphid_in_jsonb_array ( id graphid, elements jsonb ) -> boolean`	Returns true when a `graphid` is one of the graph elements in a `jsonb` array.	2.17 and earlier
`graphid_labid`	`graphid_labid ( id graphid ) -> integer`	Returns the label part of a `graphid` — the number before the dot.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability
`graphid_locid`	`graphid_locid ( id graphid ) -> bigint`	Returns the local part of a `graphid` — the number after the dot, taken from the label's sequence.	2.17 and earlier
`head`	`head ( list jsonb ) -> jsonb` `head ( list anyarray ) -> anyelement`	Returns the first element of a list.	2.17 and earlier
`id`	`id ( v vertex ) -> graphid` `id ( e edge ) -> graphid`	Returns a graph element's identity.	2.17 and earlier
`isEmpty`	`isEmpty ( value jsonb ) -> boolean`	Returns true when a list has no elements or a map has no keys.	2.17 and earlier
`jsonb_add`	`jsonb_add ( a jsonb, b jsonb ) -> jsonb`	Implements the backtick `+` operator: addition, or concatenation of two strings or two lists.	2.17 and earlier
`jsonb_array_length`	`jsonb_array_length ( list jsonb ) -> integer`	Returns the number of elements in a `jsonb` array, as an `integer`.	2.17 and earlier
`jsonb_div`	`jsonb_div ( a jsonb, b jsonb ) -> jsonb`	Implements the backtick `/` operator. Integer division when both operands are integers.	2.17 and earlier
`jsonb_mod`	`jsonb_mod ( a jsonb, b jsonb ) -> jsonb`	Implements the backtick `%` operator.	2.17 and earlier
`jsonb_mul`	`jsonb_mul ( a jsonb, b jsonb ) -> jsonb`	Implements the backtick `*` operator.	2.17 and earlier
`jsonb_pow`	`jsonb_pow ( a jsonb, b jsonb ) -> jsonb`	Implements the backtick `^` operator.	2.17 and earlier
`jsonb_sub`	`jsonb_sub ( a jsonb, b jsonb ) -> jsonb`	Implements the backtick binary `-` operator.	2.17 and earlier
`jsonb_uminus`	`jsonb_uminus ( a jsonb ) -> jsonb`	Implements the backtick prefix `-` operator.	2.17 and earlier
`jsonb_uplus`	`jsonb_uplus ( a jsonb ) -> jsonb`	Implements the backtick prefix `+` operator.	2.17 and earlier
`keys`	`keys ( map jsonb ) -> jsonb`	Returns a list of the key names of a map, or of a graph element's properties.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability
`label`	`label ( v vertex ) -> jsonb` `label ( e edge ) -> jsonb`	Returns the name of the label a graph element was created with, as a `jsonb` string.	2.17 and earlier
`labels`	`labels ( v vertex ) -> jsonb`	Returns a list holding the vertex's own label and every label it inherits from, nearest first.	2.17 and earlier
`last`	`last ( list jsonb ) -> jsonb` `last ( list anyarray ) -> anyelement`	Returns the final element of a list.	2.17 and earlier
`left`	`left ( string jsonb, n jsonb ) -> jsonb` `left ( string text, n integer ) -> text`	Returns the first `n` characters of a string.	2.17 and earlier

`length`	`length ( p graphpath ) -> jsonb`
	`length ( vs vertex[] ) -> jsonb`
	`length ( es edge[] ) -> jsonb`
	`length ( ps graphpath[] ) -> jsonb`
		Edges in a path, or elements of an array of graph elements.	2.17 and earlier

`length`	`length ( value jsonb ) -> jsonb`	Characters in a string, elements in a list, or keys in a map.	2.17 and earlier
`ln`	`ln ( x double precision ) -> double precision` `ln ( x numeric ) -> numeric`	Natural logarithm of `x`. The function `log` resolves to in Cypher.	2.17 and earlier
`localdatetime`	`localdatetime ( ) -> timestamp with time zone` `localdatetime ( timezone "any" ) -> timestamp without time zone`	Returns the current transaction timestamp, or the current wall-clock reading in a named zone.	2.17 and earlier
`log`	`log ( x jsonb ) -> jsonb`	Natural logarithm of `x` — not base 10, and with no base argument.	2.17 and earlier
`log10`	`log10 ( x double precision ) -> double precision` `log10 ( x numeric ) -> numeric`	Returns the base-10 logarithm of `x`.	2.17 and earlier
`lower`	`lower ( string text ) -> text`	Returns the string in lower case.	2.17 and earlier
`lTrim`	`lTrim ( string jsonb ) -> jsonb`	Removes whitespace from the start of a string.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability
`max`	`max ( expression jsonb ) -> jsonb` `max ( expression numeric ) -> numeric`	Returns the largest value in the group.	2.17 and earlier
`min`	`min ( expression jsonb ) -> jsonb` `min ( expression numeric ) -> numeric`	Returns the smallest value in the group.	2.17 and earlier
`nodes`	`nodes ( p graphpath ) -> vertex[]`	The vertices of a path, in order.	2.17 and earlier
`none`	`none ( variable IN list WHERE predicate ) -> boolean`	Returns true when the predicate holds for no element of the list — the negation of `any`.	2.17 and earlier
`percentileCont`	`percentileCont ( list jsonb, percentile numeric ) -> double precision`	Percentile of a list of numbers, interpolated between neighbouring values.	2.17 and earlier
`percentileDisc`	`percentileDisc ( list jsonb, percentile numeric ) -> double precision`	Percentile of a list of numbers, rounded down to a value in the list.	2.17 and earlier
`pi`	`pi ( ) -> double precision`	Returns π.	2.17 and earlier
`properties`	`properties ( v vertex ) -> jsonb` `properties ( e edge ) -> jsonb`	Returns a graph element's whole property bag as a map.	2.17 and earlier
`radians`	`radians ( x jsonb ) -> jsonb` `radians ( x double precision ) -> double precision`	Converts degrees to radians.	2.17 and earlier
`rand`	`rand ( ) -> jsonb`	Returns a number in the half-open range 0 to 1.	2.17 and earlier
`range`	`range ( start integer, stop integer ) -> jsonb` `range ( start integer, stop integer, step integer ) -> jsonb`	Returns a list of integers from `start` to `stop`, inclusive at both ends.	2.17 and earlier
`regather_graphmeta`	`regather_graphmeta ( ) -> boolean`	Empties `ag_graphmeta` and rebuilds it by scanning every edge label in the current graph.	2.17 and earlier
`relationships`	`relationships ( p graphpath ) -> edge[]`	Returns the edges of a path, in order.	2.17 and earlier
`replace`	`replace ( string jsonb, old jsonb, new jsonb ) -> jsonb`	Replaces every occurrence of one substring with another.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability

reverse	`reverse ( string text ) -> text`
	`reverse ( list jsonb ) -> jsonb`
	`reverse ( list anyarray ) -> anyarray`
		Reverses a list, or a string given as a literal or cast to `text`.	2.17 and earlier

right	`right ( string jsonb, n jsonb ) -> jsonb` `right ( string text, n integer ) -> text`	Returns the last `n` characters of a string.	2.17 and earlier
round	`round ( x jsonb ) -> jsonb` `round ( x numeric [, scale integer ] ) -> numeric`	Rounds `x` to the nearest integer, halves away from zero.	2.17 and earlier
rTrim	`rTrim ( string jsonb ) -> jsonb`	Removes whitespace from the end of a string.	2.17 and earlier
shortestpath	`shortestpath ( pattern ) -> graphpath`	Finds one shortest path between two vertices, by number of edges.	2.17 and earlier
sign	`sign ( x jsonb ) -> jsonb` `sign ( x numeric ) -> numeric`	Returns −1, 0 or 1 according to the sign of `x`.	2.17 and earlier
sin	`sin ( x double precision ) -> double precision`	Returns the sine of an angle given in radians.	2.17 and earlier
sind	`sind ( x double precision ) -> double precision`	Returns the sine of an angle given in degrees.	2.17 and earlier
single	`single ( variable IN list WHERE predicate ) -> boolean`	Returns true when the predicate holds for exactly one element of the list.	2.17 and earlier
size	`size ( value ) -> jsonb` `size ( pattern ) -> bigint`	The length of a value, or the number of times a pattern matches.	2.17 and earlier
split	`split ( string text, delimiter text ) -> anyarray`	Cuts a string into a list at every occurrence of a delimiter.	2.17 and earlier
sqrt	`sqrt ( x jsonb ) -> jsonb` `sqrt ( x numeric ) -> numeric`	Returns the square root of `x`.	2.17 and earlier
start_id	`start_id ( e edge ) -> graphid`	Returns the `graphid` of the vertex an edge leaves, without reading that vertex.	2.17 and earlier
start_vertex	`start_vertex ( e edge ) -> vertex`	Returns the vertex an edge leaves.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability
startNode	`startNode ( e edge ) -> vertex`	Returns the vertex an edge leaves.	2.17 and earlier
stdev	`stdev ( expression numeric ) -> numeric`	Sample standard deviation of the values in the group.	2.17 and earlier
stdevp	`stdevp ( expression numeric ) -> numeric`	Population standard deviation of the values in the group.	2.17 and earlier
str_size	`str_size ( string text ) -> integer`	Returns the number of characters in a string, as an `integer`.	2.17 and earlier
string_contains	`string_contains ( a jsonb, b jsonb ) -> boolean`	Implements `CONTAINS`. True when the right string occurs anywhere in the left.	2.17 and earlier
string_ends_with	`string_ends_with ( a jsonb, b jsonb ) -> boolean`	Implements `ENDS WITH`. True when the left string ends with the right.	2.17 and earlier
string_regex	`string_regex ( a jsonb, b jsonb ) -> boolean`	Implements `=~`. True when the left string matches the right as a regular expression.	2.17 and earlier
string_starts_with	`string_starts_with ( a jsonb, b jsonb ) -> boolean`	Implements `STARTS WITH`. True when the left string begins with the right.	2.17 and earlier
substring	`substring ( string jsonb, start jsonb ) -> jsonb` `substring ( string jsonb, start jsonb, count jsonb ) -> jsonb`	Returns part of a string, starting at `start` and running to the end or for `count` characters.	2.17 and earlier
sum	`sum ( expression numeric ) -> numeric`	Returns the total of the non-null values in the group.	2.17 and earlier
tail	`tail ( list jsonb ) -> jsonb` `tail ( list anyarray ) -> anyarray`	Returns every element of a list except the first.	2.17 and earlier
tan	`tan ( x jsonb ) -> jsonb` `tan ( x double precision ) -> double precision`	Returns the tangent of an angle given in radians.	2.17 and earlier
tand	`tand ( x double precision ) -> double precision`	Returns the tangent of an angle given in degrees.	2.17 and earlier
to_jsonb	`to_jsonb ( value anyelement ) -> jsonb`	Converts any SQL value to `jsonb`, a composite becoming an object.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability
`toBoolean`	`toBoolean ( value text ) -> boolean` `toBoolean ( value "any" ) -> boolean`	Returns the boolean a value names, or null when it names none.	2.17 and earlier
`toBooleanOrNull`	`toBooleanOrNull ( value text ) -> boolean` `toBooleanOrNull ( value "any" ) -> boolean`	Returns the boolean a value names, or null.	2.17 and earlier
`toFloat`	`toFloat ( value "any" ) -> double precision`	Returns the floating-point number a value names, or null.	Added in 2.18
`toFloatList`	`toFloatList ( list jsonb ) -> jsonb` `toFloatList ( list anyarray ) -> double precision[]`	Converts every element of a list to a floating-point number.	2.17 and earlier
`toInteger`	`toInteger ( value "any" ) -> bigint`	Returns the integer a value names, or null.	Added in 2.18
`toIntegerList`	`toIntegerList ( list jsonb ) -> jsonb` `toIntegerList ( list anyarray ) -> bigint[]`	Converts every element of a list to an integer.	2.17 and earlier
`toLower`	`toLower ( string jsonb ) -> jsonb`	Returns the string in lower case.	2.17 and earlier
`toString`	`toString ( value jsonb ) -> jsonb`	Returns the string form of a value.	2.17 and earlier
`toStringList`	`toStringList ( list jsonb ) -> jsonb` `toStringList ( list anyarray ) -> text[]`	Converts every element of a list to a string.	2.17 and earlier
`toStringOrNull`	`toStringOrNull ( value "any" ) -> text`	Returns the string form of a value, or null when the value has no string form.	2.17 and earlier
`toUpper`	`toUpper ( string jsonb ) -> jsonb`	Returns the string in upper case.	2.17 and earlier
`trim`	`trim ( string jsonb ) -> jsonb`	Removes whitespace from both ends of a string.	2.17 and earlier
`type`	`type ( e edge ) -> jsonb`	Returns the name of an edge's label, as a `jsonb` string — the same value as `label(e)`, under the name Cypher uses for relationship types.	2.17 and earlier

© 2026 SKAI Worldwide Co., Ltd.

Function	Signature	Description	Availability
`unnest`	`unnest ( from_json jsonb ) -> setof jsonb`	Expands a list into one row per element.	Added in 2.18
`upper`	`upper ( string text ) -> text`	Returns the string in upper case.	2.17 and earlier
`vertex_to_jsonb`	`vertex_to_jsonb ( v vertex ) -> jsonb`	Returns a vertex's property bag.	2.17 and earlier
`vertices`	`vertices ( p graphpath ) -> vertex[]`	Returns the vertices of a path, in order.	2.17 and earlier

`collect` is older than 2.18, but its native-array results arrived with this release: `vertex[]`, `edge[]` and `graphpath[]` in place of `jsonb`. So did the three-valued implementations behind `all`, `any`, `none` and `single`. The **2.18 release notes** carry the full list of changes.

Functions documented elsewhere

The `meta` schema adds functions for inspecting graphs, labels and their sizes from SQL: `meta.labels()`, `meta.count()`, `meta.label_size()` and the rest. It is an extension rather than part of the core catalog, so a database has it only once it is installed. Those functions are documented on their own page, **Graph metadata**, and are not repeated here.

Every function PostgreSQL itself provides is also reachable from a Cypher query, subject to the type rules described on each page here. The **PostgreSQL documentation** is the reference for those.

What this reference leaves out

The functions AgensGraph adds occupy the object-identifier range 7000 to 8100. Not all of them are meant to be called. The range also holds type input and output functions, comparison and hash support for the B-tree and GIN operator classes, and aggregate transition and final functions. It holds a planner support function, two `pg_upgrade` helpers, and four placeholder entries the path-finding executor resolves. This reference documents the ones a query can use, and omits those.

List what your own build exposes

The catalog is the authority on what a given server has. This lists every AgensGraph function in that range, aggregates included:

```
SELECT p.proname AS function,
       pg_catalog.pg_get_function_identity_arguments(p.oid) AS arguments,
       pg_catalog.format_type(p.prorettype, NULL) AS returns,
       CASE p.prokind WHEN 'a' THEN 'aggregate' ELSE 'function' END AS kind,
       obj_description(p.oid, 'pg_proc') AS description
FROM pg_catalog.pg_proc p
WHERE p.oid BETWEEN 7000 AND 8100
ORDER BY p.proname, p.oid;
```

On this 2.18 server that returns 195 rows. That is more than this reference documents, because of the support entries listed above. To count them instead:

```
SELECT count(*) FROM pg_catalog.pg_proc WHERE oid BETWEEN 7000 AND 8100;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 count
-------
   195
```

A function whose name AgensGraph does not add will not appear, because it belongs to PostgreSQL. `avg`, `min`, `left` and `log10` are among those. Drop the `WHERE` clause and filter by name to find one of them.

See also

- Aggregate functions — `avg`, `collect`, `count`, `min`, `max`, `sum`
- Predicate functions — `all`, `any`, `none`, `single`, `exists`, `isEmpty`
- Scalar functions — `id`, `label`, `properties`, `size`, `length`
- List functions — `head`, `keys`, `range`, `tail`, `unnest`
- String functions — `substring`, `split`, `toLower`, `trim`
- Numeric functions — `abs`, `round`, `log`, the trigonometric family
- Path functions — `nodes`, `relationships`, `shortestpath`, `dijkstra`
- Type conversion functions — the `to…` family
- jsonb operators and functions — the operators underneath every property
- Graph metadata — the `meta.*` schema

Aggregate functions

Functions that reduce a group of rows to one value. Cypher forms a group the same way SQL does: every non-aggregated expression in the `RETURN` or `WITH` list is a grouping key. With no key at all, the whole result is one group.

Examples run against **the example dataset**.

avg

```
avg ( expression numeric ) -> numeric
```

Returns the arithmetic mean of the non-null values in the group.

Considerations

- A property arrives as `jsonb` and is coerced to `numeric`, so the result is `numeric` and keeps full precision rather than rounding to `double precision`.
- Rows whose value is null are skipped. A group with no non-null value returns null.

```
MATCH (m:movie) RETURN avg(m.year) AS mean_year;
```

```
      mean_year
----------------------
 2002.0555555555555556
```


© 2026 SKAI Worldwide Co., Ltd.

collect

```
collect ( expression anyelement ) -> jsonb
collect ( v vertex ) -> vertex[]
collect ( e edge ) -> edge[]
collect ( p graphpath ) -> graphpath[]
```

Gathers the values in the group into a list, in the group's row order.

Considerations

- A `vertex`, `edge` or `graphpath` argument produces a native array rather than a `jsonb` array: `vertex[]`, `edge[]` or `graphpath[]`. Everything else produces `jsonb`. Added in 2.18
- A native array is indexed from 0 by Cypher's subscript, like a `jsonb` list.
- Null values are collected. `collect(DISTINCT expr)` removes duplicates.
- `length()` counts the elements of any of the four result types.

```
MATCH (p:person)-[:directed]->(m:movie)
WITH p, collect(m.title) AS films
RETURN p.name AS director, films ORDER BY p.name LIMIT 2;
```

```
     director       |             films
--------------------+-------------------------------
 "Andrew Niccol"    | ["Gattaca"]
 "Christopher Nolan" | ["Inception", "Interstellar"]
```

Collecting the graph elements themselves gives an array of them:

```
MATCH (:movie {title:'The Matrix'})-[e:related_to]->()
WITH collect(e) AS edges
RETURN length(edges) AS n, pg_typeof(edges) AS type;
```

```
 n |   type
---+----------
 3 | "edge[]"
```

count

```
count ( * ) -> bigint
count ( expression ) -> bigint
```

Counts rows in the group. `count(*)` counts every row; `count(expression)` counts the rows whose expression is not null.

Considerations

- `count(v)` over a whole vertex or edge counts the element's identity: the parser replaces the argument with its `graphid`, so the property bag is never read.
- `count(DISTINCT expression)` counts distinct non-null values.
- A group with no rows returns 0, not null.

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie)
RETURN count(*) AS films, count(m.genre) AS with_genre, count(DISTINCT m.genre) AS genres;
```

```
 films | with_genre | genres
-------+------------+--------
 18    | 18         | 3
```

max

```
max ( expression jsonb ) -> jsonb
max ( expression numeric ) -> numeric
```

Returns the largest value in the group.

Considerations

- A `jsonb` argument is ordered by `jsonb`'s own type ordering, in which every number sorts below every string. A numeric or text property read from a typed column is compared in that column's type.
- Nulls are skipped; a group with no non-null value returns null.

```
MATCH (m:movie) RETURN min(m.year) AS earliest, max(m.year) AS latest;
```

```
 earliest | latest
----------+--------
 1982     | 2021
```

min

```
min ( expression jsonb ) -> jsonb
min ( expression numeric ) -> numeric
```

Returns the smallest value in the group. The considerations for **`max`** apply unchanged.

percentileCont

```
percentileCont ( list jsonb, percentile numeric ) -> double precision
```

Returns the percentile of a list of numbers, interpolating between the two values that straddle the requested position.

Considerations

- This is a plain function over a list, not an aggregate. To take a percentile of a group, build the list first: `percentileCont(collect(m.year), 0.5)`.
- The argument must be a list. A scalar is refused, and the message names the wrong function:

```
RETURN percentileCont(2010, 0.5);
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  reverse(): list is expected but 2010
```

- Returns null when the list holds fewer than two elements, or when the percentile is outside the range 0 to 1.
- Non-numeric elements are ignored rather than rejected.

```
MATCH (m:movie) RETURN percentileCont(collect(m.year), 0.5) AS median;
```

```
 median
--------
 2002.5
```

percentileDisc

```
percentileDisc ( list jsonb, percentile numeric ) -> double precision
```

Returns the percentile of a list of numbers, rounded down to an element that is actually in the list. Its considerations are those of `percentileCont`.

```
MATCH (m:movie) RETURN percentileDisc(collect(m.year), 0.5) AS median;
```

```
 median
--------
 2002
```

stdev

```
stdev ( expression numeric ) -> numeric
```

Returns the sample standard deviation of the values in the group: the estimate that divides by n − 1.

Considerations

- The parser rewrites the name to `stddev_samp`, so that is the name the result column carries and the name an error message uses.
- Unquoted identifiers fold to lower case, so `stDev` and `STDEV` are the same call.
- `stdDev` is a different function: it folds to `stddev`, PostgreSQL's own alias for `stddev_samp`. It computes the same number.
- A group of one row returns null.

```
MATCH (m:movie) RETURN stdev(m.year) AS sample_sd, stdevp(m.year) AS population_sd;
```

```
      sample_sd      |    population_sd
---------------------+---------------------
 11.3628508344062724 | 11.0427061006209596
```


© 2026 SKAI Worldwide Co., Ltd.

stdevp

```
stdevp ( expression numeric ) -> numeric
```

Returns the population standard deviation of the values in the group: the estimate that divides by n. The parser rewrites the name to `stddev_pop`, and `stDevP` is the same call.

sum

```
sum ( expression numeric ) -> numeric
```

Returns the total of the non-null values in the group.

Considerations

- Nulls are skipped. A group with no non-null value returns null, not 0.
- Nothing deduplicates the input: a value that a pattern matches twice is added twice. Use `sum(DISTINCT expr)` if that is not what you want.

```
MATCH (p:person)-[:acted_in]->(:movie) RETURN sum(p.born) AS born_total;
```

```
 born_total
------------
 49017
```

Compatibility

openCypher

`stDev` and `stDevP` are openCypher's names for the sample and population standard deviation; both spellings work here. openCypher's `percentileCont` and `percentileDisc` are aggregates that take a value per row. These take a finished list, so a query written for openCypher needs `collect()` around the first argument.

ISO/IEC 39075

`avg`, `count`, `max`, `min`, `stddev_pop`, `stddev_samp` and `sum` are the GQL aggregate functions and are spelled as the standard spells them. `collect` corresponds to the GQL `COLLECT` value expression, which is documented with the **subquery expressions**.

See also

- **All functions** — every function in one table
- **Predicate functions** — `all`, `any`, `none`, `single`
- **List functions** — working with the list `collect` produces

Predicate functions

Functions that answer yes or no. The four list quantifiers, `all`, `any`, `none` and `single`, are grammar forms rather than ordinary calls. Each takes a variable, a list and a `WHERE` clause. Each reduces the per-element results with three-valued logic, so a list holding a null can make the answer null rather than false.

Examples run against **the example dataset**.

© 2026 SKAI Worldwide Co., Ltd.

all

```
all ( variable IN list WHERE predicate ) -> boolean
```

Returns true when the predicate holds for every element of the list.

Considerations

- An empty list returns true.
- If the predicate is null for some element and true for all the rest, the result is null. One false element makes the result false whatever else is null.
- The predicate is evaluated for every element; nothing short-circuits.
- The call is lowered to `cypher_all()`. An error message from inside the predicate carries that name.

```
MATCH (m:movie) WHERE all(x IN m.embedding WHERE x > -0.1) RETURN m.title ORDER BY m.title;
```

```
        title
---------------------
 "Blade Runner 2049"
```

any

```
any ( variable IN list WHERE predicate ) -> boolean
```

Returns true when the predicate holds for at least one element of the list.

Considerations

- An empty list returns false.
- One true element makes the result true even when other elements are null.
- If no element is true and any is null, the result is null.

```
MATCH (p:person)-[:acted_in]->(m:movie)
WITH p, collect(m.genre) AS genres
WHERE any(g IN genres WHERE g = 'Adventure')
RETURN p.name AS actor;
```

```
    actor
-------------
 "Sam Neill"
```

exists

```
exists ( pattern ) -> boolean
exists ( property_reference ) -> boolean
```

With a pattern, returns true when the pattern matches at least once. With a property reference, returns true when that property is present and not null.

© 2026 SKAI Worldwide Co., Ltd.

Considerations

- The two forms are different constructs, not overloads of one function. The pattern form becomes an `EXISTS` subquery, and the property form becomes an `IS NOT NULL` test.
- The argument of the property form has to be a property reference such as `m.year`. An arbitrary expression is read as a pattern and fails to parse.
- A pattern inside `exists()` may bind variables from the enclosing query, as `(p)` does below, but any variable it introduces is not visible outside.

```
MATCH (m:movie {title:'Inception'}) RETURN exists(m.year) AS has_year, exists(m.budget) AS
has_budget;
```

```
 has_year | has_budget
----------+------------
 t        | f
```

```
MATCH (p:person) WHERE exists((p)-[:directed]->()) RETURN p.name ORDER BY p.name LIMIT 3;
```

```
        name
--------------------
 "Andrew Niccol"
 "Christopher Nolan"
 "Denis Villeneuve"
```

isEmpty

```
isEmpty ( value jsonb ) -> boolean
```

Returns true when a list has no elements or a map has no keys.

Considerations

- The argument must resolve to `jsonb`. A bare string literal is `unknown` and matches the range and multirange `isempty` as well:

```
RETURN isEmpty('');
```

```
ERROR:  function isempty(unknown) is not unique
LINE 1: RETURN isEmpty('');
               ^
HINT:  Could not choose a best candidate function. You might need to add explicit type casts.
```

Cast it: `isEmpty(''::jsonb)`.

- A property reference is already `jsonb`, so `isEmpty(n.tags)` needs no cast.
- Returns true for an empty string, which is the empty `jsonb` scalar.

© 2026 SKAI Worldwide Co., Ltd.

```
RETURN isEmpty([]) AS empty_list, isEmpty([1]) AS one_element, isEmpty({}) AS empty_map;
```

```
 empty_list | one_element | empty_map
------------+-------------+-----------
 t          | f           | t
```

none

```
none ( variable IN list WHERE predicate ) -> boolean
```

Returns true when the predicate holds for no element of the list. It is the negation of `any`.

Considerations

- An empty list returns true.
- One true element makes the result false. If no element is true and any is null, the result is null.

```
MATCH (m:movie) WHERE none(x IN m.embedding WHERE x > 0.4) RETURN count(*) AS films;
```

```
 films
-------
 15
```

single

```
single ( variable IN list WHERE predicate ) -> boolean
```

Returns true when the predicate holds for exactly one element of the list.

Considerations

- An empty list returns false.
- A null result for any element makes the whole answer null, because a null could have been the second match.

```
MATCH (p:person)-[:directed]->(m:movie)
WITH p, collect(m.year) AS years
WHERE single(y IN years WHERE y > 2010)
RETURN p.name AS director, years ORDER BY p.name;
```

```
      director       |          years
---------------------+--------------------------
 "Christopher Nolan" | [2010, 2014]
 "Denis Villeneuve"  | [2017]
 "Joss Whedon"       | [2012]
 "Lana Wachowski"    | [2003, 1999, 2021, 2003]
 "Neill Blomkamp"    | [2013]
```

Three-valued results side by side

The quantifiers differ most where the list holds a null and where the list is empty. Both rows below are real output:

© 2026 SKAI Worldwide Co., Ltd.

```
RETURN all(x IN [1, null] WHERE x > 0) AS every, any(x IN [1, null] WHERE x > 0) AS some,
       none(x IN [null] WHERE x > 0) AS not_any, single(x IN [1, null] WHERE x > 0) AS exactly
_one;
```

```
 every | some | not_any | exactly_one
-------+------+---------+-------------
       | t    |         |
```

```
RETURN all(x IN [] WHERE x = 0) AS every, any(x IN [] WHERE x = 0) AS some,
       none(x IN [] WHERE x = 0) AS not_any, single(x IN [] WHERE x = 0) AS exactly_one;
```

```
 every | some | not_any | exactly_one
-------+------+---------+-------------
 t     | f    | t       | f
```

A `RETURN` leaves a boolean column as `boolean` rather than boxing it to `jsonb`. So these print as `t`, `f` and blank for null, not as `true`, `false` and `null`.

Compatibility

openCypher

`exists()` over a property is openCypher's older spelling of `IS NOT NULL`. The pattern-comprehension and `exists { ... }` block forms are not supported; use the pattern form above or a **subquery expression**.

See also

- **All functions** — every function in one table
- **Subquery expressions** — `EXISTS`, `COUNT`, `COLLECT`
- **List functions** — building the lists these quantify over

Scalar functions

Functions that take a value and return a single value. They cover the parts of a graph element, the pieces of a `graphid`, and the length of a string or list. The server-information functions here report what a build is and what the last write did.

Examples run against **the example dataset**.

agversion

```
agversion ( ) -> text
```

Returns the AgensGraph version of the running server, without the PostgreSQL part that `version()` also carries.

Considerations

- `agversion` is also a read-only setting, so `SHOW agversion` returns the same string. The function is the form you can use inside an expression.
- The value is the fork's own version, not PostgreSQL's. Read `version()` for both.

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT agversion();
```

```
 agversion
------------
 2.18.6.1
```

ag_get_graphconstraintdef

```
ag_get_graphconstraintdef ( constraint oid ) -> text
```

Reconstructs the `CREATE CONSTRAINT` command for a graph constraint, given its OID. This is the graph counterpart of `pg_get_constraintdef` and is what `pg_dump` uses.

ag_get_propindexdef

```
ag_get_propindexdef ( index oid ) -> text
```

Reconstructs the `CREATE PROPERTY INDEX` command for a property index, given its OID.

Considerations

- A property index is an expression index over the `properties` column, so `pg_get_indexdef` prints it as SQL. This function prints the Cypher form instead.

coalesce

```
coalesce ( value [, value ] ... ) -> jsonb
```

Returns the first argument that is not null.

Considerations

- This is an expression form rather than a function, so it will not appear in a catalog listing of functions.
- A property reference is always `jsonb`. If any argument is `jsonb`, every argument is coerced to `jsonb` and the result is `jsonb`. Otherwise the arguments settle on a common type and keep it, so `coalesce(v1, v2)` over two vertices returns a `vertex`.
- Mixing a graph element with anything else therefore fails, because the element cannot be boxed. Pass `m::jsonb` instead.
- Arguments after the first non-null one are not evaluated.

```
MATCH (m:movie {title:'Inception'}) RETURN coalesce(m, 0);
```

```
ERROR:  COALESCE types vertex and integer cannot be matched
LINE 1: MATCH (m:movie {title:'Inception'}) RETURN coalesce(m, 0);
                                                            ^
```

```
MATCH (m:movie {title:'Inception'}) RETURN coalesce(m.budget, m.year, 0) AS first_present;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 first_present
---------------
 2010
```

datetime

```
datetime ( ) -> timestamp with time zone
datetime ( timezone "any" ) -> timestamp without time zone
datetime ( year integer, month integer, mday integer, hour integer, min integer, sec integer,
millisec integer, nanosec integer, timezone "any" ) -> timestamp without time zone
```

Returns the current transaction timestamp, the current time in a named zone, or a timestamp built from its parts.

Name	Type	Description
`year` , `month` , `mday`	`integer`	Calendar date.
`hour` , `min` , `sec`	`integer`	Time of day.
`millisec` , `nanosec`	`integer`	Sub-second parts. Nanoseconds beyond microsecond resolution are truncated.
`timezone`	`"any"`	Zone name or abbreviation, for example `'Asia/Seoul'` .

Considerations

- The no-argument form is stable, so every row of one query sees the same instant, and it changes between transactions.
- The zone forms return `timestamp without time zone` : the wall-clock reading in that zone, with the zone itself discarded.
- A `RETURN` boxes the result to a `jsonb` string in ISO 8601 form.

```
SELECT datetime('Asia/Seoul'), datetime(2020, 1, 2, 3, 4, 5, 0, 0, 'UTC');
```

```
          datetime          |      datetime
----------------------------+---------------------
 2026-09-02 16:52:34.625259 | 2020-01-02 03:04:05
```

endNode

```
endNode ( e edge ) -> vertex
```

Returns the vertex an edge points at. `end_vertex` is the same function under a different name; see **`end_vertex`** on the path page.

© 2026 SKAI Worldwide Co., Ltd.

end_id

```
end_id ( e edge ) -> graphid
```

Returns the `graphid` of the vertex an edge points at, without reading that vertex.

Considerations

- An edge table stores `start` and `end` as columns, so this costs nothing beyond the edge row itself, while `endNode()` has to fetch the vertex.
- Use it whenever you only need to compare or group identities.

```
MATCH ()-[e:acted_in]->() WITH e ORDER BY id(e) LIMIT 1
RETURN start_id(e) AS from_id, end_id(e) AS to_id;
```

```
 from_id | to_id
---------+-------
 4.1     | 3.1
```

get_last_graph_write_stats

```
get_last_graph_write_stats ( ) -> record
```

Reports what the most recent graph write in this session changed, as one row of five counters: `insertedvertices`, `insertededges`, `deletedvertices`, `deletededges` and `updatedproperties`.

Considerations

- The counters describe the last graph-writing statement only, not the session total.
- A statement that writes nothing leaves the previous statement's counters in place rather than zeroing them, so read the row immediately after the write you care about.
- Call it from SQL: it returns a record, and Cypher has no way to expand one.

```
SELECT * FROM get_last_graph_write_stats();
```

```
 insertedvertices | insertededges | deletedvertices | deletededges | updatedproperties
------------------+---------------+-----------------+--------------+-------------------
                1 |             0 |               0 |            0 |                 0
```

get_time

```
get_time ( ) -> time without time zone
get_time ( timezone "any" ) -> time without time zone
```

Returns the current time of day, either locally or in a named zone.

© 2026 SKAI Worldwide Co., Ltd.

Considerations

- Both forms are marked immutable, so the planner folds the call to a constant. Every row of a query sees the same time, and so does every execution of a prepared statement. Read the time from `localdatetime()`, which is stable, or from SQL's `localtime`, when you need it to advance.

```
SELECT get_time(), get_time('Asia/Seoul');
```

```
    get_time    |    get_time
----------------+-----------------
 12:52:34.62488 | 16:52:34.625259
```

graphid

```
graphid ( labid integer, locid bigint ) -> graphid
graphid ( id numeric ) -> graphid
```

Builds a `graphid` from a label id and a local id, or from the `labid.locid` number that a `graphid` prints as.

Considerations

- This is how a label's `id` column default is written; see **how a graph maps onto PostgreSQL**.
- The numeric form is a convenience for hand-written SQL. It does not check that the label exists.

```
SELECT graphid(4, 1), graphid(4.1::numeric);
```

```
 graphid | graphid
---------+---------
 4.1     | 4.1
```

graph_labid

```
graph_labid ( qualified_label cstring ) -> integer
```

Returns the label id of a label named as `graph.label`.

```
SELECT graph_labid('moviekb.person');
```

```
 graph_labid
-------------
           4
```

graphid_labid

```
graphid_labid ( id graphid ) -> integer
```

Returns the label part of a `graphid`: the number before the dot.

© 2026 SKAI Worldwide Co., Ltd.

Considerations

- A `graphid` packs the label id into its top 16 bits and the local id into the rest. A whole label therefore occupies one contiguous range of `graphid` values, and a label test can be answered by an index range scan without touching a catalog.

graphid_locid

```
graphid_locid ( id graphid ) -> bigint
```

Returns the local part of a `graphid`: the number after the dot, taken from the label's sequence.

```
MATCH (n:person {name:'Keanu Reeves'})
RETURN id(n) AS id, graphid_labid(id(n)) AS label_id, graphid_locid(id(n)) AS local_id;
```

```
 id  | label_id | local_id
-----+----------+----------
 4.1 | 4        | 1
```

id

```
id ( v vertex ) -> graphid
id ( e edge ) -> graphid
```

Returns a graph element's identity.

Considerations

- This is not a catalog function. A `vertex` is the composite type `(id, properties)` and an `edge` is `(id, start, end, properties)`. So `id(v)` is PostgreSQL's functional spelling of field selection (`parse_func.c:1961`). The same mechanism gives you `properties(v)`.
- `n.id` is **not** the same thing. It reads a property named `id`, which the example dataset does not have, so it returns null. Write `id(n)`.
- A `RETURN` leaves the result as `graphid`, not boxed to `jsonb`. Aggregating it does box it: `min(id(n))` comes back as the `jsonb` string `"4.1"`.

```
MATCH (n:person {name:'Keanu Reeves'}) RETURN id(n) AS identity, n.id AS property_named_id;
```

```
 identity | property_named_id
----------+-------------------
 4.1      |
```

label

```
label ( v vertex ) -> jsonb
label ( e edge ) -> jsonb
```

Returns the name of the label a graph element was created with, as a `jsonb` string.

© 2026 SKAI Worldwide Co., Ltd.

Considerations

- `label` and **`labels`** are different functions, not spellings of one. `label` returns one string, the element's own label. `labels` returns a list of that label followed by the labels it inherits from.
- `label` works on edges as well as vertices; `labels` does not. For an edge, `label(e)` and **`type(e)`** are the same call.

```
MATCH (n:person {name:'Keanu Reeves'}) RETURN label(n) AS own, labels(n) AS own_and_inherited;
```

```
   own    | own_and_inherited
----------+-------------------
 "person" | ["person"]
```

The two answers diverge only where a label inherits. Add a label with a parent:

```
CREATE VLABEL creature;
CREATE VLABEL dragon INHERITS (creature);
CREATE (:dragon {name: 'Smaug'});
```

```
MATCH (d:dragon) RETURN label(d) AS own, labels(d) AS own_and_inherited;
```

```
   own    |   own_and_inherited
----------+------------------------
 "dragon" | ["dragon", "creature"]
```

Then take it back out:

```
MATCH (d:dragon) DETACH DELETE d;
DROP VLABEL dragon;
DROP VLABEL creature;
```

An edge has no `labels()`:

```
MATCH ()-[e:acted_in]->() WITH e LIMIT 1 RETURN labels(e);
```

```
ERROR:  function labels(edge) does not exist
LINE 1: MATCH ()-[e:acted_in]->() WITH e LIMIT 1 RETURN labels(e);
                                                        ^
HINT:  No function matches the given name and argument types. You might need to add explicit t
ype casts.
```

labels

```
labels ( v vertex ) -> jsonb
```

Returns a list holding the vertex's own label and every label it inherits from, nearest first.

© 2026 SKAI Worldwide Co., Ltd.

Considerations

- Vertices only. There is no `labels(edge)`.
- The list has one element for a label with no parent, which is the common case.

length

```
length ( value jsonb ) -> jsonb
```

Returns the number of characters in a string, elements in a list, or keys in a map.

Considerations

- **`size`** is the Cypher spelling of this function; the parser rewrites it, so the result column is named `length`.
- Over a path or an array of graph elements, `length` is a different function with a different meaning: the number of edges in the path. See **`length`** on the path page.
- A number returns the length of its text form rather than raising.

```
RETURN length('The Matrix') AS characters, length([1,2,3]) AS elements, length({a:1,b:2}) AS
keys;
```

```
 characters | elements | keys
------------+----------+------
 10         | 3        | 2
```

localdatetime

```
localdatetime ( ) -> timestamp with time zone
localdatetime ( timezone "any" ) -> timestamp without time zone
```

Returns the current transaction timestamp, or the current wall-clock reading in a named zone.

Considerations

- The no-argument form returns the same value as `datetime()`, including the same type. The two names exist for GQL compatibility; neither drops the zone.
- The zone form returns `timestamp without time zone`, as `datetime(zone)` does.
- Both forms are stable, so a value taken from them does advance between transactions, unlike **`get_time`**.

properties

```
properties ( v vertex ) -> jsonb
properties ( e edge ) -> jsonb
```

Returns a graph element's whole property bag as a map.

Considerations

- Like **`id`**, this is field selection on the composite type rather than a catalog function, so it costs no more than reading the column.
- `vertex_to_jsonb` and `edge_to_jsonb` return the same map, and are the functions the cast `v::jsonb` uses.

© 2026 SKAI Worldwide Co., Ltd.

- Reading the bag detoasts it in full. If you only want one key, read that key.

```
MATCH (n:person {name:'Keanu Reeves'}) RETURN properties(n) AS bag;
```

```
                  bag
----------------------------------------
 {"born": 1964, "name": "Keanu Reeves"}
```

regather_graphmeta

```
regather_graphmeta ( ) -> boolean
```

Empties `ag_graphmeta` and rebuilds it by scanning every edge label in the current graph. `ag_graphmeta` records which label triples occur and how many edges make up each, and the planner uses it to skip label tables a pattern cannot match.

Considerations

- The scan reads every edge in the graph, so the cost grows with the graph.
- `auto_gather_graphmeta` keeps the table current as writes happen. Rebuild by hand after a bulk load done with that setting off.
- Emits `NOTICE: regathering graphmeta...` and returns true.

```
SELECT regather_graphmeta();
```

```
NOTICE:  regathering graphmeta...
 regather_graphmeta
--------------------
 t
```

size

```
size ( value ) -> jsonb
size ( pattern ) -> bigint
```

With a value, returns its length. With a pattern, returns the number of times the pattern matches.

Considerations

- The two forms are told apart by the opening character. A pattern always starts with `(`, so anything that cannot is read as a value. The value form becomes **`length`**, so the result column is named `length`. The pattern form becomes a counting subquery, so it is named `count`.
- A `shortestpath` or `allshortestpaths` argument keeps its established meaning, the number of paths found, rather than being read as a pattern.

```
MATCH (a:person {name:'Keanu Reeves'}) RETURN size((a)-[]->()) AS edges, size(a.name) AS
name_length;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 edges | name_length
-------+-------------
 4     | 12
```

startNode

```
startNode ( e edge ) -> vertex
```

Returns the vertex an edge leaves. `start_vertex` is the same function under a different name; see **`start_vertex`** on the path page.

start_id

```
start_id ( e edge ) -> graphid
```

Returns the `graphid` of the vertex an edge leaves, without reading that vertex. The considerations for **`end_id`** apply.

type

```
type ( e edge ) -> jsonb
```

Returns the name of an edge's label, as a `jsonb` string. It is the same value as `label(e)`, under the name Cypher uses for relationship types.

Considerations

- Edges only. Over a vertex it is refused; use **`label`**.

```
MATCH (m:movie {title:'Inception'}) RETURN type(m);
```

```
ERROR:  function type(vertex) does not exist
LINE 1: MATCH (m:movie {title:'Inception'}) RETURN type(m);
                                                   ^
HINT:  No function matches the given name and argument types. You might need to add explicit t
ype casts.
```

```
MATCH ()-[e:acted_in]->() WITH e LIMIT 1 RETURN type(e) AS relationship_type;
```

```
 relationship_type
-------------------
 "acted_in"
```

Compatibility

openCypher

`id()` returns a `graphid`, a label id and a sequence number, not the integer openCypher returns. It is stable for the life of the element and unique across the graph. But it is not a small dense integer, and arithmetic on it is meaningless.

© 2026 SKAI Worldwide Co., Ltd.

openCypher's `labels()` returns every label a node carries, because a node there may carry several. A vertex here has exactly one label, and `labels()` returns that label plus the labels it inherits from.

See also

- **All functions** — every function in one table
- **Path functions** — `nodes`, `relationships`, `length` over a path
- **jsonb operators and functions** — reading the map `properties()` returns
- **Graph metadata functions** — the `meta.*` schema

List functions

Functions over lists and maps. A Cypher list is a `jsonb` array, so these functions take and return `jsonb`. `collect()` over a graph element instead produces a native array, and the functions that have an array overload work on both.

Cypher subscripts a list from 0, and a slice `list[a..b]` excludes the upper bound.

Examples run against **the example dataset**.

array_size

```
array_size ( list jsonb ) -> integer
array_size ( list json ) -> integer
array_size ( list anyarray ) -> integer
```

Returns the number of elements in an array.

Considerations

- Arrays only. Unlike **`cardinality`** it does not count a map's keys.
- The three overloads let the same name work on a Cypher list, a `json` value and a PostgreSQL array.

```
RETURN array_size([1,2,3]) AS n;
```

```
 n
---
 3
```

cardinality

Added in 2.18

```
cardinality ( value jsonb ) -> jsonb
cardinality ( p graphpath ) -> jsonb
```

Returns the number of elements in a list, entries in a map, or vertices and edges in a path.

Considerations

- A string is refused. Use **`length`** for a string.

© 2026 SKAI Worldwide Co., Ltd.

```
RETURN cardinality('abc');
```

```
ERROR:  cardinality(): list or map is expected but "abc"
```

- Over a path this counts vertices and edges, so a one-edge path gives 3, while `length` over the same path gives 1.
- PostgreSQL's own `cardinality(anyarray)` still applies to a native array and returns `integer` rather than `jsonb`.

```
RETURN cardinality([1,2,3]) AS list, cardinality({a:1,b:2}) AS map;
```

```
 list | map
------+-----
 3    | 2
```

```
MATCH p=(:person {name:'Christopher Nolan'})-[:directed]->(:movie {title:'Inception'})
RETURN cardinality(p) AS elements;
```

```
 elements
----------
 3
```

head

```
head ( list jsonb ) -> jsonb
head ( list anyarray ) -> anyelement
```

Returns the first element of a list.

Considerations

- An empty list returns null rather than raising.
- The array overload returns the element's own type, so `head(collect(n))` over vertices returns a `vertex`.

```
RETURN head([10,20,30]) AS first, last([10,20,30]) AS final, tail([10,20,30]) AS rest;
```

```
 first | final |   rest
-------+-------+----------
 10    | 30    | [20, 30]
```

keys

```
keys ( map jsonb ) -> jsonb
```

Returns a list of the key names of a map, or of a graph element's properties.

© 2026 SKAI Worldwide Co., Ltd.

Considerations

- The order is `jsonb`'s own key order: by key length, then bytewise. It is not insertion order and not alphabetical. Sort the result if you need a fixed order.
- A vertex or edge argument is accepted because it coerces to its property map.

```
MATCH (p:person {name:'Keanu Reeves'}) RETURN keys(p) AS property_names;
```

```
  property_names
------------------
 ["born", "name"]
```

```
RETURN keys({title:'Inception', year:2010}) AS map_keys;
```

```
     map_keys
--------------------
 ["year", "title"]
```

last

```
last ( list jsonb ) -> jsonb
last ( list anyarray ) -> anyelement
```

Returns the final element of a list. An empty list returns null. See **`head`** for a worked example.

range

```
range ( start integer, stop integer ) -> jsonb
range ( start integer, stop integer, step integer ) -> jsonb
```

Returns a list of integers from `start` to `stop`, inclusive at both ends.

Name	Type	Description
`start`	`integer`	First value in the list.
`stop`	`integer`	Last value the list may reach. Included when the step lands on it.
`step`	`integer`	Increment. Omitted, it is 1 counting up and −1 counting down.

Considerations

- The result is `jsonb`, not a PostgreSQL array, so it can be subscripted and iterated like any Cypher list.
- Both ends are inclusive. `stop` appears in the list only when the step lands on it: `range(1,10,3)` ends at 10, while `range(1,9,3)` ends at 7.
- `stop` below `start` counts down without needing a negative step.

© 2026 SKAI Worldwide Co., Ltd.

```
RETURN range(1,5) AS up, range(1,10,3) AS step, range(5,1) AS down, range(5,1,-2) AS
down_step;
```

```
      up        |     step      |      down       | down_step
----------------+---------------+-----------------+-----------
 [1, 2, 3, 4, 5] | [1, 4, 7, 10] | [5, 4, 3, 2, 1] | [5, 3, 1]
```

tail

```
tail ( list jsonb ) -> jsonb
tail ( list anyarray ) -> anyarray
```

Returns every element of a list except the first.

Considerations

- An empty list returns an empty list, not null, unlike `head` and `last`.

```
RETURN head([]) AS first, tail([]) AS rest;
```

```
 first | rest
-------+------
       | []
```

unnest

Added in 2.18

```
unnest ( from_json jsonb ) -> setof jsonb
```

Expands a list into one row per element.

Considerations

- This is a set-returning function. In Cypher it multiplies the rows of the clause it sits in. `FOR x IN list` is the clause form of the same idea, and it is easier to read when you need the element bound to a name.
- The output column is named `value`, so in SQL you write `SELECT value FROM unnest(...)`.
- PostgreSQL's `unnest(anyarray)` still applies to a native array.

```
MATCH (m:movie {title:'The Matrix'}) RETURN unnest(m.embedding) AS component;
```

```
  component
-------------
 -0.07594558
 0.04081754
 0.29592122
 -0.11921061
```


© 2026 SKAI Worldwide Co., Ltd.

Reversing and splitting

Two functions that produce lists are documented with the strings they come from: `reverse`, which reverses a list as well as a string, and `split`, which cuts a string into a list. The type-conversion page has the three list-typed conversions: `toStringList`, `toIntegerList` and `toFloatList`.

Compatibility

openCypher

`range()` is inclusive of `stop` here, as in openCypher. openCypher's `reduce()` is not supported; use a list comprehension or `FOR`. Pattern comprehensions are not supported.

See also

- **All functions** — every function in one table
- **Aggregate functions** — `collect`, which builds the lists
- **Predicate functions** — `all`, `any`, `none`, `single`, `isEmpty`
- **jsonb operators and functions** — the operators underneath a list

String functions

Functions over text. A property arrives as `jsonb`, and several of these functions have both a `text` and a `jsonb` overload. Which one the parser picks depends on what you pass, so a literal and a property do not always behave alike. Each entry says so where it matters.

Examples run against **the example dataset**.

left

```
left ( string jsonb, n jsonb ) -> jsonb
left ( string text, n integer ) -> text
```

Returns the first `n` characters of a string.

Considerations

- A property with an integer literal is **ambiguous**, because `left(jsonb, jsonb)` and `left(text, integer)` match equally well:

```
MATCH (m:movie {title:'Blade Runner 2049'}) RETURN left(m.title, 12);
```

```
ERROR:  function left(jsonb, integer) is not unique
LINE 1: ...ATCH (m:movie {title:'Blade Runner 2049'}) RETURN left(m.tit...
                                                             ^
HINT:  Could not choose a best candidate function. You might need to add explicit type casts.
```

Cast the string, as `left(m.title::text, 12)`, or use `substring`, which has no such clash.

- An `n` larger than the string returns the whole string.

```
MATCH (m:movie {title:'Blade Runner 2049'})
RETURN left(m.title::text, 12) AS head, right(m.title::text, 4) AS tail;
```


© 2026 SKAI Worldwide Co., Ltd.

```
      head       |  tail
-----------------+--------
 "Blade Runner" | "2049"
```

lower

```
lower ( string text ) -> text
```

Returns the string in lower case. This is PostgreSQL's own function, reachable from Cypher because a `jsonb` string coerces to `text`. **`toLower`** is the Cypher spelling and takes `jsonb` directly.

```
RETURN lower('AbCdeFG') AS lower, upper('AbCdeFG') AS upper;
```

```
   lower   |   upper
-----------+-----------
 "abcdefg" | "ABCDEFG"
```

lTrim

```
lTrim ( string jsonb ) -> jsonb
```

Removes whitespace from the start of a string.

Considerations

- Whitespace only. There is no form that takes a set of characters to strip; use PostgreSQL's `ltrim(text, text)` for that.

```
RETURN lTrim('    ABCD    ') AS l, rTrim('    ABCD    ') AS r, trim('    ABCD    ') AS both;
```

```
     l      |     r      |  both
------------+------------+--------
 "ABCD    " | "    ABCD" | "ABCD"
```

replace

```
replace ( string jsonb, old jsonb, new jsonb ) -> jsonb
```

Replaces every occurrence of one substring with another.

Name	Type	Description
`string`	`jsonb`	The string to search.
`old`	`jsonb`	Substring to find. Every occurrence is replaced, not just the first.

© 2026 SKAI Worldwide Co., Ltd.

Name	Type	Description
`new`	`jsonb`	Replacement. May be longer or shorter than `old`, or empty.

Considerations

- `old` and `new` need not be the same length.
- An `old` that does not occur returns the string unchanged.

```
MATCH (m:movie {title:'Terminator 2: Judgment Day'})
RETURN replace(m.title, 'Judgment', 'Judgement') AS british;
```

```
           british
-------------------------------
 "Terminator 2: Judgement Day"
```

reverse

```
reverse ( string text ) -> text
reverse ( list jsonb ) -> jsonb
reverse ( list anyarray ) -> anyarray
```

Reverses a string or a list.

Considerations

- Which overload you get depends on the argument, and the two disagree about strings. A string literal resolves to `reverse(text)` and is reversed. A string property is already `jsonb`, so it resolves to `reverse(jsonb)`, which reverses arrays only:

```
RETURN reverse('Blade Runner 2049'::jsonb);
```

```
ERROR:  reverse(): list is expected but "Blade Runner 2049"
```

A property reference behaves as that cast does, because it is already `jsonb`. Cast the other way to reverse the characters: `reverse(m.title::text)`.

- A list property needs no cast. The `jsonb` overload reverses a list.

```
RETURN reverse('ABCDEFG') AS text, reverse([1,2,3]) AS list;
```

```
   text    |   list
-----------+-----------
 "GFEDCBA" | [3, 2, 1]
```


© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie {title:'Blade Runner 2049'}) RETURN reverse(m.title::text) AS backwards;
```

```
      backwards
____________________
 "9402 rennuR edalB"
```

right

```
right ( string jsonb, n jsonb ) -> jsonb
right ( string text, n integer ) -> text
```

Returns the last `n` characters of a string. It shares `left`'s ambiguity over a property with an integer literal, and the same two ways of resolving it.

rTrim

```
rTrim ( string jsonb ) -> jsonb
```

Removes whitespace from the end of a string. See `lTrim` for a worked example.

split

```
split ( string text, delimiter text ) -> anyarray
```

Cuts a string into a list at every occurrence of a delimiter.

Considerations

- The value is an array of strings. The catalog declares the return type as `anyarray`, so `pg_typeof` reports `anyarray` in SQL. A Cypher `RETURN` boxes it to a `jsonb` list.
- A delimiter that does not occur returns a one-element list holding the whole string.
- The delimiter is a literal string, not a pattern. Use PostgreSQL's `regexp_split_to_array` for a regular expression.

```
RETURN split('Action/Sci-Fi/Drama', '/') AS genres;
```

```
           genres
______________________________
 ["Action", "Sci-Fi", "Drama"]
```

str_size

```
str_size ( string text ) -> integer
```

Returns the number of characters in a string, as an `integer`.

© 2026 SKAI Worldwide Co., Ltd.

Considerations

- `size` and `length` are the Cypher spellings and return `jsonb`. This function exists so that a caller wanting a native `integer` does not have to unbox one. A driver, or SQL around a Cypher query, is such a caller.

```
RETURN str_size('hello') AS n;
```

```
 n
---
 5
```

substring

```
substring ( string jsonb, start jsonb ) -> jsonb
substring ( string jsonb, start jsonb, count jsonb ) -> jsonb
```

Returns part of a string, starting at `start` and running to the end or for `count` characters.

Name	Type	Description
`string`	`jsonb`	The string to cut.
`start`	`jsonb`	0-based offset of the first character.
`count`	`jsonb`	How many characters to take. Omitted, the rest of the string. A `count` past the end simply stops at the end.

Considerations

- `start` is 0-based, unlike SQL's `substring`, which is 1-based. The parser adds the 1 for you, so write the offset you mean.
- Unlike `left` and `right`, this is unambiguous over a property, so it is the safer way to take a prefix of one.

```
RETURN substring('ABCDEFG', 2) AS from_two, substring('ABCDEFG', 2, 3) AS three_from_two,
       substring('ABCDEFG', 4, 10) AS past_end;
```

```
 from_two | three_from_two | past_end
----------+----------------+----------
 "CDEFG"  | "CDE"          | "EFG"
```

toLower

```
toLower ( string jsonb ) -> jsonb
```

Returns the string in lower case.

© 2026 SKAI Worldwide Co., Ltd.

Considerations

- Takes `jsonb`, so it works on a property with no cast, unlike **`reverse`**.
- **`lower`** is PostgreSQL's equivalent and also works. They differ only in the type they take and return.

```
MATCH (m:movie {title:'The Fifth Element'}) RETURN toLower(m.title) AS lower,
toUpper(m.title) AS upper;
```

```
        lower         |        upper
----------------------+----------------------
 "the fifth element" | "THE FIFTH ELEMENT"
```

toUpper

```
toUpper ( string jsonb ) -> jsonb
```

Returns the string in upper case. See **`toLower`** for a worked example.

trim

```
trim ( string jsonb ) -> jsonb
```

Removes whitespace from both ends of a string.

Considerations

- `trim` is a grammar form as well as a function name, so it takes exactly one argument here. SQL's `trim(BOTH 'x' FROM str)` spelling is not available in Cypher:

```
RETURN trim(BOTH 'x' FROM 'xxaxx');
```

```
ERROR:  syntax error at or near "BOTH"
LINE 1: RETURN trim(BOTH 'x' FROM 'xxaxx');
                    ^
```

upper

```
upper ( string text ) -> text
```

Returns the string in upper case. PostgreSQL's own function; see **`lower`**.

Matching without a function

Four string tests are operators rather than functions, and each has an entry in the catalog under a name you do not type:

© 2026 SKAI Worldwide Co., Ltd.

Operator	Implemented by	Returns true when
`STARTS WITH`	`string_starts_with`	The left string begins with the right one
`ENDS WITH`	`string_ends_with`	The left string ends with the right one
`CONTAINS`	`string_contains`	The right string occurs anywhere in the left
`=~`	`string_regex`	The left string matches the right as a regular expression

```
MATCH (m:movie) WHERE m.title STARTS WITH 'The Matrix' RETURN count(*) AS n;
```

```
 n
---
 4
```

None of the four can use a property index. Each is a function call over the property rather than a comparison against it, so a query that filters on one scans the label.

Compatibility

openCypher

openCypher spells the case functions `toLower` and `toUpper` and the trims `lTrim`, `rTrim` and `trim`; all five work. openCypher's `substring` is 0-based, as it is here. `left` and `right` are PostgreSQL functions that openCypher also defines. The ambiguity over a property with an integer literal is a deviation, and `substring` is the portable form.

See also

- **All functions** — every function in one table
- **Type conversion functions** — `toString`, `toStringList`
- **List functions** — what `split` returns

Numeric functions

Arithmetic, logarithmic and trigonometric functions. Most have a `jsonb` overload that takes a property directly, and a `double precision` overload from PostgreSQL. Either way, a Cypher `RETURN` prints the result as a `jsonb` number.

Angles are in radians unless the name ends in `d`.

Examples run against **the example dataset**.

abs

```
abs ( x jsonb ) -> jsonb
abs ( x numeric ) -> numeric
```

Returns the absolute value of a number.

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie {title:'Inception'}) RETURN abs(m.year - 2000) AS gap;
```

```
 gap
-----
 10
```

acos

```
acos ( x jsonb ) -> jsonb
acos ( x double precision ) -> double precision
```

Returns the arc cosine of `x`, in radians. `x` must be between −1 and 1.

asin

```
asin ( x jsonb ) -> jsonb
asin ( x double precision ) -> double precision
```

Returns the arc sine of `x`, in radians. `x` must be between −1 and 1.

atan

```
atan ( x jsonb ) -> jsonb
atan ( x double precision ) -> double precision
```

Returns the arc tangent of `x`, in radians.

atan2

```
atan2 ( y jsonb, x jsonb ) -> jsonb
atan2 ( y double precision, x double precision ) -> double precision
```

Returns the arc tangent of `y / x`, in radians, using the signs of both arguments to place the result in the correct quadrant.

Considerations

- Prefer this to `atan(y / x)`, which cannot tell −15/3 from 15/−3 and so loses the quadrant.

```
RETURN cot(1.5) AS cot, asin(1) AS asin, acos(0.5) AS acos, atan(-1) AS atan, atan2(-1.5,
1.3) AS atan2;
```

```
        cot         |        asin        |        acos        |        atan         |
atan2
--------------------+--------------------+--------------------+---------------------
+---------------------
 0.07091484430265245 | 1.5707963267948966 | 1.0471975511965979 | -0.7853981633974483 | -0.8567
056281827387
```


© 2026 SKAI Worldwide Co., Ltd.

ceil

```
ceil ( x jsonb ) -> jsonb
ceil ( x numeric ) -> numeric
```

Returns the smallest integer that is not less than `x`.

```
RETURN ceil(3.1) AS up, floor(3.1) AS down, round(3.6) AS nearest, sign(-19.93) AS sign;
```

```
 up | down | nearest | sign
----+------+---------+------
 4  | 3    | 4       | -1
```

cos

```
cos ( x jsonb ) -> jsonb
cos ( x double precision ) -> double precision
```

Returns the cosine of an angle given in radians.

cosd

```
cosd ( x double precision ) -> double precision
```

Returns the cosine of an angle given in degrees.

Considerations

- Exact at the quarter turns, where converting the angle first is not: `cosd(90)` is 0 while `cos(radians(90))` is 0.00000000000000006123233995736766.
- The degree forms have no `jsonb` overload, so a property coerces to `double precision`.

```
RETURN sind(30) AS sind, cosd(60) AS cosd, tand(45) AS tand;
```

```
 sind | cosd | tand
------+------+------
 0.5  | 0.5  | 1
```

cot

```
cot ( x jsonb ) -> jsonb
cot ( x double precision ) -> double precision
```

Returns the cotangent of an angle given in radians: the reciprocal of the tangent. See **`atan2`** for a worked example.

degrees

```
degrees ( x jsonb ) -> jsonb
degrees ( x double precision ) -> double precision
```


© 2026 SKAI Worldwide Co., Ltd.

Converts radians to degrees.

```
RETURN degrees(pi()) AS half_turn, radians(180) AS in_radians;
```

```
 half_turn |    in_radians
-----------+------------------
 180       | 3.141592653589793
```

e

```
e ( ) -> numeric
```

Returns Euler's number as a `numeric`.

Considerations

- The result is `numeric`, so it carries more digits than `exp(1)`, which is `double precision`: `2.7182818284590452` against `2.718281828459045`.

```
RETURN e() AS e, exp(1) AS exp_one;
```

```
         e          |      exp_one
--------------------+-------------------
 2.7182818284590452 | 2.718281828459045
```

exp

```
exp ( x jsonb ) -> jsonb
exp ( x numeric ) -> numeric
```

Returns e raised to the power `x`.

floor

```
floor ( x jsonb ) -> jsonb
floor ( x numeric ) -> numeric
```

Returns the largest integer that is not greater than `x`. See **`ceil`** for a worked example.

ln

```
ln ( x double precision ) -> double precision
ln ( x numeric ) -> numeric
```

Returns the natural logarithm of `x`. This is the function **`log`** resolves to in Cypher, and it can also be called under this name.

log

```
log ( x jsonb ) -> jsonb
```


© 2026 SKAI Worldwide Co., Ltd.

Returns the **natural** logarithm of `x`.

Considerations

- In a Cypher query the parser rewrites `log` to **`ln`**, so the result column is named `ln` and there is **no two-argument form**:

```
RETURN log(2, 8);
```

```
ERROR:  function ln(integer, integer) does not exist
LINE 1: RETURN log(2, 8);
               ^
HINT:  No function matches the given name and argument types. You might need to add explicit t
ype casts.
```

- This is the opposite of PostgreSQL's `log`, which is base 10. For base 10 in Cypher, use **`log10`**.

```
RETURN log(27) AS natural, log10(27) AS base_ten;
```

```
      natural      |      base_ten
-------------------+--------------------
 3.295836866004329 | 1.4313637641589874
```

log10

```
log10 ( x double precision ) -> double precision
log10 ( x numeric ) -> numeric
```

Returns the base-10 logarithm of `x`. Not rewritten, so this name means what it says.

pi

```
pi ( ) -> double precision
```

Returns π. See **`degrees`** for a worked example.

radians

```
radians ( x jsonb ) -> jsonb
radians ( x double precision ) -> double precision
```

Converts degrees to radians.

rand

```
rand ( ) -> jsonb
```

Returns a number in the half-open range 0 to 1.

© 2026 SKAI Worldwide Co., Ltd.

Considerations

- `rand()` is marked immutable, so the planner folds it to a single constant. Every row of a query, and every execution of a prepared statement, sees the **same** number. Call PostgreSQL's `random()` instead, which is volatile and works in Cypher.

```
FOR x IN [1,2,3] RETURN x, rand(), random();
```

```
 x |       rand         |       random
---+--------------------+--------------------
 1 | 0.533950385997692 | 0.33269661597409694
 2 | 0.533950385997692 | 0.10517878758331145
 3 | 0.533950385997692 | 0.3874591104544447
```

round

```
round ( x jsonb ) -> jsonb
round ( x numeric [, scale integer ] ) -> numeric
```

Rounds `x` to the nearest integer, halves away from zero.

Considerations

- The `jsonb` form rounds to an integer only. To keep decimal places, cast and use the `numeric` form: `round(x::numeric, 2)`.

sign

```
sign ( x jsonb ) -> jsonb
sign ( x numeric ) -> numeric
```

Returns −1, 0 or 1 according to the sign of `x`. See **`ceil`** for a worked example.

sin

```
sin ( x double precision ) -> double precision
```

Returns the sine of an angle given in radians.

Considerations

- Of the trigonometric functions, sine alone has no `jsonb` overload under its own name: the `jsonb` sine is catalogued as `jsonb_sin`. Nothing is lost. A property coerces to `double precision` and `sin()` resolves to the PostgreSQL function. But the result type differs from its neighbours over a `jsonb` argument.

```
RETURN sin(0.5) AS sin, cos(0) AS cos, tan(0) AS tan;
```

```
        sin         | cos | tan
--------------------+-----+-----
 0.479425538604203 | 1   | 0
```


© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie {title:'The Matrix'})
RETURN pg_typeof(cos(m.embedding[0])) AS cos_t, pg_typeof(sin(m.embedding[0])) AS sin_t,
       pg_typeof(tan(m.embedding[0])) AS tan_t;
```

```
  cos_t  |       sin_t        |  tan_t
---------+--------------------+---------
 "jsonb" | "double precision" | "jsonb"
```

sind

```
sind ( x double precision ) -> double precision
```

Returns the sine of an angle given in degrees. See `cosd` for a worked example.

sqrt

```
sqrt ( x jsonb ) -> jsonb
sqrt ( x numeric ) -> numeric
```

Returns the square root of `x`.

Considerations

- A negative argument is an error rather than null:

```
RETURN sqrt(-16);
```

```
ERROR:  cannot take square root of a negative number
```

```
MATCH (m:movie {title:'The Matrix'}) RETURN sqrt(m.year) AS root;
```

```
        root
--------------------
 44.710177812216314
```

tan

```
tan ( x jsonb ) -> jsonb
tan ( x double precision ) -> double precision
```

Returns the tangent of an angle given in radians. See `sin` for a worked example.

tand

```
tand ( x double precision ) -> double precision
```

Returns the tangent of an angle given in degrees. See `cosd` for a worked example.

© 2026 SKAI Worldwide Co., Ltd.

Arithmetic on properties

The operators `+`, `-`, `*`, `/`, `%` and `^` work on properties, but on `jsonb` operands they follow `jsonb`'s own rules, not SQL's: two integers divide as integers.

```
RETURN 7 % 3 AS modulo, 2 ^ 10 AS power, 7 / 2 AS integer_division, 7.0 / 2 AS real_division;
```

```
 modulo | power | integer_division |   real_division
--------+-------+------------------+--------------------
 1      | 1024  | 3                | 3.5000000000000000
```

Each is implemented by a backtick-named operator over `jsonb`; see **arithmetic operators on jsonb**.

Compatibility

openCypher

`log()` is base e here and base 10 in PostgreSQL. openCypher spells the natural logarithm `log` too, so a Cypher query is portable and a SQL one is not. openCypher's `e()` and `rand()` exist; `rand()` deviates by being folded to a constant. `haversin()` is not supported.

See also

- **All functions** — every function in one table
- **Aggregate functions** — `avg`, `sum`, `stdev`
- **Type conversion functions** — `toInteger`, `toFloat`

Path functions

Functions over a `graphpath`, and the three path-finding forms that produce one. A `graphpath` is the alternating sequence of vertices and edges that a `MATCH p = (...)` binds.

A path is bound by naming it: `MATCH p = (a)-[r]->(b)`. These functions take the **variable**, not the pattern.

Examples run against **the example dataset**.

allshortestpaths

```
allshortestpaths ( pattern ) -> graphpath
```

Finds every shortest path between two vertices. It returns one row per path when it names a path variable, and an array of paths when it is used as an expression.

Considerations

- The pattern must hold exactly one relationship, or the parser raises `only one relationship is allowed`. Give that relationship a length range, such as `*..4`, rather than writing more hops.
- Multi-type relationship patterns are not supported anywhere, `allshortestpaths` included. Match an untyped edge and filter with `WHERE label(r) IN [...]`.
- As an expression it yields `graphpath[]`, so `length()` over it counts the paths found, not the edges in one path.

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'Inception'})
RETURN length(allshortestpaths((a)-[:related_to*..4]->(b))) AS paths_found;
```

```
 paths_found
-------------
 1
```

dijkstra

```
dijkstra ( pattern, weight ) -> graphpath
dijkstra ( pattern, weight, condition ) -> graphpath
dijkstra ( pattern, weight, LIMIT count ) -> graphpath
dijkstra ( pattern, weight, condition, LIMIT count ) -> graphpath
```

Finds the cheapest path between two vertices, where each edge costs what `weight` evaluates to.

Name	Type	Description
`pattern`	path chain	Exactly one relationship between two vertices, for example `(a)-[e:related_to]->(b)`. Give the relationship a variable if `weight` reads a property of it.
`weight`	numeric expression	Cost of traversing one edge. Must be greater than 0 for every edge considered.
`condition`	boolean expression	Optional filter on the edge. Only edges satisfying it are traversed.
`count`	integer	Optional. Number of alternative paths to return, cheapest first. Must be greater than 0.

Considerations

- Written as a pattern part it binds a path: `p = dijkstra(...)`. Written as `(p, w) = dijkstra(...)` it binds the path and its total weight, which is otherwise not available.
- Without `LIMIT` the count defaults to 1, so one path comes back.
- The pattern accepts neither a property constraint nor a variable-length relationship. Filter in `WHERE` instead.

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'Inception'}),
      p = dijkstra((a:movie {title:'The Matrix'})-[e:related_to]->(b), e.weight)
RETURN p;
```

```
ERROR:  property constraint is not supported
LINE 2:       p = dijkstra((a:movie {title:'The Matrix'})-[e:related...
                           ^
```


© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'Inception'}),
      p = dijkstra((a)-[e:related_to*]->(b), e.weight)
RETURN p;
```

```
ERROR:  variable length relationship is not supported
```

- The third positional argument is a condition, so a number there raises `cannot cast 1 (jsonb type numeric) to boolean`.
- Two more refusals come at run time, once an edge is reached, so they depend on the data rather than on the query alone. A weight that evaluates to 0 or less reports `WEIGHT must be larger than 0`, and `LIMIT 0` reports `LIMIT must be larger than 0`.
- Used as a bare expression rather than assigned, it yields `graphpath[]`.

The cheapest route is not the shortest one. Two hops through Total Recall cost 4 + 2, which beats the single-hop alternatives on this graph:

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'Inception'}),
      (p, w) = dijkstra((a)-[e:related_to]->(b), e.weight)
RETURN [x IN nodes(p) | x.title] AS route, w AS weight;
```

```
                   route                    | weight
--------------------------------------------+--------
 ["The Matrix", "Total Recall", "Inception"] | 6
```

Leaving the far endpoint unconstrained gives the cheapest route to every reachable film:

```
MATCH p = dijkstra((a:movie)-[e:related_to]->(b:movie), e.weight, LIMIT 1)
WHERE a.title = 'The Matrix'
RETURN [x IN nodes(p) | x.title] AS route ORDER BY route;
```

```
                                          route
-------------------------------------------------------------------------------------------
 ["The Matrix"]
 ["The Matrix", "The Matrix Reloaded"]
 ["The Matrix", "The Terminator"]
 ["The Matrix", "Total Recall"]
 ["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
 ["The Matrix", "The Terminator", "Terminator 2: Judgment Day"]
 ["The Matrix", "Total Recall", "Inception"]
 ["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions", "The Matrix Resurrections"]
```

The rows come back in no particular order without an `ORDER BY`; the sort above only makes the page reproducible.

NOTE

A path found by `dijkstra` is a `graphpath` like any other, so `nodes()`, `relationships()` and `length()` all read it. What no other path carries is the total weight, and only the `(p, w) =` form exposes it.

© 2026 SKAI Worldwide Co., Ltd.

edges

```
edges ( p graphpath ) -> edge[]
```

Returns the edges of a path, in order. `relationships` is the same function under the name Cypher uses.

end_vertex

```
end_vertex ( e edge ) -> vertex
```

Returns the vertex an edge points at. `endNode` is the same function under the name Cypher uses; both are listed on the scalar page.

Considerations

- Reading the vertex means fetching its row. When the identity is enough, use `end_id`, which reads a column of the edge itself.

length

```
length ( p graphpath ) -> jsonb
length ( vs vertex[] ) -> jsonb
length ( es edge[] ) -> jsonb
length ( ps graphpath[] ) -> jsonb
```

Over a path, returns the number of edges in it. Over an array, returns the number of elements.

Considerations

- The number of edges, not of elements: a one-edge path has length 1 while `cardinality` of the same path is 3.
- Because `allshortestpaths()` and an unassigned `dijkstra()` yield `graphpath[]`, `length()` over one of those counts paths.
- Over a string, a list or a map, `length` is a different function; see `length` on the scalar page.

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'The Matrix Resurrections'})
RETURN length(shortestpath((a)-[:related_to*]->(b))) AS hops;
```

```
 hops
------
 3
```

nodes

```
nodes ( p graphpath ) -> vertex[]
```

Returns the vertices of a path, in order, starting with the one the path starts at.

Considerations

- The result is `vertex[]`, a native array, so a Cypher `RETURN` does not box it to `jsonb`. Subscripting is still 0-based.
- `vertices` is the same function under another name.

© 2026 SKAI Worldwide Co., Ltd.

- Pass the path variable. A pattern written inline is not an expression and will not parse.
- Printing whole vertices is rarely what you want, since each carries its full property bag. Project the field you need with a list comprehension, as below.

```
MATCH p=(:person {name:'Christopher Nolan'})-[:directed]->(:movie {title:'Inception'})
RETURN [x IN nodes(p) | x.name] AS names, [x IN nodes(p) | x.title] AS titles;
```

```
           names            |       titles
----------------------------+---------------------
 ["Christopher Nolan", null] | [null, "Inception"]
```

Every vertex of the path is projected, so the key a given vertex does not carry comes back as null.

relationships

```
relationships ( p graphpath ) -> edge[]
```

Returns the edges of a path, in order. Identical to `edges`.

```
MATCH p=(:person {name:'Christopher Nolan'})-[:directed]->(:movie {title:'Inception'})
RETURN relationships(p) AS edges;
```

```
          edges
---------------------------
 [directed[6.8][4.8,3.5]{}]
```

shortestpath

```
shortestpath ( pattern ) -> graphpath
```

Finds one shortest path between two vertices, by number of edges. If several are equally short, one of them is returned.

Considerations

- The pattern must hold exactly one relationship, and that relationship carries the length range. Writing the hops out is refused:

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'Inception'})
RETURN shortestpath((a)-[:related_to]->()-[:related_to]->(b));
```

```
ERROR:  only one relationship is allowed
LINE 2: RETURN shortestpath((a)-[:related_to]->()-[:related_to]->(b)...
                            ^
```

- Which of several equally short paths you get is not defined. Use `allshortestpaths` when you need them all.
- Returns no row when no path exists, rather than a null path.
- Multi-type relationship patterns are not supported here either.

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'The Matrix Resurrections'})
RETURN [x IN nodes(shortestpath((a)-[:related_to*]->(b))) | x.title] AS route;
```

```
                                                route
-------------------------------------------------------------------------------------------
 ["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions", "The Matrix Resurrections"]
```

start_vertex

```
start_vertex ( e edge ) -> vertex
```

Returns the vertex an edge leaves. `startNode` is the same function under the name Cypher uses; see **`startNode`**.

A vertex is a composite, so a property can be read straight off the result:

```
MATCH (:person {name:'Keanu Reeves'})-[e:acted_in]->(:movie {title:'The Matrix'})
RETURN start_vertex(e).name AS actor, end_vertex(e).title AS film;
```

```
     actor      |     film
----------------+--------------
 "Keanu Reeves" | "The Matrix"
```

vertices

```
vertices ( p graphpath ) -> vertex[]
```

Returns the vertices of a path, in order. Identical to **`nodes`**.

Compatibility

openCypher

`shortestPath` and `allShortestPaths` are spelled without regard to case, so openCypher's capitalisation works. openCypher has no `dijkstra`; weighted shortest paths there come from a procedure library.

openCypher allows a relationship pattern to name several types, as `-[r:acted_in|directed]->`. That raises `multiple types for relationship not supported` at every length and inside every path-finding form. Match an untyped relationship and test the label instead:

```
MATCH (p:person)-[r]->(m:movie) WHERE label(r) IN ['acted_in','directed'] RETURN count(r) AS
n;
```

```
 n
----
 46
```

See also

- All functions — every function in one table
- Scalar functions — `startNode`, `endNode`, `start_id`, `end_id`
- List functions — `cardinality` over a path

© 2026 SKAI Worldwide Co., Ltd.

- **Syntax** — path patterns and flexible length

Type conversion functions

Functions that turn a value into another type. Two families sit here. The `to…` functions read a value and return null when it does not name one. The `to…OrNull` variants do the same but never raise.

A cast is the other way to convert, and Cypher accepts `::` with most type names:

```
RETURN '42'::int AS i, 42::numeric AS n, 'true'::boolean AS b, '2020-01-01'::date AS d;
```

```
 i  | n  | b |      d
----+----+---+--------------
 42 | 42 | t | "2020-01-01"
```

`::timestamp` and `::time` are the exceptions:

```
RETURN pg_typeof('2020-01-01 10:00'::timestamp);
```

```
ERROR:  syntax error at or near "timestamp"
LINE 1: RETURN pg_typeof('2020-01-01 10:00'::timestamp);
                                             ^
```

A Cypher cast accepts an ordinary type name, and it has rules of its own for the numeric, character, bit, json and interval spellings. It has no rule for the date and time spellings. `TIMESTAMP` and `TIME` are keywords, so a cast to either needs such a rule and there is none. `date`, `timestamptz` and `timetz` are not keywords, so they arrive as ordinary type names and work. `::timestamptz` is the usual way round it, and wrapping the Cypher in SQL is the other.

Examples run against **the example dataset**.

toBoolean

```
toBoolean ( value text ) -> boolean
toBoolean ( value "any" ) -> boolean
```

Returns the boolean a value names, or null when it names none.

Considerations

- Accepts `true`, `false`, `t`, `f`, `on`, `off`, `1` and `0`, in any case. Anything else returns null rather than raising.
- It is stricter than the cast: `toBoolean('yes')` is null, while `'yes'::boolean` is true.
- A `RETURN` leaves a boolean unboxed, so it prints as `t` or `f` and null prints as blank.

```
RETURN toBoolean('TRUE') AS t, toBoolean('false') AS f, toBoolean(true) AS already;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 t | f | already
---+---+---------
 t | f | t
```

toBooleanOrNull

```
toBooleanOrNull ( value text ) -> boolean
toBooleanOrNull ( value "any" ) -> boolean
```

Returns the boolean a value names, or null. Where `toBoolean` may raise on a type it cannot read at all, this returns null.

```
RETURN toBooleanOrNull('yes') AS unrecognised, toBooleanOrNull('true') AS recognised;
```

```
 unrecognised | recognised
--------------+------------
              | t
```

toFloat

Added in 2.18

```
toFloat ( value "any" ) -> double precision
```

Returns the floating-point number a value names, or null.

Considerations

- One overload, taking `"any"`, so it accepts a string, an integer or a `jsonb` property without a cast.
- A value that is not a number returns null rather than raising.
- The result is `double precision`. A Cypher `RETURN` boxes it to a `jsonb` number.

```
RETURN toFloat('3.5') AS from_string, toFloat(7) AS from_integer, toFloat('abc') AS
unreadable;
```

```
 from_string | from_integer | unreadable
-------------+--------------+------------
 3.5         | 7            |
```

toFloatList

```
toFloatList ( list jsonb ) -> jsonb
toFloatList ( list anyarray ) -> double precision[]
```

Converts every element of a list to a floating-point number.

Considerations

- **Which overload you get decides the return type.** A Cypher list is `jsonb`, so `toFloatList([...])` returns `jsonb`. Only a PostgreSQL array argument gives `double precision[]`, and that form is reachable from SQL.

© 2026 SKAI Worldwide Co., Ltd.

- An element that is not a number becomes null; the list keeps its length.

```
RETURN toFloatList(['1.4','2.5','true']) AS floats, pg_typeof(toFloatList(['1.4'])) AS type;
```

```
      floats      |  type
------------------+---------
 [1.4, 2.5, null] | "jsonb"
```

```
SELECT toFloatList(ARRAY['1.5','2']) AS floats, pg_typeof(toFloatList(ARRAY['1.5','2'])) AS
type;
```

```
 floats  |        type
---------+--------------------
 {1.5,2} | double precision[]
```

toInteger

Added in 2.18

```
toInteger ( value "any" ) -> bigint
```

Returns the integer a value names, or null.

Considerations

- Truncates towards zero rather than rounding: `toInteger(3.7)` is 3.
- A boolean converts, with true becoming 1.
- A value that is not a number returns null rather than raising.
- The result is `bigint`.

```
RETURN toInteger('42') AS from_string, toInteger(3.7) AS truncated,
       toInteger('abc') AS unreadable, toInteger(true) AS from_boolean;
```

```
 from_string | truncated | unreadable | from_boolean
-------------+-----------+------------+--------------
 42          | 3         |            | 1
```

toIntegerList

```
toIntegerList ( list jsonb ) -> jsonb
toIntegerList ( list anyarray ) -> bigint[]
```

Converts every element of a list to an integer. Which overload applies decides the return type, exactly as for `toFloatList`: a Cypher list gives `jsonb`, a PostgreSQL array gives `bigint[]`.

```
RETURN toIntegerList(['1','2','x']) AS integers;
```


© 2026 SKAI Worldwide Co., Ltd.

```
   integers
--------------
 [1, 2, null]
```

toString

```
toString ( value jsonb ) -> jsonb
```

Returns the string form of a value.

Considerations

- A value that is already a string comes back unchanged.
- A number, a boolean, a list and a map all convert. The result is a `jsonb` string, so a `RETURN` prints it with quotes.

```
RETURN toString(11.5) AS number, toString('already a string') AS string, toString(true) AS
boolean;
```

```
 number |       string       | boolean
--------+--------------------+---------
 "11.5" | "already a string" | "true"
```

toStringList

```
toStringList ( list jsonb ) -> jsonb
toStringList ( list anyarray ) -> text[]
```

Converts every element of a list to a string. Which overload applies decides the return type, as for **`toFloatList`**: a Cypher list gives `jsonb`, a PostgreSQL array gives `text[]`.

```
RETURN toStringList([1, 2.5, true]) AS strings;
```

```
       strings
---------------------
 ["1", "2.5", "true"]
```

toStringOrNull

```
toStringOrNull ( value "any" ) -> text
```

Returns the string form of a value, or null when the value has no string form.

Considerations

- Returns `text`, not `jsonb`, unlike **`toString`**. A Cypher `RETURN` boxes it back to a `jsonb` string, so on the page the two look alike; in SQL they do not.
- A list returns null: a list has no single string form.

© 2026 SKAI Worldwide Co., Ltd.

```
RETURN toStringOrNull(11.5) AS number, toStringOrNull([1,2]) AS list,
       pg_typeof(toStringOrNull(1)) AS type;
```

```
 number | list |  type
--------+------+--------
 "11.5" |      | "text"
```

What is not converted

`::timestamp` and `::time` cannot be written in Cypher, as above. There is also no `toMap`, no `toList` and no `toDate`. Build a map with `{...}`, a list with `[...]`, and read a date with `::date`.

A graph element converts to its property bag and nothing else. `v::jsonb` gives the map, the same value as `properties(v)`. There is no conversion from a map back to a vertex.

Compatibility

openCypher

`toInteger`, `toFloat`, `toString` and `toBoolean` are openCypher's names and behave as openCypher describes. The one exception is that a value they cannot read returns null rather than raising. The `…OrNull` variants and the three list conversions are extensions.

openCypher's `toInteger` rounds; this one truncates.

See also

- **All functions** — every function in one table
- **String functions** — `split`, `toLower`, `toUpper`
- **List functions** — the lists these conversions produce
- **jsonb operators and functions** — `to_jsonb` and the `#>> '{}'` idiom

jsonb operators and functions

Properties **are** `jsonb`. A vertex is the composite `(id graphid, properties jsonb)`, and an edge adds `start` and `end`. So every property you read in Cypher, and every property column you touch in SQL, is a value of PostgreSQL's `jsonb` type. That has two consequences to take in before anything else on this page. A string property comes back quoted, and the whole `jsonb` operator set applies to it.

Examples run against **the example dataset**.

Property access operators

These four take a `jsonb` value on the left. The two with a doubled arrow return `text`; the two with a single arrow return `jsonb`.

Operator	Right operand	Returns	Result
`->`	`text` key or `integer` index	`jsonb`	The value at that key or index
`->>`	`text` key or `integer` index	`text`	The same value, with a string unquoted

© 2026 SKAI Worldwide Co., Ltd.

Operator	Right operand	Returns	Result
`#>`	`text[]` path	`jsonb`	The value at that path
`#>>`	`text[]` path	`text`	The same value, with a string unquoted

```
SELECT properties -> 'name' AS quoted, properties ->> 'name' AS plain
FROM moviekb.person ORDER BY id LIMIT 1;
```

```
     quoted     |    plain
----------------+--------------
 "Keanu Reeves" | Keanu Reeves
```

```
SELECT '{"a":{"b":[1,2]}}'::jsonb #> '{a,b}' AS at_path,
       '{"a":{"b":[1,2]}}'::jsonb #>> '{a,b,0}' AS at_path_text;
```

```
 at_path | at_path_text
---------+--------------
 [1, 2]  | 1
```

Getting an unquoted scalar out

A Cypher `RETURN` boxes a scalar output column to `jsonb`. So `RETURN m.title` prints `"Inception"`, quotes included, and so does `RETURN m.title::text`. The quotes are part of the JSON, not an artefact of `psql`.

The way out is `#>> '{}'`: an empty path selects the whole document, and `#>>` returns it as `text`. Wrap the Cypher in SQL and apply it there:

```
SELECT t AS boxed, t #>> '{}' AS unboxed
FROM (MATCH (m:movie {title:'Inception'}) RETURN m.title AS t) q;
```

```
    boxed     |  unboxed
--------------+-----------
 "Inception" | Inception
```

Reach for this idiom whenever a Cypher result has to join against, or be compared with, ordinary `text`. A join onto a relational table is one such case.

Containment and existence operators

Operator	Right operand	Returns true when
`@>`	`jsonb`	The left document contains the right one
`<@`	`jsonb`	The left document is contained in the right one

© 2026 SKAI Worldwide Co., Ltd.

Operator	Right operand	Returns true when
`?`	`text`	The key or string element is present
`?|`	`text[]`	Any of the keys is present
`?&`	`text[]`	All of the keys are present

```
SELECT properties @> '{"name":"Keanu Reeves"}' AS contains,
       properties ? 'born' AS has_born,
       properties ?& ARRAY['name','born'] AS has_both
FROM moviekb.person ORDER BY id LIMIT 1;
```

```
 contains | has_born | has_both
----------+----------+----------
 t        | t        | t
```

A property index cannot serve any of these; each is a containment test over the whole bag rather than a comparison against one key. A GIN index on `properties` can.

Modification operators

Operator	Right operand	Result
`||`	`jsonb`	The two documents merged, the right winning on a shared key
`-`	`text` key or `integer` index	The document without that key or element
`#-`	`text[]` path	The document without the value at that path

```
SELECT '{"a":1}'::jsonb || '{"b":2}'::jsonb AS merged,
       '{"a":1,"b":2}'::jsonb - 'a' AS without_a,
       '{"a":{"b":1}}'::jsonb #- '{a,b}' AS without_path;
```

```
      merged      | without_a | without_path
------------------+-----------+--------------
 {"a": 1, "b": 2} | {"b": 2}  | {"a": {}}
```

In Cypher these are spelled as clauses instead: `SET n += {city:'Seoul'}` merges, and `REMOVE n.city` deletes a key.

Arithmetic operators on jsonb

PostgreSQL already gives `-` and `||` a meaning on `jsonb`, so AgensGraph could not reuse them for arithmetic. It adds eight operators whose names are wrapped in backticks. The Cypher parser emits those when you write `+`, `-`, `*`, `/`, `%` or `^` on properties.

© 2026 SKAI Worldwide Co., Ltd.

Operator	Implemented by	OID	Meaning
`` `+` ``	`jsonb_add`	7176	Addition; concatenation for two strings or two lists
`` `-` ``	`jsonb_sub`	7178	Subtraction
`` `*` ``	`jsonb_mul`	7180	Multiplication
`` `/` ``	`jsonb_div`	7182	Division, integer when both operands are integers
`` `%` ``	`jsonb_mod`	7184	Modulo
`` `^` ``	`jsonb_pow`	7186	Exponentiation
`` `+` `` (prefix)	`jsonb_uplus`	7190	Unary plus
`` `-` `` (prefix)	`jsonb_uminus`	7191	Unary minus

Backtick is an operator character in this fork's lexer (`scan.l:369`), so these names are typeable in SQL as well. Both operands must be `jsonb` :

```
SELECT '"a"'::jsonb `+` '"b"'::jsonb AS strings, '[1]'::jsonb `+` '[2]'::jsonb AS lists;
```

```
 strings | lists
---------+--------
 "ab"    | [1, 2]
```

NOTE

Write Cypher arithmetic with the ordinary operators — `n.a + n.b` — and let the parser choose these. The backtick names are here so you can recognise them in an `EXPLAIN` plan, in an error message, or in a `pg_operator` listing. The implementation functions can also be called directly: `jsonb_add('1'::jsonb, '2'::jsonb)` returns 3.

ag_property_text

Added in 2.18

```
ag_property_text ( properties jsonb, key text, expected_kind "char" ) -> text
```

Returns a property's value as `text` , but only when the stored value is of the JSON kind asked for.

Name	Type	Description
`properties`	`jsonb`	The property bag to read.

© 2026 SKAI Worldwide Co., Ltd.

Name	Type	Description
`key`	`text`	Name of the property.
`expected_kind`	`"char"`	`'n'` for a number, `'s'` for a string, `'b'` for a boolean.

Considerations

- This backs typed-column property promotion. `->>` returns the text of a value, and text carries no kind, so the string `"5"` and the number `5` are indistinguishable through it. A column of integers would answer 5 for a property that holds a string. This function asks the kind first.
- A kind mismatch **raises**, it does not return null: `property "n" holds a number` with the detail `A column promoting this property holds strings.`
- A property that is absent, or present and JSON null, returns null.
- It is not intended to be written by hand. Read a property with `n.key` in Cypher or `properties ->> 'key'` in SQL.

```
SELECT ag_property_text('{"name":"x","n":5}'::jsonb, 'n', 'n') AS number,
       ag_property_text('{"name":"x"}'::jsonb, 'missing', 's') AS absent;
```

```
 number | absent
--------+--------
 5      |
```

cypher_to_jsonb

```
cypher_to_jsonb ( value anyelement ) -> jsonb
```

Boxes a value to `jsonb` using Cypher's rules, which differ from `to_jsonb`'s for string types.

Considerations

- This is the function a Cypher `RETURN` inserts around a scalar output column. It appears in a plan at the point where the boxing happens.
- It is strict, so a null stays null.
- The parser refuses to wrap a graph element in it. That refusal makes `coalesce(m, 0)` and a `jsonb` list of vertices fail. It lives in the parser's `coerce_to_jsonb()`, not in the function. Called by name on a `vertex`, the function returns the whole composite, `id` and `tid` included, which is not the property bag that `v::jsonb` gives you. Use **`vertex_to_jsonb`** or the cast for that.

```
SELECT cypher_to_jsonb('abc'::text) AS from_text, cypher_to_jsonb(42) AS from_integer;
```

```
 from_text | from_integer
-----------+--------------
 "abc"     | 42
```


© 2026 SKAI Worldwide Co., Ltd.

edge_to_jsonb

```
edge_to_jsonb ( e edge ) -> jsonb
```

Returns an edge's property bag. This is the function the cast `e::jsonb` uses, and it returns the same value as `properties(e)`.

Considerations

- The endpoints are not included — only the properties. Read them with `start_id` and `end_id`.

```
MATCH ()-[e:related_to]->() WITH e LIMIT 1 RETURN edge_to_jsonb(e) AS bag;
```

```
      bag
---------------
 {"weight": 1}
```

graphid_in_jsonb_array

```
graphid_in_jsonb_array ( id graphid, elements jsonb ) -> boolean
```

Returns true when a `graphid` is one of the graph elements in a `jsonb` array.

Considerations

- The parser emits this for `id(n) IN [...]` so that a list of elements can be tested without unboxing each one. A `graphid` boxes to a `jsonb` string, and this compares against that form.
- Written by hand it is a plain membership test.

```
SELECT graphid_in_jsonb_array('4.1'::graphid, '["4.1","4.2"]'::jsonb) AS present;
```

```
 present
---------
 t
```

jsonb_array_length

```
jsonb_array_length ( list jsonb ) -> integer
```

Returns the number of elements in a `jsonb` array, as an `integer`.

Considerations

- PostgreSQL's own function. `cardinality` and `length` are the Cypher spellings and return `jsonb`; this one returns a native `integer` and raises on a value that is not an array.

```
SELECT jsonb_array_length('[1,2,3]');
```


© 2026 SKAI Worldwide Co., Ltd.

```
 jsonb_array_length
--------------------
                  3
```

to_jsonb

```
to_jsonb ( value anyelement ) -> jsonb
```

Converts any SQL value to `jsonb`, mapping a composite to an object and an array to an array.

Considerations

- This is how you get relational rows into a graph. `LOAD FROM t AS r CREATE (:label = to_jsonb(row_to_json(r)))` turns each row into a property bag.
- On a graph element it converts the whole composite, not the property bag. `v::jsonb` and **`vertex_to_jsonb`** give the bag.

```
SELECT to_jsonb(42) AS number, to_jsonb('x'::text) AS string;
```

```
 number | string
--------+--------
 42     | "x"
```

vertex_to_jsonb

```
vertex_to_jsonb ( v vertex ) -> jsonb
```

Returns a vertex's property bag. This is the function the cast `v::jsonb` uses, and it returns the same value as **`properties(v)`**.

```
MATCH (n:person {name:'Keanu Reeves'}) RETURN vertex_to_jsonb(n) AS bag;
```

```
                  bag
-----------------------------------------
 {"born": 1964, "name": "Keanu Reeves"}
```

Ordering and comparison

`jsonb` has its own total order, and it is not the order the values would have in their own types. Kind comes first: null, then string, then number, then boolean, then array, then object. Only values of the same kind compare by value:

```
SELECT v FROM (VALUES ('1'::jsonb),('"a"'),('true'),('null'),('[1]'),('{"k":1}')) t(v) ORDER
BY v;
```

```
    v
----------
 null
 "a"
 1
```


© 2026 SKAI Worldwide Co., Ltd.

```
true
[1]
{"k": 1}
```

This matters wherever a Cypher result is sorted or aggregated. `min()` and `max()` over a `jsonb` column follow that order, so a property that holds numbers in some rows and strings in others will not sort the way a reader expects. Read it through a typed column, or cast it, when the order has to be the type's own.

Two integers also divide as integers, because `jsonb` keeps the distinction:

```
RETURN 7 / 2 AS integer_division, 7.0 / 2 AS real_division;
```

```
 integer_division |   real_division
------------------+--------------------
 3                | 3.5000000000000000
```

See also

- **All functions** — every function in one table
- **Scalar functions** — `properties`, `keys`, `length`
- **Type conversion functions** — the `to…` family
- **Combining SQL and Cypher** — where `#>> '{}'` earns its keep

Server settings

AgensGraph adds eleven settings to PostgreSQL's several hundred. This page documents those eleven. Every other setting is PostgreSQL's, unchanged, and **PostgreSQL's server-configuration chapter** describes it. For **how** and **where** to set something, see **Configuration**, which covers `postgresql.conf`, `ALTER SYSTEM`, `ALTER DATABASE`, `ALTER ROLE` and `SET`.

One of the eleven, `graph_path`, is needed by every session that runs Cypher. The rest have working defaults and you can ignore them until you have a reason not to.

The settings

Every default below is `pg_settings.boot_val`, read from a running server:

```
SELECT name, vartype AS type, boot_val AS default, coalesce(unit,'') AS unit, context
FROM pg_settings
WHERE name IN ('allow_null_properties','auto_gather_graphmeta','cypher_allow_unsafe_ddl',
               'eager_mem','enable_eager','enable_graph_ddl','enable_graph_dml',
               'enable_graph_endpoint_elision','enable_multiple_update',
               'enable_property_promotion','graph_path')
ORDER BY name;
```

```
            name              |  type   | default | unit |  context
------------------------------+---------+---------+------+-----------
 allow_null_properties        | bool    | off     |      | user
 auto_gather_graphmeta        | bool    | off     |      | superuser
 cypher_allow_unsafe_ddl      | bool    | off     |      | user
 eager_mem                    | integer | 4096    | kB   | user
 enable_eager                 | bool    | on      |      | user
```


© 2026 SKAI Worldwide Co., Ltd.

```
enable_graph_ddl              | bool    | off     |       | user
enable_graph_dml              | bool    | off     |       | superuser
enable_graph_endpoint_elision | bool    | on      |       | user
enable_multiple_update        | bool    | on      |       | user
enable_property_promotion     | bool    | on      |       | user
graph_path                    | string  |         |       | user
```

Read the two context values this way:

`user`	Any role can change it, in any session, at any time. It can also be set in `postgresql.conf`, per database or per role.
`superuser`	Only a superuser can change it in a session, or a role that has been granted `SET` on that parameter. Everyone else gets `permission denied to set parameter`.

Query `pg_settings` rather than reading `postgresql.conf.sample`. The sample file shipped in the installation directory lists `enable_eager` and `enable_multiple_update` as `off`. Both really default to `on`, so a table copied from that file would be wrong in two rows.

agversion, a preset rather than a setting

`agversion` reports the AgensGraph version. Its context is `internal`, which makes it a preset: something the server tells you, not something you configure. It cannot appear in `postgresql.conf`, and setting it is refused:

```
SET agversion = '1';
```

```
ERROR:  parameter "agversion" cannot be changed
```

```
SELECT name, setting, context, category FROM pg_settings WHERE name = 'agversion';
```

```
   name    |  setting   | context  |    category
-----------+------------+----------+-----------------
 agversion | 2.18.6.1 | internal | Preset Options
```

Read it with `SHOW`, which is the usual way to check what you are connected to:

```
SHOW agversion;
```

```
 agversion
------------
 2.18.6.1
```

`version()` and `server_version` report the PostgreSQL version the release is built on. A client that needs to know both should read both.

© 2026 SKAI Worldwide Co., Ltd.

graph_path

```
string · default '' (empty) · context user
```

Names the graph a session works in. Every unqualified label name in a Cypher or GQL statement resolves in that graph. Every graph DDL statement creates or alters a label there. It is the graph analogue of `search_path`, and the one setting you cannot skip.

Nothing resolves without it:

```
RESET graph_path;
MATCH (n:person) RETURN count(n);
```

```
ERROR:  graph_path is NULL
HINT:  Use SET graph_path
```

Set it and the same statement runs:

```
SET graph_path = moviekb;
SHOW graph_path;
```

```
 graph_path
------------
 moviekb
```

`CURRENT_GRAPH` reads the same value as a bare keyword inside a statement:

```
SELECT CURRENT_GRAPH;
```

```
 current_graph
---------------
 moviekb
```

Exactly one graph name

Unlike `search_path`, `graph_path` is not a list. A comma-separated value is not parsed as two names. It is taken as one name, and that name does not exist:

```
SET graph_path = 'moviekb, twins';
```

```
ERROR:  invalid value for parameter "graph_path": "moviekb, twins"
DETAIL:  graph "moviekb, twins" does not exist.
```

So a session reads and writes one graph. To reach two graphs in one statement, use SQL, which sees them as two schemas. **Combining SQL and Cypher** has the patterns.

When the value is checked

A graph that does not exist is refused at `SET` time rather than at query time:

© 2026 SKAI Worldwide Co., Ltd.

```
SET graph_path = nosuchgraph;
```

```
ERROR:  invalid value for parameter "graph_path": "nosuchgraph"
DETAIL:  graph "nosuchgraph" does not exist.
```

The name is an identifier, so it folds to lower case unless you quote it. A graph created with a mixed-case name needs the quotes:

```
CREATE GRAPH "MixedCase";
SET graph_path = MixedCase;
```

```
ERROR:  invalid value for parameter "graph_path": "mixedcase"
DETAIL:  graph "mixedcase" does not exist.
```

```
SET graph_path = "MixedCase";
SHOW graph_path;
```

```
 graph_path
------------
 MixedCase
```

```
DROP GRAPH "MixedCase" CASCADE;
SET graph_path = moviekb;
```

One value for a database or a role

`SET` lasts for the session. Two wider scopes save repeating it, and both take effect on the next connection rather than the current one.

For everyone who connects to a database:

```
ALTER DATABASE docs SET graph_path = moviekb;
```

```
ALTER DATABASE
```

For one role, whichever database it connects to:

```
ALTER ROLE analyst SET graph_path = moviekb;
```

```
ALTER ROLE
```

Either is visible in `pg_db_role_setting`, and a role setting wins over a database setting:

```
SELECT r.rolname, s.setconfig FROM pg_db_role_setting s
JOIN pg_roles r ON r.oid = s.setrole WHERE r.rolname = 'analyst';
```


© 2026 SKAI Worldwide Co., Ltd.

```
 rolname |      setconfig
---------+----------------------
 analyst | {graph_path=moviekb}
```

A role that connects afterwards has it already:

```
SHOW graph_path;
```

```
 graph_path
------------
 moviekb
```

`ALTER DATABASE … RESET graph_path` and `ALTER ROLE … RESET graph_path` remove them again. A cluster-wide default in `postgresql.conf` works too, but it is rarely what you want. It makes every database in the cluster claim to have that graph.

The value is checked against the catalog, so `ALTER DATABASE db SET graph_path` has to be run while connected to `db`. The check looks for the graph in the current database. From anywhere else it fails:

```
ALTER DATABASE docs SET graph_path = moviekb;
```

```
ERROR:  invalid value for parameter "graph_path": "moviekb"
DETAIL:  graph "moviekb" does not exist.
```

allow_null_properties

```
bool · default off · context user
```

Decides whether a null property value is stored or dropped.

Off, assigning null removes the property. A property set to null and a property that was never there are then the same thing. On, the null is stored as a JSON null and the key stays.

```
CREATE GRAPH nulls;
SET graph_path = nulls;
CREATE VLABEL t;
CREATE (:t {name: 'a', extra: null});
SET allow_null_properties = on;
CREATE (:t {name: 'b', extra: null});
MATCH (n:t) RETURN n ORDER BY id(n);
```

```
                n
-----------------------------------
 t[3.1]{"name": "a"}
 t[3.2]{"name": "b", "extra": null}
```

Turn it on when you are loading data whose absent and null values mean different things and you need to keep the distinction. That is typically an import from a relational source.

The cost is that a stored null is a key that is present without a value. The tests disagree about it:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (n:t) RETURN exists(n.extra) AS exists, n.extra IS NULL AS is_null,
                   properties(n) ? 'extra' AS key_present;
```

```
 exists | is_null | key_present
--------+---------+-------------
 f      | t       | t
```

`REMOVE n.extra` removes the key whatever the setting says. It is still the way to get rid of a property.

```
RESET allow_null_properties;
DROP GRAPH nulls CASCADE;
SET graph_path = moviekb;
```

auto_gather_graphmeta

```
bool · default off · context superuser
```

Keeps `ag_graphmeta` up to date as a side effect of every graph write.

For each (start label, edge label, end label) triple in a graph, `ag_graphmeta` records how many edges there are. The planner uses it to prune scans: a `MATCH` that cannot possibly match a label is not scanned at all. With this setting off, the table is only as current as the last `regather_graphmeta()` call.

Turning it on triggers an immediate full gather, so the table is correct from that point:

```
SET auto_gather_graphmeta = on;
```

```
NOTICE:  regathering graphmeta...
SET
```

```
SELECT g.graphname, se.labname AS start, e.labname AS edge, ee.labname AS "end", m.edgecount
FROM pg_catalog.ag_graphmeta m
JOIN pg_catalog.ag_graph g ON g.oid = m.graph
JOIN pg_catalog.ag_label e  ON e.graphid  = m.graph AND e.labid  = m.edge
JOIN pg_catalog.ag_label se ON se.graphid = m.graph AND se.labid = m.start
JOIN pg_catalog.ag_label ee ON ee.graphid = m.graph AND ee.labid = m."end"
WHERE g.graphname = 'moviekb' ORDER BY 3;
```

```
 graphname | start  |    edge    |  end   | edgecount
-----------+--------+------------+--------+-----------
 moviekb   | person | acted_in   | movie |        25
 moviekb   | person | directed   | movie |        21
 moviekb   | movie  | related_to | movie |        11
```

```
RESET auto_gather_graphmeta;
```

Turn it on for a read-mostly graph, where accurate metadata is worth a little work on each write. Leave it off for a write-heavy load and call `regather_graphmeta()` afterwards. You must call it after `pg_upgrade` or a restore too. **Graph metadata** covers the table and the pruning it enables.

© 2026 SKAI Worldwide Co., Ltd.

cypher_allow_unsafe_ddl

```
bool · default off · context user
```

Skips two catalog checks that normally stand between you and a graph the catalog no longer describes. It exists to repair a graph, not to work in.

It lets `graph_path` name a graph that does not exist. The check moves from `SET` time to query time, where the message is different:

```
SET cypher_allow_unsafe_ddl = on;
SET graph_path = nosuchgraph;
SHOW graph_path;
```

```
 graph_path
-------------
 nosuchgraph
```

```
MATCH (n) RETURN count(n);
```

```
ERROR:  current graph_path "nosuchgraph" is invalid
HINT:  Use SET graph_path
```

```
RESET cypher_allow_unsafe_ddl;
SET graph_path = moviekb;
```

It lets a label be dropped as the wrong kind. `DROP VLABEL` on an edge label is normally refused. With the setting on it succeeds:

```
CREATE GRAPH unsafe;
SET graph_path = unsafe;
CREATE ELABEL e;
DROP VLABEL e;
```

```
ERROR:  DROP VLABEL cannot DROP edge label
```

```
SET cypher_allow_unsafe_ddl = on;
DROP VLABEL e;
```

```
DROP VLABEL
```

```
RESET cypher_allow_unsafe_ddl;
DROP GRAPH unsafe CASCADE;
SET graph_path = moviekb;
```

Both are useful in exactly one situation: a graph whose catalog and tables disagree, which you are trying to take apart. Set it for the statement that needs it and reset it immediately.

© 2026 SKAI Worldwide Co., Ltd.

eager_mem

```
integer · default 4096 · unit kB · context user · minimum 1024
```

How much memory one eager plan node may use before it spills to disk. It is `work_mem` for eager plans, and like `work_mem` it is per node, not per query or per connection.

```
SHOW eager_mem;
SET eager_mem = '64MB';
SHOW eager_mem;
```

```
 eager_mem
-----------
 4MB
(1 row)

SET
 eager_mem
-----------
 64MB
(1 row)
```

The minimum is 1 MB, and a smaller value is refused rather than clamped:

```
SET eager_mem = '512kB';
```

```
ERROR:  512 kB is outside the valid range for parameter "eager_mem" (1024 kB .. 2147483647 kB)
```

```
RESET eager_mem;
```

Raise it when a write that reads what it writes is spilling; see **`enable_eager`**. The default of 4 MB is the same conservative starting point as `work_mem`'s. A statement that materialises a large intermediate result will beat it.

enable_eager

```
bool · default on · context user
```

Allows the planner to build an eager plan: one that materialises a write's output before a later clause reads it.

Some statements need one. Take a `CREATE` followed by a `MERGE` that could match what the `CREATE` just made. The two cannot be interleaved row by row, because the second clause would see a half-written graph. So the planner puts the first clause's result in a buffer. `EXPLAIN` marks the node `eager`:

```
CREATE GRAPH eager;
SET graph_path = eager;
CREATE VLABEL src;
CREATE VLABEL dst;
CREATE (:src {n: 1}), (:src {n: 2});
EXPLAIN (COSTS OFF) MATCH (a:src) CREATE (:dst {n: a.n}) MERGE (:dst {n: a.n + 10});
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
------------------
 Graph Merge
   ->  Nested Loop CypherMerge Join
         Join Filter: ("<0000000000>".properties.'n' = ((_agens_default_s_1.a).properties.'n'
`+` '10'::jsonb))
         ->  Subquery Scan on _agens_default_s_1
               ->  Graph Create eager
                     ->  Seq Scan on src a
         ->  Seq Scan on dst "<0000000000>"
```

Turning it off does not make such a statement run a different way. It makes it fail:

```
SET enable_eager = off;
MATCH (a:src) CREATE (:dst {n: a.n}) MERGE (:dst {n: a.n + 10});
```

```
ERROR:  eagerness plan is not allowed.
```

```
RESET enable_eager;
DROP GRAPH eager CASCADE;
SET graph_path = moviekb;
```

So this is not a cost-based toggle like PostgreSQL's `enable_hashjoin`. Leave it on. Turn it off only to find out whether a statement depends on an eager plan, which matters when you are trying to make a large write cheaper.

enable_graph_ddl

```
bool · default off · context user
```

Allows SQL DDL to reshape a graph label.

A label's columns make it a graph label, and the graph catalog describes them. `ALTER TABLE` does not update that catalog. Reshaping a label with it can leave the graph describing something that is no longer true. Off, the attempts are refused, with a hint at the statement that does keep the catalog in step:

```
ALTER TABLE moviekb.movie ADD COLUMN note text;
```

```
ERROR:  cannot add a column to graph label "movie" with ALTER TABLE
HINT:  Use ALTER VLABEL or ALTER ELABEL instead, which keeps the graph catalog in step.
```

```
ALTER TABLE moviekb.movie DROP COLUMN properties;
```

```
ERROR:  cannot drop a column of graph label "movie" with ALTER TABLE
HINT:  Use ALTER VLABEL or ALTER ELABEL instead, which keeps the graph catalog in step.
```

```
ALTER TABLE moviekb.movie NO INHERIT moviekb.ag_vertex;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  cannot remove a parent label from graph label "movie" with ALTER TABLE
HINT:  A label's parents are fixed when it is created.
```

The full list of what is refused: adding, dropping or retyping a column; detaching or changing a column's generation expression; dropping a not-null constraint, named either as the column or as the constraint; adding or removing a parent label. The same setting also refuses renaming a label's column and moving a label to another schema.

Turning it on lifts all of those. It does not grant anything: whoever may alter the label still decides what happens to it. It only asks that the reshaping be deliberate.

Two things it never lifts. The type of `id`, `start`, `end` or `properties` cannot be changed at all, because every vertex and edge is made of those columns:

```
SET enable_graph_ddl = on;
ALTER TABLE moviekb.movie ALTER COLUMN properties TYPE text;
```

```
ERROR:  cannot change the type of column "properties" of graph label "movie"
DETAIL:  Every vertex and edge is made of that column.
```

```
RESET enable_graph_ddl;
```

And renaming or dropping the table itself stays refused, with its own message (`cannot rename table "movie"`, `cannot drop table … because vlabel … requires it`). Use `ALTER VLABEL … RENAME TO` and `DROP VLABEL`.

You will meet this setting whether or not you set it yourself. For a label that carries ordinary columns, `pg_dump` brackets the `ALTER TABLE` statements with `SET enable_graph_ddl = on`, so that a restore works. See **Server-side tools**.

enable_graph_dml

```
bool · default off · context superuser
```

Allows SQL `INSERT`, `UPDATE`, `DELETE` and `MERGE` against a graph label.

Reading a label with SQL needs no setting. Writing one does, because Cypher's write path keeps a graph consistent: endpoints that exist, an id that matches the label, and the metadata that goes with them. Off, every SQL write is refused:

```
INSERT INTO moviekb.movie (properties) VALUES ('{"title": "x"}');
```

```
ERROR:  DML query to graph objects is not allowed
```

```
UPDATE moviekb.movie SET properties = '{}';
```

```
ERROR:  DML query to graph objects is not allowed
```

```
DELETE FROM moviekb.movie;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  DML query to graph objects is not allowed
```

Turning it on gives up the guarantee that the graph means what it says. An `INSERT` into an edge label does not check that the endpoints exist, so it can create a dangling edge. Cypher cannot produce one:

```
CREATE GRAPH dangling;
SET graph_path = dangling;
CREATE VLABEL v;
CREATE ELABEL e;
CREATE (:v {n: 1});
SET enable_graph_dml = on;
INSERT INTO dangling.e (start, "end", properties)
VALUES ('3.1'::graphid, '99.99'::graphid, '{}');
RESET enable_graph_dml;
```

```
INSERT 0 1
```

The result is an edge that two different queries disagree about. A pattern that binds both endpoints finds nothing, because the join to the missing vertex fails:

```
MATCH (a)-[r:e]->(b) RETURN a, r, b;
```

```
 a | r | b
---+---+---
(0 rows)
```

A pattern that binds neither endpoint counts it, because nothing has to be joined:

```
MATCH ()-[r:e]->() RETURN count(r) AS edges;
```

```
 edges
-------
 1
```

```
DROP GRAPH dangling CASCADE;
SET graph_path = moviekb;
```

Turn it on for a bulk load you have written carefully, or to repair rows that Cypher cannot reach, and turn it off again. It is one of the two `superuser` settings. An ordinary role that tries `SET enable_graph_dml = on` gets `ERROR: permission denied to set parameter "enable_graph_dml"`, and cannot enable it for itself.

enable_graph_endpoint_elision

```
bool · default on · context user
```

Lets the planner drop the scan for a pattern endpoint that the query never uses.

© 2026 SKAI Worldwide Co., Ltd.

An anonymous endpoint such as `->()` contributes nothing but a join. If the endpoint carries no label and no property filter, that join cannot remove a row either. The planner removes it. It removes a named endpoint too when the name is only used to constrain a label. A label test on an edge's endpoint id is a `graphid` range check on the edge itself:

```
EXPLAIN (COSTS OFF) MATCH (p:person)-[r:acted_in]->() RETURN count(r);
```

```
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate
   ->  Seq Scan on acted_in r
         Filter: ((start >= '4.0'::graphid) AND (start <= '4.281474976710655'::graphid))
```

Without it, the same query joins three relations and scans the whole vertex hierarchy for the endpoint it will not look at:

```
SET enable_graph_endpoint_elision = off;
EXPLAIN (COSTS OFF) MATCH (p:person)-[r:acted_in]->() RETURN count(r);
```

```
                                              QUERY PLAN
-------------------------------------------------------------------------------------------------
-------
 Aggregate
   ->  Hash Join
         Hash Cond: (r."end" = "<0000000001>".id)
         ->  Hash Join
               Hash Cond: (r.start = p.id)
               ->  Seq Scan on acted_in r
                     Filter: ((start >= '4.0'::graphid) AND (start <= '4.281474976710655'::gra
phid))
               ->  Hash
                     ->  Seq Scan on person p
         ->  Hash
               ->  Append
                     ->  Seq Scan on ag_vertex "<0000000001>_1"
                     ->  Seq Scan on movie "<0000000001>_2"
                     ->  Seq Scan on person "<0000000001>_3"
```

```
RESET enable_graph_endpoint_elision;
```

This is the only one of the eleven that `EXPLAIN (SETTINGS)` reports, so a plan taken from a session that changed it says so. Turn it off only to compare plans.

enable_multiple_update

```
bool · default on · context user
```

Decides whether writing the same element more than once in one statement is silent or warned about.

A statement whose pattern matches an element several times writes it once per match. On, that happens quietly. Off, the server warns for each repeat. The write still happens, so this is a diagnostic rather than a restriction:

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE GRAPH repeats;
SET graph_path = repeats;
CREATE VLABEL t;
CREATE (:t {n: 0}), (:t {n: 0});
MATCH (a:t) MATCH (b:t) SET b.n = b.n + 1 RETURN count(b) AS rows;
MATCH (n:t) RETURN n ORDER BY id(n);
```

```
 rows
------
 4
```

```
       n
----------------
 t[3.1]{"n": 1}
 t[3.2]{"n": 1}
```

The cross join produced four rows and each vertex was reached twice, but `n` went up by one. With the setting off, the same statement says so:

```
SET enable_multiple_update = off;
MATCH (a:t) MATCH (b:t) SET b.n = b.n + 1 RETURN count(b) AS rows;
```

```
WARNING:  graph element(3,1) has been SET multiple times
WARNING:  graph element(3,2) has been SET multiple times
 rows
------
 4
```

```
RESET enable_multiple_update;
DROP GRAPH repeats CASCADE;
SET graph_path = moviekb;
```

Turn it off while you are working out why a `SET` did not do what you expected. A repeated write is usually a sign that the pattern matches more than you meant. The warning names the label id and local id, which is the element's `graphid`.

enable_property_promotion

```
bool · default on · context user
```

Decides whether a Cypher property read is answered from a promoted typed column when one exists.

A property can be materialised as a real column of a real type; see **Promoting a property to a column**. On, `n.key` reads that column and comes back in the column's type. Off, every read goes to the `jsonb` bag and comes back as `jsonb`:

```
CREATE GRAPH promoted;
SET graph_path = promoted;
CREATE VLABEL item (score int GENERATED);
CREATE (:item {score: 3}), (:item {score: 20}), (:item {score: 100});
MATCH (n:item) RETURN n.score AS score, pg_typeof(n.score) AS type ORDER BY n.score;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 score |   type
-------+----------
     3 | "integer"
    20 | "integer"
   100 | "integer"
```

```
SET enable_property_promotion = off;
MATCH (n:item) RETURN n.score AS score, pg_typeof(n.score) AS type ORDER BY n.score;
```

```
 score |  type
-------+--------
 3     | "jsonb"
 20    | "jsonb"
 100   | "jsonb"
```

WARNING

Unlike the cost-only planner toggles, turning this off can change query **results**. A resolved `n.key` uses the typed column's native semantics — numeric or collation ordering, and type-aware comparison — while the `jsonb` fallback uses `jsonb` ordering and comparison. A sort, a range predicate or a comparison across types can come out differently.

```
RESET enable_property_promotion;
DROP GRAPH promoted CASCADE;
SET graph_path = moviekb;
```

Leave it on. Turn it off to establish whether a promoted column changed a query's answer. Then turn it back on and fix the query or the column instead.

See also

- **Configuration** — where a setting can be set, and the PostgreSQL settings that matter for a graph workload
- **Syntax** — the lexical rules `graph_path` 's value follows
- **Data types** — what `allow_null_properties` and `enable_property_promotion` change about a property's type
- **Graph metadata** — the table `auto_gather_graphmeta` maintains
- **Server-side tools** — `pg_dump` , `pg_upgrade` and the settings they rely on
- **PostgreSQL: server configuration** — every other setting

The agens client

`agens` is the interactive terminal client. It is PostgreSQL's `psql` under an AgensGraph name, plus five meta-commands that describe a graph.

Because it is `psql` , **PostgreSQL's `psql` reference** documents every option, every variable and every other meta-command. None of that is repeated here. This page covers two things: the graph meta-commands, which appear nowhere else, and the two environment details that are AgensGraph's.

© 2026 SKAI Worldwide Co., Ltd.

The same program under two names

The build produces two binaries from the same source and installs both. The only difference is branding: the name each one prints, and the bug-report and home-page lines in its help.

```
agens --version
psql --version
```

```
agens (PostgreSQL) 18.6 (AgensGraph 2.18.6.1)
psql (PostgreSQL) 18.6 (AgensGraph 2.18.6.1)
```

Diffing the two help texts shows the whole of the difference:

```
diff <(agens --help) <(psql --help)
```

```
1c1
< agens is the AgensGraph interactive terminal.
---
> psql is the PostgreSQL interactive terminal.
4c4
<   agens [OPTION]... [DBNAME [USERNAME]]
---
>   psql [OPTION]... [DBNAME [USERNAME]]
62,63c62,63
< Report bugs to <inquiry@skaiworldwide.com>.
< AgensGraph home page: <https://skaiworldwide.com/>
---
> Report bugs to <pgsql-bugs@lists.postgresql.org>.
> PostgreSQL home page: <https://www.postgresql.org/>
```

Two consequences follow, and both save work.

Everything you know about `psql` applies. The same options, the same `~/.psqlrc`, the same `PG*` environment variables, the same `\copy`, `\watch`, `\gexec`, `\timing`, `\x`, `\pset`, tab completion and variable interpolation.

```
agens -d docs -c "SHOW agversion;"
```

```
 agversion
------------
 2.18.6.1
```

The graph meta-commands are in both binaries. The two are built from the same objects, so a `psql` from this installation lists graphs exactly as `agens` does. If you already have `psql` on your path you do not need to switch clients to describe a graph.

The server tools follow the same pattern, with one pair renamed:

AgensGraph name	PostgreSQL name	Difference
`agens`	`psql`	Branding only. Both installed.

© 2026 SKAI Worldwide Co., Ltd.

AgensGraph name	PostgreSQL name	Difference
`ag_ctl`	`pg_ctl`	Branding only. Both installed.

Every other tool keeps its PostgreSQL name and only that name. Those are `initdb`, `postgres`, `pg_dump`, `pg_dumpall`, `pg_restore`, `pg_basebackup`, `pg_upgrade`, `pg_controldata`, `pg_checksums`, `pg_rewind`, `pg_waldump`, `createdb`, `dropdb`, `createuser`, `dropuser`, `vacuumdb`, `reindexdb`, `clusterdb`, `pgbench`, `ecpg` and the rest. There is no `ag_dump` and no `ag_basebackup`. **Server-side tools** covers what they do with a graph.

Graph meta-commands

Five meta-commands describe a graph. `+` adds columns to each, and a pattern narrows what is listed.

Command	Lists
`\dG[+] [PATTERN]`	Graphs.
`\dGl[+] [PATTERN]`	Labels, both kinds.
`\dGv[+] [PATTERN]`	Vertex labels, then a description of each.
`\dGe[+] [PATTERN]`	Edge labels, then a description of each.
`\dGi[+] [PATTERN]`	Property indexes.

Case matters. `\dG` lists graphs. `\dg` is PostgreSQL's own command and lists roles.

These five are the complete set, and none of them appears in `\?`. The `x` and `S` modifiers that other `\d` commands take are not accepted directly after `\dG`:

```
\dGx moviekb
```

```
invalid command \dGx
```

`x` does work on the four sub-commands, so `\dGvx` prints the list in expanded mode.

Every example below runs against **the example dataset**.

List the graphs in a database

```
\dG moviekb
```

```
 List of graphs
  Name   | Owner
---------+-------
 moviekb | agens
```


© 2026 SKAI Worldwide Co., Ltd.

`+` adds the graph's access privileges and its comment. A graph is a schema, so the privileges are schema privileges. They are empty until something is granted, so grant one to see the column filled in:

```
GRANT USAGE ON SCHEMA moviekb TO PUBLIC;
```

```
\dG+ moviekb
```

```
                 List of graphs
  Name   | Owner | Access privileges | Description
---------+-------+-------------------+-------------
 moviekb | agens | agens=UC/agens   +|
         |       | =U/agens          |
```

Revoke it again, so the example dataset is left as it was:

```
REVOKE USAGE ON SCHEMA moviekb FROM PUBLIC;
```

With no pattern, `\dG` lists every graph in the database.

List a graph's labels

`\dGl` lists both kinds together, ordered by graph, then kind, then name:

```
\dGl moviekb.*
```

```
           List of labels
  Graph   |    Name    |  Type  | Owner
---------+------------+--------+-------
 moviekb | acted_in   | edge   | agens
 moviekb | ag_edge    | edge   | agens
 moviekb | directed   | edge   | agens
 moviekb | related_to | edge   | agens
 moviekb | ag_vertex  | vertex | agens
 moviekb | movie      | vertex | agens
 moviekb | person     | vertex | agens
```

`ag_vertex` and `ag_edge` are there because every graph has them. See **Label inheritance**.

`+` adds the label's on-disk size and its comment. The sizes are `pg_table_size` of the label's table, so they reflect whatever you have loaded:

```
\dGl+ moviekb.*
```

```
                                 List of labels
  Graph   |    Name    |  Type  | Owner |    Size    |            Description
---------+------------+--------+-------+------------+-----------------------------------
 moviekb | acted_in   | edge   | agens | 16 kB      |
 moviekb | ag_edge    | edge   | agens | 8192 bytes | base edge label of graph moviekb
 moviekb | directed   | edge   | agens | 16 kB      |
 moviekb | related_to | edge   | agens | 16 kB      |
 moviekb | ag_vertex  | vertex | agens | 8192 bytes | base vertex label of graph moviekb
 moviekb | movie      | vertex | agens | 16 kB      |
 moviekb | person     | vertex | agens | 16 kB      |
```


© 2026 SKAI Worldwide Co., Ltd.

Describe the vertex labels

`\dGv` narrows the list to vertex labels and then prints a block for each one, the way `\d` prints a table's footers. The block carries the label's property indexes, its constraints, the labels it inherits from and the labels that inherit from it:

```
\dGv moviekb.*
```

```
           List of labels
  Graph   |   Name    |  Type  | Owner
----------+-----------+--------+-------
 moviekb | ag_vertex | vertex | agens
 moviekb | movie     | vertex | agens
 moviekb | person    | vertex | agens

Vertex label "moviekb.ag_vertex"
--
Child labels: moviekb.movie,
              moviekb.person

Vertex label "moviekb.movie"
--
Inherits: moviekb.ag_vertex

Vertex label "moviekb.person"
--
Inherits: moviekb.ag_vertex
```

A label with an index and a constraint shows both:

```
CREATE GRAPH demo;
SET graph_path = demo;
CREATE VLABEL book;
CREATE PROPERTY INDEX ON book (title);
CREATE CONSTRAINT book_isbn_uq ON book ASSERT isbn IS UNIQUE;
```

```
\dGv demo.book
```

```
        List of labels
 Graph | Name |  Type  | Owner
-------+------+--------+-------
 demo  | book | vertex | agens

Vertex label "demo.book"
--
Property Indexes:
    "book_title_idx" btree (title)
Constraints:
    "book_isbn_uq" ASSERT (isbn) IS UNIQUE
Inherits: demo.ag_vertex
```

Note what the block does **not** show: the label's columns. To see `id`, `properties` and any promoted column, use `\d` on the table, since a label is a table:

```
\d moviekb.movie
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                                  Table "moviekb.movie"
   Column    |  Type   | Collation | Nullable |                                      Defaul
t
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------
 id          | graphid |           | not null | graphid(graph_labid('moviekb.movie'::cstring),
nextval('moviekb.movie_id_seq'::regclass))
 properties | jsonb   |           | not null | jsonb_build_object()
Indexes:
    "movie_pkey" PRIMARY KEY, btree (id)
Inherits: moviekb.ag_vertex
```

Describe the edge labels

`\dGe` is `\dGv` for edges, and its blocks have the same shape:

```
\dGe moviekb.acted_in
```

```
          List of labels
  Graph   |   Name    | Type | Owner
----------+-----------+------+-------
 moviekb | acted_in | edge | agens

Edge label "moviekb.acted_in"
--
Inherits: moviekb.ag_edge
```

`\dGl` is the one that does not print these blocks. It is the summary of both kinds, and nothing more.

List the property indexes

`\dGi` reads `ag_property_indexes` and shows each index's Cypher definition, the one you would type to recreate it. A graph with no property indexes lists nothing:

```
\dGi moviekb.*
```

```
No matching property indexes found.
```

```
\dGi demo.*
```

```
                                          List of property indexes
 Graph | LabelName |   IndexName    | Unique | Owner |                              Indexdef
-------+-----------+----------------+--------+-------
+-------------------------------------------------------------------
 demo  | book      | book_title_idx | f      | agens | CREATE PROPERTY INDEX book_title_idx ON
book USING btree (title)
```

`+` adds the index's size and its comment:

```
\dGi+ demo.*
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                                                List of property indexes
 Graph | LabelName |   IndexName    | Unique | Owner |                                Indexdef
                       |    Size    | Description
-------+-----------+----------------+--------+-------
+-------------------------------------------------------------------------+------------+-------------
 demo  | book      | book_title_idx | f      | agens | CREATE PROPERTY INDEX book_title_idx ON
book USING btree (title) | 8192 bytes |
```

A uniqueness constraint is not a property index and does not appear here. It shows up in the `\dGv` or `\dGe` block for its label, and as an exclusion constraint in `\d`.

```
DROP GRAPH demo CASCADE;
```

```
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to sequence demo.ag_label_seq
drop cascades to vlabel ag_vertex
drop cascades to elabel ag_edge
drop cascades to vlabel book
DROP GRAPH
```

Narrow a listing with a pattern

The pattern is PostgreSQL's `\d` pattern language, applied to two name parts separated by a dot. That language has `*` and `?` wildcards, character classes, and double quotes to defeat case folding.

Which two parts differ by command, and the difference catches people:

Command	first `.` second means
`\dG`	The graph name. There is no second part.
`\dGl`, `\dGv`, `\dGe`	Graph, then **label** name.
`\dGi`	Graph, then **index** name.

So `\dGi moviekb.movie` finds nothing, because `movie` is matched against index names:

```
\dGi moviekb.movie
```

```
No matching property indexes found.
```

Name the index instead, or use `moviekb.*` and read the `LabelName` column. A pattern that matches nothing says so, and it is not an error:

```
\dGv nosuchgraph.*
```

```
No matching labels found.
```


© 2026 SKAI Worldwide Co., Ltd.

AGDATA

`AGDATA` names a data directory, the way `PGDATA` does. Exactly three programs read it: `initdb`, `pg_ctl` and `ag_ctl`. No other tool does, not `agens`, not `psql`, not `pg_dump` and not `pg_basebackup`. So `PGDATA` is still the variable to set if you want one that everything honours.

Export it once and the three of them stop needing `-D`:

```
export AGDATA=/tmp/agensdata
ag_ctl status
```

```
ag_ctl: no server running
```

The precedence differs between the two programs that read both, so check it before you set both:

Program	Order
`initdb`	`-D`, then `AGDATA`, then `PGDATA`.
`pg_ctl` and `ag_ctl`	`-D`, then `PGDATA`, then `AGDATA`.

`initdb` also sets `AGDATA` in its own environment once it has decided, and its closing hint names `ag_ctl`:

```
env -u PGDATA AGDATA=/tmp/agensdata initdb -U agens --no-sync
```

```
Success. You can now start the database server using:

    ag_ctl -D /tmp/agensdata -l logfile start
```

With neither variable set and no `-D`, the message names `PGDATA` only. `AGDATA` would have worked just as well:

```
ag_ctl status
```

```
ag_ctl: no database directory specified and environment variable PGDATA unset
Try "ag_ctl --help" for more information.
```

Deployment covers where to put a data directory and how to start a server from an init system.

See also

- **Server-side tools** — `ag_ctl`, `pg_dump`, `pg_upgrade` and a graph
- **The example dataset** — the graph every listing above describes
- **System catalogs** — `ag_graph`, `ag_label` and `ag_property_indexes`, which these commands query
- **Server settings** — `graph_path`, which a session needs before it can run Cypher
- **PostgreSQL: psql** — every option and every other meta-command
- **PostgreSQL: pg_ctl** — what `ag_ctl` is

© 2026 SKAI Worldwide Co., Ltd.

Guides

Task-oriented how-tos for building on AgensGraph.

Loading data

There are three ways to get data into a graph, and the right one depends on how much data there is.

Route	Good for	Speed
Cypher `CREATE` / `MERGE`	Up to a few hundred thousand elements; anything where you want Cypher's checks	Slowest per row, safest
`LOAD FROM` a table	Data already in a table, or in a file exposed as a foreign table	Fast, and stays inside Cypher
`COPY` into the label tables	Millions of elements, from files you control	Fastest by a wide margin

Whichever you use, the section on **indexes during a bulk load** applies, and so does the **after the load** checklist. Skipping the second one is the usual reason a freshly loaded graph queries badly.

Cypher CREATE

The obvious route, and the one to use until it is too slow. Batch it: one statement creating many elements beats many statements creating one:

```
CREATE (:person {name: 'Ann', born: 1990}),
       (:person {name: 'Bo',  born: 1985}),
       (:person {name: 'Cy',  born: 1979});
```

Edges need both endpoints found first, so match them in the same statement:

```
MATCH (a:person {name: 'Ann'}), (b:person {name: 'Bo'})
CREATE (a)-[:knows {since: 2019}]->(b);
```

Loading edges one `MATCH` at a time is the reason this route is slow: each statement re-finds its endpoints. If you are loading edges in bulk, use one of the other two routes and let the server do the lookup as a join.

`MERGE` instead of `CREATE` when the load may run twice and you want it to be idempotent, but be aware it costs a lookup per element.

LOAD FROM a table

`LOAD FROM` reads rows from an ordinary table and hands them to a Cypher write clause, so the endpoint lookup becomes a single join rather than one query per row. It is the natural route when the data is already relational. That includes a file, once `file_fdw` has exposed it as a foreign table.

© 2026 SKAI Worldwide Co., Ltd.

Say the rows have already landed in two staging tables:

```
CREATE TABLE staging_people (name text, born int);
INSERT INTO staging_people VALUES ('Ann', 1990), ('Bo', 1985);

CREATE TABLE staging_knows (from_name text, to_name text, since int);
INSERT INTO staging_knows VALUES ('Ann', 'Bo', 2019);
```

Then one statement turns each row into a vertex:

```
LOAD FROM staging_people AS row
CREATE (:person {name: row.name, born: row.born});
```

Or take the whole row as the property map:

```
LOAD FROM staging_people AS row
CREATE (:person = to_jsonb(row_to_json(row)));
```

Edges are the case that matters, because the join is where the speed comes from:

```
LOAD FROM staging_knows AS row
MATCH (a:person {name: row.from_name}), (b:person {name: row.to_name})
CREATE (a)-[:knows {since: row.since}]->(b);
```

Index the properties you match on before running that; see below.

Reading a file as a table

`file_fdw` turns a file on the server's filesystem into a read-only table. It is a contrib module, so it needs `make install-world` at build time (see **Install AgensGraph**):

> NOTE
>
> The two blocks below need `file_fdw`, so they are the one part of this page you cannot run without having built the contrib modules.

```
CREATE EXTENSION file_fdw;
CREATE SERVER import FOREIGN DATA WRAPPER file_fdw;

CREATE FOREIGN TABLE staging_people (name text, born int)
SERVER import
OPTIONS (filename '/data/people.csv', format 'csv', header 'true');
```

Then `LOAD FROM staging_people AS row …` as above. The file is read on the server, so the path is a server path and the server's user must be able to read it.

If the file is on your machine rather than the server's, load it into an ordinary table with `\copy` in **the agens client** first, and `LOAD FROM` that.

COPY into the label tables

The fastest route, and the one to use for millions of elements. **A label is a table**, so you can `COPY` straight into it. You are writing the `properties` column directly and letting the `id` column default fill itself in.

Writing to a label with SQL is gated, so turn the gate on for the load:

© 2026 SKAI Worldwide Co., Ltd.

```
SET enable_graph_dml = on;

COPY moviekb.person (properties) FROM STDIN;
{"name": "Ann", "born": 1990}
{"name": "Bo", "born": 1985}
\.
```

For edges, supply the two endpoint ids as well, in `labid.locid` form:

```
SET enable_graph_dml = on;

COPY moviekb.acted_in (start, "end", properties) FROM STDIN;
4.1     3.1     {"role": "Neo"}
4.2     3.1     {"role": "Trinity"}
\.
```

So the work is in producing those ids. Two ways:

Build them arithmetically. If you control the numbering, `graphid()` composes an id from a label id and a local id:

```
SELECT graphid(graph_labid('moviekb.person'::cstring), 100::int8);
```

```
 made_id
---------
 4.100
```

So a loader that assigns row numbers can emit both vertex ids and the edge endpoints referring to them, with no lookup at all. This is the fastest possible load.

Or resolve them with a join. Load the vertices first, then build the edge rows from a staging table by joining on the business key:

```
CREATE TABLE staging_roles (person_name text, movie_title text, role text);
INSERT INTO staging_roles VALUES ('Keanu Reeves', 'The Matrix', 'Neo (again)');

SET enable_graph_dml = on;

INSERT INTO moviekb.acted_in (start, "end", properties)
SELECT p.id, m.id, jsonb_build_object('role', s.role)
FROM staging_roles s
  JOIN moviekb.person p ON p.properties->>'name'  = s.person_name
  JOIN moviekb.movie  m ON m.properties->>'title' = s.movie_title;
```

That is one set-based statement rather than a query per edge, which is the whole point. Index the keys you join on first.

```
DROP TABLE staging_roles;
```

> **WARNING**
>
> `COPY` and `INSERT` bypass the checks Cypher performs. Nothing stops you writing an edge whose `start` names a vertex that does not exist, and the result is a graph that no query can traverse correctly. Verify the endpoints before you load, and turn `enable_graph_dml` off again afterwards.

© 2026 SKAI Worldwide Co., Ltd.

Indexes during a bulk load

Maintaining an index while millions of rows arrive is expensive. A label can be created with its indexes switched off:

```
CREATE VLABEL bulk_person DISABLE INDEX;
```

or switched off later:

```
ALTER VLABEL bulk_person DISABLE INDEX;
```

Load the data, then rebuild:

```
REINDEX VLABEL bulk_person;
```

> NOTE
>
> There is no `ENABLE INDEX`. `ALTER VLABEL person ENABLE INDEX` is a syntax error; `REINDEX` is what brings the indexes back. Until you run it, the label's indexes exist but are not usable, so queries will fall back to sequential scans and any unique constraint is not being enforced.

Two more things that help a large load:

Create the property indexes after the data, not before — building an index once over a full label is much cheaper than maintaining it row by row. The same goes for **promoting a property to a column**: the `ALTER` rewrites the whole label, so promote once after the load rather than before it.

Consider an unlogged label for data you can reload. It skips the WAL entirely:

```
CREATE UNLOGGED VLABEL staging_person;
```

An unlogged label does not survive a crash and is not replicated. So this is for staging and for graphs you can rebuild, not for the final data.

Raise `maintenance_work_mem` before the reindex; it is used by one command at a time, so a large temporary value is safe. See **Configuration**.

After the load

Three steps, none optional.

Rebuild the indexes if you disabled them, as above.

Promote the properties you filter and sort on, if you are going to. `ALTER VLABEL person ADD COLUMN age int GENERATED` gives the property `age` a real `int` column beside the jsonb bag, filled at write time, and an index over it is a plain column index. The property key is what a query keeps naming — `p.age` still reads `age`, and the column name never appears in Cypher. Use `GENERATED (sourceKey)` when the column should be named differently from the key. See **Promoting a property to a column**.

Analyze. The planner has no statistics for a freshly loaded label, and a graph traversal plan is very sensitive to them:

```
ANALYZE moviekb.person;
ANALYZE moviekb.acted_in;
```

Gather the graph metadata. `ag_graphmeta` records which start-label, edge-label, end-label triples occur in the graph and how many edges make up each. That lets the planner prune scans of labels a pattern cannot reach:

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT regather_graphmeta();
```

```
SELECT graphname, graphmeta_valid FROM pg_catalog.ag_graph;
```

Then check the shape of what you loaded, which is also the quickest way to catch a load that went wrong:

```
SELECT * FROM pg_catalog.ag_graphmeta_view WHERE graphname = 'moviekb';
```

```
 graphname | start  |    edge    |  end   | edgecount
-----------+--------+------------+--------+-----------
 moviekb   | movie  | related_to | movie |        11
 moviekb   | person | directed   | movie |        21
 moviekb   | person | acted_in   | movie |        25
```

An edge shape you did not expect, or a count that is a multiple of what it should be, shows up here immediately.

Clean up the staging tables and the scratch label when the load is done:

```
DROP TABLE staging_people, staging_knows;
```

```
DROP VLABEL bulk_person;
DROP VLABEL IF EXISTS staging_person;
```

Exporting

`pg_dump` covers a graph, and naming the schema exports one graph. See **Backup and recovery**. For data rather than structure, `COPY … TO` works on a label table, and a Cypher query wrapped in SQL exports exactly the shape you want:

```
\copy (SELECT p.properties->>'name' AS name, m.properties->>'title' AS film
       FROM moviekb.acted_in r
         JOIN moviekb.person p ON p.id = r.start
         JOIN moviekb.movie  m ON m.id = r."end") TO 'roles.csv' CSV HEADER
```

Vector search

Overview

AgensGraph is built on PostgreSQL, so the **pgvector** extension adds vector similarity search to a graph directly. Embeddings then live next to your graph data, and one database can combine semantic (vector) search with Cypher graph traversal. That is the foundation for retrieval-augmented generation (RAG) and GraphRAG applications.

Vector search matches by **meaning** rather than by words: two texts can be close in vector space even when they share no vocabulary. That is its strength over keyword matching, which is **full-text search**.

Hybrid Search covers combining the two. The examples in all three pages use the same set of movies, so you can compare the behaviour directly.

An embedding is stored as a numeric array **property** on a vertex or edge. Properties are `jsonb`, so the array is kept as a JSON array. A `::vector(n)` cast turns it into pgvector's `vector` type wherever a vector operation needs one.

© 2026 SKAI Worldwide Co., Ltd.

That cast happens on every read. That is fine for a few thousand rows and wasteful at scale. From 2.18 you can **promote the property to a real `vector` column** instead.

The cast then happens once at write time, and an HNSW index can be built on the column itself. The queries on this page keep working unchanged and keep using the index, because a leftover `::vector(n)` on a promoted column is folded away.

Two things change when you promote.

The value's type. A promoted `d.emb` comes back as `vector`, where an unpromoted one comes back as `jsonb`, so a client that parses the JSON form needs updating.

The shape of the index. Over a promoted column it is an ordinary column index rather than an index over a cast expression, which makes it the form to prefer for an index you dump and restore.

Installing pgvector

pgvector is a separate extension that must be built and installed against your AgensGraph installation. Once it is available, enable it per database:

```
CREATE EXTENSION vector;
```

Example data

The examples use **the example dataset**: 18 films, each with a `plot` and a four-dimensional `embedding` of that plot. Load it before going further.

Real embeddings come from a model, such as `text-embedding-3-small` or a local `sentence-transformers` model, and they have hundreds of dimensions. These are reduced to four so the geometry stays readable on the page.

Similarity search

pgvector provides a set of distance operators and matching functions. Cast both operands to `vector(n)` and order by the distance.

Operator	Function	Distance	HNSW operator class
`<=>`	`cosine_distance`	cosine distance	`vector_cosine_ops`
`<->`	`l2_distance`	Euclidean (L2)	`vector_l2_ops`
`<#>`	`inner_product`	(negative) inner product	`vector_ip_ops`
`<+>`	`l1_distance`	taxicab (L1)	`vector_l1_ops`

Find the movies whose plots are closest to a query vector (cosine distance). Here the query is the embedding of The Matrix's plot, so the search returns the films most similar in meaning:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie)
RETURN m.title AS title
ORDER BY m.embedding::vector(4) <=> '[-0.07594558, 0.04081754, 0.29592122, -0.11921061]'::vect
or(4)
LIMIT 4;
```

```
           title
----------------------------
 "The Matrix"
 "The Matrix Reloaded"
 "The Matrix Resurrections"
 "Total Recall"
(4 rows)
```

The first three results are the obvious sequels, but the fourth is the interesting one. Total Recall's plot is "a construction worker discovers that his memories have been implanted". It shares **no keywords** with The Matrix's "a hacker … a simulated world controlled by machines".

Vector search still ranks it fourth, because both films are about a character discovering that their reality is fabricated. A keyword search could never connect the two.

Semantic search captures precisely that, and lexical search cannot. Compare the **Full Text Search** results on the same data, and see **Hybrid Search** for combining both.

> `ORDER BY` may reference the distance expression even though it is not in the `RETURN` list (AgensGraph 2.17 and later). Returning only `title` keeps the output clean and, more importantly, lets the HNSW index serve the ordering. Returning the raw distance would convert it to `jsonb` and prevent the index from being used.

The function form is convenient for comparing against another stored vector, such as "movies most similar to The Terminator". It saves writing the query vector by hand:

```
MATCH (m:movie), (q:movie {title: 'The Terminator'})
RETURN m.title AS title
ORDER BY m.embedding::vector(4) <=> q.embedding::vector(4)
LIMIT 4;
```

```
            title
----------------------------
 "The Terminator"
 "Terminator 2: Judgment Day"
 "Minority Report"
 "Blade Runner"
(4 rows)
```

Indexing with HNSW

Without an index, every similarity query scans all rows and computes the distance for each. For larger data sets, create an HNSW index so the planner can use an index scan. Because the indexed value is a cast expression, use a **property index**:

```
CREATE PROPERTY INDEX ON movie
    USING hnsw ((embedding::vector(4)) vector_cosine_ops);
```


© 2026 SKAI Worldwide Co., Ltd.

Match the operator class to the distance you query with (`vector_cosine_ops` for `<=>` , `vector_l2_ops` for `<->` , and so on). The earlier similarity query then uses the index:

```
EXPLAIN
MATCH (m:movie)
RETURN m.title AS title
ORDER BY m.embedding::vector(4) <=> '[-0.07594558, 0.04081754, 0.29592122, -0.11921061]'::vect
or(4)
LIMIT 4;
```

```
Limit
  ->  Index Scan using movie_embedding_idx on movie m
        Order By: ((properties.'embedding'::text)::vector(4) <=> '[...]'::vector(4))
```

> On this 18-row demo the planner may still prefer a sequential scan because it is cheaper for so few rows; add `SET enable_seqscan = off` to observe the index plan. On a real corpus the planner chooses the index automatically.
>
> HNSW property indexes over a `::vector(n)` cast require AgensGraph 2.17 or later.

Indexing a promoted column

The index above is built over a cast expression. Promoting the property removes the cast, so the index becomes an ordinary one over a column.

Promotion gives the property a real typed column beside the jsonb bag, filled at write time. The column name is what you are declaring; the property it mirrors is named by the same word:

```
DROP PROPERTY INDEX movie_embedding_idx;
ALTER VLABEL movie ADD COLUMN embedding vector(4) GENERATED;
```

`GENERATED` on its own mirrors the property that shares the column's name, here `embedding` . Write `GENERATED (sourceKey)` when the column should be called something else — `ALTER VLABEL movie ADD COLUMN emb vector(4) GENERATED (embedding)` stores the same property in a column named `emb` . Either way **the query keeps naming the property, never the column**: resolution is by property key, and the column name never appears in Cypher. **Promoting a property to a column** is the full treatment.

The index is then an ordinary one over a column, with no cast and no parentheses:

```
CREATE PROPERTY INDEX ON movie USING hnsw (embedding vector_cosine_ops);
```

Every query on this page still works, unchanged. A leftover `::vector(4)` on a promoted column is folded away, so `m.embedding` and `m.embedding::vector(4)` produce the same plan:

```
EXPLAIN (COSTS OFF)
MATCH (m:movie)
RETURN m.title AS title
ORDER BY m.embedding <=> '[-0.07594558, 0.04081754, 0.29592122, -0.11921061]'::vector(4)
LIMIT 4;
```


© 2026 SKAI Worldwide Co., Ltd.

```
Limit
  ->  Index Scan using movie_embedding_idx on movie m
        Order By: (embedding <=> '[-0.07594558,0.04081754,0.2959212,-0.11921061]'::vector(4))
```

IMPORTANT

Drop an expression index before promoting the key it reads, as above. A property index over an expression is built against the jsonb bag, and a query over a promoted key reads the column, so the two no longer meet. See **An expression property index is built over the bag**.

The value's type changes with promotion: `m.embedding` now comes back as `vector`, where the unpromoted property came back as `jsonb`.

GraphRAG

Vector search inside a graph database has a particular strength: a semantic match can be the **starting point** of a graph traversal.

A typical GraphRAG retrieval does two things in one query. It finds the items most relevant to a question by vector similarity, then expands along relationships to gather connected context for the language model.

The **example dataset** already carries `related_to` edges between films: franchise sequels and thematically linked titles. So the traversal below has something to follow.

At query time, embed the user's question with the same model used for the plots. Then run a single Cypher query that retrieves the closest movies and expands to their neighbors. Here the question embedding is The Matrix's vector:

```
MATCH (seed:movie)
WITH seed
ORDER BY seed.embedding::vector(4) <=> '[-0.07594558, 0.04081754, 0.29592122, -0.11921061]'::v
ector(4)
LIMIT 2
MATCH (seed)-[:related_to]-(ctx:movie)
RETURN seed.title AS seed, collect(DISTINCT ctx.title) AS related_context;
```

```
         seed          |                      related_context
-----------------------+-----------------------------------------------------------
 "The Matrix"          | ["The Matrix Reloaded", "The Terminator", "Total Recall"]
 "The Matrix Reloaded" | ["The Matrix", "The Matrix Revolutions"]
(2 rows)
```

The `WITH … ORDER BY … LIMIT 2` selects the two movies that best match the question by vector similarity. Those are the **seeds**. The second `MATCH` follows `related_to` edges from each seed to collect neighboring films.

The combined set is the seeds plus their graph neighbors. It forms a richer context than vector search alone, and it can be passed to a language model to generate the final answer.

This pattern generalizes. Seeds can be expanded over multiple hops, filtered by edge type or properties, or joined to other labels (people, studios, sources) to assemble exactly the context a RAG pipeline needs.

If you would rather a framework assembled it, **AI and LLM integrations** lists the libraries that do.

© 2026 SKAI Worldwide Co., Ltd.

Full-text search

Overview

AgensGraph is built on PostgreSQL and inherits its full-text search engine, so you can run linguistic keyword search directly over text stored in graph properties. Full-text search normalizes documents and queries into lexemes (stemming, stop-word removal, case folding). That makes it well suited to **lexical** retrieval: matching the words a user typed.

It is the natural complement to **vector search**, which matches by **meaning**. **Hybrid Search** covers combining the two. All three pages use the same movies, so you can compare the results directly.

A graph property is stored as `jsonb`, so a text property is cast to `text` (with `::text`) before being converted to a `tsvector`.

Example data

The examples use **the example dataset**: 18 films, each with a `plot` and a four-dimensional `embedding` of that plot. Load it before going further.

Real embeddings come from a model, such as `text-embedding-3-small` or a local `sentence-transformers` model, and they have hundreds of dimensions. These are reduced to four so the geometry stays readable on the page.

Searching a text property

Convert the property to a `tsvector` with `to_tsvector`, parse the user's input with one of the `*_to_tsquery` functions, and match them with the `@@` operator. `websearch_to_tsquery` accepts a forgiving, web-style syntax: quoted phrases, `OR`, and `-` to exclude a term. That makes it a good default for user input.

```
MATCH (m:movie)
WHERE to_tsvector('english', m.plot::text) @@ websearch_to_tsquery('english', 'machines')
RETURN m.title AS title;
```

```
           title
---------------------------
 "The Matrix"
 "The Matrix Reloaded"
 "The Matrix Revolutions"
 "The Matrix Resurrections"
(4 rows)
```

The query is matched against lexemes, not raw text. `machines` is reduced to the stem `machin`, so the singular `machine` would match exactly the same rows.

Look also at what is **not** returned. **Total Recall** and **The Terminator** are thematically about machine-dominated futures, but their plots never use the word, so keyword search cannot find them. That recall gap is exactly where **vector search** helps.

Other query parsers are available depending on the input you expect:

Function	Input style
`websearch_to_tsquery`	Web-style: `machines "virtual reality" -documentary`

© 2026 SKAI Worldwide Co., Ltd.

Function	Input style
`plainto_tsquery`	Plain text, all terms `AND` -ed together
`to_tsquery`	Explicit operators: `machines & reality`

Ranking results

`ts_rank` scores how well each document matches the query, so results can be ordered by relevance. A plot that matches more of the query terms ranks higher:

```
MATCH (m:movie)
WHERE to_tsvector('english', m.plot::text) @@ websearch_to_tsquery('english', 'machines OR
reality')
RETURN m.title AS title,
       ts_rank(to_tsvector('english', m.plot::text),
               websearch_to_tsquery('english', 'machines OR reality')) AS rank
ORDER BY rank DESC;
```

```
          title            |    rank
---------------------------+-------------
 "The Matrix"              |  0.06079271
 "The Matrix Reloaded"     | 0.030396355
 "The Matrix Revolutions"  | 0.030396355
 "The Matrix Resurrections" | 0.030396355
(4 rows)
```

The Matrix ranks above its sequels because its plot contains **both** `machines` and `reality`, while the sequels match only `machines`.

Exact-term matching

Full-text search is the right tool when the query is a specific term that must match exactly: a name, a code, a rare keyword. Asked for the film about dinosaurs, it returns precisely one:

```
MATCH (m:movie)
WHERE to_tsvector('english', m.plot::text) @@ websearch_to_tsquery('english', 'dinosaurs')
RETURN m.title AS title;
```

```
     title
----------------
 "Jurassic Park"
(1 row)
```

Vector search has no notion of the literal word `dinosaurs`. Asked for the films most similar to Jurassic Park, it returns The Avengers as the nearest neighbor. Both are large-scale spectacle blockbusters, a reasonable guess by meaning, but not what someone typing “dinosaurs” wants:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie), (q:movie {title: 'Jurassic Park'})
RETURN m.title AS title
ORDER BY m.embedding::vector(4) <=> q.embedding::vector(4)
LIMIT 3;
```

```
      title
-----------------
 "Jurassic Park"
 "The Avengers"
 "Interstellar"
(3 rows)
```

So the two methods have opposite blind spots. Full-text search is precise but misses synonyms and intent, while vector search captures meaning but drifts on exact terms. **Hybrid Search** runs both and fuses the results to get the strengths of each.

Indexing with GIN

Recomputing `to_tsvector` for every row on each search does not scale. Create a GIN index on the same `to_tsvector` expression so matches are found through the index. The indexed value is an expression over a property, so use a **property index** and wrap the expression in parentheses:

```
CREATE PROPERTY INDEX ON movie
    USING gin ((to_tsvector('english', plot::text)));
```

The search then uses the index instead of scanning every row:

```
EXPLAIN
MATCH (m:movie)
WHERE to_tsvector('english', m.plot::text) @@ websearch_to_tsquery('english', 'machines')
RETURN m.title;
```

```
 Bitmap Heap Scan on movie m
   Recheck Cond: (to_tsvector('english'::regconfig, (properties.'plot'::text)::text) @@ '''mac
hin'''::tsquery)
   ->  Bitmap Index Scan on movie_plot_idx
         Index Cond: (to_tsvector('english'::regconfig, (properties.'plot'::text)::text) @@ ''
'machin'''::tsquery)
```

The text search configuration used at query time (`'english'` above) must match the one in the index expression for the index to be used.

> On this small demo the planner may still prefer a sequential scan because it is cheaper for so few rows; add `SET enable_seqscan = off` to observe the index plan. On a real corpus the planner chooses the index automatically.
>
> GIN property indexes over a `to_tsvector(...)` expression require AgensGraph 2.17 or later.

© 2026 SKAI Worldwide Co., Ltd.

Indexing a promoted column

From 2.18 a property can be **promoted to a real typed column**, which removes the per-row `jsonb` read and the `::text` cast. `ALTER VLABEL movie ADD COLUMN plot text GENERATED` mirrors the property named by the column, here `plot`; `GENERATED (sourceKey)` mirrors a differently named key. Queries keep naming the property, never the column — resolution is by property key.

The index on this page needs rebuilding when you promote, because it is an **expression** over the property. A property index over an expression is built against the jsonb bag, and a query over a promoted key reads the column, so the two would no longer meet.

Write this one as SQL over the column. `\dGi` still lists it, and the query uses it:

```
DROP PROPERTY INDEX movie_plot_idx;
ALTER VLABEL movie ADD COLUMN plot text GENERATED;
```

```
CREATE INDEX movie_plot_idx ON moviekb.movie USING gin (to_tsvector('english', plot));
```

```
EXPLAIN (COSTS OFF)
MATCH (m:movie)
WHERE to_tsvector('english', m.plot) @@ websearch_to_tsquery('english', 'machines')
RETURN m.title;
```

```
 Bitmap Heap Scan on movie m
   Recheck Cond: (to_tsvector('english'::regconfig, plot) @@ '''machin'''::tsquery)
   ->  Bitmap Index Scan on movie_plot_idx
         Index Cond: (to_tsvector('english'::regconfig, plot) @@ '''machin'''::tsquery)
```

`m.plot` needs no `::text` now: the promoted column is already `text`. See **An expression property index is built over the bag**.

GraphRAG

Full-text search can also seed a graph traversal. Find the movies whose text best matches the question, then expand along relationships to gather connected context for a language model.

This is the lexical counterpart to the GraphRAG example in **Vector Search**. It shines when the question contains specific terms (names, codes, exact phrases) that keyword search handles better than embeddings.

The **example dataset** already carries `related_to` edges between films: franchise sequels and thematically linked titles. So the traversal below has something to follow.

A single Cypher query ranks the movies by full-text relevance to the question (here the term `resistance`), keeps the best matches, and expands them to their neighbors:

```
MATCH (seed:movie)
WHERE to_tsvector('english', seed.plot::text) @@ websearch_to_tsquery('english', 'resistance')
WITH seed, ts_rank(to_tsvector('english', seed.plot::text),
                   websearch_to_tsquery('english', 'resistance')) AS rank
ORDER BY rank DESC, seed.title
LIMIT 2
MATCH (seed)-[:related_to]-(ctx:movie)
RETURN seed.title AS seed, collect(DISTINCT ctx.title) AS related_context;
```


© 2026 SKAI Worldwide Co., Ltd.

```
             seed              |                related_context
-------------------------------+-----------------------------------------------
 "Terminator 2: Judgment Day" | ["The Terminator"]
 "The Terminator"             | ["Terminator 2: Judgment Day", "The Matrix"]
(2 rows)
```

The `WHERE` filter and `WITH … ORDER BY rank LIMIT` select the movies whose text best matches the query. Those are the seeds, here the two Terminator films. The second `MATCH` follows `related_to` edges to gather related titles.

The seeds plus their neighbors form the context passed to the model. Notice that the traversal reaches The Matrix. The keyword `resistance` never matched that film, but it is linked to the Terminator films by their shared war-against-machines theme. To combine this lexical retrieval with semantic (vector) retrieval before expanding, see Hybrid Search.

For the same retrieval through a framework rather than by hand, see **AI and LLM integrations**.

Hybrid search

Overview

Vector search matches by meaning. **Full-text search** matches by words. Each is strong where the other is weak.

Vector search finds paraphrases and related concepts, but it can miss an exact term such as a product code, a name or a rare keyword. Full-text search nails exact terms, but it misses synonyms and intent.

Hybrid search runs both and merges their results, so retrieval is semantically aware and lexically precise. AgensGraph provides both **pgvector** and PostgreSQL full-text search in one database, so a hybrid query is a single statement over the same graph.

> This page is about search-result fusion. It is unrelated to the **Combining SQL and Cypher** page, which is about mixing SQL and Cypher in one statement.

Reciprocal Rank Fusion

The two searches produce scores on different scales (cosine distance vs. `ts_rank`), so they cannot be added. **Reciprocal Rank Fusion (RRF)** avoids the problem by combining ranks rather than scores.

Each result contributes `1 / (k + rank)` from each search it appears in, and the contributions are summed:

```
score(d) = 1/(k + rank_vector(d)) + 1/(k + rank_fulltext(d))
```

`k` is a smoothing constant that limits how much the very top ranks dominate. 60 is the commonly used default. A document that ranks well in either search scores reasonably; a document that ranks well in both scores highest.

Example data

The examples use **the example dataset**: 18 films, each with a `plot` and a four-dimensional `embedding` of that plot. Load it before going further.

Real embeddings come from a model, such as `text-embedding-3-small` or a local `sentence-transformers` model, and they have hundreds of dimensions. These are reduced to four so the geometry stays readable on the page.

© 2026 SKAI Worldwide Co., Ltd.

The hybrid query

Reciprocal Rank Fusion needs a rank from each search, and SQL window functions produce one. So the fusion is a SQL query whose graph data comes from Cypher `MATCH` subqueries.

One subquery ranks the movies by vector distance and another by full-text relevance. `row_number()` turns each ordered result into a rank, and the outer query sums the reciprocal ranks. Each subquery returns only the vertex id and orders by its ranking expression, which it does not return. So the vector and full-text property indexes can serve the ordering and the filtering directly.

The query below looks for films in the vein of The Matrix that also literally mention `machines`. The vector query is that film's plot embedding:

```
WITH vec AS (
    SELECT id, row_number() OVER () AS rnk
    FROM (MATCH (m:movie)
          RETURN id(m) AS id
          ORDER BY m.embedding::vector(4) <=> '[-0.07594558, 0.04081754, 0.29592122,
-0.11921061]'::vector(4)
          LIMIT 10) v
),
fts AS (
    SELECT id, row_number() OVER () AS rnk
    FROM (MATCH (m:movie)
          WHERE to_tsvector('english', m.plot::text) @@ websearch_to_tsquery('english', 'machi
nes')
          RETURN id(m) AS id
          ORDER BY ts_rank(to_tsvector('english', m.plot::text),
                           websearch_to_tsquery('english', 'machines')) DESC) f
)
SELECT m.title AS title,
       round((COALESCE(1.0/(60 + vec.rnk), 0)
            + COALESCE(1.0/(60 + fts.rnk), 0))::numeric, 5) AS rrf_score,
       vec.rnk AS vec_rank,
       fts.rnk AS fts_rank
FROM (MATCH (m:movie) RETURN id(m) AS id, m.title AS title) m
LEFT JOIN vec ON vec.id = m.id
LEFT JOIN fts ON fts.id = m.id
WHERE vec.id IS NOT NULL OR fts.id IS NOT NULL
ORDER BY rrf_score DESC
LIMIT 6;
```

```
           title             | rrf_score | vec_rank | fts_rank
-----------------------------+-----------+----------+----------
 "The Matrix"                |   0.03279 |        1 |        1
 "The Matrix Reloaded"       |   0.03226 |        2 |        2
 "The Matrix Resurrections"  |   0.03150 |        3 |        4
 "The Matrix Revolutions"    |   0.03080 |        7 |        3
 "Total Recall"              |   0.01563 |        4 |
 "Avatar"                    |   0.01538 |        5 |
(6 rows)
```

Fusion does two things at once here. Total Recall (vector rank 4) never uses the word `machines`, so the full-text arm misses it entirely. It survives in the result anyway, because vector search recognized it as a false-reality story like The Matrix.

The Matrix Revolutions goes the other way. It is only the 7th-nearest vector, low enough to fall outside a vector-only top five, but the full-text arm ranks it 3rd because its plot literally says machines. Fusion pulls it back up.

The films that satisfy **both** arms (The Matrix, Reloaded, Resurrections) take the top spots. Vector search contributes recall, full-text search contributes precision, and RRF blends the two.

© 2026 SKAI Worldwide Co., Ltd.

Indexing both arms

Each arm of the query is an ordinary similarity or full-text search, so the same **property indexes** described in Vector Search and Full Text Search accelerate it. Those are an HNSW index over the embedding and a GIN index over the plain-text plot:

```
CREATE PROPERTY INDEX ON movie
    USING hnsw ((embedding::vector(4)) vector_cosine_ops);
CREATE PROPERTY INDEX ON movie
    USING gin ((to_tsvector('english', plot::text)));
```

Each Cypher subquery returns only the id and orders by its raw ranking expression, so the planner can serve each arm from its index. The HNSW index orders the vector candidates and the GIN index filters the full-text matches, so neither search scans the whole label.

The relevant parts of `EXPLAIN` for the fusion query are:

```
 ...
 ->  Subquery Scan on vec
       ->  Limit
             ->  Index Scan using movie_embedding_idx on movie m
                   Order By: ((properties.'embedding'::text)::vector(4) <=> '[...]'::vector(4)
)
 ...
 ->  Subquery Scan on fts
       ->  Bitmap Heap Scan on movie m
             Recheck Cond: (to_tsvector('english', (properties.'plot'::text)::text) @@
'...'::tsquery)
             ->  Bitmap Index Scan on movie_plot_idx
```

> On a tiny demo data set the planner may still choose a sequential scan because it is cheaper than an index for a handful of rows; add `SET enable_seqscan = off` to observe the index plan. On a real corpus the planner selects the indexes automatically.

Both indexes above are expressions over a property. If you **promote either property** to a typed column, rebuild that arm's index: an expression property index is built over the jsonb bag, while a query over a promoted key reads the column. The embedding arm then becomes a plain index over the column, and the full-text arm a SQL index over it:

```
ALTER VLABEL movie ADD COLUMN embedding vector(4) GENERATED;
CREATE PROPERTY INDEX ON movie USING hnsw (embedding vector_cosine_ops);
```

The fusion query itself does not change. **Vector search** and **full-text search** each show their side.

Tuning

- `k` **constant** — larger values flatten the contribution of the top ranks; smaller values let the first few results dominate. 60 is a reasonable starting point.
- **Weighting** — to favor one method, multiply its term, e.g. `1.5 * COALESCE(1.0/(60 + vec.rnk), 0)` to lean semantic.
- **Candidate depth** — the `LIMIT` inside each CTE controls how deep each search reaches before fusion; increase it for higher recall at some cost in latency.

© 2026 SKAI Worldwide Co., Ltd.

GraphRAG

Hybrid retrieval is the strongest starting point for a graph traversal. Rank the films by both semantic similarity and keyword relevance (RRF), then expand the top results along relationships to assemble context for a language model.

Reciprocal Rank Fusion needs SQL window functions, but every piece that touches the graph is written as a Cypher `MATCH` subquery in the `FROM` clause: each retrieval **and** the expansion. That is the **Combining SQL and Cypher** mechanism. The query stays graph-native and reuses the property indexes built above.

The **example dataset** already carries `related_to` edges between films: franchise sequels and thematically linked titles. So the traversal below has something to follow.

Here the vector arm retrieves films like The Matrix while the full-text arm matches the keyword `memories`. The SQL `row_number()` turns each ordered result into a rank. The `seeds` CTE fuses the two with RRF and keeps the top two, and a final Cypher traversal collects each seed's neighbors:

```
WITH vec AS (
    SELECT id, row_number() OVER () AS rnk
    FROM (MATCH (m:movie)
          RETURN id(m) AS id
          ORDER BY m.embedding::vector(4) <=> '[-0.07594558, 0.04081754, 0.29592122,
-0.11921061]'::vector(4)
          LIMIT 10) v
),
fts AS (
    SELECT id, row_number() OVER () AS rnk
    FROM (MATCH (m:movie)
          WHERE to_tsvector('english', m.plot::text) @@ websearch_to_tsquery('english', 'memor
ies')
          RETURN id(m) AS id
          ORDER BY ts_rank(to_tsvector('english', m.plot::text),
                           websearch_to_tsquery('english', 'memories')) DESC) f
),
seeds AS (
    SELECT m.id, m.title,
           COALESCE(1.0/(60 + vec.rnk), 0) + COALESCE(1.0/(60 + fts.rnk), 0) AS score
    FROM (MATCH (m:movie) RETURN id(m) AS id, m.title AS title) m
    LEFT JOIN vec ON vec.id = m.id
    LEFT JOIN fts ON fts.id = m.id
    WHERE vec.id IS NOT NULL OR fts.id IS NOT NULL
    ORDER BY score DESC
    LIMIT 2
)
SELECT s.title AS seed,
       jsonb_agg(DISTINCT ctx.title) AS expanded_context
FROM seeds s
LEFT JOIN (MATCH (a:movie)-[:related_to]-(b:movie)
           RETURN id(a) AS aid, b.title AS title) ctx ON ctx.aid = s.id
GROUP BY s.title
ORDER BY s.title;
```

```
      seed      |                   expanded_context
----------------+-----------------------------------------------------------
 "The Matrix"   | ["The Matrix Reloaded", "The Terminator", "Total Recall"]
 "Total Recall" | ["Inception", "The Matrix"]
(2 rows)
```

Fusion changes which films seed the traversal. Total Recall is only the 4th-nearest vector to The Matrix, so vector-only seeding would have expanded The Matrix and The Matrix Reloaded. But the keyword `memories` matches Total Recall exactly, and RRF lifts it into the top two.

© 2026 SKAI Worldwide Co., Ltd.

With Total Recall among the seeds, the traversal follows its `related_to` edge and reaches Inception: context that neither arm alone would have surfaced.

The result is hybrid-ranked seeds plus their graph neighborhood. It is the richest context of the three approaches, drawing on semantic similarity, exact keywords, and explicit relationships at once.

The frameworks in **AI and LLM integrations** build this same shape for you, and several expose it as one retriever.

AI and LLM integrations

Overview

AgensGraph is a common backing store for retrieval-augmented generation, because a knowledge graph and its embeddings can live in one database. The graph is tables, and a property is `jsonb`, so `pgvector` indexes an embedding stored on a vertex.

One query can therefore find the semantically closest documents and then walk the graph out from them. GraphRAG needs exactly that.

The integrations live in their own manual: **AgensGraph AI**. Each has its own installation and usage pages there; this page says what each one is for.

Every one of them runs on the **AgensGraph Python driver**, which is built on **psycopg 3** and runs Cypher through it. So a vertex, an edge or a path reaches the framework as an object rather than as text, whichever framework you pick. The 0.3.1 integrations require version 2.0 or later of the driver. See **Drivers and client libraries** for the driver on its own.

`pgvector` handles the embeddings and is not bundled, so you install it into the database once. See **Vector search**.

Source for every integration: the **agensgraph-ai repository**, which also ships runnable demos.

LangChain

A graph store, a pgvector-backed vector store, chat-message history, a LangGraph checkpointer, and an LLM graph transformer.

LangChain integration

LlamaIndex

`PropertyGraphIndex` and `VectorStoreIndex` stores, with hybrid search, metadata filtering and AgensGraph-tuned Text2Cypher.

LlamaIndex integration

LightRAG

All four LightRAG storage roles on one database, with workspace multi-tenancy.

LightRAG integration

cognee

Graph and vector adapters for the cognee AI-memory framework.

cognee integration

MCP servers

Three Model Context Protocol servers — Cypher, memory and data modeling — that expose AgensGraph to clients such as Claude Desktop.

MCP servers

© 2026 SKAI Worldwide Co., Ltd.

Without a framework

If you are building the retrieval yourself rather than through a framework, the three search guides are the place to start. **Vector**, **full-text** and **hybrid** search each end with a GraphRAG section that combines the search with a traversal in one statement.

© 2026 SKAI Worldwide Co., Ltd.

Drivers and client libraries

AgensGraph speaks the PostgreSQL wire protocol, so **any PostgreSQL driver connects to it and can run Cypher** — a Cypher statement is just a statement as far as the protocol is concerned. You do not need a graph-specific client to use AgensGraph.

What a driver cannot do without help is decode the graph types. A query that returns `m.title` hands back `jsonb`, which every driver understands; a query that returns `m` hands back a `vertex`, and `p` from a path pattern hands back a `graphpath`. A stock driver sees those as text. A driver written for AgensGraph adds the type handling, so a vertex arrives as an object with its label, id and properties.

That gives you two ways to work:

Use the PostgreSQL driver you already have. Return properties rather than whole elements, and wrap the Cypher in SQL where you want plain scalars instead of `jsonb` — see **Combining SQL and Cypher**. This is the least code and the fewest dependencies, and in any language other than Python it is the route to take.

Use the Python driver when you want vertices, edges and paths as native objects, or when you are building on one of the AI integrations.

The Python driver

AgensGraph for Python is the driver to build on. It adds the graph type handling on top of psycopg 3, and it is what every one of the **AI integrations** runs on. Its manual has installation and usage.

LangChain, LlamaIndex, LightRAG, cognee and the MCP servers all sit on that driver. See **AI and LLM integrations**.

Other languages

For Java, Node.js and Go, use a stock PostgreSQL driver — pgjdbc, node-postgres or pgx — and read properties rather than whole elements, as above. That is the combination to build on today.

Drivers for those three languages do exist, and each is kept here for the projects already using it rather than as a starting point:

- Java / JDBC legacy — a fork of pgjdbc, published as `net.bitnine:agensgraph-jdbc:1.4.3` in January 2023.
- Node.js legacy — wraps node-postgres, against `pg >= 6.1.2`. Published as `@skaiworldwide/ag-driver` 1.1.0 in September 2025.
- Go legacy — built on `lib/pq`, distributed as a Go module at tag `v0.1.1`.

Connection and parameters

Set `graph_path` on the connection. It is per-session, so a pooled connection needs it set each time it is handed out — or set it once for the role or the database:

```
ALTER ROLE appuser SET graph_path = moviekb;
```

The Python driver's `ConnectionPool` takes a `graph=` argument and sets it once on each connection it opens, not again on each handout, so a borrower that changes `graph_path` and commits passes the change on to the next borrower.

© 2026 SKAI Worldwide Co., Ltd.

Give Cypher parameters a type. A parameter whose type the driver leaves unspecified is parsed as JSON, so a plain string fails:

```
ERROR:  invalid input syntax for type json
DETAIL:  Token "The" is invalid.
CONTEXT:  JSON data, line 1: The...
```

Either declare the parameter's type through the driver's API, or cast it in the statement:

```
MATCH (m:movie {title: $1::text}) RETURN m.year;
```

The cast is the portable form, and it costs nothing. It matters most where a value could be read either way: without a type, `'123'` arrives as the number 123 and stops matching a property holding the string `"123"`.

The Python driver declares the type for you. From version 2.0 it sends a Python `str` as `text`, so a string parameter needs no cast there. The cast is still needed where the statement wants another type from a string, such as `%s::date`.

© 2026 SKAI Worldwide Co., Ltd.

Operations

Running AgensGraph in production: deployment, configuration, access control, backup and the server-side tools.

Deployment

What to decide before an AgensGraph server carries production traffic. The mechanics of getting the software onto a machine are in **Install AgensGraph**; this page is about the choices around it.

Platforms

AgensGraph builds and runs on Linux and on Windows, and on macOS for development. Most deployments are on Linux. The tested distributions are the current releases of Rocky Linux, RHEL, Debian and Ubuntu. The **container images** are built on Debian trixie and bookworm, Rocky 9 and 10, and Alpine.

Nothing about AgensGraph narrows PostgreSQL's platform support, so **PostgreSQL's supported-platforms notes** apply.

Sizing

Start from the data, not from a table of recommendations.

Memory is the lever that matters most. A graph traversal follows edges by index into scattered pages, so it is random-access work. Random access out of cache is slow. Aim for the working set to fit in the operating system's cache: the labels a query walks, plus their indexes. Size `shared_buffers` and `effective_cache_size` against that, as described in **Configuration**.

To see what you are dealing with:

```
SELECT l.labname,
       pg_size_pretty(pg_total_relation_size(l.relid)) AS total
FROM pg_catalog.ag_label l
  JOIN pg_catalog.ag_graph g ON g.oid = l.graphid
WHERE g.graphname = 'moviekb'
ORDER BY pg_total_relation_size(l.relid) DESC;
```

An edge label is usually the largest object in a graph, and it carries three indexes of its own, so budget for the indexes as well as the rows.

Storage should be flash. Not for throughput, but for random-read latency. That is where traversal spends its time. It also changes how you configure the planner — see `random_page_cost` in **Configuration**.

CPU matters less than you would expect for traversal, which is largely serial, and more than you would expect for analytical queries over whole labels, which parallelise.

A minimum for anything real is 4 GB of RAM. For disk, allow for the data, its indexes, the WAL, and room to build an index. A base install itself needs about 2.5 GB.

Before the first start

Run the server as its own unprivileged user. Not root, and not a login account shared with an application:

```
useradd --system --home /var/lib/agensgraph --shell /usr/sbin/nologin agens
```


© 2026 SKAI Worldwide Co., Ltd.

Put the data directory on its own filesystem if you can, so filling it cannot take the rest of the machine with it, and so you can snapshot it independently.

```
install -o agens -g agens -m 0700 -d /var/lib/agensgraph/18/data
sudo -u agens initdb -D /var/lib/agensgraph/18/data
```

`initdb` insists on mode 0700 or 0750 on the data directory. On PostgreSQL 18 it also enables data page checksums by default — worth keeping, and worth knowing about, because it decides a flag if you later upgrade into this cluster. See **Upgrading to 2.18**.

Kernel resources rarely need touching. AgensGraph, like PostgreSQL since 9.3, uses anonymous mmap for its main shared memory and only a tiny System V segment, so the old `SHMMAX` tuning is not needed. What can still bite is `vm.overcommit_memory` and the OOM killer on a memory-constrained host, and the open-file limit on a server with many connections. **PostgreSQL's kernel-resources section** is the reference.

Running under systemd

A unit for a single instance:

```
[Unit]
Description=AgensGraph 2.18
After=network-online.target
Wants=network-online.target

[Service]
Type=notify
User=agens
Group=agens
Environment=AGDATA=/var/lib/agensgraph/18/data
ExecStart=/usr/local/agensgraph/bin/postgres -D ${AGDATA}
ExecReload=/bin/kill -HUP $MAINPID
KillMode=mixed
KillSignal=SIGINT
TimeoutStopSec=120
OOMScoreAdjust=-900

[Install]
WantedBy=multi-user.target
```

`Type=notify` with `postgres` started directly is the arrangement systemd handles best — it lets systemd know when the server is actually accepting connections, and it keeps `systemctl stop` doing a clean shutdown. `OOMScoreAdjust` makes the kernel prefer to kill a backend rather than the postmaster.

Use `ag_ctl` for ad-hoc work, and systemd for the service. Mixing them — starting by hand and stopping with systemd — is how you end up with a stale postmaster.

Connections

`listen_addresses` is `localhost` after `initdb`. Widen it deliberately, and only together with the authentication rules below.

`max_connections` defaults to 100. Every connection costs memory, and a graph query can use several `work_mem` allocations at once, so raising this is not free. Put a connection pooler in front of an application that opens many short-lived connections; AgensGraph speaks the PostgreSQL protocol, so PgBouncer and Pgpool-II work unchanged.

Authentication is `pg_hba.conf`, and the default `trust` for local connections is not something to leave in place. See **Security**.

© 2026 SKAI Worldwide Co., Ltd.

Replication and high availability

Streaming replication, logical replication and synchronous commit all work on a graph, because a graph is tables and the WAL does not know the difference. So a standby carries graph data with no extra configuration, and read-only graph queries run on a hot standby.

Two AgensGraph-specific notes:

- Graph writes are not parallelised, so a write-heavy graph workload will not use more than one core per statement on the primary. This affects how you plan for write throughput, not how replication behaves.
- After promoting a standby, or after any physical restore, check `ag_graph.graphmeta_valid` and run `regather_graphmeta()` if it is false — see **Graph metadata**.

Patroni is the usual choice for automated failover and works with AgensGraph as it does with PostgreSQL.

A pre-production checklist

- `postgresql.conf` reviewed against **Configuration**, in particular `shared_buffers`, `work_mem`, `effective_cache_size` and `random_page_cost`.
- `pg_hba.conf` has no `trust` rule, and `listen_addresses` is no wider than it needs to be.
- Backups running, and a restore tested — including the graph counts, per **Backup and recovery**.
- WAL archiving on, if losing recent writes is not acceptable.
- `log_min_duration_statement` set, and `auto_explain` loaded, so a slow query leaves evidence.
- Monitoring on connection count, replication lag, cache hit ratio and disk headroom.
- `autovacuum` left on. An edge label churns, and turning autovacuum off is how a graph database ends up with a table twice the size of its data.
- `auto_gather_graphmeta` decided deliberately: on, and pay a little on every write for planner scan pruning, or off, and re-gather on a schedule.
- Statistics targets on edge labels checked, especially if the cluster was upgraded from 2.17 — see **Upgrading to 2.18**.

Configuration

AgensGraph's configuration is PostgreSQL's, plus eleven settings of its own. This page covers how to set things and which settings matter for a graph workload. For the settings AgensGraph adds, see **Server settings**. For the several hundred PostgreSQL settings this page does not list, see **PostgreSQL's server-configuration chapter** — it applies unchanged.

Where a setting can be set

The scopes go from widest to narrowest, and the narrowest one that applies wins.

How	Scope	Notes
`postgresql.conf`	The whole cluster	Edit the file, then `ag_ctl reload` — or restart, for a setting whose context is `postmaster`.

© 2026 SKAI Worldwide Co., Ltd.

How	Scope	Notes
`ALTER SYSTEM SET`	The whole cluster	Writes `postgresql.auto.conf`; still needs a reload or restart.
`ALTER DATABASE db SET`	One database	Takes effect on the next connection.
`ALTER ROLE r SET`	One role	Takes effect on the next connection.
`SET`	The current session	Reverts on disconnect. `SET LOCAL` reverts at the end of the transaction.

`graph_path` is the setting you will meet first, and it is per-session by nature. Set it for the database if every session wants the same graph:

```
ALTER DATABASE mydb SET graph_path = moviekb;
```

To see what a setting is now, and where the value came from:

```
SELECT name, setting, source, sourcefile, sourceline
FROM pg_settings WHERE name = 'work_mem';
```

`SHOW ALL` lists everything. `pg_settings` is the same information as a table you can filter, which is more useful — its `boot_val` column is the built-in default, `reset_val` is what a `RESET` would give you, and `context` tells you whether a change needs a restart.

NOTE

Read a setting's built-in default from `pg_settings.boot_val` rather than from the commented `postgresql.conf.sample` in the installation directory. `boot_val` is the value the server actually starts with.

Memory

These four decide most of a server's behaviour, and PostgreSQL's defaults are deliberately small enough to start anywhere.

`shared_buffers` — the server's own page cache. The default is 128 MB. On a dedicated machine, 25% of RAM is a reasonable starting point. Going far above 40% is usually counterproductive, because the operating system's cache is also holding those pages. Needs a restart.

`work_mem` — how much memory one sort or hash may use before it spills to disk. The default is 4 MB. It is per-operation, not per-connection: a graph query can hold several hashes at once, and a query with many joins can use a multiple of this value. So raise it in steps, and watch for the server running out of memory.

`maintenance_work_mem` — for `VACUUM`, `ANALYZE`, `CREATE INDEX` and `REINDEX`. Worth raising substantially and temporarily when you build indexes after a bulk load, because it is only used by one command at a time.

`effective_cache_size` — not an allocation, a hint. It tells the planner roughly how much memory the machine has for caching. That estimate is what makes an index scan look affordable. Set it to about 50-75% of RAM. Leaving it at the 4 GB default on a large machine is the most common reason a graph traversal chooses a sequential scan when an index scan would win.

© 2026 SKAI Worldwide Co., Ltd.

Planner settings that matter for traversal

`random_page_cost` — the planner's estimate of a random page read relative to a sequential one, default 4. That ratio describes a spinning disk. On SSD or NVMe the real ratio is far closer to 1, and a graph traversal is almost all random access: it follows edges by index into scattered heap pages. Lowering it to around 1.1 on flash storage is the single change most likely to make traversal plans better.

Do not take that on faith and do not push it to an extreme. Measure it: run your slowest traversal with `EXPLAIN (ANALYZE, BUFFERS)` at 4 and at 1.1 and compare, then keep the value that wins on your data. A value near zero tells the planner that random I/O is free, which is not true even in memory, and it will happily choose a plan that thrashes.

`max_parallel_workers_per_gather` — default 2. Graph writes do not run in parallel, and a traversal is largely serial, so most of the benefit is on the sequential and bitmap scans a `MATCH` lowers to. Raising it helps analytical queries over whole labels.

`jit` — on by default. It pays for itself on long-running analytical queries and costs compile time on short ones. If your workload is many small traversals, measure it off.

Asynchronous I/O on PostgreSQL 18

PostgreSQL 18 introduced asynchronous I/O, and graph queries get some of it for free. This is worth understanding because the win is a configuration change rather than a query change.

A fixed-length `MATCH` lowers to ordinary sequential, bitmap and index scans over the label tables, and PostgreSQL 18's read-ahead is used by **sequential scans and bitmap heap scans**. Those therefore benefit automatically. Two things do not: a plain index scan's heap fetches get no read-ahead in PostgreSQL 18, and the variable-length, shortest-path and Dijkstra engines walk one vertex at a time by design.

Two settings control it:

`effective_io_concurrency` — defaults to 16 in PostgreSQL 18, up from 1. It gates the whole mechanism, so if you carried a `postgresql.conf` forward from an older release and it pins this to 1, you have turned asynchronous I/O off.

`io_method` — `worker` by default, which performs the reads in background workers. `io_uring` is faster on Linux for cold scans, sometimes substantially, but it is only available if the server was built with liburing. Check before you try to set it:

```
SELECT enumvals FROM pg_settings WHERE name = 'io_method';
```

```
 enumvals
---------------
 {sync,worker}
```

That build has no `io_uring`; one built with it lists three values. Changing `io_method` needs a restart.

The benefit is largest when the data does not fit in cache, and near zero when it does. So test with a cold cache if you want to see it. Do not expect it to help a shortest-path query, which is bounded by serial probes rather than by I/O.

Logging

`log_statement` — `none` by default. Note the 2.18 change: `mod` now records Cypher writes. Before 2.18 a Cypher statement counted as loggable only at `all`, so a server asked to log data-modifying statements logged a SQL `INSERT` and silently skipped the `CREATE` beside it.

`log_min_duration_statement` — the setting to reach for when a graph workload is slow but you do not know which query. It logs any statement over the threshold, Cypher included.

`auto_explain` — a contrib module, and the most useful diagnostic on a graph workload, because a slow traversal usually means the plan changed rather than the data grew:

© 2026 SKAI Worldwide Co., Ltd.

```
shared_preload_libraries = 'auto_explain'
auto_explain.log_min_duration = '3s'
auto_explain.log_analyze = on
```

Settings AgensGraph adds

Eleven, covered in **Server settings**: `allow_null_properties`, `auto_gather_graphmeta`, `cypher_allow_unsafe_ddl`, `eager_mem`, `enable_eager`, `enable_graph_ddl`, `enable_graph_dml`, `enable_graph_endpoint_elision`, `enable_multiple_update`, `enable_property_promotion`, `graph_path`.

The ones with an operational bearing:

- **`graph_path`** — the graph a session works in. Nothing resolves without it.
- **`auto_gather_graphmeta`** — keeps `ag_graphmeta` current so the planner can prune scans. It costs something on every write, so it is off by default; see **Graph metadata** for the trade-off and for `regather_graphmeta()`.
- **`enable_graph_dml`** and **`enable_graph_ddl`** — off by default, and they gate writing to and reshaping a label with SQL. Leave them off outside of maintenance work.
- **`eager_mem`** — the memory an eager graph-write plan may use, alongside `work_mem`.

Where to go next

- **Deployment** — sizing, the data directory, and running under systemd.
- **Security** — authentication, roles, and permissions on a graph.
- **Query performance** — plan-shaped tuning, which is a different problem from this page.

Security

AgensGraph's security model is PostgreSQL's. Roles, `pg_hba.conf`, TLS, `GRANT` and row-level security all work exactly as they do there, and **PostgreSQL's client-authentication** and **database-roles** chapters are the reference.

What this page adds is how those mechanisms land on a graph. A graph is a schema and a label is a table, so the grants a Cypher query needs are not always the ones you would guess.

Authentication

`pg_hba.conf` in the data directory decides who may connect, from where, to which database, and how. Rules are matched top to bottom and the first match wins.

```
# TYPE  DATABASE  USER      ADDRESS          METHOD
local   all       all                        peer
host    mydb      appuser   10.0.0.0/24      scram-sha-256
hostssl mydb      analyst   0.0.0.0/0        scram-sha-256
host    all       all       0.0.0.0/0        reject
```

Reload after editing:

```
ag_ctl -D $AGDATA reload
```

Three things to get right before a server is reachable:

© 2026 SKAI Worldwide Co., Ltd.

Replace the default `trust` rules. `initdb` writes `trust` for local connections, which means anyone with a shell account can connect as any role, superuser included. Use `peer` for local Unix-socket connections and `scram-sha-256` for anything over the network.

Use `scram-sha-256`, not `md5`. Set `password_encryption = scram-sha-256` before you create or change passwords, so the stored verifier is the strong one.

Require TLS for network connections. `hostssl` in the rule, plus `ssl = on` and a certificate and key in `postgresql.conf`. `sslmode=verify-full` on the client is what actually prevents an interception; `require` only encrypts.

The methods available are PostgreSQL's full set — `peer`, `scram-sha-256`, `cert`, `ldap`, `gss`, `sspi`, `pam`, `radius`, `ident`, `trust`, `reject` — with the same options. See **PostgreSQL's authentication-methods chapter**.

When a connection is refused, the server log says why, and the message is more specific than what the client is told. Check it before changing rules.

Roles

Roles are cluster-wide — they belong to the server, not to a database — and a role with `LOGIN` is what other systems call a user.

```
CREATE ROLE appuser LOGIN PASSWORD 'secret';
CREATE ROLE readonly;                       -- a group, no LOGIN
GRANT readonly TO appuser;
```

The attributes that matter operationally are `SUPERUSER`, `CREATEDB`, `CREATEROLE`, `REPLICATION` and `BYPASSRLS`; grant none of them to an application role. `ALTER ROLE … SET` pins a setting for a role, which is a clean way to force `graph_path`, or to keep `enable_graph_dml` off for everyone but a maintenance role:

```
ALTER ROLE appuser SET graph_path = moviekb;
ALTER ROLE appuser SET enable_graph_dml = off;
```

Permissions on a graph

A graph is a schema and each label is a table, so the grants are ordinary ones. The surprises come from elsewhere: a Cypher query touches more objects than the labels it names.

Reading

A read needs `USAGE` on the graph's schema and `SELECT` on each label it touches. Follow this sequence to see where each grant becomes necessary. Start with the schema only:

```
CREATE ROLE analyst;
GRANT USAGE ON SCHEMA moviekb TO analyst;
```

```
SET ROLE analyst;
```

A Cypher read now fails as a table permission error, naming the label:

```
MATCH (m:movie) RETURN count(m);
```

```
ERROR:  permission denied for table movie
```


© 2026 SKAI Worldwide Co., Ltd.

Grant the label and it works:

```
RESET ROLE;
GRANT SELECT ON moviekb.movie TO analyst;
SET ROLE analyst;
```

```
MATCH (m:movie) RETURN count(m);
```

```
 count
-------
 18
```

An unlabelled pattern needs `ag_vertex` as well. `MATCH (n)` scans the graph's `ag_vertex` parent and every label under it, so a role that can read `movie` but not `ag_vertex` still cannot run it:

```
MATCH (n) RETURN count(n);
```

```
ERROR:  permission denied for table ag_vertex
```

The same applies to `ag_edge` for an unlabelled edge. So for a read-only analyst role, grant across the schema rather than label by label:

```
RESET ROLE;
```

```
GRANT SELECT ON ALL TABLES IN SCHEMA moviekb TO analyst;
ALTER DEFAULT PRIVILEGES IN SCHEMA moviekb GRANT SELECT ON TABLES TO analyst;
```

The `ALTER DEFAULT PRIVILEGES` line is what makes a label created next month readable too.

Writing

Creating a vertex or an edge needs `INSERT` on the label **and** `USAGE` **on the label's id sequence**. That second one is easy to miss. Grant only the `INSERT`:

```
RESET ROLE;
```

```
GRANT INSERT ON moviekb.movie TO analyst;
```

```
SET ROLE analyst;
```

```
CREATE (:movie {title: 'X'});
```

```
ERROR:  permission denied for sequence movie_id_seq
```

Add the sequence and the write goes through:

© 2026 SKAI Worldwide Co., Ltd.

```
RESET ROLE;
```

```
GRANT USAGE ON SEQUENCE moviekb.movie_id_seq TO analyst;
```

`SET` and `REMOVE` need `UPDATE` on the label. `DELETE` needs `DELETE` on it. And `DETACH DELETE` needs a grant on `ag_edge`, because that is how it finds the edges to remove:

```
ERROR:  permission denied for table ag_edge
```

An error naming `ag_edge` or `ag_vertex` is a missing grant on the graph's parent label, not a row-security result — worth knowing, because it is easy to read it as one.

A working set of grants for a role that reads and writes a whole graph:

```
GRANT USAGE ON SCHEMA moviekb TO appuser;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA moviekb TO appuser;
GRANT USAGE ON ALL SEQUENCES IN SCHEMA moviekb TO appuser;
ALTER DEFAULT PRIVILEGES IN SCHEMA moviekb
  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO appuser;
ALTER DEFAULT PRIVILEGES IN SCHEMA moviekb GRANT USAGE ON SEQUENCES TO appuser;
```

Creating labels

Creating a label needs `CREATE` on the graph's schema. Creating a graph needs `CREATE` on the database.

Row-level security

Row-level security works on a label, so a policy can restrict which vertices or edges a role sees and writes:

```
ALTER TABLE moviekb.movie ENABLE ROW LEVEL SECURITY;

CREATE POLICY movie_visible ON moviekb.movie
  FOR SELECT TO analyst
  USING ((properties->>'genre') = 'Sci-Fi');
```

> **WARNING**
>
> **Re-test any RLS policy you had on 2.17.** Before 2.18, policies were not properly enforced on graph writes: the `USING` clause of an `UPDATE` or `DELETE` policy was attached to a relation that is never scanned, so it filtered nothing — a role with no `DELETE` policy at all could delete rows. `INSERT` policies were checked against the wrong row, and edge creation was not checked at all, so an edge label with row security accepted every insert.
>
> 2.18 enforces all of it. That is the correct behaviour, and it means writes your application performs today may start being refused. Test before you upgrade, not after. See **Release notes**.

Two practical notes:

- A policy expression reads the label's columns, so it works on `properties` — and on a **promoted column**, which is faster and lets the policy use an index.
- `BYPASSRLS` and table ownership both skip policies. An application role should have neither.

© 2026 SKAI Worldwide Co., Ltd.

Two settings that are part of the security posture

`enable_graph_dml` and `enable_graph_ddl` are off by default, and they gate writing to and reshaping a label with SQL rather than Cypher. Leaving them off keeps an application on the Cypher path, where the graph's own checks run — for example that an edge's endpoints exist.

They are not the whole of a graph's integrity, and should not be read as it. `COPY` writes a label without consulting either setting. What does hold on every path, whatever the settings say, is that a row's id names the label that stores it: a write that would break that is refused, whether it comes from Cypher, `INSERT`, `UPDATE`, `COPY` or a replication apply.

Turn them on for maintenance work and turn them off again, or pin them off per role with `ALTER ROLE … SET`.

`cypher_allow_unsafe_ddl` likewise. See **Server settings**.

Encryption at rest

AgensGraph does not encrypt data files itself. Use filesystem or block-device encryption, or **Agens TDE** where transparent data encryption is a requirement. `pgcrypto` covers column-level encryption, and a property is an ordinary `jsonb` value, so it can hold a `pgcrypto` ciphertext like any other column.

Backup and recovery

A graph is stored as PostgreSQL schemas and tables, so every PostgreSQL backup method covers it, and covers it at the same point in time as your relational data. There is no separate graph backup to take and no separate restore to coordinate.

That means **PostgreSQL's backup chapter** is the reference, and this page covers the choices and the graph-specific details.

Which method

Method	What it gives you	When to use it
`pg_dump` / `pg_dumpall`	A logical copy: SQL statements, or an archive. Portable across major versions and platforms.	Moving a database, copying one graph, keeping a version-independent copy.
`pg_basebackup`	A physical copy of the whole cluster.	The base of a point-in-time-recovery setup, and for seeding a replica.
Continuous archiving	A base backup plus every WAL segment since, so you can recover to any moment.	Production, where losing an hour of writes is not acceptable.
A filesystem snapshot	A physical copy taken by the storage layer.	Only if the snapshot is atomic across every filesystem the cluster uses.

Copying the data directory with `cp` while the server is running is not a backup. Copying individual table files is never a backup — the commit log is cluster-wide, so a file taken on its own has no way to say which of its rows are committed.

© 2026 SKAI Worldwide Co., Ltd.

Logical backup

```
pg_dump -d mydb -Fc -f mydb.dump          # one database, archive format
pg_dumpall -f cluster.sql                  # every database plus roles
pg_dump -d mydb --schema=moviekb -f g.sql  # one graph
```

Restoring:

```
pg_restore --exit-on-error -d mydb mydb.dump
psql -d postgres -f cluster.sql
```

> **WARNING**
>
> Both tools carry on past an error by default, and `psql` exits 0 having done so, so a restore can report success with objects missing — a constraint the server refuses leaves the table without it, and rows that reference nothing are then accepted. `--exit-on-error` makes `pg_restore` stop at the first one.
>
> `psql` has `ON_ERROR_STOP`, but it cannot be used on a `pg_dumpall` file: the file re-creates the bootstrap superuser, which already exists, so the restore would stop on line 14 and load nothing. Read the output of that one, or dump each database separately with `pg_dump -Fc` and restore it with `pg_restore --exit-on-error`.

Naming the schema is how you dump a single graph, since a graph is a schema. The dump contains the `CREATE GRAPH`, `CREATE VLABEL` and `CREATE ELABEL` statements as well as the data, so a restore rebuilds the graph structure.

Four graph-specific points, all covered in more detail in **Server-side tools**:

- Neither tool checks the AgensGraph version, only PostgreSQL's.
- A dump of a label carrying ordinary columns needs `enable_graph_ddl` during the restore, and brackets its own statements with it.
- A 2.18 dump will not restore into 2.17 if any graph uses promoted columns.
- A view over a Cypher query is dumped as the SQL the Cypher lowered to.

After any restore, rebuild the graph metadata and the statistics:

```
SET graph_path = <your graph>;
SELECT regather_graphmeta();
```

```
vacuumdb -d mydb --analyze-in-stages
```

Physical backup and point-in-time recovery

The mechanics are PostgreSQL's, unchanged. In outline:

Set the server up for archiving, then restart:

```
wal_level = replica
archive_mode = on
archive_command = 'test ! -f /mnt/archive/%f && cp %p /mnt/archive/%f'
```

`wal_level = minimal` does not record enough to recover from an archive, so it rules this method out.

© 2026 SKAI Worldwide Co., Ltd.

Take a base backup:

```
pg_basebackup -D /mnt/backup/base -Ft -z -P -X stream
```

To recover, restore the base backup into an empty data directory, then tell the server where to fetch the archived WAL:

```
restore_command = 'cp /mnt/archive/%f %p'
recovery_target_time = '2026-09-02 14:30:00'
```

Those go in `postgresql.conf`, and an empty file named `recovery.signal` in the data directory is what puts the server into recovery. When recovery finishes, the server removes the signal file and starts accepting connections.

> NOTE
>
> If you are working from older AgensGraph or PostgreSQL notes, three things have moved. Recovery settings live in `postgresql.conf` with a `recovery.signal` file, not in a `recovery.conf` — that file was removed in PostgreSQL 12, and a server will refuse to start if it finds one. The WAL directory is `pg_wal`, not `pg_xlog`, and the commit log is `pg_xact`, not `pg_clog`; both were renamed in PostgreSQL 10.

Verify a base backup before you rely on it:

```
pg_verifybackup /mnt/backup/base
```

Higher-level tools

`pgBackRest` and `Barman` both manage retention, parallel compression, incremental backups and restore orchestration, and both work on an AgensGraph cluster with no graph-specific configuration. If you are running more than one server, either is a better answer than a hand-written `archive_command`.

What to check

A backup you have not restored is a hope, not a backup. For a graph specifically, a restore test should confirm more than that the server starts:

```
SET graph_path = <your graph>;
MATCH (n) RETURN label(n) AS label, count(*) ORDER BY label;
MATCH ()-[e]->() RETURN label(e) AS label, count(*) ORDER BY label;
```

```
SELECT graphname, graphmeta_valid FROM pg_catalog.ag_graph;
SELECT * FROM pg_catalog.ag_property_indexes;
```

Counts that match the source, plus the indexes and constraints being present, is what tells you the graph came back whole.

Server-side tools

The tools that come with AgensGraph are PostgreSQL's, so **PostgreSQL's reference for them** applies in full and is not repeated here. This page covers what is different, and what a graph adds to the ones you use most.

For the interactive client and its graph meta-commands, see **The agens client**.

© 2026 SKAI Worldwide Co., Ltd.

What AgensGraph adds to the tool set

Two tools ship under a second name:

AgensGraph name	PostgreSQL name	Difference
`agens`	`psql`	Branding only. Both are installed, and the graph meta-commands work in either.
`ag_ctl`	`pg_ctl`	Branding only. Both are installed.

```
agens --version
```

```
agens (PostgreSQL) 18.6 (AgensGraph 2.18.6.1)
```

NOTE

A build from a development branch reports a `-devel` suffix instead.

```
psql --version
```

```
psql (PostgreSQL) 18.6 (AgensGraph 2.18.6.1)
```

Everything else — `initdb`, `pg_dump`, `pg_dumpall`, `pg_restore`, `pg_upgrade`, `pg_basebackup`, `pg_controldata`, `pg_checksums`, `vacuumdb`, `reindexdb`, `createdb` and the rest — keeps its PostgreSQL name and its PostgreSQL behaviour.

`AGDATA` works alongside `PGDATA` as the default data directory, so `export AGDATA=/path/to/data` lets you drop `-D`. Exactly three binaries read it — `initdb`, `pg_ctl` and `ag_ctl` — and they disagree on precedence: `initdb` prefers `AGDATA` over `PGDATA`, while `pg_ctl` and `ag_ctl` prefer `PGDATA`. Every other tool takes `PGDATA` only.

Starting and stopping

```
ag_ctl -D /path/to/data -l /path/to/data/server.log start
ag_ctl -D /path/to/data status
ag_ctl -D /path/to/data reload
ag_ctl -D /path/to/data stop -m fast
```

Under systemd, use PostgreSQL's own unit conventions and point them at the AgensGraph binaries; there is nothing graph-specific about it. See **Deployment**.

Dumping a graph

`pg_dump` and `pg_dumpall` handle a graph without any special option, because a graph is a schema and its labels are tables. A dump therefore contains the `CREATE GRAPH`, `CREATE VLABEL` and `CREATE ELABEL` statements and the label data.

© 2026 SKAI Worldwide Co., Ltd.

```
pg_dump -d mydb -f mydb.sql
pg_dump -d mydb -Fc -f mydb.dump
```

To dump one graph, name its schema:

```
pg_dump -d mydb --schema=moviekb -f moviekb.sql
```

Four things to know that are specific to graphs:

- **Neither tool checks the AgensGraph version.** Only PostgreSQL's own `server_version` range check applies, so restoring a newer dump into an older server shows up as a parse failure partway through rather than as a refusal up front.
- **A dump of a label that carries ordinary columns brackets its `ALTER TABLE` statements with `SET enable_graph_ddl = on`.** That setting is writable by any role, so an ordinary user can restore it — unless the role has it forced off with `ALTER ROLE … SET`.
- **A 2.18 dump will not restore into 2.17** if any graph uses **promoted columns**: the emitted `CREATE VLABEL` form is not one a 2.17 parser accepts, and in a plain-text stream everything after that statement is lost.
- **A view defined over a Cypher query is dumped as the SQL that the Cypher lowered to**, not as your Cypher. It restores and works, but the definition you get back is not the definition you wrote. See **Combining SQL and Cypher**.

After a restore, rebuild the graph metadata and re-analyze — see **Upgrading to 2.18** for the exact commands, which apply to any restore, not just an upgrade.

Upgrading between major versions

`pg_upgrade` works on a graph with no special handling, with two AgensGraph-specific conditions:

- It refuses a graph whose label carries a column beyond `id`, `start`, `end` and `properties`, rather than corrupting the physical column layout.
- Afterwards, `ag_graphmeta` is empty and the graph is marked invalid, so the planner's scan pruning is off until you run `regather_graphmeta()`.

Upgrading to 2.18 is the full procedure, and **Upgrade walkthrough** is a worked example with real output.

Base backups and WAL

`pg_basebackup`, WAL archiving, `pg_receivewal`, `pg_verifybackup` and `pg_rewind` all treat a graph as what it is — schemas and tables in the same cluster — so a physical backup captures graph and relational data together and at the same point in time. See **Backup and recovery**.

© 2026 SKAI Worldwide Co., Ltd.

System Catalogs

The catalogs and views that describe a graph, and the metadata AgensGraph gathers about it.

Graph catalogs

AgensGraph adds seven tables to `pg_catalog`. Four of them describe the graph structure, and those are the ones you will query. Three are the shared parent relations that every label inherits from. PostgreSQL's own catalogs are unchanged and are documented in its catalogs chapter.

Catalog	What it holds
`ag_graph`	One row per graph.
`ag_label`	One row per label, vertex and edge alike.
`ag_label_property`	One row per property promoted to a typed column.
`ag_graphmeta`	Edge counts per (start label, edge label, end label) triple.
`ag_vertex`, `ag_edge`	The parent relation of every vertex or edge label in a graph. One pair exists inside each graph's schema, not in `pg_catalog`.
`ag_graphpath`	The composite that backs the `graphpath` type.

Two views present the same information in a readable form: see Graph views.

ag_graph

One row per graph. A graph is a schema, so `nspid` points at the schema that carries its labels.

Column	Type	Description
`oid`	`oid`	Row identifier.
`graphname`	`name`	The graph's name. Unique across the database.
`nspid`	`oid`	The `pg_namespace` row for the graph's schema.
`graphmeta_valid`	`boolean`	Whether `ag_graphmeta` is up to date for this graph. False means the planner cannot use it for scan pruning.

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT oid, graphname, nspid, graphmeta_valid FROM pg_catalog.ag_graph;
```

```
  oid   | graphname | nspid | graphmeta_valid
-------+-----------+-------+-----------------
 16386 | g         | 16385 | t
 16434 | moviekb   | 16433 | f
```

`graphmeta_valid` is set false when the metadata is known to be stale, after a binary upgrade for example. `SELECT regather_graphmeta();` rebuilds it and sets the flag. See **Graph metadata**.

ag_label

One row per label. Both vertex and edge labels live here, distinguished by `labkind`.

Column	Type	Description
`oid`	`oid`	Row identifier.
`labname`	`name`	The label's name. Unique within a graph.
`graphid`	`oid`	The `ag_graph` row this label belongs to.
`labid`	`integer`	The label's id within the graph, and the high 16 bits of every `graphid` in it.
`relid`	`oid`	The `pg_class` row for the label's table.
`labkind`	`"char"`	`v` for a vertex label, `e` for an edge label.

```
SELECT labname, labid, labkind FROM pg_catalog.ag_label
WHERE graphid = (SELECT oid FROM pg_catalog.ag_graph WHERE graphname = 'moviekb')
ORDER BY labid;
```

```
  labname   | labid | labkind
------------+-------+---------
 ag_vertex  |     1 | v
 ag_edge    |     2 | e
 movie      |     3 | v
 person     |     4 | v
 acted_in   |     5 | e
 directed   |     6 | e
 related_to |     7 | e
```

`labid` 1 and 2 are always the graph's `ag_vertex` and `ag_edge`, which are created with the graph. Because `labid` is 16 bits and 0 is not valid, a graph holds at most 65,535 labels. See **Limits**.

Joining `relid` to `pg_class` gets you from a label to its table. That join is what makes the ordinary PostgreSQL introspection work on a graph:

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT l.labname, pg_size_pretty(pg_total_relation_size(l.relid)) AS size
FROM pg_catalog.ag_label l
  JOIN pg_catalog.ag_graph g ON g.oid = l.graphid
WHERE g.graphname = 'moviekb' AND l.labkind = 'v'
ORDER BY 1;
```

ag_label_property

One row per property that has been **promoted to a typed column**. This catalog is new in 2.18.

Column	Type	Description
`laboid`	`oid`	The `ag_label` row the promoted column belongs to.
`propname`	`name`	The property key the column carries.
`attnum`	`smallint`	The column's attribute number in the label's table.
`semantics`	`"char"`	How the column relates to the property.

The primary key is `(laboid, propname)`, so a label promotes a given key at most once.

A property read resolves to a promoted column only when a row exists here. That mechanism is behind the promotion form that surprises people. `GENERATED ALWAYS AS (expr) STORED` creates a column but registers no row in this catalog. So the column is readable from SQL and invisible to a Cypher property read.

ag_graphmeta

One row per (start label, edge label, end label) triple that actually occurs in a graph, with how many edges make it up. The planner uses it two ways: to estimate a traversal, and to prune scans of labels that cannot participate in a pattern.

Column	Type	Description
`graph`	`oid`	The `ag_graph` row.
`edge`	`smallint`	The edge label's `labid`.
`start`	`smallint`	The start vertex label's `labid`.
`end`	`smallint`	The end vertex label's `labid`.
`edgecount`	`bigint`	How many edges of that shape exist.

Read it through **`ag_graphmeta_view`**, which resolves the label ids to names. The table is maintained only when `auto_gather_graphmeta` is on, and `regather_graphmeta()` rebuilds it from scratch. **Graph metadata** covers both.

© 2026 SKAI Worldwide Co., Ltd.

ag_vertex and ag_edge

These are not in `pg_catalog`. Every graph's schema contains its own pair, and every label you create inherits from one of them. That inheritance is what gives an unlabelled pattern its meaning. See **How a graph is stored**.

`<graph>.ag_vertex` has the vertex column pair, `(id graphid, properties jsonb)`. `<graph>.ag_edge` adds the endpoints, `(id, start, end, properties)`.

ag_graphpath

The composite type behind **`graphpath`**. It is an implementation detail. Build and read a path with the pattern syntax and the path functions rather than touching this directly.

Graph views

Two views in `pg_catalog` present the graph catalogs in a readable form, resolving oids and label ids to names. Both are ordinary views, so you can join and filter them.

ag_property_indexes

One row per property index, with the statement that would recreate it.

Column	Type	Description
`graphname`	`name`	The graph the index belongs to.
`labelname`	`name`	The label the index is on.
`indexname`	`name`	The index's name.
`tablespace`	`name`	The tablespace, or null for the default.
`unique`	`boolean`	Whether the index is unique.
`indexdef`	`text`	A `CREATE PROPERTY INDEX` statement that recreates it.
`owner`	`name`	The role that owns the index.
`size`	`text`	The index's size, already formatted.
`description`	`text`	The comment on the index, if any.

The **example dataset** has no property indexes, so create two to have something to look at:

```
CREATE PROPERTY INDEX ON movie (title);
CREATE PROPERTY INDEX ON person (name);
```


© 2026 SKAI Worldwide Co., Ltd.

```
SELECT labelname, indexname, indexdef, size
FROM pg_catalog.ag_property_indexes
WHERE graphname = 'moviekb' ORDER BY labelname, indexname;
```

```
 labelname |    indexname    |                              indexdef
|  size
-----------+-----------------
+-------------------------------------------------------------------------+-------
 movie     | movie_title_idx | CREATE PROPERTY INDEX movie_title_idx ON movie USING btree (tit
le) | 16 kB
 person    | person_name_idx | CREATE PROPERTY INDEX person_name_idx ON person USING btree (na
me) | 16 kB
```

Drop them again with `DROP PROPERTY INDEX movie_title_idx;` when you are done.

The `indexdef` column is the useful one. It gives back the property-level statement rather than the expression index the server built, and that is the form you want when scripting a schema. `\dGi` in **the agens client** prints the same information.

The view lists indexes on **promoted columns** as well as indexes over the property bag. So it is the single place to look for "what is indexed on this label".

ag_graphmeta_view

One row per edge shape that occurs in a graph, with names instead of label ids.

Column	Type	Description
`graphname`	`name`	The graph.
`start`	`name`	The label of the edge's start vertex.
`edge`	`name`	The edge label.
`end`	`name`	The label of the edge's end vertex.
`edgecount`	`bigint`	How many edges of that shape exist.

```
SELECT * FROM pg_catalog.ag_graphmeta_view WHERE graphname = 'moviekb';
```

```
 graphname | start  |    edge    |  end   | edgecount
-----------+--------+------------+--------+-----------
 moviekb   | movie  | related_to | movie  |        11
 moviekb   | person | directed   | movie  |        21
 moviekb   | person | acted_in   | movie  |        25
```

This is the clearest picture of a graph's actual shape that AgensGraph can give you. Read it before tuning a query: it tells you which traversals exist and how big each one is. The planner uses the same rows.

© 2026 SKAI Worldwide Co., Ltd.

The view is empty unless the metadata has been gathered. See **Graph metadata** for `auto_gather_graphmeta` and `regather_graphmeta()`, and check `ag_graph.graphmeta_valid` if the numbers look wrong.

Graph metadata

Introduction

The meta extension is a collection of helper functions for AgensGraph that simplify working with graph metadata. Its functions explore graph structures, validate labels, estimate sizes, analyze relationships, and handle Cypher types.

They let you inspect, monitor, and manage graphs in AgensGraph with less repetition and less effort.

Installation

The meta extension is included in the contrib directory of AgensGraph. To install it, first build and install AgensGraph with the contrib modules enabled.

Once it is installed, you can add the extension to any database with:

```
CREATE EXTENSION meta;
```

That makes all meta functions available in the database.

Functions

The meta extension includes the following functions:

meta.graphs

Description

Returns the list of all graphs available.

Arguments

- None

Returns

- Set of graph names

Example

```
SELECT * FROM meta.graphs();

 graphname
-----------
 social
 company
 test
```

meta.labels

Description

Returns all labels for a given graph. The system labels `ag_vertex` and `ag_edge` are not included in the result. If the `graph` argument is not provided, the function uses the current `graph_path` setting.

© 2026 SKAI Worldwide Co., Ltd.

Arguments

- `graph` (name, optional): Name of the graph. Defaults to the current `graph_path`.

Returns

- `label_name` (name): Names of all labels in the specified graph, excluding system labels `ag_vertex` and `ag_edge`.

Example

```
SELECT * FROM meta.labels('social');

 label_name
------------
 Person
 Product
 Transaction
```

meta.vertex_labels

Description

Returns all vertex labels defined in a given graph. The system label `ag_vertex` is excluded. If the `graph` argument is not provided, the function uses the current `graph_path` setting to determine the graph.

Arguments

- `graph` (name, optional): Name of the graph. Defaults to the current `graph_path`.

Returns

- `label_name` (name): Names of all vertex labels in the specified graph.

Example

```
SELECT * FROM meta.vertex_labels('social');

 label_name
------------
 Person
 Product
 Comment
```

meta.edge_labels

Description Returns all edge labels defined in a given graph. The system label ag_edge is excluded. If the graph argument is not provided, the function uses the current graph_path setting to determine the graph.

Arguments

- `graph` (name, optional): Name of the graph. Defaults to the current `graph_path`.

Returns

- `label_name` (name): Names of all edge labels in the specified graph.

Example

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT * FROM meta.edge_labels('social');

 label_name
------------
 Follows
 Likes
 Purchases
```

meta.label_id

Description

Returns the internal id of a specific label in a given graph. If the `graph` argument is not provided, the function uses the current `graph_path` setting. Raises an exception if the label does not exist in the graph.

Arguments

- `label` (name, required): the name of the label whose id is to be retrieved.
- `graph` (name, optional): name of the graph. defaults to the current `graph_path`.

Returns

- `labid` (int): internal id of the specified label.

Example

```
SELECT meta.label_id('Person', 'social');

 labid
-------
     5
```

meta.label_kind

Description

Returns the kind of a label (`v` for vertex, `e` for edge) in a given graph. If the `graph` argument is not provided, the function uses the current `graph_path` setting. Returns `NULL` if the label does not exist.

Arguments

- `label` (name, required): the name of the label whose kind is to be retrieved.
- `graph` (name, optional): name of the graph. defaults to the current `graph_path`.

Returns

- `kind` (char): single character representing the label kind (`v` or `e`).

Example

```
SELECT meta.label_kind('Person', 'social');

 kind
------
 v
```


© 2026 SKAI Worldwide Co., Ltd.

meta.label_name

Description

Returns the name of a label given its ID (`labid`) or OID in a specific graph. If the `graph` argument is not provided, the function uses the current `graph_path` setting. Raises a notice if the label does not exist.

Arguments

- `labelid` (int, required): the ID or OID of the label.
- `graph` (name, optional): name of the graph. defaults to the current `graph_path` .

Returns

- `label_name` (text): the name of the label.

Example

```
SELECT meta.label_name(5, 'social');

 label_name
------------
 Person
```

meta.graph_size

Description

Returns the total size of a graph, including all its labels. If the `graph_name` argument is not provided, the function uses the current `graph_path` setting. The size can be returned in a human-readable format or as raw bytes.

Arguments

- `graph_name` (name, optional): the name of the graph. defaults to the current `graph_path` .
- `pretty` (boolean, optional): if `true` (default), returns a human-readable size (e.g., `42 MB`). if `false` , returns the size in bytes as text.

Returns

- `size` (text): total size of the graph.

Examples

```
SELECT meta.graph_size('social');

 graph_size
------------
 123 MB

SELECT meta.graph_size('social', false);

 graph_size
------------
 129394816
```


© 2026 SKAI Worldwide Co., Ltd.

meta.label_size

Description

Returns the total size of a specific label, including its indexes. If the `graph` argument is not provided, the function uses the current `graph_path` setting. The size can be returned in a human-readable format or as raw bytes.

Arguments

- `labelname` (text): the name of the label.
- `graph` (name, optional): the graph that contains the label. defaults to the current `graph_path`.
- `pretty` (boolean, optional): if `true` (default), returns a human-readable size (e.g., `42 MB`). if `false`, returns the size in bytes as text.

Returns

- `size` (text): total size of the label including indexes.

Examples

```
SELECT meta.label_size('person');

 label_size
------------
 42 MB

SELECT meta.label_size('person', pretty => false);

 label_size
------------
 44040192
```

meta.count

Description

Returns the exact number of nodes or edges for the specified label. If the `graph` argument is not provided, the function uses the current `graph_path` setting.

Arguments

- `labelname` (text): the label whose elements are being counted.
- `graph` (name, optional): the graph that contains the label. defaults to the current `graph_path`.

Returns

- `count` (bigint): the exact number of elements (nodes or edges) in the label.

Example

```
-- Count all nodes in the 'person' label
SELECT meta.count('person');

-- Count all edges in the 'friendship' label of the 'social' graph
SELECT meta.count('friendship', 'social');
```


© 2026 SKAI Worldwide Co., Ltd.

meta.estimated_count

Description

Returns an estimated number of elements (nodes or edges) for a given label based on PostgreSQL statistics (`pg_class.reltuples`). If the `graph` argument is not provided, the function uses the current `graph_path` setting. This is faster than `meta.count` but may be inaccurate.

> Note: If statistics are not available, the function may return NULL. Run `ANALYZE "graph"."label"` to gather statistics.

Arguments

- `labelname` (text): the label whose estimated element count is returned.
- `graph` (name, optional): the graph containing the label. defaults to the current `graph_path` .

Returns

- `estimated_count` (bigint): estimated number of elements in the label.

Example

```
-- Get estimated count of nodes in the 'person' label
SELECT meta.estimated_count('person');

-- Get estimated count of edges in the 'friendship' label of the 'social' graph
SELECT meta.estimated_count('friendship', 'social');
```

meta.graph_exists

Description

Checks whether a graph with the specified name exists.

Arguments

- `graph` (name): Name of the graph to check.

Returns

- `boolean` : `true` if the graph exists, `false` otherwise.

Example

```
SELECT meta.graph_exists('social');

 ?column?
----------
 t
```

meta.label_exists

Description

Checks whether a label exists in the specified graph. If the graph is not provided, it uses the current `graph_path` setting.

Arguments

- `labelname` (text): Name of the label to check.

© 2026 SKAI Worldwide Co., Ltd.

- `graph` (name, optional): Name of the graph. Defaults to `graph_path` if not provided.

Returns

- `boolean` : `true` if the label exists, `false` otherwise.

Example

```
SELECT meta.label_exists('Person', 'social');

 ?column?
----------
 t
```

meta.vertex_label_exists

Description

Checks whether a vertex label exists in the specified graph. If the graph is not provided, it uses the current `graph_path` setting.

Arguments

- `labelname` (text): Name of the vertex label to check.
- `graph` (name, optional): Name of the graph. Defaults to `graph_path` if not provided.

Returns

- `boolean` : `true` if the vertex label exists, `false` otherwise.

Example

```
SELECT meta.vertex_label_exists('Person', 'social');

 ?column?
----------
 t
```

meta.edge_label_exists

Description

Checks whether an edge label exists in the specified graph. If the graph is not provided, it uses the current `graph_path` setting.

Arguments

- `labelname` (text): Name of the edge label to check.
- `graph` (name, optional): Name of the graph. Defaults to `graph_path` if not provided.

Returns

- `boolean` : `true` if the edge label exists, `false` otherwise.

Example

```
SELECT meta.edge_label_exists('Follows', 'social');
```


© 2026 SKAI Worldwide Co., Ltd.

```
 ?column?
----------
 t
```

meta.get_child_labels

Description

Returns all child labels that inherit from the specified parent label. If the graph is not provided, it uses the current `graph_path` setting.

Arguments

- `parent_label` (text): Name of the parent label.
- `graph` (name, optional): Name of the graph. Defaults to `graph_path` if not provided.

Returns

- `table(labelname name)` : A set of child label names.

Example

```
SELECT * FROM meta.get_child_labels('Person', 'social');

 labelname
-----------
 Employee
 Customer
```

meta.get_parent_labels

Description

Returns all parent labels from which the specified child label inherits. If the graph is not provided, it uses the current `graph_path` setting.

Arguments

- `child_label` (text): Name of the child label.
- `graph` (name, optional): Name of the graph. Defaults to `graph_path` if not provided.

Returns

- `table(labelname name)` : A set of parent label names.

Example

```
SELECT * FROM meta.get_parent_labels('Employee', 'social');

 labelname
-----------
 Person
```

meta.cypher_type

Description

Determines the corresponding Cypher type of a given value. Supports standard types, JSON/JSONB, and numeric distinctions.

© 2026 SKAI Worldwide Co., Ltd.

Arguments

- `element` (anyelement): The value whose Cypher type should be determined.

Returns

- `text` : The Cypher type as a string. Possible values include `INTEGER` , `FLOAT` , `STRING` , `BOOLEAN` , `MAP` , `LIST` , or `NUMBER` .

Example

```
SELECT meta.cypher_type(42) AS result;
result
-------
INTEGER

SELECT meta.cypher_type('{"key": "value"}'::jsonb) AS result;
result
-------
MAP
```

meta.cypher_types

Description

Returns a JSONB object mapping each key of the input JSONB object to its corresponding Cypher type. Raises an error if the input is not a JSON object.

Arguments

- `properties` (jsonb): A JSONB object whose keys will be mapped to Cypher types.

Returns

- `jsonb` : A JSONB object where each key corresponds to the input keys and the values are the detected Cypher types (`INTEGER` , `FLOAT` , `STRING` , `BOOLEAN` , `MAP` , `LIST` , `NULL` or `NUMBER`).

Example

```
SELECT meta.cypher_types('{"key": "value", "arr": [1, 2, 3], "num": 123, "float": 45.67,
"bool": true, "null": null}'::jsonb);

cypher_types
-------------------------------------------------------------------------------------------
-----------
 {"arr": "LIST", "key": "STRING", "num": "INTEGER", "bool": "BOOLEAN", "null": "NULL",
"float": "FLOAT"}
```

meta.is_cypher_type

Description

Checks if a given value matches the specified Cypher type. Supports types: `NULL` , `INTEGER` , `FLOAT` , `STRING` , `BOOLEAN` , `MAP` , `LIST` , and `NUMBER` .

Arguments

- `value` (anyelement): The value to check.
- `expected_type` (text): The expected Cypher type.

© 2026 SKAI Worldwide Co., Ltd.

Returns

- `boolean` : `true` if the value matches the expected type, `false` otherwise.

Example

```
SELECT meta.is_cypher_type(123, 'INTEGER') AS result;
result
-------
t

SELECT meta.is_cypher_type('Alice', 'STRING') AS result;
result
-------
t

SELECT meta.is_cypher_type('NULL'::jsonb, 'NULL') AS result;
result
-------
t
```

meta.count_self_loops

Description

Returns the number of self-loops (edges where the start and end vertices are the same) in a specified graph. If no relationship is provided, all edges are considered regardless of edge label. If no graph is provided, the `graph_path` setting is used.

Arguments

- `relationship` (text, optional): The name of the relationship/edge label to check. Defaults to `ag_edge`.
- `graph` (text, optional): The graph name. Defaults to the `graph_path` setting.

Returns

- `bigint` : Number of self-loop edges.

Example

```
MATCH (a:person {name: 'Bob'})
CREATE (a)-[:"KNOWS"]->(a);
MATCH (a:animal {name: 'Spot'})
CREATE (a)-[:"HAS_PET"]->(a);
SELECT meta.count_self_loops(graph=>'social');
 count_self_loops
------------------
                2
(1 row)

SELECT meta.count_self_loops('HAS_PET');
 count_self_loops
------------------
                1
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

meta.edge_density

Description

Calculates the edge density of a graph, defined as the ratio of existing edges to the maximum possible number of edges.
If no graph is provided, the `graph_path` setting is used.

Arguments

- `graph` (name, optional): The graph name. Defaults to the `graph_path` setting.

Returns

- `float`: Edge density of the graph.

Example

```
-- Calculate edge density for the current graph
SELECT meta.edge_density() AS result;

-- Calculate edge density for a specific graph
SELECT meta.edge_density('social_graph') AS result;
```

meta.estimated_edge_density

Description

Provides an estimated edge density of a graph using PostgreSQL statistics (`pg_class.reltuples`) instead of exact counts.
This is faster for large graphs but may be inaccurate. Uses `graph_path` if no graph name is provided.

Arguments

- `graph` (name, optional): The graph name. Defaults to the `graph_path` setting.

Returns

- `float`: Estimated edge density of the graph.

Example

```
-- Estimate edge density for the current graph
SELECT meta.estimated_edge_density() AS result;

-- Estimate edge density for a specific graph
SELECT meta.estimated_edge_density('social_graph') AS result;
```


© 2026 SKAI Worldwide Co., Ltd.

Release Notes

What changed in each release, and how to move a running installation forward.

AgensGraph 2.18

2.18.6.1

Release 2.18.6.1 changes no query, catalog or storage behaviour. It is built on PostgreSQL 18.6, as 2.18.6.0 is, and starts on a 2.18.6.0 data directory as it is.

- `agens --help` and `ag_ctl --help` send bug reports to **inquiry@skaiworldwide.com** and name **https://skaiworldwide.com/** as the home page. 2.18.6.0 named `bitnine.net`, a domain the company no longer holds.

2.18.6.0

Release 2.18.6.0 moves AgensGraph onto PostgreSQL 18 and adds a substantial part of the ISO/IEC 39075 (GQL) standard. It also brings typed-column property promotion, which lets a hot `jsonb` property become a real indexable column without changing how queries are written.

Several changes will break a working 2.17 application. Read **Migrating to 2.18** below before upgrading, and follow the procedure in **Upgrading to 2.18** — a 2.17 data directory cannot be started by 2.18 binaries.

Overview

- **PostgreSQL 18 base.** 2.17 was built on PostgreSQL 17.10; 2.18.6.0 is built on 18.6, so everything new in PostgreSQL 18 comes with it.
- **GQL.** New statements `LET`, `NEXT`, `FILTER`, `FINISH`, `INSERT` and `FOR`; `CALL { }` subqueries and `CALL … YIELD`; `OPTIONAL CALL`; the subquery value expressions `EXISTS { }`, `COUNT { }`, `COLLECT { }`, `ARRAY { }`, `VALUE { }` and `x IN { }`; standalone `ORDER BY`, `SKIP`, `OFFSET` and `LIMIT`; the `XOR` operator; and `CURRENT_GRAPH` / `CURRENT_PROPERTY_GRAPH`. See **GQL conformance**.
- **Typed-column property promotion.** `ALTER VLABEL movie ADD COLUMN year int GENERATED` gives a property a real, indexable column of a declared type, and a Cypher property read resolves to it without the query changing. See **Promoting a property to a column**.
- **Graph elements in native arrays.** `collect()` over a vertex, edge or path now builds a `vertex[]`, `edge[]` or `graphpath[]` instead of a jsonb picture, so the collected elements stay usable.
- **Planner work on traversal**, including scan pruning driven by `ag_graphmeta`, elision of unused pattern endpoints, and better estimates for variable-length edges.
- **Fifteen crash classes fixed**, several of them reachable with read privileges alone.

© 2026 SKAI Worldwide Co., Ltd.

New features

GQL statements

Statement	What it does
`LET`	Adds computed values to the row without dropping anything in scope.
`NEXT`	Chains one query onto the result of another.
`FILTER`	Filters rows; `FILTER` and `FILTER WHERE` forms.
`FINISH`	Ends a query that projects nothing.
`INSERT`	Creates elements; the GQL spelling of `CREATE`.
`FOR`	Iterates a list, with `WITH OFFSET` for the index.
`CALL { }`	A subquery, with an explicit variable import list.
`CALL … YIELD`	Calls a function and names its output columns.
`OPTIONAL CALL`	As `CALL`, but a call yielding nothing still produces a row.
Subquery expressions	`EXISTS { }` and `VALUE { }` over a subquery. `COUNT { }`, `COLLECT { }`, `ARRAY { }` and `x IN { }` do the same job but are AgensGraph extensions, not GQL.

Standalone `ORDER BY`, `SKIP`, `OFFSET` and `LIMIT` became clauses in their own right rather than parts of `RETURN` and `WITH`, so they can sit anywhere in a pipeline. They cannot end a query: a statement still has to finish with `RETURN`, `FINISH` or a write clause.

Also new: the `XOR` operator, `NULLIF`, `IS [NOT] UNKNOWN`, `RETURN ALL` / `WITH ALL`, `exists(property)`, `size(value)`, `cardinality()`, `toInteger()`, `toFloat()`, and SQL value functions inside Cypher.

Casts can now name a keyword type:
`::int`, `::numeric`, `::varchar`, `::bit`, `::interval`, `::boolean`, `::date`, `::timestamptz` and the rest. Several of those had no working spelling at all before. `::timestamp` and `::time` remain unwritable in Cypher; wrap the expression in SQL for those.

Typed-column property promotion

A property lives in a per-element `jsonb` document, which means a read walks a JSON document and cannot use an index on the value's own type. Promotion gives one property a generated, stored column of a declared type:

```
ALTER VLABEL movie ADD COLUMN year int GENERATED;
CREATE PROPERTY INDEX ON movie (year);
```

Queries do not change — `MATCH (m:movie) WHERE m.year > 2010` now resolves `m.year` to the column and can use the index. Promotion also works at creation time, and a column can take its value from a differently named property:

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE VLABEL doc (words int GENERATED (word_count), score numeric GENERATED);
```

The column carries the property named in the parentheses, so the read that resolves to `words` is `d.word_count` — the property key, not the column name.

Extension types work too, which is the case that matters for retrieval: with `pgvector` installed, `CREATE VLABEL doc (emb vector(1024) GENERATED)` gives an embedding a real indexable column instead of a `jsonb` array cast on every read.

A promoted key is no longer schemaless: a write whose value does not fit the column's type aborts. **Promoting a property to a column** covers the trade-offs.

Other additions

- `ALTER VLABEL … ADD COLUMN` / `DROP COLUMN`.
- A new catalog, `ag_label_property`, recording which column carries which property, and a new `ag_graph.graphmeta_valid` column that gates scan pruning. See **System catalogs**.
- Three new settings: `enable_graph_ddl`, `enable_property_promotion` and `enable_graph_endpoint_elision`. See **Server settings**.
- `log_statement = mod` now records Cypher writes. Previously a server asked to log data-modifying statements logged none of them — a SQL `INSERT` was recorded and a Cypher `CREATE` beside it was not.

Migrating to 2.18

These are the changes most likely to break something that works today. **Upgrading to 2.18** has the full list and the procedure.

New reserved words

`xor`, `current_graph` and `current_property_graph` are now reserved. An identifier, label, variable or alias with any of those names needs quoting, and quoting means **double quotes**, in Cypher as in SQL — `MATCH (n:"xor")`. A backtick is not a quoting character here, so `` `xor` `` is a syntax error.

`insert`, `finish` and `let` are also reserved, but two uses that worked on 2.16 and 2.17 were deliberately restored: `insert` as a variable, and `insert` or `finish` as a label in a pattern. Creating such a label still needs quoting — `CREATE VLABEL insert;` is a syntax error, `CREATE VLABEL "insert";` works, and the unquoted pattern spelling then finds it. What still needs quoting in every position is `let` or `finish` as a variable name, `let` as a label, and sorting by an alias named `let` or `finish`.

Keywords has the full matrix of which word fails in which position.

`CALL` was left unreserved, so `call` remains usable as a column, table, alias, label, property key and variable. Exactly two spellings change meaning and need quoting: `call(...)` — a function literally named `call` invoked with arguments — and `(call { ... })`, a node pattern whose variable is named `call` and which carries a property map but no label.

See **Keywords**.

A condition takes only a boolean

`WHERE n.count` and `FILTER n.name` used to read a property for non-zero or non-empty and select rows accordingly. They now raise a cast error:

```
ERROR:  cannot cast "Jack" (jsonb type string) to boolean
```


© 2026 SKAI Worldwide Co., Ltd.

This applies in `WHERE`, `FILTER`, `AND`, `OR`, `NOT`, `XOR`, `CASE WHEN` and `dijkstra()`'s qualifier. An explicit `::boolean` still converts, and a genuine boolean, a jsonb null and an absent key are unaffected. The change is deliberate: ISO/IEC 39075 defines a search condition as a boolean value expression.

Changed result types

- **`collect()` over a vertex, edge or path** returns `vertex[]`, `edge[]` or `graphpath[]` rather than a jsonb array of pictures. Reading a property off a collected element now works, where before it silently answered empty. A `collect` over a property still returns a jsonb list, so stored data keeps its shape.
- **`min()` and `max()` over a vertex or edge are now an error**:

  ```
  ERROR:  function min(vertex) is not unique
  ```

  PostgreSQL 18 added `min(record)` and `max(record)`, which now compete with `min(jsonb)` / `max(jsonb)` through the implicit `vertex → jsonb` cast. Cast explicitly — `min(v::jsonb)` — or aggregate the identity, `min(id(v))`.
- **`range()`** returns a jsonb array rather than an integer array.
- **`toStringList`, `toIntegerList` and `toFloatList`** were rewritten. Each now has two overloads: called on a real array, from SQL, they return `text[]`, `bigint[]` and `double precision[]`; called on a Cypher list, which is `jsonb`, they return `jsonb`. The old versions gave several silent wrong answers. A bigint lost its high bits, a non-numeric string converted to 0, a one-element list came back empty, and an element longer than 64 characters ran past a fixed buffer.

Changed result sets

`WITH … WHERE` with `SKIP` or `LIMIT` now pages after the filter. Previously the paging happened first, so `WITH x WHERE x > 1 ORDER BY x LIMIT 2` could return one row. The order is now `WHERE`, then `ORDER BY`, then `SKIP` / `LIMIT`. This changes result sets.

`RETURN <expr> AS d ORDER BY d` sorts the value, not its jsonb form. Rows and ordering are unchanged for every ordered type except a non-finite float, which used to sort among strings. The gain is that an index can serve the ordering.

A write clause's changes are visible to a later read in the same statement. `MATCH (p) SET p.age = 99 WITH p MATCH (q) RETURN q.age` returned the pre-write value, and `CREATE (:Q) WITH 1 MATCH (x:Q) RETURN count(x)` returned 0 while silently discarding the create.

`ALL`, `ANY`, `NONE` and `SINGLE` follow three-valued logic. `any([null] WHERE p)` is now null rather than false. Empty lists keep their vacuous results.

Stricter validation

Each of these used to succeed, and in most cases used to produce a wrong answer:

- `ALTER TABLE` may no longer reshape a graph label — add or drop a column, change inheritance, or rename a label column. The label DDL does the same job, and `enable_graph_ddl` lifts the restriction. Changing the type of `id`, `start`, `end` or `properties` is refused even then.
- SQL `MERGE` against a label now needs `enable_graph_dml`, as `INSERT`, `UPDATE` and `DELETE` already did.
- **Row-level security is now actually enforced on graph writes.** The `USING` clause of an `UPDATE` or `DELETE` policy was attached to a range-table entry that is never scanned, so it filtered nothing — a role with no `DELETE` policy at all deleted rows anyway. `INSERT` policies were checked against the wrong row, and edge creation was never checked, so an edge label with row security accepted every insert. If you rely on RLS over a graph, test it again on 2.18: writes that used to pass will now be refused.

© 2026 SKAI Worldwide Co., Ltd.

- A `WITH` may not carry the same name twice, an `ORDER BY` key may not compute with a volatile item, a pattern variable may not name a column already in scope, a graph write may not replace a property map with a scalar, and a `CALL` subquery may not return a name already bound outside it.

Diagnostics

`EXPLAIN` output changed for `count()` over a graph element, for property reads, and for variable-length traversal nodes, which now name the edge label they walk. Refresh any regression baselines that compare plan text.

Fixes

Fifteen crash classes were fixed. Several were reachable by a user with nothing but read privileges, and took down every session on the instance:

- `head('{}'::text[])` crashed the server while planning.
- `RETURN cardinality(['a','b'])` crashed at parse time.
- `MATCH ()-[r:e*1..2 = 5]->()` — a non-map constraint on a variable-length edge.
- `CREATE PROPERTY INDEX ON doc (tags[0])`.
- An aggregate inside a list comprehension.
- A list element longer than 64 characters ran past a fixed buffer.

Four more fixes, each of which had been giving a wrong answer rather than an error. `vertex` ordering and `edge` equality were wrong in 2.17.0, so `<`, `<=`, `>`, `>=` on a vertex and `=` on an edge did not mean what they said. `min(jsonb)` and `max(jsonb)` returned an arbitrary element. `DELETE` returned an empty value for the element it removed, and emptied other variables bound to it. And `x IN <list>` over a graph element compared full serializations rather than identity.

A PostgreSQL 18 regression is also fixed inside this release: `DISTINCT`, `GROUP BY`, `ORDER BY`, `min()` and `max()` over a property read failed with “could not determine which collation to use”. Such a key now takes the database default, the same collation jsonb itself uses, so results are unchanged.

Not in this release

To be explicit, because these have been asked about:

- **Write clauses inside `CALL { }`.** The parser refuses `CREATE`, `SET` and the other write clauses in a subquery body. `CALL { } IN TRANSACTIONS` does not exist.
- **Sessions and schemas from GQL** — `SESSION_SET`, `USE` — and typed graphs / graph types.
- **`CREATE OR REPLACE CONSTRAINT`**, which parses and is then refused.
- **Non-stored and non-generated promoted columns.** A promoted column must be `GENERATED … STORED`.

Upgrading to 2.18

AgensGraph 2.18 is built on PostgreSQL 18, so moving to it is a major-version upgrade: the data directory format changed and a 2.17 data directory cannot be started by 2.18 binaries. You have two routes — `pg_upgrade`, which reuses the existing data files and is fast, or a dump and reload.

Before either one, read **what changed** and check your application against it. Several changes will break queries that work on 2.17, and none of them are caught by the upgrade tooling.

© 2026 SKAI Worldwide Co., Ltd.

Check your application first

Work through this list against your codebase. Each item is a change that turns working 2.17 code into an error or a different answer on 2.18.

- **Identifiers named `xor`, `current_graph` or `current_property_graph`** — now reserved words. Rename them, or quote them with **double quotes** (`MATCH (n:"xor")`); a backtick is not a quoting character here. `let` and `finish` as a variable name, and `let` as a label, are also refused, and `CREATE VLABEL insert;` needs quoting even though the pattern spelling `MATCH (n:insert)` still works. See **Keywords** for the matrix of which word fails in which position.
- **`call(...)` or `(call { ... })`** — a function literally named `call`, or a node pattern whose variable is `call` carrying a property map and no label. Both change meaning; quote the name.
- **A condition that is not a boolean** — `WHERE n.count`, `FILTER n.name`. These used to select on non-zero or non-empty and now raise a cast error. Add `::boolean` or compare explicitly.
- **`collect()` over a vertex, edge or path** — now returns a native array rather than a jsonb array of pictures. Client code that parsed the jsonb form must change; code that reads properties off the collected elements starts working.
- **`min()` / `max()` over a vertex or edge** — now ambiguous and therefore an error. Use `min(v::jsonb)` or `min(id(v))`.
- **`range()`, `toStringList`, `toIntegerList`, `toFloatList`** — changed return types.
- **`WITH … WHERE` with `SKIP` or `LIMIT`** — pages after the filter now, which changes result sets.
- **Row-level security over a graph** — RLS is now actually enforced on graph writes. If you have policies on labels, re-test them: writes that used to slip through will be refused.
- **`ALTER TABLE` against a label** — refused for anything structural. Use the label DDL, or set `enable_graph_ddl`.
- **`EXPLAIN` regression baselines** — plan text changed for `count()` over an element, for property reads, and for variable-length traversal nodes.
- **A foreign key whose two sides are collated differently** — refused (`42P21`) when either collation is nondeterministic. 2.17 built it. Both sides carrying the same nondeterministic collation is still fine, and so are two different deterministic ones. `NOT VALID` does not defer it, and a foreign key on a label column is refused the same way. This one is not only an application change: a 2.17 dump restores the table **without the constraint**. Find the pairs before you upgrade — see **Foreign keys with a nondeterministic collation**.

Foreign keys with a nondeterministic collation

Run this on the **2.17** cluster, in every database, before you dump or upgrade:

```
SELECT c.conrelid::regclass AS referencing, c.conname, fa.attname AS col,
       fc.collname AS referencing_collation, pc.collname AS referenced_collation
  FROM pg_constraint c
  CROSS JOIN LATERAL unnest(c.conkey, c.confkey) AS k(fkatt, pkatt)
  JOIN pg_attribute fa ON fa.attrelid = c.conrelid AND fa.attnum = k.fkatt
  JOIN pg_attribute pa ON pa.attrelid = c.confrelid AND pa.attnum = k.pkatt
  JOIN pg_collation fc ON fc.oid = fa.attcollation
  JOIN pg_collation pc ON pc.oid = pa.attcollation
 WHERE c.contype = 'f'
   AND fa.attcollation <> pa.attcollation
   AND NOT (fc.collisdeterministic AND pc.collisdeterministic);
```

Every row is a constraint 2.18 will refuse. Give the two columns the same collation on 2.17, then dump. The condition is the one the server applies, read from the same catalog column, so it selects neither more nor fewer constraints than the upgrade will refuse.

© 2026 SKAI Worldwide Co., Ltd.

WARNING

Neither route warns you. `pg_upgrade --check` ends with `*Clusters are compatible*` and the upgrade then fails when it rebuilds the constraint. A dump and reload is worse: the restore reports success and the table arrives **without its foreign key**, after which rows that reference nothing are accepted.

Route 1. Using pg_upgrade

`pg_upgrade` creates the new cluster's catalogs and then reuses the existing user data files, so it does not rewrite your data. Because a graph is stored as ordinary schemas and tables, graph data migrates exactly as relational data does.

Before you begin

- **Install 2.18 in a separate location.** Do not overwrite the 2.17 installation: `pg_upgrade` needs both sets of binaries, and keeping 2.17 in place is what makes rollback possible.
- **Disk space.** By default the data files are copied, so you need room for a second copy. `--link` hard-links them instead — almost instant, little extra space — but both data directories must be on one filesystem and rollback becomes impossible once the new server starts.
- **Install the same external modules** in the 2.18 installation that the 2.17 cluster uses, `pgvector` for example. AgensGraph's own graph features are built into the server, so there is no graph extension to recreate.
- **A label carrying an ordinary column blocks the upgrade.** `pg_upgrade` delegates to `pg_dump --binary-upgrade`, which refuses a graph label that has a column beyond `id`, `start`, `end` and `properties`, rather than silently corrupting the physical column layout. Find them before you start:

  ```
  SELECT c.relname, a.attname
  FROM pg_catalog.ag_label l
    JOIN pg_class c ON c.oid = l.relid
    JOIN pg_attribute a ON a.attrelid = c.oid AND a.attnum > 0 AND NOT a.attisdropped
  WHERE a.attname NOT IN ('id', 'start', 'end', 'properties');
  ```

  Drop those columns, or move the data elsewhere, before upgrading.
- Run every command as the operating-system user that owns the data directory.

The commands below use these variables:

```
export OLD=/usr/local/agensgraph-2.17
export NEW=/usr/local/agensgraph-2.18
export OLDDATA=$OLD/data
export NEWDATA=$NEW/data
```

Step 1. Stop the old server

Both clusters must be down while `pg_upgrade` runs.

```
$OLD/bin/ag_ctl -D $OLDDATA stop
```

Step 2. Initialize the new cluster

Use the same superuser, encoding and locale as the old cluster — and match its data-checksum setting.

```
$NEW/bin/initdb -D $NEWDATA -U postgres --encoding=UTF8 --no-data-checksums
```


© 2026 SKAI Worldwide Co., Ltd.

WARNING

`--no-data-checksums` is not optional for most upgrades. PostgreSQL 18 turns data page checksums on by default, and PostgreSQL 17 turned them off. So unless your 2.17 cluster was initialised with `-k`, the two clusters disagree and `pg_upgrade` stops before it does anything:

```
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok

old cluster does not use data checksums but the new one does
Failure, exiting
```

Check which side you are on before initialising:

```
$OLD/bin/pg_controldata -D $OLDDATA | grep checksum
```

If it reports `Data page checksum version: 0`, pass `--no-data-checksums`. If it reports `1`, leave the flag off. Turning checksums on is a separate operation — see **`pg_checksums`** — and it is worth doing, but do it after the upgrade, not as part of it.

Step 3. Check compatibility

`--check` validates both clusters and changes nothing.

```
$NEW/bin/pg_upgrade \
  -b $OLD/bin -B $NEW/bin \
  -d $OLDDATA -D $NEWDATA \
  -U postgres --check
```

A successful check ends with `*Clusters are compatible*`.

WARNING

A green check does not mean the upgrade will succeed. It does not look for **foreign keys with a nondeterministic collation**, which Step 4 then fails on while rebuilding the constraint. Run that query yourself.

Pass the database superuser with `-U`. By default `pg_upgrade` connects as your operating-system user, which is usually not a database role.

Step 4. Run the upgrade

The same command without `--check`:

```
$NEW/bin/pg_upgrade \
  -b $OLD/bin -B $NEW/bin \
  -d $OLDDATA -D $NEWDATA \
  -U postgres
```

It finishes with `Upgrade Complete`. Add `--link` to both the check and the upgrade to hard-link the data files instead of copying them.

© 2026 SKAI Worldwide Co., Ltd.

Step 5. Start the new server

```
$NEW/bin/ag_ctl -D $NEWDATA -l $NEW/logfile start
```

Confirm you are on the new server, and that the graph came across:

```
SELECT version();
```

```
SET graph_path = <your graph>;
MATCH (n) RETURN count(n);
```

Step 6. Post-upgrade tasks

Three of these are AgensGraph-specific and none of them are optional.

Rebuild statistics. `pg_upgrade` does not carry optimizer statistics across, so plans will be poor until they are rebuilt:

```
$NEW/bin/vacuumdb -U postgres --all --analyze-in-stages
```

Re-gather graph metadata. `ag_graphmeta` drives the planner's scan pruning. A binary upgrade leaves it empty and marks the graph invalid, so pruning is off until you rebuild it, once per graph:

```
SET graph_path = <your graph>;
SELECT regather_graphmeta();
```

Check that it took:

```
SELECT graphname, graphmeta_valid FROM pg_catalog.ag_graph;
```

Re-tune edge label statistics. 2.18 lowered the statistics target on a new edge label's `start` and `end` columns from 10000 to 1000. Labels created on 2.17 keep the old value and pay a large planning cost on every edge join — hundreds of milliseconds on a big label. For each edge label:

```
ALTER TABLE <graph>.<elabel> ALTER COLUMN start SET STATISTICS 1000;
ALTER TABLE <graph>.<elabel> ALTER COLUMN "end" SET STATISTICS 1000;
ANALYZE <graph>.<elabel>;
```

Then, once you have confirmed the upgraded cluster is working, remove the old data files with the script `pg_upgrade` wrote in the directory you ran it from:

```
./delete_old_cluster.sh
```

Rolling back

In copy mode, and before you run `delete_old_cluster.sh`, the 2.17 data directory is untouched. Stop the new server and start the old one:

```
$NEW/bin/ag_ctl -D $NEWDATA stop
$OLD/bin/ag_ctl -D $OLDDATA -l $OLD/logfile start
```


© 2026 SKAI Worldwide Co., Ltd.

After a `--link` upgrade, rollback is not possible once the new server has started, because the two clusters share data files. Restore from a backup.

Route 2. Dump and reload

Slower, but it rewrites everything and needs no second copy of the binaries at runtime.

```
$OLD/bin/pg_dumpall -U postgres -f cluster.sql
# start the 2.18 cluster, then:
$NEW/bin/psql -U postgres -d postgres -f cluster.sql
```

> **WARNING**
>
> `psql` does not stop on error, and exits 0 even when statements failed. Anything the new server refuses — a **foreign key with a nondeterministic collation**, for instance — is one line in the output and the restore carries on without it. Read the log before you believe the restore.

To have the restore stop instead, dump the roles apart from the data. The roles cannot be restored under `ON_ERROR_STOP`: `pg_dumpall` writes `CREATE ROLE` for the bootstrap superuser, which already exists on the new cluster, and `--clean --if-exists` only moves the failure to the `DROP ROLE` of the role you are connected as.

```
$OLD/bin/pg_dumpall -U postgres --globals-only -f globals.sql
$OLD/bin/pg_dump    -U postgres -Fc -d mydb -f mydb.dump       # once per database
# start the 2.18 cluster, then:
$NEW/bin/psql -U postgres -d postgres -f globals.sql           # roles; errors here are benign
$NEW/bin/createdb -U postgres mydb
$NEW/bin/pg_restore -U postgres --exit-on-error -d mydb mydb.dump
```

Two things to know:

- **A restore may need `enable_graph_ddl`.** A dump of a label that carries ordinary columns emits `SET enable_graph_ddl = on` around its `ALTER TABLE … ADD COLUMN` statements. That setting is writable by any role, so an ordinary user can restore — unless the role has it forced off by `ALTER ROLE … SET`.
- **The graph metadata and statistics tasks above still apply.**

Downgrading is not supported

A 2.18 dump will not restore into 2.17 if any graph uses **promoted columns**. The dump contains a `CREATE VLABEL` form that a 2.17 parser rejects, and in a plain-text stream everything after that statement is lost. Keep a 2.17 backup taken before the upgrade if you need a way back.

Note also that `pg_dump` and `pg_restore` do not check the AgensGraph version. Only PostgreSQL's `server_version` range check applies. So a mismatch shows up as a parse failure partway through a restore, rather than as a refusal up front.

A worked example

Upgrade walkthrough runs the whole procedure on a small cluster with real commands and output.

Upgrade walkthrough

A complete 2.17 to 2.18 upgrade on a small cluster, with the commands and the output they produced. **Upgrading to 2.18** is the reference; this page is what it looks like when you do it.

© 2026 SKAI Worldwide Co., Ltd.

The cluster here holds one graph and one ordinary table, so you can see that both come across:

```
CREATE GRAPH social;
SET graph_path = social;
CREATE VLABEL person;
CREATE ELABEL knows;
CREATE (:person {name: 'Ann', age: 30}), (:person {name: 'Bo', age: 41});
MATCH (a:person {name: 'Ann'}), (b:person {name: 'Bo'}) CREATE (a)-[:knows {since: 2019}]->(b)
;
CREATE PROPERTY INDEX ON person (name);
```

```
CREATE TABLE audit (id serial primary key, note text);
INSERT INTO audit(note) VALUES ('before upgrade');
```

On 2.17 that reads back as:

```
MATCH (n:person)-[r:knows]->(m:person) RETURN n.name, r.since, m.name;
```

```
 name  | since | name
-------+-------+------
 "Ann" | 2019  | "Bo"
```

1. Note the old cluster's checksum setting

This decides one flag in the next step, and getting it wrong stops the upgrade before it starts.

```
$OLD/bin/pg_controldata -D $OLDDATA | grep checksum
```

```
Data page checksum version:           0
```

Zero means checksums are off, which is the PostgreSQL 17 default.

2. Stop the old server

```
$OLD/bin/ag_ctl -D $OLDDATA stop
```

3. Initialize the 2.18 cluster

Because the old cluster reported `0`, the new one has to be initialised with `--no-data-checksums`. PostgreSQL 18 turns checksums on by default, so leaving the flag off is what produces the failure in the next section.

```
$NEW/bin/initdb -D $NEWDATA -U postgres --encoding=UTF8 --no-data-checksums
```

```
$NEW/bin/pg_controldata -D $NEWDATA | grep checksum
```

```
Data page checksum version:           0
```

Both clusters now agree.

© 2026 SKAI Worldwide Co., Ltd.

What it looks like when they do not

This is the whole output of `pg_upgrade --check` against a 2.18 cluster initialised without that flag:

```
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok

old cluster does not use data checksums but the new one does
Failure, exiting
```

It stops at the second check and changes nothing, so recovering is just a matter of re-initialising the new cluster.

4. Check compatibility

```
$NEW/bin/pg_upgrade -b $OLD/bin -B $NEW/bin -d $OLDDATA -D $NEWDATA -U postgres --check
```

```
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok
Checking database connection settings                       ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for contrib/isn with bigint-passing mismatch        ok
Checking for valid logical replication slots                ok
Checking for subscription state                             ok
Checking data type usage                                    ok
Checking for objects affected by Unicode update             ok
Checking for not-null constraint inconsistencies            ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for new cluster tablespace directories             ok

*Clusters are compatible*
```

5. Run the upgrade

```
$NEW/bin/pg_upgrade -b $OLD/bin -B $NEW/bin -d $OLDDATA -D $NEWDATA -U postgres
```

The tail of the output:

```
Copying old pg_multixact/offsets to new server              ok
Deleting files from new pg_multixact/members                ok
Copying old pg_multixact/members to new server              ok
Setting next multixact ID and offset for new cluster        ok
Resetting WAL archives                                      ok
Setting frozenxid and minmxid counters in new cluster       ok
Restoring global objects in the new cluster                 ok
Restoring database schemas in the new cluster               ok
Copying user relation files                                 ok
Setting next OID for new cluster                            ok
Sync data directory to disk                                 ok
Creating script to delete old cluster                       ok
Checking for extension updates                              ok

Upgrade Complete
----------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
Some statistics are not transferred by pg_upgrade.
Once you start the new server, consider running these two commands:
    .../bin/vacuumdb -U postgres --all --analyze-in-stages --missing-stats-only
    .../bin/vacuumdb -U postgres --all --analyze-only
Running this script will delete the old cluster's data files:
    ./delete_old_cluster.sh
```

6. Start the new server and check what came across

```
$NEW/bin/ag_ctl -D $NEWDATA -l $NEW/logfile start
```

```
SELECT version();
```

```
 PostgreSQL 18.6 (AgensGraph 2.18.6.1) on x86_64-pc-linux-gnu, compiled by gcc ...
```

The graph, including the edge property:

```
SET graph_path = social;
MATCH (n:person)-[r:knows]->(m:person) RETURN n.name, r.since, m.name;
```

```
 name  | since | name
-------+-------+------
 "Ann" | 2019  | "Bo"
```

The property index:

```
\d social.person
```

```
                            Table "social.person"
   Column    |  Type   | Nullable |                     Default
-------------+---------+----------+-------------------------------------------------
 id          | graphid | not null | graphid(graph_labid('social.person'), nextval(…))
 properties | jsonb   | not null | jsonb_build_object()
Indexes:
    "person_pkey" PRIMARY KEY, btree (id)
    "person_name_idx" btree ((properties.'name'))
```

And the ordinary table:

```
SELECT * FROM audit;
```

```
 id |      note
----+----------------
  1 | before upgrade
```


© 2026 SKAI Worldwide Co., Ltd.

7. The three post-upgrade tasks

Statistics

```
$NEW/bin/vacuumdb -U postgres --all --analyze-in-stages
```

Graph metadata

A binary upgrade leaves `ag_graphmeta` empty and the graph marked invalid, so the planner's scan pruning is off:

```
SELECT graphname, graphmeta_valid FROM pg_catalog.ag_graph;
```

```
 graphname | graphmeta_valid
-----------+-----------------
 social    | f
```

Rebuild it, once per graph:

```
SET graph_path = social;
SELECT regather_graphmeta();
```

```
NOTICE:  regathering graphmeta...
 regather_graphmeta
--------------------
 t
```

```
SELECT graphname, graphmeta_valid FROM pg_catalog.ag_graph;
```

```
 graphname | graphmeta_valid
-----------+-----------------
 social    | t
```

Edge label statistics

The upgrade carries the old statistics target across, which is the value 2.17 used:

```
SELECT attname, attstattarget FROM pg_attribute
WHERE attrelid = 'social.knows'::regclass AND attname IN ('start', 'end')
ORDER BY 1;
```

```
 attname | attstattarget
---------+---------------
 end     |         10000
 start   |         10000
```

10000 is high enough to cost real planning time on a large edge label, and 2.18 creates new edge labels with 1000. Bring the upgraded labels into line:

© 2026 SKAI Worldwide Co., Ltd.

```
ALTER TABLE social.knows ALTER COLUMN start SET STATISTICS 1000;
ALTER TABLE social.knows ALTER COLUMN "end" SET STATISTICS 1000;
ANALYZE social.knows;
```

8. Remove the old cluster

Only after you are satisfied the new one is working:

```
./delete_old_cluster.sh
```

Until you run that, the 2.17 data directory is untouched and you can roll back by stopping the new server and starting the old one — provided you did not use `--link`.

AgensGraph 2.17.0

This release of AgensGraph brings significant enhancements and bug fixes, including upstream compatibility with PostgreSQL 17.10, improved Cypher query support, and build infrastructure updates.

Upstream changes

- **Upstream Merge:** AgensGraph is now compatible with **PostgreSQL 17** up to the 17.10 release, incorporating the latest PostgreSQL improvements and security patches.
- **Incremental Backup:** `pg_basebackup` can now take incremental backups, combined later with `pg_combinebackup`, reducing the size and time of routine base backups.
- **MERGE Enhancements:** `MERGE` now supports a `RETURNING` clause (with the `merge_action()` function) and the `WHEN NOT MATCHED BY SOURCE` action.
- **SQL/JSON `JSON_TABLE`:** Added `JSON_TABLE()` along with the SQL/JSON query functions `JSON_EXISTS()`, `JSON_QUERY()`, and `JSON_VALUE()` for standards-compliant JSON processing.
- **VACUUM Memory & Performance:** A new internal data structure for storing dead tuple IDs greatly reduces VACUUM memory usage and improves performance on large tables.
- **Query Performance:** Numerous planner and executor improvements, including better handling of `IS [NOT] NULL` and streaming I/O for sequential scans.

Enhancements

- **Optimize DELETE / DETACH DELETE performance — up to ~30x faster (#781):**
  - Deleting a vertex no longer scans every edge label sequentially, making deletes of highly-connected vertices roughly **30x faster** in internal tests.
- **New `enable_graph_dml` configuration parameter (#786):**
  - Adds the `enable_graph_dml` GUC, which permits running SQL DML statements (`INSERT` / `UPDATE` / `DELETE`) directly against graph objects (the underlying vertex and edge tables). It defaults to `off` and is superuser-only, because issuing raw SQL DML on graph storage can corrupt graph data.
- **pgvector and expression property indexes on graph properties (#804):**
  - `CREATE PROPERTY INDEX` can now build and use pgvector HNSW indexes over a `::vector(n)` cast and GIN indexes over `to_tsvector(...)` on graph properties — the foundation for vector, full-text, and hybrid search inside Cypher (see the new Vector / Full Text / Hybrid Search guides).

© 2026 SKAI Worldwide Co., Ltd.

AI Library:

You can access those products at the **AI Repository**

- **LangChain AgensGraph**
  - Python plugin to provide support for GraphStore and VectorStore integration in `langchain`.
- **LlamaIndex AgensGraph**
  - Python plugin to provide support for AgensGraphStore, PropertyGraphIndex, AgensgraphVectorStore integration in `llamaindex`.
- **AgensGraph LightRag**
  - Python plugin to add support for storing and querying knowledge graphs in AgensGraph with `LightRAG`.
- **Cognee Community Graph Adapter - AgensGraph**
  - Python plugin to provide an AgensGraph graph database adapter for the `Cognee` framework.
- **AgensGraph MCP Server**
  - A Model Context Protocol (MCP) server implementation that provides database interaction and allows graph exploration capabilities through AgensGraph.

Bug fixes

- **Cannot use graph entities in list comprehensions (#788)**
  - Graph entities and the functions that operate on them — for example `nodes()`, `relationships()`, `startNode()`, `type()` and `label()` — can now be used inside list comprehensions such as `[n IN nodes(p) | n]` or `[r IN relationships(p) WHERE type(r) = 'knows']`. Previously these returned incorrect results because graph objects were collected into an untyped JSONB array; they are now collected into a properly typed array.
- **`CypherList is expected` error in the `IN` operator (#789)**
  - Fixed a spurious `CypherList is expected` error when the right-hand side of the `IN` operator is a list produced by an expression, for example `x IN split(s, ',')` or `x IN ['1', '2']`.
- **Assertion failure with NULL in the `SET +=` operator (#791)**
  - Using `NULL` with the `SET +=` (map-merge) operator no longer triggers an assertion failure; the NULL case is now handled safely with a `COALESCE` fallback.
- **`ORDER BY` could not reference variables outside `RETURN` (#805)**
  - A Cypher `RETURN` followed by `ORDER BY` on a column not in the projection — e.g. `MATCH (p:person) RETURN p.name ORDER BY p.age` — now works instead of failing.

Documentation:

- **Vector, Full Text, and Hybrid Search guides:**
  - New developer-manual pages for semantic (pgvector), full-text, and hybrid search over graph properties, including indexing and GraphRAG patterns.
- **Documentation Portal:**
  - The **documentation portal** is updated for the AgensGraph 2.17 release.

© 2026 SKAI Worldwide Co., Ltd.

Appendixes

GQL conformance

AgensGraph implements part of ISO/IEC 39075:2024, Information technology — Database languages — GQL. This appendix is not a full statement of conformance. It covers the main topics in as much detail as is reasonable and useful.

The tables below are keyed by subclause of the standard, because that is the unit the standard itself uses to attach a conformance rule to a piece of syntax. Every row was checked against a running 2.18 server. A row says **Yes** only when the syntax the subclause defines was accepted and produced the result the subclause requires. It says **Partial** when some of the subclause's syntax is accepted and the rest raises an error, and **No** when none of it parses. Where the standard offers several spellings of the same idea, the comment says which one AgensGraph takes.

Three general points hold across every row and are not repeated in each comment.

GQL is not a separate mode. A GQL statement and a Cypher statement go through the same parser and may be mixed in one pipeline, so `LET` sits next to `WITH` and `INSERT` next to `CREATE`. See **Cypher, GQL and SQL**.

A graph is a PostgreSQL schema and a label is a table. So the parts of the standard that describe a catalog of directories, schemas and graph types have a PostgreSQL counterpart rather than a GQL one. The working graph comes from the `graph_path` setting instead of a `USE` clause, and privileges are PostgreSQL privileges. See **How a graph is stored**.

Errors carry a five-character SQLSTATE, not a GQLSTATUS. See **Error codes**.

Supported features

These subclauses are wholly or partly implemented. A **Partial** comment names what is missing, so that a reader porting standard GQL knows what to look for.

Subclause	Name	Support	Comment
8.1–8.4	`<start transaction command>`, `<transaction characteristics>`, `<rollback command>`, `<commit command>`	Yes	PostgreSQL's own transaction statements answer these. `START TRANSACTION READ ONLY`, `COMMIT` and `ROLLBACK` all work.
9.2	`<procedure body>`	Partial	`NEXT` chains one statement onto the result of the previous one. `NEXT` cannot carry a `YIELD` clause, and the binding-variable definition block that 9.2 also allows does not exist — see 10.1–10.3 below.
11.1	`<graph expression>`	Partial	`CURRENT_GRAPH` and `CURRENT_PROPERTY_GRAPH` are accepted, in Cypher and in SQL. They are bare keywords: `current_graph()` is a syntax error. No other form of graph expression exists.

© 2026 SKAI Worldwide Co., Ltd.

Subclause	Name	Support	Comment
12.4	`<create graph statement>`	Partial	`CREATE GRAPH name` and `CREATE GRAPH IF NOT EXISTS name` work. The graph type the standard requires after the name (`ANY` , `LIKE` , or a graph type reference) is not accepted, nor is `OR REPLACE` , `PROPERTY GRAPH` or `AS COPY OF` .
12.5	`<drop graph statement>`	Partial	`DROP GRAPH name` and `DROP GRAPH IF EXISTS name` work, plus a non-standard `CASCADE` , which is needed because a new graph already contains `ag_vertex` and `ag_edge` . `DROP PROPERTY GRAPH` is a syntax error.
13.1	`<linear data-modifying statement>`	Yes	A statement composes read clauses and write clauses in one pipeline, and a write clause's changes are visible to a read clause after it.
13.2	`<insert statement>`	Yes	`INSERT` takes the same pattern as Cypher `CREATE` and is a synonym for it.
13.3	`<set statement>`	Partial	Property assignment works, including the non-standard `+=` merge form. The label-set assignment the subclause also defines does not exist: a vertex has exactly one label.
13.4	`<remove statement>`	Partial	`REMOVE n.prop` removes a property. `REMOVE n:Label` does not exist, for the same reason.
13.5	`<delete statement>`	Yes	`DELETE` and `DETACH DELETE` . Deleting a vertex that still has edges is refused unless `DETACH` is given.
14.2	`<composite query expression>`	Partial	`UNION` , `UNION ALL` , `EXCEPT` and `INTERSECT` compose two query statements. `OTHERWISE` is a syntax error.
14.3	`<linear query statement>` and `<simple query statement>`	Yes	A query is a sequence of the primitive query statements below, ending in a primitive result statement.
14.4	`<match statement>`	Partial	`MATCH` and `OPTIONAL MATCH` work. The braced form, `OPTIONAL { MATCH … MATCH … }` , is a syntax error.
14.5	`<call query statement>`	Partial	Both call forms work. See 15.2 and 15.3 for what each one requires.
14.6	`<filter statement>`	Yes	Both `FILTER expr` and `FILTER WHERE expr` .
14.7	`<let statement>`	Yes	Additive, row-wise, and able to lead a query. An aggregate in a `LET` expression is refused.
14.8	`<for statement>`	Partial	Iterating a list works, with `WITH OFFSET AS name` giving a 0-based index. `WITH ORDINALITY` is a syntax error, and `FOR` cannot iterate a binding table.

© 2026 SKAI Worldwide Co., Ltd.

Subclause	Name	Support	Comment
14.9	`<order by and page statement>`	Yes	`ORDER BY`, `SKIP`, `OFFSET` and `LIMIT` are statements in their own right and can sit anywhere in a pipeline. Like the standard, AgensGraph does not let one end a query — 14.3 requires a primitive result statement last — so a query ending in `ORDER BY` raises `ERROR: Cypher query must end with RETURN, FINISH or update clause`.
14.10	`<primitive result statement>`	Yes	`RETURN`, `RETURN … ORDER BY …`, and `FINISH`.
14.11	`<return statement>`	Partial	`RETURN`, `RETURN DISTINCT`, `RETURN ALL`, `RETURN *` and explicit aliases. `RETURN NO BINDINGS` is a syntax error. Grouping is implicit — see 16.15.
15.1	`<call procedure statement>` and `<procedure call>`	Yes	`CALL` and `OPTIONAL CALL`, over either call form.
15.2	`<inline procedure call>`	Partial	`CALL (v) { … }`, `CALL () { … }`, `CALL (*) { … }` and `CALL { WITH v … }` all import correctly. A bare `CALL { … }` deviates: Syntax Rule 3 says an absent variable scope clause is rewritten to one naming every field of the incoming working record, so under the standard it imports everything, as `CALL (*) { … }` does. AgensGraph imports nothing instead, and raises no error, so the query answers a different question than it appears to — always write an import list. Write clauses in the subquery body are refused, and `IN TRANSACTIONS` does not exist.
15.3	`<named procedure call>`	Partial	`CALL routine(args) YIELD item [AS alias]` and `YIELD *` work, correlated when an argument references a variable already in scope. `YIELD` is mandatory, where the subclause makes it optional, and the clause can neither lead nor end a query. The name resolves to a PostgreSQL function, not to a procedure of the catalog 17.5 describes: a stock server registers no procedures, and calling one raises `<name>() is a procedure`.
16.3	`<graph pattern binding table>`	Partial	A graph pattern produces a binding table. The pattern `YIELD` clause the subclause also allows is a syntax error.
16.4	`<graph pattern>`	Partial	Comma-separated path patterns work. The match modes (`DIFFERENT EDGES`, `REPEATABLE ELEMENTS`) and the `KEEP` clause are syntax errors.
16.5	`<insert graph pattern>`	Yes	The pattern after `INSERT` or `CREATE` may create vertices, edges, or edges between vertices already bound.
16.7	`<path pattern expression>`	Partial	Concatenation works. Path multiset alternation and path pattern union — two path patterns separated by `|` — are syntax errors.

© 2026 SKAI Worldwide Co., Ltd.

Subclause	Name	Support	Comment
16.8	`<label expression>`	Partial	A single label name only. Disjunction (`:a|b`), conjunction (`:a&b`), negation (`:!a`) and the wildcard (`:%`) are all syntax errors. Test a label with the `label()` function in `WHERE` instead.
16.9	`<path variable reference>`	Yes	`MATCH p = (…)` binds the whole path, and `nodes()` , `relationships()` and `length()` read it.
16.10	`<element variable reference>`	Yes	A pattern variable is in scope for every clause after it.
16.11	`<graph pattern quantifier>`	Partial	The edge quantifier `-[r*1..3]->` walks a variable number of hops. The standard's quantifier on a parenthesized path pattern, `(…)->{1,3}` , is a syntax error, and so is `+` in place of `*` .
16.13	`<where clause>`	Partial	`WHERE` after a `MATCH` , and inside `FILTER` , `WITH` and a list comprehension. `WHERE` inside a node pattern is a syntax error. A search condition must be a boolean; a property read is not coerced.
16.16	`<order by clause>`	Yes	Multiple sort keys, `ASC` / `DESC` , `NULLS FIRST` / `NULLS LAST` , expressions as sort keys, and a `RETURN` alias as a sort key.
16.17	`<sort specification list>`	Yes	As 16.16. An integer sort key is a constant, not an output-column number — see **Cypher compatibility**.
16.18	`<limit clause>`	Yes	`LIMIT` , including an expression as the count.
16.19	`<offset clause>`	Yes	Both spellings the subclause defines, `OFFSET` and `SKIP` .
19.1	`<search condition>`	Yes	A condition takes a boolean and nothing else, which is what the subclause requires.
19.2	`<predicate>`	Partial	The four predicates below. The nine others 19.6–19.13 defines are not supported.
19.3	`<comparison predicate>`	Yes	`=` , `<>` , `<` , `<=` , `>` , `>=` , over scalars, graph elements and lists.
19.4	`<exists predicate>`	Partial	Four of the five forms the subclause defines: `EXISTS { pattern }` (with an optional `WHERE`), `EXISTS ( pattern )` (without one — a `WHERE` inside the parentheses is a syntax error), `EXISTS { MATCH … }` , and `EXISTS { <read statement> }` . `EXISTS ( MATCH … )` is a syntax error, and a match statement block of more than one `MATCH` needs a `RETURN` to parse, so Feature GQ22 is not met.
19.5	`<null predicate>`	Yes	`IS NULL` and `IS NOT NULL` .

© 2026 SKAI Worldwide Co., Ltd.

Subclause	Name	Support	Comment
20.1	`<value expression>`	Yes	Operators, function calls, `CASE`, subquery expressions and constructors compose freely.
20.2	`<value expression primary>`	Yes	Literals, variables, property references, parenthesized expressions.
20.3	`<value specification>`	Partial	`SESSION_USER` and `CURRENT_USER`. A parameter is a PostgreSQL parameter, `$1`, supplied by the driver rather than by a GQL general-parameter declaration.
20.6	`<value query expression>`	Yes	`VALUE { … }`, which is all this subclause defines: the body's single value, in its own type, and null when the body is empty. Syntax Rule 10 requires the body to carry `LIMIT 1` or an aggregate; AgensGraph accepts any body and raises `more than one row returned by a subquery used as an expression` at run time instead. `EXISTS { }` is 19.4; `COUNT { }`, `COLLECT { }`, `ARRAY { }` and `x IN { }` are extensions — see below.
20.7	`<case expression>`	Yes	Both the searched and the simple form, plus `NULLIF` and `COALESCE`.
20.8	`<cast specification>`	Partial	The `::` operator casts, and it accepts a keyword type name (`::int`, `::numeric`, `::interval`). `CAST(x AS t)` is a syntax error in Cypher; it works in SQL.
20.9	`<aggregate function>`	Partial	`COUNT(*)`, and the general set functions `AVG`, `COUNT`, `MAX`, `MIN`, `SUM`, `STDDEV_SAMP` and `STDDEV_POP`, each accepting `DISTINCT` or `ALL`. `MIN` and `MAX` over a vertex or edge are ambiguous and must be given a cast or an identity. `COLLECT_LIST` is spelled `collect`. The binary set functions `PERCENTILE_CONT` and `PERCENTILE_DISC` are unreachable from Cypher: both need `WITHIN GROUP`, which is a syntax error there.
20.11	`<property reference>`	Yes	`n.prop`, nested as `n.a.b`, and subscripted as `n.list[0]`.
20.12	`<binding variable reference>`	Yes	Names bound by a pattern, `WITH`, `LET`, `FOR` or `YIELD`.
20.13	`<path value expression>`	Partial	A path bound by `MATCH p = (…)` is a first-class value of type `graphpath`.
20.15	`<list value expression>`	Partial	`[a, b, c]`, subscripting, `IN`, and list comprehensions. `LIST[…]` is a syntax error.
20.17	`<list value constructor>`	Partial	As 20.15.

© 2026 SKAI Worldwide Co., Ltd.

Subclause	Name	Support	Comment
20.18	`<record constructor>`	Partial	`{a: 1, b: 2}` builds a map, and `.a` reads a field back. The `RECORD { … }` spelling is a syntax error.
20.19	`<field>`	Yes	Field access on a map, `({a: 1}).a`.
20.20	`<boolean value expression>`	Yes	`AND`, `OR`, `NOT`, `XOR`, and `IS [NOT] UNKNOWN`.
20.21	`<numeric value expression>`	Yes	`+`, `-`, `*`, `/`, `%`, `^`.
20.22	`<numeric value function>`	Yes	`abs`, `ceil`, `floor`, `round`, `sign`, `sqrt`, `exp`, `ln`, `log`, `log10`, `power`, `mod`, `pi`, the trigonometric functions, `degrees`, `radians`, `char_length`, `octet_length` and `rand`.
20.23	`<string value expression>`	Yes	Concatenation with `||`, and with `+`.
20.24	`<character string function>`	Partial	`upper`, `lower`, `btrim`, `ltrim`, `rtrim`, `trim`, `substring(s, from, count)`, `left`, `right`, `replace`, `reverse`, `split`. `NORMALIZE` and the `SUBSTRING(s FROM a FOR b)` spelling are syntax errors in Cypher.
20.26	`<datetime value expression>`	Partial	Date and timestamp arithmetic through PostgreSQL's types.
20.27	`<datetime value function>`	Partial	`CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, `LOCALTIMESTAMP`. The underscored spellings the standard uses, such as `LOCAL_TIME`, are not recognized.
21.1	Names and variables	Partial	An unquoted identifier folds to lower case; a double-quoted identifier keeps its case. The standard's backtick-delimited identifier is a syntax error.
21.2	`<literal>`	Partial	Integers including hexadecimal, octal, binary and underscore-separated forms; floats; strings; `true`, `false` and `null`; lists; maps. The typed literals — `DATE '…'`, `DURATION '…'`, byte strings — are syntax errors, as is `UNKNOWN` used as a value.
21.3	`<token>`, `<separator>`, and `<identifier>`	Partial	`--` and `/* */` delimit comments, as the standard requires. This makes the openCypher anonymous arrows `-->` and `--` unusable — see **Cypher compatibility**.

Features not supported

None of the syntax these subclauses define is accepted. The comment gives the AgensGraph route to the same result where there is one.

© 2026 SKAI Worldwide Co., Ltd.

Subclause	Name	Support	Comment
7.1	`<session set command>`	No	`SESSION SET` is a syntax error. Set the working graph with `SET graph_path = name`, which is a PostgreSQL setting and follows PostgreSQL's session and transaction scoping.
7.2	`<session reset command>`	No	Use `RESET graph_path` or `RESET ALL`.
7.3	`<session close command>`	No	Close the connection.
7.4	`<session parameter specification>`	No	None.
10.1	`<graph variable definition>`	No	None. A query addresses one graph, named by `graph_path`.
10.2	`<binding table variable definition>`	No	A `WITH` or `LET` binding, or a SQL common table expression around the Cypher, carries an intermediate result forward.
10.3	`<value variable definition>`	No	Use `LET`, which is 14.7 and is supported.
11.2	`<binding table expression>`	No	None.
11.3	`<object expression primary>`	No	None.
12.1	`<linear catalog-modifying statement>`	No	Catalog changes are PostgreSQL DDL statements and are not composable into a query pipeline.
12.2	`<create schema statement>`	No	A graph is a schema; `CREATE GRAPH` creates it. SQL `CREATE SCHEMA` also exists but does not make a graph.
12.3	`<drop schema statement>`	No	`DROP GRAPH`.
12.6	`<create graph type statement>`	No	None. A label is untyped, except for a property promoted to a column, which acquires that column's type. See **`CREATE VLABEL`**.
12.7	`<drop graph type statement>`	No	None.
12.8	`<call catalog-modifying procedure statement>`	No	None.
13.6	`<call data-modifying procedure statement>`	No	A write clause inside a `CALL` body is refused.

© 2026 SKAI Worldwide Co., Ltd.

Subclause	Name	Support	Comment
14.1	`<composite query statement>`	No	The optional `USE` prefix a composite query statement carries has no counterpart; the composite query expression itself, 14.2, is supported.
14.12	`<select statement>`	No	`SELECT … FROM MATCH (…)` is a syntax error. SQL `SELECT` over a Cypher subquery does the same job: see **SQL and Cypher**.
16.1	`<at schema clause>`	No	A statement runs in the database it is connected to.
16.2	`<use graph clause>`	No	`SET graph_path = name`.
16.6	`<path pattern prefix>`	No	Neither the path modes (`WALK` , `TRAIL` , `SIMPLE` , `ACYCLIC`) nor the path search prefixes (`ALL` , `ANY` , `ANY SHORTEST` , `ALL SHORTEST` , `SHORTEST k` , `SHORTEST k GROUPS`) parse. `shortestpath()` , `allshortestpaths()` and `dijkstra()` cover shortest-path search.
16.12	`<simplified path pattern expression>`	No	The `-/ … /->` form is a syntax error.
16.15	`<group by clause>`	No	An explicit `GROUP BY` is a syntax error in Cypher. Grouping is implicit: the non-aggregated items of a `RETURN` or `WITH` are the grouping keys. SQL `GROUP BY` is available in SQL.
17.1–17.8	`<schema reference>` and the other object references	No	An object is named by a PostgreSQL schema-qualified name. The standard's directory paths, `/dir/schema/graph` , do not parse.
18.1–18.10	`<nested graph type specification>` and the other type elements	No	None.
19.6	`<value type predicate>`	No	`IS TYPED` and `IS ::` are syntax errors. Cast and compare, or use `pg_typeof` .
19.7	`<normalized predicate>`	No	None.
19.8	`<directed predicate>`	No	Every edge is directed.
19.9	`<labeled predicate>`	No	`IS LABELED` is a syntax error, and so is `n:Label` in an expression. Write `label(n) = 'person'` .
19.10	`<source/destination predicate>`	No	`IS SOURCE OF` and `IS DESTINATION OF` are syntax errors. Compare identities: `id(a) = id(startNode(r))` .
19.11	`<all_different predicate>`	No	Compare identities pairwise.

© 2026 SKAI Worldwide Co., Ltd.

Subclause	Name	Support	Comment
19.12	`<same predicate>`	No	`id(a) = id(b)` .
19.13	`<property_exists predicate>`	No	`PROPERTY_EXISTS(n, key)` is a syntax error. Write `exists(n.key)` .
20.4	`<dynamic parameter specification>`	No	The driver's parameter placeholder, `$1` , takes its place.
20.5	`<let value expression>`	No	`LET … IN` inside an expression is a syntax error. The `LET` statement, 14.7, is supported.
20.10	`<element_id function>`	No	`element_id()` does not exist. `id()` returns the identity, as a `graphid` .
20.14	`<path value constructor>`	No	A path comes from a pattern, not from a constructor.
20.16	`<list value function>`	No	`trim(list, n)` and `elements()` do not exist. `head` , `last` , `tail` , `size` and `cardinality` do.
20.25	`<byte string function>`	No	PostgreSQL's `bytea` functions are reachable from SQL.
20.28	`<duration value expression>`	No	PostgreSQL's `interval` type covers durations, but only from SQL: `INTERVAL '1 day'` is a syntax error in Cypher.
20.29	`<duration value function>`	No	`duration()` does not exist.
21.4	`<GQL terminal character>`	No	The lexer is PostgreSQL's, extended for Cypher; it does not implement the standard's character set rules.
22.1–22.20	Additional common rules	No	These subclauses define the standard's evaluation machinery rather than syntax. AgensGraph evaluates a query through the PostgreSQL planner and executor, so they are not implemented as written.
23.1	GQLSTATUS	No	An error carries a SQLSTATE. See **Error codes**.
23.2	Diagnostic records	No	Diagnostics come from PostgreSQL's error report fields.

Extensions beyond the standard

AgensGraph accepts a good deal that ISO/IEC 39075 does not define. Some of it is openCypher, which the product supported first. Some of it exists because a graph here is stored in PostgreSQL and both halves are addressable.

© 2026 SKAI Worldwide Co., Ltd.

Extension	What it adds
A SQL statement over a Cypher subquery, and Cypher over a label table	A Cypher query is a table expression and a label is a table, so one statement can join graph and relational data. See **SQL and Cypher**.
`LOAD FROM table AS r CREATE ( …)`	Creates graph elements from the rows of a relation, including a foreign table. See **Loading data**.
Label inheritance	`CREATE VLABEL child INHERITS (parent)` makes a hierarchy of labels, and a match on the parent also returns the children. `MATCH (n:parent ONLY)` restricts the match to the parent. See **Label inheritance**.
`SET n += map`	Merges a map into the properties instead of replacing them.
`dijkstra()`, `shortestpath()`, `allshortestpaths()`	Weighted and unweighted shortest-path search, in place of the standard's path search prefixes.
Promoted properties	`ALTER VLABEL movie ADD COLUMN year int GENERATED` gives a property a typed, indexable column, and a property read resolves to it. See **`ALTER VLABEL`**.
Graph DDL	`CREATE VLABEL`, `CREATE ELABEL`, `CREATE PROPERTY INDEX`, `CREATE CONSTRAINT`, `REINDEX VLABEL` and `ALTER VLABEL`. See **Data definition**.
openCypher clauses	`UNWIND`, `MERGE` with `ON CREATE` and `ON MATCH`, and `WITH`.
`COUNT { }`, `COLLECT { }`, `ARRAY { }`, `x IN { }`	Four subquery value expressions the standard has no form for — 20.9 defines only `COUNT(*)`, `COUNT(expr)` and the `COLLECT_LIST` aggregate over rows. `COUNT { }` and `COLLECT { }` come from openCypher, `ARRAY { }` from SQL's `ARRAY(subquery)`, and `x IN { }` from SQL's `IN (subquery)`. See **Subquery value expressions**.
openCypher operators and functions	`STARTS WITH`, `ENDS WITH`, `CONTAINS`, `=~`, list comprehensions, `all` / `any` / `none` / `single`, `keys`, `properties`, `id`, `label`, `labels`, `type`, `startNode`, `endNode`, `nodes`, `relationships`, `length`, `range`, `head`, `last`, `tail`, `toString`, `toInteger`, `toFloat`.
PostgreSQL's whole surface	Every SQL type, function, index method, extension and setting is available to a graph query, which is how vector and full-text search work. See **vector search**.

See also

- **Cypher, GQL and SQL** — how the three languages relate in one statement
- **Cypher compatibility** — where AgensGraph differs from other Cypher implementations
- **Keywords** — the words the standard's new statements reserved
- **AgensGraph 2.18.6.1 release notes** — the GQL statements this release added

© 2026 SKAI Worldwide Co., Ltd.

Cypher compatibility

AgensGraph's Cypher is close enough to other implementations that most queries move across unchanged. It is different enough that a handful fail, or, worse, succeed with a different answer. This appendix lists those differences, with the error the server raises or the result it returns, so you can recognize one when a ported query misbehaves.

Every example runs against **the example dataset** with `graph_path` set to `moviekb`.

Two structural facts explain most of what follows. A vertex has exactly one label, because a label is a table and a row lives in one table. And a Cypher query is a PostgreSQL query, so the lexer, the type system and the error codes are PostgreSQL's.

Constructs that do not exist

Several types on one relationship pattern

A relationship pattern takes at most one type:

```
MATCH (p:person)-[r:acted_in|directed]->(m:movie) RETURN count(r);
```

```
ERROR:  multiple types for relationship not supported
```

The same error comes back with a variable length (`-[r:acted_in|directed*1..2]->`) and inside `shortestpath()` . The `-[r:acted_in|:directed]->` spelling is a plain syntax error. Match an untyped edge and test the label instead:

```
MATCH (p:person)-[r]->(m:movie)
WHERE label(r) IN ['acted_in','directed']
RETURN count(r);
```

```
 count
-------
 46
```

Node label alternation fails the same way. `MATCH (n:person|movie)` is a syntax error at the `|` , and so are the other label-expression operators the GQL standard defines: `:person&movie` , `:!person` and `:%` .

A label set on a vertex

There is no set of labels and no way to change the one a vertex has. Each of these is a syntax error at the second colon:

```
MATCH (n:person {name:'Keanu Reeves'}) SET n:actor RETURN n;
MATCH (n:person {name:'Keanu Reeves'}) REMOVE n:actor RETURN n;
MATCH (n:person:movie) RETURN count(n);
CREATE (:aa:bb);
```

```
ERROR:  syntax error at or near ":"
```

Label inheritance is the substitute. `CREATE VLABEL actor INHERITS (person)` makes a separate label that a match on the parent also returns. So `MATCH (n:person)` finds actors too, and `MATCH (n:person ONLY)` excludes them. A vertex is created as an actor rather than promoted to one. See **Label inheritance**.

© 2026 SKAI Worldwide Co., Ltd.

FOREACH

```
FOREACH (x IN [1,2] | CREATE (:zz {v:x}));
```

```
ERROR:  syntax error at or near "FOREACH"
```

`UNWIND` and the GQL `FOR` statement both iterate a list:

```
UNWIND [1,2] AS x RETURN x;
```

```
 x
---
 1
 2
```

A write clause may follow either of them, so `UNWIND list AS x CREATE (…)` covers what a `FOREACH` body carries.

CREATE UNIQUE

```
MATCH (a:person {name:'Keanu Reeves'}), (b:movie {title:'Avatar'})
CREATE UNIQUE (a)-[:acted_in]->(b);
```

```
ERROR:  syntax error at or near "UNIQUE"
```

Use `MERGE`, which matches the pattern and creates it only if no match exists. Here the edge already exists, so `MERGE` returns the existing one rather than adding a second:

```
MATCH (a:person {name:'Sam Worthington'}), (b:movie {title:'Avatar'})
MERGE (a)-[r:acted_in]->(b)
RETURN id(r);
```

```
  id
------
 5.21
```

LOAD CSV

```
LOAD CSV WITH HEADERS FROM 'file:///tmp/movies.csv' AS row RETURN row;
```

```
ERROR:  syntax error at or near "CSV"
```

The import route goes through a table. `COPY` or a foreign table gets the CSV into a relation, and `LOAD FROM` turns its rows into graph elements:

```
CREATE TABLE people_src (name text, age int);
INSERT INTO people_src VALUES ('Ann', 30), ('Bo', 41);
```


© 2026 SKAI Worldwide Co., Ltd.

```
LOAD FROM people_src AS r CREATE (:person {name: r.name, age: r.age});
```

```
UPDATE 2
```

```
DROP TABLE people_src;
```

See **Loading data** for the whole procedure, including the foreign-table wrapper.

The anonymous arrows `-->` and `--`

`--` starts a comment, as it does in SQL, so an anonymous arrow silently ends the statement:

```
MATCH (a:movie)-->(b) RETURN count(*);
```

```
ERROR:  Cypher query must end with RETURN, FINISH or update clause
```

`<--` fails differently, because the `<` is consumed before the comment starts:

```
MATCH (a:movie)<--(b) RETURN count(*);
```

```
ERROR:  syntax error at end of input
```

Write the bracketed forms. `-[]->` , `<-[]-` and `-[]-` all work:

```
MATCH (a:movie)-[]->(b) RETURN count(*);
```

```
 count
-------
 11
```

Map projection

```
MATCH (m:movie {title:'Avatar'}) RETURN m {.title, .year};
```

```
ERROR:  syntax error at or near "{"
```

Build the map explicitly. A map constructor is an ordinary expression:

```
MATCH (m:movie {title:'Avatar'}) RETURN {title: m.title, year: m.year};
```

```
            ?column?
---------------------------------
 {"year": 2009, "title": "Avatar"}
```

`properties(m)` returns the whole map, which covers `m {.*}` .

© 2026 SKAI Worldwide Co., Ltd.

Pattern comprehension

```
MATCH (m:movie {title:'Avatar'}) RETURN [(m)<-[:acted_in]-(p) | p.name];
```

```
ERROR:  syntax error at or near ":"
```

A subquery expression does the same job, and reads more like the rest of the language:

```
MATCH (m:movie {title:'Avatar'})
RETURN COLLECT { MATCH (m)<-[:acted_in]-(p) RETURN p.name };
```

```
       array
--------------------
 ["Sam Worthington"]
```

`reduce()`, `extract()` and `filter()`

None of the three exists.

```
RETURN reduce(acc = 0, x IN [1,2,3] | acc + x);
```

```
ERROR:  variable does not exist
```

`extract()` and `filter()` are syntax errors instead, because `extract` and `filter` are already taken. `extract` is taken by SQL's `EXTRACT(field FROM source)`, and `filter` by the GQL `FILTER` statement.

List comprehensions cover `extract()` and `filter()` and all three of their spellings work:

```
RETURN [x IN [1,2,3] | x * 10], [x IN [1,2,3] WHERE x > 1], [x IN [1,2,3] WHERE x > 1 | x *
10];
```

```
   ?column?   | ?column? | ?column?
--------------+----------+----------
 [10, 20, 30] | [2, 3]   | [20, 30]
```

For a fold, aggregate in SQL over the Cypher result, or use `sum()` where the fold is a sum.

A bare pattern as a predicate

A pattern is not a boolean expression:

```
MATCH (m:movie) WHERE (m)<-[:acted_in]-() RETURN count(m);
```

```
ERROR:  syntax error at or near ":"
```

Wrap it in `exists()`, or in the GQL `EXISTS { }` subquery expression:

```
MATCH (m:movie) WHERE exists((m)<-[:acted_in]-()) RETURN count(m);
```


© 2026 SKAI Worldwide Co., Ltd.

```
 count
-------
 18
```

Backtick-quoted identifiers

A backtick is not a quote character:

```
MATCH (m:`movie`) RETURN count(m);
```

```
ERROR:  syntax error at or near "`"
```

Identifiers are quoted the SQL way, with double quotes: `MATCH (m:"movie")`. This matters for the words that became reserved in 2.18. See **Keywords**.

Temporal, spatial and geodistance constructors

`duration()`, `distance()` and `haversin()` do not exist:

```
RETURN haversin(1.0);
```

```
ERROR:  function haversin(numeric) does not exist
HINT:  No function matches the given name and argument types. You might need to add explicit t
ype casts.
```

`point()` and `timestamp()` fail more confusingly, because each collides with a PostgreSQL name. `point({x:1, y:2})` finds PostgreSQL's geometric `point`, whose overloads take pairs of floats and other geometries, so the call is ambiguous rather than absent:

```
RETURN point({x:1, y:2});
```

```
ERROR:  function point(jsonb) is not unique
HINT:  Could not choose a best candidate function. You might need to add explicit type casts.
```

`timestamp()` is a syntax error, because `timestamp` is a type name.

`datetime()` exists but is a different function, and it is the one to watch. It takes a time zone or a list of date parts, never an ISO-8601 string. Handing it a timestamp string reads that string as a time zone:

```
RETURN datetime('2020-01-01T00:00:00');
```

```
ERROR:  time zone "2020-01-01T00:00:00" not recognized
```

It does accept a zone name, or nine date parts:

```
RETURN datetime(2020, 1, 1, 0, 0, 0, 0, 0, 'UTC');
```


© 2026 SKAI Worldwide Co., Ltd.

```
      datetime
----------------------
 "2020-01-01T00:00:00"
```

For a timestamp from a string, cast: `'2020-01-01'::date`, `'2020-01-01 00:00'::timestamptz`. `CURRENT_DATE` and `CURRENT_TIMESTAMP` work as bare keywords. Durations are PostgreSQL intervals, reachable with `::interval` from SQL. Spatial work goes through a PostgreSQL extension, reached from a graph query the same way vector search is. See **vector search**.

The `db.*` procedures

There is no `db` namespace:

```
CALL db.labels();
```

```
ERROR:  schema "db" does not exist
```

The catalogs answer the same questions. `ag_label` lists the labels of a graph. See **Graph catalogs**.

PROFILE

```
PROFILE MATCH (m:movie) RETURN count(m);
```

```
ERROR:  syntax error at or near "PROFILE"
```

`EXPLAIN` works, with PostgreSQL's options: `EXPLAIN (ANALYZE, BUFFERS) MATCH …`.

Behaviour that differs

Every column a RETURN produces is jsonb

A property read is a `jsonb` value, and it stays one through the projection. A string therefore prints with its JSON quotes:

```
MATCH (m:movie {title:'Avatar'}) RETURN m.title, pg_typeof(m.title);
```

```
  title   | pg_typeof
----------+-----------
 "Avatar" | "jsonb"
```

`::text` does not remove the quotes. It renders the jsonb value, quotes included. To get a plain scalar, extract it in SQL around the Cypher:

```
SELECT t #>> '{}' AS title
FROM (MATCH (m:movie) RETURN m.title AS t) q
ORDER BY title LIMIT 1;
```

```
  title
---------
 Avatar
```


© 2026 SKAI Worldwide Co., Ltd.

See SQL and Cypher.

`ORDER BY 2` sorts nothing

An integer in a sort key is a constant, not an output-column number. Sorting every row by the same constant leaves the order untouched, and nothing warns you:

```
MATCH (m:movie) RETURN m.title AS t, m.year AS y ORDER BY 2 LIMIT 5;
```

```
             t              |  y
----------------------------+------
 "The Matrix Resurrections" | 2021
 "Inception"                | 2010
 "Interstellar"             | 2014
 "Avatar"                   | 2009
 "Blade Runner"             | 1982
```

Name the key. `ORDER BY y` sorts as expected. The integer form does work in SQL, where an integer sort key means the output column at that position. So wrapping the Cypher in a `SELECT … ORDER BY 2` sorts too.

```
MATCH (m:movie) RETURN m.title AS t, m.year AS y ORDER BY y LIMIT 5;
```

```
              t               |  y
------------------------------+------
 "Blade Runner"               | 1982
 "The Terminator"             | 1984
 "Total Recall"               | 1990
 "Terminator 2: Judgment Day" | 1991
 "Jurassic Park"              | 1993
```

`n.id` is not the identity

`id` is an ordinary property name. Unless something wrote one, it is absent:

```
MATCH (m:movie {title:'Avatar'}) RETURN m.id, id(m);
```

```
 id | id
----+-----
    | 3.7
```

`id(m)` returns the identity, a `graphid` printed as labid `.` locid. See Limits for what those two halves hold.

A bare `CALL { }` answers a different question

A subquery body does not see the outer variables. The name is rebound to a fresh, unconstrained variable, so the body runs over the whole graph rather than the current row:

```
MATCH (d:person)-[:directed]->(m:movie)
CALL { MATCH (a:person)-[:acted_in]->(m) RETURN count(a) AS cast_size }
RETURN m.title AS film, cast_size ORDER BY film LIMIT 3;
```

```
        film         | cast_size
---------------------+-----------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Avatar"            | 25
 "Blade Runner"      | 25
 "Blade Runner 2049" | 25
```

25 is every `acted_in` edge in the graph. With an import list the same query is correct:

```
MATCH (d:person)-[:directed]->(m:movie)
CALL (m) { MATCH (a:person)-[:acted_in]->(m) RETURN count(a) AS cast_size }
RETURN m.title AS film, cast_size ORDER BY film LIMIT 3;
```

```
        film         | cast_size
---------------------+-----------
 "Avatar"            | 1
 "Blade Runner"      | 1
 "Blade Runner 2049" | 2
```

IMPORTANT

Always give a `CALL` subquery an import list — `CALL (v) { … }`, `CALL (*) { … }` or `CALL { WITH v … }`. All three are correct and agree. The bare form is the one shape of this divergence that answers a different question rather than raising an error.

`shortestpath` endpoints must be named variables

An inline pattern in an endpoint is refused:

```
MATCH p = shortestpath((:movie {title:'The Matrix'})-[:related_to*..4]->(:movie
{title:'Inception'}))
RETURN length(p);
```

```
ERROR:  node must be a reference to a specific node
```

Bind both ends first:

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'Inception'})
MATCH p = shortestpath((a)-[:related_to*..4]->(b))
RETURN length(p);
```

```
 length
--------
 2
```

The same rule applies to `allshortestpaths()` and `dijkstra()`.

An unquoted label folds to lower case

Identifiers follow SQL's folding rule, so a label written in mixed case is looked up in lower case:

```
MATCH (m:Movie) RETURN count(m);
```


© 2026 SKAI Worldwide Co., Ltd.

```
 count
-------
 18
```

`Movie`, `MOVIE` and `movie` are the same label. A label created with a quoted mixed-case name has to be quoted at every use.

A condition takes only a boolean

Reading a property for truthiness raises a cast error rather than testing for non-empty:

```
MATCH (m:movie {year: 1999}) WHERE m.title RETURN count(m);
```

```
ERROR:  cannot cast "The Matrix" (jsonb type string) to boolean
```

The message names the first value the executor reached, so an unconstrained `MATCH` over several rows reports whichever one it happened to read first.

This holds in `WHERE`, `FILTER`, `AND`, `OR`, `NOT`, `XOR`, `CASE WHEN` and `dijkstra()`'s qualifier. Write the test you mean, as in `WHERE m.title IS NOT NULL` or `WHERE m.count <> 0`. Or cast explicitly with `::boolean`.

A missing label is not an error

A pattern naming a label that does not exist matches nothing and returns a row count of zero:

```
MATCH (n:no_such_label) RETURN count(n);
```

```
 count
-------
 0
```

A typo in a label name therefore looks like an empty result. Check the label list in `ag_label` when a query returns nothing you expected.

`size()` and `length()` are the same function

`size()` is another spelling of `length()`, and the output column is named `length` whichever you write:

```
RETURN size([1,2,3]), size('abcd'), length('abcd');
```

```
 length | length | length
--------+--------+--------
 3      | 4      | 4
```

`size()` over a pattern counts matches, and names its column `count`:

```
MATCH (m:movie {title:'Avatar'}) RETURN size((m)<-[:acted_in]-());
```

```
 count
-------
 1
```


© 2026 SKAI Worldwide Co., Ltd.

`labels()` always returns one label

`labels()` exists and returns a list, but the list has exactly one element, because a vertex has one label:

```
MATCH (n:person) RETURN labels(n) LIMIT 1;
```

```
   labels
------------
 ["person"]
```

`label(n)` returns that one label as a scalar and is the form to prefer. `type(r)` and `label(r)` both work on an edge.

`collect()` over a graph element returns a native array

Collecting vertices gives a `vertex[]`, not a list of pictures, so the collected elements stay usable:

```
MATCH (m:movie) RETURN pg_typeof(collect(m));
```

```
 pg_typeof
------------
 "vertex[]"
```

```
MATCH (m:movie) WITH collect(m) AS ms RETURN ms[0].title;
```

```
          title
-----------------------------
 "The Matrix Resurrections"
```

Collecting a property still returns a jsonb list.

A vertex with edges will not delete

`DELETE` refuses to leave a dangling edge:

```
MATCH (m:movie {title:'Avatar'}) DELETE m;
```

```
ERROR:  Vertex with connected edges cannot be removed
HINT:  Use DETACH DELETE or remove connected edges before deleting the vertex.
```

A query needs a graph to run against

There is no default graph. With `graph_path` unset, every graph statement is refused:

```
RESET graph_path;
MATCH (m:movie) RETURN count(m);
```

```
ERROR:  graph_path is NULL
HINT:  Use SET graph_path
```


© 2026 SKAI Worldwide Co., Ltd.

Set it for the session, or set it in `postgresql.conf` or per role. See **Configuration**:

```
SET graph_path = moviekb;
```

AgensGraph additions

These have no counterpart in other Cypher implementations, so a query using them does not port outward.

`MATCH (n:label ONLY)` restricts a match to one label, excluding the labels that inherit from it. Without `ONLY` a match on a parent label returns its children too.

`SET n += map` merges a map into the properties rather than replacing them:

```
MATCH (n:person {name:'Keanu Reeves'}) SET n += {city:'Seoul'} RETURN properties(n);
```

```
                       properties
-------------------------------------------------------
 {"born": 1964, "city": "Seoul", "name": "Keanu Reeves"}
```

`LOAD FROM table AS r CREATE (…)` creates graph elements from the rows of a relation, which is how bulk loading works.

`dijkstra((a)-[e:related_to]->(b), e.weight)` finds a weighted shortest path. It respects the weights, so the cheapest route need not be the shortest one in hops:

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'Inception'})
MATCH p = dijkstra((a)-[e:related_to]->(b), e.weight)
RETURN [x IN nodes(p) | x.title];
```

```
                  ?column?
--------------------------------------------
 ["The Matrix", "Total Recall", "Inception"]
```

A Cypher query is a table expression and a label is a table, so SQL and Cypher can be mixed in one statement:

```
SELECT q.title #>> '{}' AS film, r.score
FROM (MATCH (p:person)-[:directed]->(m:movie)
      WHERE p.name = 'Christopher Nolan'
      RETURN m.title AS title) q
JOIN rating r ON r.title = q.title #>> '{}'
WHERE r.source = 'critics'
ORDER BY r.score DESC;
```

```
     film     | score
--------------+-------
 Inception    |   8.8
 Interstellar |   8.7
```

See also

- **GQL conformance** — which parts of ISO/IEC 39075 are supported
- **Keywords** — the words 2.18 reserved, and what breaks
- **SQL and Cypher** — the crossing, in full

© 2026 SKAI Worldwide Co., Ltd.

- Cypher, GQL and SQL — how the three languages relate

Keywords

AgensGraph inherits PostgreSQL's keyword list and adds the words its graph statements need. A keyword's category decides where the word can still be used as a name. An **unreserved** word may name anything. A **reserved** word may not, and has to be double-quoted first.

PostgreSQL's Key Words appendix lists the inherited SQL words with their categories. This page covers only what AgensGraph adds and changes. Two categories PostgreSQL uses, type/function name and column name, have no AgensGraph entries. Every word below is either unreserved or fully reserved.

Identifiers are quoted the SQL way, with double quotes. A backtick is a syntax error, so `` `let` `` does not work as an escape — write `"let"`.

Words AgensGraph adds

These twenty-four words are in AgensGraph's keyword list and not in PostgreSQL 18's. The comment names the clause or statement the word belongs to.

Word	Category	Comment
`allshortestpaths`	Unreserved	Path function: every shortest path between two vertices.
`assert`	Unreserved	`CREATE CONSTRAINT … ASSERT`.
`collect`	Unreserved	The `COLLECT { }` subquery expression.
`contains`	Unreserved	The `CONTAINS` string predicate.
`count`	Unreserved	The `COUNT { }` subquery expression.
`current_graph`	**Reserved**	The working graph, as a bare keyword.
`current_property_graph`	**Reserved**	Synonym for `CURRENT_GRAPH`.
`dijkstra`	Unreserved	Path function: weighted shortest path.
`elabel`	Unreserved	`CREATE ELABEL`, `ALTER ELABEL`, `DROP ELABEL`.
`ends`	Unreserved	The `ENDS WITH` string predicate.
`finish`	**Reserved**	The `FINISH` clause, which ends a query that projects nothing.
`graph`	Unreserved	`CREATE GRAPH`, `DROP GRAPH`, and the `graph_path` setting's statements.
`let`	**Reserved**	The `LET` clause.

© 2026 SKAI Worldwide Co., Ltd.

Word	Category	Comment
`optional`	Unreserved	`OPTIONAL MATCH` and `OPTIONAL CALL`.
`property`	Unreserved	`CREATE PROPERTY INDEX`, `CURRENT_PROPERTY_GRAPH`.
`remove`	Unreserved	The `REMOVE` clause.
`shortestpath`	Unreserved	Path function: one shortest path.
`single`	Unreserved	The `single()` list predicate.
`size`	Unreserved	The `size()` function.
`starts`	Unreserved	The `STARTS WITH` string predicate.
`unwind`	**Reserved**	The `UNWIND` clause.
`vlabel`	Unreserved	`CREATE VLABEL`, `ALTER VLABEL`, `DROP VLABEL`.
`xor`	**Reserved**	The `XOR` boolean operator.
`yield`	Unreserved	`CALL … YIELD`.

Four words AgensGraph inherits are reserved here and unreserved in PostgreSQL, because each one leads a Cypher clause: `insert`, `match`, `merge` and `return`.

`CALL` was deliberately left unreserved, so `call` still names a column, table, alias, label, property key or variable. Exactly two spellings change meaning, because the lexer treats `CALL` specially when a `(` or a `{` follows it:

- `call(...)` — a function literally named `call`, invoked with arguments. Quote it: `"call"(1)`.
- `(call { ... })` — a node pattern whose variable is named `call` and which carries a property map but no label. Quote it: `("call" { ... })`.

Words that became reserved in 2.18

Six words are reserved in 2.18 and were not in 2.17: `insert`, `finish`, `let`, `xor`, `current_graph` and `current_property_graph`. Reserving a word can break an application that used it as a name, so each was made as narrow as the grammar allowed. The result is not uniform. Some positions still work, and the table below is the measured outcome rather than the general rule.

Each cell is the result of running the position against a 2.18 server. **Works** means the unquoted word was accepted.

Word	Label in a pattern	`CREATE VLABEL`	Variable	`RETURN … AS` alias	Alias as a sort key	Property key
`insert`	Works	Fails	Works	Works	Works	Works

© 2026 SKAI Worldwide Co., Ltd.

Word	Label in a pattern	`CREATE VLABEL`	Variable	`RETURN … AS` alias	Alias as a sort key	Property key
`finish`	Works	Fails	Fails	Works	Fails	Works
`let`	Fails	Fails	Fails	Works	Fails	Works
`xor`	Fails	Fails	Fails	Works	Fails	Works
`current_graph`	Fails	Fails	Fails	Works	Works	Works
`current_property_graph`	Fails	Fails	Fails	Works	Works	Works

Two columns in that table read oddly, and both need stating in words.

For `insert` and `finish`, a label works in a pattern but not in the label DDL. `CREATE (:insert)` and `MATCH (n:insert)` are accepted. `CREATE VLABEL insert` is not:

```
CREATE VLABEL insert;
```

```
ERROR:  syntax error at or near "insert"
```

Create the label with a quoted name, `CREATE VLABEL "insert"`. The unquoted pattern spelling still finds it afterwards, because an unquoted identifier folds to lower case.

An alias always works, but sorting by that alias does not always. `RETURN n.k AS let` is accepted and the `ORDER BY let` after it is not:

```
MATCH (n:t) RETURN n.k AS let ORDER BY let;
```

```
ERROR:  syntax error at or near "let"
```

`current_graph` and `current_property_graph` are the exception. An alias of either name sorts.

Double-quoting the name fixes every failing cell, at both its definition and every use:

```
MATCH (n:t) RETURN n.k AS "let" ORDER BY "let";
```

A property key never needs quoting. A key is a string inside a `jsonb` document, not an identifier, so `CREATE (:t {let: 1})` and `RETURN n.let` are unaffected by any of this.

On the SQL side the same six words need quoting as column names. `CREATE TABLE t (xor int)` is a syntax error. `CREATE TABLE t ("xor" int)` works.

The reserved set is wider than the graph words

Reserving a word for a graph clause is only part of the picture. AgensGraph inherits PostgreSQL's reserved words too. A Cypher label or variable is an identifier in the same grammar, so any of them is refused unquoted. `both` is a good example: a PostgreSQL reserved word with nothing to do with graphs:

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE VLABEL both;
```

```
ERROR:  syntax error at or near "both"
```

The same is true of `all`, `and`, `any`, `array`, `as`, `case`, `cast`, `end`, `from`, `in`, `not`, `null`, `or`, `select`, `table`, `to`, `union`, `when`, `where` and the rest of PostgreSQL's reserved list. Before choosing a label, variable or property-index name, check it against PostgreSQL's **Key Words** appendix, which gives the category of every SQL word AgensGraph carries.

> NOTE
>
> The keyword list is a single file in the AgensGraph source, one line per word with its category. Consult it when a name is refused and this page does not say why.

See also

- **Syntax** — the lexical rules, including identifier folding
- **Cypher compatibility** — the other differences a ported query hits
- **Upgrading to 2.18** — the full pre-upgrade checklist
- **AgensGraph 2.18.6.1 release notes** — the clauses these words serve

Error codes

Every message the AgensGraph server produces carries a five-character SQLSTATE code, exactly as PostgreSQL does. The first two characters name the error class, and the last three the specific condition within it. So an application that does not recognize a code can still act on its class.

AgensGraph defines no error codes of its own. Its error-code list is byte-identical to PostgreSQL 18's, and a graph statement that fails raises one of PostgreSQL's existing codes. The complete list of codes, with the condition name PL/pgSQL uses for each, is PostgreSQL's **Error Codes** appendix. It is not reproduced here.

For a client, the practical consequence is that a graph-specific failure cannot be recognized by its code. `multiple types for relationship not supported` and `Cypher query must end with RETURN, FINISH or update clause` arrive as `0A000` and `42601`. Those are the same codes an unsupported SQL feature and a SQL syntax error use. Match on the message text when you need to tell them apart, and treat the class as the reliable part.

Reading a code

Run `\set VERBOSITY verbose` in `psql` and every error afterwards carries its SQLSTATE and the source location that raised it:

```
\set VERBOSITY verbose
```

```
MATCH (m:movie {year: 1999}) WHERE m.title RETURN count(m);
```

```
ERROR:  42846: cannot cast "The Matrix" (jsonb type string) to boolean
LOCATION:  ExecEvalCypherTypeCast, execExprInterp.c:5900
```


© 2026 SKAI Worldwide Co., Ltd.

Setting `VERBOSITY` to `sqlstate` prints the code alone. From a driver, the code is a field of the error object: `sqlstate` in most PostgreSQL clients.

Codes a graph statement commonly raises

These are the codes to expect from graph work, with a real message for each. The condition name is the one PL/pgSQL and most drivers use.

Code	Condition name	Raised by, for example
`0A000`	`feature_not_supported`	`multiple types for relationship not supported` — a relationship pattern with more than one type.
`22023`	`invalid_parameter_value`	`labid out of range: 65536` — a `graphid` outside the range in **Limits**.
`3F000`	`invalid_schema_name`	`graph_path is NULL` — no working graph is set. A graph is a schema, which is why the class is this one.
`42601`	`syntax_error`	`Cypher query must end with RETURN, FINISH or update clause`, and every parse failure.
`42703`	`undefined_column`	`variable does not exist` — a name that is not in scope.
`42712`	`duplicate_alias`	`WITH passes on "t" more than once`, and the other name-collision refusals.
`42846`	`cannot_coerce`	`cannot cast "…" (jsonb type string) to boolean` — a condition given something other than a boolean.
`42P07`	`duplicate_table`	`label "movie" already exists` — a label is a table, so a duplicate label is a duplicate table.
`54000`	`program_limit_exceeded`	`no more new labels are available` — the label ids of a graph are exhausted.
`55000`	`object_not_in_prerequisite_state`	`Vertex with connected edges cannot be removed` — a `DELETE` that would leave a dangling edge.

Three of those rows show the pattern to expect. A graph is a schema, a label is a table, and a property is a `jsonb` value. So a graph error is reported in the class of the PostgreSQL object it is really about.

See also

- **Cypher compatibility** — the errors a query ported from another Cypher hits, with their text
- **Limits** — the two limits that produce `22023` and `54000`
- **Glossary** — the terms these messages use

© 2026 SKAI Worldwide Co., Ltd.

Limits

Two of AgensGraph's limits come from the identity it gives every vertex and edge. The rest are PostgreSQL's, because a graph is a schema and a label is a table.

Graph limits

Every vertex and edge carries an identity of type `graphid`, and its layout sets the two limits a graph can reach. A `graphid` is a 64-bit unsigned integer split in two. The top 16 bits are the **label id**, and the low 48 bits are the **local id**, the row's serial number within its label. An identity therefore prints as two numbers. `movie[3.7]` is label 3, row 7.

Limit	Value	Comes from
Labels per graph	65,535	The 16-bit label id. Zero is reserved as the invalid label id, so the usable range is 1 to 65,535.
Rows per label	281,474,976,710,655	The 48-bit local id: $2^{48} - 1$.
Label name length	63 bytes	PostgreSQL's identifier length.

A graph itself has no separate limit: a graph is a schema, and PostgreSQL sets no limit on schemas per database. A large number of graphs consumes relations, since every graph creates a sequence and two root labels of its own.

Both halves of a `graphid` are range-checked. The constructor refuses either one out of range:

```
SELECT graphid(65536, 1);
```

```
ERROR:  labid out of range: 65536
```

```
SELECT graphid(1, 281474976710656);
```

```
ERROR:  locid out of range: 281474976710656
```

The same two messages, without the offending value, come back when a `graphid` literal is parsed: `SELECT '65536.1'::graphid` raises `labid out of range`.

The counters that hand out these numbers are ordinary sequences, so you can inspect them. Each graph has one `ag_label_seq` for label ids and each label an `<label>_id_seq` for local ids:

```
SELECT sequencename, max_value FROM pg_sequences
WHERE schemaname = 'moviekb' AND sequencename IN ('ag_label_seq', 'movie_id_seq');
```

```
 sequencename |    max_value
--------------+-----------------
 ag_label_seq |           65535
 movie_id_seq | 281474976710655
```


© 2026 SKAI Worldwide Co., Ltd.

`ag_label_seq` cycles, and label creation skips a number already in use, so dropping a label frees its id for reuse. When no free id is left, creating a label fails:

```
ERROR:  no more new labels are available
```

Two labels arrive in every new graph: `ag_vertex` with label id 1 and `ag_edge` with label id 2. They are the two roots every vertex and edge label inherits from. So 65,533 ids are available to your own labels.

A label name over 63 bytes is truncated

An identifier is 63 bytes. A longer label name is cut to 63 bytes with a notice. So **two label names that share their first 63 bytes are one label**. The second `CREATE VLABEL` sees the first:

```
CREATE VLABEL a_label_name_that_is_deliberately_longer_than_sixty_three_bytes_aaa;
CREATE VLABEL a_label_name_that_is_deliberately_longer_than_sixty_three_bytes_bbb;
```

```
NOTICE:  identifier "a_label_name_that_is_deliberately_longer_than_sixty_three_bytes_aaa" will
be truncated to "a_label_name_that_is_deliberately_longer_than_sixty_three_bytes"
CREATE VLABEL
NOTICE:  identifier "a_label_name_that_is_deliberately_longer_than_sixty_three_bytes_bbb" will
be truncated to "a_label_name_that_is_deliberately_longer_than_sixty_three_bytes"
ERROR:  label "a_label_name_that_is_deliberately_longer_than_sixty_three_bytes" already exists
```

This bites in a pattern, because a pattern creates a label it does not find and nothing fails. Writing to the two names above puts both rows in the same label:

```
CREATE (:a_label_name_that_is_deliberately_longer_than_sixty_three_bytes_aaa {via:'aaa'});
CREATE (:a_label_name_that_is_deliberately_longer_than_sixty_three_bytes_bbb {via:'bbb'});
MATCH (n) RETURN label(n) AS lab, n.via ORDER BY n.via;
```

```
                               lab                                |  via
------------------------------------------------------------------+-------
 "a_label_name_that_is_deliberately_longer_than_sixty_three_bytes" | "aaa"
 "a_label_name_that_is_deliberately_longer_than_sixty_three_bytes" | "bbb"
```

Keep a label name inside 63 bytes, and make the first 63 bytes distinct. The limit is in bytes, not characters, so a name in a multi-byte script reaches it sooner.

PostgreSQL limits a graph runs into

A label is a table, so a graph is subject to the same limits as the rest of the database. These are the ones graph work reaches. PostgreSQL's **Limits** appendix has the full list.

Limit	Value	What it means for a graph
Relation size	32 TB	One label's storage, at the default 8 kB block size. A label past this needs partitioning, or splitting into child labels.
Field size	1 GB	The whole property map of one vertex or edge, since the properties are one `jsonb` field. Long values are compressed and moved out of line first.

© 2026 SKAI Worldwide Co., Ltd.

Limit	Value	What it means for a graph
Columns per relation	1600	Reached only by promoting properties to columns. A vertex label starts with two columns and an edge label with four.
Identifier length	63 bytes	Label names, graph names, property-index names, and variable names. Property keys are not identifiers and are not limited by this.
Rows per table	Bounded by relation size	The 48-bit local id is far larger, so the storage limit is what you hit first.
Indexes per table	Unlimited	Property indexes on one label.

NOTE

Nothing limits how many properties one element may hold, or how deeply a property may nest, short of the 1 GB field size. Reading a property does have a cost that grows with the size of the map, because the whole map is fetched to reach one key. Promoting the key to a column, with `ALTER VLABEL`, is the way out of that.

See also

- How a graph is stored — the storage layout these limits describe
- Label inheritance — splitting one large label into several
- Data definition — the statements that create labels and indexes

Glossary

The vocabulary this manual uses, with a link to the page that treats each term properly. Terms are grouped by what they describe rather than alphabetically, because several of them only make sense next to each other.

The model

property graph

A data model in which the data is a set of vertices connected by edges, and both carry named values. AgensGraph stores one property graph and one set of relational tables in the same database, and one statement may read both. See The property graph model.

graph

A named property graph. It is the container for a set of labels, and the unit `graph_path` selects. A graph is a PostgreSQL schema, created with `CREATE GRAPH`. See How a graph is stored.

© 2026 SKAI Worldwide Co., Ltd.

vertex	One entity in a graph: a person, a film, an account. A vertex has an identity, exactly one label, and a set of properties. It prints as `label[graphid]{properties}`, for example `movie[3.7]{"title": "Avatar"}`. See **The property graph model**.
edge	One directed connection between two vertices, with its own label and properties. An edge cannot exist without both endpoints. Every edge is directed, and a pattern may traverse it in either direction. See **The property graph model**.
property	One named value on a vertex or an edge, such as `title`, `year` or `weight`. Properties are schemaless: two vertices of the same label need not carry the same keys. They live together in one `jsonb` document per element. A property key is not an identifier, so it is unaffected by keyword reservation and by case folding. See **How a graph is stored**.
label	The name that classifies a vertex or an edge, and the unit of storage: a label is a PostgreSQL table. A vertex has exactly one label, and it cannot be changed after the vertex is created. There is no label set. See **How a graph is stored**.
vlabel	A vertex label, the kind created by `CREATE VLABEL`. Its table is `(id graphid, properties jsonb)` and inherits from the graph's `ag_vertex`. See **Data definition**.
elabel	An edge label, the kind created by `CREATE ELABEL`. Its table adds `start` and `end` to a vlabel's columns and inherits from the graph's `ag_edge`. See **Data definition**.
label inheritance	A label may inherit from another, so a match on the parent also returns rows of every child. This is how one entity plays several roles, in place of a label set. `MATCH (n:parent ONLY)` opts out of it. See **Label inheritance**.

Identity and addressing

graphid	The identity of a vertex or an edge, and a first-class type. It is a 64-bit value printed as two numbers, label id `.` local id. So `3.7` is row 7 of label 3. `id(n)` returns it. `n.id` does not, because `id` is only an ordinary property name. See **Limits**.

© 2026 SKAI Worldwide Co., Ltd.

graph_path

The setting that names the graph a statement runs against. There is no default: with `graph_path` unset, every graph statement is refused with `ERROR: graph_path is NULL`. Set it per session with `SET graph_path = name`, or per role or cluster. See **Configuration**.

Querying

path

A sequence of alternating vertices and edges, produced by a pattern rather than by a constructor. `MATCH p = (a)-[e]->(b)` binds one as a value of type `graphpath`, which `nodes()`, `relationships()` and `length()` then read. See **Reading a graph**.

traversal

Following edges from one vertex to reach others. A graph query does this instead of joining on a key. A traversal of unknown depth is written with a quantified edge, `-[r*1..3]->`. See **Reading a graph**.

property index

An index on a property expression, created with `CREATE PROPERTY INDEX ON label (key)`. It is a PostgreSQL index whose key is an expression over the `properties` document, and a plain `CREATE INDEX` cannot express that key. See **How a graph is stored**.

promoted column

A property given a real column of a declared type, generated and stored from the `properties` document. A property read resolves to the column, so an ordinary index on it serves the query and the value keeps its own type. `movie.year` in the example dataset is one. See `CREATE VLABEL` and `ALTER VLABEL`.

The languages

Cypher

The graph query language AgensGraph has always accepted. It is declarative, and it matches by drawing a pattern in ASCII. AgensGraph's dialect is close to other implementations but not identical. See **Cypher, GQL and SQL** and **Cypher compatibility**.

GQL

ISO/IEC 39075:2024, the international standard for graph query languages. Release 2.18 implements a substantial part of it, including the statements `LET`, `NEXT`, `FILTER`, `FINISH`, `INSERT` and `FOR`. GQL is not a separate mode: its statements and Cypher's mix freely in one pipeline. See **GQL conformance**.

© 2026 SKAI Worldwide Co., Ltd.

See also

- **The property graph model** — the model, with worked examples
- **How a graph is stored** — how each term above is stored
- **The example dataset** — the graph every example uses

© 2026 SKAI Worldwide Co., Ltd.