SKAI WORLDWIDE

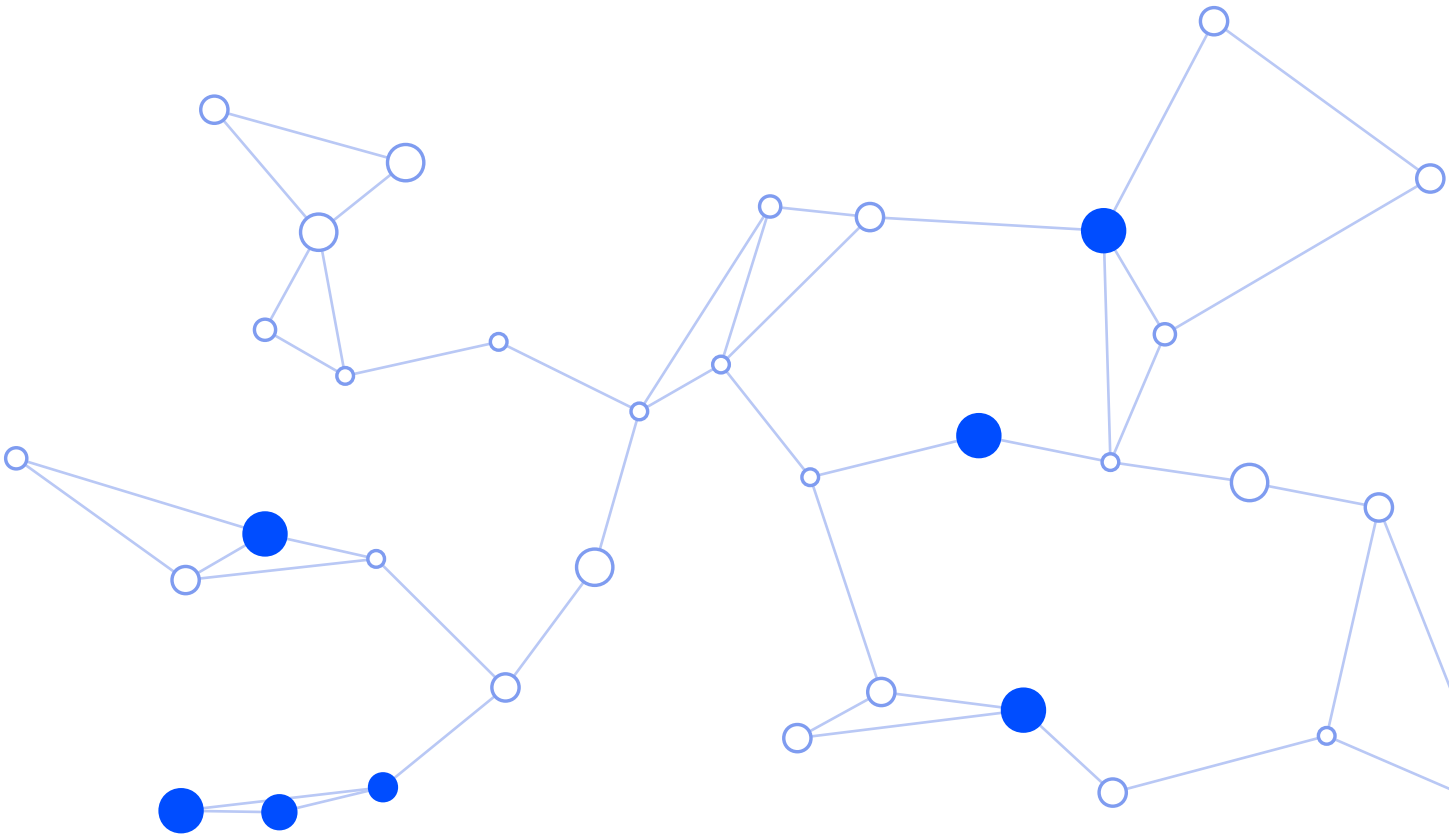
TECHNICAL DOCUMENTATION

AgensGraph 2.18

Documentation

PostgreSQL 위의 멀티모델 데이터베이스. 하나의 그래프를 Cypher, GQL, SQL로 질의합니다.

릴리스 2.18.6.1



목차

시작하기 **5**
- AgensGraph란 5
- AgensGraph 설치 6
- 예제 데이터셋 9
- 첫 쿼리 15

개념 **20**
- 속성 그래프 모델 20
- 그래프가 어떻게 저장되는가 23
- 레이블 상속 27
- Cypher, GQL, SQL 29

쿼리하기 **32**
- 그래프 읽기 32
- 그래프 쓰기 39
- 쿼리 조합하기 45
- 경로 52
- 인덱스와 제약 58
- 속성을 컬럼으로 승격하기 69
- SQL과 Cypher 섞기 78
- 쿼리 튜닝 85

레퍼런스 **94**
- 구문 94
- 데이터 타입 105
- 절 115
- 데이터 정의 216
- 함수 274
- 서버 설정 337
- agens 클라이언트 351

가이드 **359**
- 데이터 적재 359
- 벡터 검색 363
- 전문 검색 367
- 하이브리드 검색 371
- AI·LLM 연동 375

드라이버와 클라이언트 라이브러리 **376**
- Python 드라이버 376
- 다른 언어 376
- 접속과 매개변수 376

운영 **378**
- 배포 378
- 설정 380
- 보안 383

© 2026 SKAI Worldwide Co., Ltd.

백업과 복구 386
서버 측 도구 389

시스템 카탈로그 392
그래프 카탈로그 392
그래프 뷰 395
그래프 메타데이터 397

릴리스 노트 408
AgensGraph 2.18 408
2.18로 업그레이드하기 412
업그레이드 실습 417
AgensGraph 2.17.0 421

부록 424
GQL 적합성 424
Cypher 호환성 433
키워드 444
오류 코드 447
제한 449
용어집 452

© 2026 SKAI Worldwide Co., Ltd.

AgensGraph 2.18

AgensGraph는 스카이월드와이드(SKAI Worldwide, 구 비트나인)가 Apache License 2.0 오픈소스로 개발·유지하는, PostgreSQL 위에 만들어진 멀티모델 데이터베이스입니다. 데이터를 **속성 그래프**로, 그리고 동시에 관계형 테이블로 저장합니다. 그 데이터를 **Cypher**로, **GQL**(ISO/IEC 39075)로, 또는 **SQL**로 조회하며, 그것이 답에 이르는 가장 짧은 길이라면 한 문장에서 그중 두 언어를 함께 쓸 수 있습니다.

이 매뉴얼은 AgensGraph 2.18.6.1을 다룹니다. 2.18 릴리스는 PostgreSQL 18 위에 만들어졌고 GQL 표준의 상당 부분을 더합니다. 무엇이 바뀌었는지는 **AgensGraph 2.18**이 나열하며, 애플리케이션이 알아야 할 변경도 거기에 있습니다.

이 매뉴얼의 구성

시작하기는 서버를 설치하고, 이 매뉴얼의 모든 예제가 쓰는 작은 영화 그래프를 적재하고, 결과를 하나 얻게 해 줍니다. 이것을 먼저 읽으십시오.

개념은 속성 그래프 모델과, 그래프가 PostgreSQL에 어떻게 놓이는지를 설명합니다. 그래프는 스키마이고 레이블은 테이블입니다. 바로 이 구조 덕분에 문장 하나에서 그래프 작업과 관계형 작업을 섞을 수 있습니다.

쿼리하기는 언어를 만나는 순서대로 가르칩니다. 읽기, 쓰기, 파이프라인 엮기, 경로 따라가기, 인덱싱, 그리고 SQL과 Cypher를 넘나드는 것입니다.

레퍼런스는 같은 언어의 형식적 정의입니다. 어휘 규칙, 타입, 그리고 절·문장·함수마다 한 페이지씩입니다. 무엇을 원하는지는 알고 정확한 구문이 필요할 때 여기를 보십시오.

가이드는 데이터 적재와 검색 기능을 다룹니다. **드라이버와 클라이언트 라이브러리**는 클라이언트 라이브러리를 가리킵니다. **운영**은 서버를 운영하기 위한 것입니다. **시스템 카탈로그**는 그래프를 설명하는 카탈로그를 문서화하고, **부록**에는 GQL 적합성 표, 호환성 안내, 키워드와 오류 코드 목록, 그리고 2.17의 페이지 이름을 여기에 대응시킨 표가 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

다음에 볼 것

시작하기

AgensGraph를 설치하고, 예제 그래프를 적재하고, 첫 쿼리를 실행합니다.

개념

속성 그래프 모델, 그리고 그래프가 PostgreSQL에 어떻게 저장되는가.

쿼리

읽기, 쓰기, 경로, 인덱스, 그리고 SQL과 Cypher를 섞어 쓰기.

레퍼런스

절, 데이터 정의, 함수, 타입, 설정.

가이드

데이터 적재, 벡터·전문·하이브리드 검색, 그리고 AI 연동.

드라이버

어떤 PostgreSQL 드라이버로도 접속되며, Python 드라이버는 그래프 타입까지 다룹니다.

운영

서버를 배포하고, 설정하고, 지키고, 백업합니다.

시스템 카탈로그

그래프를 설명하는 카탈로그와 뷰.

릴리스 노트

2.18에서 무엇이 바뀌었고 어떻게 업그레이드하는가.

© 2026 SKAI Worldwide Co., Ltd.

시작하기

AgensGraph를 설치하고, 이 매뉴얼의 나머지가 사용하는 그래프를 적재한 뒤, 첫 쿼리를 실행합니다.

AgensGraph란

AgensGraph는 멀티모델 데이터베이스입니다. 서버 하나, 데이터베이스 하나, 연결 하나입니다. 데이터는 그곳에 **속성 그래프**와 **관계형 테이블**로 동시에 존재하며, **Cypher**, **GQL**, **SQL**로 쿼리합니다. 이 중 둘을 한 문장에서 함께 쓸 수 있습니다.

AgensGraph는 PostgreSQL 위에 만들어졌습니다. 2.18 릴리스는 PostgreSQL 18에 그래프 모델을 더한 것이므로, PostgreSQL이 주는 모든 것이 그대로 따라옵니다. 트랜잭션, 롤과 행 수준 보안, `pg_dump`와 `pg_basebackup`, 스트리밍 복제, 플래너와 `EXPLAIN`, 확장 생태계, 그리고 모든 PostgreSQL 드라이버가 이미 말하는 통신 프로토콜입니다.

무엇이 다른가

그래프는 PostgreSQL 객체 옆에 있는 것이 아니라, PostgreSQL 객체로 저장됩니다. 그래프는 스키마이고, 레이블은 테이블이며, 정점은 행입니다. 따로 보조를 맞춰 줄 그래프 저장소가 없고, 건너야 하는 다리도 없습니다. 한 트랜잭션 안의 Cypher 쓰기와 SQL 쓰기는 함께 커밋됩니다. 둘 다 같은 데이터베이스에 있는 테이블에 쓰는 것이기 때문입니다. **그래프가 어떻게 저장되는가**가 이 매뉴얼의 나머지를 이해하게 해 주는 페이지입니다.

순회는 직접 작성하는 재귀 조인이 아니라, 일급 쿼리입니다. "이 영화와 배우를 공유하는 영화는 무엇인가", "이 두 사람은 얼마나 떨어져 있는가", "세 홉 안에 도달할 수 있는 것은 무엇인가"는 모두 패턴입니다:

```
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"
```

확장은 그래프 데이터에도 적용됩니다. 속성은 일반 테이블에 놓인 `jsonb` 입니다. 그래서 `pgvector`는 정점에 저장된 임베딩을 인덱싱하고, PostgreSQL의 전문 검색은 텍스트 속성을 인덱싱합니다. 그러면 하나의 쿼리가 의미적으로 유사한 문서를 찾고 거기서 그래프를 따라 바깥으로 뻗어 나갈 수 있습니다. **벡터 검색**을 참고하십시오.

ISO 표준인 GQL이 같은 언어의 일부입니다. 2.18은 ISO/IEC 39075의 상당 부분을 추가하며, 별도 모드도 별도 파서도 없습니다. `LET`, `NEXT`, `FILTER`, `FINISH`, `INSERT`, `FOR`, `CALL { }`과 서브쿼리 값 표현식은 Cypher가 허용되는 곳이면 어디서나 허용됩니다. **GQL 적합성**을 참고하십시오.

누구를 위한 것인가

관계형 데이터를 갖고 있고, 그 데이터에 대해 그래프 모양의 질문이 있으며, 그 질문을 하려고 두 번째 데이터베이스를 운영하고 싶지는 않은 팀입니다. 흔한 형태는 추천과 유사도, 사기와 위험 사슬, 권한과 소유 계층, 네트워크와 의존성 토폴로지, 그리고 검색 증강 생성(RAG) 뒤의 지식 그래프입니다.

세 경우에 잘 맞습니다. 그래프와 테이블이 같은 기록 시스템에 속할 때. 이미 PostgreSQL을 운영하고 있어 그 기술과 도구를 계속 쓰고 싶을 때. 그리고 쿼리가 순회와 집계, 조인, 윈도 함수를 섞을 때입니다.

한 그래프를 여러 장비에 샤딩하는 분산 그래프 저장소가 필요하거나, PageRank나 커뮤니티 탐지처럼 그래프 전체를 다루는 알고리즘을 대규모로 돌리는 그래프 분석 엔진이 필요하다면 맞는 도구가 아닙니다. AgensGraph는 PostgreSQL이 확장되는 방식으로 확장됩니다 — 하나의 주 서버, 읽기 복제본, 그리고 수직 확장입니다.

© 2026 SKAI Worldwide Co., Ltd.

이번 릴리스에 담긴 것

- 기반이 되는 PostgreSQL 18.
- GQL 적합성을 향한 큰 진전.
- 타입 컬럼 속성 승격. 자주 쓰이는 `jsonb` 속성을 쿼리 작성 방식을 바꾸지 않고 인덱싱 가능한 실제 컬럼으로 바꿉니다. **속성을 컬럼으로 승격하기**를 참고하십시오.
- 그래프 순회에 대한 플래너 및 추정 작업.

업그레이드 전에 애플리케이션이 살펴봐야 하는 변경이 몇 가지 있습니다. 새로 추가된 예약어와 `WHERE`에 올 수 있는 것에 대한 더 엄격한 규칙이 그렇습니다. **릴리스 노트**가 그것들을 나열하고, **2.18로 업그레이드하기**에 절차가 있습니다.

다음

AgensGraph 설치를 하고, 이 매뉴얼의 모든 페이지가 사용하는 **예제 데이터셋을 적재**한 뒤, **첫 쿼리를 실행**하십시오.

AgensGraph 설치

경로는 세 가지입니다. Docker는 쿼리할 수 있는 서버에 가장 빨리 도달하는 방법입니다. 패키지는 워크스테이션이나 단일 서버에 흔히 쓰는 선택입니다. 무언가를 바꿔야 하거나 사용하는 플랫폼에 패키지가 없다면 소스에서 빌드하십시오.

AgensGraph를 그저 시험해 보고 싶을 뿐이라면 Docker를 쓰고 **예제 데이터셋**으로 건너뛰십시오.

Docker

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

그다음 컨테이너 안에서 클라이언트를 엽니다:

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

이 이미지는 공식 `postgres` 이미지와 같은 환경 변수를 받습니다 — `POSTGRES_USER`, `POSTGRES_DB`, `PGDATA` 등입니다. 그러므로 포트를 공개하고 볼륨을 마운트하는 방식도 평소와 같습니다:

```
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
```

태그는 릴리스에 기반 이미지를 붙인 형태입니다: `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`. 하나를 고정하십시오. `skaiworldwide/agensgraph`만 쓰면 가리키는 대상이 계속 움직입니다.

> **경고**
>
> 2.18.6.1 이미지는 PostgreSQL 18의 디렉터리 배치 변경을 따릅니다. `PGDATA`의 기본값이 이제 `/var/lib/postgresql/18/docker`이고, 선언된 볼륨이 `/var/lib/postgresql/data`에서 `/var/lib/postgresql`로 옮겨졌습니다. 이전 태그가 초기화한 호스트 볼륨은 명시적으로 가리키지 않는 한 2.18.6.1 컨테이너가 찾지 못합니다:
>
> ```
> -e PGDATA=/var/lib/postgresql/data -v /custom/mount:/var/lib/postgresql/data
> ```


© 2026 SKAI Worldwide Co., Ltd.

어느 경우든 기존 데이터 디렉터리를 재사용하기 전에 **2.18로 업그레이드하기**를 읽으십시오 — 2.17 데이터 디렉터리는 2.18로 기동할 수 없습니다.

패키지

AgensGraph 다운로드 페이지에서 사용하는 플랫폼용 빌드를 받아 원하는 곳에 풀어 놓으십시오. 예를 들어 `/usr/local/AgensGraph` 입니다:

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

어떤 빌드가 필요한지 확실하지 않다면:

```
uname -sm
```

그다음 **데이터베이스 준비**로 이어 가십시오.

소스에서 빌드

빌드 의존성 설치

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
```

빌드와 설치

```
git clone https://github.com/skaiworldwide-oss/agensgraph.git
cd agensgraph
./configure --prefix=$(pwd)
make install
```

`--prefix`가 바이너리가 놓이는 곳을 결정합니다. 이것을 주지 않으면 `make install`이 `/usr/local/pgsql`에 씁니다. 위처럼 소스 트리를 가리키게 하면 이것저것 시험해 보는 동안 모든 것이 한곳에 모여 있습니다.

contrib 모듈까지 함께 빌드하려면 — `pgvector`는 그중에 없지만, **적재 가이드**가 사용하는 `file_fdw`는 있습니다:

```
make install-world
```


© 2026 SKAI Worldwide Co., Ltd.

`configure`가 필요한 헤더를 찾지 못하면 `--with-includes=/path/to/headers`로 어디를 볼지 알려 주십시오. `./configure --help`가 모든 옵션을 나열합니다. 그 옵션들은 PostgreSQL의 것이고, **PostgreSQL 설치** 장이 그것들을 문서화하고 있습니다.

바이너리를 PATH에 넣기

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

계속 유지되게 하려면 위 내용을 `~/.bashrc`에, macOS라면 `~/.zshrc`에 추가하십시오.

데이터베이스 준비

도구는 PostgreSQL의 것이며, 두 개의 이름이 바뀌었습니다. `agens`가 대화형 클라이언트이고 `ag_ctl`이 서버를 기동하고 정지합니다. **agens 클라이언트**를 참고하십시오.

데이터 디렉터리를 만듭니다:

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

서버를 기동합니다:

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

데이터베이스를 만듭니다:

```
createdb mydb
```

`AGDATA`를 설정하면 이후 명령에서 `-D`를 생략할 수 있습니다:

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

설치 확인

접속합니다:

```
agens mydb
```

서버에게 자신이 무엇인지 물어봅니다:

```
SELECT version();
```

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

그다음 그래프 쪽이 있는지 확인합니다. 이 부분이 순수 PostgreSQL 빌드라면 실패하는 대목입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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;
```

그래프가 그 안의 레이블과 시퀀스를 소유하고 있으므로 `CASCADE` 가 필요합니다.

다음

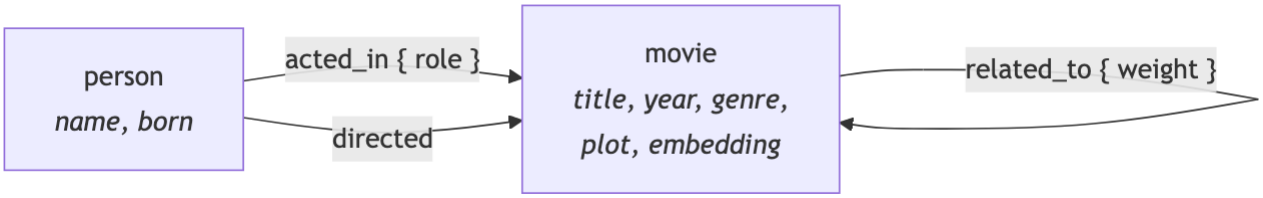
이 매뉴얼의 모든 페이지가 사용하는 **예제 데이터셋을 적재**한 뒤, **첫 쿼리를 실행**하십시오.

운영 설치는 — 데이터 디렉터리를 어디에 둘지, `postgresql.conf` 에 무엇을 설정할지, systemd 아래에서 어떻게 실행할지, 보안을 어떻게 강화할지 — **배포**와 **구성**을 참고하십시오.

예제 데이터셋

이 매뉴얼의 모든 예제는 영화와 그것을 만든 사람들로 이루어진 하나의 작은 그래프를 대상으로 실행되며, 그래프 이름은 `moviekb` 입니다. 한 번 준비해 두면 모든 페이지가 동작합니다.

이 그래프에는 정점 레이블 두 개와 간선 레이블 세 개가 있습니다:



그 옆에 일반 테이블인 `rating` 이 함께 있습니다. **SQL과 Cypher 섞기** 페이지가 조인할 실제 관계형 데이터를 갖도록 하기 위한 것입니다.

그래프 만들기

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

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

`graph_path` 는 그래프 이름으로 한정하지 않은 레이블 이름이 어느 그래프에 속하는지를 세션에 알려 줍니다. 세션 단위 설정이므로 새 세션마다 다시 설정하거나, 데이터베이스에 설정해 두십시오:

© 2026 SKAI Worldwide Co., Ltd.

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

영화 적재

각 영화에는 `plot`과 그 줄거리의 `embedding`이 있습니다. 실제 임베딩은 모델에서 나오고 수백 차원을 갖지만, 여기서는 지면에서 기하 구조를 읽을 수 있도록 4차원으로 줄였습니다. **벡터 검색**과 **하이브리드 검색**은 이것을 그대로 사용합니다.

```
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:
'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
```


© 2026 SKAI Worldwide Co., Ltd.

```
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]});
```

사람 적재

```
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});
```

연결하기

```
-- 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);
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
```


© 2026 SKAI Worldwide Co., Ltd.

```
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);
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
```


© 2026 SKAI Worldwide Co., Ltd.

```
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);
```

관계형 테이블 추가

```
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);
```

적재한 것 확인

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


© 2026 SKAI Worldwide Co., Ltd.

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

같은 데이터를 테이블로 보기

그래프는 스키마이고 레이블은 테이블이므로, 방금 만든 그래프는 SQL로 쿼리할 수 있는 일반 테이블 여덟 개이기도 합니다. `\dGv` 는 정점 레이블을 나열하고, `\dGe` 는 간선 레이블을 나열하며, `\d` 는 레이블의 모양을 보여 줍니다:

```
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
```

간선 테이블은 양 끝점의 id도 함께 담고 있습니다:

```
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"}
```

그래프가 어떻게 저장되는가가 그 컬럼들이 무엇이고 어떤 인덱스가 함께 만들어지는지 설명합니다.

처음부터 다시

```
DROP GRAPH moviekb CASCADE;
```


© 2026 SKAI Worldwide Co., Ltd.

```
DROP TABLE rating;
```

첫 쿼리

이 페이지는 **예제 데이터셋**을 훑습니다 — 영화 18편, 사람 28명, 그리고 그 사이의 간선들입니다. 먼저 데이터셋을 적재하고 페이지를 위에서 아래로 따라가십시오. 각 쿼리는 앞의 쿼리 위에 쌓입니다.

여기 있는 것은 모두 Cypher입니다. 이에 대응하는 GQL 표기는 **레퍼런스**에 적혀 있습니다. SQL이 다시 등장하는 곳은 **SQL과 Cypher 섞기**입니다.

어느 그래프인지 말하기

```
SET graph_path = moviekb;
```

Cypher가 필요로 하는 세션 상태는 이것 하나입니다. 이것이 없으면 모든 레이블 이름을 해석할 수 없습니다. 이제 그래프에 있는 모든 것을 세어 봅니다:

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

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

`MATCH (n)`은 가능한 가장 비어 있는 패턴입니다 — 정점 하나, 레이블 없음, 속성 없음 — 그래서 모든 정점에 일치합니다. 46은 영화 18편에 사람 28명을 더한 것입니다.

하나 찾기

패턴은 두 가지 방식으로 범위를 좁힙니다. 레이블로, 그리고 속성으로입니다.

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

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

`(m:movie {title: 'The Matrix'})`은 이렇게 읽습니다 — `movie` 레이블이 붙고 `title` 속성이 `'The Matrix'`인 정점, 그리고 그것을 변수 `m`에 바인딩한다. 그다음 `RETURN`이 `m`에서 값을 꺼냅니다.

`"Action"`에는 따옴표가 붙고 `1999`에는 붙지 않은 것을 보십시오. `RETURN`은 속성을 `jsonb`로 돌려주므로 문자열은 JSON 따옴표가 붙은 채로 옵니다. 결과를 클라이언트 쪽 문자열과 비교하기 전에 알아 둘 만한 점입니다 — 평범한 값을 얻는 방법은 **SQL과 Cypher 섞기**를 참고하십시오.

모든 정점에는 속성이 아닌 식별자도 있습니다:

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


© 2026 SKAI Worldwide Co., Ltd.

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

`3.1`은 `graphid` 입니다. 레이블 3, 행 1입니다. 이것이 정점의 식별자이고, `id(m)`이 그것을 묻는 방법입니다 — `m.id`는 `id`라는 이름의 **속성**을 찾는데, 그런 속성을 가진 영화는 없습니다.

간선 따라가기

그래프의 요점은 그다음에 오는 것입니다. 패턴에 간선을 더합니다:

```
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"
```

새로운 것이 세 가지 있습니다. `-[r:acted_in]->`는 `acted_in` 레이블이 붙은 간선이며 사람에서 영화를 향합니다. 이것이 `r`에 바인딩되므로 그 속성을 쓸 수 있습니다. `r.role`은 양 끝점이 아니라 **관계**의 속성입니다. 그리고 `(:movie {title: 'The Matrix'})`에는 변수가 없습니다. 패턴이 그 영화에 도달할 필요는 있지만 결과 투영에서는 그것을 언급하지 않기 때문입니다.

두 번 따라가기

패턴을 반대 방향으로 늘리면, SQL이라면 자기 조인이 필요했을 질문을 한 것이 됩니다:

```
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"
```

패턴을 왼쪽에서 오른쪽으로 읽습니다. Keanu Reeves에서 출발해 `acted_in` 간선을 따라 영화로 나가고, 다시 다른 `acted_in` 간선을 **거꾸로** 따라 두 번째 사람에게 갑니다. 영화 세 편을 함께한 두 사람은 행 세 개를 만들기 때문에 `DISTINCT`가 필요합니다.

집계

집계 함수는 PostgreSQL의 것이고, 예상하는 대로 동작합니다:

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "James Cameron"      | 3
 "Lilly Wachowski"    | 3
 "Christopher Nolan" | 2
 "Steven Spielberg"   | 2
```

`RETURN` 안에서 집계 함수 안에 들어 있지 않은 것은 모두 그룹화 키가 됩니다. 이것도 SQL과 같습니다.

없을 수도 있는 것을 묻기

`MATCH` 는 패턴이 맞지 않으면 그 행을 버립니다. `OPTIONAL MATCH` 는 행을 남겨 두고 채우지 못한 변수를 null로 채웁니다 — 왼쪽 외부 조인입니다:

```
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"     |
```

이 데이터셋에는 연출과 연기를 모두 한 사람이 없으므로 `also_acted_in` 은 모두 null입니다 — 그것이 바로 요점입니다. 그냥 `MATCH` 를 썼다면 여섯 행이 전부 사라졌을 것입니다.

가운데 있는 `WITH` 는 파이프라인 구분자입니다. 쿼리의 한 부분을 끝내고 다음 부분을 시작하며, 덕분에 선택적 일치가 실행되기 전에 `DISTINCT` 가 감독들에게 적용됩니다.

몇 홉인지 모르는 길을 걷기

경로를 모른 채로 두 사람이 어떻게 연결되는지 묻습니다:

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

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

`[*..6]` 은 가변 길이 간선입니다. 어떤 레이블이든, 어느 방향이든, 최대 여섯 개까지입니다. `p = ...` 는 경로 전체를 바인딩하고, `length(p)` 는 그 간선 수입니다. 이것이 SQL로는 쓸 만한 형태가 없는 쿼리입니다.

`shortestpath` 의 양 끝점에는 변수가 있어야 합니다. 그 변수를 쓰지 않더라도 그렇습니다. 익명으로 두면 `ERROR: node must be a reference to a specific node` 가 나옵니다.

나머지는 경로가 다릅니다 — 범위를 정한 구간, `allshortestpaths` , 그리고 `dijkstra` 를 쓰는 가중 탐색입니다.

무언가 쓰기

쓰기는 동사만 다른 읽기처럼 보입니다. `CREATE` 는 정점을 만듭니다:

© 2026 SKAI Worldwide Co., Ltd.

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

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

무언가에 연결하려면 먼저 양쪽 끝을 찾고 그다음 간선을 만듭니다:

```
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"}
```

간선은 `label[id][start,end]{properties}` 형태로 출력되므로 결과에서 양 끝점을 볼 수 있습니다.

`SET` 은 속성을 추가하거나 교체합니다:

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

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

`REMOVE` 는 속성을 없앱니다:

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

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

그리고 `DETACH DELETE` 는 정점을 그에 딸린 간선과 함께 지웁니다. 아직 간선이 남아 있는 정점에 그냥 `DELETE` 를 쓰면 거부됩니다. AgensGraph는 끝점이 없는 간선을 남겨 두지 않기 때문입니다:

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

이렇게 하면 데이터셋이 46개 요소로 돌아가므로 매뉴얼의 나머지 예제가 계속 맞습니다.

SQL 섞어 쓰기

데이터셋과 함께 만든 `rating` 테이블은 그래프의 일부가 아닌 일반 테이블입니다. SQL이 테이블을 기대하는 자리에 Cypher를 놓아 Cypher 쿼리와 조인합니다:

```
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 #>> '{}'
```


© 2026 SKAI Worldwide Co., Ltd.

```
WHERE r.source = 'critics'
ORDER BY r.score DESC;
```

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

`#>> '{}'`가 `jsonb` 제목을 평범한 `text`로 바꿔 `text` 컬럼과 비교할 수 있게 해 줍니다. 그리고 여기서는 영화 이름이 따옴표 없이 돌아오는 것을 보십시오. 이것은 Cypher가 아니라 SQL의 결과 투영이기 때문입니다.

다음에 갈 곳

- **쿼리하기**가 여기 나온 생각들을 하나씩 제대로 다룹니다. 읽기, 쓰기, 파이프라인, 경로, 인덱스입니다.
- **레퍼런스**에는 절마다 전체 구문을 담은 페이지가 있습니다.
- **그래프가 어떻게 저장되는가**가 방금 만든 모든 것에 대해 서버가 실제로 무엇을 했는지 설명합니다.
- **벡터 검색, 전문 검색, 하이브리드 검색**은 이 페이지가 다루지 않은 `plot`과 `embedding` 속성을 사용합니다.

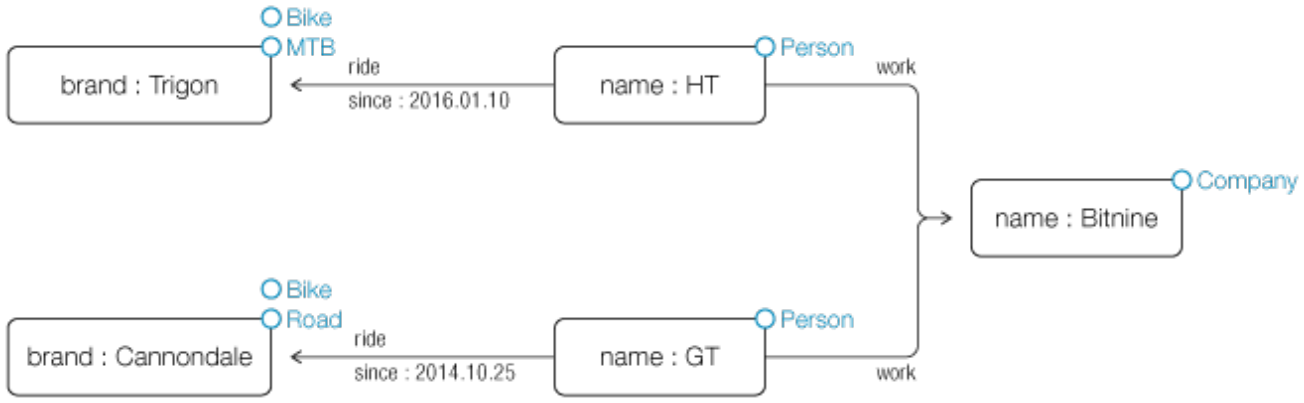
© 2026 SKAI Worldwide Co., Ltd.

개념

속성 그래프가 무엇인지, AgensGraph가 그것을 PostgreSQL에 어떻게 저장하는지, 그리고 세 가지 쿼리 언어가 서로 어떤 관계인지 설명합니다.

속성 그래프 모델

AgensGraph는 데이터를 두 가지 모양으로 동시에 저장합니다. 관계형 테이블로, 그리고 **속성 그래프**로입니다. 이 페이지는 그중 그래프 쪽을 정의합니다 — 그래프를 이루는 다섯 가지, 그리고 AgensGraph에서 그 각각이 지키는 규칙입니다.



정점

정점은 개체입니다. 사람, 영화, 계정, 센서 같은 것입니다. 정점에는 식별자와 레이블이 있고, 속성은 몇 개든 가질 수 있습니다.

정점은 **정확히 하나의 레이블**에 속합니다. 이전에 써 본 그래프 데이터베이스와 가장 다를 만한 규칙입니다 — 정점마다 레이블 집합을 갖는 방식이 아닙니다. 한 정점에 여러 레이블을 붙이고 싶은 상황이라면 대신 **레이블 상속**을 쓰십시오. 하나의 패턴이 레이블 계열 전체에 일치하게 해 줍니다.

간선

간선은 두 정점을 연결합니다. 간선에는 식별자, 레이블, 몇 개든 가질 수 있는 속성, 그리고 방향이 있습니다. 간선에는 항상 시작 정점과 끝 정점이 있습니다.

AgensGraph에서 성립하는 규칙이 세 가지 있습니다:

- **간선에는 항상 레이블이 있습니다.** 레이블 없는 간선은 없습니다.
- **간선에는 항상 양 끝점이 있습니다.** 끝점이 없는 간선은 존재할 수 없으므로, 아직 간선이 남아 있는 정점을 지우려 하면 거부됩니다:

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

- **방향은 기록되지만, 쿼리에서 방향을 따질지는 선택입니다.** 간선은 한 방향으로 저장해 두고, 방향이 중요한지는 쿼리마다 정합니다. `-[:acted_in]->`는 방향을 따르고, `-[:acted_in]-`는 무시합니다. 간선을 거울처럼 뒤집어 하나 더 저장할 이유가 없습니다.

속성

속성은 정점이나 간선에 담긴 데이터입니다. 속성에는 문자열 이름과 값이 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

속성은 요소마다 하나의 `jsonb` 문서로 저장되므로, 값은 어떤 JSON 타입이든 될 수 있습니다. 문자열, 숫자, 불리언, 배열, 중첩된 객체입니다. 여기서 두 가지가 따라옵니다:

- **레이블은 컬럼 목록을 강제하지 않습니다.** 같은 레이블을 가진 두 정점이 완전히 다른 속성 이름을 가질 수 있습니다. 미리 선언해 둘 것이 없습니다.
- **`null` 은 속성 값이 아닙니다.** `null` 을 할당하면 그 속성이 지워집니다:

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

  그러므로 null로 설정한 속성과 애초에 없던 속성은 같은 것입니다. `allow_null_properties` 를 켜면 이 동작이 달라져 null이 저장됩니다:

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

읽기가 JSON 문서 안으로 들어가야 하므로, 모든 쿼리가 필터에 쓰는 속성이라면 **타입 컬럼으로 승격**할 만합니다. 이것은 모델링의 선택이 아니라 성능의 선택입니다 — 속성은 그대로 속성으로 남습니다.

참고

`n.id` 는 정점의 식별자가 아닙니다. `id` 라는 이름의 속성을 읽는 것이며, 그런 속성은 보통 존재하지 않습니다. 식별자는 `id(n)` 입니다.

레이블

레이블은 같은 종류의 요소를 묶습니다. `movie` , `person` , `acted_in` 이 레이블입니다.

레이블은 인덱스를 붙이는 대상이고, 권한을 부여하는 대상이며, 패턴이 검색 범위를 좁히려고 이름으로 부르는 대상입니다. 정점 레이블과 간선 레이블은 서로 다른 종류이고, 각각 `CREATE VLABEL` 과 `CREATE ELABEL` 로 만들며, 한 그래프 안에서 이름은 한 번만 쓸 수 있습니다.

모든 그래프는 선언하지 않은 레이블 두 개와 함께 만들어집니다. `ag_vertex` 와 `ag_edge` 입니다. 직접 만드는 모든 레이블은 그중 하나를 상속합니다. `MATCH (n)` 처럼 레이블 없는 패턴이 "임의의 정점"을 뜻하게 되는 것이 이 상속 덕분입니다. 레이블을 전혀 주지 않고 만든 정점은 `ag_vertex` 에 놓입니다:

```
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
```

기본 레이블을 직접 이름으로 지정할 수는 없습니다:

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

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

레이블 이름은 큰따옴표로 감싸지 않으면 소문자로 바뀝니다. PostgreSQL이 따옴표 없는 식별자를 소문자로 바꾸는 것과 같습니다. 여기서 백틱은 인용 문자가 아닙니다 — 구문을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

경로

경로는 간선으로 이어진 정점의 나열입니다. 경로 자체가 하나의 값이므로, 변수에 바인딩하고, 반환하고, 그 길이나 구성원을 물을 수 있습니다.

```
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"
```

경로의 길이는 그 간선의 개수입니다. 그러므로 정점 하나는 길이 0인 경로이고, 자기 자신으로 가는 간선은 길이 1인 경로입니다. "몇 홉이든"을 표현하는 방법이 경로입니다. 가변 길이 패턴, 최단 경로, 가중 탐색은 **경로**를 참고하십시오.

그래프

그래프는 레이블을 담는 그릇이고, 데이터베이스는 여러 개를 가질 수 있습니다. 그래프는 동시에 PostgreSQL 스키마이므로 그래프 이름과 스키마 이름은 충돌할 수 없습니다 — **그래프가 어떻게 저장되는가**를 참고하십시오.

세션은 한 번에 **하나**의 그래프에서 작업하며, 그 그래프는 `graph_path`로 지정합니다:

```
SET graph_path = moviekb;
```

이것이 없으면 아무것도 해석되지 않습니다:

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

`graph_path`는 그래프 이름을 정확히 하나만 받으며, 패턴에서 레이블을 그래프 이름으로 한정할 수는 없습니다 — `MATCH (n:other.work)`는 구문 오류입니다. 한 문장에서 두 그래프를 읽으려면 SQL을 쓰십시오. SQL은 그 둘을 두 개의 스키마로 봅니다:

```
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;
```

`graph_path`는 세션 단위 설정이므로, 모든 세션이 같은 그래프를 쓴다면 데이터베이스에 설정해 두십시오:

© 2026 SKAI Worldwide Co., Ltd.

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

그래프가 어떻게 저장되는가

AgensGraph는 PostgreSQL에 그래프 엔진을 덧붙이지 않습니다. 이미 거기 있는 스키마, 테이블, 인덱스, 상속을 써서 그래프를 PostgreSQL 객체 **그 자체로** 저장합니다. 이 대응 관계를 알고 나면 PostgreSQL 운영에 대해 아는 것 대부분이 그대로 적용됩니다. 그리고 한 문장이 Cypher와 SQL을 섞을 수 있다는 것도 더는 놀랍지 않게 됩니다. 테이블은 처음부터 한 벌뿐이기 때문입니다.

대응 관계는 세 문장입니다.

그래프는 스키마입니다. `CREATE GRAPH moviekb` 는 `moviekb` 라는 이름의 스키마를 만듭니다.

레이블은 테이블입니다. `CREATE VLABEL movie` 는 `moviekb.movie` 테이블을 만듭니다.

정점이나 간선은 그 테이블의 **행입니다.**

레이블 테이블은 어떻게 생겼는가

정점 레이블에는 컬럼이 두 개 있습니다:

```
\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
```

간선 레이블에는 같은 컬럼 두 개에 양 끝점의 id가 더해집니다:

```
\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.

레이블은 그 컬럼들보다 더 많은 것을 담을 수 있습니다. **속성 승격**은 `properties` 옆에 실제 타입이 있는 컬럼을 추가하고, 쓰기 시점에 그 값을 채웁니다. 그러면 자주 쓰는 속성이 두 번 저장됩니다 — 한 번은 jsonb 키로, 한 번은 플래너가 인덱싱하고 그 타입 그대로 비교할 수 있는 컬럼으로입니다. 어떤 속성 키가 어느 컬럼에 대응하는지는 카탈로그 `ag_label_property`가 기록합니다. Cypher에서는 계속 키 이름을 부르므로 컬럼 이름이 쿼리에 나타나는 일은 없습니다.

그러므로 간선은 두 정점을 지목하는 행이고, 간선을 따라가는 것은 `graphid`에 대한 조인입니다. `(start, "end")`와 `("end", start)`에 걸린 btree 인덱스 두 개가 그 조인을 어느 방향으로든 빠르게 만들어 주며, 레이블을 만들 때 AgensGraph가 이 인덱스를 함께 만들어 줍니다. `id`에 걸린 BRIN 인덱스는 작고, 한 레이블의 id 범위를 스캔할 때 도움이 됩니다.

레이블 종류마다 어떤 인덱스가 붙는지 보십시오. 정점 레이블에는 `id`에 대한 기본 키가 있지만 간선 레이블에는 없습니다. 간선은 id로 찾는 것이 아니라 양 끝점을 통해 도달하기 때문입니다.

`properties`가 하나의 `jsonb` 컬럼이므로, 레이블은 자기 행들에 컬럼 목록을 강제하지 않습니다. 두 개의 `movie` 정점이 완전히 다른 키를 가질 수 있습니다. 여기서 "스키마 없는 속성"이 뜻하는 것이 그것입니다. 그리고 속성을 읽는 것이 JSON 문서 안을 읽는 일이 되는 이유도 그것입니다. 그 비용이 언제 문제가 되고 무엇을 할 수 있는지는 **속성을 컬럼으로 승격하기**가 다룹니다.

식별자: graphid

`graphid`는 두 부분으로 이루어진 64비트 식별자입니다. 상위 16비트가 **레이블 id**이고, 하위 48비트가 레이블마다 매겨지는 시퀀스인 **로컬 id**입니다. 출력 형태는 `labid.locid`입니다:

```
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
```

이 배치에서 두 가지가 따라오고, 둘 다 중요합니다.

첫째, id가 자기 레이블을 함께 지니고 있습니다. 정점이 어느 테이블에서 왔는지 알기 위해 무언가를 찾아볼 필요가 없습니다.

둘째, 레이블 id가 **상위** 비트이므로 한 레이블의 모든 id는 `graphid` 값의 연속된 한 구간을 이룹니다. 그래서 `id`에 걸린 평범한 btree 인덱스가 `movie` 테이블을 전혀 건드리지 않고도 "이 정점이 `movie`인가?"에 답할 수 있습니다.

레이블 id 자체는 `ag_label`에 있습니다:

```
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`는 16비트이고 0은 유효한 레이블이 아니므로, 데이터베이스는 최대 65,535개의 레이블을 가집니다. 하나의 레이블은 최대 $2^{48} - 1$개의 행을 가집니다. **제한**을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

모든 그래프에는 만들지 않은 레이블 두 개가 있다

`ag_vertex`와 `ag_edge`는 그래프와 함께 만들어지고, 직접 만드는 모든 레이블은 그중 하나를 상속합니다:

```
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
```

레이블 없는 패턴에 뜻을 주는 것이 이 상속입니다. `MATCH (n)`은 모든 것을 담은 특별한 인덱스를 스캔하는 것이 아닙니다. `ag_vertex`를 스캔하고, 상속을 통해 그 아래의 모든 테이블을 스캔합니다:

```
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
```

레이블을 이름으로 부르면 스캔 범위가 그 테이블과 그 자식들로 좁혀지고, `ONLY` 수정자를 쓰면 정확히 그 테이블로만 좁혀집니다. 그것으로 무엇을 만들 수 있는지는 **레이블 상속**이 다룹니다.

속성 인덱스와 제약은 무엇이 되는가

Cypher의 속성 인덱스는 `properties` 컬럼에 대한 PostgreSQL 표현식 인덱스입니다:

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

```
\d moviekb.movie
```

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

고유성 제약은 같은 표현식에 대한 배제 제약이 됩니다:

```
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 =)
```

그 정의에 나오는 `properties.'title'` 형태는 Cypher 파서가 속성 읽기를 표기하는 방식입니다. SQL에서 직접 입력하는 구문이 아닙니다.

튜닝할 때 기억할 점은 이것들이 평범한 인덱스라는 것입니다. 표현식에 걸린 btree 인덱스입니다. 플래너는 평소의 규칙대로 이 인덱스를 고르며, 그 표현식과 맞출 수 있는 조건에만 도움이 됩니다. 어떤 조건이 해당되는지는 **인덱스와 제약**이 다룹니다.

이것으로 얻는 것

PostgreSQL이 테이블에 해 주는 모든 것을 레이블에도 해 줍니다. 이것은 구호가 아니라, 두 번 배우지 않아도 되는 것들의 목록입니다:

- **트랜잭션.** 한 트랜잭션 안의 Cypher 쓰기와 SQL 쓰기는 함께 커밋되거나 함께 롤백됩니다. 같은 데이터베이스에 있는 테이블에 쓰는 것이기 때문입니다.
- **접근 제어.** `GRANT SELECT ON moviekb.movie TO analyst`가 동작하고, 레이블에 대한 행 수준 보안도 동작합니다. **보안**을 참고하십시오.
- **백업과 복제.** `pg_dump`, `pg_basebackup`, WAL 아카이빙, 스트리밍 복제는 그래프를 있는 그대로 — 스키마와 테이블로 — 다룹니다.
- **확장.** `pgvector`가 그래프 옆에 임베딩을 저장할 수 있는 것은, 속성이 테이블 안의 `jsonb`이고 확장이 그것을 인덱싱할 수 있기 때문입니다. **벡터 검색** 가이드 전체가 이 사실에 기반합니다.
- **플래너.** Cypher 쿼리에 `EXPLAIN`을 쓰면 순차 스캔, 인덱스 스캔, 조인이 보이고, 다른 PostgreSQL 쿼리를 튜닝하는 방식으로 그것을 튜닝합니다.

그리고 이것이 SQL과 Cypher가 다리를 필요로 하지 않고 한 문장 안에서 조합되는 이유입니다: **SQL과 Cypher 섞기**.

일부러 막아 둔 것

대응 관계는 실제이지만, 그래프를 일반 테이블처럼 편집해도 된다는 뜻은 아닙니다. 관문 두 개가 길을 막고 있으며 둘 다 기본적으로 닫혀 있습니다. 두 작업 모두 그래프 카탈로그의 내용을 실제 데이터와 어긋나게 만들 수 있기 때문입니다.

행을 직접 쓰는 것은 거부됩니다:

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

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

레이블의 모양을 바꾸는 것도 거부됩니다:

```
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`과 `enable_graph_ddl`이 각각 그 문을 엽니다. 그렇게 함으로써 무엇을 감당하게 되는지는 **서버 설정**에 적혀 있습니다. SQL로 레이블을 읽는 데는 어떤 설정도 필요하지 않습니다 — 이 페이지의 모든 `SELECT`는 그대로 실행됩니다.

© 2026 SKAI Worldwide Co., Ltd.

레이블 상속

레이블은 다른 레이블을 상속할 수 있습니다. 자식은 자기 id 시퀀스와 자기 행을 가진 별개의 레이블이지만, 부모를 이름으로 부르는 패턴은 자식의 행에도 일치합니다. 덕분에 계열 전체를 하나의 패턴으로 쿼리하는 능력을 포기하지 않고도 타입 계층을 가질 수 있습니다.

이 상속은 PostgreSQL의 테이블 상속입니다 — **레이블은 테이블**이고, 레이블에 준 `INHERITS` 는 그 테이블의 `INHERITS` 가 됩니다.

계층 선언하기

이 페이지의 예제는 **예제 데이터셋**을 쓰지 않고 자체의 작은 그래프를 만듭니다. 상속은 영화와 사람보다 계층 구조에서 보기가 쉽기 때문입니다. 다 보고 나면 `DROP GRAPH inh CASCADE;` 를 실행하십시오.

```
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'});
```

부모를 이름으로 부르면 계열의 모든 행에 일치합니다:

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

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

자식을 이름으로 부르면 그 자식에만 일치합니다:

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

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

`label(n)` 이 무엇을 반환하는지 보십시오. 정점이 만들어질 때의 레이블이고, 부모는 결코 아닙니다. 정점은 정확히 하나의 레이블에 속합니다. "부모에 일치한다"는 것은 패턴이 어느 테이블을 스캔하는지에 대한 이야기이고, 정점이 레이블 두 개를 갖는다는 이야기가 아닙니다.

ONLY: 자식을 제외하기

`ONLY` 는 패턴을 이름으로 부른 그 레이블 자체로 제한합니다:

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

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


© 2026 SKAI Worldwide Co., Ltd.

간선에도 쓸 수 있고, 대괄호 안에 적습니다:

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

```
 n
---
 0
```

이 0이 바로 유용한 경우입니다. `rel` 에는 자식 `sequel` 이 있고 모든 간선이 `sequel` 로 만들어졌으므로, `rel ONLY` 는 그중 하나도 일치시키지 못하지만 그냥 `rel` 은 전부 일치시킵니다.

간선도 같은 방식으로 상속한다

```
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"
```

부모가 둘 이상일 때

레이블은 여러 레이블을 한 번에 상속할 수 있습니다:

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

그러면 `hybrid` 의 행은 `MATCH (n:work)` 에도 일치하고 `MATCH (n:film)` 에도 일치합니다.

모든 그래프가 이미 상속하고 있는 것

직접 만드는 모든 레이블은 `INHERITS` 절을 쓰지 않아도 `ag_vertex` 나 `ag_edge` 를 상속합니다. 레이블 없는 `MATCH (n)` 이 모든 것에 일치하는 이유가 그것입니다. **그래프가 어떻게 저장되는가**를 참고하십시오.

비용은 무엇인가

부모에 대한 패턴은 부모의 테이블과 그 아래 모든 테이블을 스캔하므로, 실행 계획에 계열 전체를 훑는 `Append` 가 붙습니다:

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

```
             QUERY PLAN
------------------------------------
 Result
   ->  Append
         ->  Seq Scan on work n_1
         ->  Seq Scan on film n_2
         ->  Seq Scan on series n_3
         ->  Seq Scan on hybrid n_4
```

`ONLY` 는 그 `Append` 를 없앱니다:

© 2026 SKAI Worldwide Co., Ltd.

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

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

여기서 두 가지가 따라옵니다. 계층이 깊으면, "한 레이블의 행을 달라"는 가장 값싼 쿼리조차 필요보다 많은 테이블에 대해 계획을 세우게 됩니다. 그러므로 한 레이블의 행만 원한다면 자식을 이름으로 부르거나 `ONLY` 를 붙이십시오. 그리고 인덱스는 스캔될 테이블마다 있어야 합니다. `work` 에 걸린 인덱스가 `film` 의 행을 인덱싱해 주지는 않습니다. 계층 전체에 인덱스를 만드는 방법은 **인덱스와 제약**이 다룹니다.

> **참고**
>
> 쿼리가 정말로 계열 전체를 원할 때 상속을 꺼내십시오 — 어떤 때는 `film` 이고 어떤 때는 `series` 인 `work` 같은 경우입니다. 어차피 모든 쿼리가 자식을 이름으로 부른다면, 레이블을 따로 두는 편이 더 단순하고 계획도 더 잘 나옵니다.

Cypher, GQL, SQL

AgensGraph는 세 가지 언어로 질문에 답합니다. 세 가지 제품도, 세 가지 모드도, 세 가지 연결도 아닙니다. 파서 하나, 플래너 하나, 테이블 한 벌, 통신 프로토콜 하나가 있고, 문장마다 언어를 고릅니다 — 또는 한 문장에서 둘을 섞습니다.

Cypher는 그래프 언어이며 기능을 가장 완전하게 지원하는 인터페이스입니다. 그래프가 할 수 있는 모든 것에 Cypher에서 닿을 수 있습니다.

GQL은 표준 그래프 쿼리 언어인 ISO/IEC 39075입니다. AgensGraph에서 이것은 별도의 방언이 아닙니다. GQL 문장은 Cypher 문장이 허용되는 곳이면 어디서나 허용됩니다. `LET` , `NEXT` , `FILTER` , `FINISH` , `INSERT` , `FOR` , `CALL { }` , 그리고 서브쿼리 값 표현식인 `EXISTS { }` 와 `VALUE { }` 입니다. 켜야 할 스위치도, 별도의 파서도 없습니다. 표준의 어느 부분이 들어와 있는지는 **GQL 적합성**이 기록합니다.

SQL은 PostgreSQL의 것 그대로입니다. PostgreSQL에서 알고 있는 모든 테이블, 뷰, 함수, 확장, 롤, 트랜잭션이 여기 있고, 그래프는 SQL에게 일반 테이블로 보입니다.

무엇을 고를까

취향이 아니라 질문의 모양으로 고르십시오:

- **관계를 따라가는 것** — 어떤 사람이 어떤 영화와 어떻게 연결되는지, 두 영화가 얼마나 떨어져 있는지, 세 홉 안에 무엇에 도달할 수 있는지 — 이것이 그래프 패턴이 있는 이유입니다. SQL의 조인 사슬은 같은 것을 몇 배의 길이로 말하며, "몇 홉이든"은 아예 표현할 수 없습니다.
- **집계, 윈도, 집합 연산, 일반 테이블과의 조인, 그리고 관리 작업 전반**은 SQL의 것이고, Cypher는 그것을 대신하려 하지 않습니다.
- **GQL**은 표준의 표기를 쓰고 싶을 때, 또는 그래프 데이터베이스를 여러 개 상대로 작성하며 이식 가능한 형태를 원할 때 씁니다. `INSERT` 와 `CREATE` 는 같은 일을 하고, `FILTER` 와 `WHERE` 도 같은 일을 합니다.

한 번 고르고 끝낼 필요는 없습니다. **SQL과 Cypher 섞기**가 그 교차점을 다루는 페이지 하나를 온전히 쓰고 있으며, 이미 SQL을 아는 독자에게는 이 매뉴얼에서 가장 유용한 부분입니다.

SQL을 알고 있다면

그래프는 스키마이고 레이블은 테이블이므로, 두 언어의 어휘는 구문에서 보이는 것보다 훨씬 가깝게 대응됩니다:

© 2026 SKAI Worldwide Co., Ltd.

SQL	Cypher	비고
`CREATE SCHEMA s`	`CREATE GRAPH s`	그래프가 곧 스키마입니다.
`CREATE TABLE t`	`CREATE VLABEL t`	정점 레이블입니다. 간선은 `CREATE ELABEL` 입니다.
`INSERT INTO t VALUES (…)`	`CREATE (:t {…})`	`INSERT (:t {…})` 가 GQL 표기입니다.
`SELECT … FROM t WHERE …`	`MATCH (n:t) WHERE … RETURN …`	`RETURN` 이 결과 투영이며 맨 뒤에 옵니다.
`SELECT … FROM a JOIN b ON …`	`MATCH (a)-[:e]->(b)`	패턴은 간선을 따라가는 조인입니다.
`LEFT JOIN`	`OPTIONAL MATCH`	일치하지 않은 행은 null과 함께 돌아옵니다.
`UPDATE t SET c = …`	`MATCH (n:t) SET n.c = …`	먼저 찾고 그다음 씁니다.
`DELETE FROM t WHERE …`	`MATCH (n:t) WHERE … DELETE n`	`DETACH DELETE` 는 간선까지 함께 지웁니다.
`WITH cte AS (…)`	`WITH …` / `NEXT`	Cypher의 `WITH` 는 파이프라인을 나눕니다. CTE가 아닙니다.
`ORDER BY` / `LIMIT` / `OFFSET`	`ORDER BY` / `LIMIT` / `SKIP`	`SKIP` 과 `OFFSET` 은 같은 뜻입니다.
`CREATE INDEX ON t ((c))`	`CREATE PROPERTY INDEX ON t (c)`	표현식 인덱스가 됩니다.
`count(*)` , `sum` , `avg`	같습니다	집계 함수는 PostgreSQL의 것입니다.
`information_schema` , `pg_catalog`	`ag_graph` , `ag_label`	시스템 카탈로그를 참고하십시오.

SQL 사용자가 초반에 걸리는 차이가 세 가지 있고, 세 가지 모두 오류 없이 조용히 결과만 달라집니다:

속성 읽기는 텍스트가 아니라 `jsonb` 를 반환합니다. `RETURN m.title` 은 `"The Matrix"` 를 따옴표와 함께 출력하며, 쿼리 안에서 형 변환을 해도 달라지지 않습니다. 평범한 스칼라를 얻으려면 바깥에서 벗기십시오: `SELECT t #>> '{}' FROM (MATCH (m:movie) RETURN m.title AS t) q` .

`ORDER BY 2` 는 "두 번째 컬럼"을 뜻하지 않습니다. Cypher에서 `2` 는 상수이므로 모든 행이 같다고 비교되어 아무것도 정렬되지 않습니다. 별칭이나 표현식으로 정렬하십시오.

`n.id` 는 id가 아닙니다. `id(n)` 이 `graphid` 를 반환합니다. `n.id` 는 `id` 라는 이름의 속성을 찾고, 그런 속성이 없으면 아무것도 반환하지 않습니다.

한 문장, 두 언어

SQL이 테이블을 기대하는 자리에 Cypher 쿼리가 올 수 있습니다:

```
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 #>> '{}'
```


© 2026 SKAI Worldwide Co., Ltd.

```
WHERE r.source = 'critics'
ORDER BY r.score DESC;
```

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

그리고 Cypher가 값을 기대하는 자리에 SQL이 올 수 있습니다:

```
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"
```

위 둘은 모두 **예제 데이터셋**에서 실행됩니다.

이번 릴리스에서 "GQL 지원"이 뜻하는 것

GQL은 두 번째 언어로서가 아니라 기존 문법에 추가된 문장으로 AgensGraph에 들어왔습니다. 여기서 분명히 말해 둘 만한 결과가 두 가지 따라옵니다.

GQL 문장은 한 쿼리 안에서 Cypher와 섞일 수 있습니다 — `MATCH … LET … RETURN` 도 유효한 문장이고 `MATCH … FILTER … NEXT RETURN …` 도 그렇습니다. 어떤 쿼리를 "GQL"이라고 표시하는 것은 없습니다.

그리고 적합성은 부분적이며, 모드 단위가 아니라 기능 단위입니다. **GQL 적합성**이 ISO/IEC 39075:2024의 하위 조항에 대해 기능별로 그것을 나열하며, **레퍼런스**의 각 절 페이지도 그 절에 대한 같은 내용을 자기 **호환성** 제목 아래에 반복해 둡니다.

© 2026 SKAI Worldwide Co., Ltd.

쿼리하기

그래프에 질문하는 방법입니다. 이 페이지들은 언어를 만나는 순서대로 가르치고, **레퍼런스**는 모든 절과 함수의 정식 정의를 줍니다.

그래프 읽기

모든 읽기는 패턴에서 시작합니다. 패턴은 찾고 있는 모양을 ASCII로 그린 그림입니다. 정점은 소괄호로, 간선은 대괄호와 하이픈으로 그립니다. `MATCH`는 그 그림을 플래너에게 넘기고, 플래너는 그것을 스캔과 조인으로 바꾸며, 그림이 데이터에 들어맞는 모든 경우마다 행 하나씩을 반환합니다.

이 페이지는 그 그림에 대한 것이고, 플래너가 그것으로 무엇을 하는지에 대한 것입니다. 정식 구문은 **`MATCH`**와 **`WHERE`**에 있습니다. 여기 있는 것은 모두 **예제 데이터셋**에서 실행됩니다.

정점 하나

가장 작은 패턴은 소괄호 한 쌍입니다. 레이블을 더하면 테이블 하나로 좁혀집니다:

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

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

`m`은 일치하는 각 정점에 바인딩되는 변수이고, `movie`는 레이블입니다. 둘 중 어느 것이든 생략할 수 있습니다. `(m)`은 모든 정점에 일치하고, `(:movie)`는 이름을 붙이지 않고 모든 영화에 일치하며, `()`는 이름을 붙이지 않고 모든 정점에 일치합니다.

레이블을 이름으로 부르면 그것을 상속한 레이블까지 포함됩니다. `ONLY`는 그것들을 제외합니다:

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

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

여기서는 `movie`를 상속한 것이 없으므로 어느 쪽이든 개수가 같습니다. 계층 구조에서는 그 차이가 `Append` 전체입니다. **레이블 상속**을 참고하십시오.

범위를 좁히는 두 가지 방법

패턴 안의 속성 맵은 속성이 같은지를 검사합니다:

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

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


© 2026 SKAI Worldwide Co., Ltd.

`WHERE` 는 어떤 조건이든 받으므로, 같음이 아닌 것은 모두 그리로 갑니다:

```
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
```

둘은 함께 쓸 수 있고, 그냥 `MATCH` 에서는 정확히 같은 뜻입니다. 어느 쪽으로 쓰든 플래너는 하나의 논리곱을 받습니다:

```
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
```

찾으려는 대상을 식별하는 키는 맵에 쓰고, 그 대상에 대한 조건은 `WHERE` 에 쓰십시오. 이 선택이 실제로 달라지는 곳은 `OPTIONAL MATCH` 하나뿐이며, 아래에 나옵니다.

조건은 불리언이어야 합니다. `WHERE m.title` 은 "제목이 있는 것"을 뜻하지 않고 `ERROR: cannot cast "The Matrix" (jsonb type string) to boolean` 을 냅니다. 뜻하는 검사를 그대로 쓰십시오: `WHERE m.title IS NOT NULL` .

간선

간선은 하이픈 두 개 사이에 놓인 대괄호 한 쌍이며, 방향은 화살촉으로 나타냅니다:

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

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

빈 대괄호를 보십시오. 대괄호는 생략할 수 없습니다. `-->` 와 `--` 는 패턴이 아닙니다. `--` 가 SQL의 한 줄 주석을 시작하기 때문입니다. `MATCH (p:person)-->(m:movie) RETURN count(p)` 는 첫 하이픈 이후를 모두 주석으로 지워 버리고 `ERROR: Cypher query must end with RETURN, FINISH or update clause` 로 실패합니다. `-[]->` 와 `-[]-` 로 쓰십시오.

간선을 변수에 바인딩하면 그 레이블과 속성을 쓸 수 있게 됩니다:

```
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" |
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Harrison Ford"    | "acted_in" | "Rick Deckard"
 "Ryan Gosling"     | "acted_in" | "K"
```

`label(r)`은 간선의 레이블이고 `r.role`은 간선 자체의 속성입니다. `directed` 간선에는 `role`이 없으므로 그 칸은 null입니다.

방향

`-[]->`는 저장된 대로 간선을 따라가고, `<-[]-`는 거꾸로 따라가며, `-[]-`는 방향을 무시합니다. 방향은 데이터의 성질이 아니라 쿼리의 성질입니다. 간선은 한 방향으로 저장해 두고, 방향이 중요한지는 패턴마다 정합니다.

```
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"
```

간선마다 레이블 하나

간선 패턴은 레이블을 최대 하나만 이름으로 부릅니다. 택일 표기는 없습니다:

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

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

이것은 모든 길이에서, 그리고 `shortestpath` 안에서도 성립합니다. 간선 레이블을 지정하지 말고 레이블을 조건으로 검사하십시오:

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

이 형태에는 대가가 있습니다. 레이블을 지정하지 않은 간선은 그래프의 모든 간선 레이블을 스캔하고, 레이블 검사는 그 뒤에 적용됩니다. 택일 대상이 몇 개 레이블뿐이고 그래프가 크다면, **`UNION`**으로 이은 두 패턴이 각 스캔을 좁게 유지해 줍니다.

경로 전체

패턴 앞의 `p =`는 양 끝만이 아니라 경로 자체를 바인딩합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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)`는 간선 개수, `nodes(p)`는 순서대로의 정점, `relationships(p)`는 간선입니다. 경로는 값이므로 반환할 수도, 길이로 정렬할 수도, 모을 수도 있습니다. 길이를 모르는 패턴을 비롯한 나머지는 **경로**가 다룹니다.

패턴이 둘 이상일 때

`MATCH`는 쉼표로 구분한 패턴 목록을 받습니다. 그중 둘에 함께 나타나는 변수가 그 둘을 조인합니다:

```
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`은 첫 패턴에서 바인딩되고 두 번째 패턴에서 다시 쓰입니다. 그래서 이것은 곱집합이 아니라 영화를 기준으로 한 내부 조인이 됩니다. 공통 변수가 없는 두 패턴은 **실제로** 곱집합이며, 그것도 허용되고 때로는 원하는 것입니다:

```
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"
```

두 패턴을 연달아 오는 두 개의 `MATCH` 절로 쓰는 것도 같은 뜻입니다. 위의 감독·배우 쿼리에서 쉼표 형태와 두 절 형태는 모두 31행을 내고 실행 계획의 모양도 같습니다. 더 잘 읽히는 쪽을 쓰십시오. "이것들을 함께"라면 쉼표를, "그다음"이라면 두 번째 `MATCH`를 쓰십시오.

들어맞지 않을 수도 있는 패턴

`MATCH`는 패턴이 아무것도 찾지 못한 행을 버립니다:

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


© 2026 SKAI Worldwide Co., Ltd.

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

`OPTIONAL MATCH`는 그 행을 남겨 두고 패턴의 변수를 null로 둡니다. 이것은 왼쪽 외부 조인이며, 모든 면에서 그렇게 동작합니다:

```
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" |
```

속성 맵과 `WHERE`가 더 이상 같지 않게 되는 곳이 여기입니다. `OPTIONAL MATCH`에 붙여 쓴 `WHERE`는 조인 조건에 속하므로, 그 조건에 맞지 않는 행도 null과 함께 살아남습니다:

```
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" |
```

같은 조건을 한 절 뒤에 쓰면 조인된 결과에 대한 필터가 되고, 그 행을 버립니다:

```
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)
```

SQL을 안다면 이미 만나 본 것입니다. `LEFT JOIN`에서 조건을 `ON`에 쓰는 것과 그 뒤 `WHERE`에 쓰는 것의 차이입니다. **`OPTIONAL MATCH`**를 참고하십시오.

패턴이 존재하는지 묻기

패턴은 그 자체로 조건이 아닙니다. `WHERE (p)-[:acted_in]->(:movie)`는 구문 오류입니다. `EXISTS { }` 서브쿼리에 넣으십시오:

```
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"
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Keanu Reeves"
 "Laurence Fishburne"
```

`EXISTS`, `COUNT`, `COLLECT`, `ARRAY`, `VALUE`, `IN`은 모두 이런 서브쿼리를 받습니다. **쿼리 조합하기**가 그것들을 함께 다룹니다.

패턴은 어떻게 실행 계획이 되는가

패턴은 그래프를 하나씩 따라가는 방식으로 실행되지 않습니다. 레이블 테이블에 대한 평범한 관계 대수로 컴파일되며, `EXPLAIN`이 정확히 그것을 보여 줍니다:

```
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
```

여기서 읽어 낼 만한 것이 세 가지 있습니다.

패턴의 각 요소가 스캔입니다. `person`, `acted_in`, `movie`는 세 개의 테이블이고, 패턴은 그들 사이의 조인 두 개입니다. `r.start = p.id`와 `r."end" = m.id`입니다. 간선을 따라가는 것이 곧 `graphid`에 대한 조인입니다. 그러므로 모든 간선 레이블과 함께 만들어지는 `(start, "end")`와 `("end", start)` 인덱스가 대단히 중요합니다.

끝점에 대한 레이블 검사는 간선 쪽에서 답이 나옵니다. `p`가 `person`이므로 `r.start`는 `person`의 id일 수밖에 없습니다. 레이블 id는 `graphid`의 상위 비트에 있으므로, 한 레이블의 모든 id는 연속된 한 구간을 이룹니다.

플래너는 "`start`가 person이다"를 `start >= '4.0' AND start <= '4.281474976710655'`로 바꿉니다. 이것은 간선 스캔이나 그 인덱스로 밀어 넣을 수 있는 범위 검사이고, `person`을 전혀 방문하지 않습니다. **식별자: graphid**를 참고하십시오.

패턴을 한쪽에 못 박으면 조인 순서가 바뀝니다. 한쪽 끝을 정점 하나에 고정하면 실행 계획이 테이블 전체를 해시하는 일을 멈춥니다:

```
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))
```


© 2026 SKAI Worldwide Co., Ltd.

```
               ->  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")
```

이제 `acted_in`에는 그 한 사람에 대해 `acted_in_start_idx`를 통해 접근하고, `movie`에는 찾아낸 간선 하나하나에 대해 자기 기본 키를 통해 접근합니다.

출발점이 정해진 순회에서 원하는 모양이 이것입니다. 그리고 선택도가 높은 출발점, 즉 인덱스가 걸린 속성이 보통 다른 어떤 튜닝보다 값어치가 큰 이유도 이것입니다. 어떤 조건이 출발점 역할을 할 수 있는지는 **인덱스와 제약**이 다룹니다.

쓰지 않는 끝점의 비용

쿼리가 값을 전혀 읽지 않는 끝점은 아예 가져올 필요가 없습니다. 여기서 `m`은 바인딩되고 레이블도 붙었지만, 결과 투영에서 그것을 요구하는 것은 없습니다:

```
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
```

`m`에 대한 스캔이 없습니다. `m`이 남긴 기여는 그 필터의 뒷부분 하나뿐입니다. `"end"`가 `movie` 범위 안에 있어야 한다는 것입니다. 즉 `(m:movie)` 전체가, 간선 스캔이 어차피 읽고 있던 컬럼에 대한 비교 두 개로 접혀 버린 것입니다.

이 생략을 끄면, 쿼리가 한 번도 묻지 않은 정점을 `ag_vertex` 계열 전체에서 가져옵니다:

```
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)
         ->  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
```

이 설정은 기본적으로 켜져 있고, 이런 실험 말고는 끌 이유가 없습니다.

© 2026 SKAI Worldwide Co., Ltd.

이 비교가 절약의 크기와 그 제한을 보여 줍니다. 그 `RETURN`에 `m.title`을 더하면 `Append`가 돌아옵니다. 쿼리가 실제로 그 영화를 필요로 하기 때문입니다. 쓰는 끝점만 투영하고 그 이상은 하지 마십시오.

> 참고
>
> 실행 계획을 보관해 두려는 패턴이라면 모든 요소에 이름을 붙이십시오. 이름을 붙이지 않은 요소에는 `"<0000000002>"` 같은 합성 별칭이 주어지고, 그 뒤의 카운터는 문장마다 올라갑니다 — 그래서 같은 쿼리를 두 번째로 실행하면 다른 계획이 출력됩니다.

> 참고
>
> 이름을 생략하는 것이 어디서나 공짜는 아닙니다. `shortestpath`의 양 끝점은 반드시 이름 있는 변수여야 하며, 쓰지 않을 변수라도 그렇습니다 — 그 자리에 익명 끝점을 두면 `ERROR: node must be a reference to a specific node`가 나옵니다.

다음에 갈 곳

- **쿼리 조합하기** — `WITH`, 집계, 서브쿼리, 그리고 `MATCH` 뒤에 이어지는 파이프라인의 나머지입니다.
- **경로** — 가변 길이 간선, 최단 경로, 가중 탐색입니다.
- **그래프 쓰기** — 같은 패턴에 쓰기 절을 뒤에 붙인 것입니다.
- **인덱스와 제약** — 어떤 조건에 인덱스가 쓰일 수 있는지입니다.

그래프 쓰기

쓰기 절에는 패턴 일치라고 부를 만한 자체 기능이 없습니다. 앞에서 온 것이 무엇이든 그 행들을 받아 각 행에 대해 무언가를 합니다. 그래서 맨 처음의 `CREATE`를 빼면 모든 쓰기가 같은 모양을 따릅니다. 먼저 찾고, 그다음 씁니다.

이 페이지의 계획이 그것뿐입니다. 필요한 순서대로 쓰기 절을 훑습니다. `CREATE`, `MERGE`, `SET`, `REMOVE`, `DELETE`입니다. 그리고 트랜잭션이 그래프에 무엇을 하는지로 마칩니다. 각각의 정식 구문은 레퍼런스에 있습니다. **`CREATE`**, **`MERGE`**, **`SET`**, **`REMOVE`**, **`DELETE`**입니다.

예제는 **예제 데이터셋**에서 실행되며 실행 뒤에는 처음 상태로 되돌려 놓습니다. 그래서 페이지를 따라 내려가며 실행해도 다른 페이지가 출력하는 개수와 계속 맞습니다.

새로 만들기

기존의 무엇도 바인딩하지 않는 패턴을 준 `CREATE`는 정점을 만듭니다:

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

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

`RETURN`은 생략할 수 있지만 레이블은 그렇지 않습니다. 레이블 없이 만든 정점은 `ag_vertex`에 놓이고, 간선에는 항상 레이블이 있어야 합니다. 결과에 나온 id는 그 레이블의 시퀀스가 어디까지 갔는지에 따른 값이므로, 직접 실행하면 다른 값이 나옵니다.

이미 존재하는 두 대상을 연결하려면 먼저 그것들을 찾고 그다음 간선을 만듭니다. 일치시키지 않은 정점에 간선을 쓰는 방법은 없습니다:

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


© 2026 SKAI Worldwide Co., Ltd.

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

간선은 `label[id][start,end]{properties}` 형태로 출력됩니다. 여기서 `(p)` 와 `(m)` 에는 레이블도 속성 맵도 없습니다. 이미 바인딩된 변수를 가리키는 참조이며, 여기에 레이블을 붙이면 그 대신 새 정점을 만들게 됩니다.

요소가 하나보다 긴 `CREATE` 패턴은 그 안의 모든 것을 만듭니다:

```
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` 는 같은 절의 ISO/IEC 39075 표기이며 같은 패턴을 받습니다: `INSERT (:person {name: 'Ada Lovelace'})` . **`INSERT`** 를 참고하십시오.

없을 때만 만들기

`MERGE` 는 자기 패턴을 일치시키고, 일치하는 것이 없을 때만 그것을 만듭니다:

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

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

이것은 두 번째 정점을 만든 것이 아니라 앞 절에서 만든 정점을 찾은 것입니다. **무엇을** 기준으로 일치시키는지 주의하십시오. 맵에 있는 모든 속성을 포함한 패턴 전체입니다. 속성을 하나 더하면 그 패턴은 이미 가진 정점을 설명하지 않게 되므로, `MERGE` 는 그 패턴이 설명하는 정점을 새로 만듭니다:

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

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

이제 둘이 됐습니다:

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

그러므로 어떤 대상에 대해 아는 모든 것을 맵에 담아 `MERGE` 하는 것은 중복 생성기입니다. 대상을 **식별하는** 키만으로 `MERGE` 하고 나머지 값은 그 뒤에 적용하는 것이 멱등합니다. `ON CREATE SET` 과 `ON MATCH SET` 이 그것을 위해 있습니다:

```
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}
```

다시 실행하면 `ON MATCH` 분기만 발동합니다:

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

`MERGE` 는 간선에도 같은 방식으로 동작하며, 간선 패턴을 일치시킨다는 것은 그 레이블, 방향, 속성 맵, **그리고** 양 끝점까지 일치시킨다는 뜻입니다. 끝점은 먼저 `MATCH` 로 바인딩해서 `MERGE` 가 간선만 판단하게 하십시오. 아래 간선은 이미 있으므로 아무것도 만들어지지 않습니다:

```
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]{}
```

계속하기 전에 Gloria Foster를 하나로 되돌립니다:

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

식별 속성에 걸어 둔 고유성 제약이 이 규율의 오래가는 형태입니다. 올바른 `MERGE` 를 쓰는 데 의존하지 않고 중복 자체를 거부하기 때문입니다. **인덱스와 제약**을 참고하십시오.

속성 바꾸기

속성 하나에 대한 `SET` 은 그 속성을 추가하거나 교체합니다:

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

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

`SET p += { ... }` 는 맵을 병합해 넣으며, 키를 추가하고 겹치는 키는 덮어씁니다:

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


© 2026 SKAI Worldwide Co., Ltd.

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

`SET p = { ... }` 는 속성 맵 전체를 교체합니다. 새 맵에 없는 것은 모두 사라집니다:

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

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

`+=` 와 `=` 의 차이는 `UPDATE ... SET one_column` 과 행 전체를 교체하는 것의 차이이며, 잘못 입력하기 쉽습니다.

null을 할당하는 것은 null을 저장하는 것이 아닙니다. 그 속성을 지웁니다:

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

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

그러므로 null로 설정한 속성과 애초에 없던 속성은 같은 상태이고, 그 뒤에 둘을 구별할 방법이 없습니다. `allow_null_properties` 가 이 동작을 바꿔 null을 저장하며, 기본적으로 꺼져 있습니다. `REMOVE` 는 같은 일을 명시적으로 표기한 것입니다:

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

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

간선의 속성은 정점의 속성과 똑같이 씁니다:

```
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}
```

참고

승격된 속성은 "속성은 무엇이든 담을 수 있다"의 유일한 예외입니다. 어떤 키 뒤에 타입 있는 컬럼이 놓이면, JSON 타입이 그 컬럼에 맞지 않는 쓰기는 중단됩니다. **속성을 컬럼으로 승격하기**를 참고하십시오.

지우기

`DELETE` 는 일치시킨 것을 지우고 그것을 돌려주므로, 무엇이 사라졌는지 볼 수 있습니다:

© 2026 SKAI Worldwide Co., Ltd.

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

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

아직 간선이 남아 있는 정점은 지울 수 없습니다. 끝점이 없는 간선은 존재할 수 없기 때문입니다:

```
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.
```

간선을 직접 지우거나:

```
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}
```

`DETACH DELETE`를 쓰십시오. 이것은 정점을 그에 닿아 있는 모든 간선과 함께 지웁니다:

```
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;
```

데이터셋이 원래대로 돌아왔습니다:

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

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

이 거부는 불편함이 아니라 안전장치입니다. 허브 정점에 `DETACH DELETE`를 쓰면 의도한 것보다 훨씬 많은 것이 지워질 수 있으므로, 무엇이든 지우기 전에 함께 지워질 간선을 일치시켜 세어 보십시오.

방금 쓴 것을 읽기

쓰기 절이 문장의 끝은 아닙니다. 그 출력 행은 파이프라인을 따라 계속 내려가고, 그 뒤의 읽기는 쓰기가 남겨 놓은 상태의 그래프를 봅니다:

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


© 2026 SKAI Worldwide Co., Ltd.

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

플래너는 다음 읽기가 실행되기 전에 쓰기의 효과를 구체화해서 그것을 보장합니다. 이것이 "eager" 계획이며, `EXPLAIN`도 그렇게 표시합니다. 그 비용과 그것을 금지하는 설정은 **쿼리 튜닝**이 다룹니다.

트랜잭션

그래프는 데이터베이스 안의 테이블이므로, 그래프에 쓰는 것은 평범한 트랜잭션 쓰기입니다. 켜야 할 것도 없고 두 번째 커밋을 생각할 필요도 없습니다. 한 트랜잭션 안의 Cypher 쓰기와 SQL 쓰기는 함께 커밋되거나 함께 롤백됩니다.

```
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
```

트랜잭션 안에서는 두 쓰기가 모두 있습니다. 롤백하면 둘 다 없습니다:

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

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

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

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

```
DROP TABLE import_log;
```

PostgreSQL이라는 점에서 따라오는 나머지 모든 것이 여기에도 따라옵니다. 격리 수준, 세이브포인트, 레이블 테이블에 대한 `SELECT ... FOR UPDATE`, 같은 정점에 쓰는 두 세션 사이의 교착 상태 탐지가 그렇습니다. 원한다면 대량 적재를 한 트랜잭션으로 할 수 있고, 트랜잭션 안의 `CREATE GRAPH`는 다른 DDL과 마찬가지로 롤백됩니다.

반대 방향으로는 관문이 둘 있습니다. SQL `INSERT`, `UPDATE`, `DELETE`, `MERGE`로 레이블에 행을 쓰는 것은 `enable_graph_dml`이 켜져 있지 않으면 거부됩니다. `ALTER TABLE`로 레이블의 모양을 바꾸는 것은 `enable_graph_ddl`이 켜져 있지 않으면 거부됩니다. 어느 쪽이든 그래프 카탈로그의 내용이 실제 데이터와 어긋나게 될 수 있습니다. **그래프가 어떻게 저장되는가**를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

다음에 갈 곳

- 데이터 적재 — 기존 테이블에 대한 `LOAD FROM`을 포함한 대량 적재입니다.
- 쿼리 조합하기 — `WITH`, `UNWIND`, 집계입니다. 모두 쓰기 절 앞에서 유용합니다.
- 보안 — 레이블에 대한 권한 부여와 행 수준 보안입니다.

쿼리 조합하기

Cypher 문장과 GQL 문장은 파이프라인입니다. 각 절은 행의 흐름을 받아 그것에 한 가지 일을 하고, 결과를 다음 절에 넘깁니다. `MATCH`가 행을 만들고, `WHERE`가 일부를 걸러 내고, `WITH`가 모양을 바꾸고, `RETURN`이 문장을 끝냅니다.

이것은 SQL과 다른 사고 모형입니다. SQL에서 `SELECT`는 이름 붙은 부분들이 정해진 순서로 평가되는 하나의 블록입니다. 여기서는 순서가 곧 작성한 순서입니다. 하나의 쿼리가 일치시키고, 집계하고, 그 집계를 걸러 내고, 살아남은 것에서 다시 일치시키고, 또 집계할 수 있습니다. 서브쿼리도, CTE도, `HAVING`도 없이 말입니다.

여기 있는 것은 모두 **예제 데이터셋**에서 실행됩니다.

WITH가 구분자다

`WITH`는 파이프라인의 한 부분을 끝내고 다음 부분을 시작합니다. `RETURN`과 똑같이 투영 목록을 받으며, 그 목록에 있는 것만 유효 범위에 남습니다:

```
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
```

`WITH` 뒤에는 `m`이 사라지고 `d`와 `films`가 남습니다. 그 뒤의 `WHERE`는 집계를 걸러 내며, SQL에서 `HAVING`이 필요한 일입니다.

`WITH` 안의 순서는 고정되어 있고 외워 둘 만합니다. `WHERE`가 먼저, 그다음 `ORDER BY`, 그다음 `SKIP`과 `LIMIT`입니다. 그래서 아래 쿼리는 영화 두 편을 먼저 고른 뒤 걸러 내는 것이 아니라, 다섯 편으로 걸러 낸 뒤 그중 두 편을 취합니다:

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

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

이름은 한 번만 넘길 수 있습니다:

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


© 2026 SKAI Worldwide Co., Ltd.

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

CTE와 무엇이 다른가

두 `WITH` 키워드는 모두 쿼리를 나누지만, 같은 키워드가 아닙니다.

SQL의 CTE는 **이름을 붙여 참조**합니다. `WITH films AS (...)`로 선언하고 그다음 `films`를 원하는 곳에서, 원하는 횟수만큼, 어떤 순서로든 언급합니다.

Cypher의 `WITH`는 **위치로 정해집니다**. 이름이 없고, 바로 다음에 온다는 것으로 참조되며, 정확히 한 번 소비됩니다. Cypher 파이프라인을 두 갈래로 나눈 뒤 다시 합치는 방법은 없습니다.

둘은 섞이지 않습니다. Cypher의 `WITH`로는 이름 붙은 서브쿼리를 도입할 수 없습니다. `WITH d AS (MATCH ...)`는 구문 오류입니다. 그러나 SQL의 CTE가 Cypher 쿼리를 담을 수 있으며, CTE의 동작이 필요할 때 그것을 얻는 방법이 이것입니다:

```
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
```

위에서 바깥 문장은 SQL입니다. SQL의 CTE로 시작하는 문장은 SQL 문장이기 때문입니다. 그 경계를 다루는 페이지는 **SQL과 Cypher 섞기**입니다.

Cypher 파이프라인이 `WITH`로 **시작할** 수 없는 이유도 그것입니다. 맨 앞의 `WITH`는 CTE로 읽히므로, 파서는 이름과 괄호로 감싼 쿼리를 찾습니다:

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

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

`LET`에는 그런 충돌이 없으므로, 계산된 값으로 문장을 시작할 때 쓰는 절은 이것입니다:

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

```
 x
---
 1
```

집계

집계 함수는 PostgreSQL의 집계 함수이고, 그룹화는 암시적입니다. 투영 목록에서 집계 함수 안에 들어 있지 않은 것은 모두 그룹화 키가 됩니다.

© 2026 SKAI Worldwide Co., Ltd.

```
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(*)` 는 행을 셉니다. `count(x)` 는 `x` 가 null이 아닌 행을 셉니다. 그래서 `count(x)` 는 `OPTIONAL MATCH` 뒤에서 유용합니다. `count(DISTINCT x)` 는 서로 다른 값을 세며, 이 둘의 차이는 보통 패턴이 퍼진 정도입니다:

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

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

`collect()` 는 가장 자주 꺼내 쓸 집계 함수입니다. 퍼진 것을 다시 한 행으로 되돌려 주기 때문입니다. 정점, 간선, 경로에 대해 쓰면 그 요소들의 네이티브 배열을 만들므로, 모아 놓은 요소를 계속 쓸 수 있습니다:

```
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"
```

속성에 대한 `collect()` 는 여전히 jsonb 리스트를 반환하므로, 저장된 데이터는 자기 모양을 유지합니다.

투영 전체에 붙인 `DISTINCT` 는 행의 중복을 없앱니다. 같은 대상에 여러 경로로 도달하는 패턴 뒤에 필요한 것이 이것입니다:

```
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` 는 `collect()` 의 반대입니다. 리스트를 받아 요소마다 행 하나씩을 만듭니다.

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

```
  y   |     title
------+---------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 1999 | "The Matrix"
 2010 | "Inception"
 2014 | "Interstellar"
```

같은 파이프라인에서 앞서 만든 리스트를 풀어내는 데도 씁니다. 집계로 걸러 낸 다음 다시 요소마다 행 하나로 돌아가는 방법이 이것입니다:

```
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`가 GQL 표기이며, 요소의 0부터 시작하는 인덱스를 위해 `WITH OFFSET`을 더해 줍니다. `UNWIND`를 참고하십시오.

정렬과 페이지 나누기

`ORDER BY`, `SKIP`, `OFFSET`, `LIMIT`은 그 자체로 절입니다. SQL처럼 `RETURN`이나 `WITH`의 끝에 놓일 수 있고:

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

```
            title             | year
------------------------------+------
 "Total Recall"               | 1990
 "Terminator 2: Judgment Day" | 1991
 "Jurassic Park"              | 1993
```

파이프라인 가운데에 홀로 설 수도 있습니다. 이것은 `RETURN` 형태로는 할 수 없습니다:

```
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
```

이것들이 문장을 **끝낼** 수는 없습니다. 쿼리는 여전히 `RETURN`, `FINISH`, 또는 쓰기 절로 끝나야 합니다.

정렬 키는 jsonb로 그려진 모습이 아니라 그 값 자체로 정렬되므로, 그 속성에 걸린 인덱스가 정렬에 쓰일 수 있습니다. 이것이 이 대목을 신경 써야 하는 이유 전부입니다: **인덱스와 제약**.

UNION

`UNION`은 컬럼 이름이 같은 두 결과 집합을 쌓고 중복을 없앱니다. `UNION ALL`은 중복을 남깁니다.

© 2026 SKAI Worldwide Co., Ltd.

```
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"
```

간선 레이블 택일 표기가 없는 것을 깔끔하게 우회하는 방법도 이것입니다. 좁은 패턴 둘을 `UNION`으로 합치면 간선 테이블 두 개를 스캔하지만, 레이블을 지정하지 않은 패턴 하나는 그 전부를 스캔합니다. **그래프 읽기**를 참고하십시오.

GQL 절들

ISO/IEC 39075의 네 개 절이, Cypher에서는 여러 절이 필요한 일을 해냅니다. 다른 모든 것과 같은 문장에서 허용되며 별도의 모드는 없습니다.

`LET`은 유효 범위의 어떤 것도 버리지 않으면서 계산된 값을 더합니다. `WITH`는 계속 쓰고 싶은 변수를 전부 나열하게 하지만, `LET`은 그 전부를 그대로 둡니다:

```
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
```

`FILTER`는 행을 걸러 내는 일만 하므로, 필터를 더 이상 `WITH`에 붙일 필요가 없습니다. 조건을 그냥 쓰거나 `FILTER WHERE` 표기를 씁니다.

`NEXT`는 한 쿼리를 다른 쿼리의 결과에 이어 붙입니다. `WITH` 없이 투영 뒤에서 걸러 내는 방법이 이것입니다:

```
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`는 아무것도 투영하지 않는 쿼리를 끝냅니다. 효과만을 위해 실행하는 문장에서 `RETURN`에 대응하는 것입니다:

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

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


© 2026 SKAI Worldwide Co., Ltd.

각각 레퍼런스 페이지가 있습니다. `LET`, `FILTER`, `NEXT`, `FINISH` 입니다.

서브쿼리

`CALL { }` 은 들어오는 행마다 서브쿼리를 한 번 실행하고 그 결과를 다시 조인합니다. 행마다 원하는 것이 표현식이 아니라 쿼리 전체일 때 쓰십시오. 행별 집계나 행별 상위 N개 같은 경우입니다.

이것에 관한 모든 것이 평범하지만 한 가지가 다르고, 그것은 오류를 내는 대신 답을 바꿔 버립니다.

> 중요
>
> 가져오기 목록이 없는 `CALL { }` 은 바깥 변수를 **가져오지 않습니다.** 본문 안에서 쓴 이름이 바깥에도 있다면, 그 이름은 제약이 없는 새 변수로 다시 바인딩되므로 서브쿼리가 현재 행이 아니라 그래프 전체를 대상으로 실행됩니다. 가져오기 목록을 항상 쓰십시오.

차이는 이렇습니다. `m` 이 바깥에서 바인딩되고 본문도 `m` 을 언급하는데, 답이 모든 영화에 대해 같습니다:

```
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
```

25는 그래프에 있는 모든 `acted_in` 간선의 수입니다. 서브쿼리의 `m` 은 바깥의 `m` 이 아니었습니다.

가져올 것을 이름으로 적으면 바로잡힙니다:

```
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 (*) { ... }` 은 유효 범위의 모든 것을 가져오고, `CALL { WITH m ... }` 은 같은 것의 옛 표기입니다. 세 형태 모두 같은 답을 냅니다.

서브쿼리는 바깥에서 이미 바인딩된 이름을 돌려줄 수 없습니다:

```
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
```

서브쿼리 본문에는 쓰기 절을 쓸 수 없습니다. `OPTIONAL CALL` 이 왼쪽 외부 조인 형태이며, 본문이 아무것도 내놓지 않아도 행을 만듭니다. `CALL { }` 과 `OPTIONAL CALL` 을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

값으로서의 서브쿼리

여섯 개의 값 표현식이 서브쿼리를 받아 `WHERE` 나 `RETURN` 에 넣을 수 있는 것으로 줄여 줍니다. 가져오기 목록이 필요하지 않습니다. 바깥 변수가 별도의 쿼리가 아니라 표현식으로서 유효 범위에 있기 때문입니다.

`EXISTS { }` 는 서브쿼리가 행을 하나라도 반환하면 참입니다:

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

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

`COUNT { }` 는 서브쿼리가 반환하는 행의 수입니다:

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

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

`COLLECT { }` 와 `ARRAY { }` 는 서브쿼리의 단일 컬럼을 리스트로 모읍니다:

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

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

`VALUE { }` 는 그 단일 값을 취합니다:

```
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"
```

그리고 `x IN { }` 은 그 결과에 속하는지를 검사합니다:

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


© 2026 SKAI Worldwide Co., Ltd.

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

여섯 중 가장 많이 쓸 것은 `EXISTS { }` 입니다. 패턴만으로는 조건이 되지 않기 때문입니다. **그래프 읽기**를 참고하십시오. 여섯 모두 **서브쿼리 표현식**에 정의돼 있습니다.

다음에 갈 곳

- **경로** — 길이를 미리 모르는 패턴입니다.
- **SQL과 Cypher 섞기** — CTE, 일반 테이블과의 조인, 뷰입니다.
- **쿼리 튜닝** — 파이프라인이 실행 계획으로는 어떻게 보이는지입니다.

경로

고정된 패턴은 모양을 이미 아는 질문을 합니다. `(a)-[]->(b)-[]->(c)` 는 언제나 두 홉입니다. 그래프가 있는 이유가 되는 질문은 모양을 모르는 질문입니다. 이 두 사람은 어떻게 연결되는가, 여기서 무엇에 도달할 수 있는가, 가장 값싼 경로는 무엇인가 같은 것입니다.

세 가지 도구가 그 질문에 답하며, 서로 다릅니다. 가변 길이 간선은 어떤 길이 범위 안의 모든 경로를 열거합니다. `shortestpath` 는 길이가 최소인 경로 하나를 찾습니다. `dijkstra` 는 **가중치**가 최소인 경로 하나를 찾는데, 그것은 흔히 같은 경로가 아닙니다.

여기 있는 것은 모두 **예제 데이터셋**에서 실행되며, 그 데이터셋의 `related_to` 간선에는 `weight` 가 있습니다. 아래 두 절은 정류장 네 개로 된 작은 그래프도 필요하므로 지금 만들어 둡니다. 이 그래프에는 비용 10인 A → B 직행 구간, 각각 비용 1인 세 구간의 우회로, 그리고 A → B → A를 순환으로 만드는 복귀 구간이 있습니다:

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

아래의 각 블록은 두 그래프 중 필요한 쪽으로 `graph_path` 를 설정합니다.

홉 수가 정해지지 않은 경우

간선 레이블 뒤의 `*` 가 그 길이를 범위로 만듭니다. 경계는 별표 뒤에 씁니다:

표기	뜻
`-[:related_to*2]->`	정확히 간선 두 개
`-[:related_to*1..3]->`	간선 한 개, 두 개, 또는 세 개
`-[:related_to*2..]->`	두 개 이상
`-[:related_to*..3]->`	최대 세 개, 최소 한 개

© 2026 SKAI Worldwide Co., Ltd.

표기	뜻
`-[:related_to*]->`	한 개 이상, 상한 없음
`-[:related_to*0..1]->`	그 정점 자신, 또는 간선 하나 거리

길이를 고정하면 패턴을 길게 늘여 쓴 것과 같이 동작합니다:

```
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"
```

범위는 그 안의 길이들을 합집합으로 묶으므로, 더 짧은 도달 결과도 함께 돌아옵니다:

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

하한이 0이면 시작 정점 자신이 포함되며, 이때 시작 정점은 간선 없이 그대로 일치합니다:

```
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"
```

상한을 생략하면 "간선이 이어지는 곳까지"를 뜻합니다:

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


© 2026 SKAI Worldwide Co., Ltd.

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

영원히 돌지는 않는다

순환이 있는 그래프에서 상한 없는 순회도 끝납니다. 어떤 경로도 같은 간선을 두 번 쓸 수 없기 때문입니다. `routelab`에는 A → B → A 순환이 있습니다. A에서 A로 돌아오는, 간선 여덟 개까지의 경로를 물어봅니다:

```
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"]
```

무한이 아니라 경로 두 개입니다. 각각 어떤 간선도 다시 쓰지 않는 보행입니다. **정점**은 반복될 수 있으며, 여기서 `A`가 양쪽 끝에 나타나는 것이 그렇습니다.

이 보장은 비용이 아니라 정확성에 관한 것입니다. 밀도가 높은 그래프에서 상한 없는 탐색은 간선이 바닥나기 전까지 엄청난 수의 경로를 열거할 수 있습니다. 상한이 얼마쯤이어야 할지 짐작이 가는 경우라면 언제든 범위에 상한을 주십시오.

경로를 바인딩하기

패턴 앞의 `p =`는 경로 전체를 바인딩하고, 세 함수가 그것을 분해합니다. `length(p)`는 간선 개수, `nodes(p)`는 순서대로의 정점, `relationships(p)`는 간선입니다.

```
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,
       [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]
```

`[x IN list | expression]` 형태는 리스트 컴프리헨션이며, 경로를 읽기 좋게 표현하는 방법입니다. `edges(p)`는 `relationships(p)`와 같은 뜻입니다.

레이블 두 종류를 지나는 경로는 속성마다 컴프리헨션이 하나씩 필요합니다. `person`에는 `title`이 없고 `movie`에는 `name`이 없기 때문입니다:

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

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

경로 자체를 반환하면 모든 요소가 전부 출력됩니다. 탐색할 때는 유용하지만 그 밖의 경우에는 과합니다. 경로의 모양을 간결하게 보는 방법은 `label(x)` 입니다.

최단 경로

`shortestpath` 는 가변 길이 간선이 하나 있는 패턴을 받아 길이가 최소인 경로 하나를 반환하고, 그런 경로가 없으면 아무 행도 반환하지 않습니다. 이것은 "경로들을 길이로 정렬해 하나만 취하는 것"이 아닙니다. 양쪽 탐색 경계가 만나는 순간 멈추는 양방향 탐색이므로, 버릴 경로를 열거하지 않습니다.

```
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"]
```

위 쿼리에서 세 가지는 의도한 것입니다. 간선 레이블을 지정하지 않았고 방향도 주지 않았습니다. 경로가 서로 다른 간선 레이블 두 종류를 지나며, 한쪽은 정방향으로 다른 쪽은 역방향으로 따라가기 때문입니다.

상한 6이 붙어 있는 것은, 레이블을 지정하지 않은 간선에 대한 상한 없는 최단 경로 탐색이 이 매뉴얼에서 가장 비싼 쿼리이기 때문입니다. 그리고 양 끝점이 변수에 바인딩돼 있습니다.

마지막 것은 반드시 지켜야 합니다:

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

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

이름을 쓰지 않을 때에도 양 끝에 이름을 붙이십시오. 변수를 생략하는 것이 낭비에 그치지 않고 아예 거부되는 곳은 여기 하나입니다.

`allshortestpaths` 는 같은 탐색이지만 최소 길이인 경로 중 하나가 아니라 전부를 반환합니다:

```
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"]
```

여기서는 경로가 하나입니다. 이 데이터셋의 `related_to` 간선이 트리를 이루기 때문입니다. 똑같이 짧은 경로가 여러 개인 그래프에서는 각각 한 행씩 나옵니다. 그 동률 자체가 관심사일 때 원하는 것입니다.

두 함수 모두 간선 레이블의 택일 표기를 받지 않습니다. `[:acted_in|directed*..4]` 는 `ERROR: multiple types for relationship not supported` 를 냅니다. 간선 레이블을 지정하지 말고 나중에 걸러 내십시오. **그래프 읽기**를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

가장 값싼 경로

`shortestpath`는 간선을 셉니다. 간선이 항상 서로 바꿔 놓을 수 있는 것은 아닙니다. 거리, 지연, 가격, 유사도를 담고 있을 수 있습니다. 그럴 때 간선을 세는 것은 틀린 질문에 답하는 것입니다. `dijkstra`는 가중치 표현식을 받아 그 합을 최소화합니다.

차이는 `routelab`에서 가장 잘 보입니다. A에서 B로 가는 쓸 만한 경로가 두 개 있습니다:

```
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`는 한 홉 경로를 고릅니다:

```
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`는 세 홉 경로를 고릅니다. 비용이 10이 아니라 3이기 때문입니다:

```
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"]
```

구문에서 틀리기 쉬운 것이 두 가지 있습니다.

패턴에는 가변 길이 간선이 아니라 간선 하나를 씁니다. `dijkstra`는 확장을 스스로 하므로, 한 홉만 쓰면 필요한 만큼 알아서 따라갑니다. 거기에 `*`를 쓰면 `ERROR: variable length relationship is not supported`로 거부되고, 간선을 둘 이상 쓰면 `ERROR: only one relationship is allowed`로 거부됩니다.

가중치 표현식은 간선 변수를 이름으로 부릅니다. `e.cost`는 패턴 안에서 바인딩된 `e`를 가리키며, 탐색이 확장될 때마다 간선별로 평가됩니다.

`(p, total) = dijkstra(...)`는 두 가지를 바인딩합니다. 경로와, 그 경로의 총 가중치입니다. 변수 하나 형태인 `p = dijkstra(...)`는 경로만 줍니다.

이 매뉴얼의 데이터셋에서는 가중치가 `related_to`에 있습니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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]
```

세 번째 인자는 한정자입니다. 탐색이 후보 간선마다 적용하는 불리언이며, 그래프를 미리 걸러 내는 대신 간선을 제외할 수 있습니다. 아래에서는 아무것도 살아남지 않습니다. The Matrix에서 Inception 쪽으로 나가는 유일한 경로가 가중치 4인 간선을 지나기 때문입니다:

```
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)
```

모든 조건이 그래야 하듯 한정자도 불리언이어야 합니다. 괄호 안의 `LIMIT n` 은 반환되는 경로 수의 상한을 정합니다. 둘 다 **`dijkstra`** 에 설명돼 있습니다.

경로 쿼리의 비용

가변 길이 간선의 실행 계획에는 그것만의 노드가 있고, 그 노드는 따라가는 레이블과 따라가는 범위를 이름으로 보여 줍니다:

```
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
```

읽어야 할 것은 `Graph VLE on related_to [1..3]` 입니다. 레이블을 생략하면 이것이 `Graph VLE on ag_edge` 가 되며, 매 홉마다 그래프의 모든 간선 레이블을 훑습니다:

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

```
                        QUERY PLAN
-----------------------------------------------------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
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` 는 `Hash2Side` 노드 한 쌍 위의 `Shortestpath VLE [min..max]` 로 나타납니다. 그 두 노드가 양방향 탐색의 두 탐색 경계입니다. `dijkstra` 는 `Dijkstra` 노드로 나타납니다. 세 경우 모두 홉마다 하는 일은 간선 레이블의 `(start, "end")` 또는 `("end", start)` 인덱스를 찾아보는 것입니다. 그래서 경로 쿼리의 비용은 탐색 경계가 닿는 간선의 개수입니다. 그것을 줄이는 것이 네 가지 있고, 효과가 큰 순서대로 적으면 이렇습니다:

1. **한쪽 끝을 고정하십시오.** 선택도가 높고 인덱스가 걸린 속성에 고정합니다. 알고 있는 정점 하나에서 출발하는 순회는 전부에서 출발하는 순회와 다른 쿼리입니다.
2. **간선 레이블을 이름으로 부르십시오.** 홉마다 테이블 하나를 보는 것과 전부를 보는 것의 차이입니다.
3. **길이에 상한을 두십시오.** 상한은 실행기가 쓸 수 있는 약속입니다.
4. **간선 레이블에 `ANALYZE` 를 돌리십시오.** 순회 비용 산정은 차수 추정이고, 분석하지 않은 간선 레이블에는 추정에 쓸 차수가 없습니다. 쿼리 튜닝을 참고하십시오.

정리

```
DROP GRAPH routelab CASCADE;
```

다음에 갈 곳

- 쿼리 튜닝 — 실행 계획 읽기, 그리고 계획의 모양을 바꾸는 설정들입니다.
- 그래프 읽기 — 이 페이지가 기반으로 삼는 고정 패턴들입니다.
- 벡터 검색 — 유사도 일치를 순회의 출발점으로 쓰는 방법입니다.

인덱스와 제약

속성 인덱스는 레이블의 `jsonb` 컬럼인 `properties` 에서 키 하나를 읽는 표현식에 걸린 PostgreSQL btree입니다. 이 한 문장이 언제 인덱스가 도움이 되는지에 대해 거의 모든 것을 설명합니다. 플래너가 여러분의 조건을 그 표현식과 맞출 수 있을 때 정확히 도움이 됩니다. 맞출 수 없으면 전혀 도움이 되지 않습니다.

이 페이지는 어떤 조건이 그 자격을 갖추는지 실험으로 확인합니다. 답이 대칭적이지 않고 몇 가지는 뜻밖이므로, 아래의 표들이 이 페이지의 요점입니다. 인덱스가 디스크에서 어떻게 생겼는지는 **그래프가 어떻게 저장되는가**가 보여 줍니다. 전체 구문은 `CREATE PROPERTY INDEX` 와 `CREATE CONSTRAINT` 에 있습니다.

측정할 그래프

예제 데이터셋에는 영화가 18편뿐이어서, 플래너가 테이블을 그냥 읽는 것보다 인덱스를 선호하기에는 너무 적습니다. 작업할 더 큰 레이블을 만들고, 페이지 끝에서 지웁니다:

```
CREATE TABLE seed_film AS
SELECT 'Film ' || lpad(g::text, 6, '0') AS title,
       1900 + (g % 130) AS year,
```


© 2026 SKAI Worldwide Co., Ltd.

```
       (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`는 인덱스만큼 중요합니다. 플래너가 통계를 갖고 있지 않은 인덱스는 플래너가 쓰기를 거절하는 인덱스입니다.

아래의 모든 실행 계획은 50,000행에서 플래너가 강요 없이 스스로 고른 것입니다. 논지를 만들려고 무언가를 끈 것은 없습니다.

인덱스가 처리할 수 있는 조건

같음 비교가 다른 모든 것을 견주는 기준입니다:

```
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)
```

찾아봐야 할 단어는 `Index Cond`입니다. 그 조건이 인덱스 탐색이 됐다는 뜻입니다. 대신 `Filter`로 나타난 조건은 스캔이 이미 가져온 행에 대해 평가된 것입니다. 그것이 바로 인덱스가 피하려던 일입니다.

패턴 안의 속성 맵은 같은 같음 비교로 컴파일되고 같은 실행 계획을 받습니다:

```
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)
```

범위 조건은 됩니다. btree가 순서를 갖기 때문입니다:

```
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))
```


© 2026 SKAI Worldwide Co., Ltd.

비교는 jsonb 자체의 순서로 이루어지고, 그 순서는 숫자를 수치로 정렬하므로 `>= 2025`는 뜻하는 대로 동작합니다. 그러나 jsonb는 값보다 **타입**을 먼저 정렬합니다. 문자열 `"2025"`를 담은 속성은 두 숫자 사이에 있지 않습니다.

리터럴로 된 `IN` 목록은 배열 탐색이 됩니다:

```
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[]))
```

인덱스가 처리할 수 없는 조건

`<>`는 인덱스를 쓸 수 없습니다. "이 값 하나가 아닌 것"이 테이블의 대부분이라는 평범한 이유 때문입니다:

```
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)
```

사람들이 걸리는 것은 문자열 조건입니다. `STARTS WITH`는 접두 범위여야 할 것처럼 보이기 때문입니다:

```
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)
```

그렇지 않습니다. `STARTS WITH`는 함수 호출인 `string_starts_with`로 컴파일되고, `properties.'title'`에 걸린 btree에는 그 함수에 대한 연산자 클래스 항목이 없습니다. 그래서 인덱스 경로가 거부된 것이 아니라 아예 없습니다.

`enable_seqscan`을 꺼도 인덱스 경로가 생기지 않습니다. 계획은 순차 스캔으로 남고 `Disabled: true`로 표시됩니다.

`CONTAINS`, `ENDS WITH`, `=~`도 똑같이 동작하며, 각각 자기 함수로 컴파일됩니다:

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


© 2026 SKAI Worldwide Co., Ltd.

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

`IS NULL`과 `IS NOT NULL`도 승격되지 않은 속성에서는 스캔으로 되돌아갑니다:

```
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)
```

그 이유는 알아 둘 만합니다. 승격 뒤에 `IS NULL`이 **다르게** 동작하는 까닭도 함께 알려 주기 때문입니다. `properties`에 `director` 키가 아예 없는 행에서는 `properties.'director'`가 null입니다. 그 키에 걸린 표현식 인덱스는 그런 행마다 null을 저장하므로, 인덱스를 거의 끝까지 읽어야 합니다.

승격된 컬럼에서는 같은 검사가 실제 컬럼에 대한 실제 `IS NULL`이고, btree가 그것에 곧바로 답합니다.

표

`title`에 `film_title_idx`, `year`에 `film_year_idx`를 걸고 `ANALYZE`를 돌린 상태에서 50,000행으로 측정했습니다:

조건	인덱스 사용
`f.title = 'x'`	예 — `Index Cond`
패턴 안의 `(f:film {title: 'x'})`	예 — `Index Cond`
`f.year >= 2025 AND f.year < 2030`	예 — `Bitmap Index Scan`
`f.title IN ['x', 'y']`	예 — `= ANY`가 붙은 `Index Cond`
`f.year <> 2020`	아니요 — `Seq Scan`
`f.title STARTS WITH 'x'`	아니요 — `string_starts_with` 필터
`f.title CONTAINS 'x'`	아니요 — `string_contains` 필터
`f.title ENDS WITH 'x'`	아니요 — `string_ends_with` 필터
`f.title =~ '^x'`	아니요 — `string_regex` 필터
`f.director IS NULL`	아니요 — `Seq Scan`
`f.title IS NOT NULL`	아니요 — `Seq Scan`
`ORDER BY f.title LIMIT n`	예 — `Index Scan`, 정렬 없음

© 2026 SKAI Worldwide Co., Ltd.

Cypher 조건에서 문자열 조건을 구해 주는 인덱스 종류는 없습니다. 그것들은 함수 호출이고, 어떤 연산자 클래스도 그것을 맡지 않습니다. 접두 검색이나 부분 문자열 검색이 필요하다면 **전문 검색**이 그 도구입니다. SQL에서 그 컬럼에 닿는 방법도 있습니다. **아래**를 참고하십시오.

정렬

인덱스는 같은 속성에 대한 `ORDER BY`를 처리해 줍니다. 그래서 "속성 기준 상위 N개" 쿼리에는 정렬이 전혀 필요하지 않습니다:

```
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
```

이것이 되는 이유는 정렬 키가 jsonb로 그려진 모습이 아니라 그 값 자체로 정렬되기 때문입니다. 인덱스가 담고 있지 않은 표현식으로 정렬하면 여전히 정렬 작업이 일어납니다.

복합 인덱스와 부분 인덱스

속성 인덱스는 여러 키를 받으며, 가장 왼쪽 접두 규칙은 PostgreSQL의 것과 같습니다:

```
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'::j
sonb))
   ->  Bitmap Index Scan on film_genre_year_idx
         Index Cond: ((properties.'genre' = '"Comedy"'::jsonb) AND (properties.'year' >= '2025
'::jsonb))
```

`year`만으로 하는 쿼리는 이것을 쓸 수 없고 `year`에 걸린 단일 키 인덱스로 돌아갑니다. 그러므로 SQL에서 하듯 같음 비교를 앞에 두는 순서로 키를 배치하십시오.

인덱스에 붙인 `WHERE` 절이 그것을 부분 인덱스로 만듭니다. 큰 레이블의 작고 자주 쓰이는 일부에 인덱스를 거는 방법이 이것입니다:

```
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"'::j
sonb))
```


© 2026 SKAI Worldwide Co., Ltd.

```
   ->  Bitmap Index Scan on film_comedy_year_idx
         Index Cond: (properties.'year' >= '2025'::jsonb)
```

`genre` 검사가 인덱스 조건에서 빠졌습니다. 이 인덱스의 모든 항목이 이미 코미디이기 때문이며, 그 검사는 재확인에만 남습니다. 이 인덱스는 `year` 에 걸린 전체 인덱스의 4분의 1 크기입니다. 524,288 바이트 대 2,056,192 바이트입니다. 플래너는 다른 두 대안보다 이것을 골랐습니다.

`CREATE PROPERTY INDEX` 는 `USING btree | hash | gin | brin | gist | spgist` , `CONCURRENTLY` , `IF NOT EXISTS` , `INCLUDE` , 그리고 테이블스페이스도 받습니다. 모두 PostgreSQL에서와 같은 뜻입니다.

> 참고
>
> `CREATE PROPERTY INDEX (...)` 안의 모든 맨 이름은 **속성 키**이며, `properties` 라는 단어도 그렇습니다. `CREATE PROPERTY INDEX ON film USING gin (properties)` 는 `properties` 라는 이름의 속성에 인덱스를 거는 것이고, 그런 속성을 가진 영화는 없습니다. `properties` 자체에 인덱스를 걸려면 평범한 SQL을 쓰십시오 — `CREATE INDEX film_bag_idx ON idxlab.film USING gin (properties)` — 그리고 그 인덱스는 `properties @> '{"genre": "Comedy"}'` 같은 SQL 조건만 쓸 수 있다는 점에 유의하십시오. Cypher의 속성 읽기는 `properties.'genre'` 로 컴파일되고, `properties` 에 걸린 GIN 인덱스는 그것과 맞지 않습니다.

승격이 바꾸는 것

속성을 승격하면 실제 타입이 있는 컬럼이 생기고, 그 뒤로는 모든 조건이 `properties` 가 아니라 그 컬럼을 대상으로 컴파일됩니다. 왜 그리고 어떻게 하는지는 **속성을 컬럼으로 승격하기**가 다룹니다. 여기서 중요한 것은 그것이 위 표의 답을 좋은 쪽으로도 나쁜 쪽으로도 바꾼다는 점입니다.

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

이미 갖고 있는 인덱스에 승격이 무슨 일을 하는지부터 봅니다. `properties.'year'` 에 걸린 인덱스는 어떤 것과도 맞지 않게 됩니다. 더 이상 어떤 쿼리도 그 표현식으로 컴파일되지 않기 때문입니다:

```
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))
```

인덱스는 여전히 거기 있고, 쓰기마다 여전히 유지되며, 이제 아무것도 그것을 쓰지 않습니다. 그러므로 승격은 항상 두 단계입니다. 승격하고, 그다음 컬럼에 인덱스를 다시 만듭니다.

속성 인덱스로 다시 만드는 것은 맨 키에 대해서는 잘 됩니다. 그것이 흔한 경우이고 아래의 경우입니다. 그 키에 대한 **표현식**은 다른 형태가 필요합니다. `CREATE PROPERTY INDEX` 는 맨 키는 승격된 컬럼으로 컴파일하지만, 그 키에 대한 표현식은 jsonb `properties` 로 컴파일하는데 그것은 쿼리가 읽는 대상이 아닙니다. 그런 인덱스는 컬럼에 대한 SQL로 쓰십시오. `\dGi` 가 그것도 여전히 나열해 줍니다:

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

표현식 속성 인덱스는 `properties` 에 걸린다 절을 참고하십시오.

```
DROP PROPERTY INDEX film_year_idx;
DROP PROPERTY INDEX film_title_idx;
DROP PROPERTY INDEX film_genre_year_idx;
```


© 2026 SKAI Worldwide Co., Ltd.

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

레이블은 이제 이런 모양입니다. 새 인덱스는 표현식 인덱스가 아니라 일반 컬럼 인덱스입니다:

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

같음 비교와 범위 비교가 이제 jsonb가 아니라 네이티브 값을 비교합니다:

```
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` 이 인덱스로 처리 가능해집니다. 전에는 그렇지 않았습니다:

```
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` 는 인덱스 스캔에서 인덱스 전용 스캔으로 나아집니다. 쿼리가 원하는 값이 인덱스 안에 있기 때문입니다:

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

```
                      QUERY PLAN
------------------------------------------------------
```


© 2026 SKAI Worldwide Co., Ltd.

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

그리고 두 조건은 **나빠집니다**. `IN`은 더 이상 인덱스를 쓰지 않습니다. jsonb 리스트와 비교하려고 컬럼이 다시 jsonb로 감싸이기 때문입니다:

```
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`도 같은 방식으로 감싸이므로, 승격이 이것을 구해 주지도 않습니다:

```
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)
```

그러나 그 컬럼은 실제 `text` 컬럼이므로 SQL에서는 실제 `text` 연산자로 그 컬럼을 조회할 수 있고, 그러면 **진짜** 접두 범위 검색이 됩니다:

```
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)
```

자주 쓰는 키에 대한 접두 검색의 실용적인 답이 이것입니다. 속성을 승격하고 접두 조건을 SQL로 쓰되, Cypher 부분이 필요하면 그것을 감싸십시오. **SQL과 Cypher 섞기**를 참고하십시오.

승격 뒤의 표

조건	승격 전(jsonb)	승격 후(타입 컬럼)
`f.title = 'x'`	예	예
`f.year >= a AND f.year < b`	예	예
`f.title IN ['x', 'y']`	예	**아니요** — jsonb로 감싸임
`f.year <> 2020`	아니요	아니요

© 2026 SKAI Worldwide Co., Ltd.

조건	승격 전(jsonb)	승격 후(타입 컬럼)
`f.title STARTS WITH 'x'`	아니요	아니요
`f.title CONTAINS 'x'`	아니요	아니요
`f.title =~ '^x'`	아니요	아니요
`f.title IS NULL`	아니요	**예**
`ORDER BY f.title LIMIT n`	예 — 인덱스 스캔	예 — 인덱스 전용 스캔
SQL `title LIKE 'x%'`	해당 없음	예 — 접두 범위

레이블 계층에 걸쳐서

인덱스는 하나의 테이블에 속합니다. 부모 레이블에 대한 패턴은 부모와 그 아래의 모든 레이블을 스캔하고, 그 각각에는 자기 인덱스가 필요합니다. 확인할 계열을 하나 만들어 봅니다:

```
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')});
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;
```

`work`에만 인덱스가 있고, 모든 자식은 전부 읽힙니다:

```
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)
```

자식마다 인덱스를 걸면 `Append`의 모든 가지가 인덱스를 얻습니다:

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


© 2026 SKAI Worldwide Co., Ltd.

```
                              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` 자신은 순차 스캔을 유지합니다. 자기 행이 없고, 빈 테이블을 읽는 것이 인덱스를 여는 것보다 값싸기 때문입니다. 그것이 옳은 판단입니다. 그리고 이것이 눈여겨봐야 할 모양이기도 합니다. 계열을 검색하는 속성마다, 계층은 레이블당 인덱스 하나의 비용을 영구히 요구합니다.

제약

형태가 두 가지 있고, 각각 내부적으로는 PostgreSQL의 제약 두 종류에 대응합니다.

`ASSERT <key> IS UNIQUE` 는 그 키의 표현식에 배제 제약을 만듭니다:

```
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});
```

```
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>` 는 검사 제약을 만듭니다:

```
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"}).
```

둘 다 보통 제약으로 보이고, 고유성 제약은 그 표현식에 걸린 실제 btree가 뒤를 받치므로 그 키를 찾는 인덱스 역할도 겸합니다:

```
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 =)
```

두 제약 모두 그 속성이 없는 정점에 대해서는 아무것도 말하지 않습니다. 없는 키는 null로 읽히고, null은 다른 null과 같지 않으며, 알 수 없는 값에 대한 검사 제약은 통과합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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
```

이메일이 아예 없는 계정 둘이 아무 문제 없이 공존합니다. 어떤 키가 고유해야 하는 데다 반드시 있어야 한다면, null이 아님을 확인하는 제약을 하나 더 추가하십시오. `properties`가 그것을 대신 해 주지 않습니다.

`CREATE UNIQUE PROPERTY INDEX`가 고유성을 표기하는 또 다른 방법이며, 배제 제약이 아니라 고유 인덱스를 만듭니다:

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

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

빠른 고유 조회에는 이쪽을 택하십시오. `pg_constraint`와 덤프에서 직접 정한 이름으로 보고 싶은 규칙이라면 이름 붙은 제약을 택하십시오. 둘 다 자식 레이블에 상속되지 않습니다. 부모에 걸린 제약은 부모 자신의 행만 제약합니다.

실제 작업 순서

"왜 이게 느린가" 하는 질문 대부분에 답하는 순서입니다:

1. 관련된 레이블에 `ANALYZE`를 돌리십시오. 간선 레이블도 포함합니다. 분석하지 않은 간선 레이블은 빠진 인덱스보다 비쌉니다. **쿼리 튜닝**을 참고하십시오.
2. 패턴의 **출발점**을 정하는 속성에 인덱스를 거십시오. 시작 정점을 골라 내는, 선택도가 높은 같음 비교 하나가 그것입니다. 그 뒤의 모든 것은 이미 존재하는 간선 인덱스를 통한 조인입니다.
3. 그 조건이 `Filter`가 아니라 `Index Cond`로 도착하는지 `EXPLAIN`으로 확인하십시오.
4. 그러고 나서야 속성 승격을 검토하십시오. 승격은 탐색이 아니라 읽기 자체가 비용일 때의 답입니다.

정리

```
DROP GRAPH idxlab CASCADE;
```

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

다음에 갈 곳

- **속성을 컬럼으로 승격하기** — 위 이야기의 나머지 절반입니다.
- **쿼리 튜닝** — 실행 계획의 모양, 통계, 그리고 그것들을 바꾸는 설정입니다.

© 2026 SKAI Worldwide Co., Ltd.

- 전문 검색 — 부분 문자열과 접두 일치에 대한, 인덱스로 처리되는 답입니다.

속성을 컬럼으로 승격하기

레이블에 스키마가 없는 것은 속성이 요소마다 하나씩 있는 `jsonb` 문서 안에 있기 때문입니다. 그 대가로 두 가지를 치릅니다.

`m.year`를 읽는 것은 쿼리가 닿는 모든 행에 대해 JSON 문서 안에서 `year` 키를 찾는 일입니다. 그리고 그 읽기가 내놓는 값은 jsonb 값이므로, 그것에 걸린 인덱스는 jsonb에 걸린 인덱스입니다. 결코 `integer`나 `vector`에 걸린 인덱스가 아닙니다.

승격은 고른 키 하나에 대해 그 두 비용을 없앱니다. `ALTER VLABEL film ADD COLUMN year int GENERATED`는 `year` 속성에 실제 `integer` 컬럼을 주고, 그 컬럼은 `properties` 옆에 저장되며 서버가 유지합니다. 또한 Cypher 컴파일러가 `f.year`를 그 컬럼에서 읽도록 만듭니다.

쿼리는 바뀌지 않습니다. 속성에 관한 것도 아무것도 바뀌지 않습니다. 여전히 속성이고, 레이블은 다른 모든 키에 대해 여전히 스키마가 없으며, 승격은 되돌릴 수 있습니다.

이 페이지는 그 교환이 언제 할 만한지, 그리고 그것으로 무엇을 감당하게 되는지를 다룹니다. DDL 구문은 **`ALTER VLABEL`**과 **`CREATE VLABEL`**에 있습니다.

승격을 시험할 자기 레이블

승격은 레이블의 모양을 바꾸는 일이므로, 매뉴얼이 공유하는 그래프가 아니라 사본에서 작업하십시오. 레이블 테이블에 대한 `LOAD FROM`은 그 `properties`를 그대로 옮겨 줍니다:

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

이렇게 하면 모양을 마음대로 바꿔도 되는 그래프에 **예제 데이터셋**의 영화 18편이 생깁니다. 이 페이지 전체가 거기서 실행되고 끝에서 그것을 지웁니다.

승격이 하는 일

승격 전에는 `year`에 대한 필터가 `properties` 안으로 들어갑니다. `EXPLAIN (VERBOSE)`가 속성 읽기를 `properties.'key'`로 보여 줍니다:

```
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)
```

그것을 승격합니다:

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

```
\d promolab.film
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                                  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
```

이 컬럼은 `GENERATED ALWAYS AS ... STORED` 입니다. `properties` 에서 키 하나를 꺼내는 표현식을 가진 평범한 PostgreSQL 생성 컬럼입니다. `ag_property_text(properties, 'year', 'n')` 은 값이 그 컬럼이 담을 수 있는 종류일 때만 그 키의 텍스트를 반환합니다. `'n'` 은 숫자, `'s'` 는 문자열, `'b'` 는 불리언입니다.

나머지는 `integer` 로의 형 변환이 합니다. 아무도 이 컬럼에 쓰지 않습니다. 서버가 삽입과 갱신마다 계산합니다.

같은 쿼리가 이제 컬럼을 대상으로 필터합니다:

```
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)
```

`year` 만 승격했으므로 `f.title` 은 여전히 `properties` 읽기입니다. 답은 그대로입니다:

```
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
```

기존 행은 `ALTER` 의 일부로 채워지므로, 컬럼은 생기는 순간 이미 완전합니다. 그리고 값은 **양쪽**에 있습니다. `properties` 도 여전히 그것을 담고 있습니다:

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

이 중복은 의도한 것입니다. 속성은 그대로 속성이므로, `properties`를 읽는 클라이언트는 늘 보던 것을 봅니다. 덤프는 데이터를 `properties`에 한 번 담고 컬럼은 다시 생성합니다. 컬럼을 지워도 잃는 것이 없습니다. 대가는 저장 공간입니다. 그 값을 두 번 치르는 것입니다.

속성 키에서 컬럼으로의 대응은 `ag_label_property`에 기록됩니다:

```
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
```

이 행이 읽기를 컬럼으로 해석되게 만듭니다. 이것이 없으면 컬럼은 존재하지만 Cypher가 그것을 알지 못합니다. **긴 형태는 아무것도 등록하지 않는다**를 참고하십시오.

승격된 속성 읽기는 컬럼의 타입을 갖는다

Cypher의 `RETURN`은 속성 읽기를 jsonb로 돌려주므로 `RETURN f.title`은 `"The Matrix"`를 따옴표와 함께 출력합니다. 승격된 키를 읽는 것이 그 예외입니다. 컬럼에 선언된 타입으로 돌아옵니다. 아래에서 `year`는 승격됐고 `title`은 아닙니다.

```
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
```

값과 순서는 어느 쪽이든 같지만, 클라이언트가 받는 **타입**은 같지 않습니다. Cypher의 모든 컬럼이 JSON으로 올 것이라 기대하는 드라이버는 여기서 `integer`를 보게 됩니다.

보통 이것은 단순화입니다. **평범한 값**으로 가는 가장 짧은 길이기 때문입니다. 그러나 결과의 모양이 바뀌는 것이므로, 그것을 받아 쓰는 쪽과 맞는지 확인할 만합니다.

승격된 컬럼에 인덱스 걸기

승격된 키에 대한 `CREATE PROPERTY INDEX`는 `properties`에 대한 표현식 인덱스가 아니라 그 컬럼에 대한 평범한 btree를 만듭니다. 플래너는 다른 컬럼 인덱스와 똑같이 그것을 씁니다. 그 인덱스가 어떤 조건을 처리할 수 있는지, 그리고 승격 뒤에 **나빠지는** 두 조건이 무엇인지는 **인덱스와 제약**이 측정해 놓았습니다. 짧게 적으면 이렇습니다:

- 같음 비교, 범위, `ORDER BY`가 처리되고, `ORDER BY`는 인덱스 전용 스캔으로 나아집니다.
- `IS NULL`이 인덱스로 처리 가능해집니다. `properties`에서는 그렇지 않았습니다.
- `IN`과 `STARTS WITH`는 인덱스를 쓰지 않게 됩니다. jsonb 피연산자와 만나려고 컬럼이 다시 jsonb로 감싸이기 때문입니다.

여기서 규칙 하나가 따라오고, 놓치기 쉽습니다. **키를 승격하는 순간 `properties`에 걸린 인덱스는 죽습니다.** 더 이상 아무것도 `properties.'year'`로 컴파일되지 않으므로, 그 표현식에 걸린 인덱스는 어떤 쿼리와도 일치하지 않아 쓰이지 않습니다. 그런데도 쓰기마다 계속 유지됩니다. 승격은 항상 두 단계입니다. 승격하고, 그다음 컬럼에 인덱스를 다시 만듭니다.

레이블과 함께 승격 컬럼 선언하기

레이블을 만들 때는 컬럼을 레이블 정의 안에 씁니다:

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE VLABEL doc (words int GENERATED, tag text GENERATED (category));
```

`words int GENERATED`는 `words` 속성을 같은 이름의 컬럼으로 승격합니다. 괄호를 쓴 형태는 다른 키를 승격합니다. `tag text GENERATED (category)`는 "컬럼 `tag`를 속성 `category`에서 채운다"는 뜻입니다. 원하는 키가 컬럼 이름으로 쓸 수 없거나 적절하지 않을 때 쓰십시오.

카탈로그는 컬럼 이름이 아니라 키를 기록합니다:

```
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
---------+----------+---------+---------
 doc     | category | tag     | text
 doc     | words    | words   | integer
```

그래서 그 컬럼으로 해석되는 Cypher 읽기는 키인 `d.category`입니다. `d.tag`는 아무도 설정한 적 없는 속성입니다:

```
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`은 간선 속성을 같은 방식으로 승격합니다. 순회에 쓰는 가중치라면 해 둘 만합니다.

확장 타입

컬럼의 타입은 확장에서 올 수도 있습니다. 승격이 미세 최적화 이상이 되는 경우가 바로 이것입니다. 속성으로 저장된 임베딩은 JSON 배열이므로, 모든 유사도 쿼리가 그것을 형 변환해야 합니다. 매번, 모든 행에 대해 `p.emb::vector(4)`를 해야 합니다. 승격하면 그 형 변환이 쓰기 시점에 한 번만 일어납니다:

```
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 ->>
```


© 2026 SKAI Worldwide Co., Ltd.

```
'emb'::text))::vector(4)) stored
Indexes:
    "page_pkey" PRIMARY KEY, btree (id)
Inherits: promolab.ag_vertex
```

생성 표현식을 보십시오. JSON에 대응물이 없는 타입에 대해서는 서버가 `ag_property_text` 대신 평범한 `->>` 를 쓰고, 그 텍스트를 받아들일지 거부할지는 타입 자신의 입력 함수에 맡깁니다. `vector(4)` 는 대괄호 안의 숫자 네 개가 아닌 것은 모두 거부합니다.

그러면 그 컬럼에 실제 벡터 인덱스를 걸 수 있습니다. 인덱스 정의에 형 변환 표현식이 필요하지 않습니다. 남아 있는 형 변환이 없기 때문입니다. 먼저 인덱스를 고를 만한 행 수를 적재합니다:

```
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;
```

```
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;
```

그리고 Cypher 유사도 검색이 그것을 사용합니다:

```
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))
```

같은 쿼리의 승격 전 형태는 모든 행에 대해 거리를 계산하고 그것을 정렬해야 합니다:

```
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
```

승격된 컬럼에 `::vector(4)` 를 그대로 남겨 두어도 거부되지 않고 접혀 사라지므로, 기존 쿼리는 계속 동작하고 인덱스도 계속 씁니다.

`((emb::vector(4)) vector_cosine_ops)` 같은 **형 변환 표현식**에 속성 인덱스를 거는 것보다 승격을 택하십시오. 승격된 컬럼에 걸린 인덱스는 일반 컬럼 인덱스이고, 덤프·복원을 거쳐도 더 오래가는 형태입니다. **벡터 검색**을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

승격의 대가

승격된 키는 더 이상 스키마에서 자유롭지 않다

이것이 진짜 교환입니다. 레이블의 다른 모든 키는 여전히 어떤 JSON 값이든 받습니다. 승격된 키는 그 컬럼이 담을 수 있는 것만 받고, 맞지 않는 쓰기는 중단됩니다:

```
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"
```

```
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"
```

여기서 서로 다른 검사 두 가지가 보입니다. 첫째는 종류 검사입니다. 문자열은 무엇이라고 적혀 있든 숫자가 아닙니다. 둘째는 컬럼 타입 자신의 검사입니다. 확장 타입에는 둘째만 적용됩니다:

```
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"
```

그 키를 아예 빼는 것은 언제나 괜찮습니다. 없는 속성은 null 컬럼을 만들며, 없는 속성이 null로 읽히는 것과 정확히 같습니다:

```
CREATE (:film {title: 'Undated'});
MATCH (f:film {title: 'Undated'}) RETURN f.year AS year;
```

```
 year
------
```

그러므로 승격은 속도에 관한 결정이면서 그만큼 데이터 품질에 관한 결정입니다. 운영 중인 레이블의 키를 승격하기 전에, 기존 값이 모두 들어맞는지 확인하십시오. `ALTER`는 이미 있는 행에도 같은 검사를 적용하고, 맞지 않는 첫 행에서 실패합니다:

```
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.
```

이 실패는 유용한 종류입니다. 아무것도 커밋되기 전에 그 레이블의 데이터가 생각과 다르다는 것을 알려 줍니다.

© 2026 SKAI Worldwide Co., Ltd.

ALTER는 테이블을 다시 쓴다

저장되는 생성 컬럼을 추가하면 레이블의 힙을 다시 쓰고, 그동안 `ACCESS EXCLUSIVE` 락을 잡습니다. 18행짜리 레이블에서는 즉시 끝납니다. 큰 레이블에서는 다른 `ALTER TABLE` 재작성과 마찬가지로 계획을 세우십시오. 실제로 그런지 보고 싶다면 테이블의 파일 번호를 확인하십시오:

```
SELECT relfilenode FROM pg_class WHERE oid = 'promolab.film'::regclass;
```

`ALTER` 전의 번호와 그 뒤의 번호가 다릅니다.

승격된 컬럼은 자기 콜레이션을 가질 수 없다

속성 맵은 문자열을 데이터베이스 기본 콜레이션으로 비교합니다. 다른 콜레이션이 붙은 컬럼은 같은 값을 다르게 정렬하고 다르게 비교합니다. 그러면 "이 읽기에 둘 중 어느 쪽이 답하는가"가 속도에 관한 결정이 아니라 결과에 관한 결정이 됩니다.

그래서 거부되며, 콜레이션이 직접 적혀 있지 않고 도메인을 통해 들어오는 경우도 마찬가지입니다:

```
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.
```

Cypher 쿼리 위의 뷰는 컬럼을 붙잡아 둔다

뷰는 한 번 컴파일되며, 어떤 속성을 컬럼으로 해석한 Cypher 쿼리 위의 뷰는 그 컬럼에 대한 의존성을 갖습니다:

```
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.
```

이것이 옳습니다. 그렇지 않으면 뷰가 조용히 `properties`를 읽기 시작할 것입니다. 다만 그 말은 뷰가 하나의 약속이라는 뜻입니다. 뷰를 지우고, 컬럼을 지우고, 뷰를 다시 만드십시오. 이 의존성은 `pg_get_viewdef`에서 보이고, 거기 나오는 컴파일된 SQL이 `f.year`를 이름으로 부릅니다. **Cypher 쿼리 위의 뷰는 어떻게 보이는가**를 참고하십시오.

```
DROP VIEW recent_films;
```

표현식 속성 인덱스는 `properties`에 걸린다

맵 키와, 표현식 안에 든 같은 키는 `CREATE PROPERTY INDEX`에서 다르게 해석됩니다. 맵 키는 승격된 컬럼이 됩니다. 그 키에 대한 표현식은 jsonb `properties`에 대한 표현식이 됩니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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
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)
```

두 경우 모두 쿼리는 컬럼을 읽으므로, 쿼리가 쓸 수 있는 인덱스를 주는 것은 맨 키 형태입니다.

`\dGi` 는 둘 다 Cypher 형태로 보고하며, 둘 다 `title` 로 표기합니다:

```
\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)
```

그러므로 어떤 레이블의 인덱스가 둘 중 무엇에 걸려 있는지 알고 싶으면 `pg_get_indexdef` 를 읽으십시오.

승격된 키에 인덱스를 거는 방법은 두 가지입니다. 맨 키로 걸면 쿼리가 그 컬럼을 직접 씁니다. 또는 표현식 자체에 인덱스를 걸어야 한다면, 컬럼에 대한 SQL로 쓰십시오. 그것이 쿼리가 읽는 것과 맞습니다:

```
CREATE INDEX film_lower_col_idx ON promolab.film (lower(title));
```

어느 쪽이든, 어떤 키를 승격하기 전에 그 키를 읽는 표현식 인덱스를 지우고, 위 두 형태 중 하나로 다시 만드십시오.

긴 형태는 키 하나가 아니면 아무것도 등록하지 않는다

긴 방식으로 쓴 생성 컬럼은 평범한 생성 컬럼입니다. 그 표현식이 `properties` 에서 키 하나를 꺼내는 것과 정확히 같을 때만 승격으로 인식됩니다. `(properties ->> 'key')::type` , 또는 덤프가 내보내는 `ag_property_text` 형태입니다. 키 하나보다 많은 것에서 끌어냈거나 계산한 것은 어떤 대응도 등록하지 않습니다:

```
CREATE VLABEL post;
ALTER VLABEL post ADD COLUMN body_len int GENERATED ALWAYS AS (length(properties->>'body')) ST
ORED;
CREATE (:post {body: 'hello there'});
```

컬럼은 있고 SQL은 그것을 읽을 수 있습니다:

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT id, body_len, properties FROM promolab.post;
```

```
 id  | body_len |       properties
-----+----------+------------------------
 8.1 |       11 | {"body": "hello there"}
```

Cypher는 읽을 수 없습니다:

```
MATCH (p:post) RETURN p.body_len AS body_len;
```

```
 body_len
----------
```

`p.body_len` 은 `body_len` 이라는 이름의 속성을 읽는데 그런 속성을 가진 글은 없고, 그 컬럼의 값을 가져올 속성 키도 없습니다. 그래서 승격되지 않았는데도 승격된 것처럼 보입니다. 승격에는 `GENERATED` 축약형을 쓰고, SQL에서만 읽으려는 파생 컬럼에는 긴 형태를 쓰십시오.

해석 기능 끄기

`enable_property_promotion` 은 Cypher의 속성 읽기가 컬럼으로 해석되는지를 제어합니다. 기본적으로 켜져 있습니다. 끄면 같은 쿼리가 `properties` 로 돌아갑니다:

```
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)
```

그리고 같은 답을 냅니다. `properties` 가 여전히 그 값을 담고 있기 때문입니다. 이 설정의 목적이 그것입니다. 컬럼을 지우지 않고도, 예상하지 못한 실행 계획의 원인이 승격인지 확인하는 것입니다. 이 설정이 컬럼의 유지를 멈추지는 않습니다.

승격 되돌리기

```
ALTER VLABEL film DROP COLUMN year;
```

컬럼과 그 카탈로그 행이 사라지고, 그것에 걸린 인덱스도 함께 사라지며, 읽기는 같은 결과를 내면서 `properties` 로 되돌아갑니다:

```
MATCH (f:film) WHERE f.year > 2010 RETURN count(f) AS films;
```

```
 films
-------
 5
```

잃는 것은 없습니다. 컬럼에만 저장된 것은 처음부터 없었기 때문입니다. 먼저 정리해야 할 것은 그 컬럼을 붙잡아 둔 뷰뿐입니다.

© 2026 SKAI Worldwide Co., Ltd.

언제 승격할까

네 가지가 모두 참일 때 키를 승격하십시오:

- **쿼리가 그 키로 필터하거나 조인하거나 정렬한다.** 단순히 투영하는 것이 아니어야 합니다. 반환만 하는 키는 얻는 것이 없습니다. `properties` 읽기가 이미 그것을 얻는 가장 값싼 방법입니다.
- **값이 균일하다.** 어떤 행은 숫자를, 어떤 행은 문자열을 담고 있다면 `ALTER`가 거부하며, 그것들을 한 타입으로 밀어 넣는 것은 튜닝 변경이 아니라 데이터 변경입니다.
- **값 자신의 타입에 인덱스를 걸고 싶다** — 실제 `integer`에 대한 범위, 인덱스 전용 `ORDER BY`, `vector`에 대한 HNSW 인덱스 같은 것입니다.
- **레이블이 읽기 비용이 문제가 될 만큼 크다.** 18행에서는 여기 나온 차이가 측정할 수 없을 만큼 작습니다.

모든 정점에 나타난다는 이유로 키를 승격하지는 마십시오. 균일성은 전제 조건이지 이유가 아닙니다. 그리고 나빠지는 두 조건 `IN`과 `STARTS WITH`를 애플리케이션이 실제로 보내는 것과 맞춰 확인하십시오.

정리

```
DROP GRAPH promolab CASCADE;
```

```
DROP TABLE seed_page;
DROP DOMAIN sortkey;
```

다음에 갈 곳

- **인덱스와 제약** — 승격 전과 후의 인덱스 처리 가능성 표입니다.
- **쿼리 튜닝** — 실행 계획을 좌우하는 다른 선택들 사이에서 승격이 놓이는 자리입니다.
- **벡터 검색** — 승격된 컬럼이 있을 때와 없을 때의 임베딩입니다.

SQL과 Cypher 섞기

AgensGraph에서 SQL과 Cypher 사이에는 다리가 없습니다. 이을 것이 없기 때문입니다. 그래프는 일반 테이블과 같은 데이터베이스에 있는 PostgreSQL 테이블의 묶음이고, 두 언어는 같은 실행 계획 트리로 컴파일됩니다.

그래서 둘은 중첩으로 섞입니다. SQL이 테이블을 기대하는 자리에 Cypher 쿼리가 오고, Cypher가 값을 기대하는 자리에 SQL 쿼리가 옵니다.

이 페이지는 필요할 만한 순서대로 그 교차점들을 훑고, 그다음 모두가 놀라는 규칙 하나를 다룹니다. Cypher의 `RETURN`이 무엇을 돌려주는가입니다.

여기 있는 것은 모두 **예제 데이터셋**에서 실행되며, 그 데이터셋에는 그래프의 일부가 아닌 일반 테이블 `rating`이 포함돼 있습니다.

평범한 SQL로 레이블 읽기

레이블은 테이블이므로, 레이블을 읽는 데 Cypher가 필요하지 않습니다:

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

```
 3.15 | The Terminator | 1984
 3.11 | Total Recall   | 1990
```

이것을 위해 켜야 할 것은 없습니다. 다만 SQL로 레이블에 행을 쓰려면 설정을 켜야 합니다. **일부러 막아 둔 것**을 참고하십시오. 읽기는 그냥 읽기입니다.

질문이 관계형일 때는 SQL을 그대로 쓰는 것이 맞는 도구입니다. 레이블별 개수, 속성 값으로 참조 테이블과 조인하기, 속성에 대한 윈도 함수 같은 것입니다. 홉 수가 정해지지 않은 질문에는 맞지 않는 도구입니다. 이 페이지의 나머지가 그런 경우를 다룹니다.

SQL의 FROM 안에 놓는 Cypher 쿼리

SQL의 `FROM`이 테이블을 기대하는 자리에 Cypher 쿼리를 괄호로 감싸 넣고 별칭을 주십시오. 그 `RETURN` 목록이 컬럼이 됩니다:

```
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
```

별칭은 필수입니다:

```
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.
```

규칙이 두 개 더 있습니다. 모든 컬럼에는 이름이 있어야 합니다. 바깥 쿼리가 그것을 가리킬 수 있어야 하기 때문입니다. 그래서 `AS` 없는 `RETURN m.title`은 거부됩니다. 그리고 중첩된 쿼리는 읽기여야 합니다. `FROM` 안의 쓰기 절은 구문 오류입니다.

하나보다 많이 중첩할 수 있고, 여느 서브쿼리 둘처럼 서로 조인할 수 있습니다:

```
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
```

그래프를 일반 테이블과 조인하기

대부분의 사람이 이 페이지에 오는 이유가 이것입니다. 그래프에는 그래프 질문을 하고, 그 답을 관계형 데이터와 조인합니다:

```
SELECT q.title #>> '{}' AS film, r.score
FROM (MATCH (:person {name: 'Christopher Nolan'})-[:directed]->(m:movie) RETURN m.title AS tit
le) q
JOIN rating r ON r.title = q.title #>> '{}'
```


© 2026 SKAI Worldwide Co., Ltd.

```
WHERE r.source = 'critics'
ORDER BY r.score DESC;
```

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

양쪽에 붙은 `#>> '{}'` 이 요령의 전부이며, 다음 절이 그것을 설명합니다. 이것이 없으면 `jsonb` 문자열을 `text` 컬럼과 비교하는 것이 되어 타입 검사를 통과하지 못합니다.

그래프와 테이블 사이에는 외래 키가 없으므로, 조인 키는 값입니다. 제목, 코드, 외부 id 같은 것입니다. 모델을 직접 정할 수 있다면 그 id를 속성으로 저장하고 인덱스를 걸어 두십시오.

그 조인이 자주 쓰인다면 컬럼으로 승격하십시오. `text` 컬럼과 `text` 컬럼은 형 변환 없이 조인됩니다. **속성을 컬럼으로 승격하기**를 참고하십시오.

CTE 안에 놓는 Cypher 쿼리

`WITH` 절은 `SELECT` 와 마찬가지로 Cypher 쿼리도 손쉽게 받습니다. 그러면 두 번 이상 참조할 수 있는 이름을 얻게 됩니다:

```
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
```

문장이 어느 언어인지 보십시오. `WITH cte AS (...)` 로 시작하는 문장은 SQL이고, 그 본문도 SQL이어야 합니다. 이것은 Cypher의 `WITH` 가 아닙니다. Cypher의 `WITH` 는 파이프라인 구분자이며 서브쿼리에 이름을 붙일 수 없습니다. **무엇이 다른가**를 참고하십시오.

이 방식은 순회를 관계형으로 집계하는 데까지 일반화됩니다. 아래에서는 Cypher가 영화마다 출연진 수를 세고 SQL이 평점을 평균합니다:

```
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.

Cypher 쿼리 안의 SQL

중첩은 반대 방향으로도 됩니다. SQL의 스칼라 서브쿼리는 표현식이므로, Cypher가 표현식을 원하는 곳이면 어디든 올 수 있습니다. `WHERE` 안, `RETURN` 안, 그리고 쓰려는 속성 값 안입니다:

```
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
```

SQL에서와 마찬가지로 서브쿼리는 한 행, 한 컬럼을 반환해야 합니다. 여기서는 형 변환이 필요하지 않았습니다. 서브쿼리의 `text`가 `jsonb` 속성과 비교됐고 그 비교가 해결됐습니다. `#>> '{}'`이 필요한 쪽은 반대 방향입니다. `jsonb` 속성을 `text` 컬럼에 맞추는 경우입니다.

사용자 정의 함수도 표현식이므로, SQL 함수를 Cypher의 `RETURN`에서 호출할 수 있습니다:

```
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
```

`m.year`에 붙인 `::int`가 jsonb 속성을 함수 시그니처가 받는 인자로 바꿔 줍니다. 어떤 언어든 됩니다. PL/pgSQL, PL/Python, C 모두 됩니다. 집계 함수나 윈도 함수도 마찬가지입니다. Cypher의 집계 함수가 곧 PostgreSQL의 것이기 때문입니다.

평범한 값으로 꺼내기

Cypher의 `RETURN`이 속성에서 끌어낸 것은 무엇이든 `jsonb`로 돌아옵니다. 이것은 표시상의 특이점이 아니라 그 컬럼의 타입입니다. 쿼리 안에서 형 변환을 하거나 벗겨 내도 달라지지 않습니다. 투영이 결과를 다시 감싸기 때문입니다.

이 경계에서 가장 흔한 혼란의 원인이 이것이므로, 여기서 모든 각도에서 살펴봅니다.

모든 컬럼이 감싸이는 것은 아니고, 속성에서 끌어낸 컬럼만 그렇습니다. Cypher 서브쿼리의 컬럼들을 바깥에서 타입 검사해 보면 규칙이 정확히 드러납니다:

```
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
```

즉 불리언 표현식은 `boolean`으로, `id(m)`은 `graphid`로, 그래프 요소는 `vertex` · `edge` · `graphpath`로 남습니다. 속성에서 끌어낸 세 컬럼은 `jsonb`로 옵니다. `text`로 형 변환한 것과 이미 벗겨 낸 것까지 그렇습니다.

문자열 속성은 JSON 따옴표가 붙은 채로 돌아옵니다:

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.title AS t;
```

```
      t
--------------
 "The Matrix"
```

Cypher 쿼리 안에서 형 변환을 해도 도움이 되지 않습니다. 바깥 투영이 결과를 다시 감싸기 때문입니다:

```
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
```

Cypher 안에서 `#>> '{}'`으로 텍스트를 꺼내도 같은 이유로 마찬가지입니다:

```
MATCH (m:movie {title: 'The Matrix'}) RETURN m.title #>> '{}' AS t;
```

```
      t
--------------
 "The Matrix"
```

빠져나가는 길은 한 단계 위인 SQL에서 벗겨 내는 것입니다. Cypher 쿼리를 감싸고 그 바깥에서 `#>> '{}'`을 적용하십시오. 이 연산자는 jsonb의 "빈 경로에 있는 값을 텍스트로 꺼내기"입니다:

```
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
```

거기서부터 그 값은 다른 `text`와 똑같이 동작합니다. 컬럼과 비교하고, 함수에 넘기고, 숫자가 필요하면 `(q.year #>> '{}')::int`로 이어서 형 변환하십시오.

이 규칙에 대해 세 가지를 적어 둡니다.

가장 바깥의 투영에만 적용됩니다. Cypher 표현식 **안에서** 읽은 속성은 끝까지 자기 타입을 유지합니다. 그래서 위의 `decade(m.year::int)`가 동작합니다.

승격된 속성은 예외입니다. 타입 있는 컬럼이 뒤에 있는 속성은 jsonb가 아니라 그 컬럼의 타입으로 돌아옵니다. 그래서 승격된 `year`에 대한 `RETURN m.year`는 이미 `integer`이며 벗겨 낼 필요가 없습니다. **승격된 속성 읽기는 컬럼의 타입을 갖는다**를 참고하십시오.

정점이나 간선은 jsonb가 아니라 자기 복합 타입으로 돌아오며, SQL이 필드 표기로 그것을 분해할 수 있습니다:

```
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
```

`m`을 감싼 괄호는 SQL의 것이며, 복합 컬럼의 필드를 읽을 때는 항상 필요합니다.

Cypher 쿼리 위에 만드는 뷰

뷰는 Cypher 절로 시작할 수 없습니다:

```
CREATE VIEW film_rating AS MATCH (m:movie) RETURN m.title AS title;
```

```
ERROR:  syntax error at or near "MATCH"
```

`SELECT`로 감싸면 됩니다. 그래서 그래프 순회를 SQL을 말하는 모든 것 — BI 도구, ORM, 외부 데이터 래퍼 — 을 위해 관계형 뷰로 내보낼 수 있습니다:

```
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
----------------------------+------+-------
 Inception                  | 2010 |   8.8
```


© 2026 SKAI Worldwide Co., Ltd.

```
 Interstellar               | 2014 |   8.7
 The Matrix                 | 1999 |   8.7
 Terminator 2: Judgment Day | 1991 |   8.6
```

이 뷰는 보통 뷰이므로 그래프가 아니라 SQL 검색 경로에 놓입니다. `graph_path`는 레이블 이름이 어느 그래프에 속하는지를 말할 뿐, 뷰가 어디로 가는지에 대해서는 아무것도 말하지 않습니다:

```
SELECT schemaname, viewname FROM pg_views WHERE viewname = 'film_rating';
```

```
 schemaname |  viewname
------------+-------------
 public     | film_rating
```

서버가 저장한 것

뷰는 작성한 텍스트가 아니라 파싱된 쿼리 트리를 담고 있습니다. 정의를 다시 물어보면 작성한 Cypher가 아니라 그 Cypher가 컴파일된 SQL이 나옵니다:

```
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;
```

이것은 한 번 읽어 볼 만합니다. 이 엔진에서 Cypher 쿼리가 **무엇인지**를 가장 분명하게 보여 주는 진술이기 때문입니다. `MATCH (m:movie)`는 `moviekb.movie`를 스캔하고 그 행들을 `vertex` 값으로 조립하는 것이 됐습니다. `m.title`은 `(m).properties.'title'`이 됐습니다. `_agens_default_s`로 시작하는 이름들은 파이프라인 단계에 자동으로 붙은 별칭입니다.

여기서 두 가지가 따라옵니다.

작성한 Cypher는 카탈로그에서 되찾을 수 없습니다. 다른 스키마 객체와 마찬가지로 뷰의 원본을 버전 관리에 두십시오. 이것은 뷰에만 해당하는 이야기가 아닙니다. 저장된 쿼리 트리 위에 세워진 것은 모두 컴파일된 형태를 기록합니다. 룰과 구체화된 뷰가 그렇습니다.

왕복은 됩니다. `pg_dump`가 그 컴파일된 SQL을 그대로 내보내고, 복원하면 동작하는 뷰가 다시 만들어집니다. 새 데이터베이스로 덤프·복원한 뒤의 정의는 글자 하나까지 같고, 뷰는 같은 행을 반환합니다.

그러므로 Cypher 쿼리 위의 뷰는 작성한 대로 읽히지 않더라도 운영에서 신뢰하고 쓸 수 있습니다.

한 가지 주의할 점이 있습니다. Cypher가 어떤 속성을 **승격된 컬럼**으로 해석했다면, 그 뷰는 그 컬럼에 대한 의존성을 갖습니다. 뷰를 없애기 전에는 그 컬럼을 지우는 것이 거부됩니다. 이것은 의도한 동작입니다. 그렇지 않으면 뷰의 뜻이 조용히 바뀌기 때문입니다.

정리

```
DROP VIEW film_rating;
DROP FUNCTION decade(int);
```


© 2026 SKAI Worldwide Co., Ltd.

다음에 갈 곳

- **그래프가 어떻게 저장되는가** — 이 모든 것이 동작하는 이유, 그리고 SQL로 레이블에 쓰는 것을 막는 두 관문입니다.
- **쿼리 조합하기** — Cypher 자체의 파이프라인, 그리고 그 `WITH`가 CTE와 다른 지점입니다.
- **데이터 적재** — 일반 테이블을 그래프로 읽어 들이는 `LOAD FROM` 입니다.
- **속성을 컬럼으로 승격하기** — 늘 쓰는 조인 키를 위한 것입니다.

쿼리 튜닝

Cypher 쿼리는 PostgreSQL의 플래너가 계획합니다. 자기 규칙을 가진 별도의 그래프 엔진이 없으므로, `EXPLAIN`이 동작하고, 통계가 중요하며, 이미 알고 있는 튜닝이 그대로 적용됩니다.

AgensGraph에 특유한 부분은 순회를 위한 몇 개의 실행 계획 노드, 새 그래프마다 반드시 걸리는 통계 문제 하나, 그리고 실행 계획의 모양을 바꾸는 설정 다섯 개입니다.

이 페이지는 실행 계획의 모양만 다룹니다. 자원 쪽 튜닝은 `shared_buffers`, `work_mem`, `effective_cache_size`, 병렬성, autovacuum을 다루며, 그중 그래프에 특유한 것은 없고 구성에 있습니다.

측정할 그래프

예제 데이터셋은 요소가 46개여서, 플래너가 흥미로운 결정을 내리기에는 부족합니다. 차수 분포가 있는 것을 만들고 끝에서 지웁니다:

```
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;
```

계정 20,000개와 `follows` 간선 120,000개입니다. 정점 레이블만 분석했습니다. 그 누락이 뒤에 나오는 절의 요점입니다.

Cypher의 EXPLAIN 읽기

```
EXPLAIN (COSTS OFF) MATCH (a:account {handle: 'user42'})-[r1:follows]->(b:account)-
[r2:follows]->(c:account) RETURN c.handle;
```

```


QUERY PLAN

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


© 2026 SKAI Worldwide Co., Ltd.

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

위에서 AgensGraph에 특유한 것은 네 가지이고, 나머지는 전에 읽어 본 적 있는 실행 계획입니다.

속성 읽기는 `properties.'key'`로 출력됩니다. 그것이 Cypher 파서가 속성 접근을 표기하는 방식이며, 직접 입력할 수 있는 SQL이 아닙니다. `Index Cond`에 있으면 표현식 인덱스와 맞춰지고 있는 것입니다. `Filter`에 있으면 행마다 평가되고 있는 것입니다. 둘 중 어느 쪽을 받는지가 인덱스 페이지의 주제입니다.

`graphid` 범위 검사는 레이블 검사입니다. `start >= '3.0' AND start <= '3.281474976710655'`는 "간선의 이쪽 끝에 있는 정점이 `account` 다"라는 뜻입니다. 레이블 id는 `graphid`의 상위 비트에 있으므로 한 레이블의 id는 연속된 구간을 이루고, 그 검사는 간선 자신의 컬럼에서 답할 수 있습니다. **식별자: graphid**를 참고하십시오.

두 번째 간선에 붙은 **`Filter: (r1.id <> id)` 조건**은 고유성 규칙입니다. 어떤 경로도 한 간선을 두 번 쓸 수 없습니다.

이름을 붙이지 않은 패턴 요소에는 합성 별칭이 주어지고, `"<0000000002>"`처럼 출력됩니다. 그 뒤의 카운터는 문장마다 올라가므로, 그것이 들어 있는 실행 계획은 실행할 때마다 달라집니다. 기준선과 비교하려는 계획이라면 패턴의 모든 요소에 이름을 붙이십시오.

`EXPLAIN (ANALYZE)`도 동작하며, 추정치를 실제와 견주어 볼 수 있는 유일한 방법입니다:

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, SUMMARY OFF, BUFFERS OFF)
MATCH (a:account {handle: 'user42'})-[r1:follows]->(b:account) RETURN count(b);
```

```


QUERY PLAN

------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------
----------------
 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)
```


© 2026 SKAI Worldwide Co., Ltd.

```
                          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
```

참고

가변 길이 계획과 최단 경로 계획은 기본 텍스트 형식으로 읽으십시오. 세 순회 노드(가변 길이, 최단 경로, `Dijkstra`) 중 `EXPLAIN (FORMAT JSON)`으로도 계획이 제대로 읽히는 것은 `Dijkstra`입니다.

순회를 위한 실행 계획 노드

길이가 고정된 패턴에는 특별한 노드가 필요하지 않습니다. 위에서 본 것처럼 스캔과 조인입니다. 아래 노드들은 SQL에 대응 연산자가 없는 것들을 위해 나타납니다.

`Graph VLE`는 가변 길이 간선이며, 간선 레이블과 범위를 이름으로 보여 줍니다:

```
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
```

읽어야 할 줄은 `on related_to`입니다. 레이블을 지정하지 않은 가변 길이 간선은 `Graph VLE on ag_edge`가 되고, 그것은 매 홉마다 그래프의 모든 간선 레이블을 훑습니다. 경로 쿼리에서 피할 수 있는 가장 큰 비용이 그것입니다. 비교는 **경로**에 있습니다.

`Shortestpath VLE [min..max]`는 `shortestpath` 또는 `allshortestpaths`이며, `Hash2Side` 노드 두 개 위에 놓입니다. 그 두 노드는 양쪽 끝에서 각각 자라나는 양방향 탐색의 두 탐색 경계입니다.

`Dijkstra`는 가중 탐색입니다. 그 자식이 확장에 쓰는 스캔이므로, 그것이 훑는 것은 간선 레이블의 `start` 컬럼에 걸린 인덱스입니다.

`Graph Create`, `Graph Set`, `Graph Delete`가 쓰기 노드입니다:

```
SET graph_path = perflab;
EXPLAIN (COSTS OFF) MATCH (a:account {handle: 'user42'}) SET a.seen = 1;
```


© 2026 SKAI Worldwide Co., Ltd.

```
                            QUERY PLAN
---------------------------------------------------------------
 Graph Set
   ->  Index Scan using account_handle_idx on account a
         Index Cond: (properties.'handle' = '"user42"'::jsonb)
```

`eager` **로 표시된 쓰기 노드**는 그 위의 어떤 것이 실행되기 전에 끝나야 합니다. 그 위의 무언가가 그 쓰기가 바꾼 것을 읽기 때문입니다:

```
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")
```

같은 문장 안의 뒤따르는 읽기에 쓰기가 보이게 만드는 것이 이 eager 실행입니다. 대가는 쓰기 출력의 구체화이고, 그 크기는 `eager_mem` 으로 제한됩니다. `MERGE` 는 항상 eager이며, 문장의 마지막 절이 아닌 `SET` · `REMOVE` · `DELETE` 도 그렇습니다.

`_agens_default_s` 로 시작하는 이름은 파이프라인 단계에 자동으로 붙은 별칭이고, 대략 `WITH` 하나에 하나씩입니다. 한 실행 계획 안에서는 안정적이며 튜닝할 대상이 아닙니다.

간선 레이블의 통계

새 그래프에서 가장 먼저 확인할 것이 이것이고, 어떤 인덱스보다 값어치가 큽니다.

순회의 비용은 차수의 곱입니다. 출발한 정점에서 나가는 간선이 몇 개인지, 도달한 정점에서 나가는 간선이 몇 개인지, 그런 식으로 이어집니다. 간선 레이블의 `start` 와 `end` 컬럼에 대한 통계가 그 차수를 설명합니다. AgensGraph는 레이블을 만들 때 두 컬럼의 통계 목표를 최댓값인 1,000으로 설정합니다:

```
SELECT attname, attstattarget FROM pg_attribute
WHERE attrelid = 'perflab.follows'::regclass AND attnum > 0 ORDER BY attnum;
```

```
  attname   | attstattarget
------------+---------------
 id         |
 start      |          1000
 end        |          1000
 properties |
```

목표는 표본이 아닙니다. 무언가가 그 레이블을 분석하기 전까지는 아무것도 없습니다:

```
SELECT count(*) AS stats_rows FROM pg_stats WHERE schemaname = 'perflab' AND tablename =
'follows';
```

```
 stats_rows
------------
          0
```

그리고 추정치는 짐작입니다. 갓 적재한 그래프에서 두 홉을 세어 보면 이렇습니다:

```
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)
```

계정 하나에서 나가는 간선 433개, 두 홉 쌍 186,358개, 그리고 먼 쪽 끝점을 해결하기 위해 계정 20,000개 전체에 대한 해시입니다. 이제 간선 레이블을 분석합니다:

```
ANALYZE perflab.follows;
```

```
EXPLAIN MATCH (a:account {handle: 'user42'})-[r1:follows]->(b:account)-[r2:follows]-
>(c:account) RETURN count(c);
```


© 2026 SKAI Worldwide Co., Ltd.

```
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")
```

간선 6개, 쌍 36개, 먼 쪽 끝점에는 인덱스 전용 스캔, 그리고 추정 비용은 10,088 대신 51입니다. 그리고 36은 정확히 맞습니다:

```
MATCH (a:account {handle: 'user42'})-[r1:follows]->(b:account)-[r2:follows]->(c:account)
RETURN count(c) AS pairs;
```

```
 pairs
-------
    36
```

여기서 따라오는 규칙은 이것입니다. **대량 적재 뒤에는 정점 레이블만이 아니라 간선 레이블에도 `ANALYZE` 를 돌리십시오.** autovacuum도 언젠가는 하지만, 그 "언젠가"는 첫 사용자들이 이미 그 쿼리를 실행한 뒤입니다. 허브 정점이 들어와 차수의 모양이 바뀐 그래프라면, 행 수 임계치를 기다리지 말고 명시적으로 다시 분석할 만합니다.

실행 계획의 모양을 바꾸는 설정

그래프 쿼리가 어떻게 계획되는지에 영향을 주는 설정이 다섯 개 있습니다. 아래는 `pg_settings` 에서 읽은 실제 값입니다:

설정	기본값	범위	하는 일
`enable_eager`	`on`	세션	끄면, eager 계획이 필요한 문장은 계획되지 않고 거부됩니다.
`eager_mem`	4 MB	세션	eager 노드가 디스크로 넘기기 전에 쓸 수 있는 메모리입니다.

© 2026 SKAI Worldwide Co., Ltd.

설정	기본값	범위	하는 일
`enable_graph_endpoint_elision`	`on`	세션	쿼리가 값을 전혀 읽지 않는 패턴 끝점의 스캔을 없앱니다.
`enable_property_promotion`	`on`	세션	승격된 타입 컬럼이 있으면 Cypher의 속성 읽기를 그 컬럼으로 해석합니다.
`auto_gather_graphmeta`	`off`	슈퍼유저	`ag_graphmeta`의 간선 연결 정보를 유지하며, 그것으로 플래너가 도달할 수 없는 레이블을 스캔에서 잘라 냅니다.

참고

`postgresql.conf.sample`은 `enable_eager`와 `enable_multiple_update`의 기본값을 틀리게 적어 놓았습니다. 둘 다 켜져 있습니다. 샘플 파일이 아니라 `pg_settings`를 읽으십시오.

이 중 어느 것도 운영에서 돌릴 손잡이가 아닙니다. 네 개는 이미 켜져 있고, 다섯째는 카탈로그를 유지할지에 대한 결정입니다. 진단을 위해 알아 두십시오. 하나를 끄면 그것이 예상하지 못한 실행 계획의 원인인지 알 수 있습니다.

끝점 생략

쿼리가 값을 전혀 읽지 않는 끝점은 아예 스캔되지 않습니다. 여기서 `b`는 바인딩되고 레이블도 붙었지만 투영되지 않습니다:

```
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 (start >= '3.0'::graphid) AND (start <= '3.281474976710655'::graphid))
   ->  Hash
         ->  Seq Scan on account a
```

조인 하나이고, `(b:account)`는 간선의 `"end"` 컬럼에 대한 범위 검사로만 남습니다. 이 생략을 끄면 두 번째 스캔과 조인이 돌아옵니다:

```
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))
```


© 2026 SKAI Worldwide Co., Ltd.

```
         ->  Hash
               ->  Seq Scan on account a
   ->  Hash
         ->  Seq Scan on account b
```

교훈은 그 설정에 관한 것이 아닙니다. 필요하지 않은 끝점을 투영하면 매번 스캔 하나와 조인 하나의 비용이 듭니다. 그러므로 `RETURN`은 호출하는 쪽이 읽는 것으로 한정하십시오.

ag_graphmeta로 스캔 잘라 내기

`ag_graphmeta`는 그래프에 실제로 나타나는 (시작 레이블, 간선 레이블, 끝 레이블) 조합마다 한 행을 담고, 그것을 이루는 간선이 몇 개인지도 함께 담습니다. 그것이 있으면 레이블 없는 패턴 끝점을 그 간선이 도달한다고 알려진 레이블로 좁힐 수 있습니다. 이 정보는 `auto_gather_graphmeta`가 켜져 있을 때만 유지되므로, 기본적으로 레이블 없는 끝점은 `ag_vertex` 계열 전체를 스캔합니다:

```
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
```

수집을 켜면 `Append` 둘이 접혀 사라집니다. 수집은 데이터베이스의 모든 그래프에 대해 기준선을 만들므로, 의도를 갖고 하십시오:

```
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
   ->  Hash
         ->  Seq Scan on account m
```

`follows`는 언제나 `account` → `account`로만 이어지므로, 레이블 없는 두 끝점은 계정일 수밖에 없습니다. 레이블이 많은 그래프에서는 이것이 끝점마다 스캔 하나와 레이블마다 스캔 하나의 차이입니다. 대가는 간선 쓰기마다 드는 유지 비용, 그리고 기준선이 무효화될 때마다 드는 재스캔입니다.

이 예제에는 레이블이 둘뿐이므로 여기서의 절약은 작습니다. 요점은 그 모양입니다. `pg_catalog.ag_graphmeta`가 그 개수를 담고 `ag_graph.graphmeta_valid`가 그 사용을 통제합니다. **그래프 메타데이터**를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

가장 단순한 대안은 패턴에 레이블을 이름으로 쓰는 것입니다. `MATCH (n:account)-[r:follows]->(m:account)`에는 카탈로그가 전혀 필요하지 않습니다.

eager 계획 금지하기

`enable_eager = off`는 다른 실행 계획을 만들지 않습니다. 계획을 만들기를 거부합니다:

```
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.
```

작업 부하 안에서 어느 문장이 eager 실행에 의존하는지 알아내는 데 쓰십시오. 이것을 끈 채로 운영하지는 마십시오. 그런 문장에서 eager 실행은 비용의 선택이 아니라 정확성의 요구 조건이므로, 되돌아갈 더 값싼 계획이 없고 서버는 대신 오류를 냅니다.

승격된 컬럼으로의 해석

`enable_property_promotion = off`는 타입 컬럼이 있는 곳에서도 속성 읽기를 `jsonb` 컬럼인 `properties`로 되돌려 보냅니다. 실행 계획은 바뀌고 답은 바뀌지 않습니다. 값이 양쪽에 다 있기 때문입니다. 컬럼을 지우지 않고도 예상하지 못한 실행 계획의 배후가 승격인지 확인하는 방법입니다. 전과 후는 해석 기능 끄기에 있습니다.

무엇을 어떤 순서로 할까

1. **쿼리가 닿는 모든 레이블에 `ANALYZE`를 돌리십시오. 간선 레이블도 포함합니다.** 이것을 하기 전까지는 이 목록의 다른 어떤 것도 의미가 없습니다.
2. **출발점을 확인하십시오.** 순회에는 선택도가 높고 인덱스가 걸린 출발점 하나가 필요합니다. 그 출발점을 골라 내는 속성에 대해 `EXPLAIN`이 `Filter`가 아니라 `Index Cond`를 보여 줘야 합니다 — 어떤 조건이 그럴 수 있는지는 **인덱스와 제약**이 말해 줍니다.
3. **모든 가변 길이 패턴에 간선 레이블을 이름으로 쓰십시오.** 실행 계획에 나온 `Graph VLE on ag_edge`는 플래너가 아니라 쿼리를 가리킵니다.
4. 가변 길이 패턴의 **길이에 상한을 두십시오.**
5. **호출하는 쪽이 읽는 것만 투영하십시오.** 그러면 쓰지 않는 끝점이 생략됩니다.
6. 늘 필터·조인·정렬에 쓰는 **속성의 승격을 검토하십시오** — 속성을 컬럼으로 승격하기.
7. **그러고 나서 자원을 보십시오.** 구성.

정리

```
DROP GRAPH perflab CASCADE;
```

```
DROP TABLE seed_account;
DROP TABLE seed_follows;
```

다음에 갈 곳

- 인덱스와 제약 — 어떤 조건에 인덱스가 쓰일 수 있는지입니다.
- 경로 — 이 페이지의 순회 특화판입니다.
- 속성을 컬럼으로 승격하기 — 읽기 자체가 비용일 때입니다.
- 구성 — 메모리, I/O, autovacuum입니다.

© 2026 SKAI Worldwide Co., Ltd.

레퍼런스

언어의 정식 정의입니다. 어휘 규칙, 타입, 모든 절, 모든 데이터 정의 문장, 모든 함수, 그리고 AgensGraph가 더한 모든 서버 설정입니다. 같은 내용을 이야기 형태로 소개하는 것은 **쿼리하기**를 참고하십시오.

구문

어느 언어로 쓰였든 모든 문장이 지키는 어휘 층입니다. 이 레퍼런스가 구문에 쓰는 표기법, 그리고 식별자·리터럴·연산자·주석·null의 규칙입니다.

Cypher와 GQL과 SQL이 파서 하나를 공유하므로, 이 규칙들은 셋 모두에서 같습니다. 규칙이 셋 사이에서 다른 곳에서는 그 차이를 따로 적었습니다.

구문 표기 규약

이 레퍼런스의 모든 개요는 같은 방식으로 적혀 있습니다:

```
MATCH pattern [, pattern ] ... [ WHERE condition ]
```

개요에서 그 밖의 것은 표기법이 아닙니다. 쉼표·괄호·화살표는 문장의 일부이며, 보인 대로 입력합니다.

굵게	키워드입니다. 보이는 대로 입력하십시오. 대소문자는 상관없습니다.
기울임	직접 채우는 자리입니다. 각각 같은 페이지의 **매개변수** 아래에 정의돼 있습니다.
`[ ]`	감싼 부분은 생략할 수 있습니다.
`{ ... | ... }`	세로 막대로 구분한 선택지를 중괄호로 감싼 것입니다. 정확히 하나를 고르십시오.
`...`	그 앞의 항목을 반복할 수 있습니다.
`[, ... ]`	그 앞의 항목을 쉼표로 구분해 반복할 수 있습니다.
개요가 두 줄 이상	같은 문장의 다른 형태들입니다. 그중 하나를 쓰십시오.
`where` placeholder `is one of:`	한 줄에 담기에 너무 큰 자리표를 뒤따르는 줄에서 펼쳐 보입니다.

대괄호와 중괄호는 결코 글자 그대로가 아닙니다. 어떤 문장에는 실제로 대괄호가 들어갑니다. 리스트 리터럴, 맵, 간선 패턴이 그렇습니다. 그런 경우에는 개요가 산문으로 그 사실을 적거나, `-[` variable `]->` 처럼 대괄호를 다른 자리에 보여 줍니다.

© 2026 SKAI Worldwide Co., Ltd.

식별자

식별자는 그래프·레이블·변수·속성 키·컬럼·인덱스·제약의 이름입니다.

대소문자 변환

따옴표 없는 식별자는 **소문자**로 바뀝니다. PostgreSQL이 따옴표 없는 SQL 식별자를 소문자로 바꾸는 것과 정확히 같습니다. 그래서 `MOVIE`, `Movie`, `movie`가 모두 같은 레이블을 가리킵니다:

```
MATCH (n:MOVIE {title: 'The Matrix'}) RETURN n.title AS title;
```

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

큰따옴표로 감싼 식별자는 적힌 대로 받아들여집니다. `"MOVIE"`는 `movie`와 다른 이름입니다. `MOVIE`라는 레이블은 없으므로, 그 패턴은 실패하는 것이 아니라 아무것에도 일치하지 않습니다:

```
MATCH (n:"MOVIE" {title: 'The Matrix'}) RETURN n.title AS title;
```

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

따옴표는 변수에도 통하며, 따옴표를 붙인 변수는 언급할 때마다 따옴표를 붙여야 합니다:

```
MATCH ("Film":movie {title: 'The Matrix'}) RETURN "Film".title AS title;
```

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

속성 키도 같은 방식으로 소문자로 바뀝니다. `{MixedKey: 1}`은 키 `mixedkey`를 저장하고, `{"MixedKey": 1}`은 `MixedKey`를 저장합니다. 둘은 서로 다른 키이고 한 정점이 둘 다 가질 수 있으므로, 키에 따옴표를 붙일 때는 일관되게 붙이거나 아예 붙이지 마십시오.

백틱은 인용 문자가 아닙니다. `` `movie` ``는 인용된 식별자가 아니라 구문 오류입니다:

```
MATCH (n:`movie`) RETURN n.title;
```

```
ERROR:  syntax error at or near "`"
LINE 1: MATCH (n:`movie`) RETURN n.title;
                 ^
```

문자

따옴표 없는 식별자는 글자나 밑줄로 시작하고, 그 뒤로 글자·숫자·밑줄·달러 기호가 이어집니다. "글자"에는 ASCII가 아닌 글자도 포함되므로 `café`도 유효한 식별자입니다. 숫자로 시작할 수는 없습니다. `1abc`는 뒤에 군더더기가 붙은 숫자로 읽힙니다.
큰따옴표로 감싼 식별자에는 null 바이트를 뺀 어떤 문자든 들어갈 수 있고, 그 안의 큰따옴표는 두 번 적습니다(`"say ""hi"""`).

길이 제한은 PostgreSQL의 것입니다. 63바이트이고, 그보다 긴 이름은 그 길이로 잘립니다. 이 제한은 글자가 아니라 바이트를 세므로, 여러 바이트 글자로 된 이름은 더 일찍 제한에 닿습니다.

© 2026 SKAI Worldwide Co., Ltd.

예약어

어떤 키워드는 **예약돼** 있습니다. 문법이 그 단어를 이름이 아니라 키워드로 읽습니다. `both`, `let`, `finish`, `insert`, `offset`, `xor`가 모두 예약어입니다. 일부는 PostgreSQL에서 물려받은 것이고 일부는 그래프 절을 위해 더한 것입니다. 각각이 어느 자리에서 여전히 동작하는지는 단어마다 다르므로, 자리별 표는 **키워드**에 있습니다. 패턴 변수 자리에서는 그중 어느 것도 쓸 수 없습니다:

```
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;
               ^
```

따옴표를 붙이면 그 제한이 풀립니다. 인용된 식별자는 결코 키워드로 읽히지 않기 때문입니다:

```
MATCH ("both":movie {title: 'The Matrix'}) RETURN "both".title AS title;
```

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

예약어는 `RETURN` 별칭과 속성 키로는 여전히 허용됩니다. 그 두 자리는 키워드와 혼동될 수 없기 때문입니다:

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

```
     both
--------------
 "The Matrix"
```

리터럴

문자열

문자열 리터럴은 **작은**따옴표로 감쌉니다. 큰따옴표는 문자열이 아니라 식별자를 만들므로, `"plain"`은 변수 참조이고 그런 변수가 유효 범위에 없으면 실패합니다.

리터럴 안의 작은따옴표는 두 번 적습니다. 역슬래시 이스케이프는 JSON의 규칙을 따릅니다. Cypher의 문자열이 `jsonb` 문자열이 되기 때문입니다:

이스케이프	뜻
`''`	작은따옴표입니다.
`\"`	큰따옴표입니다.
`\\`	역슬래시입니다.
`\/`	슬래시입니다.

© 2026 SKAI Worldwide Co., Ltd.

이스케이프	뜻
`\b` `\f` `\n` `\r` `\t`	백스페이스, 폼 피드, 줄바꿈, 캐리지 리턴, 탭입니다.
`\u` 뒤에 16진수 네 자리	그 코드 포인트의 문자입니다.

그 밖의 역슬래시 조합은 거부됩니다. `'a\qb'`는 `Escape sequence "\q" is invalid`를 냅니다.

```
RETURN 'plain', 'it''s', 'a\tb', 'a\\tb', 'say \" hi';
```

```
 ?column? | ?column? | ?column? | ?column? |  ?column?
----------+----------+----------+----------+-------------
 "plain"  | "it's"   | "a\tb"   | "a\\tb"  | "say \" hi"
```

출력은 각 값의 `jsonb` 표현이고, 그것은 출력하는 것을 다시 이스케이프합니다. 세 번째 컬럼은 문자 세 개, `a`와 탭과 `b`를 담고 있습니다.

숫자

숫자 리터럴은 SQL에서와 같이 적으며, PostgreSQL의 비십진 표기와 구분자 표기도 받아들여집니다. 16진 `0x`, 8진 `0o`, 2진 `0b`, 그리고 자릿수 구분자로 쓰는 밑줄입니다.

```
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
```

소수점 없이 적은 숫자는 JSON 정수이고, 그 둘을 나누면 버림이 일어납니다. `7 / 2`는 `3`입니다. `3.5`를 원하면 `7.0 / 2`로 쓰십시오.

불리언, null, 리스트, 맵

`true`, `false`, `null`은 대소문자와 무관한 키워드입니다. 리스트는 대괄호로, 맵은 중괄호로 적습니다. 둘 다 중첩되고, 둘 다 값의 타입을 섞을 수 있습니다. 둘 다 `jsonb`가 되기 때문입니다.

```
RETURN true, false, null, [1, 'two', true, null], {a: 1, b: [2], c: {d: 3}};
```

```
 ?column? | ?column? | ?column? |        ?column?        |             ?column?
----------+----------+----------+------------------------+----------------------------------
 t        | f        |          | [1, "two", true, null] | {"a": 1, "b": [2], "c": {"d": 3}}
```

맵의 키는 식별자이고 식별자처럼 소문자로 바뀝니다. `[]`와 `{}`는 빈 리스트와 빈 맵입니다. 리터럴이 저장되는 `jsonb` 값과 어떤 관계인지는 **데이터 타입**을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

연산자

우선순위

연산자는 이 순서로 결합하며, 느슨한 것이 먼저입니다. 같은 행의 두 연산자는 우선순위가 같고 그 행이 적은 대로 결합합니다.

우선순위	연산자	결합 방향
1 (가장 느슨)	`UNION` `EXCEPT`	왼쪽
2	`INTERSECT`	왼쪽
3	`OR` `XOR`	왼쪽
4	`AND`	왼쪽
5	`NOT`	오른쪽
6	`IS NULL` `IS NOT NULL` `IS [NOT] UNKNOWN`	없음
7	`<` `<=` `>` `>=` `=` `<>` `!=`	없음
8	`IN` `BETWEEN`	없음
9	`||` 와 `jsonb` 연산자를 포함한 그 밖의 모든 연산자	왼쪽
10	`+` `-` (이항)	왼쪽
11	`*` `/` `%`	왼쪽
12	`^`	왼쪽
13	`-` (단항)	오른쪽
14	`STARTS WITH` `ENDS WITH` `CONTAINS` `=~`	없음
15	`[` `]` (첨자), `(` `)` (호출)	왼쪽
16	`::`	왼쪽
17 (가장 단단)	`.`	왼쪽

이 표는 Cypher 표현식이 담을 수 있는 연산자를 나열합니다. SQL은 자기만의 단계에 몇 개를 더합니다. `AT TIME ZONE`, `COLLATE`, `LIKE`, `ILIKE`, `SIMILAR TO`입니다. Cypher의 표현식 문법은 그중 어느 것도 받지 않습니다.

이 표에서 흔한 기대와 다른 것이 세 가지 있습니다.

`^`는 **왼쪽** 결합이고 단항 마이너스보다 느슨하게 결합합니다. 산술의 규칙이 아니라 PostgreSQL의 규칙입니다. 그래서 `2 ^ 3 ^ 2`는 `(2 ^ 3) ^ 2`이고 `-2 ^ 2`는 `(-2) ^ 2`입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
RETURN 2 ^ 3 ^ 2 AS pow, -2 ^ 2 AS neg;
```

```
 pow | neg
-----+-----
 64  | 4
```

문자열 일치 연산자는 첨자·호출·형 변환·속성 읽기를 뺀 모든 것보다 **단단히** 결합합니다. 이어 붙이기보다도, 산술보다도 단단히 결합합니다. 그래서 아래 `||` 의 오른쪽 피연산자는 `'c'` 만이 아니라 `STARTS WITH` 검사 전체입니다:

```
RETURN 'ab' || 'c' STARTS WITH 'a';
```

```
 ?column?
-----------
 "abfalse"
```

다른 쪽으로 읽히길 원한다면 괄호를 쓰십시오. `('ab' || 'c') STARTS WITH 'a'` 는 `true` 입니다.

비교는 이어지지 않습니다. `1 < 2 < 3` 은 두 개의 비교가 아니라 구문 오류입니다.

논리 연산자

`AND` , `OR` , `NOT` , `XOR` 는 불리언을 받아 불리언을 반환하며, 세 개의 값을 다룹니다. 진리표는 **null과 삼치 논리**를 참고하십시오. `XOR` 는 SQL의 집합에 AgensGraph가 더한 것이고, 우선순위는 `OR` 와 같습니다.

비교와 조건

`=` , `<>` (또는 `!=`), `<` , `<=` , `>` , `>=` 는 두 값을 비교합니다. `jsonb` 타입이 서로 다른 값을 비교해도 실패하지 않습니다. `jsonb` 자신의 타입 순서로 그것들을 정렬합니다.

`IN` 은 리스트에 속하는지 검사합니다. `IS NULL` 과 `IS NOT NULL` 은 어떤 값이든 비교하지 않고 null인지 검사합니다. `IS UNKNOWN` 과 `IS NOT UNKNOWN` 은 불리언에 대한 그 검사를 표준이 표기하는 방식이며, 여기서도 받아들여집니다:

```
RETURN 2 IN [1, 2, 3] AS present, null IS NULL AS nul, null IS UNKNOWN AS unk;
```

```
 present | nul | unk
---------+-----+-----
 t       | t   | t
```

문자열 일치

문자열을 일치시키는 연산자가 네 개 있습니다. `LIKE` , `ILIKE` , `SIMILAR TO` 는 **SQL 전용**입니다. Cypher의 표현식 문법의 일부가 아니고, 거기서는 구문 오류입니다. PostgreSQL의 정규 표현식 연산자(`~` , `~*` , `!~` , `!~*`)는 Cypher에서 동작합니다.

연산자	검사
string `STARTS WITH` prefix	문자열이 그 접두 문자열로 시작합니다.
string `ENDS WITH` suffix	문자열이 그 접미 문자열로 끝납니다.
string `CONTAINS` substring	그 부분 문자열이 문자열 어디에든 나타납니다.

© 2026 SKAI Worldwide Co., Ltd.

연산자	검사
string `=~` pattern	문자열 전체가 POSIX 정규 표현식과 일치합니다.

```
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
```

넷 중 어느 것도 인덱스로 답할 수 없습니다. 인덱스로 처리돼야 하는 조건에는 `=` 이나 범위가 필요합니다. 인덱스가 속성을 어떤 형태로 보는지는 **데이터 타입**에서 다룹니다.

산술

`+`, `-`, `*`, `/`, `%`, `^`는 숫자에 대해 동작하고, 단항 `+`와 `-`는 부호를 바꿉니다. 속성에 대해서는 SQL의 수치 타입이 아니라 `jsonb`를 상대로 동작하며, 그래서 `+`에 일이 둘 더 생깁니다. 문자열 둘을 이어 붙이고, 리스트에 덧붙입니다.

```
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]
```

맵 둘에 대한 `+`는 `invalid expression: {"a": 1} + {"b": 2}`로 거부됩니다. 맵은 대신 `jsonb`의 이어 붙이기 연산자로 병합하십시오:

```
RETURN {a: 1} || {b: 2} AS merged;
```

```
     merged
-----------------
 {"a": 1, "b": 2}
```

값이 아니라 요소에 병합하려면 `SET n +=` map을 쓰십시오.

jsonb 연산자

속성은 `jsonb`이므로 PostgreSQL의 `jsonb` 연산자가 속성에 곧바로 적용됩니다. 알아 둘 만한 것들은 이렇습니다:

연산자	결과
json `->` key / index	그 멤버를 `jsonb`로
json `->>` key / index	그 멤버를 `text`로

© 2026 SKAI Worldwide Co., Ltd.

연산자	결과
json `#>` path	텍스트 배열 경로에 있는 값을 `jsonb` 로
json `#>>` path	텍스트 배열 경로에 있는 값을 `text` 로
json `?` key	그 키가 있는지
json `@>` json	왼쪽 값이 오른쪽을 포함하는지
json `<@` json	왼쪽 값이 오른쪽에 포함되는지
json `||` json	이어 붙이기. 객체 둘은 병합되고, 그 밖의 조합은 배열을 만듭니다.

```
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
```

`->>` 은 `text` 를 반환했지만, Cypher의 `RETURN` 이 스칼라 컬럼을 다시 `jsonb` 로 감싸므로 따옴표가 그대로 남습니다. 평범한 문자열로 꺼내려면 그 쿼리를 SQL로 감싸십시오. **데이터 타입**을 참고하십시오.

Cypher 안에서 PostgreSQL의 나머지 `jsonb` 연산자를 지배하는 규칙이 두 가지 있고, 둘 다 산술 표기를 Cypher가 차지했다는 점에서 따라옵니다.

`jsonb` 값 둘 사이의 `-` 는 뺄셈이므로, `jsonb` 의 키 삭제 연산자는 Cypher에서 닿을 수 없습니다. `properties(m) - 'plot'` 은 `invalid expression` 을 냅니다. 키는 SQL로 감싸서 지우거나 `REMOVE` 로 지우십시오.

네이티브 `jsonb` 연산자 옆에 문자열 리터럴만 두면 그것은 JSON 문자열이 아니라 JSON 문서 전체로 파싱됩니다. 그래서 `m.title || '!'` 은 `Token "!" is invalid` 로 실패합니다. `m.title || to_jsonb('!'::text)` 처럼 그 연산자에 타입이 있는 피연산자를 주십시오. 또는 Cypher 쪽 해석을 취하는 `+` 를 쓰십시오.

주석

주석은 `--` 부터 그 줄의 끝까지, 또는 `/*` 부터 짝이 되는 `*/` 까지입니다. 블록 주석은 중첩되므로, 주석이 들어 있는 영역을 주석 처리해도 짝이 맞습니다. `//` 는 주석이 아니고 구문 오류입니다.

```
MATCH (m:movie {title: 'The Matrix'})   -- one film
/* a block comment
   over two lines */
RETURN m.title AS title;
```

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


© 2026 SKAI Worldwide Co., Ltd.

null과 삼치 논리

`null`은 **알 수 없음**을 뜻합니다. 비교에 쓸 수 있는 값이 아니고, 속성 값도 아닙니다. 속성에 null을 할당하면 그 속성이 지워집니다. `allow_null_properties`를 켜 둔 경우는 예외입니다.

진리표

`AND`, `OR`, `XOR`는 세 개의 값으로 다음과 같이 확장됩니다. 아래 표는 모든 조합을 실행한 출력입니다:

```
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`은 참을 거짓으로, 거짓을 참으로, null을 null로 바꿉니다:

```
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
```

그래서 `AND`는 한쪽이 거짓이면 곧바로 거짓이고, `OR`는 한쪽이 참이면 곧바로 참이며, 다른 쪽이 무엇이든 그렇습니다. null이 얽힌 그 밖의 모든 것은 null입니다.

`IN`도 같은 규칙에서 따라옵니다. null을 담은 리스트에서 찾지 못한 값이라도 실은 거기 있었을 수 있으므로, 답은 거짓이 아니라 알 수 없음입니다.

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

```
          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
```

빈 리스트가 예외입니다. 그 안에는 아무것도 없으므로 null에 대해서도 답이 거짓입니다.

null을 내놓는 표현식

오류나 값이 아니라 null을 내놓는 표현식들입니다:

표현식	이유
`n.no_such_property`	그 요소에 그런 속성이 없습니다.
`[][0]`, `head([])`	그 리스트에 그런 요소가 없습니다.
`1 + null`	알 수 없는 피연산자와의 산술입니다.
`1 < null`, `null = null`	알 수 없는 값과의 비교입니다.
`sin(null)`, `toUpper(null)`	null을 주고 호출한 함수입니다.

```
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
------------------+-----------------+---------------+------------+------------+----------
+----------
 null             | null            | null          | null       | null       | null     | nul
l
```

`IS NULL`이 있는 것은 `null = null`이 null이기 때문입니다. `IS NULL`은 null을 검사하는 유일한 방법입니다.

조건은 불리언만 받는다

`WHERE`, `FILTER`, `CASE WHEN`과 그 밖의 모든 조건은 불리언만 받고 그 밖의 것은 받지 않습니다. `jsonb` 문자열·숫자·리스트·맵으로 평가되는 조건은 오류입니다:

```
CREATE GRAPH demo;
SET graph_path = demo;
CREATE VLABEL t;
```


© 2026 SKAI Worldwide Co., Ltd.

```
CREATE (:t {name: 'Jack'});
MATCH (n:t) WHERE n.name RETURN n.name;
```

```
ERROR:  cannot cast "Jack" (jsonb type string) to boolean
```

그 오류는 조건을 평가하는 데서 나오므로, 오류가 이름으로 부르는 값은 실행기가 먼저 도달한 행의 값입니다. 애초에 `jsonb`가 아닌 조건은 그보다 앞서, 문장이 파싱될 때 거부됩니다:

```
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;
                          ^
```

뜻하는 검사를 그대로 쓰십시오. `WHERE n.name IS NOT NULL`은 그 속성이 있는지 묻습니다:

```
MATCH (n:t) WHERE n.name IS NOT NULL RETURN n.name AS name;
```

```
  name
--------
 "Jack"
```

`WHERE n.name <> ''`은 그것이 비어 있지 않은지 묻고, `WHERE n.active`는 그 속성이 실제로 불리언을 담고 있을 때 동작합니다. `jsonb` 불리언이 불리언으로 형 변환되기 때문입니다.

null은 여전히 받아들여지고 SQL에서와 같이 동작합니다. 조건이 null인 행은 반환되지 않고 아무 오류도 나지 않습니다. 그래서 `WHERE n.no_such_property`는 아무 행도 반환하지 않습니다.

> 참고
>
> 이것은 비어 있지 않은 값을 조건으로 받아들였던 이전 릴리스와의 의도적인 호환성 단절입니다. 그런 방식으로 쓴 쿼리는 뜻하지 않은 것을 기준으로 걸러 내는 대신 실패합니다. 무엇을 고쳐야 하는지는 **Cypher 호환성**이 나열합니다.

호환성

openCypher

다른 곳에서 Cypher를 알고 온 독자가 걸릴 어휘상의 차이가 세 가지 있습니다.

- **식별자는 소문자로 바뀌고, 백틱은 인용 문자가 아닙니다.** 큰따옴표를 쓰십시오. `` `My Label` ``은 구문 오류입니다.
- **문자열 리터럴은 작은따옴표만 받습니다.** 큰따옴표는 식별자를 만듭니다.
- **`//`는 주석이 아닙니다.** `--`나 `/* */`를 쓰십시오.

`XOR`, `STARTS WITH`, `ENDS WITH`, `CONTAINS`, `=~`는 모두 있고, openCypher에서와 같은 뜻입니다.

ISO/IEC 39075

이 레퍼런스의 표기법은 표준의 구문 생성 규칙과 같은 규약을 따르지만, 표준의 BNF는 아닙니다.

`IS [NOT] UNKNOWN`은 불리언에 대한 null 검사를 표준이 표기하는 방식이고 `IS [NOT] NULL`과 함께 받아들여집니다. `XOR`도 그렇습니다. 둘 다 표준의 `<boolean value expression>`(ISO/IEC 39075:2024 §20.20)에 속합니다. `UNKNOWN`을 **값**으로 쓰면 구문 오류입니다. 나머지는 **GQL 적합성**을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

관련 항목

- 데이터 타입 — `graphid`, `vertex`, `edge`, `graphpath`, 그리고 `jsonb` 속성
- 서버 설정 — 모든 세션에 필요한 `graph_path`를 포함
- 키워드 — 전체 키워드 목록과 단어별 범주
- Cypher, GQL, SQL — 어느 언어를 꺼내 쓸까
- PostgreSQL: SQL 구문 — 같은 렉서의 SQL 쪽 절반

데이터 타입

AgensGraph는 PostgreSQL의 타입 집합에 다섯 개를 더합니다. `graphid`, `vertex`, `edge`, `graphpath`, `rowid`입니다. 그래프가 저장하는 그 밖의 모든 것은 레이블의 `properties` 컬럼에 담긴 `jsonb` 입니다. PostgreSQL의 모든 타입은 여전히 쓸 수 있고 PostgreSQL이 문서화한 대로 동작합니다.

이 다섯은 고정된 OID로 `pg_type`에 정의돼 있으므로 모든 데이터베이스에서 같고 확장이 필요하지 않습니다.

타입	저장 방식	담는 것
`graphid`	8바이트, 값으로	정점 하나 또는 간선 하나의 식별자입니다.
`vertex`	복합, 가변	id, 속성 맵, 그리고 그 행의 물리적 위치입니다.
`edge`	복합, 가변	id, 끝점 id 둘, 속성 맵, 그리고 그 행의 위치입니다.
`graphpath`	복합, 가변	정점의 배열과 간선의 배열입니다.
`rowid`	10바이트, 참조로	테이블 OID와 행 포인터입니다. 내부용입니다.

graphid

`graphid`는 정점이나 간선의 식별자입니다. 간선은 자기 끝점을 지목하려고 그것을 저장합니다. 두 부분으로 된 64비트 값입니다. 상위 16비트가 **레이블 id**, 하위 48비트가 레이블마다 매겨지는 시퀀스인 **로컬 id**입니다. 출력 형태는 labid`.`locid입니다.

그 배치가 왜 플래너에게 중요한지는 **그래프가 어떻게 저장되는가**가 설명합니다. 이 절은 그 정식 세부 내용입니다.

두 함수가 그 부분들을 다시 분해합니다:

```
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 `)`가 그 부분들로 하나를 만듭니다. 레이블의 `id` 컬럼이 기본값 표현식에서 호출하는 것과 같은 함수입니다.

© 2026 SKAI Worldwide Co., Ltd.

범위

레이블 id는 16비트, 로컬 id는 48비트이므로 가장 큰 `graphid`는 이렇습니다:

```
SELECT graphid(65535, 281474976710655) AS highest;
```

```
        highest
----------------------
 65535.281474976710655
```

어느 부분이든 범위를 벗어나면 잘리는 것이 아니라 거부됩니다:

```
SELECT graphid(65536, 1);
```

```
ERROR:  labid out of range: 65536
```

```
SELECT graphid(3, 281474976710656);
```

```
ERROR:  locid out of range: 281474976710656
```

0은 **레이블 없음**으로 예약돼 있으므로, 데이터베이스는 최대 65,535개의 레이블을, 레이블은 최대 2^{48} – 1개의 요소를 가집니다. 이 예약은 검사가 아니라 관례이므로 `graphid(0, 1)`은 `0.1`을 만듭니다. 다만 레이블 id 0을 배정하는 것은 아무것도 없습니다. 이런 수치는 PostgreSQL 자신의 것과 함께 **제한**이 모아 두었습니다.

형 변환

`graphid`로 들어가는 형 변환은 `numeric`에서 오는 것 하나뿐이고, 그것은 암시적입니다. `integer`에서 오는 것은 없으므로, 소수점 없이 적은 정수는 변환되지 않습니다:

```
SELECT 1::graphid;
```

```
ERROR:  cannot cast type integer to graphid
LINE 1: SELECT 1::graphid;
                ^
```

두 부분을 적으십시오. 소수점이 있는 수치 리터럴은 형 변환되고, labid `.` locid 형태의 문자열은 파싱됩니다:

```
SELECT 1.1::graphid AS from_numeric, '3.1'::graphid AS from_text;
```

```
 from_numeric | from_text
--------------+-----------
 1.1          | 3.1
```

`numeric` 형 변환은 그 타입이 출력하는 방식으로 숫자를 읽습니다. 그래서 `1.1`은 1.1이라는 수가 아니라 레이블 1의 요소 1입니다. 소수 자리가 없는 numeric에는 로컬 id가 없어서 거부됩니다. `1::numeric::graphid`는 `invalid input syntax for type graphid: "1"`을 냅니다.

`graphid`에서 나가는 형 변환은 없습니다. 부분이 필요하면 `graphid_labid()`와 `graphid_locid()`를 쓰거나, 그냥 출력되게 두십시오.

© 2026 SKAI Worldwide Co., Ltd.

vertex와 edge

`vertex`는 필드 세 개로 된 복합 값입니다. `id graphid`, `properties jsonb`, 그리고 그 행의 물리적 위치를 담은 `tid`입니다. 쓰기는 그 `tid`로 그 행을 다시 찾습니다. `edge`는 같은 세 필드에 `start graphid`와 `end graphid`가 더해진 것입니다.

레이블과 id, 속성을 보여 주는 형태로 출력됩니다:

```
MATCH (p:person {name: 'Keanu Reeves'}) RETURN p;
```

```
                          p
-------------------------------------------------
 person[4.1]{"born": 1964, "name": "Keanu Reeves"}
```

간선은 자기 id와 속성 사이에 양 끝점을 출력하며, 시작점이 먼저 옵니다:

```
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"}
```

그래서 모양은 정점이 `label[id]{properties}`, 간선이 `label[id][start,end]{properties}`입니다. 속성이 없는 요소는 빈 맵을 출력합니다. `directed[6.8][4.8,3.5]{}`입니다.

`tid` 필드는 두 출력 형태 어디에도 들어 있지 않고, 의존할 만한 것도 아닙니다. 그 행이 갱신되면 값이 바뀝니다.

그것으로 할 수 있는 일

`vertex`와 `edge`에는 각각 비교 연산자 여섯 개가 있고, 그 밖에는 아무것도 없습니다:

타입	연산자
`graphid`	`=` `<>` `<` `<=` `>` `>=`
`vertex`	`=` `<>` `<` `<=` `>` `>=`
`edge`	`=` `<>` `<` `<=` `>` `>=`
`rowid`	`=` `<>` `<` `<=` `>` `>=`
`graphpath`	없음

비교는 식별자만으로 합니다. `vertex_eq`와 그 형제 함수들은 `id` 필드를 비교하고 속성은 전혀 보지 않습니다. 속성이 똑같고 id가 다른 두 정점은 같지 않습니다:

```
CREATE GRAPH twins;
SET graph_path = twins;
CREATE VLABEL pair;
CREATE (:pair {k: 1}), (:pair {k: 1});
```


© 2026 SKAI Worldwide Co., Ltd.

```
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
```

순서도 같은 규칙에서 따라옵니다. `a < b`는 `graphid` 값을 비교하므로, 먼저 레이블 순서이고 그다음 레이블 안에서는 삽입 순서입니다. 이 순서는 전순서이고 안정적이어서 `ORDER BY`의 동률 해소에 좋습니다. 데이터를 뜻 있게 정렬해 주는 것은 아닙니다.

```
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
```

대신 속성을 비교하려면 속성을 비교하십시오. `properties(a) = properties(b)`처럼 쓰거나, 뜻하는 키를 이름으로 부르십시오.

두 타입 모두 암시적으로 `jsonb`로 형 변환됩니다. 그 변환은 속성 맵을 남기고 식별자는 버립니다. `pg_cast`의 `vertex_to_jsonb`와 `edge_to_jsonb`입니다. 그래서 `m::jsonb`는 `properties(m)`입니다. 반대 방향의 형 변환은 없습니다.

graphpath

`graphpath`는 배열 두 개로 된 복합 타입입니다. `vertices vertex[]`와 `edges edge[]`입니다. 간선 n개인 경로는 정점 n + 1개를 담으므로, 정점 하나에 바인딩된 경로는 정점 하나에 간선이 없습니다:

```
MATCH p = (a:person {name: 'Keanu Reeves'}) RETURN p;
```

```
                          p
-----------------------------------------------------
 [person[4.1]{"born": 1964, "name": "Keanu Reeves"}]
```

보행 순서대로 정점·간선·정점이 번갈아 오는 구성원 목록을 대괄호로 감싸고 쉼표로 구분해 출력합니다. 두 배열이 두 개의 배열로 출력되는 것이 아니라 서로 엇갈려 놓입니다. 그래서 간선 하나인 경로는 구성원 세 개를, 간선 n개인 경로는 2n + 1개를 출력합니다.

구성원에는 `nodes()`와 `edges()`로 닿습니다. 그 두 함수가 두 배열을 돌려줍니다. `length()`는 간선 개수입니다:

```
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`에는 **자기 연산자가 없습니다.** 같음도, 순서도 없습니다. Cypher에서 경로 둘을 비교하는 것은 거부됩니다. 호출할 연산자가 없으니 그 표현식은 `jsonb`로 되돌아가는데, 경로는 `jsonb` 값이 아닙니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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;
               ^
```

SQL에서는 비교가 됩니다. 복합 값에 대한 PostgreSQL의 일반 연산자를 통해서이고, 그것은 두 배열을 필드 단위로 훑습니다:

```
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
```

그 비교에는 `tid` 필드도 들어가므로, 그 경로가 뜻하는 것을 비교하는 것이 아니라 보행 전체의 식별자를 비교하는 것입니다. 추론할 수 있는 것을 원하면 `nodes(p)` 나 id를 비교하십시오.

`to_jsonb()` 는 Cypher에서든 SQL에서든 경로를 변환합니다. 그 결과는 내부 구조를 드러냅니다. 두 배열과 모든 구성원의 `tid` 입니다. `pg_cast` 에는 `graphpath` 에서 `jsonb` 로 가는 형 변환이 없습니다. 그래서 `p::jsonb` 는 그것을 컴파일해 주는 Cypher에서는 동작하고, SQL에서는 `cannot cast type graphpath to jsonb` 로 실패합니다.

rowid

`rowid` 는 테이블 OID와 행 포인터를 한 쌍으로 묶습니다. 최단 경로 실행기가 간선 식별자를 나르려고 내부에서 씁니다. 비교 연산자 여섯 개와 btree 연산자 클래스가 있고, **출력 함수는 없습니다**:

```
SELECT rowid('moviekb.person'::regclass, ctid) FROM moviekb.person LIMIT 1;
```

```
ERROR:  "rowid" type does not support output function
```

`rowid_tableoid()` 와 `rowid_ctid()` 가 두 절반을 읽으며, 그 값을 볼 수 있는 방법은 그것뿐입니다:

```
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)
```

그 OID는 해당 데이터베이스에서 `moviekb.person` 이 우연히 갖게 된 값이므로, 설치할 때마다 다릅니다.

Cypher에서 `rowid` 를 만들어 내는 것은 없고, 애플리케이션 코드에서 그것을 쓸 이유도 없습니다. `pg_type` 과 `\dT` 에 나타나기 때문에 여기 문서화해 둔 것입니다.

© 2026 SKAI Worldwide Co., Ltd.

그래프 타입의 배열

`vertex[]`, `edge[]`, `graphpath[]`는 자기 출력 함수를 가진 평범한 PostgreSQL 배열 타입이므로, 그 요소와 같은 대괄호 방식으로 출력됩니다.

정점·간선·경로에 대한 `collect()`는 이런 네이티브 배열 중 하나를 반환합니다. 그 밖의 것에 대한 `collect()`는 `jsonb`를 반환합니다:

```
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"
```

네이티브 배열은 각 요소의 `jsonb` 사본이 아니라 요소 자체를 담으므로, 그 구성원이 자기 식별자를 유지합니다. `films[0]`은 `vertex`이고, 거기서 속성을 읽거나 뒤따르는 절에서 그것을 따라갈 수 있습니다.

리스트 함수 몇 개는 `jsonb`용 오버로드와 네이티브 배열용 오버로드를 함께 갖고 있습니다. 어느 것을 받는지는 인자에서 따라옵니다. Cypher에서 인자는 보통 `jsonb`이므로 결과도 `jsonb`입니다. SQL에서 같은 함수에 실제 배열을 주면 실제 배열이 돌아옵니다. 시그니처는 **타입 변환 함수**에 있습니다.

배열 첨자의 시작

Cypher는 리스트의 첨자를 0부터 세고 음수 첨자를 받으며, 그 규칙을 네이티브 배열에도 적용합니다:

```
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;
```

```
 cypher_index_0 |    cypher_index_1     | n
----------------+-----------------------+---
 "The Matrix"   | "The Matrix Reloaded" | 4
```

SQL에서 닿은 같은 배열은 PostgreSQL의 규칙을 따라 1부터 시작하고, 첨자 0은 범위를 벗어납니다:

```
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()`가 그것을 다시 행으로 바꾸며, SQL에서는 보통 그것이 원하는 것입니다:

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

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

속성: jsonb

레이블에는 `jsonb` 타입의 `properties` 컬럼이 하나 있고, 그 레이블의 모든 정점과 간선의 모든 속성이 그 안에 있습니다. 그래서 속성에는 선언된 타입이 없습니다. 그 타입은 값이 지닌 JSON 타입이 무엇이든 그것입니다.

속성 값이 될 수 있는 것

JSON의 여섯 타입 모두입니다:

JSON 타입	Cypher 표기	비고
문자열	`'text'`	작은따옴표입니다. 이스케이프는 JSON의 규칙을 따릅니다.
숫자	`42` , `1.5` , `1e3`	JSON 숫자 타입 하나뿐이고, 정수와 실수가 따로 없습니다.
불리언	`true` , `false`	
배열	`[1, 'two', null]`	타입을 섞을 수 있고 중첩됩니다.
객체	`{a: {b: 1}}`	키는 따옴표를 붙이지 않으면 소문자로 바뀝니다.
null	`null`	null을 할당하면 그 속성이 **지워집니다.** `allow_null_properties` 를 켜 둔 경우는 예외입니다.

```
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
```

없는 속성과 null인 속성은 같은 것입니다. 그래서 없는 키에 대한 `jsonb_typeof` 는 `"null"` 이 아니라 null입니다.

숫자 타입은 하나이지만 적은 소수 자릿수가 저장되고, 그것이 나눗셈에서 버림이 일어나는지를 결정합니다:

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 a |  b   | a_div |       b_div
---+------+-------+--------------------
 7 | 7.0 | 3     | 3.5000000000000000
```

`n.a` 와 `n.b` 는 둘 다 `jsonb_typeof` 가 `number` 이고 같다고 비교되므로, 이것은 두 개의 타입이 아닙니다. 소수 자릿수를 몇 개 받았는지 기억하는 하나의 타입입니다.

선언된 타입이 없으므로, 타입이 서로 다른 값의 순서는 `jsonb` 비교가 결정합니다. null, 그다음 문자열, 그다음 숫자, 그다음 불리언, 그다음 배열, 그다음 객체입니다. 그 순서는 PostgreSQL의 것입니다. 타입 컬럼으로 승격한 속성이 같은 속성을 `properties` 에서 읽은 것과 다르게 정렬될 수 있는 이유가 그것입니다.

RETURN이 돌려주는 것

Cypher의 `RETURN` 은 각 스칼라 출력 컬럼을 `jsonb` 로 감쌉니다. `boolean` 은 불리언으로 남고, 그래프 타입은 자기 자신으로 남습니다:

```
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
----------+----------------------+----------------+----------------+---------+---------
+--------+-----------
 jsonb    | jsonb                | jsonb          | jsonb          | boolean | graphid | vert
ex | graphpath
```

두 번째 컬럼을 보십시오. `RETURN` 안의 `::text` 형 변환은 그 감싸기에 의해 무효가 되므로 아무것도 바꾸지 않습니다. 그래서 문자열은 JSON 따옴표와 함께 출력됩니다:

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

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

평범한 스칼라를 얻으려면 SQL에서 그 값을 `jsonb` 에서 꺼내십시오. `#>> '{}'` 이 문서 전체를 텍스트로 꺼냅니다:

```
SELECT title #>> '{}' AS title
FROM (MATCH (m:movie {title: 'The Matrix'}) RETURN m.title AS title) q;
```

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

감싸는 방법은 SQL과 Cypher 섞기가 온전히 다룹니다. 이 감싸기는 속성에서 끌어낸 컬럼에만 적용됩니다. 불리언 표현식은 `boolean` 으로, `id(n)` 은 `graphid` 로, 그래프 요소는 `vertex` · `edge` · `graphpath` 로 남습니다. 타입 컬럼으로 승격된 속성도 예외입니다. 그 읽기는 그 컬럼에서 답하고 그 컬럼 자신의 타입으로 돌아옵니다.

© 2026 SKAI Worldwide Co., Ltd.

:: 형 변환

Cypher의 `::` 은 SQL의 것보다 적은 타입 이름 집합을 받습니다. `text` , `date` , `uuid` , `jsonb` , `inet` 같은 맨 타입 이름이나 함수 이름을 받습니다. SQL 문법이 특별하게 다루는 묶음 표기도 받습니다. 수치 이름(`int` , `integer` , `smallint` , `bigint` , `real` , `float` , `double precision` , `decimal` , `numeric` , `boolean`)과 문자 이름(`char` , `varchar` , `character varying`), 그리고 `bit` , `bit varying` , `json` , `interval` 입니다.

```
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"
```

이유는 형 변환이 타입을 어떻게 지칭하는가에 있습니다. Cypher의 형 변환은 평범한 타입 이름을 받습니다. 그리고 언어가 특별하게 다루는 수치·문자·비트·json·interval 표기도 받습니다. `TIMESTAMP` 와 `TIME` 도 특별한 표기인데, 빠진 것이 바로 그 둘입니다. 그래서 둘 다 거부됩니다:

```
RETURN '2020-01-01 10:00'::timestamp;
```

```
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;
                        ^
```

그 타입 이름에 따옴표를 붙이십시오. 인용된 이름은 키워드가 아니라 식별자이므로, 동작하는 규칙을 타고 갑니다. 같은 타입을 가리킵니다:

```
RETURN '2020-01-01 10:00'::"timestamp" AS ts, '10:00'::"time" AS t;
```

```
          ts           |     t
-----------------------+------------
 "2020-01-01T10:00:00" | "10:00:00"
```

그 밖의 시간 관련 타입은 모두 평범한 이름이라 따옴표가 필요하지 않으며, `date` 와 `timestamptz` 도 그렇습니다:

```
RETURN pg_typeof('2020-01-01'::date) AS d, pg_typeof('2020-01-01'::timestamptz) AS tz;
```

```
   d    |             tz
--------+----------------------------
 "date" | "timestamp with time zone"
```

다른 길은 Cypher 밖에서 형 변환하는 것이며, 거기서는 SQL 문법 전체를 쓸 수 있습니다:

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT (t #>> '{}')::timestamp AS released FROM (RETURN '2020-01-01 10:00' AS t) q;
```

```
      released
--------------------
 2020-01-01 10:00:00
```

두 결과의 차이를 보십시오. Cypher 안에서는 값이 다시 `jsonb`로 감싸이므로 JSON의 ISO 형태로 따옴표와 함께 출력됩니다. SQL 형 변환은 실제 `timestamp`를 주고, 그 타입의 출력 함수가 그것을 출력합니다.

알아 둘 비대칭이 하나 더 있습니다. 문자열 리터럴에 쓴 `'…'::jsonb`는 파싱된 문서가 아니라 `jsonb` **문자열**을 만듭니다. 그 리터럴이 이미 JSON 문자열이기 때문입니다. Cypher에서 `jsonb` 객체를 만들려면 맵 리터럴을 쓰십시오:

```
RETURN '{"a": 1}'::jsonb AS as_string, {a: 1} AS as_map;
```

```
  as_string   |  as_map
--------------+----------
 "{\"a\": 1}" | {"a": 1}
```

쓸 수 있는 PostgreSQL 타입

전부입니다. 그래프는 여러분의 테이블과 같은 데이터베이스에 있으므로, **PostgreSQL 데이터 타입** 장의 모든 타입을 그대로 쓸 수 있습니다. 수치, 문자, 날짜·시각, 불리언, 열거, 기하, 네트워크 주소, 비트 문자열, 텍스트 검색, UUID, XML, JSON, 배열, 복합, 범위, 도메인, 객체 식별자가 그렇습니다. 확장이 더한 타입도 쓸 수 있습니다.

PostgreSQL 타입이 그래프와 만나는 자리가 세 곳입니다:

- 그래프 옆의 **일반 테이블**이며, 한 문장에서 그래프와 조인합니다. `timestamptz`, `numeric(10,2)`, `tsvector`의 의미와 연산자가 필요할 때 그것들이 있어야 할 자리가 여기입니다.
- 레이블의 **승격된 컬럼**입니다. 속성 하나를 실제 타입의 실제 컬럼으로 구체화하므로, 그 속성이 그 타입의 순서·비교·인덱스 지원을 얻습니다.
- 속성이나 승격된 컬럼에 저장한 **확장 타입**입니다. `pgvector`의 `vector`가 **벡터 검색**이 기반으로 삼는 경우입니다.

속성 자체는 그 타입들 중 어느 것도 될 수 없습니다. 속성은 `jsonb`이고, 그 안에 넣은 값은 들어가는 길에 JSON 타입으로 변환됩니다.

호환성

openCypher

`vertex`, `edge`, `graphpath`가 openCypher 타입 체계의 `node`·`relationship`·`path`에 해당합니다. 여기서 스칼라 타입은 따로 있지 않습니다. 문자열·숫자·불리언·리스트·맵이 모두 `jsonb` 입니다. 그래서 어떤 숫자가 정수인지 소수인지는 선언된 타입이 아니라 그 값에서 옵니다.

시간 값은 그래프 타입이 아닙니다. `date`, `timestamptz`, `interval`은 PostgreSQL의 것이고, 형 변환, 일반 컬럼, 승격된 컬럼을 통해 닿습니다. 그래프 수준의 `datetime`이나 `duration`은 없습니다. 차이의 전체 목록은 **Cypher 호환성**입니다.

ISO/IEC 39075

표준의 속성 그래프 타입도 같은 방식으로 `vertex`·`edge`·`graphpath`에 대응됩니다. 속성마다 타입이 선언되는 표준의 모형은 승격된 컬럼이 제공합니다. **GQL 적합성**을 참고하십시오.

관련 항목

- 구문 — 리터럴, 연산자, 그리고 값을 적는 방법

© 2026 SKAI Worldwide Co., Ltd.

- 서버 설정 — `allow_null_properties`와 `enable_property_promotion`
- 그래프가 어떻게 저장되는가 — 레이블 테이블은 무엇으로 되어 있는가
- SQL과 Cypher 섞기 — 경계를 넘어 값을 옮기기
- 시스템 카탈로그 — `ag_graph`, `ag_label` 등
- PostgreSQL: 데이터 타입 — 그 밖에 쓸 수 있는 모든 타입

절

Cypher 문장과 GQL 문장은 절로 이루어진 파이프라인입니다. 이 페이지들은 알파벳순이 아니라 절이 그 파이프라인에서 하는 일로 묶여 있고, 모든 페이지가 같은 모양입니다.

GQL로 표시된 절은 ISO/IEC 39075에서 온 것입니다. Cypher 문장이 허용되는 곳이면 어디서나 허용되며 별도의 모드는 없습니다. 표준의 어느 부분이 지원되는지는 GQL 적합성을 참고하십시오.

읽기

MATCH

GQL 절

그래프에서 패턴이 들어맞는 모든 자리를 찾고, 일치마다 변수 한 행을 바인딩합니다.

개요

```
[ OPTIONAL ] MATCH pattern [, pattern ] ... [ WHERE condition ]
```

여기서 pattern은 정점 패턴과 간선 패턴을 이어 놓은 것이며, 이름을 붙일 수도 있습니다:

```
[ path_variable = ] node [ edge node ] ...
```

node는 소괄호로 씁니다:

```
( [ variable ] [ : label [ ONLY ] ] [ { property_map } ] )
```

edge는 하이픈 두 개 사이의 대괄호로 쓰고, 그 안에 edge_body를 둡니다:

```
[ variable ] [ : label [ ONLY ] ] [ * [ min ] [ .. [ max ] ] ] [ { property_map } ]
```

그 주위의 하이픈과 화살촉이 방향을 나타냅니다. 세 가지가 있습니다:

```
-[ edge_body ]->     left to right
<-[ edge_body ]-     right to left
-[ edge_body ]-      either direction
```


© 2026 SKAI Worldwide Co., Ltd.

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`MATCH`는 그래프를 읽습니다. 자기 패턴을 저장된 정점과 간선 위에 얹을 수 있는 모든 방식마다 행 하나를 내놓습니다. 패턴의 이름 있는 요소는 각각 그것이 일치시킨 정점·간선·경로에 바인딩됩니다. 파이프라인의 뒤따르는 절이 그 바인딩을 봅니다. 파이프라인 가운데의 `MATCH`는 들어오는 행마다 한 번 실행되어 자기 일치를 그 행에 조인하며, 관계형 쿼리에서 조인이 행을 더하는 것과 같습니다.

- **변수가 없는** 패턴 요소도 일치됩니다. 뒤에서 그것을 가리킬 수 없을 뿐입니다. 그 요소가 모양을 제약하는 데만 필요하다면 변수를 빼십시오.
- **레이블**은 요소를 그 레이블과 자식 레이블로 제한하고, `ONLY`는 그 레이블 자체로 제한합니다. 레이블이 없으면 요소는 그래프의 모든 정점 또는 모든 간선에 일치합니다.
- 패턴 안의 **속성 맵**은 그것이 적은 키마다 같음을 검사하는 것입니다. 축약형입니다. `(m:movie {year: 1999})`와 `(m:movie) WHERE m.year = 1999`는 같은 정점에 일치합니다.
- **쉼표**는 하나의 `MATCH` 안에서 독립적인 패턴을 구분합니다. 변수를 공유하는 패턴은 그 변수로 조인되고, 아무것도 공유하지 않는 패턴은 곱집합을 만듭니다.
- 하나의 패턴 일치 안에서 **같은 간선을 두 번 쓰지 않습니다.** 정점 바인딩은 반복될 수 있어서 패턴이 삼각형을 닫을 수는 있지만, 두 홉 패턴이 자기가 지나온 간선으로 되돌아가는 일은 없습니다.
- **간선 변수**는 패턴마다 한 번만 쓸 수 있습니다. 다시 쓰면 `duplicate variable`이 나옵니다. **정점 변수**는 반복할 수 있고, 그러면 두 자리가 같은 정점으로 제약됩니다.
- `MATCH`는 쿼리를 시작하거나 첫 절 뒤 어디에든 놓일 수 있습니다. 쿼리를 끝내지는 않습니다. 쿼리는 `RETURN`, `FINISH`, 또는 쓰기 절로 끝나야 합니다.

`OPTIONAL MATCH`는 패턴이 아무것도 찾지 못한 입력 행을 남기고, 그 행을 버리는 대신 패턴의 변수를 null로 바인딩합니다. 자세한 내용은 `OPTIONAL MATCH`를 참고하십시오.

매개변수

pattern	경로 패턴 하나이며, 앞에 `path_variable =`을 붙일 수도 있습니다. 쉼표로 구분해 여러 개 나열할 수 있습니다.
path_variable	일치한 경로 전체에 바인딩되는 이름이며, 그 정점과 간선을 순서대로 담은 값입니다. 경로 함수 `length()`·`nodes()`·`edges()`로 읽습니다. 그 이름은 식별자여야 합니다. `path`처럼 예약되지 않은 키워드도 따옴표로 감싸지 않으면(`"path"`) 이 자리에서는 거부됩니다.
variable	일치한 정점이나 간선에 바인딩되는 이름입니다. 생략할 수 있습니다. 이 절 뒤의 모든 절에서 보입니다.
label	정점 패턴에서는 정점 레이블, 간선 패턴에서는 간선 레이블입니다. 요소마다 레이블 하나입니다.
ONLY	자식 레이블을 제외하므로, 그 요소는 해당 레이블 자기 테이블의 행에만 일치하고 그것을 상속한 것에는 일치하지 않습니다. 레이블이 있어야 합니다.

© 2026 SKAI Worldwide Co., Ltd.

property_map	맵 리터럴, 또는 그것을 담은 쿼리 매개변수입니다. 각 키는 그 요소의 같은 이름 속성과 같은지 비교됩니다.
min, max	가변 길이 간선 패턴이 지나는 간선 수의 경계입니다. `*n` 은 정확히 n 홉, `*m..n` 은 m과 n 사이, `*m..` 은 m 이상, `*..n` 은 n 이하, `*` 만 쓰면 하나 이상입니다.
condition	일치 결과를 걸러 내는 불리언 표현식이며, 패턴의 모든 변수가 바인딩된 상태에서 평가됩니다. `WHERE` 를 참고하십시오.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

속성으로 정점 하나 찾기

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p;
```

```
                        p
-------------------------------------------------
 person[4.1]{"born": 1964, "name": "Keanu Reeves"}
(1 row)
```

정점은 `label[id]{properties}` 형태로 출력됩니다. `4.1` 이 그 그래프 id이고, `id(p)` 로 읽습니다. `p.id` 는 `id` 라는 이름의 속성을 읽는 것이고, 그런 속성은 없습니다.

한 방향으로 간선 따라가기

```
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)
```

양 끝점과 간선 모두에 레이블이 있고, 간선은 `r` 에 바인딩되어 그 `role` 속성을 투영할 수 있습니다. Cypher의 `RETURN` 이 jsonb를 내놓으므로 문자열 값은 따옴표와 함께 돌아옵니다.

필요하지 않은 요소 빼기

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m)
RETURN m.title AS film
ORDER BY film;
```

```
      film
----------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Inception"
 "Interstellar"
(2 rows)
```

간선에는 변수가 없고 먼 쪽 끝점에는 레이블이 없습니다. 둘 다 여전히 일치를 제약하며, 어느 쪽도 바인딩 비용을 들이지 않습니다.

어느 방향이든 간선 일치시키기

화살촉이 없는 간선 패턴은 간선이 어느 쪽을 가리키든 일치시킵니다:

```
MATCH (m:movie {title: 'Total Recall'})-[:related_to]-(n:movie)
RETURN n.title AS neighbour
ORDER BY neighbour;
```

```
  neighbour
--------------
 "Inception"
 "The Matrix"
(2 rows)
```

`Total Recall`에는 나가는 `related_to` 간선 하나와 들어오는 것 하나가 있습니다. 양쪽 끝에 화살촉을 붙인 `<-[:related_to]->`도 같은 뜻입니다.

공유 변수로 두 패턴 조인하기

```
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)
```

두 패턴 모두 `m`을 언급하므로 그것으로 조인됩니다. 연달아 오는 `MATCH` 절 두 개도 같은 결과를 낼 것이고, 가운데에서 만나는 하나의 패턴 `(a:person)-[:acted_in]->(m:movie)<-[:directed]-(d:person)`도 그렇습니다.

경로 전체에 이름 붙이기

경로 자체를 바인딩하려면 패턴 앞에 `variable =`을 두십시오:

```
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"]
```


© 2026 SKAI Worldwide Co., Ltd.

```
 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)
```

정해지지 않은 수의 간선 걷기

간선 패턴에 붙인 `*`가 그것을 반복합니다. 각 경계는 생략할 수 있습니다:

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

```
           reached
-------------------------------
 "Inception"
 "Terminator 2: Judgment Day"
 "The Matrix Revolutions"
(3 rows)
```

하한이 `0`이면 길이 0인 경로가 포함되므로, 시작 정점도 답 중 하나가 됩니다:

```
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)
```

ONLY로 자식 레이블 제외하기

레이블은 자식 레이블에도 일치합니다. 레이블이 테이블이고 자식 레이블이 그것을 상속하기 때문입니다. `ONLY`는 일치 범위를 그 레이블 하나로 좁힙니다. 차이를 보려면 `person`에 자식 레이블을 하나 주십시오:

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person ONLY) RETURN count(*) AS people;
```

```
 people
--------
 28
(1 row)
```

`ONLY` 는 레이블 뒤, 속성 맵 앞에 놓입니다. `(p:person ONLY {name: 'Keanu Reeves'})` 와 같이 씁니다. 간선 패턴 `-[r:acted_in ONLY]->` 에서도 같게 동작합니다. 준비한 것은 이렇게 되돌립니다:

```
MATCH (p:crew) DETACH DELETE p;
DROP VLABEL crew;
```

```
UPDATE 1
DROP VLABEL
```

WHERE로 일치 결과 걸러 내기

`MATCH` 에 붙인 `WHERE` 는 뒤에 오는 단계가 아니라 그 일치의 일부입니다:

```
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)
```

여러 레이블의 간선 일치시키기

간선 패턴에는 레이블이 최대 하나입니다. 둘을 나열하면 오류가 납니다:

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

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

대신 레이블 없는 간선을 일치시키고 그 레이블을 `WHERE` 에서 검사하십시오:

```
MATCH (p:person)-[r]->(m:movie)
WHERE label(r) IN ['acted_in', 'directed']
RETURN count(r) AS edges;
```

```
 edges
-------
 46
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

`label()` 은 요소의 레이블을 문자열로 반환합니다. 그래서 이름 목록에 대한 `IN` 이 일반적인 형태이고, 간선 레이블의 집합에 일치시키는 유일한 방법입니다.

참고

레이블 검사는 `WHERE` 안의 콜론이 아니라 `label(n) = '...'` 로 씁니다. `WHERE n:movie` 는 구문 오류이고, 패턴만 조건으로 쓴 것도 그렇습니다. `WHERE (m)-[:related_to]->(:movie)` 는 파싱되지 않습니다. 패턴이 존재하는지 검사하려면 `WHERE EXISTS { MATCH (m)-[:related_to]->(:movie) }` 를 쓰십시오.

속성 맵은 jsonb 값을 비교하고, 속성과의 비교도 그렇습니다. 리터럴의 타입이 저장된 타입과 맞지 않을 때 그것이 문제가 됩니다. **`WHERE`** 를 참고하십시오.

> 참고
>
> 이름 붙인 경로 변수는 요소 변수보다 좁게 파싱됩니다. `(path)` 는 받아들여지는 정점 변수이지만, `path` 가 키워드이므로 `MATCH path = ...` 는 구문 오류입니다. `MATCH p = ...` 로 쓰거나 그 이름을 따옴표로 감싸십시오.

호환성

레이블 표현식 (ISO/IEC 39075 §16.8)

표준의 `<label expression>` 은 레이블 이름을 논리합은 `|` , 논리곱은 `&` , 부정은 `!` 로 엮습니다. AgensGraph는 패턴 요소마다 레이블 이름 하나만 받으므로 셋 중 어느 것도 쓸 수 없습니다. `-[r:acted_in|directed]->` 는 `multiple types for relationship not supported` 를 내고, `-[r:acted_in|:directed]->` 는 구문 오류입니다. openCypher도 논리합을 파이프로 표기하는데, 여기서는 고정 길이 간선, 가변 길이 간선, `shortestpath` 안에서 모두 거부됩니다. 위에 보인 `WHERE label(r) IN [...]` 을 쓰십시오.

레이블 술어 (ISO/IEC 39075 §19.9)

표준은 레이블 검사를 `n IS LABELED movie` 또는 `n:movie` 로 씁니다. AgensGraph는 두 표기 모두 받지 않습니다. `label(n) = 'movie'` 를 쓰십시오.

ONLY 수정자 (ISO/IEC 39075 §16.7)

표준의 `<element pattern filler>` 는 변수 선언, 레이블 표현식, 조건을 담고 그 밖에는 아무것도 담지 않습니다. `ONLY` 가 없습니다. AgensGraph가 그것을 더한 것은, 여기서 레이블이 자식 레이블이 상속하는 테이블이고 따라서 레이블 일치가 상속 트리 전체에 대한 일치이기 때문입니다. **레이블 상속**을 참고하십시오.

관련 항목

- **`OPTIONAL MATCH`** — 패턴이 일치시키지 못한 행 남기기
- **`WHERE`** — 일치가 지닐 수 있는 조건
- **`CREATE`** 와 **`MERGE`** — 쓰기에 쓰는, 같은 패턴 구문
- **`LOAD FROM`** — 같은 파이프라인으로 테이블을 읽어 들이기
- **함수** — `length()` , `nodes()` , `edges()` 와 그 밖의 경로 함수
- **쿼리하기** — `shortestpath` · `allshortestpaths` · `dijkstra` 형태를 포함한 경로 쿼리

OPTIONAL MATCH

GQL 절

`MATCH` 와 같은 방식으로 패턴을 일치시키지만, 패턴이 아무것도 찾지 못한 입력 행을 남겨 두고 그 행에 대해서는 패턴의 변수를 null로 바인딩합니다.

© 2026 SKAI Worldwide Co., Ltd.

개요

OPTIONAL MATCH pattern [, pattern] ... [**WHERE** condition]

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`OPTIONAL MATCH` 는 `MATCH` 와 같은 패턴을 받아 같은 방식으로 일치시킵니다. 차이는 패턴이 일치하는 것을 찾지 못했을 때 무슨 일이 일어나는가입니다. 그냥 `MATCH` 는 그 행을 버립니다. `OPTIONAL MATCH` 는 그 행을 남기고 패턴이 도입하는 모든 변수를 null로 바인딩합니다. `LEFT JOIN` 에 대응하는 그래프 쪽 절입니다.

- 이미 유효 범위에 있는 변수는 자기 값을 유지합니다. null이 될 수 있는 것은 선택적 패턴이 도입하는 변수뿐입니다.
- `OPTIONAL MATCH` 에 붙인 `WHERE` 는 **그 선택적 일치의 일부**입니다. 패턴은 일치했지만 조건을 통과하지 못한 행은 버려지지 않고 null과 함께 남습니다. 버리려면 절 뒤에서 `WITH` 나 `FILTER` 로 걸러 내십시오.
- `OPTIONAL MATCH` 로 쿼리를 시작하면, 패턴이 아무것도 일치시키지 못했을 때 행이 없는 것이 아니라 전부 null인 행 하나가 나옵니다.
- 한 절 안에 쉼표로 구분한 패턴이 여럿이면 그것들은 함께 선택적입니다. 전부 일치하거나, 절 전체가 null을 내놓습니다.

매개변수

pattern	경로 패턴이며, `MATCH` 가 쓰는 것과 같은 구문입니다. 정점·간선·경로· `ONLY` 형태는 `MATCH` 를 참고하십시오.
condition	그 일치의 일부로 평가되는 불리언 표현식입니다. 이 조건을 통과하지 못한 행은 패턴의 변수가 null인 채로 남습니다.

예제

모든 예제는 예제 데이터셋에서 실행됩니다.

연출 여부와 무관하게 사람을 나열하기

```
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)
```

셋 중 둘은 아무것도 연출하지 않았으므로 그 둘에 대해 `m` 이 null이고, 따라서 `m.title` 도 null입니다. 둘째 줄을 그냥 `MATCH` 로 바꾼 같은 쿼리는 한 행을 반환합니다:

```
MATCH (p:person) WHERE p.born > 1974
MATCH (p)-[:directed]->(m:movie)
```


© 2026 SKAI Worldwide Co., Ltd.

```
RETURN p.name AS person, m.title AS directed
ORDER BY person;
```

```
      person      | directed
------------------+-----------
 "Neill Blomkamp" | "Elysium"
(1 row)
```

일치하지 못한 행만 남기기

절 뒤에서 그 선택적 변수가 null인지 검사하십시오. 아래는 나가는 `related_to` 간선이 없는 공상 과학 영화를 찾습니다:

```
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;
```

```
        film
---------------------
 "Avatar"
 "Blade Runner 2049"
 "Elysium"
 "Gattaca"
 "Inception"
 "The Fifth Element"
(6 rows)
```

WHERE가 어디에 놓이는지 보기

`Jurassic Park`에는 1947년생 Sam Neill 한 명이 배우로 있습니다. 조건을 선택적 일치 안에 두면 그 영화는 배우가 null인 채로 살아남습니다:

```
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)
```

같은 조건을 절 뒤에 적용하면 그 영화는 버려집니다:

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

선택적 일치로 쿼리를 시작하기

```
OPTIONAL MATCH (m:movie {title: 'Nope'})
RETURN m;
```

```
 m
---

(1 row)
```

`m`이 null인 행 하나입니다. 여기서 그냥 `MATCH`를 쓰면 행이 아예 없습니다.

참고

null인 정점의 속성을 읽으면 오류가 아니라 null이 나오므로, 패턴이 일치시키지 못한 행에서도 `m.title`은 안전합니다. 그 행을 알아내는 믿을 만한 방법은 변수 자체를 검사하는 것, `WHERE n IS NULL`입니다. 일치한 정점이라도 검사하려던 그 속성을 정당하게 갖고 있지 않을 수 있기 때문입니다.

호환성

선택적 블록 (ISO/IEC 39075 §14.4)

표준의 `<optional match statement>`는 중괄호나 소괄호로 감싼 일치 문장 블록, `OPTIONAL { MATCH ... MATCH ... }`도 허용합니다. AgensGraph는 `MATCH` 바로 앞에 오는 `OPTIONAL`만 받으며, 블록 형태는 구문 오류입니다. `OPTIONAL`은 `CALL` 앞에도 붙습니다.

관련 항목

- **`MATCH`** — 패턴 구문과 선택적이 아닌 형태
- **`WHERE`** — 조건, 그리고 그것이 효력을 갖는 자리
- **`WITH`** — 선택적 일치 뒤에서 걸러 내는 절
- **`CALL { subquery }`** — `OPTIONAL`이 이 절 앞에도 붙는다

WHERE

GQL 절

패턴을 제약하거나 투영을 걸러 냅니다. 조건이 참이 아닌 행은 버려집니다.

개요

WHERE condition

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`WHERE`는 그 자체로 절이 아닙니다. 세 절 중 하나에 붙고, 붙은 절에서 뜻을 얻습니다:

- **`MATCH`**에 붙으면 그 일치의 일부입니다. 조건이 실행될 때 패턴의 모든 변수가 바인딩돼 있고, 조건을 통과하지 못한 일치는 만들어지지 않습니다.
- **`OPTIONAL MATCH`**에 붙으면 그 선택적 일치의 일부입니다. 조건을 통과하지 못한 행은 패턴의 변수가 null인 채로 **남습니다.**
- **`WITH`**에 붙으면 집계가 끝난 뒤 투영된 행을 걸러 냅니다. 집계에 대한 조건이 놓이는 자리가 여기입니다.

© 2026 SKAI Worldwide Co., Ltd.

조건은 불리언 표현식이어야 합니다. 다른 타입의 값은 강제 변환되지 않습니다. 조건에 놓인 jsonb 문자열이나 숫자는 참으로 취급되는 것이 아니라 오류를 냅니다. `WHERE`에는 집계 함수를 쓸 수 없습니다. 그것은 `WITH`에 넣고 그 결과를 걸러 내십시오.

`WHERE`만 쓰는 것은 구문 오류이고, 쿼리가 그것으로 끝날 수도 없습니다. 홀로 서고 이어 쓸 수 있는 필터는 **`FILTER`**이며, `FILTER condition`과 `FILTER WHERE condition` 둘 다 받습니다.

매개변수

condition	불리언 타입의 표현식입니다. 여러 검사는 `AND`, `OR`, `XOR`, `NOT`으로 엮으십시오.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

속성 비교하기

```
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)
```

조건 엮기

```
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)
```

`AND`, `OR`, `NOT`과 함께 `XOR`도 쓸 수 있습니다:

```
MATCH (m:movie) WHERE m.genre = 'Action' XOR m.year < 1995
RETURN count(*) AS n;
```

```
 n
---
 8
(1 row)
```

레이블 검사하기

`label()`은 요소의 레이블을 문자열로 반환하며, 정점과 간선 모두 마찬가지입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (n) WHERE label(n) = 'person'
RETURN count(*) AS n;
```

```
 n
----
 28
(1 row)
```

이름 목록에 대한 `IN`이, 간선 패턴으로는 표현할 수 없는 「여러 간선 레이블에 일치시키기」를 하는 방법입니다:

```
MATCH (p:person)-[r]->(m:movie)
WHERE label(r) IN ['acted_in', 'directed']
RETURN count(r) AS edges;
```

```
 edges
-------
 46
(1 row)
```

부분 문자열이나 패턴으로 일치시키기

`STARTS WITH`, `ENDS WITH`, `CONTAINS`는 문자열 속성을 검사합니다:

```
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)
```

`=~`는 POSIX 정규 표현식으로 일치시킵니다:

```
MATCH (m:movie) WHERE m.title =~ '^Blade.*'
RETURN m.title AS film
ORDER BY film;
```


© 2026 SKAI Worldwide Co., Ltd.

```
        film
---------------------
 "Blade Runner"
 "Blade Runner 2049"
(2 rows)
```

부정은 `NOT`으로 하십시오. 속성에 쓰는 `!~`도, `NOT LIKE`도 없습니다:

```
MATCH (m:movie) WHERE NOT m.title STARTS WITH 'The'
RETURN count(*) AS n;
```

```
 n
----
 11
(1 row)
```

없는 속성 검사하기

설정된 적 없는 속성은 null로 읽히므로, 그것이 없는지 검사하는 것이 `IS NULL`입니다:

```
MATCH (m:movie) WHERE m.runtime IS NULL
RETURN count(*) AS n;
```

```
 n
----
 18
(1 row)
```

패턴 검사하기

서브쿼리 표현식은 패턴에서 아무것도 바인딩하지 않고 그 패턴이 존재하는지 검사합니다:

```
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)
```

WITH로 집계를 걸러 내기

`WHERE` 안의 집계 함수는 거부됩니다:

```
MATCH (p:person)-[:acted_in]->(m:movie) WHERE count(m) > 1
RETURN p.name;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  aggregate functions are not allowed in WHERE
```

`WITH`에서 집계하고, 그다음 그 결과를 걸러 내십시오. `WITH` 절 자신의 `WHERE`는 그룹화 뒤에 실행됩니다:

```
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)
```

참고

속성은 jsonb 값을 담고 있고, 그것을 리터럴과 비교하는 것은 jsonb 값끼리의 비교입니다. jsonb 순서에서 숫자는 모든 문자열보다 크므로, 따옴표로 감싼 숫자를 수치 속성과 비교하면 뜻한 비교가 되지 않습니다:

```
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)
```

저장된 타입이 예상과 다르다면, 리터럴을 그 속성의 타입으로 쓰거나 `m.year::int > 1990`처럼 속성을 형 변환하십시오.

> **참고**
>
> `WHERE`는 불리언만 받습니다. `WHERE m.title`은 `cannot cast "The Matrix" (jsonb type string) to boolean`을 냅니다. 참·거짓을 느슨하게 보는 언어라면 빈 문자열이 아닌 것을 참으로 취급할 자리입니다. 명시적으로 비교하거나 `IS NOT NULL`을 쓰십시오.

호환성

레이블 술어 (ISO/IEC 39075 §19.9)

표준은 레이블 검사를 `n IS LABELED movie`로 쓰고 `n:movie`를 축약형으로 둡니다. AgensGraph는 둘 다 받지 않습니다. `label(n) = 'movie'`를 쓰십시오.

패턴 술어

패턴만 쓴 것은 표현식이 아니므로 조건으로 쓸 수 없습니다. `WHERE (m)-[:related_to]->(:movie)`는 구문 오류이고, 레이블 없는 `WHERE (m)-[]->()`도 그렇습니다. 표준이 §19.4에 정의하고 AgensGraph가 지원하는 `EXISTS { MATCH ... }`로 감싸십시오.

SQL의 문자열 연산자

`LIKE`와 `NOT LIKE`는 Cypher 조건의 일부가 아닙니다. `WHERE m.title LIKE 'Blade%'`는 형 변환을 붙이든 붙이지 않든 구문 오류입니다. `STARTS WITH`, `ENDS WITH`, `CONTAINS`, `=~`를 쓰십시오. SQL 연산자를 쓰려면 Cypher를 SQL로 감싸십시오. SQL과 Cypher 섞기를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

관련 항목

- `MATCH` — `WHERE`가 가장 자주 붙는 절
- `OPTIONAL MATCH` — 조건을 통과하지 못한 행이 남는 자리
- `WITH` — 투영이나 집계 뒤에 걸러 내기
- `FILTER` — 홀로 서는 필터 절
- 서브쿼리 표현식 — `EXISTS`, `COUNT`, `COLLECT` 등

LOAD FROM

일반 테이블이나 뷰를 Cypher 파이프라인으로 읽어 들이며, 테이블의 행 하나마다 작업 행 하나를 만듭니다.

개요

```
LOAD FROM table_name AS alias
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`LOAD FROM`은 관계형 데이터를 공급하는 절입니다. 테이블을 스캔해 각 행을 alias에 바인딩하고, 그 뒤 파이프라인의 어디서든 그 컬럼을 `alias.column`으로 읽을 수 있게 합니다. 그래프와 테이블이 같은 데이터베이스에 있으므로 가져오기 단계도, 외부 데이터 래퍼도 필요하지 않습니다.

- 별칭의 컬럼은 자기 **SQL 타입**을 유지합니다. `numeric(3,1)` 컬럼에 대한 `r.score`는 numeric이므로, `r.score > 8.5`는 jsonb 비교가 아니라 numeric 비교입니다.
- 별칭 자체는 **행 값**입니다. 그것을 투영하면 행이 jsonb 객체로 나오므로 `CREATE (:label = to_jsonb(alias))`는 행 전체를 적재하는 것이 됩니다.
- `LOAD FROM`은 쿼리를 **시작**할 수도, 다른 절 뒤에 올 수도 있으며, 한 쿼리에 여러 개가 나올 수도 있습니다. 쿼리를 끝내지는 않습니다.
- 이 절에는 **자기 `WHERE`가 없습니다.** 뒤따르는 `MATCH`의 `WHERE`, `WITH`의 `WHERE`, 또는 `FILTER`로 걸러 내십시오.
- `SELECT`가 읽을 수 있는 것이면 무엇이든 됩니다. 테이블, 뷰, 스키마로 한정한 이름이 그렇습니다.

매개변수

table_name	읽을 테이블이나 뷰의 이름이며, 스키마로 한정할 수도 있습니다. 존재하고 읽을 수 있어야 합니다. 알 수 없는 이름은 `relation "..." does not exist`를 냅니다.
alias	각 행에 바인딩되는 이름입니다. 컬럼은 `alias.column`으로, 행 전체는 `alias`로 읽습니다.

예제

모든 예제는 **예제 데이터셋**에서 실행되며, 그 데이터셋에는 일반 테이블 `rating`이 포함돼 있습니다.

테이블 읽기

```
LOAD FROM rating AS r
RETURN r.title AS title, r.source AS source, r.score AS score
```


© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

값에 따옴표가 붙은 것은 컬럼이 jsonb이기 때문이 아니라, Cypher의 `RETURN`이 출력 컬럼을 jsonb로 감싸기 때문입니다. 그 SQL 타입은 그대로입니다:

```
LOAD FROM rating AS r RETURN pg_typeof(r.score) AS t LIMIT 1;
```

```
     t
-----------
 "numeric"
(1 row)
```

테이블을 그래프와 조인하기

테이블을 적재하고 그것에 대해 일치시킵니다. 관계형 컬럼과 그래프 속성이 곧바로 비교됩니다:

```
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)
```

행 전체 읽기

별칭 자체를 투영하면 그 행을 맵으로 얻습니다:

```
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)
```

뷰 읽기

`SELECT`가 읽을 수 있는 테이블(릴레이션)이면 무엇이든 유효한 원본입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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
 "The Matrix"    | 8.7
(3 rows)
```

```
DROP VIEW critic;
```

```
DROP VIEW
```

테이블 행으로 정점 만들기

원하는 속성을 이름으로 적으십시오:

```
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
```

또는 맵 리터럴 대신 `= expression` 을 써서 행 전체를 속성 맵으로 삼으십시오:

```
LOAD FROM tmp_people AS t
CREATE (:person = to_jsonb(t));
```

```
UPDATE 2
```

두 적재를 되돌리고 그 테이블을 지웁니다:

```
MATCH (p:person) WHERE p.name IN ['Ana Lily Amirpour', 'Rian Johnson'] DETACH DELETE p;
```


© 2026 SKAI Worldwide Co., Ltd.

```
UPDATE 4
```

```
DROP TABLE tmp_people;
```

```
DROP TABLE
```

참고

`LOAD FROM`은 읽습니다. 자체 대량 적재 장치도, 병렬 기록기도, 실패한 행을 따로 보내는 경로도 없습니다. 그 뒤에 오는 `CREATE`가 그래프 쓰기 경로를 통해 행 단위로 삽입합니다. 큰 테이블을 적재하는 방법과 그것을 빠르게 만드는 인덱스·로깅 설정은 **데이터 적재**를 참고하십시오.

관련 항목

- `CREATE` — 적재한 행을 정점과 간선으로 바꾸는 절
- `MATCH` — 적재한 행을 그래프와 조인하기
- `WITH` — 적재한 행을 걸러 내고 집계하기
- SQL과 Cypher 섞기 — 반대 방향, SQL 안의 Cypher 쿼리
- 데이터 적재 — 실제 적재 가이드

투영

RETURN

GQL 절

쿼리를 끝내고 그 결과의 컬럼에 이름을 붙입니다.

개요

```
RETURN [ DISTINCT | ALL ] { * | expression [ AS alias ] } [, ...]
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`RETURN`은 마지막 투영입니다. 작업 행마다 한 번씩 자기 표현식을 평가해 그것을 쿼리의 결과 컬럼으로 내놓습니다. 읽기 쿼리는 `RETURN`이나 `FINISH`로 끝나야 합니다. 쓰기 쿼리는 그 둘 중 하나로 끝나거나 쓰기 절 자체로 끝날 수 있습니다.

- **속성에서 끌어낸 컬럼은 jsonb로 돌아옵니다.** `RETURN m.title`은 `"The Matrix"`를 따옴표와 함께 출력하고, `RETURN m.title::text`도 그렇습니다. 평범한 스칼라를 얻으려면 그 쿼리를 SQL로 감싸십시오. **참고**를 보십시오.
- `*`는 유효 범위의 **이름 있는** 바인딩을 모두 투영합니다. 이름을 붙이지 않은 패턴 요소는 투영되지 않습니다.
- 목록 안의 **집계 함수**는 암시적으로 그룹화합니다. 집계되지 않은 표현식이 그룹화 키가 되며, SQL에서 `GROUP BY`를 쓴 것과 같습니다.
- `DISTINCT`는 중복된 결과 행을 없앱니다. `ALL`은 기본값을 명시적으로 적은 것이고 중복을 남깁니다.
- `ORDER BY`, `SKIP` / `OFFSET`, `LIMIT`은 `RETURN`의 일부가 아니라 그 뒤에 오는 **별개의 절**입니다. 각각 자기 페이지가 있습니다. `ORDER BY`와 `SKIP` · `LIMIT`입니다.
- `WITH`와 달리 `RETURN`은 같은 출력 이름을 두 번 허용합니다. 그 뒤에서 그것을 해석해야 하는 것이 없기 때문입니다.

© 2026 SKAI Worldwide Co., Ltd.

매개변수

DISTINCT	결과에서 중복 행을 없앱니다.
ALL	중복 행을 그대로 둡니다. 기본값입니다.
*****	지금 유효 범위에 있는 이름 있는 바인딩을 모두 투영합니다.
expression	유효 범위의 바인딩에 대한 Cypher 표현식이면 무엇이든 됩니다. 정점·간선 변수, 속성 참조, 경로 변수, 함수 호출, 집계 함수, 산술 표현식이나 `CASE` 표현식, 리스트·맵 생성자, 상수가 그렇습니다.
alias	결과 컬럼의 이름입니다. 큰따옴표가 없으면 소문자로 바뀝니다. 적은 대소문자를 유지하려면 따옴표로 감싸십시오. 별칭이 없으면 속성 참조는 그 속성의 이름을 가져가고, 그 밖의 것은 `?column?`이 됩니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

정점 전체 반환하기

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p;
```

```
                         p
-------------------------------------------------
 person[4.1]{"born": 1964, "name": "Keanu Reeves"}
(1 row)
```

정점은 `label[id]{properties}` 형태로 출력됩니다. 간선은 양 끝점을 더해 출력합니다:

```
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)
```

별칭이 있을 때와 없을 때의 속성 반환

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p.born;
```

```
 born
------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 1964
(1 row)
```

별칭 없는 속성 참조는 그 속성의 이름을 가져갑니다. 계산된 표현식에는 가져갈 이름이 없습니다:

```
MATCH (p:person {name: 'Keanu Reeves'})
RETURN p.born + 1;
```

```
 ?column?
----------
 1965
(1 row)
```

따옴표 없는 별칭은 소문자로 바뀌고, 따옴표를 붙인 별칭은 그렇지 않습니다:

```
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)
```

이름 있는 것 전부 반환하기

```
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)
```

이름 있는 요소마다 하나씩 세 컬럼입니다. 그 패턴을 `(p:person {name: 'Ryan Gosling'})-[:acted_in]->(m:movie)`로 적었다면 `RETURN *`은 둘을 줬을 것입니다.

© 2026 SKAI Worldwide Co., Ltd.

중복 행 없애기

`acted_in` 간선 스물다섯 개가 세 장르의 영화를 가리킵니다:

```
MATCH (p:person)-[:acted_in]->(m:movie)
RETURN DISTINCT m.genre AS genre
ORDER BY genre;
```

```
    genre
-------------
 "Action"
 "Adventure"
 "Sci-Fi"
(3 rows)
```

`DISTINCT`가 없으면 같은 쿼리가 25행을 반환합니다. `ALL`이 그 기본값을 명시적으로 적은 형태입니다.

집계

다른 표현식 없이 쓴 집계 함수는 입력 전체를 한 행으로 접습니다:

```
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)
```

그 옆에 다른 표현식을 나열하면 그 표현식이 그룹화 키가 됩니다:

```
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()`는 한 그룹을 리스트로 모읍니다:

```
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)
```

상수 반환하기

`RETURN` 앞에 패턴이 있어야 하는 것은 아닙니다:

© 2026 SKAI Worldwide Co., Ltd.

```
RETURN 1 + 2 AS sum, 'Hello' AS greeting;
```

```
 sum | greeting
-----+----------
 3   | "Hello"
(1 row)
```

참고

속성에서 끌어낸 컬럼이 jsonb로 돌아오므로, 네이티브 타입을 원하는 애플리케이션에게 `RETURN`은 좋지 않은 인터페이스입니다. Cypher를 `SELECT` 서브쿼리에 넣고 거기서 jsonb 값을 벗겨 내십시오. jsonb 연산자가 그 일을 합니다. `->>`는 키를 텍스트로 읽고, `#>> '{}'`은 jsonb 스칼라 컬럼을 텍스트로 꺼냅니다.

```
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)
```

여기서 영화 제목은 따옴표 없는 텍스트이고, `score`는 그 테이블 자신의 numeric입니다. 이 방식은 SQL과 Cypher 섞기가 온전히 다룹니다.

호환성

컬럼 타입

ISO/IEC 39075 §14.11의 `<return statement>`는 결과 컬럼에 그 표현식의 선언된 타입을 줍니다. AgensGraph는 표현식의 타입과 무관하게, 명시적으로 형 변환한 뒤에도 모든 Cypher 결과 컬럼을 jsonb로 반환합니다. 위에서처럼 SQL에서 jsonb 값을 벗겨 내십시오.

DISTINCT와 ORDER BY

`RETURN DISTINCT` 뒤의 `ORDER BY`는 투영에 나타나는 표현식으로만 정렬할 수 있습니다:

```
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
```

SQL의 `SELECT DISTINCT` 규칙과 같습니다. 정렬 키를 투영하거나, `DISTINCT`를 빼고 그룹화 집계로 중복을 없애십시오.

관련 항목

- **`WITH`** — 파이프라인 가운데에 있는, 같은 투영
- **`ORDER BY`**, **`SKIP`** · **`LIMIT`** — `RETURN` 뒤에 오는 절들
- **`FINISH`** — 행을 반환하지 않고 쿼리 끝내기
- **`UNION`** — 여러 쿼리의 결과 합치기

© 2026 SKAI Worldwide Co., Ltd.

- SQL과 Cypher 섞기 — 네이티브 타입으로 꺼내기

WITH

작업 행을 새 컬럼 집합으로 투영해 다음 절에 넘깁니다. 파이프라인 가운데에 있는 `RETURN` 입니다.

개요

```
WITH [ DISTINCT | ALL ] { * | expression [ AS alias ] } [, ...] [ WHERE condition ]
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`WITH` 는 파이프라인의 한 구역을 닫고 다음 구역을 엽니다. `RETURN` 과 정확히 같은 방식으로 행마다 표현식을 평가하고, 그 결과 컬럼을 뒤따르는 절에 넘깁니다. 세 가지 일에 이것이 필요합니다:

- **집계한 다음 걸러 내거나 더 일치시키기.** 집계 함수는 `WHERE` 에 올 수 없으므로, `WITH` 에서 집계하고 그 출력을 걸러 내십시오.
- **값을 한 번만 계산해** 뒤따르는 여러 절에서 쓰기.
- 쿼리의 비싼 부분이 실행되기 전에 `DISTINCT` · `ORDER BY` · `LIMIT` 으로 **파이프라인을 줄이기.**

이것이 투영이라는 점에서 따라오는 규칙들입니다:

- `WITH` 는 유효 범위를 **교체합니다.** 투영한 것만 살아남습니다. 빼놓은 변수는 사라지고, 그것을 뒤에서 언급하면 `variable does not exist` 가 나옵니다. 버리지 않고 더하려면 `WITH *, expr AS v` 로 쓰거나 `LET` 을 쓰십시오.
- 각 출력 이름은 **그 절 안에서 고유**해야 합니다. 같은 이름을 두 번 투영하면 `WITH passes on "..." more than once` 가 나옵니다. `RETURN` 에는 그런 규칙이 없습니다.
- 이 절의 `WHERE` 는 투영과 집계가 끝난 **뒤에** 실행되므로, 집계의 별칭을 검사할 수 있습니다.
- `ORDER BY` , `SKIP` / `OFFSET` , `LIMIT` 은 `WITH` 뒤에 오는 별개의 절이고, 따라서 그 `WHERE` 뒤에도 옵니다. 필터가 먼저 실행되고, 페이지 나누기는 살아남은 것에 적용됩니다.
- `WITH` 는 쿼리를 **시작**할 수도, 끝낼 수도 없습니다. 문장 맨 앞의 `WITH` 는 SQL의 공통 테이블 표현식이고, 그것은 이름과 괄호로 감싼 쿼리를 받는 다른 절입니다. `LET` 에는 그런 충돌이 없어서 쿼리를 시작할 수 있습니다.

매개변수

DISTINCT	투영한 행을 넘기기 전에 중복 행을 없앱니다.
ALL	중복 행을 그대로 둡니다. 기본값입니다.
*****	유효 범위의 이름 있는 모든 바인딩을 그대로 통과시킵니다. `WITH *, expr AS v` 처럼 다른 항목과 함께 쓰면 어떤 컬럼도 버리지 않고 컬럼을 더할 수 있습니다.
expression	유효 범위의 바인딩에 대한 Cypher 표현식이면 무엇이든 되고, 집계 함수도 됩니다.

© 2026 SKAI Worldwide Co., Ltd.

alias	다음 절이 쓰는 이름입니다. 변수 참조만 있는 것이 아니라면 필수입니다. 뒤따르는 절이 그것을 이름으로 부를 수 있어야 하기 때문입니다.
condition	투영된 행을 걸러 내는 불리언 표현식입니다. 투영 뒤에 평가되므로 이 절에서 정의한 별칭을 가리킬 수 있습니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

개수로 걸러 내기

```
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)`이 `p.name`으로 그룹화하고, `WHERE`가 그 결과인 `films`를 검사합니다. 같은 조건을 `MATCH ... WHERE`에 쓰면 `aggregate functions are not allowed in WHERE`로 거부됩니다.

다른 컬럼을 잃지 않고 컬럼 더하기

```
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 *`가 `p`와 `m`을 유효 범위에 남기므로 `RETURN`이 그 속성을 계속 읽을 수 있습니다. **`LET`**은 `*` 없이 같은 일을 합니다.

WITH가 무엇을 버리는지 보기

투영이 빼놓은 것은 그 뒤로 유효 범위 밖입니다:

© 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;
                                                               ^
```

파이프라인 도중에 중복 없애기

```
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)
```

더 일치시키기 전에 파이프라인 줄이기

`WITH` 뒤의 `ORDER BY`와 `LIMIT`은 다음 절이 다뤄야 하는 행 수를 줄입니다:

```
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)
```

가장 최근 영화 두 편만 둘째 `MATCH`에 도달합니다.

페이지 나누기가 필터 뒤에 오는 것 확인하기

`WHERE`는 `WITH`에 속하고, `ORDER BY`와 `LIMIT`은 그 뒤의 절입니다. 그래서 `LIMIT`은 필터를 통과한 행을 셉니다:

```
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)
```

필터를 빼면 같은 페이지 나누기가 완전히 다른 행을 고릅니다. 이제 제목순으로 앞의 세 행이 남기 때문입니다:

```
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)
```

같은 출력 이름은 거부된다

```
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...
                                                  ^
```

변수를 그대로 넘기면서 다른 것에 그 이름을 별칭으로 붙이는 것도 같은 식으로 충돌합니다. `WITH m, m.title AS m`은 `WITH passes on "m" more than once`를 냅니다.

참고

`WHERE`는 투영 목록 바로 뒤, `ORDER BY` · `SKIP` · `LIMIT` 앞에 써야 합니다. 그보다 뒤에 두면 구문 오류입니다. `WITH ... ORDER BY film WHERE genre = 'Action'`은 파싱되지 않습니다.

관련 항목

- **`RETURN`** — 쿼리를 끝내는, 같은 투영
- **`LET`** GQL — 어떤 바인딩도 버리지 않고 더하기
- **`WHERE`** — `WITH`가 지닐 수 있는 조건
- **`ORDER BY`**, **`SKIP`** · **`LIMIT`** — `WITH` 뒤에 오는 절들
- **`UNWIND`** — `collect()`를 되돌릴 때의 흔한 짝

LET

GQL 절 2.18에 추가

작업 중인 레코드에 계산된 값을 더하며, 이미 유효 범위에 있는 컬럼을 버리지 않습니다. Cypher의 `WITH *, expr AS v`에 대응하는 GQL 표준 절입니다.

© 2026 SKAI Worldwide Co., Ltd.

개요

```
LET variable = expression [, variable = expression ] ...
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`LET` 은 이미 유효 범위에 있는 변수에 대한 표현식으로 계산한 새 변수 하나 이상을 현재 행에 도입합니다. `WITH` 와 달리 **더하기만 합니다**. 유효 범위의 모든 바인딩이 자동으로 이어지므로, `LET` 이 파이프라인에서 변수를 실수로 떨어뜨리는 일은 결코 없습니다.

- 각 변수는 **새로 도입할 수만 있습니다.** 이미 유효 범위에 있는 이름은 다시 바인딩할 수 없습니다. 최상위에서와 마찬가지로, 서브쿼리 표현식의 본문이나 변수를 가져오는 `CALL { }` 의 본문 안에서도 그렇습니다.
- 표현식은 **행 단위로** 평가되며, 집계 함수는 허용되지 않습니다.
- `LET` 은 쿼리를 **시작**하거나 가운데에 놓일 수 있지만 쿼리를 끝낼 수는 없습니다. 문장을 시작할 때 쓰는 절이 이것입니다. 그 자리의 `WITH` 는 SQL의 공통 테이블 표현식이기 때문입니다.

매개변수

variable	새 바인딩의 이름입니다. 이미 유효 범위에 있어서는 안 됩니다. 뒤따르는 모든 절에서 보입니다.
expression	행 단위의 값 표현식이면 무엇이든 됩니다. 속성 접근, 산술, 함수 호출, `CASE` , 리스트·맵 생성자가 그렇습니다. 입력 행마다 한 번 평가됩니다.

예제

이 예제들은 **예제 데이터셋**에서 실행됩니다.

파생 값을 더하면서 유효 범위를 그대로 두기

GQL 형태와 Cypher 형태는 같습니다. 탭을 바꿔 비교해 보십시오:

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
 "Neill Blomkamp"   | 47
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Sam Worthington" | 50
(3 rows)
```

차이는 `p`에 무슨 일이 일어나는가입니다. `LET`은 그것을 자동으로 이어 줍니다. `WITH` 형태는 `*`로 그것을 달라고 해야 하고, `*`를 잊은 `WITH`는 `p`를 떨어뜨리므로 그다음 `RETURN`이 실패합니다.

계산된 값으로 쿼리를 시작하기

`LET`은 더하기만 하므로 문장을 시작할 수 있고, 유효 범위에 아직 아무것도 없는 상태에서 값을 심을 수 있습니다:

```
LET radius = 7, area = 3.14159 * 7 * 7
RETURN radius, area;
```

```
 radius |   area
--------+----------
 7      | 153.93791
(1 row)
```

도입만 된다

이미 유효 범위에 있는 이름은 다시 바인딩할 수 없습니다:

```
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 적합성

GQL의 LET 문장(ISO/IEC 39075:2024 §14.7)을 구현합니다.

- **지원** — 변수 정의, 여러 바인딩, 맨 앞 자리, 그리고 행 단위 표현식.
- **미지원** — 집계 표현식 위의 `LET`. 진행 상황은 **GQL 적합성 표**에서 확인하십시오.

참고

`LET`은 2.18에서 **예약 키워드**가 됐습니다. 이전 버전에서 `let`을 레이블 이름이나 변수 이름으로 썼다면 따옴표로 감싸거나(`"let"` — 백틱이 아니라 큰따옴표입니다) 이름을 바꾸십시오. 속성 키와 `RETURN` 별칭은 영향을 받지 않습니다.

관련 항목

- `WITH` — Cypher의 대응 절
- `NEXT` GQL
- `RETURN`
- GQL 적합성 표

ORDER BY

GQL 절

작업 행을 정렬합니다. 그 자체로 절이므로 끝에서만이 아니라 파이프라인의 어디서나 정렬할 수 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

개요

```
ORDER BY sort_key [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...]
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`ORDER BY` 는 지금 파이프라인에 있는 행을 정렬합니다. 자기가 정렬할 행을 내놓은 절 뒤에 쓰며, 그 절의 출력에 적용됩니다. 그 절은 `RETURN` , `WITH` , 또는 `MATCH` 입니다.

- 행은 첫 정렬 키로 비교되고, 같으면 다음 키로 넘어가며, 그렇게 이어집니다.
- `DESC` 를 쓰지 않으면 각 키는 오름차순으로 정렬됩니다. `ASC` 가 명시적인 기본값입니다.
- `NULLS FIRST` 나 `NULLS LAST` 로 달리 말하지 않으면, null은 `ASC` 에서 **맨 뒤**로, `DESC` 에서 **맨 앞**으로 갑니다.
- 정렬 키는 컬럼 번호가 아니라 표현식입니다. `ORDER BY 2` 는 상수 2로 정렬하므로 아무것도 정렬하지 않습니다. **호환성**을 보십시오.
- `ORDER BY` 는 쿼리를 **시작**할 수도, **끝낼** 수도 없습니다.
- `RETURN DISTINCT` 뒤에서는 모든 정렬 키가 투영에 나타나야 합니다.

홀로 서는 절이므로 파이프라인 가운데에서도 정렬할 수 있습니다. 뒤따르는 절이 처리해야 하는 행을 줄이려고 보통 `LIMIT` 이 뒤에 옵니다.

매개변수

sort_key	유효 범위의 바인딩에 대한 표현식입니다. `RETURN` 이나 `WITH` 뒤에서는 그 절의 별칭도 포함되며, 아예 투영되지 않은 표현식이어도 됩니다.
ASC, DESC	오름차순 또는 내림차순입니다. `ASC` 가 기본값입니다.
NULLS FIRST, NULLS LAST	null 값이 놓이는 자리입니다. 기본값은 오름차순 키에서 `NULLS LAST` , 내림차순 키에서 `NULLS FIRST` 입니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

투영한 컬럼으로 정렬하기

```
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.

투영하지 않은 것으로 정렬하기

정렬 키는 평범한 표현식이므로 결과 컬럼일 필요가 없습니다:

```
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)
```

여러 키로, 서로 다른 방향으로 정렬하기

```
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`는 오름차순이고, 각 장르 안에서 `year`는 내림차순입니다.

null의 자리 정하기

`OPTIONAL MATCH`는 null을 남기므로 기본 동작이 눈에 보입니다. 오름차순은 그것을 맨 뒤에 둡니다:

```
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)
```

내림차순은 그것을 맨 앞에 둡니다:

```
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`는 정렬 방향은 그대로 두고 null만 맨 앞에 둡니다:

```
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)
```

파이프라인 가운데에서 정렬하기

`MATCH` 뒤의 `ORDER BY`에 `LIMIT`을 붙이면 쿼리의 나머지가 보는 범위가 좁아집니다:

```
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)
```

둘째 `MATCH`는 열여덟 행이 아니라 두 행을 상대로 실행됩니다.

ORDER BY가 놓일 수 없는 자리

쿼리를 시작할 수 없습니다:

© 2026 SKAI Worldwide Co., Ltd.

```
ORDER BY 1;
```

```
ERROR:  syntax error at or near "ORDER"
LINE 1: ORDER BY 1;
        ^
```

그리고 쿼리를 닫을 수도 없습니다. 정렬은 그 자체로 결과를 내놓지 않기 때문입니다:

```
MATCH (m:movie) ORDER BY m.title;
```

```
ERROR:  Cypher query must end with RETURN, FINISH or update clause
```

참고

속성을 정렬하는 것은 jsonb 값을 정렬하는 것입니다. jsonb는 값보다 타입을 먼저 정렬합니다. 숫자는 모든 문자열보다 크고, 둘 다 null보다 큽니다(오름차순이면 null, 문자열, 숫자 순). 그래서 값의 타입이 섞인 키는 타입별로 먼저 묶입니다. 저장된 타입이 섞여 있는데 그 전체를 하나의 순서로 정렬하고 싶다면 그 키를 형 변환하십시오.

호환성

숫자는 컬럼 참조가 아니라 정렬 키다

SQL에서 `ORDER BY 2` 는 두 번째 출력 컬럼으로 정렬합니다. 여기서 정렬 키는 값 표현식이므로 그 정수는 상수입니다. 모든 행이 같다고 비교되고, 아무것도 재배열되지 않으며, 오류도 나지 않습니다.

```
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)
```

행이 연도순이 아니라 스캔 순서로 돌아옵니다. 이 페이지의 첫 예제가 보여 주듯 키를 이름으로 부르면 제대로 정렬됩니다. 이것은 ISO/IEC 39075 §16.17을 따르는 것이며, 거기서 `<sort key>` 는 순번 형태가 없는 `<aggregating value expression>` 입니다. 벗어나 있는 것은 SQL 쪽입니다. 정렬 키는 항상 이름으로 부르십시오.

ASCENDING과 DESCENDING (ISO/IEC 39075 §16.17)

표준은 `ASC` · `DESC` 와 함께 `ASCENDING` · `DESCENDING` 도 받습니다. AgensGraph는 짧은 표기만 받습니다. `ORDER BY film ASCENDING` 은 구문 오류입니다.

관련 항목

- **`SKIP`** · **`LIMIT`** — 정렬 뒤에 흔히 오는 페이지 나누기 절
- **`RETURN`** , **`WITH`** — 정렬이 적용되는 투영
- **`MATCH`** — 작업을 줄이려고 파이프라인 도중에 정렬하기

© 2026 SKAI Worldwide Co., Ltd.

SKIP과 LIMIT

GQL 절

작업 행을 페이지로 나눕니다. `SKIP` 은 앞쪽 행을 버리고, `LIMIT` 은 남길 행 수의 상한을 정합니다. `OFFSET` 은 `SKIP` 과 같은 뜻입니다.

개요

```
{ SKIP | OFFSET } count
```

```
LIMIT count
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`SKIP` 과 `LIMIT` 은 그 자체로 절이므로, 둘 중 어느 것이든 파이프라인의 어디서나 행을 페이지로 나눌 수 있습니다. `RETURN` 이나 `WITH` 뒤에 쓰거나 `MATCH` 바로 뒤에 쓰십시오. 둘 중 하나만 써도 됩니다.

- `SKIP` 과 `OFFSET` 은 같은 절입니다. 선호하는 표기를 쓰십시오. ISO/IEC 39075가 먼저 나열하는 것은 `OFFSET` 입니다.
- count는 리터럴만이 아니라 표현식이며, 한 번 평가됩니다.
- 음수인 개수는 오류입니다. 행 수보다 큰 개수는 오류가 아닙니다. 끝을 넘어선 `SKIP` 은 행을 내놓지 않고, 끝을 넘어선 `LIMIT` 은 전부를 내놓습니다.
- 소수인 개수는 가장 가까운 정수로 반올림됩니다.
- 두 절 모두 쿼리를 **시작**할 수 없고, 혼자서 쿼리를 끝낼 수도 없습니다. 쿼리는 여전히 `RETURN` , `FINISH` , 또는 쓰기 절로 끝나야 합니다.
- `ORDER BY` 가 없으면 어느 행을 받는지가 정해져 있지 않습니다. 페이지가 일정해야 한다면 언제든 먼저 정렬하십시오.
- 투영 뒤에 쓸 때는 수정자를 `ORDER BY` , `SKIP` / `OFFSET` , `LIMIT` 순서로 써야 합니다. **참고**를 보십시오.

매개변수

count	음수가 아닌 정수 표현식입니다. `SKIP` 은 앞에서 그만큼의 행을 버리고, `LIMIT` 은 최대 그만큼의 행을 남깁니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

앞의 몇 행만 취하기

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
LIMIT 2;
```


© 2026 SKAI Worldwide Co., Ltd.

```
      film
----------------
 "Avatar"
 "Blade Runner"
(2 rows)
```

가운데에서 한 페이지 취하기

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

```
        film
---------------------
 "Blade Runner 2049"
 "Elysium"
 "Gattaca"
(3 rows)
```

`OFFSET` 도 같게 읽힙니다:

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
OFFSET 2
LIMIT 3;
```

```
        film
---------------------
 "Blade Runner 2049"
 "Elysium"
 "Gattaca"
(3 rows)
```

개수를 계산하기

어떤 표현식이든 됩니다:

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
SKIP 1+1
LIMIT 2;
```

```
        film
---------------------
 "Blade Runner 2049"
 "Elysium"
(2 rows)
```

소수는 반올림됩니다. `LIMIT 1.5` 는 두 행을 남깁니다:

© 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)
```

끝을 넘어서 페이지 넘기기

```
MATCH (m:movie)
RETURN m.title AS film
ORDER BY film
SKIP 20;
```

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

영화가 18편이므로 남는 것이 없습니다. 같은 쿼리에 `LIMIT 100`을 쓰면 실패하는 것이 아니라 18편 전부를 반환합니다.

쿼리 도중에 파이프라인을 줄이기

`MATCH` 뒤의 `LIMIT`은 뒤따르는 절이 조인해야 하는 행 수를 줄입니다:

```
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)
```

음수인 개수는 거부된다

```
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`이 자신을 `OFFSET`으로 보고하는 것은 그 둘이 하나의 절이기 때문입니다.

놓일 수 없는 자리

두 절 모두 쿼리를 시작할 수 없습니다:

```
LIMIT 3;
```

```
ERROR:  syntax error at or near "LIMIT"
LINE 1: LIMIT 3;
        ^
```

그리고 둘 다 쿼리를 닫지 못합니다:

```
MATCH (m:movie) LIMIT 3;
```

```
ERROR:  Cypher query must end with RETURN, FINISH or update clause
```

참고

투영 뒤에서는 수정자를 `ORDER BY`, 그다음 `SKIP` / `OFFSET`, 그다음 `LIMIT` 순서로 쓰십시오. 그 순서만 투영 안으로 접혀 들어갑니다. `LIMIT`을 `SKIP`보다 앞에 두면 뒤에 남은 `SKIP`이 그 자체로 하나의 절로 읽히므로, 그 쿼리는 더 이상 투영으로 끝나지 않습니다. 오류가 보고하는 것이 그것입니다:

```
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
```

파이프라인 가운데에서는 두 순서 모두 파싱되고, 뜻이 서로 다릅니다. 각 절은 그 앞의 절이 내놓은 행에 적용됩니다. 먼저 건너뛰면 세 번째부터 여섯 번째 행을 취합니다:

```
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)
```

먼저 제한하면 네 행으로 잘라 낸 뒤 그중 둘을 건너뛰어 두 행이 남습니다:

```
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.

호환성

SKIP과 OFFSET (ISO/IEC 39075 §16.19)

표준의 `<offset synonym>`은 `OFFSET | SKIP`이므로 두 표기 모두 표준에 맞습니다. openCypher에는 `SKIP`만 있습니다.

관련 항목

- **`ORDER BY`** — 먼저 정렬해 페이지를 재현 가능하게 만들기
- **`RETURN`**, **`WITH`** — 이 절들이 페이지로 나누는 투영
- **`MATCH`** — 작업을 줄이려고 파이프라인 도중에 페이지 나누기

쓰기

CREATE

패턴의 요소마다 하나씩, 정점과 간선을 그래프에 추가합니다.

개요

CREATE pattern [, pattern] ...

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`CREATE`는 준 패턴을 그래프에 씁니다. **`MATCH`**가 읽을 때 쓰는 것과 같은 패턴 구문을 받지만, 모든 요소를 찾는 것이 아니라 만듭니다.

- `CREATE`는 **언제나 만듭니다.** 똑같은 정점이나 간선이 이미 있는지 확인하지 않습니다. 없는 것만 만들려면 **`MERGE`**를 쓰십시오.
- **이미 유효 범위에 있는 변수**를 이름으로 부른 요소는 다시 만들어지지 않고 재사용됩니다. `MATCH`가 찾은 정점에 간선을 붙이는 방식이 그것입니다. 양 끝점이 바인딩된 변수이므로 간선만 새로 만들어집니다.
- **새 변수를 이름으로 부르거나 이름이 없는** 요소는 만들어집니다. 그래서 하나의 패턴이 정점과 그 간선을 한 번에 만들 수 있습니다.
- 모든 요소에는 만들기에 충분한 정보가 있어야 합니다. 간선에는 레이블과 방향이 필요하고, 레이블 없는 정점은 그냥 `ag_vertex`가 됩니다.
- 속성 맵은 **맵 리터럴**, 쿼리 매개변수, 또는 jsonb 객체를 내놓는 표현식을 쓴 `= expression`이면 됩니다. 마지막 형태가 테이블 행 전체를 `properties`로 만드는 방법입니다. **`LOAD FROM`**을 참고하십시오.
- `CREATE`는 들어오는 행마다 한 번 실행됩니다. 앞에 아무 절도 없으면 한 번 실행됩니다.
- `CREATE`는 쿼리를 시작하거나 다른 절 뒤에 올 수 있고, 쿼리를 끝낼 수도 있습니다. 무엇을 만들었는지 보려면 **`RETURN`**을 붙이십시오.

`INSERT`가 완전히 같은 뜻이며, ISO/IEC 39075가 쓰는 표기입니다.

매개변수

pattern	**`MATCH`** 구문의 경로 패턴입니다. 쉼표로 구분해 여러 개 나열할 수 있고, 적은 순서대로 만들어지며 변수를 공유할 수 있습니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다. 마지막 예제가 이들을 되돌립니다.

© 2026 SKAI Worldwide Co., Ltd.

정점 하나 만들기

```
CREATE (:movie {title: 'Tenet', year: 2020, genre: 'Action'});
```

```
UPDATE 1
```

그래프 쓰기는 절이 무엇이었든 자기가 손댄 그래프 요소의 수를 `UPDATE` 태그로 보고합니다.

정점을 만들고 무엇을 만들었는지 보기

```
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)
```

이미 있는 두 정점 사이에 간선 만들기

양 끝점을 찾고, 그다음 간선만 만듭니다:

```
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` 와 `m` 은 이미 바인딩돼 있으므로 재사용됩니다. 만약 그 패턴이 `CREATE` 안에서 `(d:person {name: 'Christopher Nolan'})` 를 다시 적었다면 두 번째 Christopher Nolan이 만들어졌을 것입니다.

정점과 그 간선을 함께 만들기

패턴에서 아직 바인딩되지 않은 것은 무엇이든 만들어집니다:

```
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)
```

여러 패턴을 한 번에 만들기

쉼표로 구분한 패턴은 한 절에서 만들어지고 변수를 공유할 수 있습니다:

© 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
```

요소 세 개입니다. `acted_in` 간선 하나, `related_to` 간선 하나, 그리고 `Memento` 정점입니다. 이제 `Tenet`에는 간선이 네 개 있습니다:

```
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)
```

CREATE가 중복을 만드는 것 보기

`CREATE`에는 멱등한 구석이 없습니다:

```
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)
```

첫 문장을 다시 실행하면 네 개가 됩니다. 어떤 키가 고유해야 한다면 **`MERGE`** 나 고유성 제약을 쓰십시오.

예제 되돌리기

```
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;
```

```
UPDATE 2
UPDATE 6
UPDATE 2
```


© 2026 SKAI Worldwide Co., Ltd.

참고

정점의 id는 그 레이블 자신의 시퀀스에서 배정되므로, 새 정점이 받는 번호는 그 레이블에 앞서 무엇이 만들어졌는지에 달려 있습니다. `id(v)`로 읽으십시오. 특정 값에 의존하지 마십시오.

> 참고
>
> 레이블 없는 `CREATE` — `CREATE ()` — 는 문법에 맞고, 그래프의 기반 레이블인 `ag_vertex`의 정점을 만듭니다. 그런 정점은 어떤 `MATCH (n:label)`에도 보이지 않아 놓치기 쉽습니다. 레이블은 항상 주십시오.

호환성

절 이름 (ISO/IEC 39075 §13.2)

표준의 데이터 삽입 문장은 `CREATE`가 아니라 `INSERT`입니다. AgensGraph는 같은 절에 대해 두 표기를 모두 받습니다. **`INSERT`**를 참고하십시오. `CREATE GRAPH`나 `CREATE VLABEL` 같은 SQL의 `CREATE` 문장은 별개의 문장이며 영향을 받지 않습니다.

관련 항목

- **`INSERT`** GQL — 이 절의 표준 표기
- **`MERGE`** — 이미 있지 않은 것만 만들기
- **`MATCH`** — 패턴 구문, 그리고 끝점을 먼저 바인딩하는 방법
- **`SET`** — 만든 뒤 속성 바꾸기
- **`DELETE`** — 만든 것을 지우기
- **`LOAD FROM`** — 테이블의 행으로 만들기

INSERT

GQL 절 | 2.18에 추가

패턴이 기술하는 정점과 간선을 만듭니다. Cypher가 `CREATE`라고 부르는 절의 GQL 이름입니다.

개요

```
INSERT [ path_variable = ] pattern [, [ path_variable = ] pattern ] ...
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`INSERT`와 **`CREATE`**는 이름이 둘인 같은 절입니다. 파서가 둘에 대해 똑같은 트리를 만들므로, 같은 패턴을 받고, 같은 오류를 내고, 같은 실행 계획을 만듭니다.

- 패턴의 모든 요소가 만들어집니다. 이미 유효 범위의 변수에 바인딩된 요소만 예외입니다. 그 요소는 끝점으로 재사용되고 중복해서 만들어지지 않습니다.
- `INSERT`는 입력 행마다 한 번 실행됩니다. 앞에 절이 없으면 한 번 실행됩니다.
- 패턴이 바인딩하는 변수는 뒤따르는 절과, 같은 `INSERT` 안의 뒤쪽 패턴에서 보입니다.
- 모든 간선에는 레이블 하나와 방향이 정확히 있어야 합니다. 레이블을 암시적으로 만들도록 `enable_graph_ddl`을 켜 두지 않았다면, 레이블이 이미 존재해야 합니다.

© 2026 SKAI Worldwide Co., Ltd.

매개변수

path_variable	패턴이 만든 경로 전체에 바인딩되는 이름이며, 생략할 수 있습니다. 그것을 반환하면 새 정점과 간선을 순서대로 나열한 `graphpath` 값이 나옵니다.
pattern	구문에 설명된 정점 패턴 또는 경로 패턴입니다. 속성 맵은 그 자리에 (`{title: 'Tenet'}` 처럼) 적을 수도 있고, SQL 서브쿼리를 포함한 표현식에서 `=` 로 통째로 줄 수도 있습니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

영화 하나를 넣습니다. `INSERT` 와 `CREATE` 는 바꿔 쓸 수 있습니다. 탭을 바꿔 비교해 보십시오:

GQL — INSERT

```
INSERT (:movie {title: 'Tenet', year: 2020, genre: 'Action'});
```

Cypher — CREATE

```
CREATE (:movie {title: 'Tenet', year: 2020, genre: 'Action'});
```

```
UPDATE 1
```

두 형태는 실행 계획이 똑같습니다:

```
EXPLAIN (COSTS OFF) INSERT (:movie {title: 'Tenet'});
```

```
  QUERY PLAN
---------------
 Graph Create
   ->  Result
(2 rows)
```

만든 것을 반환하기

`INSERT` 도 다른 절과 마찬가지로 자기 변수를 바인딩하므로, 뒤따르는 `RETURN` 이 그것을 보고할 수 있습니다:

```
INSERT (p:person {name: 'John David Washington', born: 1984}) RETURN p.name AS person;
```

```
          person
-------------------------
 "John David Washington"
(1 row)
```

그래프에 이미 있는 것에 새 요소를 붙이기

앞선 `MATCH` 가 바인딩한 변수는 새 정점이 아니라 끝점이므로, 간선만 만들어집니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person {name: 'Christopher Nolan'}), (m:movie {title: 'Tenet'})
INSERT (p)-[:directed]->(m)
RETURN count(*) AS edges;
```

```
 edges
-------
 1
(1 row)
```

한 절에서 여러 패턴 만들기

패턴은 쉼표로 구분하며, 앞선 패턴이 바인딩한 변수를 가리킬 수 있습니다:

```
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
```

이 태그는 한 절이 만든 요소 네 개를 셉니다:

```
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)
```

SQL 쿼리에서 속성 맵 가져오기

`= expression` 형태는 맵 전체를 한 번에 줍니다. 테이블에서 속성을 적재할 때 쓰십시오:

```
INSERT (:movie = (SELECT jsonb_build_object('title', 'Dunkirk', 'year', 2017)));
```

```
UPDATE 1
```

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

```
 year
------
 2017
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

참고

이 절은 `CREATE` 태그가 아니라 평범한 그래프 쓰기 명령 태그인 `UPDATE n`을 보고합니다. 쓰기는 하고 투영은 하지 않으려면 `FINISH`와 함께 쓰십시오.

`INSERT`라고 썼어도 거부 메시지는 `CREATE`를 이름으로 부릅니다. 두 표기가 같은 절을 만들기 때문입니다. 알아 둘 만한 것들입니다:

패턴	오류
`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`
`m`이 이미 바인딩된 상태의 `INSERT (m)`	`there must be at least one relationship`

참고

`insert`는 2.18에서 예약 키워드가 됐습니다. 문법이 모호해지지 않는 자리에서는 따옴표 없이도 여전히 받아들여집니다. 변수 이름(`MATCH (insert:person)`)과 패턴 안의 레이블 이름(`MATCH (n:insert)`)이 그렇습니다. `CREATE VLABEL` 이름, `RETURN` 별칭을 가리키는 `ORDER BY` 키, SQL 테이블·컬럼 이름으로는 받아들여지지 않습니다. 그 자리에는 `"insert"`로 쓰거나 이름을 바꾸십시오.

호환성

GQL 적합성

§16.5의 insert graph pattern에 대한 GQL INSERT 문장(ISO/IEC 39075:2024 §13.2)을 구현합니다.

- **지원** — 정점과 간선 삽입, 문장 하나에 여러 패턴, 바인딩된 요소를 끝점으로 재사용하는 것, 인라인 속성 맵, 그리고 삽입한 패턴 전체에 대한 경로 변수.
- **미지원** — 그래프 타입을 요구하는 표준의 타입 그래프 검사(§18). AgensGraph의 그래프에는 타입이 없으므로, 삽입은 선언된 정점·간선 타입이 아니라 실제로 있는 레이블에 견주어 검증됩니다.

진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

openCypher는 이 절을 `CREATE`로 표기하며, AgensGraph는 그것을 같은 뜻으로 받습니다. `INSERT`라는 이름은 GQL과 SQL에서 왔고, SQL에서는 테이블에 대해 같은 일을 합니다.

© 2026 SKAI Worldwide Co., Ltd.

관련 항목

- `CREATE` — Cypher 쪽 같은 뜻의 표기, 같은 절
- `MERGE` — 이미 있지 않은 것만 만들기
- `FOR` GQL — 리스트 요소마다 `INSERT` 를 한 번씩 돌리기
- `FINISH` GQL — 투영 없이 쓰기
- 구문 — `INSERT` 가 받는 패턴 문법
- GQL 적합성 표

MERGE

패턴이 이미 있으면 그것을 일치시키고 없으면 만들며, 어느 쪽이든 그 변수를 바인딩합니다.

개요

```
MERGE pattern
  [ ON { CREATE | MATCH } SET set_item [, set_item ] ... ] ...
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`MERGE` 는 `MATCH` 와 `CREATE` 의 일을 하나의 절이 하는 것입니다. 자기 패턴을 찾고, 그 패턴이 있으면 `MATCH` 처럼 동작하고 없으면 `CREATE` 처럼 동작합니다. 어느 쪽이든 패턴의 변수는 결국 바인딩되므로, 쿼리의 나머지는 둘 중 무엇이 일어났는지 알 필요가 없습니다.

`MERGE` 에 관한 모든 것을 규칙 하나가 결정합니다. 이 절은 **패턴 전체**를 봅니다:

- 패턴 전체가 일치하면 아무것도 쓰이지 않습니다.
- 그중 어느 부분이라도 없으면 **패턴 전체가 만들어집니다.** 그래프의 다른 곳에 이미 있는 부분까지 포함해서입니다. `a` 와 `b` 는 있고 간선만 없는 상태에서 `(a)-[:e]->(b)` 를 `MERGE` 하면, 간선과 함께 두 번째 `a` 와 두 번째 `b` 가 만들어집니다.
- 그러므로 이미 무엇이 있는지 확실하지 않을 때는 **요소를 하나씩** `MERGE` 하십시오. 정점마다 `MERGE` 를 쓰고, 그다음 그것들이 바인딩한 변수 사이의 간선에 `MERGE` 를 쓰십시오. 그렇게 쓰면 `MERGE` 는 멱등합니다.

그 밖의 규칙입니다:

- `MERGE` 는 **패턴을 정확히 하나** 받습니다. 쉼표로 구분한 패턴은 구문 오류입니다. `MERGE` 절을 여러 개 쓰십시오.
- `ON CREATE SET` 은 패턴이 만들어졌을 때만, `ON MATCH SET` 은 일치했을 때만 실행됩니다. 둘 다 어느 순서로든 줄 수 있고, 각각 `SET` 의 항목 목록을 받습니다.
- 패턴 안의 속성도 일치 대상의 일부입니다. `MERGE (m:movie {title: 'Tenet'})` 는 `title` 이 정확히 `Tenet` 인 `movie` 에만 일치하며, `title` 을 갖고 있기만 한 것에는 일치하지 않습니다.
- `MERGE` 는 들어오는 행마다 한 번 실행됩니다. 쿼리를 시작하거나, 다른 절 뒤에 오거나, 쿼리를 끝낼 수 있습니다.

매개변수

pattern	`MATCH` 구문의 경로 패턴 하나입니다. 그 안의 모든 속성은 일치 조건이면서, 만들 때는 설정할 속성이기도 합니다.

© 2026 SKAI Worldwide Co., Ltd.

ON CREATE SET	패턴이 만들어졌을 때만 적용되는 할당입니다.
ON MATCH SET	패턴을 찾았을 때만 적용되는 할당입니다.
set_item	`SET` 구문의 할당입니다. `variable.property = expression`, `variable = map`, `variable += map` 중 하나입니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다. 마지막 예제가 이들을 되돌립니다.

정점을 만들거나 찾기

첫 실행은 만듭니다:

```
MERGE (m:movie {title: 'Tenet'}) RETURN m.title AS film, m.year AS year;
```

```
  film   | year
---------+------
 "Tenet" |
(1 row)
```

패턴에 `title`만 있었으므로 `year`는 설정되지 않았습니다. 두 번째 실행은 같은 정점을 찾고 아무것도 쓰지 않습니다:

```
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)
```

만들 때와 일치할 때 서로 다른 속성 설정하기

아무것도 없는 상태에서 시작합니다:

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
UPDATE 1
```


© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

똑같은 문장을 두 번째로 실행하면 다른 분기를 탑니다:

```
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` 는 만들기 분기가 준 값을 그대로 유지합니다. `ON MATCH SET` 이 그것에 손대지 않았기 때문입니다.

패턴 전체를 MERGE하면 정점이 중복되는 것 보기

Christopher Nolan은 한 명이고, 그는 `Tenet` 을 연출하지 않았습니다:

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
MATCH (p:person {name: 'Christopher Nolan'}) RETURN count(*) AS nolans;
```

```
UPDATE 1
 nolans
--------
 1
(1 row)
```

경로 전체를 `MERGE` 하면 간선이 없으므로 일치하는 것을 찾지 못합니다. 그래서 경로 전체를 만듭니다:

```
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)
```

```
MATCH (p:person {name: 'Christopher Nolan'}) RETURN count(*) AS nolans;
```

```
 nolans
--------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 2
(1 row)
```

`born` 속성도 없고 원본의 간선도 하나 없는 두 번째 Christopher Nolan입니다. `MERGE`가 불러들이는 실수가 이것입니다.

대신 요소를 하나씩 `MERGE` 하기

그것을 되돌리고, 그다음 요소를 하나씩 따로 `MERGE` 합니다:

```
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)
```

첫 `MERGE`는 이미 있는 사람을 일치시켰고, 둘째는 영화를 만들었고, 셋째는 간선만 만들었습니다. 같은 세 절을 다시 실행해도 아무것도 바뀌지 않습니다:

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
RETURN count(*) AS edges;
```

```
 edges
-------
 3
(1 row)
```

원래 데이터셋의 둘에 `MERGE`로 만든 하나를 더한 것이며, 그 문장을 몇 번 실행하든 그대로입니다.

패턴이 둘 이상이면 거부된다

```
MERGE (a:movie {title: 'A'}), (b:movie {title: 'B'});
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  syntax error at or near ","
LINE 1: MERGE (a:movie {title: 'A'}), (b:movie {title: 'B'});
                                    ^
```

`MERGE` 절을 두 개 쓰십시오.

예제 되돌리기

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
UPDATE 2
```

참고

고유성 제약은 중복 위험을 조용한 데이터 오염이 아니라 깔끔한 실패로 바꿔 줍니다. `person`의 `name`에 고유성이 걸려 있으면, 위의 패턴 전체 `MERGE`는 두 번째 Nolan을 만들 수 없습니다:

```
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").
```

아무것도 만들어지지 않고, 트랜잭션이 그 이유를 보고합니다. `MERGE`에 쓰는 키가 무엇이든 그 키에 제약을 선언하십시오.

호환성

MERGE는 GQL 절이 아니다

ISO/IEC 39075에는 `MERGE`가 없습니다. 표준의 데이터 변경 문장은 `<insert statement>`(§13.2), `<set statement>`(§13.3), `<remove statement>`(§13.4), `<delete statement>`(§13.5)입니다. `MERGE`는 Cypher에서 왔고 AgensGraph는 그것을 유지합니다. 표준만으로 같은 일을 하려면 **`CALL { subquery }`**를 쓰거나, `EXISTS` 검사 뒤에 **`INSERT`**를 쓰십시오.

관련 항목

- **`CREATE`** — 조건 없이 쓰기
- **`MATCH`** — 패턴 구문
- **`SET`** — `ON CREATE SET`과 `ON MATCH SET`이 받는 할당 구문
- **`DELETE`** — `MERGE`가 만든 것을 지우기

SET

GQL 절

쿼리가 바인딩한 정점과 간선의 속성을 바꿉니다.

© 2026 SKAI Worldwide Co., Ltd.

개요

```
SET { target = expression | target += expression } [, ...]
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`SET` 은 앞선 절이 이미 바인딩한 요소에 속성을 씁니다. 그 효과는 연산자 왼쪽에 무엇이 있는지, 그리고 어느 연산자를 쓰는지에 달려 있습니다:

적는 방식	효과
`n.key = expression`	속성 하나를 설정하고, 그 키가 없으면 추가합니다.
`n.key = NULL`	그 키를 지웁니다.
`n = {…}`	`properties` 전체를 그 맵으로 **교체합니다.** 맵에 없는 키는 사라집니다.
`n = other`	`n` 의 `properties` 를 `other` 의 것으로 교체합니다.
`n += {…}`	맵을 `properties` 에 **병합합니다.** 적은 키는 설정되고 나머지는 유지됩니다.

- 하나의 `SET` 안의 할당은 쉼표로 구분하며 왼쪽에서 오른쪽으로 적용됩니다.
- `SET` 은 들어오는 행마다 한 번 실행되므로, `MATCH` 가 정점 10개를 찾으면 그 10개가 모두 갱신됩니다.
- `SET` 에는 바인딩된 변수가 필요합니다. 변수를 도입할 수 없고, 쿼리를 시작할 수도 없습니다. 항상 `MATCH` , `MERGE` , 또는 무언가를 바인딩하는 다른 절 뒤에 옵니다.
- `SET` 은 쿼리를 끝낼 수 있습니다. 결과를 보려면 `RETURN` 을 붙이십시오.
- `SET` 은 속성만 씁니다. 레이블을 추가하거나 바꿀 수 없습니다.
- `MERGE` 의 `ON CREATE SET` 과 `ON MATCH SET` 도 이와 같은 할당 구문을 받습니다.

매개변수

target

속성 참조인 `variable.key` 이며 `variable.key.subkey` 처럼 중첩할 수도 있습니다. 또는 `properties` 전체를 교체하거나 병합할 때 쓰는 요소 변수 자체입니다.

expression

저장할 값입니다. Cypher 표현식이면 무엇이든 됩니다. 리터럴, 다른 요소의 속성, 산술 표현식이나 `CASE` 표현식, 리스트, 맵이 그렇습니다. `+=` 와, 변수 자체에 대한 `=` 의 경우에는 맵이거나 다른 그래프 요소여야 합니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다. 마지막 예제가 이들을 되돌립니다.

© 2026 SKAI Worldwide Co., Ltd.

속성 하나 설정하기

```
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()` 는 `properties` 전체를 반환하며, `SET` 이 무엇을 했는지 보는 가장 쉬운 방법입니다.

한 절에서 여러 속성 설정하기

```
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)
```

NULL로 속성 지우기

null을 할당하면 null이 저장되는 것이 아니라 그 키가 지워집니다:

```
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` 가 같은 것을 더 분명하게 표현합니다.

`+=` 로 맵 병합하기

`+=` 는 맵에 적힌 키를 설정하고 나머지는 그대로 둡니다:

```
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)
```

`=` 로 `properties` 전체 교체하기

변수 자체에 쓴 `=` 는 맵에 없는 모든 것을 버립니다. `genre` , `runtime` , `imax` 가 사라졌습니다:

```
MATCH (m:movie {title: 'Tenet'})
SET m = {title: 'Tenet', year: 2020}
RETURN properties(m) AS props;
```

```
              props
---------------------------------
 {"year": 2020, "title": "Tenet"}
(1 row)
```

이것이 두 연산자의 차이이고 조심해야 할 지점입니다. `=` 는 교체이고 `+=` 는 갱신입니다.

한 요소의 속성을 다른 요소로 복사하기

오른쪽에 다른 요소를 둘 수 있고, 그러면 그 `properties` 전체가 복사됩니다:

```
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)
```

이제 `name` 이 `Keanu Reeves` 인 정점이 둘입니다:

```
MATCH (p:person {name: 'Keanu Reeves'}) RETURN count(*) AS keanus;
```

```
 keanus
--------
 2
(1 row)
```

`properties` 를 복사하면 식별에 쓰이는 속성까지 복사됩니다. 간선은 복사되지 않고, 두 정점이 병합되지도 않습니다.

간선에 속성 설정하기

간선은 정점과 똑같이 속성을 받습니다:

```
MATCH (:person {name: 'Keanu Reeves'})-[r:acted_in]->(:movie {title: 'The Matrix'})
SET r.credited = true
RETURN properties(r) AS props;
```


© 2026 SKAI Worldwide Co., Ltd.

```
              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)
```

이 `MATCH`는 원래 Keanu Reeves에게 속한 간선 하나를 찾습니다. 위에서 만든 복사본에는 간선이 없기 때문입니다.

`properties` 비우기

빈 맵은 모든 속성을 지우고 요소는 그 자리에 남깁니다:

```
MATCH (m:movie {title: 'Tenet'})
SET m = {}
RETURN properties(m) AS props;
```

```
 props
-------
 {}
(1 row)
```

이제 그 정점에는 어떤 속성으로도 닿을 수 없습니다. 그래서 아래 정리 작업은 `Tenet`을 찾는 대신 `title`이 없는 것을 찾습니다.

예제 되돌리기

```
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
```

참고

기본 설정인 `allow_null_properties = off`는 `SET n.key = NULL`을 삭제로 만듭니다. 그것을 켜면 그 키가 JSON null 값과 함께 남습니다:

```
SET allow_null_properties = true;
CREATE (m:movie {title: 'Tenet', year: 2020})
SET m.runtime = NULL
RETURN properties(m) AS props;
```

```
SET
                      props
-----------------------------------------------------
```


© 2026 SKAI Worldwide Co., Ltd.

```
 {"year": 2020, "title": "Tenet", "runtime": null}
(1 row)
```

```
SET allow_null_properties = false;
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
SET
UPDATE 1
```

저장된 null은 없는 키와 같지 않습니다. `m.runtime IS NULL`은 어느 쪽이든 참이지만, 그 키는 `properties()`와 `jsonb` 키 검사에 나타납니다.

호환성

레이블 설정 (ISO/IEC 39075 §13.3)

표준의 `<set label item>`은 요소에 레이블을 추가하며, `SET n:blockbuster` 또는 `SET n IS blockbuster`로 씁니다. AgensGraph는 둘 다 받지 않습니다:

```
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...
                                                ^
```

요소의 레이블은 그 요소가 사는 테이블이므로, 요소를 만들 때 정해집니다. 정점을 다른 레이블로 옮기려면 그쪽에 새로 만들고 원본을 지우십시오.

관련 항목

- **`REMOVE`** — 이름으로 속성 지우기
- **`MERGE`** — `ON CREATE SET`과 `ON MATCH SET`
- **`MATCH`** — `SET`이 쓸 요소를 바인딩하기
- **`DELETE`** — 속성이 아니라 요소를 지우기

REMOVE

GQL 절

쿼리가 바인딩한 정점과 간선에서 속성을 지웁니다.

개요

> **REMOVE** property_reference [, property_reference] ...

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`REMOVE`는 속성 참조를 받아 그 키를 지웁니다. `SET n.key = NULL`을 평범하게 표기한 것이며, 하는 일도 정확히 같습니다.

- **없는** 키를 지우는 것은 오류가 아닙니다. 그 요소는 그대로 남습니다.

© 2026 SKAI Worldwide Co., Ltd.

- 참조를 쉼표로 구분해 여러 개 나열할 수 있습니다.
- 참조는 **중첩**될 수 있습니다. `n.key.subkey` 는 안쪽 키를 지우고 바깥 맵은 남깁니다.
- `REMOVE` 는 들어오는 행마다 한 번 실행되므로, `MATCH` 가 요소를 여럿 찾으면 그 전부에서 속성이 지워집니다. 이 절 자체에는 `WHERE` 가 없습니다. `MATCH` 에서 걸러 내십시오.
- `REMOVE` 에는 바인딩된 변수가 필요하며 쿼리를 시작할 수 없습니다. 끝낼 수는 있습니다. 결과를 보려면 `RETURN` 을 붙이십시오.
- `REMOVE` 는 속성만 지웁니다. 레이블을 지울 수 없고 요소를 지울 수도 없습니다. 요소를 지우는 것은 `DELETE` 입니다.

매개변수

property_reference	바인딩된 요소의 속성이며 `variable.key` 형태입니다. `variable.key.subkey` 처럼 중첩할 수도 있습니다. 요소 변수만 쓰는 것은 거부됩니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다. 마지막 예제가 이들을 되돌립니다.

속성 하나 지우기

```
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)
```

여러 개를 한 번에 지우기

```
MATCH (m:movie {title: 'Tenet'})
REMOVE m.imax, m.genre
RETURN properties(m) AS props;
```

```
             props
-----------------------------------
 {"year": 2020, "title": "Tenet"}
(1 row)
```

없는 키를 지우기

아무 일도 일어나지 않고, 아무 불평도 없습니다:

```
MATCH (m:movie {title: 'Tenet'})
REMOVE m.nosuchkey
RETURN properties(m) AS props;
```


© 2026 SKAI Worldwide Co., Ltd.

```
              props
-----------------------------------
 {"year": 2020, "title": "Tenet"}
(1 row)
```

중첩된 맵 안의 키 지우기

속성 자체가 맵일 수도 있습니다:

```
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)
```

점을 쓴 참조가 그 안으로 들어가고, 맵의 나머지는 그대로 둡니다:

```
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)
```

바깥 키를 이름으로 부르면 맵 전체가 지워집니다:

```
MATCH (m:movie {title: 'Tenet'})
REMOVE m.crew
RETURN properties(m) AS props;
```

```
              props
-----------------------------------
 {"year": 2020, "title": "Tenet"}
(1 row)
```

간선에서 속성 지우기

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

예제 되돌리기

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
UPDATE 1
```

참고

`REMOVE` 는 요소 변수만 쓰는 것을 받지 않습니다. 그런 식으로 전부 떼어 내라고 하면 거부되고, 대신 `SET` 형태를 가리키는 힌트가 나옵니다:

```
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 = {}` 은 `properties` 를 비우고 요소는 남깁니다. `DETACH DELETE m` 은 요소를 아예 지웁니다.

호환성

레이블 제거 (ISO/IEC 39075 §13.4)

표준의 `<remove label item>` 은 요소에서 레이블을 떼어 내며, `REMOVE n:blockbuster` 또는 `REMOVE n IS blockbuster` 로 씁니다. AgensGraph는 두 표기 모두 받지 않습니다:

```
MATCH (m:movie {title: 'Tenet'}) REMOVE m:movie RETURN m.title;
```

```
ERROR:  syntax error at or near ":"
LINE 1: MATCH (m:movie {title: 'Tenet'}) REMOVE m:movie RETURN m.tit...
                                                ^
```

레이블은 요소가 사는 테이블이므로 생성 시점에 정해집니다. 레이블을 추가하는 쪽의 같은 제한은 **`SET`** 을 참고하십시오.

관련 항목

- **`SET`** — `= NULL` 과 `= {}` 를 포함한 할당 형태
- **`DELETE`** — 속성이 아니라 요소를 지우기
- **`MATCH`** — 속성을 떼어 낼 요소를 바인딩하기

© 2026 SKAI Worldwide Co., Ltd.

DELETE

GQL 절

그래프에서 정점과 간선을 지웁니다. `DETACH DELETE`는 정점의 간선까지 함께 지웁니다.

개요

[**DETACH**] **DELETE** expression [, expression] ...

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`DELETE`는 자기 표현식이 가리키는 그래프 요소를 지웁니다. 그 요소가 먼저 바인딩돼 있어야 하므로, 항상 그것들을 찾아낸 절 뒤에 옵니다.

- **그냥 쓴 `DELETE`는 아직 간선이 남은 정점을 거부합니다.** 그것을 지우면 아무것도 가리키지 않는 간선이 남으므로, 이 절은 오류를 내고 트랜잭션이 롤백됩니다.
- **`DETACH DELETE`는** 정점의 간선을 먼저 지우고 그다음 정점을 지웁니다. 그 정점이 연결돼 있을 수 있다면 언제든 이것을 쓰십시오.
- **간선**을 지우는 데는 `DETACH`가 필요하지 않고, 양 끝점에는 손대지 않습니다.
- 표현식을 여러 개 나열할 수 있습니다. 적은 순서대로 지워지므로, 간선을 그 양 끝점보다 앞에 나열하면 `DETACH` 없이도 됩니다.
- `DELETE`는 들어오는 행마다 한 번 실행됩니다. 앞선 행이 이미 지운 요소는 오류가 아닙니다.
- 각 표현식은 정점, 간선, 또는 경로여야 합니다. 속성 참조는 받지 않습니다. 속성을 지우려면 **`REMOVE`**를 쓰십시오.
- `DELETE`는 쿼리를 끝낼 수 있습니다. 무엇이 사라졌는지 보려면 **`RETURN`**을 붙이십시오.

매개변수

DETACH	각 정점에 연결된 간선을 그 정점과 함께 지웁니다.
expression	정점, 간선, 또는 경로를 내놓는 표현식이며, 보통 **`MATCH`**가 바인딩한 변수입니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다. 첫 예제가 나머지 예제들이 지울 부분 그래프를 만들므로, 데이터셋은 시작할 때 모습으로 끝납니다.

지울 것을 만들기

```
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
```

이제 `Tenet`에는 간선이 두 개 있습니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

그냥 쓴 DELETE가 연결된 정점을 거부하는 것 보기

```
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.
```

아무것도 지워지지 않습니다. 그래프는 끝점이 사라진 간선을 결코 갖지 않습니다.

간선 지우기

간선에는 `DETACH`가 필요하지 않고, 그 양 끝점은 살아남습니다:

```
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
-------
 0
(1 row)
```

연결되지 않은 정점을 지우고 무엇이 사라졌는지 보기

간선이 남아 있지 않으면 그냥 쓴 `DELETE`도 성공합니다. 그 뒤의 `RETURN`이 지워진 요소를 보고합니다:

```
MATCH (p:person {name: 'Elizabeth Debicki'})
DELETE p
RETURN properties(p) AS removed;
```

```
                  removed
-------------------------------------------
 {"born": 1990, "name": "Elizabeth Debicki"}
(1 row)
```

정점과 그 간선을 함께 지우기

`Tenet`에는 아직 `directed` 간선이 있습니다. `DETACH DELETE`가 둘 다 가져갑니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie {title: 'Tenet'}) DETACH DELETE m;
```

```
UPDATE 2
```

요소 두 개, 간선과 정점입니다. Christopher Nolan은 손대지 않았고 그의 다른 간선도 그대로입니다:

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
RETURN m.title AS film
ORDER BY film;
```

```
      film
----------------
 "Inception"
 "Interstellar"
(2 rows)
```

한 절에서 여러 요소 지우기

간선을 그것이 잇는 정점들보다 앞에 나열하면 `DETACH`가 필요하지 않습니다. 정점이 지워질 시점에는 그 간선이 이미 사라져 있기 때문입니다:

```
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;
```

```
UPDATE 3
```

데이터셋이 원래 크기로 돌아왔습니다:

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

```
  label   | n
----------+----
 "movie"  | 18
 "person" | 28
(2 rows)
```

참고

`DELETE`는 속성이 아니라 요소를 받습니다. 속성을 이름으로 부르면 조용히 무시되는 것이 아니라 거부됩니다:

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


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  node, relationship, or path is expected
LINE 1: MATCH (m:movie {title: 'The Matrix'}) DELETE m.title;
                                                      ^
```

속성에는 `REMOVE`를 쓰거나 `SET m.title = NULL`을 쓰십시오.

> 참고
>
> 연결이 많은 정점에 `DETACH DELETE`를 쓰면 한 문장에서 대단히 많은 간선이 지워질 수 있고, 그 전부가 하나의 트랜잭션입니다. 영향 범위가 얼마나 큰지 확실하지 않을 때는 먼저 세어 보십시오 — `MATCH (v)-[r]-() RETURN count(r)`.

호환성

NODETACH (ISO/IEC 39075 §13.5)

표준의 `<delete statement>`는 `[ DETACH | NODETACH ] DELETE`이고, `NODETACH`가 암시적 기본값입니다. AgensGraph는 그 기본 동작은 구현했지만 키워드는 구현하지 않았습니다. `NODETACH DELETE m`은 구문 오류입니다. 같은 뜻인 그냥 `DELETE`를 쓰십시오.

관련 항목

- `REMOVE` — 요소가 아니라 속성을 지우기
- `MATCH` — 지울 요소를 바인딩하기
- `CREATE`, `MERGE` — 그것들을 만들어 넣은 절
- 데이터 정의 — 레이블이나 그래프 전체를 지우기

조합

UNWIND

리스트를 행으로 바꿉니다. 요소 하나마다 작업 행 하나입니다.

개요

UNWIND list_expression **AS** variable

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`UNWIND`는 리스트를 파이프라인으로 펼칩니다. 요소마다 행 하나를 내놓고 variable을 그 요소에 바인딩하며, 이미 유효 범위에 있던 나머지는 그대로 함께 이어집니다. `collect()`의 역이고, 리스트를 담은 한 행에서 그 조각들을 담은 여러 행으로 가는 방법입니다.

- `AS variable`은 **필수**입니다. 이름 없는 형태는 없습니다.
- 리스트는 **리터럴**, **리스트 값을 가진 속성**, `collect()`가 만든 변수, 또는 jsonb 배열이나 PostgreSQL 배열을 내놓는 표현식이면 됩니다. 스칼라는 거부됩니다.
- **빈 리스트**는 행을 내놓지 않으므로, 입력 행 전체가 사라집니다.
- 리스트 **안의 null**도 다른 요소와 같은 요소이고 행을 만듭니다.
- `UNWIND`는 들어오는 행마다 한 번 실행되고, 내놓는 행은 곱해집니다. 입력 행 셋에 요소 둘인 리스트면 여섯 행입니다.
- `UNWIND`는 쿼리를 시작하거나 다른 절 뒤에 올 수 있습니다. 쿼리를 끝낼 수는 없습니다.
- 리스트의 리스트를 펼치려면 `UNWIND` 절 두 개를 겹쳐 쓰십시오.

© 2026 SKAI Worldwide Co., Ltd.

`FOR`가 GQL의 대응 절이고 같은 일을 하며, 각 요소의 위치를 선택적으로 함께 줍니다.

매개변수

list_expression	jsonb 배열이나 PostgreSQL 배열을 내놓는 표현식입니다.
variable	각 요소에 바인딩되는 이름입니다. 이 절 뒤의 모든 절에서 보입니다.

예제

모든 예제는 예제 데이터셋에서 실행됩니다.

리스트 리터럴 펼치기

```
UNWIND [1, 91, 203, 1, 7, 91] AS n
RETURN n;
```

```
  n
-----
 1
 91
 203
 1
 7
 91
(6 rows)
```

리스트 순서대로 여섯 행이며, 중복도 포함됩니다.

리스트를 중복 제거하고 정렬하기

리스트를 제자리에서 `DISTINCT` 처리하는 방법은 없습니다. 그것을 펼치고, 행의 중복을 없애고, 다시 모으십시오:

```
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)
```

리스트로 MATCH를 돌리기

요소마다 행 하나이고, 각각 차례로 일치됩니다:

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Inception"  | 2010
(2 rows)
```

클라이언트가 준 키 목록에 대해 쿼리를 실행하는 흔한 방법이 이것이며, 그 목록은 매개변수 하나로 넘깁니다.

리스트 값을 가진 속성 펼치기

속성이 리스트를 담을 수 있고, `UNWIND`가 그것을 엽니다:

```
MATCH (m:movie {title: 'The Matrix'})
UNWIND m.embedding AS component
RETURN component;
```

```
  component
-------------
 -0.07594558
 0.04081754
 0.29592122
 -0.11921061
(4 rows)
```

collect 되돌리기

`collect()`는 행을 리스트로 접고, `UNWIND`는 그것을 다시 펼칩니다. 집계한 다음 행으로 돌아가는 방법이 이 짝입니다:

```
MATCH (p:person)-[:acted_in]->(m:movie)
WITH collect(DISTINCT m.genre) AS genres
UNWIND genres AS genre
RETURN genre
ORDER BY genre;
```

```
    genre
-------------
 "Action"
 "Adventure"
 "Sci-Fi"
(3 rows)
```

리스트의 리스트 펼치기

`UNWIND` 절 두 개이고, 두 번째는 첫 번째가 바인딩한 변수를 대상으로 합니다:

```
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)
```

빈 리스트 펼치기

행이 하나도 나오지 않고, 입력 행도 함께 사라집니다:

© 2026 SKAI Worldwide Co., Ltd.

```
UNWIND [] AS x RETURN x;
```

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

빈 리스트에서도 행이 살아남아야 한다면 **`OPTIONAL MATCH`** 나 `CASE` 보호 장치를 쓰십시오.

null을 남기기

null인 요소도 요소입니다:

```
UNWIND [1, NULL, 3] AS x RETURN x;
```

```
  x
------
 1
 null
 3
(3 rows)
```

값이 `null` 로 출력되는 것은 출력 컬럼이 jsonb이고, jsonb의 null은 없는 것이 아니라 하나의 값이기 때문입니다.

참고

`UNWIND` 에는 리스트가 필요합니다. 스칼라도, `NULL` 만 쓴 것도 요소 하나짜리나 요소 없는 리스트로 취급되지 않습니다:

```
UNWIND 3 AS x RETURN x;
```

```
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;
                ^
```

속성이 반드시 있다고 보장되지 않는다면, `UNWIND coalesce(m.tags, []) AS tag` 처럼 리스트가 아닐 수 있는 값을 감싸십시오.

호환성

UNWIND는 GQL 절이 아니다

ISO/IEC 39075에는 `UNWIND` 가 없습니다. 리스트를 펼치는 표준의 문장은 `<for statement>` (§14.8)이고, AgensGraph는 그것도 **`FOR`** 로 구현하고 있습니다. `FOR` 에는 각 요소에 0부터 시작하는 위치를 주는 `WITH OFFSET` 절이 있는데, `UNWIND` 는 그것을 할 수 없습니다.

관련 항목

- **`FOR`** `GQL` — 위치를 주는 표준 절
- **`WITH`** — `collect()` 가 놓이는 자리, 그리고 행의 중복을 없애는 방법

© 2026 SKAI Worldwide Co., Ltd.

- `MATCH` — 펼친 행으로 일치를 돌리기
- 함수 — `collect()` 와 그 밖의 리스트 함수들

FOR

GQL 절 | 2.18에 추가

리스트를 행으로 펼치며, 요소마다 행 하나를 만들고, 각 요소의 위치를 선택적으로 바인딩합니다. Cypher의 `UNWIND` 에 대응하는 GQL 절입니다.

개요

```
FOR variable IN list [ WITH OFFSET [ AS offset_variable ] ]
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`FOR` 는 각 입력 행을 리스트의 요소와 조인하므로, 요소 n개인 리스트를 만난 입력 행은 출력 행 n개가 됩니다. 리스트가 그 행에 의존한다면 펼치기가 행마다 실행됩니다.

- 요소 변수는 **새** 이름이어야 합니다. 유효 범위에 있는 이름을 다시 쓰면 `duplicate variable "<name>"` 이 나옵니다.
- `WITH OFFSET` 은 각 요소의 위치를 바인딩합니다. 그 위치는 **0부터** 셉니다.
- 빈 리스트, null인 리스트, 없는 속성은 모두 **행을 하나도** 내놓지 않으므로, 아무것도 없는 것에 대한 `FOR` 는 자기 입력 행을 떨어뜨립니다.
- 스칼라는 리스트가 아니므로 리스트로 감싸이지 않고 거부됩니다.
- `FOR` 는 쿼리를 시작하거나, 가운데에 놓이거나, 쓰기 절을 돌릴 수 있습니다. 쿼리를 끝낼 수는 없습니다.

매개변수

variable	출력 행마다 요소 하나에 바인딩되는 이름입니다. 이미 유효 범위에 있어서는 안 됩니다.
list	리스트를 내놓는 표현식이면 무엇이든 됩니다. 리터럴, `range()` , 리스트 값을 가진 속성, 앞선 `collect()` 가 만든 리스트, `+` 로 이어 붙인 두 리스트가 그렇습니다.
offset_variable	요소의 0부터 시작하는 위치에 바인딩되는 이름입니다. `AS` 절을 생략하면 그 위치가 `offset` 에 바인딩됩니다. 그것은 예약어이므로 그 뒤로는 어디서나 `"offset"` 으로 써야 합니다. `AS` 형태를 쓰는 편이 낫습니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

제목 리스트를 펼쳐 하나씩 찾아봅니다. `FOR` 와 Cypher의 `UNWIND` 는 같은 리스트를 펼칩니다. 탭을 바꿔 비교해 보십시오:

© 2026 SKAI Worldwide Co., Ltd.

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)
```

펼치면서 요소에 번호를 붙이기

`WITH OFFSET AS`가 그 위치에 이름을 붙입니다. 순위가 매겨진 제목 목록처럼 리스트의 순서가 곧 답일 때 필요한 것입니다:

```
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"
 2 | "Gattaca"
(3 rows)
```

첫 요소는 1이 아니라 0에 있습니다.

이름을 붙이지 않고 위치 읽기

`AS` 절을 빼도 위치는 바인딩되지만, 예약 키워드인 `offset`이라는 이름으로 바인딩됩니다. 그 뒤의 모든 참조는 따옴표로 감싸야 합니다:

```
FOR t IN ['The Matrix', 'Inception'] WITH OFFSET RETURN "offset", t;
```

```
 offset |      t
--------+--------------
 0      | "The Matrix"
 1      | "Inception"
(2 rows)
```

따옴표를 빼면 그 쿼리는 파싱되지 않습니다:

```
FOR t IN ['The Matrix'] WITH OFFSET RETURN offset, t;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  syntax error at or near "offset"
LINE 1: FOR t IN ['The Matrix'] WITH OFFSET RETURN offset, t;
                                                  ^
```

리스트 값을 가진 속성 펼치기

리스트는 일치한 행에서 올 수도 있고, 그 경우 펼치기가 행마다 실행됩니다:

```
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)
```

쿼리가 필요한 행을 만들어 내기

`range()` 도 다른 것과 같은 리스트이므로, `FOR` 가 생성된 키에 대한 조회를 돌릴 수 있습니다:

```
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)
```

참고

`FOR` 는 lateral `unnest()` 로 컴파일되고, `UNWIND` 는 `ProjectSet` 으로 컴파일됩니다. 둘은 뜻으로는 바꿔 쓸 수 있지만 같은 실행 계획은 아닙니다:

```
EXPLAIN (COSTS OFF) FOR x IN [1, 2, 3] RETURN x;
```

```
                QUERY PLAN
------------------------------------------
 Function Scan on unnest _agens_default_for
(1 row)
```

펼치기가 `unnest()` 를 지나므로, 스칼라 원본은 요소 하나짜리 리스트가 아니라 타입 오류입니다. 정수는 `function unnest(integer) does not exist` 를 내고, 스칼라 속성은 `cannot extract elements from a scalar` 를 냅니다. 반대로 null은 빈 리스트로 취급되어 행을 내놓지 않습니다.

참고

`FOR` 는 PostgreSQL에서 이미 예약 키워드이므로, 이 절이 식별자 이름에 새 제한을 더하지는 않습니다. `offset` 도 예약어이고, 그래서 `WITH OFFSET` 만 쓴 형태는 다시 읽는 자리에서 따옴표로 감싸야 합니다.

© 2026 SKAI Worldwide Co., Ltd.

호환성

GQL 적합성

GQL FOR 문장(ISO/IEC 39075:2024 §14.8), 기능 GQ10 “FOR statement: list value support”, 기능 GQ24 “FOR statement: WITH OFFSET”를 구현합니다.

- **지원** — 리스트 값 표현식의 전개, 입력 행마다의 평가, null 원본을 빈 리스트로 다루는 것, 그리고 표준이 규정하는 0부터 세는 위치(§14.8 일반 규칙 2의 `j-1`)를 주는 `WITH OFFSET`.
- **미지원** — 1부터 세는 짝인 `WITH ORDINALITY`(기능 GQ11). 대신 오프셋에 1을 더하십시오. 원본 자리에 `<binding table reference value expression>`을 쓰는 것(기능 GQ23)도 지원하지 않습니다. AgensGraph에는 이름 붙은 바인딩 테이블이 없기 때문입니다.
- **이탈** — 표준은 위치 변수를 `AS` 없이 씁니다(`WITH OFFSET i`). AgensGraph는 `WITH OFFSET AS i`를 요구하며, 그냥 쓴 `WITH OFFSET`은 이름을 `offset`으로 정합니다.

진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

openCypher는 이 절을 `UNWIND … AS x`로 표기하며, AgensGraph는 그것을 그대로 받습니다. 그 표기에는 위치를 주는 형태가 없습니다.

관련 항목

- `UNWIND` — 위치가 없는 Cypher의 대응 절
- `INSERT` GQL — `FOR`가 돌릴 수 있는 쓰기
- `LET` GQL
- `FILTER` GQL — 펼친 행을 좁히기
- `ORDER BY` — `FOR`가 내놓은 행을 정렬하기
- GQL 적합성 표

FILTER

GQL 절 2.18에 추가

조건을 만족하는 행만 남깁니다. Cypher의 `WITH … WHERE`와 달리 투영이 필요하지 않은 독립적인 필터입니다.

개요

FILTER [**WHERE**] condition

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`FILTER`는 조건이 참이 아닌 행을 모두 버리고, 그 밖의 어떤 것도 바꾸지 않습니다. 그 앞에서 유효 범위에 있던 컬럼이 그 뒤에서도 정확히 그대로입니다. 생략할 수 있는 `WHERE`는 군더더기 단어입니다. `FILTER x > 1`과 `FILTER WHERE x > 1`은 같은 절로 파싱됩니다.

- `FILTER`는 **가운데** 절입니다. 쿼리를 시작할 수도, 끝낼 수도 없습니다.
- 앞의 절이 내놓은 것이 무엇이든 그것에 적용되며, 집계하는 `WITH`의 출력도 포함합니다. 그래서 `FILTER`는 Cypher 파이프라인에서 `HAVING` 역할을 합니다.

© 2026 SKAI Worldwide Co., Ltd.

- `FILTER`를 이어 쓰면 순서대로 적용되며, 각각 앞의 것이 남긴 행을 더 좁힙니다.
- 조건이 null로 평가되는 행은 SQL에서와 마찬가지로 버려집니다.

매개변수

condition	유효 범위의 변수에 대한 조건입니다. `WHERE`가 받는 불리언 표현식이면 무엇이든 되고, `EXISTS { }`나 `COUNT { }` 같은 **서브쿼리 값 표현식**도 됩니다. 조건은 불리언이어야 합니다. 숫자만 쓰면 `argument of FILTER must be type boolean, not type integer`로 거부되고, 속성만 쓰면 `cannot cast 2003 (jsonb type numeric) to boolean`으로 거부됩니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

2014년부터의 영화만 남깁니다. 패턴 뒤의 `FILTER`와 거기 붙은 Cypher의 `WHERE`는 같은 검사입니다. 탭을 바꿔 비교해 보십시오:

GQL — FILTER

```
MATCH (m:movie)
FILTER m.year >= 2014
RETURN m.title AS film, m.year AS year ORDER BY year;
```

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)
```

두 형태는 실행 계획이 똑같습니다. `movie` 스캔에 붙는 `Filter: (properties.'year' >= '2014'::jsonb)`까지 같습니다.

집계를 걸러 내기

`WHERE`는 집계하는 `RETURN` 뒤에 올 수 없고, `WITH`에 붙이면 투영을 다시 적어야 합니다. `FILTER`는 그 집계를 곧바로 검사합니다:

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

```
 "Keanu Reeves"          | 4
(2 rows)
```

두 단계로 행을 좁히기

각 `FILTER`는 그 앞의 것이 남긴 행만 봅니다:

```
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)
```

쓰기 절이 내놓은 행을 걸러 내기

쓰기 뒤의 `FILTER`는 쓰기를 먼저 실행하고, 그다음 흘러가는 행을 제한합니다. 아래에서는 `SET`이 그 영화에 손을 대고 `FILTER`가 그 행을 보고할지 결정합니다:

```
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)
```

참고

`FILTER`는 쿼리를 시작할 수 없으므로, 그것으로 시작하는 문장은 절 배치 오류가 아니라 구문 오류입니다:

```
FILTER true;
```

```
ERROR:  syntax error at or near "FILTER"
LINE 1: FILTER true;
        ^
```

`FILTER`가 맨 앞에 오는 곳이 하나 있는데 **`NEXT`** 뒤이며, 거기서는 경계를 넘어 전달된 테이블에 대해 동작합니다. `FILTER`로 쿼리를 끝내면 `Cypher query must end with RETURN, FINISH or update clause`가 나옵니다.

`filter`는 예약되지 않은 키워드이므로, 변수·레이블·속성 이름으로 계속 쓸 수 있습니다.

호환성

GQL 적합성

GQL의 FILTER 문장(ISO/IEC 39075:2024 §14.6), 기능 GQ08 “FILTER statement”를 구현합니다.

© 2026 SKAI Worldwide Co., Ltd.

- **지원** — 표준이 정의하는 두 표기 `FILTER <search condition>`과 `FILTER WHERE <search condition>`을, 맨 앞과 맨 끝이 아닌 모든 자리에서.
- **미지원** — 이 하위 절에서 빠진 것은 없습니다. `FILTER`는 완전합니다.

진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

openCypher에는 `FILTER` 문장이 없습니다. 대응하는 것은 `MATCH`나 `WITH`에 붙인 `WHERE`이고, AgensGraph는 그 둘을 그대로 받습니다. 이와 무관한 openCypher의 리스트 함수 `filter(x IN list WHERE predicate)`도 받지 않습니다. 괄호 안의 `WHERE`가 구문 오류입니다. 대신 리스트 컴프리헨션을 쓰십시오: `[x IN m.embedding WHERE x > 0]`.

관련 항목

- `WHERE` — 절에 붙여 쓰는 Cypher의 대응 절
- `WITH` — 한 절에서 투영과 필터를 함께
- `LET` GQL — 어떤 컬럼도 버리지 않고 컬럼을 더하기
- `NEXT` GQL — `FILTER`가 맨 앞에 올 수 있는 유일한 자리
- **서브쿼리 값 표현식** GQL — `FILTER`가 쓸 수 있는 조건들
- GQL 적합성 표

NEXT

GQL 절 2.18에 추가

두 쿼리를 끝과 끝으로 잇습니다. 첫 쿼리가 반환한 행이 둘째 쿼리의 입력 행이 됩니다.

개요

```
query NEXT query [ NEXT query ] ...
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`NEXT`는 왼쪽 쿼리의 결과 표를 오른쪽 쿼리를 돌리는 표로 만듭니다. 하나의 쿼리 안에서 `WITH`가 하는 조합을, 온전한 두 쿼리 사이의 경계로 올려놓은 것입니다.

- `NEXT` 왼쪽의 쿼리는 `RETURN`으로 끝나야 합니다.
- 이 경계는 **유효 범위를 초기화합니다.** 오른쪽에서는 `RETURN`이 내놓은 컬럼만 보입니다. 그 밖의 변수는 모두 사라집니다.
- 컬럼은 자기 타입을 그대로 지니고 경계를 넘습니다. 정점은 여전히 정점이고, 간선은 간선이고, 경로는 경로입니다.
- `NEXT`는 왼쪽 결합이므로 `A NEXT B NEXT C`는 `(A NEXT B) NEXT C`로 조합됩니다.
- `UNION`, `INTERSECT`, `EXCEPT` 복합 쿼리가 어느 쪽에든 올 수 있습니다.
- `NEXT`는 문장의 맨 앞에서, 그리고 `EXPLAIN`과 `PREPARE` 안에서 받아들여집니다. 서브쿼리 안에서는 받아들여지지 않으며, `CALL { }` 본문과 서브쿼리 값 표현식 둘 다 그렇습니다.

© 2026 SKAI Worldwide Co., Ltd.

매개변수

query	온전한 쿼리입니다. `NEXT` 왼쪽의 쿼리는 `RETURN` 으로 끝나야 하고, 반환하는 모든 컬럼에 이름이 있어야 합니다. 별칭이 없는 표현식은 `expression in WITH must be aliased (use AS)` 를 냅니다. 오른쪽 쿼리는 새 쿼리입니다. 쿼리를 시작할 수 있는 절로 시작합니다(`MATCH` , `OPTIONAL MATCH` , `CREATE` , `INSERT` , `MERGE` , `LOAD FROM` , `UNWIND` , `FOR` , `CALL` , `LET` , `RETURN` , `FINISH`). 전달된 표를 대상으로 동작하는 절로 시작할 수도 있습니다(`WITH` , `FILTER` , `ORDER BY` , `SKIP` , `OFFSET` , `LIMIT`). 끝맺는 방식은 다른 쿼리와 같습니다. `RETURN` , `FINISH` , 또는 쓰기 절입니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

한 감독이 만든 영화를 그 출연진을 찾는 둘째 쿼리로 넘깁니다. `NEXT` 와 중간의 `WITH` 는 같은 두 단계를 조합합니다. 탭을 바꿔 비교해 보십시오:

GQL — NEXT

```
MATCH (p:person {name: 'Christopher Nolan'})-[:directed]->(m:movie)
RETURN m
NEXT
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` 은 정점으로 경계를 넘으므로, 둘째 쿼리는 그것에서 다시 순회합니다.

첫 쿼리가 계산한 집계를 걸러 내기

`FILTER` 는 독립적인 쿼리를 시작할 수 없지만, `NEXT` 뒤의 쿼리는 시작할 수 있습니다. 그러면 `WITH` 를 더 쓰지 않고도 집계에 `HAVING` 식의 필터를 줄 수 있습니다:

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

```
          actor          | films
-------------------------+-------
 "Arnold Schwarzenegger" | 3
 "Keanu Reeves"          | 4
(2 rows)
```

쿼리 세 개 잇기

`NEXT` 하나하나가 또 하나의 경계이며, 왼쪽에서 오른쪽으로 평가됩니다:

```
MATCH (m:movie) RETURN m.genre AS genre
NEXT
RETURN DISTINCT genre AS g
NEXT
RETURN count(g) AS genres;
```

```
 genres
--------
 3
(1 row)
```

첫 쿼리가 반환하지 않은 변수 참조하기

경계를 넘어 살아남는 것은 반환된 컬럼뿐이므로, 그 밖의 것을 가리키면 파싱 시점에 실패합니다:

```
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;
               ^
```

참고

`NEXT` 는 실행 시점에 비용이 없습니다. 이 경계는 왼쪽 쿼리의 `RETURN` 을, 절 파이프라인이 이미 절 묶음 사이에 놓는 `WITH` 로 다시 쓰는 방식으로 만들어집니다. 그래서 위 탭의 두 쿼리는 실행 계획이 똑같습니다.

두 개의 안전장치가 잘못된 사용을 구문 오류가 아니라 곧바로 보고합니다. `RETURN` 으로 끝나지 않는 왼쪽 쿼리는 `a query statement before NEXT must end with RETURN` 을 냅니다. `FINISH` 로 끝나는 왼쪽 쿼리는 `FINISH cannot be followed by NEXT` 를 냅니다.

이어진 쿼리 중 마지막 쿼리도 문장을 제대로 끝맺어야 합니다. `ORDER BY` , `SKIP` , `OFFSET` , `LIMIT` 은 그 자체로 절이지만 어느 것도 쿼리를 끝낼 수 없습니다. 그런 절에서 멈춘 사슬은 `Cypher query must end with RETURN, FINISH or update clause` 를 냅니다. 뒤에 `RETURN` 이나 `FINISH` 를 붙이십시오.

`next` 는 예약되지 않은 키워드이므로, 변수·레이블·속성 이름으로 계속 쓸 수 있습니다.

호환성

GQL 적합성

프로시저 본문(ISO/IEC 39075:2024 §9.2)의 GQL `<next statement>` , 기능 GQ20 “Advanced linear composition with NEXT”를 구현합니다.

© 2026 SKAI Worldwide Co., Ltd.

- **지원** — 쿼리 문장 잇기, 왼쪽 결합의 여러 겹 잇기, 경계에서의 유효 범위 초기화, 그리고 양쪽 어디에나 오는 집합 연산 합성.
- **미지원** — 표준이 `NEXT`와 다음 문장 사이에 허용하는 선택적 `YIELD` 절 (`NEXT YIELD a, b …`). 구문 오류가 납니다. 대신 앞의 `RETURN`에서 투영하고 이름을 바꾸십시오.

진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

openCypher에는 `NEXT`가 없습니다. 거기서 조합은 언제나 위 탭에 보인 `WITH` 형태이고, AgensGraph는 그것을 그대로 받습니다.

관련 항목

- `WITH` — 하나의 쿼리 안에서 쓰는 Cypher의 대응 절
- `LET` GQL — 경계 없이 컬럼을 더하기
- `FILTER` GQL — `NEXT`가 맨 앞에 올 수 있게 해 주는 절
- `FINISH` GQL
- `RETURN` — 왼쪽 쿼리가 끝나야 하는 방식
- GQL 적합성 표

FINISH

GQL 절 2.18에 추가

아무것도 투영하지 않고 쿼리를 끝냅니다. 작업은 실행되고, 쿼리는 빈 결과를 반환합니다.

개요

```
FINISH
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

모든 쿼리는 `RETURN` , `FINISH` , 또는 쓰기 절로 끝나야 합니다. `FINISH`는 "결과를 내놓지 않는다"고 말하는 끝맺음입니다. 인자를 받지 않고, 컬럼을 투영하지 않으며, 문장을 닫습니다.

- `FINISH`는 **마지막** 절이어야 합니다. 그 뒤에 무엇이 오면 `FINISH must be at the end of the query`가 나옵니다.
- 읽기 뒤의 `FINISH`는 행을 버립니다. 플래너가 그것을 `LIMIT 0`으로 적용하므로, 스캔은 열리지만 행은 하나도 가져오지 않습니다.
- 쓰기 뒤에서도 쓰기는 그대로 일어납니다. `FINISH`는 투영만 억제합니다.
- `FINISH`로 끝나는 쿼리는 **빈 결과**를 반환합니다. 컬럼도 행도 없습니다. `psql`은 그것을 `--` 머리와 그 뒤의 `(0 rows)`로 출력하고, `\gdesc`는 `The command has no result, or the result has no columns.`라고 보고합니다.
- `FINISH`로 끝나는 쓰기는 결과 대신 자기 명령 태그를 보고합니다. 정점 하나를 만들었다면 `UPDATE 1` 같은 것입니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

실행 계획만 보려고 읽기를 실행하는 경우입니다. 이것을 표기하는 Cypher 방식은 없습니다. `RETURN`도 쓰기 절도 없는 Cypher 쿼리는 거부됩니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie) FINISH;
```

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

결과를 다시 읽지 않고 쓰기

만들어진 요소를 클라이언트로 보내고 싶지 않을 때 꺼내 쓸 형태가 `CREATE … FINISH` 입니다:

```
CREATE (:movie {title: 'Tenet', year: 2020, genre: 'Action'}) FINISH;
```

```
UPDATE 1
```

그 정점은 만들어져 있습니다:

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

```
  film   | year
---------+------
 "Tenet" | 2020
(1 row)
```

FINISH 뒤의 읽기가 비용을 들이지 않음을 확인하기

실행 계획에 이 절이 컴파일된 `LIMIT 0` 이 보이고, 그 아래에는 한 번도 값을 가져가지 않는 스캔이 있습니다:

```
EXPLAIN (COSTS OFF) MATCH (m:movie) FINISH;
```

```
        QUERY PLAN
---------------------------
 Limit
   ->  Seq Scan on movie m
(2 rows)
```

쓰기 계획은 쓰기 노드를 유지합니다. 그 쓰기는 선택이 아니기 때문입니다:

```
EXPLAIN (COSTS OFF) CREATE (:movie {title: 'Tenet'}) FINISH;
```

```
  QUERY PLAN
--------------
 Graph Create
   ->  Result
(2 rows)
```

아무것도 투영하지 않는, 정렬된 쿼리 끝맺기

`ORDER BY` , `SKIP` , `OFFSET` , `LIMIT` 은 그 자체로 절이지만 어느 것도 쿼리를 끝낼 수 없습니다. `FINISH` 가 그런 절로 끝나는 파이프라인을 닫아 줍니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie) ORDER BY m.year LIMIT 3 FINISH;
```

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

`FINISH`가 없으면 같은 쿼리가 `Cypher query must end with RETURN, FINISH or update clause`를 냅니다.

참고

결과가 필요한 자리에서는 `FINISH`가 받아들여지지 않습니다. `CALL { }` 본문은 무언가를 투영해야 하므로, `FINISH`로 끝나는 본문은 `CALL subquery must RETURN at least one variable`을 냅니다. `FINISH`로 끝나는 쿼리는 **`NEXT`**의 왼쪽 피연산자가 될 수도 없습니다. 그 경우 `FINISH cannot be followed by NEXT`가 나옵니다.

> **참고**
>
> `finish`는 2.18에서 예약 키워드가 됐습니다. 패턴 안의 레이블 이름으로는 따옴표 없이도 여전히 받아들여지며(`MATCH (n:finish)`), 업그레이드한 데이터베이스에 필요한 것이 그것입니다. 변수 이름, `CREATE VLABEL` 이름, SQL 식별자로는 받아들여지지 않습니다. 그 자리에는 `"finish"`로 쓰거나 이름을 바꾸십시오. 속성 키와 `RETURN` 별칭은 영향을 받지 않습니다.

호환성

> **GQL 적합성**
>
> GQL 기본 결과 문장(ISO/IEC 39075:2024 §14.10)의 `FINISH`를 구현합니다. 표준은 `FINISH`가 "빈 타입"을 가지며 실행 결과를 "결과가 생략된 성공"으로 설정한다고 규정하는데, 위의 컬럼 없는 빈 결과가 그것을 나타냅니다.
>
> - **지원** — 읽기 쿼리와 데이터를 바꾸는 쿼리의 마지막 문장으로 쓰는 `FINISH`. 후자의 경우 표준은 그것을 어차피 암시적인 것으로 봅니다(§13.1).
> - **미지원** — 집합 연산의 한 갈래로 쓰는 `FINISH`. 표준도 그것을 금지합니다(§14.2, 구문 규칙 5). AgensGraph는 컬럼 수 불일치로 거부합니다: `each UNION query must have the same number of columns`.
>
> 진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

openCypher에는 `FINISH`가 없습니다. 거기서는 읽기 쿼리가 `RETURN`으로 끝나야 하고, 쓰기 쿼리는 자기 쓰기 절로 끝날 수 있습니다. AgensGraph는 두 번째 형태도 받으므로, `FINISH`는 무언가를 대체하는 것이 아니라 읽기 경우를 더해 줍니다.

관련 항목

- **`RETURN`** — 쿼리를 끝내는 또 다른 방법
- **`ORDER BY`**와 **`SKIP`** · **`LIMIT`** — 끝맺는 절이 필요한 절들
- **`NEXT`** GQL — `FINISH`가 앞설 수 없는 유일한 절
- **`INSERT`** GQL — `FINISH`로 끝낼 만한 쓰기
- GQL 적합성 표

UNION

GQL 절

두 개 이상의 쿼리 결과를 하나로 합칩니다. `INTERSECT`와 `EXCEPT`는 다른 두 집합 연산을 쓴 같은 구문입니다.

© 2026 SKAI Worldwide Co., Ltd.

개요

```
query { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] query [ ... ]
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

집합 연산자는 각각 자기 `RETURN`을 가진 온전한 두 쿼리를 하나의 결과로 잇습니다:

연산자	결과
`UNION`	두 쿼리 중 어느 쪽에든 있는 행.
`INTERSECT`	두 쿼리 모두에 있는 행.
`EXCEPT`	첫 쿼리에는 있고 둘째 쿼리에는 없는 행.

셋 모두에 적용되는 규칙입니다:

- 모든 쿼리가 **같은 수의 컬럼**을 반환해야 하고, 서로 대응하는 컬럼은 비교할 수 있어야 합니다. 이름은 같지 않아도 됩니다. 결과는 첫 쿼리의 이름을 가져갑니다.
- `DISTINCT`가 기본값입니다. 중복 행은 없어집니다. `ALL`은 중복을 남깁니다. `UNION`과 `UNION DISTINCT`는 같은 뜻입니다.
- 연산자는 **이어 쓸 수 있고**, 이어진 것은 왼쪽에서 오른쪽으로 평가됩니다.
- 행 순서는 정해져 있지 않습니다. 집합 연산 뒤에 쓴 `ORDER BY`는 합쳐진 결과를 정렬하지 **않습니다.** 무엇이 정렬하는지는 **참고**를 보십시오.
- 각 쿼리는 `RETURN`으로 끝나는 온전한 쿼리여야 합니다. 한쪽 갈래가 쓰기 절이나 `MATCH`만으로 끝날 수는 없습니다.

매개변수

query	`RETURN`으로 끝나는 온전한 쿼리이며, 이어진 다른 모든 쿼리와 컬럼 수가 같아야 합니다.
ALL	중복 행을 남깁니다.
DISTINCT	중복 행을 없앱니다. 이것이 기본값입니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

두 결과 집합 합치기

`Inception`에 연결된 모든 사람이며, 연결 방식이 무엇이든 포함합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

중복을 남기거나 버리기

여기서 두 갈래 모두 영화마다 하나씩, 같은 장르 값 18개를 반환합니다. `UNION`이 그것을 셋으로 줄입니다:

```
MATCH (m:movie) RETURN m.genre AS genre
UNION
MATCH (m:movie) RETURN m.genre AS genre;
```

```
    genre
-------------
 "Action"
 "Sci-Fi"
 "Adventure"
(3 rows)
```

`UNION DISTINCT`는 같은 문장을 온전히 적은 것입니다. `UNION ALL`은 두 갈래의 모든 행을 남깁니다:

```
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)
```

그 두 행은 어차피 서로 다르므로 `ALL`은 비용에만 드러납니다. 중복 제거를 건너뜁니다.

두 쿼리 모두에 있는 행 취하기

```
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)
```

한 쿼리에서 다른 쿼리를 빼기

무언가를 연출했지만 연기한 적은 없는 사람들입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

행은 특정한 순서 없이 돌아옵니다.

합쳐진 결과를 정렬하고 페이지로 나누기

집합 연산을 `NEXT` 사슬에 넣고, 합쳐진 행을 받는 사슬의 뒤쪽 절반에서 정렬하십시오:

```
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)
```

`CALL { subquery }`로 감싸도 같은 일을 합니다:

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

컬럼 수가 맞지 않으면 거부된다

```
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
```

이 메시지는 세 연산자 모두에 대해 `UNION`을 이름으로 부릅니다.

참고

집합 연산 뒤에 쓴 `ORDER BY` · `SKIP` · `LIMIT`은 합쳐진 결과가 아니라 **이어진 것 중 마지막 쿼리**에 붙습니다. 그래서 집합이 합쳐지기 전에 그 한 갈래를 정렬하거나 페이지로 나눕니다. 여기서 `LIMIT 1`은 한 행을 내놓지 않습니다:

```
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"
 "Jurassic Park"
(2 rows)
```

공상 과학 갈래에서는 `LIMIT`을 거쳐 한 행만 남았고, 그 갈래는 `LIMIT`이 없으면 영화 10편을 반환합니다. 그 뒤에 Adventure 영화 한 편이 그 행에 합쳐졌습니다. 별칭 해석도 같은 규칙을 따르므로, 정렬 키는 마지막 갈래의 컬럼을 이름으로 불러야 합니다:

```
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;
                ^
```

합쳐진 행을 정렬하거나 페이지로 나눠야 할 때는 언제든 위에 보인 `NEXT` 나 `CALL` 형태를 쓰십시오. SQL에서는 그 집합 연산 위의 `SELECT`도 같은 일을 합니다:

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

호환성

집합 연산자 (ISO/IEC 39075 §14.2)

표준의 `<set operator>` 는 `UNION [ <set quantifier> ] | EXCEPT [ <set quantifier> ] | INTERSECT [ <set quantifier> ]` 이고, 수량자를 주지 않으면 `DISTINCT` 가 암시됩니다. AgensGraph는 세 연산자와 두 수량자를 모두, 정확히 그 규칙대로 구현합니다. openCypher에는 `UNION` 과 `UNION ALL` 만 있으므로, `INTERSECT` 와 `EXCEPT` 는 여기서는 쓸 수 있고 거기서는 쓸 수 없습니다.

첫 쿼리가 아무것도 반환하지 않을 때만 둘째 쿼리의 행을 취하는 표준의 `OTHERWISE` 접속은 구현돼 있지 않습니다.

합쳐진 결과 정렬하기 (ISO/IEC 39075 §14.9)

표준은 `<order by and page statement>` 를 현재 작업 표에 적용하는데, 복합 쿼리 뒤에서 그것은 합쳐진 결과입니다. AgensGraph는 그것을 대신 이어진 것의 마지막 쿼리에 붙이며, **참고**에 그것이 나옵니다. 그 연산이 내놓은 것을 정렬하거나 페이지로 나누려면 집합 연산을 `NEXT` , `CALL { }` , 또는 SQL의 `SELECT` 로 감싸십시오.

관련 항목

- **`RETURN`** — 집합 연산의 모든 갈래가 이것으로 끝난다
- **`NEXT`** GQL — 합쳐진 행에 쿼리를 더 이어 붙이기
- **`CALL { subquery }`** — 집합 연산을 감싸는 또 다른 방법
- **`ORDER BY`** , **`SKIP`** · **`LIMIT`** — 여기서 그것들이 붙는 대상
- SQL과 Cypher 섞기 — SQL에서 집합 연산을 정렬하기

서브쿼리

CALL

GQL 절 2.18에 추가

쿼리 파이프라인 안에서 루틴을 실행합니다. 그 루틴은 결과 행이 작업 표에 조인되는 이름 있는 함수이거나, 그 자리에 적은 서브쿼리입니다.

개요

```
[ OPTIONAL ] CALL function( [ argument [, argument ] ... ] ) YIELD yield_list
[ OPTIONAL ] CALL [ variable_scope ] { subquery }
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`CALL` 에는 두 형태가 있고, 어느 쪽이 되는지는 키워드 뒤의 토큰이 결정합니다:

- **이름**이 뒤따르는 `CALL` 은 이름 있는 루틴 형태입니다. 함수를 호출하고, 필수인 `YIELD` 목록에 적은 컬럼을 작업 행에 더합니다. **`CALL … YIELD`** 를 참고하십시오.
- `(` 나 `{` 가 뒤따르는 `CALL` 은 인라인 형태입니다. 중괄호 사이의 서브쿼리를 실행하며, 입력 행마다 한 번씩 실행하게 할 수도 있습니다. **`CALL { subquery }`** 를 참고하십시오.

`OPTIONAL` 은 두 형태 앞에 모두 붙을 수 있습니다. 루틴이나 서브쿼리가 행을 내놓지 못한 입력 행을 남겨 두고, 그 호출이 반환했을 모든 것을 null로 바인딩합니다. **`OPTIONAL CALL`** 을 참고하십시오.

CALL이 호출하는 것

GQL에서 `CALL` 이 호출하는 것은 **프로시저**입니다. AgensGraph에는 자체 프로시저 카탈로그가 없습니다. `CALL … YIELD` 는 그 이름을 PostgreSQL의 일반 함수에서 찾으며, 기본 설치 상태에서는 프로시저가 아예 등록돼 있지 않습니다.

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT count(*) AS procedures FROM pg_proc WHERE prokind = 'p';
```

```
 procedures
------------
          0
(1 row)
```

그러므로 여기서 "프로시저를 호출한다"는 것은 "함수를 호출한다"는 뜻입니다. 어떤 함수든 되고, 그 함수의 모양이 `YIELD`가 무엇을 이름으로 부를 수 있는지를 결정합니다:

함수의 모양	`YIELD`가 보는 것
스칼라, 예: `length(text)`	함수 이름을 딴 컬럼 하나
집합 반환, 예: `generate_series(int, int)`	함수 이름을 딴 컬럼 하나, 요소마다 행 하나
복합 타입이나 `RETURNS TABLE`, 예: `jsonb_each_text(jsonb)`	출력 필드마다 컬럼 하나, 필드 이름 그대로
위의 어느 모양이든, 함수 자신의 이름으로	그 이름의 출력 컬럼이 없을 때 행 전체. `RETURNS SETOF vertex` 함수는 이렇게 정점을 내놓습니다

내놓을 항목은 이들 중 하나이고 그 밖에는 없습니다. 함수가 반환하지 않는 이름은 쿼리 앞쪽에 같은 이름의 변수가 바인딩돼 있어도 거부됩니다.

PostgreSQL 자체의 `CALL` 문장은 다른 것입니다. 최상위 SQL 문장이고, 실제 프로시저를 요구하며, Cypher 절이 아닙니다. 둘은 겹치지 않고, 각각 상대의 대상을 거부합니다.

매개변수

function

호출할 함수의 이름입니다. `search_path`가 닿지 않는다면 스키마로 한정하십시오. 프로시저는 거부됩니다. `<name>() is a procedure`가 나오고 힌트로 `To call a procedure, use CALL`이 따라옵니다.

argument

값 표현식이며 입력 행마다 평가됩니다. 인자는 앞서 바인딩된 변수를 가리킬 수 있고, 그러면 그 호출이 상관 호출이 됩니다. 속성 접근은 `jsonb`를 내놓으므로, 매개변수가 다른 타입이면 형 변환을 하십시오.

variable_scope

인라인 서브쿼리가 볼 수 있는 바깥 변수입니다. 이름을 적은 것만 보려면 `(v, w)`, 전부 보려면 `(*)`, 아무것도 보지 않으려면 `()`입니다. **이것을 생략하면 아무것도 가져오지 않습니다.** 그 규칙으로 시작하는 **`CALL { subquery }`**를 참고하십시오.

yield_list

`*`, 또는 그 함수의 출력 컬럼 하나 이상이며, 각각 `AS`로 이름을 바꿀 수 있습니다. 필수입니다. **`CALL … YIELD`**를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

subquery	`RETURN`으로 끝나는 읽기 전용 쿼리입니다. 쓰기 절은 거부됩니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

집합 반환 함수를 호출해 그 행을 일치한 영화에 조인합니다:

```
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"
 "0.04081754"
 "0.29592122"
(4 rows)
```

서브쿼리로 행마다 값을 계산하기

인라인 형태는 입력 행마다 하나의 질문에 답하며, 그 본문을 주 패턴 밖에 둡니다:

```
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)
```

프로시저를 호출해 보기

`CALL … YIELD`가 받지 않는 유일한 루틴 종류가 프로시저입니다:

```
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.
```

그리고 SQL의 `CALL` 문장이 받지 않는 유일한 루틴 종류가 함수입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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.
```

참고

렉서는 토큰 하나를 앞서 보고 두 형태를 가릅니다. 덕분에 `call`을 일반 식별자로 계속 쓸 수 있습니다. `MATCH (call:movie) RETURN count(call)`도 동작하고, SQL 문장 `CALL myproc()`도 동작합니다.

이름 있는 루틴 형태는 **가운데** 절입니다. 쿼리를 시작할 수도, 끝낼 수도 없습니다. 인라인 형태는 쿼리를 시작할 수 있지만 범위 절이 없을 때만 그렇습니다. 앞에 아무것도 없는 상태의 범위 절은 `CALL has no preceding clause to import variables from`을 냅니다.

호환성

GQL 적합성

§14.5 call query statement의 쿼리 자리에서 GQL의 call procedure statement와 procedure call(ISO/IEC 39075:2024 §15.1)을 구현합니다.

- **지원** — 표준이 정의하는 프로시저 호출 두 형태, 곧 §15.2의 인라인 호출과 §15.3의 이름 호출. 각각 `OPTIONAL` 접두를 붙일 수 있습니다.
- **미지원** — call data-modifying procedure statement(§13.6). 호출된 루틴이나 서브쿼리는 그래프에 쓸 수 없습니다. GQL 프로시저 카탈로그도 없습니다 — §17.5의 프로시저 참조는 PostgreSQL 함수로 해석되고, 표준의 `QUERY PROCEDURE` 표시에는 대응하는 것이 없습니다.
- **이탈** — 표준은 이름 호출에서 `YIELD` 절을 선택으로 둡니다(§15.3). AgensGraph는 필수로 요구합니다.

진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

openCypher의 `CALL db.labels()` 방식 프로시저 호출은 이름 있는 형태에 대응되지만 이름이 다릅니다. 여기에는 `db.*`나 `dbms.*` 이름공간이 없습니다. 대신 PostgreSQL 카탈로그와 `\dGv` / `\dGe` 메타 명령을 쓰거나, **ag_label**을 직접 읽으십시오.

예제가 만든 루틴을 지웁니다:

```
DROP PROCEDURE noop();
```

관련 항목

- **`CALL … YIELD`** GQL — 이름 있는 루틴 형태의 자세한 설명
- **`CALL { subquery }`** GQL — 인라인 형태와 그 가져오기 목록
- **`OPTIONAL CALL`** GQL — 입력 행을 보존하는 형태
- **서브쿼리 값 표현식** GQL — 절이 아니라 표현식 안의 서브쿼리
- **GQL 적합성 표**

CALL … YIELD

GQL 절 | 2.18에 추가

© 2026 SKAI Worldwide Co., Ltd.

이름 있는 함수를 호출해 그 함수가 내놓는 컬럼을 작업 행에 더하며, 그 행들을 파이프라인에 조인합니다.

개요

```
[ OPTIONAL ] CALL function( [ argument [, argument ] ... ] ) YIELD *
[ OPTIONAL ] CALL function( [ argument [, argument ] ... ] ) YIELD column [ AS alias ] [, column ] ...
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

이 절은 앞선 절의 행에 조인된 함수 스캔이 되어 lateral `SELECT * FROM function(arguments)`처럼 동작하고, `YIELD` 목록은 그 결과에서 고릅니다. 내놓을 항목은 함수의 결과에서만 찾고 다른 곳에서는 찾지 않으므로, 함수가 반환하지 않는 이름은 쿼리에 같은 이름의 변수가 바인딩돼 있어도 오류입니다.

- `YIELD`는 **필수**입니다. `YIELD` 없는 `CALL f(x)`는 구문 오류입니다.
- 이 절은 쿼리를 시작할 수도, 끝낼 수도 없습니다. 문장 맨 앞의 `CALL`은 PostgreSQL의 SQL `CALL`로 읽히므로, 맨 앞에 온 `CALL f() YIELD …`는 구문 오류입니다.
- 인자가 앞서 바인딩된 변수를 가리키면 그 호출은 **상관 호출**이 되어 입력 행마다 한 번 실행됩니다. 인자가 상수면 함수가 한 번 실행되고 그 행들이 파이프라인에 교차 조인됩니다.
- 내놓인 컬럼은 그 뒤로 평범한 바인딩입니다. 걸러 내고, 집계하고, 그것으로 정렬하고, 뒤따르는 `MATCH`에 넘길 수 있습니다.
- 함수가 행을 내놓지 못한 입력 행은 내부 조인이 버리는 것과 똑같이 **버려집니다.** 그 행을 남기려면 `OPTIONAL CALL`을 쓰십시오.

매개변수

function	호출할 함수의 이름이며, `search_path`가 닿지 않으면 스키마로 한정합니다. 인자가 없어도 괄호는 필요합니다. 프로시저는 `<name>() is a procedure`로 거부됩니다.
argument	입력 행마다 평가되는 값 표현식입니다. 속성 접근은 `jsonb`를 내놓으므로, 매개변수가 다른 타입인 함수에는 명시적 형 변환이 필요합니다.
column	그 함수의 출력 컬럼입니다. 스칼라 함수나 단일 컬럼 집합 반환 함수에는 하나가 있고 함수 자신의 이름을 땁니다. 복합 타입이나 `RETURNS TABLE` 함수에는 출력 필드마다 하나가 있고 필드 이름을 그대로 씁니다. `YIELD *`는 그 전부를 취합니다. 함수 자신의 이름은 그 이름의 컬럼이 없을 때 행 전체를 가리키며, `RETURNS SETOF vertex`나 `RETURNS SETOF edge`로 선언된 함수는 이렇게 정점이나 간선 그대로 내놓습니다. 그 밖의 이름은 `function f has no output column "x"`로 거부됩니다.
alias	내놓인 컬럼의 새 이름입니다. 그 컬럼의 이름이 쿼리에 이미 바인딩돼 있다면 필수입니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

영화 한 편의 속성을 행으로 펼칩니다. `jsonb_each_text`는 컬럼 두 개를 반환하고, `YIELD`가 둘 다 이름으로 부릅니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

내놓인 컬럼 이름 바꾸기

`AS` 는 뒤따르는 모든 절에 대해 그 이름을 바꿉니다:

```
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)
```

일부만 내놓는 것도 됩니다. 여기서 `value` 는 바인딩되지 않습니다.

`YIELD *` 로 모든 컬럼 취하기

단일 컬럼 집합 반환 함수는 자기 컬럼에 자신의 이름을 붙입니다:

```
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)
```

인자가 상수이므로 `generate_series` 가 한 번 실행되고 그 두 행이 그 영화에 교차 조인됩니다.

호출을 일치한 행과 상관시키기

행에서 값을 넘기면 함수가 행마다 실행됩니다. `m.year` 는 `jsonb` 이므로 형 변환이 필요합니다:

```
MATCH (m:movie {title: 'The Matrix'})
CALL generate_series(1, (m.year)::int) YIELD generate_series AS n
RETURN count(n) AS years;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 years
-------
 1999
(1 row)
```

형 변환이 없으면 그 함수를 찾지 못합니다:

```
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
        ^
HINT:  No function matches the given name and argument types. You might need to add explicit t
ype casts.
```

호출 두 개 잇기

둘째 호출은 첫 호출이 내놓은 컬럼과 상관될 수 있습니다:

```
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)
```

행 전체를 내놓기

함수 자신의 이름을 가진 출력 컬럼이 없으면 그 이름은 행 전체를 가리킵니다. `jsonb_each_text`는 `(key, value)` 행을 반환합니다:

```
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)
```

`RETURNS SETOF vertex`로 선언된 함수도 같은 방식으로, 자신의 이름 아래 정점을 내놓습니다.

참고

실행 계획은 두 실행 모양 중 어느 것을 받았는지 보여 줍니다. 상관 호출은 내부 쪽이 함수 스캔인 `Nested Loop`입니다. 상관되지 않은 호출은 함수 스캔을 외부 쪽에 두고 그래프 스캔을 구체화합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

내놓인 이름이 이미 유효 범위에 있는 바인딩과 충돌하면, 그것을 덮어 가리는 것이 아니라 거부됩니다. 오류는 `variable "m" yielded by CALL is already bound in the outer query` 입니다. `AS` 로 이름을 바꾸십시오.

그 반대도 거부됩니다. 함수의 출력 컬럼이 아닌 이름은 앞서 같은 이름의 변수가 바인딩돼 있어도 거부됩니다. 앞선 바인딩은 `YIELD` 가 가리키는 대상이 아닙니다.

```
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
                                         ^
```

참고

가상 타입 `record` 를 반환하는 함수는 호출할 수 없습니다. PostgreSQL이 요구하는 컬럼 정의 목록을 둘 자리가 이 절에 없으므로, `CALL jsonb_to_recordset(…) YIELD *` 는 `a column definition list is required for functions returning "record"` 를 냅니다. 그런 호출은 대신 SQL 서브쿼리로 감싸십시오.

호환성

GQL 적합성

§15.1의 call procedure statement 아래에서, §16.14의 yield clause와 함께 GQL의 named procedure call(ISO/IEC 39075:2024 §15.3)을 구현합니다.

- **지원** — 인자 목록, 상관 인자, `YIELD *` 와 `YIELD column [AS alias]` 형태, 내놓을 항목이 프로시저 결과의 컬럼을 가리켜야 한다는 §16.14 구문 규칙 7, 구문 규칙 6의 중복 별칭 거부, 그리고 `OPTIONAL` 접두.
- **미지원** — GQL 프로시저 호출. §15.3은 procedure reference를 §17.5가 기술하는 카탈로그에서 찾는데 AgensGraph에는 그것이 없습니다. 이름은 대신 PostgreSQL 함수로 해석됩니다.
- **이탈** — §15.3은 `YIELD` 절을 선택으로 둡니다. AgensGraph는 그 절이 작업 표에 컬럼을 보태야 하므로 필수로 요구합니다.

진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

openCypher도 같은 절을 쓰지만 그 이름을 프로시저 레지스트리(`db.labels()` , `dbms.components()`)에서 찾고, `YIELD` 없는 `CALL proc()` 을 독립 문장으로 허용합니다. 여기에는 그 레지스트리도, 독립 형태도 없습니다. 이름은 PostgreSQL 함수로 해석되고 `YIELD` 가 필수입니다.

© 2026 SKAI Worldwide Co., Ltd.

관련 항목

- `CALL` GQL — 두 호출 형태, 그리고 여기서 루틴이란 무엇인가
- `CALL { subquery }` GQL — 인라인 형태
- `OPTIONAL CALL` GQL — 함수가 아무것도 내놓지 않은 행 남기기
- GQL 적합성 표

CALL { subquery }

GQL 절 2.18에 추가

읽기 전용 서브쿼리를 파이프라인의 한 절로 실행하며, 그 서브쿼리가 감싸는 쿼리에서 변수를 가져오면 입력 행마다 한 번 실행합니다.

> 중요
>
> `CALL { … }` **만 쓰면 아무것도 가져오지 않습니다.** 중괄호 안의 패턴은 자기 변수 이름을 선언하므로, 본문에 적은 바깥 변수의 이름은 그 바깥 변수를 가리키는 것이 아니라 같은 이름의 제약 없는 새 변수를 바인딩합니다. 필요한 변수를 이름으로 적으십시오. `CALL (v) { … }`, `CALL (*) { … }`, 또는 `CALL { WITH v … }` 입니다.

개요

[**OPTIONAL**] **CALL** { subquery }
[**OPTIONAL**] **CALL** ([variable [, variable] ...]) { subquery }
[**OPTIONAL**] **CALL** (*) { subquery }
[**OPTIONAL**] **CALL** { **WITH** variable [, variable] ... subquery }

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`CALL { … }` 은 서브쿼리를 평가해 그 결과를 작업 표에 조인합니다. 본문이 무엇을 볼 수 있는지는 가져오기 목록이 전적으로 결정합니다.

- 가져오기 목록이 있으면 본문은 그 변수들이 바인딩된 상태로 **입력 행마다** 실행되고, 반환하는 행은 그 행에 조인됩니다.
- 가져오는 것이 없으면 본문은 **상관되지 않습니다.** 한 번 실행되고 그 결과가 모든 입력 행에 교차 조인됩니다.
- 본문은 읽기 전용이며 `RETURN` 으로 끝나야 합니다. 쓰기 절은 파싱 시점에 거부되고, `FINISH` 로 끝나는 본문은 `CALL subquery must RETURN at least one variable` 을 냅니다.
- 본문이 반환하는 이름은 감싸는 쿼리의 바인딩이 됩니다. 바깥에 이미 바인딩된 이름은 덮어 가려지는 것이 아니라 거부됩니다. 본문 안에서도 가져온 변수는 그 이름을 유지합니다. 그중 하나를 다시 바인딩하게 될 `LET`, `UNWIND`, `FOR`, 이름을 바꾸는 `WITH` 는 거부됩니다.
- 본문이 아무것도 반환하지 않은 입력 행은 버려집니다. 그것을 남기려면 `OPTIONAL CALL` 을 쓰십시오.

가져오는 형태들

표기 다섯 가지가 받아들여집니다. 앞의 둘은 아무것도 가져오지 않습니다. 그 둘과 뒤의 셋의 차이가 위 경고의 요점 전부입니다:

형태	본문이 보는 것
`CALL { … }`	없음

© 2026 SKAI Worldwide Co., Ltd.

형태	본문이 보는 것
`CALL () { … }`	없음
`CALL (v, w) { … }`	`v` 와 `w`
`CALL (*) { … }`	유효 범위의 모든 변수
`CALL { WITH v, w … }`	`v` 와 `w`

범위 절과 가져오는 `WITH` 를 함께 쓸 수 있습니다. `CALL (v) { WITH w … }` 는 둘 다 가져옵니다. 비슷해 보이는 세 가지는 대안이 아니라 구문 오류입니다. `CALL (*) { WITH v … }` , `CALL (*, v) { … }` , 그리고 이름만 나열한 것이 아닌 모든 가져오기 `WITH` 입니다(`WITH *` 와 `WITH v AS x` 둘 다 거부됩니다).

매개변수

variable	감싸는 쿼리에서 바인딩된 변수이며, 중괄호 안에서 보이게 할 대상입니다. 존재하지 않는 변수는 `variable "<name>" to import into CALL does not exist` 를 냅니다. 앞에 절이 없는 상태의 범위 절은 `CALL has no preceding clause to import variables from` 을 냅니다.
subquery	`RETURN` 으로 끝나는 읽기 전용 쿼리입니다. `MATCH` , `OPTIONAL MATCH` , `WITH` , `LET` , `UNWIND` , `FOR` , `LOAD FROM` , 중첩된 `CALL` , `DISTINCT` , `ORDER BY` , `SKIP` / `LIMIT` , 집합 연산자가 모두 허용됩니다. **`NEXT`** 는 허용되지 않습니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

가져오기 목록이 중요한 이유

서브쿼리 형태는 이런 질문을 위해 만들어졌습니다. 누군가가 연출한 영화마다 출연진은 몇 명이었나? 가져오기 목록 없이 쓰면 다른 질문에 답합니다:

```
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)
```

모든 영화의 출연진 수가 같고, 25는 그래프 전체의 `acted_in` 간선 수입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH ()-[r:acted_in]->() RETURN count(r) AS acted_in_edges;
```

```
 acted_in_edges
----------------
 25
(1 row)
```

중괄호 안의 `m` 은 바깥에서 일치시킨 `m` 이 아닙니다. 본문 자신의 패턴이 선언한 완전히 새로운 변수이고, `acted_in` 간선으로만 제약됩니다. 그래서 본문은 그래프의 모든 `acted_in` 간선을 상관 없이 한 번 세고, 그 하나의 숫자를 모든 행에 교차 조인합니다.

가져올 것을 이름으로 적으면 의도한 질문을 하게 됩니다:

```
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 (*) { … }` 과 `CALL { WITH m … }` 도 같은 숫자를 냅니다.

오류가 나지 않는 이유는 변수를 가리키는 것은 **표현식**뿐이고, **패턴**은 변수를 새로 선언하기 때문입니다. 같은 실수를 `WHERE` 로 옮기면 잡힙니다:

```
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(...
                                               ^
```

그러므로 이 침묵은 패턴에서 다시 쓴 이름에 한정된 것입니다. 그리고 그것이 흔한 경우이며, 가져오기 목록을 매번 쓰는 이유입니다.

파이프라인을 접지 않고 행마다 집계하기

집계하는 본문은 계산한 값이 0일 때에도 입력 행마다 정확히 한 행을 반환하므로, 바깥의 행 수는 그대로입니다:

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

숫자 하나를 한 번만 계산하기

빈 범위 절은 본문이 상관되지 않았음을 분명히 말해 줍니다. 본문이 그 쿼리의 상수라면 그렇게 쓰십시오:

```
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)
```

변수 여러 개 가져오기

이름을 적은 모든 변수가 보이고, 본문은 그중 일부만 써도 됩니다:

```
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)
```

참고

상관된 본문이 실행 계획에서 행마다 도는 서브쿼리인 것은 아닙니다. 플래너가 그것을 끌어올리므로, 위 쿼리는 그 자리에 패턴을 적었을 때와 같은 조인으로 계획됩니다. `person` · `acted_in` · `movie` 에 대한 `Hash Join` 이고, `MATCH (p:person)-[:acted_in]->(m:movie)` 와 똑같습니다. 집계하는 본문처럼 플래너가 평평하게 만들 수 없는 본문만 `Nested Loop` 아래 `Aggregate` 가 놓인, 행마다 도는 lateral 조인으로 남습니다.

쓰기 절은 그 쓰기 키워드 자리에서 구문 오류로 거부되며, 종류마다 그렇습니다. `CREATE` , `INSERT` , `MERGE` , `SET` , `REMOVE` , `DELETE` , `DETACH DELETE` 입니다.

바깥 쿼리가 이미 바인딩한 이름을 반환하는 본문은 `variable "<name>" returned by the CALL subquery is already bound in the outer query` 를 냅니다. 허용하면 뒤따르는 모든 절에서 바깥 바인딩이 가려지기 때문입니다. 본문의 `RETURN` 에서 별칭을 붙이십시오.

© 2026 SKAI Worldwide Co., Ltd.

호환성

GQL 적합성

§14.5의 쿼리 자리에서, §15.1의 call procedure statement 아래에 GQL의 inline procedure call(ISO/IEC 39075:2024 §15.2)을 구현합니다.

- **지원** — 바인딩 변수 목록을 적은 variable scope clause와 빈 `()` 형태, 중첩, 그리고 `OPTIONAL` 접두.
- **미지원** — 데이터를 바꾸는 본문(§13.6, call data-modifying procedure statement).
- **이탈, 그리고 반드시 알아야 할 것** — §15.2 구문 규칙 3은 범위 절(variable scope clause)이 **없을** 때 들어오는 작업 레코드의 **모든** 필드를 나열한 범위 절로 호출을 다시 쓴다고 규정합니다. 표준대로라면 그냥 쓴 `CALL { … }` 은 `CALL (*) { … }` 과 똑같이 전부를 들여옵니다. AgensGraph는 범위 절이 없으면 `CALL () { … }` 처럼 아무것도 들여오지 않는 것으로 봅니다.
- **확장** — `(*)` 는 표준의 binding variable reference list에 없고, 들여오는 `WITH` 도 없습니다. 둘 다 openCypher에서 왔습니다.

진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

openCypher도 같은 가져오기 목록을 요구하지만, 그것이 없으면 알려 줍니다. 거기서는 `CALL { }` 안에서 선언되지 않은 바깥 변수를 가리키는 것이 조용한 재바인딩이 아니라 오류입니다. openCypher에서 옮겨 온 코드는 계속 동작합니다. `CALL (v) { }`, `CALL (*) { }`, `CALL { WITH v … }` 가 모두 거기서와 같은 뜻이기 때문입니다. openCypher가 거부했던 `CALL { }` 만 쓴 코드는 여기서 실행되고 다른 답을 냅니다.

관련 항목

- `CALL` GQL — 두 호출 형태
- `OPTIONAL CALL` GQL — 본문이 아무것도 반환하지 않은 행 남기기
- `CALL … YIELD` GQL — 대신 이름 있는 함수 호출하기
- 서브쿼리 값 표현식 GQL — 표현식 안의 서브쿼리
- GQL 적합성 표

OPTIONAL CALL

GQL 절 2.18에 추가

호출이 행을 내놓지 못한 입력 행을 남겨 두고, 그 호출이 반환했을 모든 것을 null로 바인딩합니다. `OPTIONAL MATCH` 에 대응하는 `CALL` 쪽 절입니다.

개요

OPTIONAL CALL [variable_scope] { subquery }
OPTIONAL CALL function([argument [, argument] ...]) **YIELD** yield_list

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

설명

`OPTIONAL` 은 `CALL` 의 두 형태 모두에 적용됩니다. 인라인 서브쿼리와 이름 있는 함수입니다. 그냥 쓴 `CALL` 은 호출의 행을 파이프라인에 조인하므로, 행을 내놓지 못한 입력 행은 떨어져 나갑니다. `OPTIONAL CALL` 은 대신 **왼쪽 외부 조인**으로 컴파일되고, 그것이 그 행을 보존합니다.

- 보존은 **입력 행마다** 결정됩니다. 어떤 행에 대해 아무것도 찾지 못한 상관 호출은 그 행만 null로 만들고 나머지는 그대로 둡니다.
- 그 호출이 바인딩했을 모든 변수가 함께 null이 되며, 타입이 무엇이든 그렇습니다. 정점, 간선, 경로, 리스트, 맵, 스칼라 모두입니다.
- 그 바인딩은 jsonb의 `null` 이 아니라 실제 SQL null이므로, 검사하는 방법은 `IS NULL` 입니다.
- `OPTIONAL` 은 행을 보존할 뿐 만들어 내지는 않습니다. 입력이 비어 있으면 출력도 없습니다.
- 보존된 행은 그 뒤로 평범한 행이므로, 뒤따르는 절이 다른 행과 똑같이 그 행을 여러 행으로 늘릴 수 있습니다.

매개변수

variable_scope	서브쿼리가 볼 수 있는 바깥 변수이며, `CALL { subquery }` 와 정확히 같습니다. `(v, w)` , `(*)` , `()` , 또는 중괄호 안에서 가져오는 `WITH` 입니다. 가져오기 규칙은 그대로 적용됩니다. `OPTIONAL CALL { … }` 만 쓰면 아무것도 가져오지 않습니다.
subquery	`RETURN` 으로 끝나는 읽기 전용 쿼리입니다.
function, argument, yield_list	`CALL … YIELD` 와 같습니다. `YIELD` 는 여전히 필수입니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

사람 셋입니다. Keanu Reeves는 연기 이력이 넷, Ryan Gosling은 하나, Christopher Nolan은 없습니다. `OPTIONAL CALL` 은 그 감독을 남깁니다:

```
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)
```

`OPTIONAL` 을 빼면 같은 본문이 그 행을 완전히 잃고, 다른 두 사람은 영향을 받지 않습니다:

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

```
     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)
```

네 행, 한 행, 그리고 보존된 한 행입니다. 이 배수는 쿼리의 성질이 아니라 입력 행마다 정해지는 것입니다.

함수가 아무것도 내놓지 않은 행 보존하기

`OPTIONAL` 은 이름 있는 형태에서도 같은 자리에 놓이며, 어떤 행에 대해 빈 집합을 반환하는 함수를 감당합니다:

```
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;
```

```
     film     | n
--------------+---
 "Avatar"     |
 "The Matrix" | 1
 "The Matrix" | 2
(3 rows)
```

`generate_series(1, 0)` 은 비어 있으므로 Avatar의 행이 `n` 이 null인 채로 살아남습니다. `OPTIONAL` 이 없으면 그 행은 사라지고 The Matrix의 두 행만 남습니다.

보존된 바인딩 검사하기

보존된 행에서는 본문이 반환했을 모든 변수가 null이며, 그것은 SQL null입니다:

```
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)
```

행이 만들어지지 않음을 확인하기

`OPTIONAL` 은 입력 행을 보존합니다. 입력 행이 없으면 보존할 것이 없습니다:

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

참고

실행 계획이 그 방식을 그대로 말해 줍니다. `generate_series` 스캔이 `Nested Loop Left Join`의 null이 될 수 있는 쪽입니다. 그 조인이 Avatar의 행을 보존합니다:

```
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)
```

어떤 본문은 항상 행을 적어도 하나 내놓고, 그런 경우 `OPTIONAL`은 아무것도 바꾸지 않습니다. 그룹화하지 않은 집계가 흔한 경우입니다. 그것은 항상 한 행을 내놓으므로, 이력이 없는 사람은 null이 아니라 `0`을 받습니다. **`CALL { subquery }`**의 `films` 컬럼과 비교해 보십시오.

`OPTIONAL`이 완화하는 것은 그 밖에 없습니다. 본문의 쓰기 절은 여전히 구문 오류이고, 바깥 이름을 반환하는 본문은 여전히 거부되며, `OPTIONAL CALL { … }`만 쓰면 여전히 아무것도 가져오지 않습니다.

호환성

GQL 적합성

GQL call procedure statement(ISO/IEC 39075:2024 §15.1)의 `OPTIONAL` 접두를 구현하며, 그것은 §15.2의 inline procedure call과 §15.3의 named procedure call을 모두 포함합니다.

- **지원** — 두 호출 형태 모두에 붙는 `OPTIONAL`. 입력 행마다 행을 보존하고 반환되는 모든 변수를 null로 바인딩합니다.
- **미지원** — 데이터를 바꾸는 호출에 붙는 `OPTIONAL`(§13.6). 애초에 호출이 그래프에 쓸 수 없기 때문입니다.

진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

`OPTIONAL CALL`은 GQL의 구문이고, openCypher에는 `OPTIONAL MATCH`만 있습니다. 둘이 같은 것을 표현할 수 있는 자리에서는 결과가 일치합니다. 표준이 `OPTIONAL MATCH p`를 `OPTIONAL CALL (imports) { MATCH p RETURN vars }`로 정의하기 때문입니다. 그러므로 `OPTIONAL CALL`은 두 번째 장치가 아니라 더 일반적인 기본 요소입니다.

관련 항목

- **`OPTIONAL MATCH`** — 패턴에 대한, 같은 보존
- **`CALL`** GQL — 두 호출 형태
- **`CALL { subquery }`** GQL — 인라인 형태와 그 가져오기 목록
- **`CALL … YIELD`** GQL — 이름 있는 함수 형태
- GQL 적합성 표

© 2026 SKAI Worldwide Co., Ltd.

서브쿼리 값 표현식

GQL 절 | 2.18에 추가

절이 아니라 서브쿼리로 질문에 답하는 여섯 개의 표현식이며, 값이 놓이는 곳이면 어디서나 쓸 수 있습니다. `RETURN`, `WHERE`, `FILTER`, `WITH`, 그리고 SQL입니다.

개요

```
EXISTS { body }
COUNT { body }
COLLECT { body }
ARRAY { body }
VALUE { body }
expression [ NOT ] IN { body }
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

각 표현식은 자기 본문을 실행해 그 결과를 하나의 값으로 줄입니다. 본문은 상관됩니다. 그 표현식이 적힌 자리의 유효 범위 변수를 보며, `CALL { subquery }`와 달리 선언할 가져오기 목록이 없습니다. 그 변수들은 본문 전체에서 유효 범위에 남으므로, 본문 안의 절은 그 이름을 다시 바인딩할 수 없습니다. 그 이름에 대한 `LET`, `UNWIND`, `FOR`, 다른 값에 그 이름을 붙이는 `WITH`, 그 이름을 반환하는 `CALL { }`은 거부됩니다. 그 이름의 `MATCH (n)`은 바깥 정점을 가리키고, `WITH n`은 그것을 그대로 넘깁니다.

아래 표는 각각이 무엇을 반환하는지, 그리고 본문이 비었을 때 무엇을 하는지 말해 줍니다. 여섯 중 다섯은 빈 본문에 null이 아니라 자기 타입의 값으로 답합니다. false, 0, 빈 리스트입니다. `VALUE { }`에는 그렇게 되돌아갈 값이 없으므로, null을 주는 것은 그것 하나입니다.

표현식	반환하는 것	본문이 비었을 때
`EXISTS { }`	불리언	`false`
`COUNT { }`	행 수, `bigint`	`0`
`COLLECT { }`	본문의 단일 컬럼을 담은 리스트	`[]`
`ARRAY { }`	`COLLECT`와 같음	`[]`
`VALUE { }`	본문의 단일 값, 그 자신의 타입으로	`null`
`x IN { }`	불리언	`false`

본문이 될 수 있는 것

본문은 네 종류가 받아들여지고, 그중 어느 것이 허용되는지는 표현식에 따라 다릅니다:

본문	`EXISTS` / `COUNT`	`COLLECT` / `ARRAY` / `VALUE` / `IN`
패턴만 쓴 것, `WHERE`를 붙일 수도 있음	예	아니요

© 2026 SKAI Worldwide Co., Ltd.

본문	`EXISTS` / `COUNT`	`COLLECT` / `ARRAY` / `VALUE` / `IN`
`MATCH` 문장, `OPTIONAL MATCH` 포함	예	아니요
`RETURN`으로 끝나는 GQL 읽기 문장	예	예
SQL `SELECT`	예	예

- **쓰기 절은 거부됩니다.** `CREATE`, `INSERT`, `MERGE`, `SET`, `REMOVE`, `DELETE`, `DETACH DELETE`는 각각 그 키워드 자리에서 구문 오류를 냅니다.
- 읽기 문장 본문에서는 `DISTINCT`, `ORDER BY`, `SKIP`, `LIMIT`, 집합 연산자, 중첩된 `CALL { }`을 쓸 수 있습니다. `NEXT`는 쓸 수 없습니다.
- 패턴만 쓴 본문에서는 경로에 이름을 붙일 수 없지만(`EXISTS { p = (a)-[]->(b) }`는 구문 오류), `shortestpath()`와 가변 길이 간선은 둘 다 됩니다.
- **SQL `SELECT` 본문은 상관되지 않습니다.** 감싸는 쿼리의 변수를 볼 수 없습니다. 그것을 가리키면 `missing FROM-clause entry for table "<name>"`이 나옵니다.

매개변수

body	위 네 가지 모양 중 하나로 된 서브쿼리입니다. `COLLECT`, `ARRAY`, `VALUE`, `IN`의 본문은 정확히 컬럼 하나를 내놓아야 합니다.
expression	`x IN { }`에서 찾을 값입니다. 정점, 간선, graphid는 `x IN [list]`에서처럼 동일성으로 비교하며, 왼쪽의 원소를 값만 반환하는 본문과 비교하면 타입 오류입니다. 그 밖의 값은 Cypher 본문의 컬럼과 `jsonb`로 만납니다. SQL 본문은 자기 컬럼 타입을 유지하므로 속성을 먼저 벗겨 내십시오. `m.title #>> '{}'`이 `text`를 냅니다.

예제

모든 예제는 **예제 데이터셋**에서 실행됩니다.

패턴이 존재하는지 검사하기

`EXISTS { }`는 조기에 끊습니다. 플래너가 그것을 일치 개수를 세는 것이 아니라 세미 조인이나 `EXISTS` 서브플랜으로 바꿉니다. 패턴만 쓴 것이 가장 짧은 본문이고, 거기에 `WHERE`를 붙일 수 있습니다:

```
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)
```

`RETURN`에서는 불리언 컬럼이며, 일치하는 것이 없으면 null이 아니라 `false`입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person {name: 'Keanu Reeves'}) RETURN EXISTS { (p)-[:directed]->(:movie) } AS
directs;
```

```
 directs
---------
 f
(1 row)
```

일치 개수 세기

패턴에 대한 `COUNT { }`는 그 패턴의 일치 개수를 셉니다:

```
MATCH (m:movie {title: 'The Matrix'}) RETURN COUNT { (:person)-[:acted_in]->(m) } AS
cast_size;
```

```
 cast_size
-----------
 3
(1 row)
```

읽기 문장 본문은 자기 `DISTINCT`·`LIMIT`·그룹화를 유지하므로, 패턴으로는 셀 수 없는 것을 셀 수 있습니다:

```
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)
```

컬럼을 리스트로 모으기

`COLLECT { }`는 리스트를 반환하며, 본문의 순서와 null을 그대로 보존합니다. `ARRAY { }`는 같은 표현식을 표준의 이름으로 부른 것입니다:

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)
```


© 2026 SKAI Worldwide Co., Ltd.

빈 본문은 null이 아니라 빈 리스트를 줍니다:

```
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)
```

값 하나 취하기

`VALUE { }`는 스칼라 서브쿼리입니다. 이 페이지의 다른 모든 것과 달리 컬럼 자신의 타입을 유지하므로, 그래프 요소가 감싸이지 않고 그대로 살아남습니다:

```
MATCH (m:movie {title: 'Inception'})
RETURN VALUE { MATCH (p:person)-[:directed]->(m) RETURN p.name } AS director;
```

```
      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)
```

본문은 최대 한 행을 내놓아야 합니다. 그보다 많으면 실행 시점 오류이므로, 본문이 이미 값 하나로 정해진 것이 아니라면 `ORDER BY … LIMIT 1`이나 집계 함수를 붙이십시오:

```
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
```

서브쿼리에 속하는지 검사하기

`x IN { }`은 구체화된 리스트가 아니라 세미 조인이므로, 본문이 커도 값싸게 유지됩니다. 값과 Cypher 본문의 컬럼은 `jsonb`로 만납니다:

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


© 2026 SKAI Worldwide Co., Ltd.

```
 "Jurassic Park"
(1 row)
```

정점은 동일성으로 비교하므로, 속성이 같은 두 정점도 여전히 서로 다른 정점입니다. 세미 조인은 그때 id로 실행됩니다:

```
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)
```

SQL 본문은 자기 컬럼 타입을 유지합니다. `rating.title`은 `text`이므로, 그것을 `jsonb`로 비교하는 대신 속성을 `#>> '{}'`으로 벗겨 내십시오:

```
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)
```

벗겨 내지 않으면 그 비교에 쓸 연산자가 없습니다. `operator does not exist: jsonb = text`입니다.

투영이나 SQL에서 쓰기

서브쿼리 표현식은 값이므로 `WITH`로 투영한 다음 검사할 수 있습니다:

```
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)
```

변수를 공급하는 Cypher 쿼리로 감싸면 SQL에서도 동작합니다:

```
SELECT (MATCH (m:movie {title: 'The Matrix'}) RETURN COUNT { (:person)-[:acted_in]->(m) }) AS
cast_size;
```

```
 cast_size
-----------
 3
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

참고

본문 어디에 있든 쓰기 절은 그 키워드 자리에서 거부되며, 무엇으로 감쌌든 그렇습니다:

```
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...
                                       ^
```

감싸는 쿼리의 변수를 가리게 될 바인딩도 그렇습니다. 그 절이 최상위에서 내는 것과 같은 오류가 납니다:

```
MATCH (m:movie) RETURN COUNT { LET m = 1 RETURN m } AS n;
```

```
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.
```

참고

`ARRAY`는 PostgreSQL에서와 마찬가지로 예약 키워드이고, `IN`도 그렇습니다. `COLLECT`, `COUNT`, `EXISTS`, `VALUE`는 예약되지 않았으므로 식별자로 계속 쓸 수 있습니다. `RETURN … AS value`도 여전히 동작합니다.

호환성

GQL 적합성

여섯 중 둘이 표준에 있습니다:

- `EXISTS { }`는 GQL의 exists predicate(ISO/IEC 39075:2024 §19.4)를 구현하며, 그 하위 조항의 `<graph pattern>`·`<match statement block>` 본문과 중첩 쿼리 지정까지 포함합니다. 온전히 **지원합니다.**
- `VALUE { }`는 GQL의 value query expression(§20.6), 기능 GQ18 "Scalar subqueries"를 구현하며, 일반 규칙 2의 「빈 본문은 null」 규칙도 따릅니다. **지원합니다.** 다만 §20.6 구문 규칙 10은 본문에 `LIMIT 1`이나 집계 함수가 있어야 한다고 요구하는데, AgensGraph는 어떤 본문이든 받고 두 행 이상을 반환하면 실행 시점 오류를 냅니다.

`COUNT { }`, `COLLECT { }`, `ARRAY { }`, `x IN { }`은 **확장**입니다. ISO/IEC 39075:2024에는 `COUNT`의 서브쿼리 형태가 없습니다 — §20.9는 `COUNT(*)`와 `COUNT(<value expression>)`만 정의합니다 — 그리고 표준의 리스트 집계는 `COLLECT_LIST(…)`이며 그것도 서브쿼리가 아니라 행에 대한 집계입니다. `COUNT { }`와 `COLLECT { }`는 openCypher에서 왔고, `ARRAY { }`는 SQL의 `ARRAY(<subquery>)`를, `x IN { }`은 SQL의 `IN (<subquery>)`를 따릅니다.

진행 상황은 **GQL 적합성 표**에서 확인하십시오.

openCypher

openCypher에는 `EXISTS { }`, `COUNT { }`, `COLLECT { }`가 같은 뜻으로 있습니다. `ARRAY { }`도, `VALUE { }`도, 서브쿼리에 대한 `x IN { }`도 없습니다. 스칼라에는 `head(COLLECT { … })`를, 속하는지 검사에는 `x IN COLLECT { … }`를 씁니다. 그 둘도 여기서 동작하지만 리스트 전체를 구체화하므로, `VALUE { }`와 `x IN { }`을 쓰는 편이 낫습니다.

관련 항목

- **`CALL { subquery }`** GQL — 명시적 가져오기 목록을 가진, 절로서의 서브쿼리

© 2026 SKAI Worldwide Co., Ltd.

- `FILTER` GQL — 이 조건들이 어울리는 절
- `WHERE` — 조건이 놓이는 또 다른 자리
- SQL과 Cypher 섞기 — SQL 본문이 어떻게 들어맞는가
- GQL 적합성 표

데이터 정의

그래프·레이블·속성 인덱스·제약을 만들고 바꾸는 문장들입니다. Cypher 문장이므로 Cypher가 실행되는 곳이면 어디서나 실행됩니다.

CREATE GRAPH

그래프를 만듭니다. 그래프는 스키마 하나, 모든 레이블이 상속하는 기반 레이블 두 개, 그리고 레이블 id를 배정하는 시퀀스로 이루어집니다.

개요

CREATE GRAPH [**IF NOT EXISTS**] graph_name [**AUTHORIZATION** role_specification]
CREATE GRAPH ONLY graph_name

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

그래프는 스키마입니다. `CREATE GRAPH` 는 그 스키마를 만들고 그 안에 객체 세 개를 넣습니다:

- 기반 정점 레이블인 `ag_vertex` ,
- 기반 간선 레이블인 `ag_edge` ,
- 레이블 id를 배정하는 시퀀스인 `ag_label_seq` 입니다.

그 뒤에 만드는 모든 레이블은 `ag_vertex` 나 `ag_edge` 를 상속합니다. `MATCH (n)` 처럼 레이블 없는 패턴에 뜻을 주는 것이 그 상속입니다. 그 객체들과 식별자 체계는 **그래프가 어떻게 저장되는가**가 온전히 설명합니다.

- 그 이름은 그래프 이름으로도, 스키마 이름으로도 쓰이고 있지 않아야 합니다. 그래프가 같은 이름의 스키마를 소유하기 때문입니다.
- 세션의 `graph_path` 가 비어 있으면 `CREATE GRAPH` 가 그것을 새 그래프로 설정합니다. 이미 설정된 `graph_path` 는 그대로 둡니다.
- `CREATE GRAPH ONLY` 는 스키마와 카탈로그 행만 만들고 그 밖에는 아무것도 만들지 않습니다. `pg_dump` 가 내보내는 형태이며, 복원할 때 기반 레이블을 원래의 레이블 id로 직접 다시 만들 수 있게 하기 위한 것입니다. 이렇게 만든 그래프는 그 기반 레이블이 생기기 전까지 레이블을 담을 수 없습니다.

매개변수

IF NOT EXISTS	그 이름의 그래프가 이미 있을 때 오류가 아니라 알림을 냅니다. 아무것도 만들어지지 않고, 이미 있는 것도 바뀌지 않습니다.
graph_name	만들 그래프의 이름입니다.

© 2026 SKAI Worldwide Co., Ltd.

AUTHORIZATION **role_specification**	그 그래프와 함께 만들어지는 객체를 소유하는 롤입니다. 롤 이름, `CURRENT_ROLE`, `CURRENT_USER`, `SESSION_USER` 중 하나를 씁니다. 이 절이 없으면 그래프는 그 문장을 실행한 롤에 속합니다.
ONLY	`ag_vertex` · `ag_edge` · `ag_label_seq` 없이 스키마와 카탈로그 행만 만듭니다. 그냥 쓴 형태와 달리 `graph_path`에 손대지 않습니다.

예제

예제는 **예제 데이터셋**을 쓰지 않고 자체 그래프인 `boxoffice`를 만들고, 끝에서 그것을 지웁니다.

그래프 만들기

```
CREATE GRAPH boxoffice;
```

```
CREATE GRAPH
```

이 문장이 비어 있던 `graph_path`를 차지했습니다:

```
SHOW graph_path;
```

```
 graph_path
------------
 boxoffice
(1 row)
```

그 그래프가 `\dG`에 나열됩니다:

```
\dG boxoffice
```

```
  List of graphs
   Name    | Owner
-----------+-------
 boxoffice | agens
(1 row)
```

그리고 그것은 이미 기반 레이블 두 개를 담고 있습니다:

```
\dGl boxoffice.*
```

```
             List of labels
   Graph    |   Name    |  Type  | Owner
------------+-----------+--------+-------
 boxoffice | ag_edge   | edge   | agens
 boxoffice | ag_vertex | vertex | agens
(2 rows)
```


© 2026 SKAI Worldwide Co., Ltd.

없을 때만 그래프 만들기

```
CREATE GRAPH IF NOT EXISTS boxoffice;
```

```
NOTICE:  graph "boxoffice" already exists, skipping
CREATE GRAPH
```

이미 있는 그래프는 손대지 않습니다. 이것은 "만들거나 교체하기"가 아닙니다.

다른 롤이 소유하는 그래프 만들기

```
CREATE ROLE curator;
```

```
CREATE GRAPH festival AUTHORIZATION curator;
```

```
CREATE GRAPH
```

```
\dG festival
```

```
   List of graphs
   Name    |  Owner
-----------+---------
 festival | curator
(1 row)
```

기반 레이블 없이 스키마만 만들기

```
CREATE GRAPH ONLY skeleton;
```

```
CREATE GRAPH
```

그래프는 존재하지만 아무것도 담고 있지 않습니다:

```
\dGl skeleton.*
```

```
No matching labels found.
```

`ag_vertex`가 없으므로 그 안에 레이블을 만들 수 없습니다:

```
SET graph_path = skeleton;
CREATE VLABEL film;
```

```
ERROR:  relation "skeleton.ag_vertex" does not exist
```

그래프 세 개와 그 롤을 정리합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
DROP GRAPH boxoffice CASCADE;
DROP GRAPH festival CASCADE;
DROP GRAPH skeleton;
```

```
DROP ROLE curator;
```

`skeleton`에는 그 안에 만들어진 것이 없으므로 `CASCADE`가 필요하지 않습니다. 그냥 쓴 형태로 만든 그래프는 항상 필요합니다. **`DROP GRAPH`**를 참고하십시오.

참고

`graph_path`는 한정하지 않은 레이블 이름이 속하는 그래프를 지정합니다. 세션 단위 설정이므로 새 세션마다 설정하거나, 데이터베이스나 롤에 붙여 두십시오:

```
ALTER DATABASE <database> SET graph_path = boxoffice;
```

그래프 전체에 작용하는 문장이 두 개 더 있습니다. 둘 다 `IF EXISTS` 형태는 없습니다:

```
CREATE GRAPH receipts;
ALTER GRAPH receipts RENAME TO takings;
ALTER GRAPH takings OWNER TO CURRENT_USER;
SHOW graph_path;
```

```
 graph_path
------------
 takings
(1 row)
```

이름을 바꾸면 그 아래의 스키마도 그래프와 함께 옮겨지고, 세션의 `graph_path`도 그것을 따라갑니다. 그래서 위에서 `receipts`가 아니라 `takings`로 읽힙니다.

```
DROP GRAPH takings CASCADE;
```

호환성

ISO/IEC 39075 (GQL)

GQL은 `<create graph statement>` (§12.4)를 `CREATE [PROPERTY] GRAPH <name> {<open graph type> | <of graph type>}`로 정의합니다. 그래프는 어떤 내용이든 받는 `ANY`로 만들거나, 정점 타입·간선 타입·속성을 제약하는 **그래프 타입**에 맞춰 만듭니다. AgensGraph의 그래프는 언제나 열려 있으므로 그래프 타입을 적지 않고, 적을 수도 없습니다. `PROPERTY` 표기는 받아들여지지 않습니다. `CREATE PROPERTY GRAPH boxoffice`는 구문 오류입니다. 그래프 타입은 **GQL 적합성 표**에 계획으로 기록돼 있습니다.

`AUTHORIZATION`과 `ONLY`는 AgensGraph 자신의 것이며, GQL에는 둘 다 없습니다. GQL의 `OR REPLACE` 형태도 받아들여지지 않습니다.

PostgreSQL

`CREATE GRAPH`는 `CREATE SCHEMA`에 대응하고, `IF NOT EXISTS`와 `AUTHORIZATION`도 거기서와 같은 뜻입니다. 차이는 내용물입니다. `CREATE SCHEMA`는 빈 스키마를 만듭니다. `CREATE GRAPH`는 기반 레이블과 레이블 id 시퀀스까지 만들고, Cypher가 찾을 수 있도록 그 스키마를 `ag_graph`에 등록합니다. `CREATE SCHEMA`만으로는 그래프가 만들어지지 않습니다.

관련 항목

- **`DROP GRAPH`**

© 2026 SKAI Worldwide Co., Ltd.

- `CREATE VLABEL`
- `CREATE ELABEL`
- 그래프가 어떻게 저장되는가

DROP GRAPH

그래프를 지우며, `CASCADE`를 쓰면 그 안의 모든 레이블·정점·간선까지 지웁니다.

개요

DROP GRAPH [**IF EXISTS**] graph_name [, ...] [**CASCADE** | **RESTRICT**]

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`DROP GRAPH`는 그 그래프의 카탈로그 행과 스키마를 지웁니다. 그래프가 담고 있는 모든 것이 그 스키마에 의존합니다. 레이블, 그 뒤의 테이블, 그 행, 인덱스, 제약, 시퀀스가 그렇습니다. 그래서 기본값인 `RESTRICT`는 그중 무엇이라도 남아 있으면 거부합니다.

그냥 쓴 `CREATE GRAPH`로 만든 그래프는 결코 비어 있지 않습니다. `ag_vertex`, `ag_edge`, `ag_label_seq`가 함께 만들어지기 때문입니다. 그래서 그렇게 만든 그래프는 모두 `CASCADE`가 필요합니다. 그 셋을 하나도 만들지 않는 `CREATE GRAPH ONLY`로 만든 그래프만 `CASCADE` 없이 지울 수 있습니다.

- 그래프를 지우는 데 그 레이블이 비어 있어야 하는 것은 아닙니다. `CASCADE`가 테이블과 함께 행도 가져갑니다.
- 지운 그래프가 세션의 `graph_path`였다면 그 설정이 비워집니다.
- 그래프의 스키마만 따로 지울 수는 없습니다. 그것에 `DROP SCHEMA`를 쓰면 거부되고, 오류 메시지가 이 문장을 가리킵니다.

매개변수

IF EXISTS	존재하지 않는 그래프에 대해 오류가 아니라 알림을 냅니다. 같은 문장의 다른 이름은 그대로 지워집니다.
graph_name	지울 그래프의 이름입니다. 쉼표로 구분해 여러 개를 줄 수 있습니다.
CASCADE	그 그래프에 의존하는 객체를 지웁니다. 그 레이블, 그 테이블과 행, 그리고 레이블 id 시퀀스입니다.
RESTRICT	그 그래프에 의존하는 것이 있으면 거부합니다. 이것이 기본값입니다.

예제

그래프와 그 안의 모든 것 지우기

```
CREATE GRAPH clearance;
SET graph_path = clearance;
CREATE VLABEL film;
CREATE (:film {title: 'Arrival'});
```


© 2026 SKAI Worldwide Co., Ltd.

`CASCADE`가 없으면 지우기가 거부되고, 메시지가 무엇이 막고 있는지 나열합니다:

```
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`는 그 넷을 모두 가져가고, `film`에 있던 정점도 함께 가져갑니다:

```
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`가 세션의 `graph_path`였으므로 그 설정이 이제 비었습니다:

```
SHOW graph_path;
```

```
 graph_path
------------

(1 row)
```

여러 그래프 지우기, 이미 없는 것은 건너뛰기

```
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
```

없는 이름은 보고되고, 존재하는 둘은 지워집니다:

```
\dG archive*
```


© 2026 SKAI Worldwide Co., Ltd.

```
No matching graphs found.
```

참고

그래프를 지우지 않고 비우려면 `MATCH (n) DETACH DELETE n`을 실행하고, 그다음 더 이상 필요하지 않은 레이블을 지우십시오. 스키마와 기반 레이블은 그대로 남습니다. `DROP VLABEL`이 레이블을 지우려면 그 레이블이 비어 있어야 하므로, 그 삭제가 보통 첫 단계입니다. `DROP GRAPH ... CASCADE`는 그 요구 조건을 아예 건너뜁니다. 레이블을 하나씩 지우는 것이 아니라 테이블을 지우기 때문입니다.

호환성

ISO/IEC 39075 (GQL)

GQL의 `<drop graph statement>` (§12.5)는 `DROP [PROPERTY] GRAPH [IF EXISTS] <name>`입니다. AgensGraph는 그중 `DROP GRAPH`와 `IF EXISTS` 부분을 받습니다. 그리고 GQL에는 없는 이름 목록과 `CASCADE` / `RESTRICT` 선택을 더했습니다. `PROPERTY` 표기는 받아들여지지 않습니다. `DROP PROPERTY GRAPH clearance`는 구문 오류입니다.

PostgreSQL

`DROP GRAPH`는 `DROP SCHEMA`에 대응하고, `IF EXISTS`와 이름 목록, `CASCADE` / `RESTRICT`도 거기서와 같이 동작합니다. 다른 점은 반쯤 지워진 그래프를 남기기를 거부한다는 것입니다. 그래프의 스키마에 `DROP SCHEMA`를 쓰면 `cannot drop schema clearance because graph clearance requires it`으로 실패하므로, 카탈로그 행과 스키마는 언제나 함께 사라집니다.

관련 항목

- `CREATE GRAPH`
- `DROP VLABEL` · `DROP ELABEL`
- 그래프가 어떻게 저장되는가

CREATE VLABEL

정점 레이블을 만듭니다. 그래프의 기반 정점 레이블을 상속하는 `(id, properties)` 테이블이며, 속성을 타입 컬럼으로 승격할 수도 있습니다.

개요

```
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 ]
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

설명

정점 레이블은 `graphid` 타입의 `id` 와 `jsonb` 타입의 `properties` 컬럼을 가진 테이블입니다. 부모를 지정하지 않으면 그래프의 `ag_vertex` 를 상속합니다. 정점 레이블을 만들면 그 id에 번호를 매기는 시퀀스와 `id` 에 대한 기본 키도 함께 만들어집니다. 그 배치는 **그래프가 어떻게 저장되는가**가 다룹니다.

레이블은 `graph_path` 가 지정한 그래프에 만들어집니다.

- **옵션은 순서가 중요합니다.** 보인 순서대로 와야 합니다. `DISABLE INDEX` , 그다음 `INHERITS` , 그다음 `WITH` , 그다음 `TABLESPACE` 입니다. 다른 순서는 구문 오류입니다.
- `INHERITS` 에 지정한 부모는 같은 그래프에 있어야 하고 정점 레이블이어야 합니다.
- `IF NOT EXISTS` 와 `ONLY` 는 서로 다른 형태에 속하므로 함께 쓸 수 없습니다.
- Cypher는 필요할 때 레이블을 만듭니다. `film` 레이블이 없는 그래프에 `CREATE (:film)` 을 쓰면 그 레이블이 만들어집니다. 옵션, 승격 컬럼, 부모가 필요할 때 `CREATE VLABEL` 을 쓰십시오.

승격된 속성

승격된 속성은 `properties` 옆에 있는 실제 타입 컬럼이며, `properties` 에서 계산해 저장한 것입니다. 그 속성을 읽으면 JSON 문서를 훑는 대신 그 컬럼으로 갑니다. 세 형태는 값이 어디서 오는지가 다릅니다:

- `column_name data_type GENERATED` 는 키가 그 컬럼 이름인 속성을 비춥니다.
- `column_name data_type GENERATED ( property_key )` 는 지정한 키를 비추므로, 컬럼 이름을 다르게 붙일 수 있습니다. 그 키는 식별자여서 소문자로 바뀌며, Cypher가 맵 키를 저장하는 방식과 같습니다.
- `column_name data_type GENERATED ALWAYS AS ( expression ) STORED` 는 그 레이블 자신의 컬럼에 대한 SQL 표현식으로 컬럼을 계산합니다.

어떤 컬럼이 속성을 대신해 답하는지는 어느 형태로 썼는지가 아니라 그 표현식이 결정합니다. 형 변환을 붙이든 붙이지 않든 `properties ->> 'year'` 처럼 `properties` 에서 키 하나를 그냥 읽는 것은 `ag_label_property` 에 기록됩니다. Cypher의 속성 읽기가 참조하는 카탈로그가 그것입니다. 그 읽기를 다른 무엇으로 감싸면 그 컬럼은 평범한 생성 컬럼이 됩니다. SQL은 그것을 읽고, `n.<key>` 는 그것을 보지 못합니다. 승격이 언제 할 만한지는 **속성을 컬럼으로 승격하기**가 설명합니다.

승격된 속성이 될 수 없는 것 네 가지입니다:

- **생성 컬럼이 아닌 것.** `column_name data_type` 만 쓰면 `a non-generated promoted column is not supported yet` 로 거부됩니다.
- **`VIRTUAL`.** 승격 컬럼은 인덱스를 지니려면 구체화돼야 하므로, `VIRTUAL` 은 `promoted generated properties must be STORED` 로 거부됩니다. `STORED` 가 기본값이며 생략할 수 있습니다.
- **`GENERATED BY DEFAULT`.** `GENERATED ALWAYS` 만 받아들여집니다.
- **`id` · `properties` · `start` · `end` 라는 이름.** 그것들은 그래프 요소의 구조 컬럼이므로, 그 이름은 예약된 것으로 거부됩니다.

데이터 타입은 텍스트 입력 함수가 불변인 타입이어야 하고, 길이를 지정하는 타입이어서도 안 됩니다. `date` , `timestamp` , `timestamptz` 와 배열 타입은 `generation expression is not immutable` 로 거부됩니다. `varchar(n)` 과 `char(n)` 은 값을 길이에 맞추는 과정에서 컬럼과 속성이 서로 다른 값을 담게 될 수 있으므로 거부됩니다. `smallint` , `int` , `bigint` , `numeric` , `real` , `double precision` , `text` , `boolean` , `uuid` , `jsonb` 는 받아들여집니다.

매개변수

UNLOGGED	레이블의 테이블을 unlogged로 만듭니다. 그 내용은 미리 쓰기 로그에 기록되지 않으며 비정상 종료 뒤에는 사라집니다.

© 2026 SKAI Worldwide Co., Ltd.

IF NOT EXISTS	그 이름의 레이블이 이미 있을 때 오류가 아니라 알림을 냅니다. 아무것도 만들어지지 않고, 이미 있는 레이블의 옵션도 바뀌지 않습니다.
label_name	만들 레이블의 이름이며, `graph_path`가 지정한 그래프 안에 만듭니다.
promoted_property	위 세 형태 중 하나로, 타입 컬럼으로 함께 비출 속성입니다. 컬럼은 적은 순서대로 `properties` 뒤에 추가됩니다.
DISABLE INDEX	레이블의 인덱스를 만들되 무효 상태로 두어, 행을 적재할 때 그것을 유지하지 않게 합니다. 읽기와 쓰기는 그대로 동작하고, 플래너는 무효 인덱스를 무시합니다. 다시 만들 때는 **`REINDEX VLABEL`**을 쓰십시오. `ENABLE INDEX`는 없습니다.
INHERITS (parent_label [, ...])	기본 부모인 `ag_vertex`를 대신하는 부모 레이블입니다. 각각 같은 그래프의 정점 레이블이어야 합니다. **레이블 상속**을 참고하십시오.
WITH (storage_parameter [= value] [, ...])	`CREATE TABLE`에서와 같이, 레이블 테이블의 저장 매개변수입니다.
TABLESPACE tablespace_name	레이블 테이블의 테이블스페이스입니다. 이것이 없으면 `default_tablespace`가 결정합니다.
ONLY label_name (label_id)	주어진 레이블 id로, 기본 부모·id 기본값·기본 키 없이 레이블을 만듭니다. 그 레이블 id는 이 레이블이 내주는 모든 `graphid`의 상위 16비트입니다. 덤프한 그래프의 id를 복원이 보존할 수 있도록 `pg_dump`가 내보내는 형태입니다. 이렇게 만든 레이블은 그것들을 되돌려 놓기 전까지 Cypher에서 쓸 수 없으므로, 덤프를 재현할 때만 쓰십시오.

예제

예제는 **예제 데이터셋**을 쓰지 않고 자체 그래프인 `filmstore`를 만들고, 끝에서 그것을 지웁니다.

정점 레이블 만들기

```
CREATE GRAPH filmstore;
SET graph_path = filmstore;
CREATE VLABEL film;
```

```
CREATE VLABEL
```

```
\d filmstore.film
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                                  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
```

없을 때만 레이블 만들기

```
CREATE VLABEL IF NOT EXISTS film;
```

```
NOTICE:  label "film" already exists, skipping
CREATE VLABEL
```

부모 아래에 레이블 만들기

```
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` 는 `ag_vertex` 대신 `work` 를 상속하므로 `MATCH (n:work)` 가 다큐멘터리에도 일치합니다. 나머지는 레이블 상속이 다릅니다.

인덱스를 만들기 전에 레이블 적재하기

```
CREATE VLABEL bulk DISABLE INDEX;
```

```
\d filmstore.bulk
```

```
                                                  Table "filmstore.bulk"
   Column    |  Type    | Collation | Nullable |                                                  Defau
lt
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------------
 id          | graphid |           | not null | graphid(graph_labid('filmstore.bulk'::cstring),
```


© 2026 SKAI Worldwide Co., Ltd.

```
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
```

기본 키는 존재하지만 `INVALID` 로 표시됩니다. 행을 적재하고, 그다음 `REINDEX VLABEL bulk` 를 실행해 그것을 만드십시오.

저장 매개변수와 테이블스페이스 설정하기

```
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)
```

속성 두 개를 타입 컬럼으로 승격하기

```
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
```

쓰기에 관해 바뀌는 것은 없습니다. 속성은 여전히 `properties` 로 들어가고, 컬럼이 그것을 따라갑니다:

```
CREATE (:title {name: 'Arrival', year: 2016});
MATCH (t:title) RETURN t.name AS name, t.year AS year;
```

```
  name   | year
---------+------
 Arrival | 2016
(1 row)
```


© 2026 SKAI Worldwide Co., Ltd.

같은 값이 컬럼에도 있고, SQL에서 닿을 수 있습니다:

```
SELECT name, year FROM filmstore.title;
```

```
  name   | year
---------+------
 Arrival | 2016
(1 row)
```

값이 그 컬럼의 종류와 맞지 않는 속성은 쓰기 시점에 거부되며, 양쪽을 모두 이름으로 알려 줍니다:

```
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"
```

키가 컬럼 이름과 다른 속성 승격하기

```
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
```

키를 `releaseDate`로 적었지만 `releasedate`로 저장됐습니다. 여기서 속성 키는 식별자이고 식별자는 소문자로 바뀌기 때문입니다. Cypher는 이 속성을 `n.releasedate`로 읽고, SQL은 그 컬럼을 `released`로 읽습니다.

`properties`에서 컬럼 계산하기

긴 형태는 속성 키가 아니라 그 레이블 자신의 컬럼에 대한 SQL 표현식을 받습니다:

```
CREATE VLABEL slug (handle text GENERATED ALWAYS AS (lower(properties ->> 'name')) STORED);
```

```
\d filmstore.slug
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                                  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
```

Cypher 쓰기는 그 쓰기가 무엇이었든 행 수를 `UPDATE` 태그로 보고하고, 그 컬럼은 계산되어 저장됩니다:

```
SELECT handle FROM filmstore.slug;
```

```
 handle
---------
 arrival
(1 row)
```

그러나 이 컬럼은 어떤 속성 키에도 대응하지 않으므로 Cypher는 그것을 보지 못합니다:

```
MATCH (s:slug) RETURN s.handle AS handle;
```

```
 handle
--------

(1 row)
```

`s.handle`은 `properties`에서 `handle` 키를 읽는데, 그것을 쓴 것은 아무것도 없습니다.

표현식이 키 하나를 그냥 읽는 것인 긴 형태는 축약형처럼 기록되고, Cypher도 그것을 봅니다:

```
CREATE VLABEL runtime (minutes int GENERATED ALWAYS AS ((properties ->> 'minutes')::int)
STORED);
CREATE (:runtime {minutes: 116});
MATCH (r:runtime) RETURN r.minutes AS minutes;
```

```
 minutes
---------
     116
(1 row)
```

그러므로 그 컬럼이 어떤 속성을 대신해 답해야 한다면 키 하나를 꺼내 쓰고, 그 컬럼이 SQL만을 위한 것이라면 계산해 쓰십시오.

© 2026 SKAI Worldwide Co., Ltd.

알아 둘 만한 거부 사례

아래는 모두 그대로 거부됩니다:

```
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.
```

속성 키는 계층 전체에서 하나의 컬럼으로 이어져야 합니다. 그래서 부모가 이미 승격한 키를 승격하는 것은 컬럼 이름이 같지 않으면 거부됩니다:

```
CREATE VLABEL feature (year int GENERATED) INHERITS (work);
CREATE VLABEL short (released int GENERATED (year)) INHERITS (feature);
```

```
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.
```

반대 경우도 거부됩니다. 조상이 이미 쓰는 컬럼 이름으로 다른 키를 승격하면, 키 둘이 상속된 컬럼 하나를 읽게 됩니다.

주어진 레이블 id로 레이블 다시 만들기

레이블 id는 그래프의 `ag_label_seq`에서 만들어진 순서대로 나옵니다. `ONLY`는 그것을 명시적으로 정하며, 복원이 덤프에 담긴 id를 재현하는 방법이 그것입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

그 레이블에는 부모도, `id` 기본값도, 기본 키도 없습니다. 그래서 그 상태로는 Cypher에서 쓸 수 없습니다:

```
\d filmstore.restored
```

```
              Table "filmstore.restored"
   Column    |   Type    | Collation | Nullable | Default
-------------+-----------+-----------+----------+---------
 id          | graphid   |           | not null |
 properties  | jsonb     |           | not null |
```

그래프가 이미 내준 id를 다시 쓰면 `(graphid, labid)` 에 걸린 고유 인덱스에서 실패합니다.

마지막으로 예제가 만든 그래프를 지웁니다:

```
DROP GRAPH filmstore CASCADE;
```

참고

레이블 이름은 한정할 수 없습니다. `CREATE VLABEL filmstore.film` 은 구문 오류이고, 그래프는 항상 `graph_path` 에서 옵니다. 설정된 것이 없으면 이 문장은 `graph_path is NULL` 로 멈춥니다.

`\dGv` 는 정점 레이블을 나열하고 일치하는 것마다 설명해 줍니다. `\dGl` 은 두 종류의 레이블을 자세한 설명 없이 나열합니다.

> 참고
>
> `enable_property_promotion` 은 Cypher의 속성 읽기를 승격된 컬럼에서 답할지 결정합니다. 기본값은 켜짐이고, 그것을 끄면 실행 계획뿐 아니라 결과도 달라질 수 있습니다. 컬럼은 자기 타입의 규칙으로 값을 비교하고 정렬하지만, `jsonb` 컬럼인 `properties` 는 `jsonb` 순서를 쓰기 때문입니다.

© 2026 SKAI Worldwide Co., Ltd.

호환성

ISO/IEC 39075 (GQL)

GQL에는 레이블 DDL이 없습니다. 표준의 스키마 장치는 **그래프 타입**, `<create graph type statement>` (§12.6)입니다. 그래프 타입은 그래프 하나 전체에 대해 정점 타입과 간선 타입을 한 번에 선언하고, 그것을 참조하는 그래프를 제약합니다. AgensGraph는 레이블을 하나씩 선언하고, 그 그래프는 열려 있습니다. 레이블은 자기 정점이 어떤 속성을 지녀야 하는지 제한하지 않습니다. 그래프 타입은 **GQL 적합성 표**에 계획으로 기록돼 있습니다.

PostgreSQL

`CREATE VLABEL` 은 `CREATE TABLE` 에 대응합니다. `UNLOGGED` , `IF NOT EXISTS` , `INHERITS` , `WITH` , `TABLESPACE` 는 거기서와 같은 뜻이고, 승격 속성 형태는 `GENERATED ALWAYS AS ... STORED` 생성 컬럼입니다. 나머지는 `CREATE TABLE` 에 없습니다. 직접 고를 수 없는 고정된 컬럼 목록, `ag_label` 과 `ag_label_property` 등록, 자동으로 붙는 부모, 그리고 순서가 중요한 옵션 배치입니다. 반대 방향으로는, `enable_graph_ddl` 을 켜 두지 않으면 레이블의 테이블을 `ALTER TABLE` 로 다시 모양 잡을 수 없습니다. `ALTER VLABEL` · `ALTER ELABEL` 을 참고하십시오.

관련 항목

- `CREATE ELABEL`
- `ALTER VLABEL` · `ALTER ELABEL`
- `DROP VLABEL` · `DROP ELABEL`
- `CREATE PROPERTY INDEX`
- 속성을 컬럼으로 승격하기
- 레이블 상속

CREATE ELABEL

간선 레이블을 만듭니다. 그래프의 기반 간선 레이블을 상속하는 `(id, start, end, properties)` 테이블이며, 속성을 타입 컬럼으로 승격할 수도 있습니다.

개요

```
CREATE [ UNLOGGED ] ELABEL [ IF NOT EXISTS ] label_name [ ( promoted_property [, ... ] ) ]
  [ DISABLE INDEX ]
  [ INHERITS ( parent_label [, ... ] ) ]
  [ WITH ( storage_parameter [= value] [, ... ] ) ]
  [ TABLESPACE tablespace_name ]

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 ]
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

설명

간선 레이블은 컬럼 네 개를 가진 테이블입니다. `graphid` 타입의 `id`, 간선의 `start` · `end` 정점의 `graphid`, 그리고 `jsonb` 타입의 `properties` 컬럼입니다. 부모를 지정하지 않으면 그래프의 `ag_edge`를 상속합니다.

간선 레이블을 만들면 그 id에 번호를 매기는 시퀀스와 인덱스 세 개도 함께 만들어집니다. 둘은 `(start, "end")`와 `("end", start)`에 걸린 btree 인덱스이고, 그것이 순회를 어느 방향으로든 빠르게 만들어 줍니다. 셋째는 `id`에 걸린 BRIN 인덱스입니다. 간선 레이블에는 `id`에 대한 기본 키가 생기지 않습니다. 간선은 양 끝점을 통해 도달하기 때문입니다. 그 배치는 **그래프가 어떻게 저장되는가**가 다릅니다.

레이블은 `graph_path`가 지정한 그래프에 만들어집니다.

- **옵션은 순서가 중요합니다.** 보인 순서대로 와야 합니다. `DISABLE INDEX`, 그다음 `INHERITS`, 그다음 `WITH`, 그다음 `TABLESPACE`입니다. 다른 순서는 구문 오류입니다.
- `INHERITS`에 지정한 부모는 같은 그래프에 있어야 하고 간선 레이블이어야 합니다. 정점 레이블을 지정하면 거부됩니다.
- `IF NOT EXISTS`와 `ONLY`는 서로 다른 형태에 속하므로 함께 쓸 수 없습니다.
- 간선 레이블은 양 끝점이 어떤 레이블이어야 하는지 제한하지 않습니다. `start`와 `end`는 외래 키가 아니라 평범한 `graphid` 컬럼입니다.
- Cypher는 필요할 때 레이블을 만듭니다. `directed` 레이블이 없는 그래프에 `CREATE (a)-[:directed]->(b)`를 쓰면 그 레이블이 만들어집니다. 옵션, 승격 컬럼, 부모가 필요할 때 `CREATE ELABEL`을 쓰십시오.

승격된 속성

승격된 속성은 `properties` 옆에 있는 타입 컬럼이며, `properties`에서 계산해 저장한 것입니다. 그 속성을 읽으면 JSON 문서 안이 아니라 그 컬럼으로 갑니다. 세 형태와 그 제한은 정점 레이블과 같고, **`CREATE VLABEL`**이 그것을 설명합니다. 간선 레이블에서 예약된 컬럼 이름은 `id`, `properties`, `start`, `end`입니다.

매개변수

UNLOGGED	레이블의 테이블을 unlogged로 만듭니다. 그 내용은 미리 쓰기 로그에 기록되지 않으며 비정상 종료 뒤에는 사라집니다.
IF NOT EXISTS	그 이름의 레이블이 이미 있을 때 오류가 아니라 알림을 냅니다. 아무것도 만들어지지 않고, 이미 있는 레이블의 옵션도 바뀌지 않습니다.
label_name	만들 레이블의 이름이며, `graph_path`가 지정한 그래프 안에 만듭니다.
promoted_property	타입 컬럼으로 함께 비출 속성입니다. 컬럼은 적은 순서대로 `properties` 뒤에 추가됩니다.
DISABLE INDEX	레이블의 인덱스 세 개를 만들되 무효 상태로 두어, 행을 적재할 때 그것을 유지하지 않게 합니다. 읽기와 쓰기는 그대로 동작하고, 플래너는 무효 인덱스를 무시합니다. 다시 만들 때는 **`REINDEX ELABEL`**을 쓰십시오. `ENABLE INDEX`는 없습니다.
INHERITS (parent_label [, ...])	기본 부모인 `ag_edge`를 대신하는 부모 레이블입니다. 각각 같은 그래프의 간선 레이블이어야 합니다. **레이블 상속**을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

WITH (storage_parameter [= value] [, ...])	`CREATE TABLE`에서와 같이, 레이블 테이블의 저장 매개변수입니다.
TABLESPACE tablespace_name	레이블 테이블의 테이블스페이스입니다. 이것이 없으면 `default_tablespace`가 결정합니다.
ONLY label_name (label_id)	주어진 레이블 id로, 기본 부모·id 기본값·끝점 인덱스 없이 레이블을 만듭니다. 덤프한 그래프의 id를 복원이 보존할 수 있도록 `pg_dump`가 내보내는 형태입니다. 덤프를 재현할 때만 쓰십시오.

예제

예제는 **예제 데이터셋**을 쓰지 않고 자체 그래프인 `castlists`를 만들고, 끝에서 그것을 지웁니다.

간선 레이블 만들기

```
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))
 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
```

레이블을 만들 때 `start`와 `end`의 통계 목표도 1000으로 설정됐습니다. 순회의 행 추정이 그 두 컬럼의 분포에 달려 있기 때문입니다:

```
SELECT attname, attstattarget FROM pg_attribute
 WHERE attrelid = 'castlists.worked_on'::regclass AND attnum > 0;
```


© 2026 SKAI Worldwide Co., Ltd.

```
  attname   | attstattarget
------------+---------------
 id         |
 start      |          1000
 end        |          1000
 properties |
(4 rows)
```

여러 간선 종류를 한 부모 아래로 묶기

```
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
```

이제 부모에 대한 패턴이 두 종류의 간선 모두에 일치하고, `label()` 은 여전히 그 간선이 만들어질 때의 레이블을 보고합니다:

```
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)
```

없을 때만 레이블 만들기

```
CREATE ELABEL IF NOT EXISTS acted_in;
```

```
NOTICE:  label "acted_in" already exists, skipping
CREATE ELABEL
```

인덱스를 만들기 전에 간선 레이블 적재하기

인덱스 세 개가 모두 무효 상태로 만들어지고, 테이블 자체는 unlogged입니다:

```
CREATE UNLOGGED ELABEL staged DISABLE INDEX WITH (fillfactor = 70) TABLESPACE pg_default;
```


© 2026 SKAI Worldwide Co., Ltd.

```
\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
```

간선을 적재하고, 그다음 `REINDEX ELABEL staged`를 실행해 인덱스 세 개를 만드십시오.

간선 속성을 타입 컬럼으로 승격하기

```
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
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
```

간선을 쓰면 두 컬럼이 모두 채워집니다:

```
MATCH (p:person), (f:film) CREATE (p)-[:scored {rating: 8.5, site: 'critics'}]->(f);
```


© 2026 SKAI Worldwide Co., Ltd.

```
UPDATE 1
```

SQL은 그것을 컬럼 이름으로 읽습니다:

```
SELECT rating, source FROM castlists.scored;
```

```
 rating | source
--------+---------
    8.5 | critics
(1 row)
```

Cypher는 그것을 속성 키로 읽으므로, 두 번째 것은 `e.source`가 아니라 `e.site`입니다:

```
MATCH ()-[e:scored]->() RETURN e.rating AS rating, e.site AS site;
```

```
 rating |  site
--------+---------
    8.5 | critics
(1 row)
```

알아 둘 만한 거부 사례

간선 레이블은 간선 레이블만 상속할 수 있습니다:

```
CREATE ELABEL wrong INHERITS (person);
```

```
ERROR:  parent graph label "person" is not edge label.
```

마지막으로 예제가 만든 그래프를 지웁니다:

```
DROP GRAPH castlists CASCADE;
```

참고

레이블 이름은 한정할 수 없습니다. `CREATE ELABEL castlists.directed`는 구문 오류이고, 그래프는 항상 `graph_path`에서 옵니다. 설정된 것이 없으면 이 문장은 `graph_path is NULL`로 멈춥니다.

`\dGe`는 간선 레이블을 나열하고 일치하는 것마다 설명해 줍니다. `\dGl`은 두 종류의 레이블을 자세한 설명 없이 나열합니다.

호환성

ISO/IEC 39075 (GQL)

GQL에는 레이블 DDL이 없습니다. 간선은 그래프 타입 안의 **간선 타입**으로 기술되며,
`<create graph type statement>` (§12.6)에 해당합니다. 간선 타입은 그 간선의 끝점 정점 타입과 속성을 지정하고, 그것을 참조하는 그래프를 제약합니다. AgensGraph는 간선 레이블을 하나씩 선언하고, 간선 레이블은 자기 끝점도 속성도 제약하지 않습니다. 그래프 타입은 **GQL 적합성 표**에 계획으로 기록돼 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

PostgreSQL

`CREATE ELABEL` 은 `CREATE TABLE` 에 대응합니다. `UNLOGGED` , `IF NOT EXISTS` , `INHERITS` , `WITH` , `TABLESPACE` 는 거기서와 같은 뜻이고, 승격 속성 형태는 `GENERATED ALWAYS AS ... STORED` 생성 컬럼이 됩니다. 다른 점은 고정된 네 컬럼 모양, 함께 만들어지는 인덱스 세 개, `ag_label` 등록, 자동으로 붙는 부모, 그리고 순서가 중요한 옵션 배치입니다. `start` 와 `end` 는 외래 키로 선언되지 않으므로, 테이블 수준에서는 간선이 이미 지워진 정점을 가리키는 것을 막는 것이 없습니다. 그 일은 `DROP VLABEL` 이 처리합니다.

관련 항목

- `CREATE VLABEL`
- `ALTER VLABEL` · `ALTER ELABEL`
- `DROP VLABEL` · `DROP ELABEL`
- `REINDEX VLABEL` · `REINDEX ELABEL`
- 레이블 상속

ALTER VLABEL과 ALTER ELABEL

레이블의 이름을 바꾸고, 저장 방식을 바꾸고, 부모를 붙이거나 떼고, 인덱스를 끄고, 승격된 타입 컬럼을 추가하거나 제거합니다.

개요

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
NO INHERIT parent_label
REPLICA IDENTITY { **DEFAULT** | **USING INDEX** index_name | **FULL** | **NOTHING** }
OWNER TO role_specification

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

설명

`ALTER VLABEL`은 정점 레이블에, `ALTER ELABEL`은 간선 레이블에 작용합니다. 키워드를 빼면 두 문장은 같습니다. **위에 나열한 모든 동작이 두 종류 모두에서 받아들여집니다.** 다른 것은 그 동작이 작용할 대상으로 각 레이블 종류가 무엇을 갖고 있는가입니다.

- 레이블이 만들어질 때 함께 생기는 인덱스가 다릅니다. 정점 레이블에는 `id`에 대한 기본 키가 생기고, 간선 레이블에는 `(start, "end")`와 `("end", start)`에 걸린 btree 인덱스와 `id`에 걸린 BRIN 인덱스가 생깁니다. `DISABLE INDEX`는 어느 쪽이든 그 레이블의 모든 인덱스를 무효로 만들므로, 갓 만든 정점 레이블에서는 무효로 만들 것이 하나이고 간선 레이블에서는 셋입니다. `CLUSTER ON`도 고를 수 있는 이름이 다릅니다.
- `DROP COLUMN`이 거부하는 구조 컬럼은 정점 레이블에서 `id`와 `properties`이고, 간선 레이블에는 `start`와 `end`가 더해집니다.
- `INHERIT`와 `NO INHERIT`는 그 레이블과 같은 종류이고 같은 그래프에 있는 부모를 요구합니다.

레이블은 `graph_path`가 지정한 그래프에서 찾습니다. 그 이름은 한정할 수 없습니다.

- `RENAME TO`는 그 자체로 하나의 형태입니다. 다른 어떤 동작과도 함께 쓸 수 없고, 여러 동작을 나열한 목록에 들어갈 수도 없습니다.
- 한 문장의 여러 동작은 쉼표로 구분하며 적은 순서대로 적용됩니다.
- 기반 레이블인 `ag_vertex`와 `ag_edge`는 아예 변경할 수 없습니다.
- `ALTER VLABEL`은 간선 레이블을 거부하고 `ALTER ELABEL`은 정점 레이블을 거부합니다.

대신 `ALTER TABLE`로 레이블의 모양을 바꾸는 것은 `enable_graph_ddl`을 켜 두지 않으면 거부됩니다. 그래프 카탈로그가 레이블의 컬럼을 설명하는데, `ALTER TABLE`은 그것을 갱신하지 않습니다. 그 관문은 **그래프가 어떻게 저장되는가**에서 다룹니다.

매개변수

IF EXISTS	그 레이블이 없을 때 오류가 아니라 알림을 냅니다.
label_name	변경할 레이블이며, `graph_path`가 지정한 그래프 안에서 찾습니다.
RENAME TO new_label_name	레이블에 새 이름을 줍니다. 정점과 간선은 자기 id를 유지하므로, 이름을 바꾸기 전에 만든 행에 대해서도 `label()`이 새 이름을 보고합니다. 그 테이블의 시퀀스와 인덱스는 처음 받은 이름을 유지하므로, `CLUSTER ON`을 쓰기 전에 그것을 확인하십시오.
ADD [COLUMN] promoted_property	승격된 타입 컬럼을 추가하고 이미 있는 행에서 그것을 채웁니다. `promoted_property`는 `CREATE VLABEL`과 같은 세 형태를 받고 제한도 같습니다. 그 레이블에서 이미 승격된 속성은 거부되며, 한 문장에서 같은 키를 두 번 승격하는 것도 거부됩니다.
DROP [COLUMN] column_name [CASCADE | RESTRICT]	컬럼을 지우고, 승격된 속성이라면 그 대응 관계도 잊습니다. 레이블의 구조 컬럼은 거부됩니다. 지운 컬럼에 걸려 있던 인덱스도 함께 사라집니다.

© 2026 SKAI Worldwide Co., Ltd.

DISABLE INDEX	그 레이블의 모든 인덱스를 무효로 표시해, 대량 적재가 그것을 유지하지 않게 합니다. 읽기와 쓰기는 계속 동작하고 플래너는 무효 인덱스를 무시합니다. `ENABLE INDEX`는 없습니다. 되돌아가는 길은 `REINDEX VLABEL`입니다.
SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }	레이블의 `properties` 컬럼의 저장 방식을 정합니다. 이 방식이 큰 속성 맵을 인라인으로 둘지 그 테이블의 TOAST 테이블에 둘지, 그리고 압축할지를 결정합니다. 레이블은 `EXTENDED`로 시작하며, 그것은 인라인이 아니면서 압축하는 방식입니다. 이 문장은 이후 쓰기에 대한 전략을 바꿀 뿐, 이미 저장된 것을 다시 쓰지는 않습니다.
SET TABLESPACE tablespace_name	레이블의 테이블을 다른 테이블스페이스로 옮깁니다. 그 인덱스는 있던 자리에 남습니다.
SET { LOGGED | UNLOGGED }	레이블 테이블의 미리 쓰기 로깅을 켜거나 끕니다. 그 레이블의 id 시퀀스도 함께 영속성이 바뀝니다.
CLUSTER ON index_name	나중의 `CLUSTER`가 쓸 인덱스로 어떤 인덱스를 기록해 둡니다. 지금 무언가를 재배열하지는 않습니다.
SET WITHOUT CLUSTER	그 기록을 잊습니다.
INHERIT parent_label	그 레이블을 같은 그래프의 같은 종류 레이블의 자식으로 붙여, 부모에 대한 패턴이 이 레이블의 행에도 일치하기 시작하게 합니다.
NO INHERIT parent_label	그 관계를 제거합니다.
REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }	논리 복제를 위해, 쓰기가 행을 식별하려고 WAL에 기록하는 이전 값을 무엇으로 할지 고릅니다.
OWNER TO role_specification	그 레이블을 다른 롤에게 넘깁니다. 롤 이름, `CURRENT_ROLE`, `CURRENT_USER`, `SESSION_USER` 중 하나를 씁니다.

예제

예제는 **예제 데이터셋**을 쓰지 않고 자체 그래프인 `catalogue`를 만들고, 끝에서 그것을 지웁니다.

```
CREATE GRAPH catalogue;
SET graph_path = catalogue;
CREATE VLABEL film;
CREATE (:film {title: 'Arrival', year: 2016}), (:film {title: 'Dune', year: 2021});
```


© 2026 SKAI Worldwide Co., Ltd.

레이블 이름 바꾸기

```
ALTER VLABEL film RENAME TO movie;
```

```
\dGl catalogue.mov*
```

```
          List of labels
   Graph    | Name  |  Type  | Owner
------------+-------+--------+-------
 catalogue | movie | vertex | agens
(1 row)
```

행도 함께 따라왔습니다:

```
MATCH (m:movie) RETURN count(m) AS movies;
```

```
 movies
--------
 2
(1 row)
```

시퀀스와 기본 키는 여전히 옛 이름을 지니고 있습니다:

```
\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
```

그래서 `ALTER VLABEL movie CLUSTER ON movie_pkey` 는 `index "movie_pkey" for table "movie" does not exist` 로 실패하고, 써야 할 이름은 `film_pkey` 입니다.

대량 적재를 위해 레이블의 인덱스 끄기

```
ALTER VLABEL movie DISABLE INDEX;
```

```
\d catalogue.movie
```

```
                                                  Table "catalogue.movie"
   Column    |  Type    | Collation | Nullable |                                                  Defau
lt
```


© 2026 SKAI Worldwide Co., Ltd.

```
------------+---------+-----------+----------
+-----------------------------------------------------------------------------------------------
-
 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
```

행을 적재하고, 그다음 인덱스를 다시 만듭니다:

```
REINDEX VLABEL movie;
```

```
REINDEX
```

행이 이미 있는 레이블에서 속성 승격하기

```
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
```

이미 있는 행이 채워지므로, 그 문장이 돌아오는 순간 컬럼은 이미 완전합니다:

```
SELECT year FROM catalogue.movie ORDER BY year;
```

```
 year
------
 2016
 2021
(2 rows)
```

레이블이 무엇을 승격했는지 조회하기

`ag_label_property`가 속성 키를 그것을 대신해 답하는 컬럼에 대응시킵니다:

```
ALTER VLABEL movie ADD COLUMN released text GENERATED (releaseDate);
```


© 2026 SKAI Worldwide Co., Ltd.

```
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`은 그 테이블에서 컬럼의 위치입니다. 컬럼 이름까지 원한다면 그것으로 `pg_attribute`와 조인하십시오. 키를 명시하는 형태를 썼다면 그 둘이 다릅니다.

승격된 컬럼 지우기

```
ALTER VLABEL movie DROP COLUMN released;
```

그 대응 관계도 컬럼과 함께 사라집니다:

```
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)
```

`releasedate`를 읽으면 `properties`로 돌아갑니다. `properties`에서 지워진 것은 없습니다.

속성 맵의 저장 방식 바꾸기

```
ALTER VLABEL movie SET STORAGE external;
```

어느 컬럼을 예상했든, 그 방식은 `properties`에 설정됩니다:

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

부모 붙이고 떼기

```
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
```

이제 `movie`에는 부모가 둘입니다. 처음부터 있던 기반 레이블과 `work` 입니다. 떼어 내도 기반 레이블은 그대로입니다:

```
ALTER VLABEL movie NO INHERIT work;
```

```
ALTER VLABEL
```

알아 둘 만한 거부 사례

구조 컬럼은 지울 수 없습니다:

```
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.
```

추가하는 컬럼은 `properties`에서 생성되는 것이어야 합니다:

```
ALTER VLABEL movie ADD COLUMN rank int;
```

```
ERROR:  a non-generated promoted column is not supported yet
HINT:  Use GENERATED to mirror the property from the jsonb bag.
```

이미 승격된 속성을 두 번 승격할 수는 없습니다:

```
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.
```

기반 레이블은 부모로든 대상으로든 손댈 수 없습니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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
```

각 문장은 한 종류의 레이블에만 작용합니다:

```
CREATE ELABEL directed;
ALTER VLABEL directed RENAME TO helmed;
```

```
ERROR:  RENAME VLABEL cannot RENAME edge label
```

그리고 `RENAME TO`는 다른 동작과 함께 쓸 수 없습니다:

```
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;
                                         ^
```

`enable_graph_ddl`이 꺼져 있는 동안에는 SQL로 레이블의 테이블 모양을 바꾸는 것이 거부됩니다:

```
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.
```

마지막으로 예제가 만든 그래프를 지웁니다:

```
DROP GRAPH catalogue CASCADE;
```

참고

`SET STORAGE`는 다섯이 아니라 네 개의 키워드를 받습니다. `DEFAULT`는 예약어이고 여기서는 구문 오류입니다. 레이블이 처음 갖는 방식으로 돌아가려면 `SET STORAGE extended`로 쓰십시오. 알 수 없는 단어는 `invalid storage type "..."`으로 보고됩니다.

`IF EXISTS` 알림은 두 형태에서 다릅니다. `RENAME TO`는 `label "nosuch" does not exist, skipping`을, 동작 목록은 `graph label "nosuch" does not exist, skipping`을 냅니다. 그다음 둘 다 성공을 보고하고 아무것도 하지 않습니다.

© 2026 SKAI Worldwide Co., Ltd.

호환성

ISO/IEC 39075 (GQL)

GQL에는 레이블을 변경하는 문장이 없습니다. 변경할 레이블이 없기 때문입니다. 요소 타입은 그래프 타입에 속하고, 그래프 타입은 통째로 만들어지고 지워집니다(`<create graph type statement>` , §12.6). 레이블 이름 바꾸기, 속성 승격, 인덱스 끄기에 대응하는 것이 GQL에는 없습니다. 그래프 타입은 **GQL 적합성 표**에 계획으로 기록돼 있습니다.

PostgreSQL

여기 있는 모든 동작은 `ALTER TABLE` 의 동작이고, 각각 거기서와 같은 뜻이며 두 가지만 조정됐습니다. `SET STORAGE` 는 컬럼을 지정하지 않습니다. 저장 방식을 고를 만한 컬럼은 `properties` 하나뿐이므로 그 컬럼에 적용됩니다. `DISABLE INDEX` 에는 `ALTER TABLE` 대응물이 아예 없습니다. 그것은 레이블의 인덱스를 무효로 표시하는데, SQL에서는 `CREATE INDEX CONCURRENTLY` 를 중단시키는 것 말고는 그렇게 만들 방법이 없습니다.

반대 방향으로는, 레이블의 테이블에 대한 `ALTER TABLE` 이 그 모양을 바꾸는 동작에 대해서는 `enable_graph_ddl` 을 켜 두지 않으면 거부됩니다. 컬럼을 추가하거나 지우거나 타입을 바꾸는 것이 그런 동작입니다. 위의 예제가 그 거부를 보여 줍니다. 이 페이지의 동작들은 그 설정을 결코 필요로 하지 않습니다. 그래프 카탈로그를 스스로 갱신하기 때문입니다.

관련 항목

- `CREATE VLABEL`
- `CREATE ELABEL`
- `DROP VLABEL` · `DROP ELABEL`
- `REINDEX VLABEL` · `REINDEX ELABEL`
- 속성을 컬럼으로 승격하기
- 레이블 상속

DROP VLABEL과 DROP ELABEL

정점 레이블이나 간선 레이블을 지우며, `CASCADE` 를 쓰면 그 안의 행까지 지웁니다.

개요

DROP VLABEL [**IF EXISTS**] label_name [, ...] [**CASCADE** | **RESTRICT**]
DROP ELABEL [**IF EXISTS**] label_name [, ...] [**CASCADE** | **RESTRICT**]

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`DROP VLABEL` 은 정점 레이블을, `DROP ELABEL` 은 간선 레이블을 지우며, 각각 다른 종류는 거부합니다. 레이블을 지우면 그 뒤의 테이블이 지워지고, 그 id 시퀀스·인덱스·제약도 함께 지워집니다.

기본값인 `RESTRICT` 가 문제 삼는 것이 두 가지 있습니다:

- **행.** 정점이나 간선을 아직 담고 있는 레이블은 `cannot drop <label> because it is not empty.` 로 거부됩니다. PostgreSQL이 테이블에 적용하는 의존성 검사보다 엄격하고, 사람들이 여기에 걸립니다.
- **자식.** 자식 레이블이 있는 레이블은 거부되고, 메시지가 그 자식을 이름으로 부릅니다.

`CASCADE` 는 그 두 검사를 모두 건너뛰고, 무엇을 담고 있든 레이블을 지웁니다. 지워지는 범위는 다음과 같습니다:

- **정점** 레이블을 지우면 그 정점에 붙어 있던 간선이 지워지며, 그 간선의 레이블이 무엇이든 그렇습니다. 간선 레이블 자체는 남습니다.

© 2026 SKAI Worldwide Co., Ltd.

- **간선** 레이블을 지우면 그 레이블의 간선만 지워집니다. 그것이 잇고 있던 정점은 남습니다.

레이블은 `graph_path`가 지정한 그래프에서 찾습니다. 그 이름은 한정할 수 없습니다.

- 기반 레이블인 `ag_vertex`와 `ag_edge`는 지울 수 없습니다.
- 레이블 뒤의 테이블만 따로 지울 수는 없습니다. 그것에 `DROP TABLE`을 쓰면 거부되고, 오류 메시지가 이 문장을 가리킵니다.
- 레이블을 지우지 않고 그래프를 비우려면 `MATCH (n) DETACH DELETE n`을 쓰십시오.

매개변수

IF EXISTS	존재하지 않는 레이블에 대해 오류가 아니라 알림을 냅니다. 같은 문장의 다른 이름은 그대로 지워집니다.
label_name	지울 레이블의 이름입니다. 쉼표로 구분해 여러 개를 줄 수 있습니다.
CASCADE	그 레이블을 행과 자식 레이블과 함께 지웁니다. 정점 레이블이라면 그 정점에 붙어 있는 간선도 지웁니다.
RESTRICT	그 레이블이 행을 담고 있거나 자식 레이블이 있으면 거부합니다. 이것이 기본값입니다.

예제

예제는 **예제 데이터셋**을 쓰지 않고 자체 그래프인 `tidygraph`를 만들고, 끝에서 그것을 지웁니다.

```
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;
```

빈 레이블 지우기

```
DROP VLABEL draft;
```

```
DROP VLABEL
```

그 밖에 보고되는 것이 없습니다. 그 밖에 영향을 받은 것이 없기 때문입니다.

정점이 아직 남은 레이블 지우기

`film`에는 정점이 있으므로 기본 동작은 거부합니다:

```
DROP VLABEL film;
```

```
ERROR:  cannot drop film because it is not empty.
```


© 2026 SKAI Worldwide Co., Ltd.

`CASCADE` 는 그 레이블과 그 정점, 그리고 그것을 가리키던 `directed` 간선까지 가져갑니다:

```
DROP VLABEL film CASCADE;
```

```
DROP VLABEL
```

```
MATCH ()-[e:directed]->() RETURN count(e) AS directed_edges;
```

```
 directed_edges
----------------
 0
(1 row)
```

간선은 사라졌지만 그 레이블은 남아 있습니다. `directed` 는 비어 있는 채로 여전히 있습니다:

```
\dGl tidygraph.dir*
```

```
           List of labels
   Graph    |   Name   | Type | Owner
------------+----------+------+-------
 tidygraph | directed | edge | agens
(1 row)
```

부모 레이블 지우기

```
CREATE VLABEL work;
CREATE VLABEL documentary INHERITS (work);
DROP VLABEL work;
```

```
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` 는 함께 가져가는 자식을 이름으로 알려 줍니다:

```
DROP VLABEL work CASCADE;
```

```
NOTICE:  drop cascades to vlabel documentary
DROP VLABEL
```

여러 레이블 지우기, 이미 없는 것은 건너뛰기

```
CREATE VLABEL a;
CREATE VLABEL b;
DROP VLABEL IF EXISTS a, b, c;
```


© 2026 SKAI Worldwide Co., Ltd.

```
NOTICE:  vlabel "c" does not exist, skipping
DROP VLABEL
```

알아 둘 만한 거부 사례

각 문장은 한 종류의 레이블만 지웁니다:

```
DROP ELABEL person;
```

```
ERROR:  DROP ELABEL cannot DROP vertex label
```

기반 레이블은 지울 수 없습니다:

```
DROP VLABEL ag_vertex;
```

```
ERROR:  DROP base vertex label is prohibited
```

그리고 레이블 아래에서 테이블만 빼내 지울 수는 없습니다:

```
DROP TABLE tidygraph.person;
```

```
ERROR:  cannot drop table tidygraph.person because vlabel person requires it
HINT:  You can drop vlabel person instead.
```

마지막으로 예제가 만든 그래프를 지웁니다:

```
DROP GRAPH tidygraph CASCADE;
```

참고

끝점 정점이 지워진 간선은 그것과 함께 지워지며, 여기서도 `DETACH DELETE`에서도 그렇습니다. `start`와 `end`는 외래 키가 아니라 평범한 `graphid` 컬럼입니다. 이 일을 하는 것은 테이블에 걸린 참조 동작이 아니라 그래프 쓰기 경로입니다.

> **참고**
>
> `cypher_allow_unsafe_ddl`은 종류 검사와 기반 레이블 검사를 끄므로, 그것을 켜면 `DROP ELABEL`이 정점 레이블을 지우고 `ag_vertex`도 지울 수 있게 됩니다. 이 설정은 카탈로그가 이미 잘못된 그래프를 수리하기 위해 있는 것입니다. 그 밖의 경우에는 꺼 두십시오.

호환성

ISO/IEC 39075 (GQL)

GQL에는 레이블을 지우는 문장이 없습니다. 요소 타입은 그래프 타입에 속하고, 그래프 타입은 `<drop graph type statement>`(§12.7)로 통째로 지워지며, 그것을 지워도 데이터는 지워지지 않습니다. 그래프 타입은 **GQL 적합성 표**에 계획으로 기록돼 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

PostgreSQL

`DROP VLABEL` 과 `DROP ELABEL` 은 `DROP TABLE` 에 대응하고, `IF EXISTS` 와 이름 목록, `CASCADE` / `RESTRICT` 도 거기서와 같이 동작합니다. `DROP TABLE` 에 대응물이 없는 추가가 하나 있습니다. `DROP TABLE` 은 테이블에 행이 몇 개든 상관하지 않지만, 이 문장들은 `CASCADE` 를 쓰지 않으면 비어 있지 않은 레이블을 거부합니다. 그리고 정점 레이블에서는 `CASCADE` 가 **다른** 레이블의 간선까지 지웁니다. 여기서 `CASCADE` 는 "그리고 그것에 의존하는 카탈로그 객체"만이 아니라 "그리고 그것에 의존하는 그래프 데이터"로 읽으십시오.

관련 항목

- `CREATE VLABEL`
- `CREATE ELABEL`
- `ALTER VLABEL` · `ALTER ELABEL`
- `DROP GRAPH`
- 레이블 상속

CREATE PROPERTY INDEX

레이블의 속성 하나 이상에 인덱스를 만듭니다.

개요

```
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 ]
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

이 문장 뒤에 별도의 인덱스 종류가 있는 것은 아닙니다. 속성 인덱스는 레이블의 `jsonb` 컬럼인 `properties` 에서 그 속성을 읽는 표현식에 걸린 평범한 PostgreSQL 인덱스입니다. `CREATE PROPERTY INDEX ON movie (title)` 은 키가 `(properties.'title')` 인 btree를 만듭니다. `\d` 가 그렇게 보고하며, 플래너가 그 인덱스를 쓰려면 조건을 그 형태로 볼 수 있어야 합니다.

이 문장의 제한은 모두 그 표현식에서 비롯됩니다:

- **인덱싱되는 값의 타입이 `jsonb` 이므로,** 접근 방식에 `jsonb` 용 기본 연산자 클래스가 있어야 합니다. 기본값인 `btree` , 그리고 `hash` 와 `gin` 에는 있습니다. `gist` · `spgist` · `brin` 에는 없고, 그중 하나를 요구하면 `data type jsonb has no default operator class for access method "gist"` 로 실패합니다.
- **콜레이션을 적용할 수 없습니다.** `jsonb` 에는 콜레이션이 없으므로, `properties` 읽기에 `COLLATE` 를 쓰면 `collations are not supported by type jsonb` 로 실패합니다.
- **그 값은 결코 포함 컬럼이 될 수 없습니다.** `INCLUDE` 는 `expressions are not supported in included columns` 로 거부되고, 속성 읽기는 언제나 표현식이므로 `INCLUDE` 가 그것에 적용되지 않습니다.

© 2026 SKAI Worldwide Co., Ltd.

승격된 속성에 걸린 인덱스는 다릅니다. 그때는 값이 표현식이 아니라 컬럼이기 때문입니다. `\d` 는 `btree ((properties.'year'))` 가 아니라 `btree (year)` 로 보고합니다. 접근 방식, 연산자 클래스, 콜레이션은 그 컬럼 자신의 타입에만 제한되므로, `int` 컬럼에 `brin (year)` 도, `text` 컬럼에 `btree (name COLLATE "C")` 도 둘 다 됩니다. 애초에 언제 승격해야 하는지는 **속성을 컬럼으로 승격하기**가 다룹니다.

규칙이 세 가지 더 있습니다:

- 레이블은 `graph_path` 가 지정한 그래프에서 찾습니다. 그 이름은 한정할 수 없습니다.
- 부모 레이블에 걸린 인덱스는 그 레이블 자신의 행만 덮습니다. `ONLY` 를 쓰든 쓰지 않든 자식 레이블은 인덱싱되지 않습니다. **레이블 상속**을 참고하십시오.
- `IF NOT EXISTS` 는 검사할 이름이 필요하므로, 이름 없는 인덱스와 함께 쓸 수 없습니다.

매개변수

UNIQUE	인덱싱되는 값이 이미 있는 값과 중복되는 행을 거부하며, 인덱스를 만들 때 이미 있는 행도 검사합니다. 고유성 규칙을 Cypher 자신의 어법으로 표현하는 방법은 `CREATE CONSTRAINT` 를 참고하십시오.
CONCURRENTLY	그 레이블에 대한 쓰기를 잠그지 않고 인덱스를 만듭니다. 대가는 더 긴 생성 시간과 테이블 스캔 두 번입니다.
IF NOT EXISTS index_name	그 이름의 릴레이션이 이미 있을 때 오류가 아니라 알림을 냅니다. 아무것도 만들어지지 않고, 이미 있는 인덱스가 요구한 그 인덱스인지 확인하는 것도 없습니다.
index_name	인덱스의 이름입니다. 생략하면 레이블과 인덱싱한 속성에서 이름이 유도됩니다. `movie` 와 `year` 는 `movie_year_idx` 가 됩니다.
ONLY	받아들여지지만 평범한 레이블에는 효과가 없습니다. 파티션으로 재귀하는 것을 막는 것인데, 레이블은 파티션된 테이블이 아니라 상속 부모입니다.
label_name	행을 인덱싱할 레이블이며, `graph_path` 가 지정한 그래프 안에서 찾습니다.
USING method	인덱스 접근 방식입니다. `properties` 에서 읽는 속성에 대해서는 기본값인 `btree` , `hash` , `gin` 중에서 고릅니다. 승격된 컬럼에 대해서는 그 컬럼 타입에 기본 연산자 클래스가 있는 방식이면 무엇이든 됩니다.
property	그 레이블의 속성이며, Cypher가 쓰는 방식으로 적습니다. `title` 이거나, 중첩된 맵 안의 키라면 `name.first` 처럼 적습니다.
(expression)	그 레이블의 속성에 대한, 괄호로 감싼 Cypher 표현식입니다. 예를 들어 `(year / 10)` 입니다.

© 2026 SKAI Worldwide Co., Ltd.

COLLATE collation	인덱싱되는 값의 콜레이션입니다. 그 값이 콜레이션을 가질 수 있는 타입의 승격된 컬럼일 때만 쓸 수 있습니다.
opclass	인덱싱되는 값의 연산자 클래스입니다. 예를 들어 `gin` 인덱스에는 `jsonb_path_ops` 입니다. 연산자 클래스 매개변수는 받아들여지지 않습니다.
ASC | DESC | NULLS FIRST | NULLS LAST	인덱스에 저장되는 정렬 순서입니다. 기본값은 `ASC` 와 `NULLS LAST` 이고, `DESC` 아래에서는 `NULLS FIRST` 가 기본값입니다.
NULLS [NOT] DISTINCT	고유 인덱스에서 null 둘을 서로 다른 값으로 셀지 여부입니다. 기본값인 `NULLS DISTINCT` 는 인덱스가 null을 여러 개 담을 수 있게 합니다.
WITH (storage_parameter [= value] [, ...])	인덱스의 저장 매개변수입니다. btree의 `fillfactor` 나 gin의 `fastupdate` 같은 것입니다.
TABLESPACE tablespace_name	인덱스의 테이블스페이스입니다. 이것이 없으면 `default_tablespace` 가 결정합니다.
WHERE condition	부분 인덱스를 만듭니다. 그 조건이 받아들이는 행만 덮습니다.

예제

예제는 **예제 데이터셋**을 쓰지 않고 자체 그래프인 `indexdemo` 를 만들고, 끝에서 그것을 지웁니다.

```
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'});
```

속성에 인덱스 걸기

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

```
CREATE PROPERTY INDEX
```

이것이 표현식에 걸린 btree를 만들었습니다:

© 2026 SKAI Worldwide Co., Ltd.

```
\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` 는 같은 인덱스를 그것을 다시 만들 문장의 형태로 나열합니다:

```
\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)
```

플래너는 `title` 에 대한 같음 비교를 그 표현식과 맞춥니다. 정점 여섯 개는 플래너가 인덱스 스캔을 선호하기에는 턱없이 적으므로, 아래 설정으로 그 계획을 보이게 합니다:

```
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;
```

인덱스 이름을 자동으로 정하기

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

```
\dGi indexdemo.movie_year_idx
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                                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)
```

접근 방식 고르기

속성 읽기에 대해 btree의 대안은 `gin`과 `hash` 입니다:

```
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
```

나머지 세 접근 방식은 AgensGraph와 무관한 이유로 안 됩니다. `jsonb` 용 기본 연산자 클래스가 없기 때문입니다:

```
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`와 `brin`도 같은 방식으로 실패합니다.

중복 값 거부하기

```
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.
```

그 메시지의 `Key`는 인덱스 표현식이며, 그것으로 고유 속성 인덱스를 컬럼에 걸린 고유성 제약과 구별합니다.

레이블의 일부에 인덱스 걸기

```
CREATE PROPERTY INDEX movie_recent_idx ON movie (title) WHERE (year > 2010);
```

```
CREATE PROPERTY INDEX
```

속성 여러 개와 표현식에 인덱스 걸기

```
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);
```


© 2026 SKAI Worldwide Co., Ltd.

그 하나하나가 표현식 인덱스이며, `\d` 가 무엇이 만들어졌는지 정확히 보여 줍니다. 산술 인덱스가 지니고 있는 Cypher 연산자까지 보여 줍니다:

```
\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
```

승격된 속성에 인덱스 걸기

속성이 컬럼으로 승격된 레이블은 그 대신 컬럼에 인덱스를 겁니다:

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

`btree ((properties.'year'))` 가 아니라 `btree (year)` 입니다. 실행 계획도 같게 읽히며, `jsonb` 가 아니라 정수를 비교합니다:

```
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;
```

인덱싱되는 값이 컬럼이므로, 콜레이션과 연산자 클래스, 그리고 그 컬럼 타입이 지원하는 접근 방식이 모두 쓸 수 있게 됩니다:

```
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
-------------+---------+-----------+----------
+-------------------------------------------------------------------------------------------------
--
 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
```

그 셋 중 어느 것도 `properties` 에서 읽는 속성에 대해서는 받아들여지지 않습니다.

알아 둘 만한 거부 사례

`INCLUDE` 는 승격 컬럼이든 아니든 아예 쓸 수 없습니다. 속성은 언제나 표현식으로 적히기 때문입니다:

```
CREATE PROPERTY INDEX title_name_incl ON title (name) INCLUDE (year);
```

```
ERROR:  expressions are not supported in included columns
```


© 2026 SKAI Worldwide Co., Ltd.

연산자 클래스 매개변수는 이 문장의 일부가 아닙니다:

```
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...
                                                             ^
```

중첩 경로는 그 속성이 맵을 담고 있을 때만 됩니다. 여기서 `person.name`은 문자열이므로, 그 안으로 들어가려 하면 인덱스를 만드는 중에 실패합니다:

```
CREATE PROPERTY INDEX person_first_idx ON person (name.first);
```

```
ERROR:  map or list is expected but scalar value
```

그리고 이미 쓰이고 있는 이름은 교체되는 것이 아니라 보고됩니다:

```
CREATE PROPERTY INDEX IF NOT EXISTS movie_title_idx ON movie (title);
```

```
NOTICE:  relation "movie_title_idx" already exists, skipping
CREATE PROPERTY INDEX
```

마지막으로 예제가 만든 그래프를 지웁니다:

```
DROP GRAPH indexdemo CASCADE;
```

참고

`CONCURRENTLY`는 받아들여지고 `CREATE INDEX`에서와 같은 일을 합니다. 쓰기를 잠그지 않고 만듭니다. `ONLY`도 받아들여지지만 평범한 레이블에서는 아무것도 바꾸지 않습니다. 부모에 걸린 인덱스는 어느 쪽이든 부모의 행만 덮으므로, 계층에서는 스캔될 레이블마다 인덱스가 하나씩 필요합니다.

승격된 키가 여기서 어떻게 해석되는가

승격된 속성이 있는 레이블에서는 `( ... )`의 두 형태가 다르게 해석됩니다:

적는 것	인덱싱되는 것
맨 키를 쓴 `( year )`	승격된 컬럼 `year`
표현식을 쓴 `( (lower(title)) )`	jsonb `properties`에 대한 `lower(properties.'title')`

승격된 키에 대한 쿼리는 두 경우 모두 컬럼을 읽으므로, 쿼리가 쓸 수 있는 인덱스를 주는 것은 맨 키 형태입니다. 승격된 키에 대한 표현식에 인덱스를 걸려면 그 컬럼에 대한 SQL로 쓰십시오 — `CREATE INDEX ON g.film (lower(title))` — 그것도 `\dGi`가 여전히 나열해 줍니다.

© 2026 SKAI Worldwide Co., Ltd.

`\dGi`는 자기 `Indexdef` 컬럼을 Cypher 형태로 렌더링하므로 둘 다 키를 같게 표기합니다. 어떤 인덱스가 둘 중 무엇에 걸려 있는지 보려면 `pg_get_indexdef`를 읽거나 레이블 테이블에 `\d`를 쓰십시오. 거기서 맨 승격 키는 `btree (year)`로, 속성 형태는 `btree ((properties.'year'))`로 읽힙니다.

호환성

ISO/IEC 39075 (GQL)

GQL에는 인덱스 DDL이 없습니다. 표준의 카탈로그 변경 문장은 스키마·그래프·그래프 타입(§12)에 한정됩니다. 거기에는 인덱스의 표준 표기가 없으므로, `CREATE PROPERTY INDEX`는 AgensGraph 자신의 것입니다.

PostgreSQL

`CREATE PROPERTY INDEX`는 표현식에 대한 `CREATE INDEX`에 대응하고, `UNIQUE`, `CONCURRENTLY`, `USING`, `COLLATE`, 연산자 클래스, 정렬 순서, `NULLS [NOT] DISTINCT`, `WITH`, `TABLESPACE`, `WHERE`는 모두 거기서와 같은 뜻입니다. 다른 점이 세 가지 중요합니다. 인덱스 대상을 컬럼이나 괄호로 감싼 SQL 표현식이 아니라 Cypher 속성으로 적으므로, 파서가 `properties` 읽기를 대신 만들어 줍니다. `CREATE INDEX`가 받는 `INCLUDE`와 연산자 클래스 매개변수는 쓸 수 없습니다. 그리고 **`DROP PROPERTY INDEX`**는 속성 인덱스만 받아들이는 반면, `DROP INDEX`는 레이블의 어떤 인덱스든, 속성 인덱스를 포함해 받아들입니다.

관련 항목

- **`DROP PROPERTY INDEX`**
- **`REINDEX VLABEL`** · **`REINDEX ELABEL`**
- **`CREATE CONSTRAINT`**
- 속성을 컬럼으로 승격하기
- 그래프가 어떻게 저장되는가

DROP PROPERTY INDEX

속성 인덱스를 지웁니다.

개요

> **DROP PROPERTY INDEX** [**IF EXISTS**] index_name [, ...] [**CASCADE** | **RESTRICT**]

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`DROP PROPERTY INDEX`는 **`CREATE PROPERTY INDEX`**가 만든 인덱스 하나 이상을 지우며, 그 밖에는 아무것도 지우지 않습니다.

- 이름은 `graph_path`가 지정한 그래프에서 해석되며 **한정할 수 없습니다.** 스키마를 붙인 이름은 `improper property index name`으로 거부됩니다. 다른 그래프의 인덱스를 지우려면 먼저 `graph_path`를 바꾸십시오.
- 속성 인덱스만 받아들여집니다. 그 레이블의 다른 인덱스는 `"<name>" is not property index`로 거부됩니다. 기본 키, 간선 레이블의 끝점 인덱스, SQL로 만든 인덱스가 그렇습니다.
- 반면 `DROP INDEX`는 레이블의 어떤 인덱스든, 속성 인덱스를 포함해 받아들이므로, 어느 문장으로 만든 인덱스든 어느 문장으로도 지울 수 있습니다.
- 제약에 속한 인덱스는 **`DROP CONSTRAINT`**로 그 제약과 함께 지워집니다.

© 2026 SKAI Worldwide Co., Ltd.

매개변수

IF EXISTS	존재하지 않는 인덱스에 대해 오류가 아니라 알림을 냅니다. 같은 문장의 다른 이름은 그대로 지워집니다.
index_name	지울 속성 인덱스의 한정하지 않은 이름입니다. 쉼표로 구분해 여러 개를 줄 수 있고, 같은 레이블에 있어야 하는 것은 아닙니다.
CASCADE	그 인덱스에 의존하는 객체를 지웁니다.
RESTRICT	그 인덱스에 의존하는 것이 있으면 거부합니다. 이것이 기본값입니다. 인덱스에 의존하는 것은 드물어서 대개 두 선택이 같게 동작합니다.

예제

예제는 **예제 데이터셋**을 쓰지 않고 자체 그래프인 `unindexed`를 만들고, 끝에서 그것을 지웁니다.

```
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);
```

```
\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)
```

인덱스 하나 지우기

```
DROP PROPERTY INDEX movie_title_idx;
```

```
DROP PROPERTY INDEX
```

두 번째 시도는 오류입니다. 그 인덱스가 이미 사라졌기 때문입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
DROP PROPERTY INDEX movie_title_idx;
```

```
ERROR:  index "movie_title_idx" does not exist
```

`IF EXISTS` 는 그것을 알림으로 바꿉니다:

```
DROP PROPERTY INDEX IF EXISTS movie_title_idx;
```

```
NOTICE:  index "movie_title_idx" does not exist, skipping
DROP PROPERTY INDEX
```

여러 인덱스를 한 번에 지우기

그 이름들이 같은 레이블에 속해야 하는 것은 아닙니다:

```
DROP PROPERTY INDEX movie_year_idx, person_name_idx;
```

```
DROP PROPERTY INDEX
```

```
\dGi unindexed.*
```

```
No matching property indexes found.
```

알아 둘 만한 거부 사례

이름에 그래프를 붙일 수 없습니다:

```
CREATE PROPERTY INDEX movie_title_idx ON movie (title);
DROP PROPERTY INDEX unindexed.movie_title_idx;
```

```
ERROR:  improper property index name
```

이 문장이 소유하지 않은 인덱스는 거부됩니다. 레이블의 기본 키가 그런 것 중 하나입니다:

```
DROP PROPERTY INDEX movie_pkey;
```

```
ERROR:  "movie_pkey" is not property index
```

`DROP INDEX` 도 속성 인덱스를 받아들입니다. 스키마를 붙인 이름으로 씁니다:

```
DROP INDEX unindexed.movie_title_idx;
```

```
DROP INDEX
```


© 2026 SKAI Worldwide Co., Ltd.

마지막으로 예제가 만든 그래프를 지웁니다:

```
DROP GRAPH unindexed CASCADE;
```

참고

이름이 `graph_path`에서 해석되므로, 다른 그래프에 있는 인덱스는 남의 것이라고 나오는 것이 아니라 없는 것으로 읽힙니다:

```
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
```

두 그래프가 같은 이름의 인덱스를 가질 수 있으므로, 한정하지 않은 이름이 어느 것에 닿을지 짐작하기 전에 `\dGi`로 확인하십시오.

```
DROP GRAPH oneplace CASCADE;
DROP GRAPH otherplace CASCADE;
```

예컨대 대량 적재를 위해 레이블의 인덱스를 지우지 않고 끄려면, 대신 `ALTER VLABEL ... DISABLE INDEX`와 `REINDEX VLABEL`을 쓰십시오. 그러면 정의는 남고 무효 표시만 됩니다.

호환성

ISO/IEC 39075 (GQL)

GQL에는 인덱스 DDL이 없습니다. 표준의 카탈로그 변경 문장은 스키마·그래프·그래프 타입(§12)에 한정되므로, `DROP PROPERTY INDEX`는 AgensGraph 자신의 것입니다.

PostgreSQL

`DROP PROPERTY INDEX`는 `DROP INDEX`에 대응하고, `IF EXISTS`와 이름 목록, `CASCADE` / `RESTRICT`도 거기서와 같이 동작합니다. 다른 점이 세 가지입니다. 이름은 `search_path`가 아니라 `graph_path`에서 해석되며 스키마로 한정할 수 없습니다. `CONCURRENTLY`는 받아들여지지 않습니다. 그리고 두 문장은 엄격히 갈려 있습니다. 각각 상대의 인덱스를 거부하므로, 속성 인덱스는 이 문장으로만 지울 수 있습니다.

관련 항목

- `CREATE PROPERTY INDEX`
- `REINDEX VLABEL` · `REINDEX ELABEL`
- `DROP CONSTRAINT`
- 그래프가 어떻게 저장되는가

REINDEX VLABEL과 REINDEX ELABEL

레이블의 모든 인덱스를 다시 만듭니다. 인덱스를 꺼 둔 레이블이 그것을 되찾는 방법도 이것입니다.

© 2026 SKAI Worldwide Co., Ltd.

개요

REINDEX [(option [, ...])] **VLABEL** label_name
REINDEX [(option [, ...])] **ELABEL** label_name

where option is one of:

CONCURRENTLY [boolean]
TABLESPACE new_tablespace
VERBOSE [boolean]

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`REINDEX VLABEL` 은 정점 레이블의 모든 인덱스를, `REINDEX ELABEL` 은 간선 레이블의 모든 인덱스를 다시 만듭니다. 그 레이블 자신의 인덱스, 거기 걸린 모든 속성 인덱스, 그리고 그 TOAST 테이블의 인덱스까지 포함합니다. 다시 만들면 무효 상태였던 인덱스가 다시 유효로 표시되므로, 이 문장이 `DISABLE INDEX` 에 대응하는 것이 됩니다.

`ENABLE INDEX` 는 없습니다. 적재하고 나서 만드는 순서는 `ALTER VLABEL <label> DISABLE INDEX` , 그다음 적재, 그다음 `REINDEX VLABEL <label>` 입니다.

- 레이블은 `graph_path` 가 지정한 그래프에서 찾습니다. 그 이름은 **한정할 수 없습니다.** `REINDEX VLABEL rebuildme.movie` 는 구문 오류입니다.
- 각 문장은 한 종류의 레이블만 받습니다. 간선 레이블에 `REINDEX VLABEL` 을 쓰면 `REINDEX VLABEL cannot REINDEX edge label` 로 실패하고, 반대의 경우도 그에 맞는 메시지로 실패합니다.
- 옵션은 레이블 키워드 앞의 괄호 안에 놓습니다. `REINDEX TABLE` 은 키워드만 쓰는 표기를 받지만, 여기서 `REINDEX VLABEL CONCURRENTLY movie` 는 구문 오류입니다.
- 옛 표기인 `REINDEX LABEL` 은 없습니다. `LABEL` 자리에서 구문 오류입니다.

매개변수

option	`CONCURRENTLY` , `TABLESPACE` , `VERBOSE` 중 하나이며, 괄호 안에 쉼표로 구분해 씁니다. `REINDEX` 에서와 같은 뜻입니다. 쓰기를 잠그지 않고 만들기, 다시 만든 인덱스를 다른 테이블스페이스로 옮기기, 인덱스가 끝날 때마다 보고하기입니다. `CONCURRENTLY` 와 `VERBOSE` 는 불리언을 명시적으로 받기도 하므로, `(VERBOSE false)` 는 `VERBOSE` 를 쓰지 않은 것과 같습니다.
label_name	인덱스를 다시 만들 레이블이며, `graph_path` 가 지정한 그래프 안에서 찾습니다.

예제

예제는 **예제 데이터셋**을 쓰지 않고 자체 그래프인 `rebuildme` 를 만들고, 끝에서 그것을 지웁니다.

© 2026 SKAI Worldwide Co., Ltd.

```
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'});
```

대량 적재 뒤에 레이블의 인덱스 만들기

인덱스를 끄면 인덱스는 자리에 남지만 무효 상태가 되므로, 적재가 그것을 유지하지 않습니다:

```
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
```

두 인덱스 모두 `INVALID`로 표시됩니다. 레이블의 기본 키와 속성 인덱스 모두입니다. 문장 하나가 그것을 만듭니다:

```
REINDEX VLABEL movie;
```

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

인덱스가 다시 만들어지는 것을 하나씩 보기

```
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
```

간선 레이블의 인덱스 세 개가 모두 다시 만들어지고, 지나치게 큰 속성 맵을 담는 TOAST 테이블의 인덱스도 그렇습니다. TOAST 인덱스 이름에는 그 테이블의 OID가 들어가므로 실행마다 달라집니다.

쓰기를 잠그지 않고 다시 만들기

```
REINDEX (CONCURRENTLY) VLABEL movie;
```

```
REINDEX
```

```
REINDEX (TABLESPACE pg_default) VLABEL movie;
```

```
REINDEX
```

알아 둘 만한 거부 사례

각 문장은 한 종류의 레이블만 받습니다:

```
REINDEX VLABEL directed;
```

```
ERROR:  REINDEX VLABEL cannot REINDEX edge label
```

```
REINDEX ELABEL movie;
```

```
ERROR:  REINDEX ELABEL cannot REINDEX vertex label
```

`REINDEX LABEL` 은 없습니다:

```
REINDEX LABEL movie;
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  syntax error at or near "LABEL"
LINE 1: REINDEX LABEL movie;
                ^
```

그 이름에 그래프를 붙일 수는 없습니다:

```
REINDEX VLABEL rebuildme.movie;
```

```
ERROR:  syntax error at or near "."
LINE 1: REINDEX VLABEL rebuildme.movie;
                                 ^
```

그리고 `CONCURRENTLY` 는 괄호 안에 놓입니다:

```
REINDEX VLABEL CONCURRENTLY movie;
```

```
ERROR:  syntax error at or near "CONCURRENTLY"
LINE 1: REINDEX VLABEL CONCURRENTLY movie;
                       ^
```

없는 레이블은 한정된 이름으로 보고됩니다. `graph_path` 가 이미 그것에 적용됐기 때문입니다:

```
REINDEX VLABEL nosuch;
```

```
ERROR:  relation "rebuildme.nosuch" does not exist
```

마지막으로 예제가 만든 그래프를 지웁니다:

```
DROP GRAPH rebuildme CASCADE;
```

참고

레이블의 테이블에 `REINDEX TABLE` 을 쓰면 같은 일을 하고 그러기 위해 어떤 설정도 필요하지 않으므로, `REINDEX TABLE rebuildme.movie` 도 꺼 둔 레이블의 인덱스를 되살립니다. 그래도 레이블 쪽 표기를 쓰는 편이 낫습니다. 그 레이블이 이름으로 부른 종류가 맞는지 검사하고, 스키마로 한정한 이름이 아니라 `graph_path` 에서 해석한 이름을 받기 때문입니다.

그래프의 스키마에 `REINDEX SCHEMA` 를 쓰면 그 그래프의 모든 인덱스가 다시 만들어지고, `REINDEX INDEX` 는 이름으로 하나를 다시 만듭니다. 둘 다 평범한 SQL 문장이고 레이블을 아는 형태는 없습니다.

호환성

ISO/IEC 39075 (GQL)

GQL에는 인덱스 DDL도, 인덱스를 다시 만드는 것도 없습니다. 표준의 카탈로그 변경 문장은 스키마·그래프·그래프 타입(§12)에 한정되므로, 이 두 표기는 AgensGraph 자신의 것입니다.

PostgreSQL

`REINDEX VLABEL` 과 `REINDEX ELABEL` 은 대상이 레이블인 `REINDEX TABLE` 입니다. 옵션은 `REINDEX` 자신의 옵션이고 뜻도 같습니다. 표기에서 두 가지 차이가 중요합니다. 대상 이름은 `graph_path` 에서 해석되며 스키마로 한정할 수 없습니다. 그리고 `CONCURRENTLY` 는 괄호 안의 옵션으로만 받아들여지고, 레이블 키워드 뒤에 키워드만 쓰는 형태로는 결코 받아들여지지 않습니다.

© 2026 SKAI Worldwide Co., Ltd.

관련 항목

- `ALTER VLABEL` · `ALTER ELABEL`
- `CREATE PROPERTY INDEX`
- `DROP PROPERTY INDEX`
- `CREATE VLABEL`

CREATE CONSTRAINT

레이블의 어떤 속성이 고유하도록 요구하거나, 그 레이블의 모든 요소가 어떤 조건을 만족하도록 요구합니다.

개요

CREATE CONSTRAINT [constraint_name] **ON** label_name **ASSERT** property **IS UNIQUE**
CREATE CONSTRAINT [constraint_name] **ON** label_name **ASSERT** condition

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

형태가 두 가지이고, 디스크에서는 서로 다른 두 가지가 됩니다.

`ASSERT property IS UNIQUE` 는 그 속성 읽기에 대한 배제 제약이 됩니다.
`EXCLUDE USING btree ((properties.'title') WITH =)` 입니다. 값이 이미 있는 쓰기는 충돌로 거부됩니다.

`ASSERT condition` 은 같은 종류의 읽기에 대한 검사 제약이 됩니다. `CHECK ((properties.'age')::integer > 9)` 입니다. 그 조건을 만족하지 못하는 행을 쓰는 것은 거부됩니다.

두 형태 모두 이름은 생략할 수 있습니다. 생략하면 이름이 유도됩니다. 고유성 형태는 `<label>_unique_constraint` , 검사 형태는 `<label>_properties_check` 입니다. 그 이름이 이미 쓰이고 있으면 숫자 접미사가 붙습니다. 직접 주는 이름은 그래프의 스키마에서 아직 쓰이지 않은 이름이어야 합니다. 고유성 형태가 그 이름의 인덱스를 만들기 때문입니다.

그 밖의 규칙입니다:

- 레이블은 `graph_path` 가 지정한 그래프에서 찾습니다. 그 이름은 한정할 수 없습니다.
- 두 형태 모두 정점 레이블과 간선 레이블에서 동작합니다.
- `CREATE OR REPLACE CONSTRAINT` 는 파싱된 뒤 `CREATE OR REPLACE CONSTRAINT is not supported` 로 거부됩니다. 제약을 바꾸려면 지우고 다시 만드십시오.
- 제약은 그 레이블에 이미 있는 행에 대해 만들어지므로, 그것을 위반하는 행이 있으면 문장이 실패합니다.
- 검사 제약은 SQL의 검사 제약이므로, **없는 속성은 그 제약을 만족합니다.** 없는 속성에 대해 조건은 알 수 없음으로 평가되고, 검사 제약은 알 수 없음을 받아들입니다. 이 제약은 키가 반드시 있게 하는 데 쓰는 것이 아니라 값을 제한하는 데 쓰십시오.

디스크에서의 대응 관계는 **그래프가 어떻게 저장되는가**에서 함께 다룹니다.

매개변수

constraint_name	그 제약의 이름입니다. 생략할 수 있고, 생략하면 레이블과 형태에서 이름이 유도됩니다. 고유성 형태에서는 그 이름이 인덱스 이름도 되므로, 그래프에 이미 있는 릴레이션과 충돌해서는 안 됩니다.

© 2026 SKAI Worldwide Co., Ltd.

label_name	그 제약이 적용되는 레이블이며, `graph_path`가 지정한 그래프 안에서 찾습니다.
property	고유해야 하는 속성이며, Cypher가 쓰는 방식으로 적습니다. `title`이거나, 중첩된 맵 안의 키라면 `name.first`처럼 적습니다.
condition	그 레이블의 속성에 대한 Cypher 조건입니다. 기록되는 요소마다 한 번 평가됩니다.

예제

예제는 **예제 데이터셋**을 쓰지 않고 자체 그래프인 `constrain`을 만들고, 끝에서 그것을 지웁니다.

```
CREATE GRAPH constrain;
SET graph_path = constrain;
CREATE VLABEL film;
CREATE ELABEL rated;
```

속성이 고유하도록 요구하기

```
CREATE CONSTRAINT film_title_uniq ON film ASSERT title IS UNIQUE;
```

```
CREATE CONSTRAINT
```

이것이 그 속성 읽기에 대한 배제 제약을 만들었습니다:

```
\d constrain.film
```

```
                                                    Table "constrain.film"
   Column   |  Type   | Collation | Nullable |                                        Defau
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
```

첫 쓰기는 들어갑니다:

```
CREATE (:film {title: 'Arrival', year: 2016});
```

```
UPDATE 1
```

두 번째는 거부되고, 메시지가 그 제약과 값을 이름으로 알려 줍니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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").
```

제약에 이름을 붙이지 않기

```
CREATE CONSTRAINT ON film ASSERT year IS UNIQUE;
```

`\dGv` 는 두 제약을 적힌 대로 보여 주고, 두 번째 것에 주어진 유도된 이름도 보여 줍니다:

```
\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
```

속성이 담을 수 있는 것 제한하기

```
CREATE VLABEL teen;
CREATE CONSTRAINT teen_age ON teen ASSERT (10 <= age::int AND age::int <= 19);
```

```
\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
```

범위 안의 행은 기록됩니다:

© 2026 SKAI Worldwide Co., Ltd.

```
CREATE (:teen {name: 'Kim', age: 15});
```

```
UPDATE 1
```

범위 밖의 행은 거부됩니다:

```
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"}).
```

`age` 가 **아예 없는** 정점은 받아들여집니다. 조건이 거짓이 아니라 알 수 없음이기 때문입니다:

```
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)
```

형 변환 없이 비교하기

속성을 그대로 두고 비교를 적으면 정수가 아니라 `jsonb` 값끼리 비교하게 됩니다:

```
CREATE CONSTRAINT ON teen ASSERT age > 9;
```

```
\dGv constrain.teen
```

```
          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
```

두 형태 모두 `{age: 'twelve'}` 를 거부하지만, 보고 방식이 다릅니다. 형 변환한 형태는 `invalid input syntax for type integer: "twelve"` 를 냅니다. 형 변환하지 않은 형태는 검사 제약 위반을 보고합니다. 속성의 범위뿐 아니라 타입까지 강제하고 싶다면 형 변환을 하십시오.

© 2026 SKAI Worldwide Co., Ltd.

간선 레이블 제약하기

```
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
```

그 제약은 그 레이블의 평범한 인덱스 세 개 옆에 놓입니다.

행이 이미 있는 레이블에 제약 추가하기

제약은 이미 있는 것에 대해 검사되므로, 그것을 깨뜨리는 행 위에는 만들 수 없습니다. 두 형태가 그것을 다르게 보고합니다:

```
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").
```

```
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
```

알아 둘 만한 거부 사례

교체는 없습니다:

```
CREATE OR REPLACE CONSTRAINT film_title_uniq ON film ASSERT title IS UNIQUE;
```

```
ERROR:  CREATE OR REPLACE CONSTRAINT is not supported
```


© 2026 SKAI Worldwide Co., Ltd.

고유성 제약의 이름은 인덱스 이름이므로 인덱스와 충돌합니다:

```
CREATE CONSTRAINT film_title_uniq ON film ASSERT year IS UNIQUE;
```

```
ERROR:  relation "film_title_uniq" already exists
```

그리고 그 레이블이 존재해야 합니다. 여기서는 `CREATE (:film)`이 하는 것처럼 Cypher가 레이블을 만들어 주지 않습니다:

```
CREATE CONSTRAINT c ON nosuch ASSERT a IS UNIQUE;
```

```
ERROR:  label "nosuch" does not exist
```

마지막으로 예제가 만든 그래프를 지웁니다:

```
DROP GRAPH constrain CASCADE;
```

참고

고유성 제약은 고유 인덱스가 아니라 배제 제약이므로 `CREATE UNIQUE PROPERTY INDEX`와 바꿔 쓸 수 없습니다. 둘 다 중복을 거부하지만 오류가 다릅니다. `conflicting key value violates exclusion constraint` 대 `duplicate key value violates unique constraint` 입니다. `\dGv`가 보고하는 것도 다릅니다. 그리고 그것을 지우는 문장도 다릅니다. 제약은 **`DROP CONSTRAINT`**로, 고유 인덱스는 **`DROP PROPERTY INDEX`**로 지웁니다.

두 형태는 **레이블 상속**에서 다르고, 그 차이에 사람들이 걸립니다. 검사 제약은 상속됩니다. 그 뒤에 만든 자식 레이블이 그 제약을 지니고 그것으로 검사됩니다. 고유성 제약은 그렇지 않습니다. 자식 레이블에는 배제 제약이 생기지 않으므로, 같은 값을 부모를 통해 한 번, 자식을 통해 한 번 쓸 수 있습니다. 그러면 부모에 대한 `MATCH`가 둘 다 반환합니다.

> 참고
>
> 계층 전체에서 속성을 고유하게 만들려면 그 계층의 레이블마다 제약을 만드십시오. 여러 레이블에 걸쳐 값 하나를 고유하게 만드는 장치는 없습니다. 제약은 그 뒤의 인덱스와 마찬가지로 테이블 하나를 덮습니다.

호환성

ISO/IEC 39075 (GQL)

GQL에는 제약 DDL이 없습니다. 표준은 그래프 타입(`<create graph type statement>`, §12.6)으로 그래프를 제약하며, 그 그래프 타입이 그래프의 요소가 어떤 레이블과 속성을 가질 수 있는지 선언합니다. 그래프 타입에는 속성별 고유성 규칙도, 검사 조건도 없습니다. 그리고 그래프가 그래프 타입을 참조할 때만 제약을 받습니다. AgensGraph의 그래프는 열려 있으므로 `CREATE CONSTRAINT`는 AgensGraph 자신의 것입니다. 그래프 타입은 **GQL 적합성 표**에 계획으로 기록돼 있습니다.

PostgreSQL

검사 형태는 `ALTER TABLE ... ADD CONSTRAINT ... CHECK`에, 고유성 형태는 `ALTER TABLE ... ADD CONSTRAINT ... EXCLUDE USING btree (... WITH =)`에 대응합니다. 표현식에는 쓸 수 없는 `UNIQUE`에는 대응하지 않습니다. 조건은 SQL 표현식이 아니라 Cypher 표현식이므로, `age`는 컬럼이 아니라 속성을 뜻합니다. `ADD CONSTRAINT`가 제공하는 `NOT NULL`, `PRIMARY KEY`, `FOREIGN KEY`, `DEFERRABLE`, `NOT VALID`는 여기에 표기가 없습니다.

관련 항목

- **`DROP CONSTRAINT`**
- **`CREATE PROPERTY INDEX`**
- **`CREATE VLABEL`**

© 2026 SKAI Worldwide Co., Ltd.

- 그래프가 어떻게 저장되는가

DROP CONSTRAINT

레이블에서 제약을 지웁니다.

개요

```
DROP CONSTRAINT constraint_name ON label_name
```

굵게 키워드 · 기울임 직접 채우는 자리 · `[ ]` 생략 가능 · `...` 반복 가능. 구문 표기 규약을 참고하십시오.

설명

`DROP CONSTRAINT`는 레이블 하나에서 제약 하나를 지웁니다. `CREATE CONSTRAINT`가 만드는 두 형태 모두를 받습니다. `IS UNIQUE`로 만든 배제 제약과, 조건으로 만든 검사 제약입니다. 고유성 제약이 소유한 인덱스도 함께 지웁니다.

이 문장은 의도적으로 작습니다. 이 문장에 없는 것들입니다:

- **이름이 필수입니다.** "이 속성에 걸린 제약이 무엇이든"이라고 말할 방법이 없습니다.
- **`IF EXISTS`가 없습니다.** `DROP CONSTRAINT IF EXISTS ...`는 구문 오류이므로, 없는 제약은 언제나 오류입니다.
- **`CASCADE`도 `RESTRICT`도 없고,** 이름 목록도 없습니다.
- 그 이름은 한정할 수 없고, 레이블 이름도 그렇습니다. 레이블은 `graph_path`가 지정한 그래프에서 찾습니다.
- Cypher가 만든 것만이 아니라, 레이블 테이블의 이름 있는 모든 제약에 닿습니다. 레이블의 기본 키도 그중 하나입니다. 실행하기 전에 그 이름을 읽으십시오.
- 속성 인덱스는 제약이 아닙니다. 그것은 `DROP PROPERTY INDEX`로 지우십시오.

매개변수

constraint_name	지울 제약의 이름이며 한정하지 않습니다. `\dGv` · `\dGe` · `\d`가 모두 그것을 보고하고, `pg_constraint`도 그렇습니다.
label_name	그 제약이 걸린 레이블이며, `graph_path`가 지정한 그래프 안에서 찾습니다.

예제

예제는 예제 데이터셋을 쓰지 않고 자체 그래프인 `rulebooks`를 만들고, 끝에서 그것을 지웁니다.

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

고유성 제약 지우기

```
DROP CONSTRAINT film_title_uniq ON film;
```

```
DROP CONSTRAINT
```

그 제약과 그 인덱스가 둘 다 사라졌습니다. 그 뒤에 만든, 이름 없는 것은 그대로입니다:

```
\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
```

두 번째로 실행하면 오류입니다. 그것을 누그러뜨릴 `IF EXISTS`가 없기 때문입니다:

```
DROP CONSTRAINT film_title_uniq ON film;
```

```
ERROR:  constraint "film_title_uniq" of relation "film" does not exist
```

```
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;
                         ^
```

이름을 붙이지 않은 제약의 이름 찾기

이름 없이 만든 제약에는 유도된 이름이 붙습니다. `pg_constraint`가 그 레이블 테이블의 모든 제약을 나열합니다:

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

`teen_age`가 이름을 붙인 검사 제약이고 `teen_properties_check`가 유도된 것입니다. 나머지 셋은 그 레이블 자신의 모양에 속한 것입니다.

검사 제약 지우기

```
DROP CONSTRAINT teen_properties_check ON teen;
DROP CONSTRAINT teen_age ON teen;
```

둘 다 사라졌고, `\dGv`가 보고할 것이 없다는 것으로 그것을 말해 줍니다:

```
\dGv rulebooks.teen
```

```
          List of labels
   Graph    | Name |  Type  | Owner
------------+------+--------+-------
 rulebooks | teen | vertex | agens
(1 row)

Vertex label "rulebooks.teen"
--
Inherits: rulebooks.ag_vertex
```

알아 둘 만한 거부 사례

속성 인덱스는 이름이 무엇이든 제약이 아닙니다:

```
CREATE PROPERTY INDEX film_year_idx ON film (year);
DROP CONSTRAINT film_year_idx ON film;
```

```
ERROR:  constraint "film_year_idx" of relation "film" does not exist
```

그리고 그 이름에 그래프를 붙일 수는 없습니다:

```
DROP CONSTRAINT rulebooks.film_unique_constraint ON film;
```

```
ERROR:  syntax error at or near "."
LINE 1: DROP CONSTRAINT rulebooks.film_unique_constraint ON film;
                                 ^
```

마지막으로 예제가 만든 그래프를 지웁니다:

```
DROP GRAPH rulebooks CASCADE;
```

참고

경고

`DROP CONSTRAINT`는 레이블 테이블의 이름 있는 모든 제약을 받으며, 레이블의 기본 키도 그중 하나입니다. `DROP CONSTRAINT film_pkey ON film`은 성공하고, 그 레이블은 `id`에 대한 기본 키가 없는 상태가 되며, 그것을 되돌려 놓는 문장은 없습니다. 먼저 `\dGv`나 `pg_constraint`로 이름을 확인하고, **`CREATE CONSTRAINT`**가 만든 제약만 지우십시오.

© 2026 SKAI Worldwide Co., Ltd.

제약을 바꾸려면 그것을 지우고 다시 만드십시오. `CREATE OR REPLACE CONSTRAINT`는 거부됩니다.

자식 레이블이 상속한 검사 제약은 그것을 선언한 레이블에서만 지울 수 있습니다:

```
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);
```

자식을 이름으로 부르면 거부됩니다:

```
DROP CONSTRAINT work_y ON feature;
```

```
ERROR:  cannot drop inherited constraint "work_y" of relation "feature"
```

부모에서 지우면 계열 전체에서 사라집니다:

```
DROP CONSTRAINT work_y ON work;
DROP GRAPH hierarchy CASCADE;
```

호환성

ISO/IEC 39075 (GQL)

GQL에는 제약 DDL이 없으므로 이 문장에 대응하는 것이 없습니다. 표준이 그래프를 제약하는 방식은 그래프 타입이고, 그것은 `<drop graph type statement>` (§12.7)로 통째로 지워집니다. 그래프 타입은 **GQL 적합성 표**에 계획으로 기록돼 있습니다.

PostgreSQL

`DROP CONSTRAINT ... ON <label>`은 `ALTER TABLE <table> DROP CONSTRAINT <name>`에 대응하며, 레이블 이름이 제약 이름 앞이 아니라 뒤에 옵니다. 제공하는 것은 더 적습니다. `IF EXISTS`도, `CASCADE` / `RESTRICT`도 없고, 한 문장에서 제약 여러 개를 지울 방법도 없습니다. 다만 닿는 범위는 정확히 같습니다. 레이블 테이블의 이름 있는 모든 제약이며, `ALTER TABLE`이 닿는 것과 같은 집합입니다.

관련 항목

- `CREATE CONSTRAINT`
- `DROP PROPERTY INDEX`
- `ALTER VLABEL` · `ALTER ELABEL`
- 그래프가 어떻게 저장되는가

함수

모든 함수

이 페이지는 이 레퍼런스에 문서화된 모든 함수, 모두 126개 항목을 알파벳순 표 하나에 나열합니다. 어느 범주에 속하는지 몰라도 이름으로 함수를 찾을 수 있습니다. **함수** 열은 고려할 점과 실제 예제가 실린 온전한 항목으로 이어집니다. **시그니처** 열은 카탈로그가 담고 있는 모든 오버로드를 보여 줍니다.

© 2026 SKAI Worldwide Co., Ltd.

이름이 같은 항목이 둘 있다면, 그것은 OID도 뜻도 다른 두 개의 함수입니다. 문자열에 대한 `length` 는 경로에 대한 `length` 가 아니며, 시그니처 열이 그 둘을 구별해 줍니다.

함수	시그니처	설명	도입 시점
abs	`abs ( x jsonb ) -> jsonb` `abs ( x numeric ) -> numeric`	숫자의 절댓값을 반환합니다.	2.17 이전부터
acos	`acos ( x jsonb ) -> jsonb` `acos ( x double precision ) -> double precision`	`x` 의 아크코사인을 라디안으로 반환합니다. `x` 는 −1과 1 사이여야 합니다.	2.17 이전부터
ag_get_graphconstraintdef	`ag_get_graphconstraintdef ( constraint oid ) -> text`	OID를 주면 그래프 제약에 대한 `CREATE CONSTRAINT` 명령을 재구성합니다. `pg_get_constraintdef` 의 그래프 쪽 대응 함수이고, `pg_dump` 가 쓰는 것입니다.	2.17 이전부터
ag_get_propindexdef	`ag_get_propindexdef ( index oid ) -> text`	OID를 주면 속성 인덱스에 대한 `CREATE PROPERTY INDEX` 명령을 재구성합니다.	2.17 이전부터
ag_property_text	`ag_property_text ( properties jsonb, key text, expected_kind “char” ) -> text`	속성의 값을 `text` 로 반환하지만, 저장된 값이 요구한 JSON 종류일 때만 그렇습니다.	2.18에 추가
agversion	`agversion ( ) -> text`	실행 중인 서버의 AgensGraph 버전을 반환하며, `version()` 이 함께 담는 PostgreSQL 부분은 빼고 반환합니다.	2.17 이전부터
all	`all ( variable IN list WHERE predicate ) -> boolean`	리스트의 모든 요소에 대해 조건이 성립하면 참을 반환합니다.	2.17 이전부터
allshortestpaths	`allshortestpaths ( pattern ) -> graphpath`	두 정점 사이의 모든 최단 경로를 찾습니다. 경로 변수에 이름을 붙이면 경로마다 한 행을 반환하고, 표현식으로 쓰면 경로의 배열을 반환합니다.	2.17 이전부터
any	`any ( variable IN list WHERE predicate ) -> boolean`	리스트의 요소 중 적어도 하나에 대해 조건이 성립하면 참을 반환합니다.	2.17 이전부터

array_size	`array_size ( list jsonb ) -> integer`
	`array_size ( list json ) -> integer`
	`array_size ( list anyarray ) -> integer`
		배열의 요소 개수를 반환합니다.	2.17 이전부터

asin	`asin ( x jsonb ) -> jsonb` `asin ( x double precision ) -> double precision`	`x` 의 아크사인을 라디안으로 반환합니다. `x` 는 −1과 1 사이여야 합니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
atan	`atan ( x jsonb ) -> jsonb` `atan ( x double precision ) -> double precision`	`x` 의 아크탄젠트를 라디안으로 반환합니다.	2.17 이전부터
atan2	`atan2 ( y jsonb, x jsonb ) -> jsonb` `atan2 ( y double precision, x double precision ) -> double precision`	`y / x` 의 아크탄젠트를 라디안으로 반환하며, 두 인자의 부호를 써서 결과를 올바른 사분면에 놓습니다.	2.17 이전부터
avg	`avg ( expression numeric ) -> numeric`	그룹에서 null이 아닌 값들의 산술 평균을 반환합니다.	2.17 이전부터
cardinality	`cardinality ( value jsonb ) -> jsonb` `cardinality ( p graphpath ) -> jsonb`	리스트의 요소 수, 맵의 항목 수, 또는 경로의 정점과 간선 수를 반환합니다.	2.18에 추가
ceil	`ceil ( x jsonb ) -> jsonb` `ceil ( x numeric ) -> numeric`	`x` 보다 작지 않은 가장 작은 정수를 반환합니다.	2.17 이전부터
coalesce	`coalesce ( value [, value ] … ) -> jsonb`	null이 아닌 첫 인자를 반환합니다.	2.17 이전부터

collect	`collect ( expression anyelement ) -> jsonb`
	`collect ( v vertex ) -> vertex[]`
	`collect ( e edge ) -> edge[]`
	`collect ( p graphpath ) -> graphpath[]`
		그룹의 값들을 그 그룹의 행 순서대로 리스트에 모읍니다.	2.17 이전부터

cos	`cos ( x jsonb ) -> jsonb` `cos ( x double precision ) -> double precision`	라디안으로 주어진 각의 코사인을 반환합니다.	2.17 이전부터
cosd	`cosd ( x double precision ) -> double precision`	도로 주어진 각의 코사인을 반환합니다.	2.17 이전부터
cot	`cot ( x jsonb ) -> jsonb` `cot ( x double precision ) -> double precision`	라디안으로 주어진 각의 코탄젠트, 즉 탄젠트의 역수를 반환합니다. 실제 예제는 **atan2** 를 참고하십시오.	2.17 이전부터
count	`count ( * ) -> bigint` `count ( expression ) -> bigint`	그룹의 행을 셉니다. `count(*)` 는 모든 행을 세고, `count(expression)` 은 그 표현식이 null이 아닌 행을 셉니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
cypher_to_jsonb	`cypher_to_jsonb ( value anyelement ) -> jsonb`	Cypher의 규칙으로 값을 `jsonb`로 감쌉니다. 그 규칙은 문자열 타입에 대해 `to_jsonb`의 규칙과 다릅니다.	2.17 이전부터

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`
		현재 트랜잭션의 타임스탬프, 지정한 시간대의 현재 시각, 또는 각 부분으로 만든 타임스탬프를 반환합니다.	2.17 이전부터

degrees	`degrees ( x jsonb ) -> jsonb` `degrees ( x double precision ) -> double precision`	라디안을 도로 변환합니다.	2.17 이전부터

dijkstra	`dijkstra ( pattern, weight ) -> graphpath`
	`dijkstra ( pattern, weight, condition ) -> graphpath`
	`dijkstra ( pattern, weight, LIMIT count ) -> graphpath`
	`dijkstra ( pattern, weight, condition, LIMIT count ) -> graphpath`
		두 정점 사이의 가장 값싼 경로를 찾습니다. 각 간선의 비용은 `weight`가 평가된 값입니다.	2.17 이전부터

e	`e ( ) -> numeric`	오일러 수를 `numeric`으로 반환합니다.	2.17 이전부터
edge_to_jsonb	`edge_to_jsonb ( e edge ) -> jsonb`	간선의 `properties`를 반환합니다. 형 변환 `e::jsonb`가 쓰는 함수가 이것이고, **`properties(e)`**와 같은 값을 반환합니다.	2.17 이전부터
edges	`edges ( p graphpath ) -> edge[]`	경로의 간선을 순서대로 반환합니다. **`relationships`**가 Cypher가 쓰는 이름으로 부른 같은 함수입니다.	2.17 이전부터
end_id	`end_id ( e edge ) -> graphid`	그 정점을 읽지 않고, 간선이 가리키는 정점의 `graphid`를 반환합니다.	2.17 이전부터
end_vertex	`end_vertex ( e edge ) -> vertex`	간선이 가리키는 정점을 반환합니다. `endNode`가 Cypher가 쓰는 이름으로 부른 같은 함수이며, 둘 다 스칼라 페이지에 나열돼 있습니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
endNode	`endNode ( e edge ) -> vertex`	간선이 가리키는 정점을 반환합니다. `end_vertex`가 다른 이름으로 부른 같은 함수입니다. 경로 페이지의 **end_vertex**를 참고하십시오.	2.17 이전부터
exists	`exists ( pattern ) -> boolean` `exists ( property_reference ) -> boolean`	패턴을 주면 그 패턴이 적어도 한 번 일치할 때 참을 반환합니다. 속성 참조를 주면 그 속성이 있고 null이 아닐 때 참을 반환합니다.	2.17 이전부터
exp	`exp ( x jsonb ) -> jsonb` `exp ( x numeric ) -> numeric`	e의 `x` 제곱을 반환합니다.	2.17 이전부터
floor	`floor ( x jsonb ) -> jsonb` `floor ( x numeric ) -> numeric`	`x`보다 크지 않은 가장 큰 정수를 반환합니다. 실제 예제는 **ceil**을 참고하십시오.	2.17 이전부터
get_last_graph_write_stats	`get_last_graph_write_stats ( ) -> record`	이 세션의 가장 최근 그래프 쓰기가 무엇을 바꿨는지 카운터 다섯 개로 된 한 행으로 보고합니다. `insertedvertices`, `insertededges`, `deletedvertices`, `deletededges`, `updatedproperties`입니다.	2.17 이전부터
get_time	`get_time ( ) -> time without time zone` `get_time ( timezone “any” ) -> time without time zone`	하루 중 현재 시각을 로컬 기준으로, 또는 지정한 시간대 기준으로 반환합니다.	2.17 이전부터
graph_labid	`graph_labid ( qualified_label cstring ) -> integer`	`graph.label` 형태로 지정한 레이블의 레이블 id를 반환합니다.	2.17 이전부터
graphid	`graphid ( labid integer, locid bigint ) -> graphid` `graphid ( id numeric ) -> graphid`	레이블 id와 로컬 id로, 또는 `graphid`가 출력되는 형태인 `labid.locid` 숫자로 `graphid`를 만듭니다.	2.17 이전부터
graphid_in_jsonb_array	`graphid_in_jsonb_array ( id graphid, elements jsonb ) -> boolean`	`graphid`가 `jsonb` 배열에 담긴 그래프 요소 중 하나이면 참을 반환합니다.	2.17 이전부터
graphid_labid	`graphid_labid ( id graphid ) -> integer`	`graphid`의 레이블 부분, 즉 점 앞의 숫자를 반환합니다.	2.17 이전부터
graphid_locid	`graphid_locid ( id graphid ) -> bigint`	`graphid`의 로컬 부분, 즉 점 뒤의 숫자를 반환하며, 그 값은 그 레이블의 시퀀스에서 나온 것입니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
head	`head ( list jsonb ) -> jsonb` `head ( list anyarray ) -> anyelement`	리스트의 첫 요소를 반환합니다.	2.17 이전부터
id	`id ( v vertex ) -> graphid` `id ( e edge ) -> graphid`	그래프 요소의 식별자를 반환합니다.	2.17 이전부터
isEmpty	`isEmpty ( value jsonb ) -> boolean`	리스트에 요소가 없거나 맵에 키가 없으면 참을 반환합니다.	2.17 이전부터
jsonb_add	`jsonb_add ( a jsonb, b jsonb ) -> jsonb`	백틱 `+` 연산자를 구현합니다. 덧셈이거나, 문자열 둘 또는 리스트 둘의 이어 붙이기입니다.	2.17 이전부터
jsonb_array_length	`jsonb_array_length ( list jsonb ) -> integer`	`jsonb` 배열의 요소 개수를 `integer` 로 반환합니다.	2.17 이전부터
jsonb_div	`jsonb_div ( a jsonb, b jsonb ) -> jsonb`	백틱 `/` 연산자를 구현합니다. 두 피연산자가 모두 정수면 정수 나눗셈입니다.	2.17 이전부터
jsonb_mod	`jsonb_mod ( a jsonb, b jsonb ) -> jsonb`	백틱 `%` 연산자를 구현합니다.	2.17 이전부터
jsonb_mul	`jsonb_mul ( a jsonb, b jsonb ) -> jsonb`	백틱 `*` 연산자를 구현합니다.	2.17 이전부터
jsonb_pow	`jsonb_pow ( a jsonb, b jsonb ) -> jsonb`	백틱 `^` 연산자를 구현합니다.	2.17 이전부터
jsonb_sub	`jsonb_sub ( a jsonb, b jsonb ) -> jsonb`	백틱 이항 `-` 연산자를 구현합니다.	2.17 이전부터
jsonb_uminus	`jsonb_uminus ( a jsonb ) -> jsonb`	백틱 접두 `-` 연산자를 구현합니다.	2.17 이전부터
jsonb_uplus	`jsonb_uplus ( a jsonb ) -> jsonb`	백틱 접두 `+` 연산자를 구현합니다.	2.17 이전부터
keys	`keys ( map jsonb ) -> jsonb`	맵의 키 이름 목록, 또는 그래프 요소의 속성 이름 목록을 반환합니다.	2.17 이전부터
label	`label ( v vertex ) -> jsonb` `label ( e edge ) -> jsonb`	그래프 요소가 만들어질 때의 레이블 이름을 `jsonb` 문자열로 반환합니다.	2.17 이전부터
labels	`labels ( v vertex ) -> jsonb`	정점 자신의 레이블과 그것이 상속하는 모든 레이블을 가까운 것부터 담은 리스트를 반환합니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
last	`last ( list jsonb ) -> jsonb` `last ( list anyarray ) -> anyelement`	리스트의 마지막 요소를 반환합니다. 빈 리스트는 null을 반환합니다. 실제 예제는 **head** 를 참고하십시오.	2.17 이전부터
left	`left ( string jsonb, n jsonb ) -> jsonb` `left ( string text, n integer ) -> text`	문자열의 앞 `n` 글자를 반환합니다.	2.17 이전부터

length	`length ( p graphpath ) -> jsonb`
	`length ( vs vertex[] ) -> jsonb`
	`length ( es edge[] ) -> jsonb`
	`length ( ps graphpath[] ) -> jsonb`
		경로에 대해서는 그 안의 간선 수를, 배열에 대해서는 요소 수를 반환합니다.	2.17 이전부터

length	`length ( value jsonb ) -> jsonb`	문자열의 글자 수, 리스트의 요소 수, 또는 맵의 키 수를 반환합니다.	2.17 이전부터
ln	`ln ( x double precision ) -> double precision` `ln ( x numeric ) -> numeric`	`x` 의 자연로그를 반환합니다. Cypher에서 **log** 가 해석되는 함수가 이것이고, 이 이름으로 직접 호출할 수도 있습니다.	2.17 이전부터
localdatetime	`localdatetime ( ) -> timestamp with time zone` `localdatetime ( timezone "any" ) -> timestamp without time zone`	현재 트랜잭션의 타임스탬프, 또는 지정한 시간대의 현재 벽시계 값을 반환합니다.	2.17 이전부터
log	`log ( x jsonb ) -> jsonb`	`x` 의 **자연**로그를 반환합니다.	2.17 이전부터
log10	`log10 ( x double precision ) -> double precision` `log10 ( x numeric ) -> numeric`	`x` 의 밑 10 로그를 반환합니다. 다시 쓰이지 않으므로 이 이름은 적힌 그대로의 뜻입니다.	2.17 이전부터
lower	`lower ( string text ) -> text`	문자열을 소문자로 반환합니다. PostgreSQL 자신의 함수이고, `jsonb` 문자열이 `text` 로 변환되기 때문에 Cypher에서 닿을 수 있습니다. **toLower** 가 Cypher 쪽 표기이고 `jsonb` 를 곧바로 받습니다.	2.17 이전부터
lTrim	`lTrim ( string jsonb ) -> jsonb`	문자열 앞쪽의 공백을 제거합니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
`max`	`max ( expression jsonb ) -> jsonb` `max ( expression numeric ) -> numeric`	그룹에서 가장 큰 값을 반환합니다.	2.17 이전부터
`min`	`min ( expression jsonb ) -> jsonb` `min ( expression numeric ) -> numeric`	그룹에서 가장 작은 값을 반환합니다. `max`의 고려할 점이 그대로 적용됩니다.	2.17 이전부터
`nodes`	`nodes ( p graphpath ) -> vertex[]`	경로의 정점을 순서대로 반환하며, 그 경로가 시작하는 정점부터 시작합니다.	2.17 이전부터
`none`	`none ( variable IN list WHERE predicate ) -> boolean`	리스트의 어떤 요소에 대해서도 조건이 성립하지 않으면 참을 반환합니다. `any`의 부정입니다.	2.17 이전부터
`percentileCont`	`percentileCont ( list jsonb, percentile numeric ) -> double precision`	숫자 리스트의 백분위수를 반환하며, 요구한 위치를 사이에 둔 두 값 사이를 보간합니다.	2.17 이전부터
`percentileDisc`	`percentileDisc ( list jsonb, percentile numeric ) -> double precision`	숫자 리스트의 백분위수를 반환하며, 실제로 리스트에 있는 요소로 내림합니다. 고려할 점은 `percentileCont`와 같습니다.	2.17 이전부터
`pi`	`pi ( ) -> double precision`	π를 반환합니다. 실제 예제는 `degrees`를 참고하십시오.	2.17 이전부터
`properties`	`properties ( v vertex ) -> jsonb` `properties ( e edge ) -> jsonb`	그래프 요소의 `properties` 전체를 맵으로 반환합니다.	2.17 이전부터
`radians`	`radians ( x jsonb ) -> jsonb` `radians ( x double precision ) -> double precision`	도를 라디안으로 변환합니다.	2.17 이전부터
`rand`	`rand ( ) -> jsonb`	0 이상 1 미만의 반열린 구간에 있는 수를 반환합니다.	2.17 이전부터
`range`	`range ( start integer, stop integer ) -> jsonb` `range ( start integer, stop integer, step integer ) -> jsonb`	`start`부터 `stop`까지의 정수 목록을 반환하며, 양 끝을 포함합니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
regather_graphmeta	`regather_graphmeta ( ) -> boolean`	`ag_graphmeta`를 비우고, 현재 그래프의 모든 간선 레이블을 스캔해 다시 만듭니다. `ag_graphmeta`는 어떤 (시작 정점 레이블, 간선 레이블, 끝 정점 레이블) 조합이 나타나는지와 각 조합의 간선이 몇 개인지를 기록하며, 플래너는 그것으로 패턴이 일치시킬 수 없는 레이블 테이블을 건너뜁니다.	2.17 이전부터
relationships	`relationships ( p graphpath ) -> edge[]`	경로의 간선을 순서대로 반환합니다. **edges**와 동일합니다.	2.17 이전부터
replace	`replace ( string jsonb, old jsonb, new jsonb ) -> jsonb`	어떤 부분 문자열이 나타나는 모든 곳을 다른 문자열로 바꿉니다.	2.17 이전부터

reverse	`reverse ( string text ) -> text`
	`reverse ( list jsonb ) -> jsonb`
	`reverse ( list anyarray ) -> anyarray`
		문자열이나 리스트를 뒤집습니다.	2.17 이전부터

right	`right ( string jsonb, n jsonb ) -> jsonb` `right ( string text, n integer ) -> text`	문자열의 뒤 `n`글자를 반환합니다. 속성에 정수 리터럴을 함께 줄 때의 모호함을 **left**와 공유하며, 그것을 푸는 두 방법도 같습니다.	2.17 이전부터
round	`round ( x jsonb ) -> jsonb` `round ( x numeric [, scale integer ] ) -> numeric`	`x`를 가장 가까운 정수로 반올림하며, 0.5는 0에서 먼 쪽으로 올립니다.	2.17 이전부터
rTrim	`rTrim ( string jsonb ) -> jsonb`	문자열 뒤쪽의 공백을 제거합니다. 실제 예제는 **lTrim**을 참고하십시오.	2.17 이전부터
shortestpath	`shortestpath ( pattern ) -> graphpath`	두 정점 사이의 최단 경로 하나를 간선 수 기준으로 찾습니다. 똑같이 짧은 것이 여럿이면 그중 하나가 반환됩니다.	2.17 이전부터
sign	`sign ( x jsonb ) -> jsonb` `sign ( x numeric ) -> numeric`	`x`의 부호에 따라 −1, 0, 1을 반환합니다. 실제 예제는 **ceil**을 참고하십시오.	2.17 이전부터
sin	`sin ( x double precision ) -> double precision`	라디안으로 주어진 각의 사인을 반환합니다.	2.17 이전부터
sind	`sind ( x double precision ) -> double precision`	도로 주어진 각의 사인을 반환합니다. 실제 예제는 **cosd**를 참고하십시오.	2.17 이전부터
single	`single ( variable IN list WHERE predicate ) -> boolean`	리스트의 요소 중 정확히 하나에 대해 조건이 성립하면 참을 반환합니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
size	`size ( value ) -> jsonb` `size ( pattern ) -> bigint`	값을 주면 그 길이를 반환합니다. 패턴을 주면 그 패턴이 일치하는 횟수를 반환합니다.	2.17 이전부터
split	`split ( string text, delimiter text ) -> anyarray`	구분자가 나타나는 모든 곳에서 문자열을 잘라 리스트로 만듭니다.	2.17 이전부터
sqrt	`sqrt ( x jsonb ) -> jsonb` `sqrt ( x numeric ) -> numeric`	`x` 의 제곱근을 반환합니다.	2.17 이전부터
start_id	`start_id ( e edge ) -> graphid`	그 정점을 읽지 않고, 간선이 떠나는 정점의 `graphid` 를 반환합니다. **`end_id`** 의 고려할 점이 그대로 적용됩니다.	2.17 이전부터
start_vertex	`start_vertex ( e edge ) -> vertex`	간선이 떠나는 정점을 반환합니다. `startNode` 가 Cypher가 쓰는 이름으로 부른 같은 함수입니다. **`startNode`** 를 참고하십시오.	2.17 이전부터
startNode	`startNode ( e edge ) -> vertex`	간선이 떠나는 정점을 반환합니다. `start_vertex` 가 다른 이름으로 부른 같은 함수입니다. 경로 페이지의 **`start_vertex`** 를 참고하십시오.	2.17 이전부터
stdev	`stdev ( expression numeric ) -> numeric`	그룹의 값들의 표본 표준편차를 반환합니다. n − 1로 나누는 추정값입니다.	2.17 이전부터
stdevp	`stdevp ( expression numeric ) -> numeric`	그룹의 값들의 모표준편차를 반환합니다. n으로 나누는 추정값입니다. 파서가 이 이름을 `stddev_pop` 으로 다시 쓰며, `stDevP` 도 같은 호출입니다.	2.17 이전부터
str_size	`str_size ( string text ) -> integer`	문자열의 글자 수를 `integer` 로 반환합니다.	2.17 이전부터
string_contains	`string_contains ( a jsonb, b jsonb ) -> boolean`	`CONTAINS` 를 구현합니다. 오른쪽 문자열이 왼쪽 문자열 어디에든 나타나면 참입니다.	2.17 이전부터
string_ends_with	`string_ends_with ( a jsonb, b jsonb ) -> boolean`	`ENDS WITH` 를 구현합니다. 왼쪽 문자열이 오른쪽 문자열로 끝나면 참입니다.	2.17 이전부터
string_regex	`string_regex ( a jsonb, b jsonb ) -> boolean`	`=~` 를 구현합니다. 왼쪽 문자열이 오른쪽을 정규 표현식으로 삼아 일치하면 참입니다.	2.17 이전부터
string_starts_with	`string_starts_with ( a jsonb, b jsonb ) -> boolean`	`STARTS WITH` 를 구현합니다. 왼쪽 문자열이 오른쪽 문자열로 시작하면 참입니다.	2.17 이전부터
substring	`substring ( string jsonb, start jsonb ) -> jsonb` `substring ( string jsonb, start jsonb, count jsonb ) -> jsonb`	문자열의 일부를 반환하며, `start` 에서 시작해 끝까지 또는 `count` 글자만큼 이어집니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
sum	`sum ( expression numeric ) -> numeric`	그룹에서 null이 아닌 값들의 합을 반환합니다.	2.17 이전부터
tail	`tail ( list jsonb ) -> jsonb` `tail ( list anyarray ) -> anyarray`	리스트의 첫 요소를 뺀 나머지 전부를 반환합니다.	2.17 이전부터
tan	`tan ( x jsonb ) -> jsonb` `tan ( x double precision ) -> double precision`	라디안으로 주어진 각의 탄젠트를 반환합니다. 실제 예제는 **sin** 을 참고하십시오.	2.17 이전부터
tand	`tand ( x double precision ) -> double precision`	도로 주어진 각의 탄젠트를 반환합니다. 실제 예제는 **cosd** 를 참고하십시오.	2.17 이전부터
to_jsonb	`to_jsonb ( value anyelement ) -> jsonb`	어떤 SQL 값이든 `jsonb` 로 변환하며, 복합 타입은 객체로, 배열은 배열로 대응시킵니다.	2.17 이전부터
toBoolean	`toBoolean ( value text ) -> boolean` `toBoolean ( value “any” ) -> boolean`	값이 나타내는 불리언을 반환하고, 아무것도 나타내지 않으면 null을 반환합니다.	2.17 이전부터
toBooleanOrNull	`toBooleanOrNull ( value text ) -> boolean` `toBooleanOrNull ( value “any” ) -> boolean`	값이 나타내는 불리언을 반환하거나 null을 반환합니다. **toBoolean** 이 아예 읽을 수 없는 타입에서 오류를 낼 수 있는 자리에서, 이 함수는 null을 반환합니다.	2.17 이전부터
toFloat	`toFloat ( value “any” ) -> double precision`	값이 나타내는 부동소수점 수를 반환하거나 null을 반환합니다.	2.18에 추가
toFloatList	`toFloatList ( list jsonb ) -> jsonb` `toFloatList ( list anyarray ) -> double precision[]`	리스트의 모든 요소를 부동소수점 수로 변환합니다.	2.17 이전부터
toInteger	`toInteger ( value “any” ) -> bigint`	값이 나타내는 정수를 반환하거나 null을 반환합니다.	2.18에 추가
toIntegerList	`toIntegerList ( list jsonb ) -> jsonb` `toIntegerList ( list anyarray ) -> bigint[]`	리스트의 모든 요소를 정수로 변환합니다. 어느 오버로드가 적용되는지가 반환 타입을 결정하며, **toFloatList** 와 정확히 같습니다. Cypher 리스트는 `jsonb` 를, PostgreSQL 배열은 `bigint[]` 를 줍니다.	2.17 이전부터
toLower	`toLower ( string jsonb ) -> jsonb`	문자열을 소문자로 반환합니다.	2.17 이전부터
toString	`toString ( value jsonb ) -> jsonb`	값의 문자열 형태를 반환합니다.	2.17 이전부터

© 2026 SKAI Worldwide Co., Ltd.

함수	시그니처	설명	도입 시점
toStringList	`toStringList ( list jsonb ) -> jsonb` `toStringList ( list anyarray ) -> text[]`	리스트의 모든 요소를 문자열로 변환합니다. 어느 오버로드가 적용되는지가 반환 타입을 결정하며, **toFloatList** 와 같습니다. Cypher 리스트는 `jsonb` 를, PostgreSQL 배열은 `text[]` 를 줍니다.	2.17 이전부터
toStringOrNull	`toStringOrNull ( value "any" ) -> text`	값의 문자열 형태를 반환하고, 그 값에 문자열 형태가 없으면 null을 반환합니다.	2.17 이전부터
toUpper	`toUpper ( string jsonb ) -> jsonb`	문자열을 대문자로 반환합니다. 실제 예제는 **toLower** 를 참고하십시오.	2.17 이전부터
trim	`trim ( string jsonb ) -> jsonb`	문자열 양쪽 끝의 공백을 제거합니다.	2.17 이전부터
type	`type ( e edge ) -> jsonb`	간선의 레이블 이름을 `jsonb` 문자열로 반환합니다. `label(e)` 와 같은 값이며, Cypher가 관계 타입에 쓰는 이름으로 부른 것입니다.	2.17 이전부터
unnest	`unnest ( from_json jsonb ) -> setof jsonb`	리스트를 요소마다 한 행으로 펼칩니다.	2.18에 추가
upper	`upper ( string text ) -> text`	문자열을 대문자로 반환합니다. PostgreSQL 자신의 함수입니다. **lower** 를 참고하십시오.	2.17 이전부터
vertex_to_jsonb	`vertex_to_jsonb ( v vertex ) -> jsonb`	정점의 `properties` 를 반환합니다. 형 변환 `v::jsonb` 가 쓰는 함수가 이것이고, **properties(v)** 와 같은 값을 반환합니다.	2.17 이전부터
vertices	`vertices ( p graphpath ) -> vertex[]`	경로의 정점을 순서대로 반환합니다. **nodes** 와 동일합니다.	2.17 이전부터

`collect` 는 2.18보다 오래됐지만, 네이티브 배열 결과는 이번 릴리스에서 왔습니다. `jsonb` 대신 `vertex[]` · `edge[]` · `graphpath[]` 입니다. `all` · `any` · `none` · `single` 뒤의 삼치 논리 구현도 그렇습니다. 변경 목록 전체는 2.18 릴리스 노트에 있습니다.

다른 곳에 문서화된 함수

`meta` 스키마는 SQL에서 그래프·레이블과 그 크기를 살펴보는 함수를 더해 줍니다. `meta.labels()` , `meta.count()` , `meta.label_size()` 등입니다. 핵심 카탈로그의 일부가 아니라 확장이므로, 설치한 데이터베이스에만 있습니다. 그 함수들은 자기 페이지인 그래프 메타데이터에 문서화돼 있고 여기서 반복하지 않습니다.

PostgreSQL 자신이 제공하는 모든 함수도 Cypher 쿼리에서 닿을 수 있으며, 여기 각 페이지에 설명된 타입 규칙을 따릅니다. 그것들에 대한 레퍼런스는 PostgreSQL 문서입니다.

© 2026 SKAI Worldwide Co., Ltd.

이 레퍼런스가 빼놓은 것

AgensGraph가 더한 함수들은 객체 식별자 7000번대부터 8100번대까지를 차지합니다. 그것들이 모두 호출하도록 만들어진 것은 아닙니다. 그 범위에는 타입 입력·출력 함수, B-tree와 GIN 연산자 클래스를 위한 비교·해시 지원 함수, 집계의 전이·최종 함수도 있습니다. 플래너 지원 함수 하나, `pg_upgrade` 도우미 둘, 그리고 경로 탐색 실행기가 해석하는 자리표 항목 넷도 있습니다. 이 레퍼런스는 쿼리가 쓸 수 있는 것을 문서화하고 그것들은 빼놓습니다.

자기 빌드가 무엇을 노출하는지 나열하기

어떤 서버가 무엇을 갖고 있는지에 대한 권위는 카탈로그입니다. 아래는 그 범위의 모든 AgensGraph 함수를 집계 함수까지 포함해 나열합니다:

```
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;
```

이 2.18 서버에서는 195행이 나옵니다. 위에 나열한 지원 항목들 때문에 이 레퍼런스가 문서화하는 것보다 많습니다. 세기만 하려면 이렇게 하십시오:

```
SELECT count(*) FROM pg_catalog.pg_proc WHERE oid BETWEEN 7000 AND 8100;
```

```
 count
-------
   195
```

AgensGraph가 더하지 않은 이름의 함수는 나타나지 않습니다. 그것은 PostgreSQL의 것이기 때문입니다. `avg`, `min`, `left`, `log10`이 그런 것들입니다. 그중 하나를 찾으려면 `WHERE` 절을 빼고 이름으로 걸러 내십시오.

관련 항목

- 집계 함수 — `avg`, `collect`, `count`, `min`, `max`, `sum`
- 조건 함수 — `all`, `any`, `none`, `single`, `exists`, `isEmpty`
- 스칼라 함수 — `id`, `label`, `properties`, `size`, `length`
- 리스트 함수 — `head`, `keys`, `range`, `tail`, `unnest`
- 문자열 함수 — `substring`, `split`, `toLower`, `trim`
- 수치 함수 — `abs`, `round`, `log`, 삼각 함수 계열
- 경로 함수 — `nodes`, `relationships`, `shortestpath`, `dijkstra`
- 타입 변환 함수 — `to…` 계열
- jsonb 연산자와 함수 — 모든 속성 아래에 있는 연산자들
- 그래프 메타데이터 — `meta.*` 스키마

집계 함수

행의 묶음을 하나의 값으로 줄이는 함수들입니다. Cypher는 SQL과 같은 방식으로 그룹을 만듭니다. `RETURN`이나 `WITH` 목록에서 집계되지 않은 모든 표현식이 그룹화 키입니다. 키가 아예 없으면 결과 전체가 하나의 그룹입니다.

예제는 예제 데이터셋에서 실행됩니다.

© 2026 SKAI Worldwide Co., Ltd.

avg

```
avg ( expression numeric ) -> numeric
```

그룹에서 null이 아닌 값들의 산술 평균을 반환합니다.

고려할 점

- 속성은 `jsonb` 로 들어와 `numeric` 으로 변환되므로, 결과는 `numeric` 이고 `double precision` 으로 둥글려지지 않고 정밀도를 온전히 유지합니다.
- 값이 null인 행은 건너뜁니다. null이 아닌 값이 하나도 없는 그룹은 null을 반환합니다.

```
MATCH (m:movie) RETURN avg(m.year) AS mean_year;
```

```
      mean_year
----------------------
 2002.0555555555555556
```

collect

```
collect ( expression anyelement ) -> jsonb
collect ( v vertex ) -> vertex[]
collect ( e edge ) -> edge[]
collect ( p graphpath ) -> graphpath[]
```

그룹의 값들을 그 그룹의 행 순서대로 리스트에 모읍니다.

고려할 점

- `vertex` · `edge` · `graphpath` 인자는 `jsonb` 배열이 아니라 그 타입 그대로의 배열을 만듭니다. 각각 `vertex[]` · `edge[]` · `graphpath[]` 입니다. 그 밖의 것은 `jsonb` 를 만듭니다. 2.18에 추가
- 그 타입 그대로의 배열도 `jsonb` 리스트처럼 Cypher의 첨자로 0부터 인덱싱합니다.
- null 값도 모읍니다. `collect(DISTINCT expr)` 은 중복을 없앱니다.
- `length()` 는 네 결과 타입 어느 것에 대해서도 요소 개수를 셉니다.

```
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"]
```

그래프 요소 자체를 모으면 그 요소들의 배열이 나옵니다:

```
MATCH (:movie {title:'The Matrix'})-[e:related_to]->()
WITH collect(e) AS edges
RETURN length(edges) AS n, pg_typeof(edges) AS type;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 n |   type
---+----------
 3 | "edge[]"
```

count

```
count ( * ) -> bigint
count ( expression ) -> bigint
```

그룹의 행을 셉니다. `count(*)`는 모든 행을 세고, `count(expression)`은 그 표현식이 null이 아닌 행을 셉니다.

고려할 점

- 정점이나 간선 전체에 대한 `count(v)`는 그 요소의 식별자를 셉니다. 파서가 그 인자를 그 `graphid`로 바꾸므로 `properties`는 전혀 읽지 않습니다.
- `count(DISTINCT expression)`은 null이 아닌 서로 다른 값을 셉니다.
- 행이 없는 그룹은 null이 아니라 0을 반환합니다.

```
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
```

그룹에서 가장 큰 값을 반환합니다.

고려할 점

- `jsonb` 인자는 `jsonb` 자신의 타입 순서로 정렬되며, 그 순서에서는 모든 숫자가 모든 문자열보다 아래에 옵니다. 타입 컬럼에서 읽은 수치·텍스트 속성은 그 컬럼의 타입으로 비교됩니다.
- null은 건너뜁니다. null이 아닌 값이 하나도 없는 그룹은 null을 반환합니다.

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

그룹에서 가장 작은 값을 반환합니다. `max`의 고려할 점이 그대로 적용됩니다.

percentileCont

```
percentileCont ( list jsonb, percentile numeric ) -> double precision
```

숫자 리스트의 백분위수를 반환하며, 요구한 위치를 사이에 둔 두 값 사이를 보간합니다.

고려할 점

- 이것은 집계 함수가 아니라 리스트에 대한 평범한 함수입니다. 그룹의 백분위수를 구하려면 먼저 리스트를 만드십시오. `percentileCont(collect(m.year), 0.5)`입니다.
- 인자는 리스트여야 합니다. 스칼라는 거부되고, 메시지는 엉뚱한 함수를 이름으로 부릅니다:

```
RETURN percentileCont(2010, 0.5);
```

```
ERROR:  reverse(): list is expected but 2010
```

- 리스트에 요소가 둘보다 적거나 백분위수가 0에서 1 범위를 벗어나면 null을 반환합니다.
- 숫자가 아닌 요소는 거부되는 것이 아니라 무시됩니다.

```
MATCH (m:movie) RETURN percentileCont(collect(m.year), 0.5) AS median;
```

```
 median
--------
 2002.5
```

percentileDisc

```
percentileDisc ( list jsonb, percentile numeric ) -> double precision
```

숫자 리스트의 백분위수를 반환하며, 실제로 리스트에 있는 요소로 내림합니다. 고려할 점은 **`percentileCont`**와 같습니다.

```
MATCH (m:movie) RETURN percentileDisc(collect(m.year), 0.5) AS median;
```

```
 median
--------
 2002
```

stdev

```
stdev ( expression numeric ) -> numeric
```

그룹의 값들의 표본 표준편차를 반환합니다. n − 1로 나누는 추정값입니다.

고려할 점

- 파서가 이 이름을 `stddev_samp`로 다시 쓰므로, 결과 컬럼이 지니는 이름과 오류 메시지가 쓰는 이름이 그것입니다.

© 2026 SKAI Worldwide Co., Ltd.

- 따옴표 없는 식별자는 소문자로 바뀌므로 `stDev` 와 `STDEV` 는 같은 호출입니다.
- `stdDev` 는 다른 함수입니다. 그것은 `stddev` 로 바뀌고, 그것은 `stddev_samp` 에 대한 PostgreSQL 자신의 별칭입니다. 계산되는 값은 같습니다.
- 행이 하나인 그룹은 null을 반환합니다.

```
MATCH (m:movie) RETURN stdev(m.year) AS sample_sd, stdevp(m.year) AS population_sd;
```

```
      sample_sd      |    population_sd
---------------------+---------------------
 11.3628508344062724 | 11.0427061006209596
```

stdevp

```
stdevp ( expression numeric ) -> numeric
```

그룹의 값들의 모표준편차를 반환합니다. n으로 나누는 추정값입니다. 파서가 이 이름을 `stddev_pop` 으로 다시 쓰며, `stDevP` 도 같은 호출입니다.

sum

```
sum ( expression numeric ) -> numeric
```

그룹에서 null이 아닌 값들의 합을 반환합니다.

고려할 점

- null은 건너뜁니다. null이 아닌 값이 하나도 없는 그룹은 0이 아니라 null을 반환합니다.
- 입력의 중복을 없애는 것은 없습니다. 패턴이 두 번 일치시킨 값은 두 번 더해집니다. 그것을 원하지 않으면 `sum(DISTINCT expr)` 를 쓰십시오.

```
MATCH (p:person)-[:acted_in]->(:movie) RETURN sum(p.born) AS born_total;
```

```
 born_total
------------
 49017
```

호환성

openCypher

`stDev` 와 `stDevP` 는 표본 표준편차와 모표준편차에 대한 openCypher의 이름이며, 두 표기 모두 여기서 동작합니다. openCypher의 `percentileCont` 와 `percentileDisc` 는 행마다 값을 받는 집계 함수입니다. 여기의 것들은 완성된 리스트를 받으므로, openCypher용으로 쓴 쿼리는 첫 인자를 `collect()` 로 감싸야 합니다.

ISO/IEC 39075

`avg` , `count` , `max` , `min` , `stddev_pop` , `stddev_samp` , `sum` 이 GQL의 집계 함수이고 표준이 표기하는 대로 표기됩니다. `collect` 는 GQL의 `COLLECT` 값 표현식에 대응하며, 그것은 **서브쿼리 표현식**에 문서화돼 있습니다.

관련 항목

- 모든 함수 — 모든 함수를 한 표에

© 2026 SKAI Worldwide Co., Ltd.

- 조건 함수 — `all`, `any`, `none`, `single`
- 리스트 함수 — `collect`가 내놓는 리스트 다루기

조건 함수

예·아니요로 답하는 함수들입니다. 리스트 수량자 네 개, `all` · `any` · `none` · `single`은 평범한 호출이 아니라 문법 형태입니다. 각각 변수, 리스트, `WHERE` 절을 받습니다. 각각 요소별 결과를 삼치 논리로 줄이므로, null을 담은 리스트는 답을 거짓이 아니라 null로 만들 수 있습니다.

예제는 **예제 데이터셋**에서 실행됩니다.

all

```
all ( variable IN list WHERE predicate ) -> boolean
```

리스트의 모든 요소에 대해 조건이 성립하면 참을 반환합니다.

고려할 점

- 빈 리스트는 참을 반환합니다.
- 어떤 요소에서 조건이 null이고 나머지 전부에서 참이면 결과는 null입니다. 거짓인 요소가 하나 있으면 다른 요소가 null이어도 결과는 거짓입니다.
- 조건은 모든 요소에 대해 평가되며, 조기에 끊는 것이 없습니다.
- 이 호출은 `cypher_all()`로 컴파일됩니다. 조건 안에서 나온 오류 메시지에는 그 이름이 실립니다.

```
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
```

리스트의 요소 중 적어도 하나에 대해 조건이 성립하면 참을 반환합니다.

고려할 점

- 빈 리스트는 거짓을 반환합니다.
- 참인 요소가 하나 있으면 다른 요소가 null이어도 결과는 참입니다.
- 참인 요소가 없고 null인 것이 하나라도 있으면 결과는 null입니다.

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

```
    actor
-------------
 "Sam Neill"
```

exists

```
exists ( pattern ) -> boolean
exists ( property_reference ) -> boolean
```

패턴을 주면 그 패턴이 적어도 한 번 일치할 때 참을 반환합니다. 속성 참조를 주면 그 속성이 있고 null이 아닐 때 참을 반환합니다.

고려할 점

- 두 형태는 하나의 함수의 오버로드가 아니라 서로 다른 구문입니다. 패턴 형태는 `EXISTS` 서브쿼리가 되고, 속성 형태는 `IS NOT NULL` 검사가 됩니다.
- 속성 형태의 인자는 `m.year` 같은 속성 참조여야 합니다. 임의의 표현식은 패턴으로 읽혀 파싱에 실패합니다.
- `exists()` 안의 패턴은 아래의 `(p)` 처럼 감싸는 쿼리의 변수를 바인딩할 수 있지만, 그것이 도입하는 변수는 밖에서 보이지 않습니다.

```
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
```

리스트에 요소가 없거나 맵에 키가 없으면 참을 반환합니다.

고려할 점

- 인자는 `jsonb` 로 해석돼야 합니다. 문자열 리터럴만 쓰면 그것은 `unknown` 이고 range·multirange의 `isempty` 에도 일치합니다:

```
RETURN isEmpty('');
```


© 2026 SKAI Worldwide Co., Ltd.

```
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.
```

형 변환을 하십시오: `isEmpty(''::jsonb)`.

- 속성 참조는 이미 `jsonb` 이므로 `isEmpty(n.tags)` 에는 형 변환이 필요하지 않습니다.
- 빈 문자열에 대해 참을 반환합니다. 그것이 빈 `jsonb` 스칼라입니다.

```
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
```

리스트의 어떤 요소에 대해서도 조건이 성립하지 않으면 참을 반환합니다. **`any`** 의 부정입니다.

고려할 점

- 빈 리스트는 참을 반환합니다.
- 참인 요소가 하나 있으면 결과는 거짓입니다. 참인 요소가 없고 null인 것이 하나라도 있으면 결과는 null입니다.

```
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
```

리스트의 요소 중 정확히 하나에 대해 조건이 성립하면 참을 반환합니다.

고려할 점

- 빈 리스트는 거짓을 반환합니다.
- 어떤 요소에서든 결과가 null이면 답 전체가 null입니다. 그 null이 두 번째 일치였을 수도 있기 때문입니다.

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

```
      director       |          years
---------------------+--------------------------
 "Christopher Nolan" | [2010, 2014]
 "Denis Villeneuve"  | [2017]
 "Joss Whedon"       | [2012]
 "Lana Wachowski"    | [2003, 1999, 2021, 2003]
 "Neill Blomkamp"    | [2013]
```

삼치 결과 나란히 보기

수량자들의 차이는 리스트가 null을 담고 있을 때와 리스트가 비어 있을 때 가장 크게 드러납니다. 아래 두 행은 실제 출력입니다:

```
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
```

`RETURN` 은 불리언 컬럼을 `jsonb` 로 감싸지 않고 `boolean` 으로 둡니다. 그래서 이것들은 `true` · `false` · `null` 이 아니라 `t` · `f` 와 null에 해당하는 빈 칸으로 출력됩니다.

호환성

openCypher

속성에 대한 `exists()` 는 `IS NOT NULL` 의 openCypher 옛 표기입니다. 패턴 컴프리헨션과 `exists { ... }` 블록 형태는 지원되지 않습니다. 위의 패턴 형태나 **서브쿼리 표현식**을 쓰십시오.

관련 항목

- **모든 함수** — 모든 함수를 한 표에
- **서브쿼리 표현식** — `EXISTS` , `COUNT` , `COLLECT`
- **리스트 함수** — 이 수량자들이 대상으로 삼는 리스트 만들기

스칼라 함수

값을 받아 하나의 값을 반환하는 함수들입니다. 그래프 요소의 각 부분, `graphid` 의 조각들, 그리고 문자열이나 리스트의 길이를 다룹니다. 여기 있는 서버 정보 함수들은 이 빌드가 무엇인지, 마지막 쓰기가 무엇을 했는지 보고합니다.

예제는 **예제 데이터셋**에서 실행됩니다.

© 2026 SKAI Worldwide Co., Ltd.

agversion

```
agversion ( ) -> text
```

실행 중인 서버의 AgensGraph 버전을 반환하며, `version()` 이 함께 담는 PostgreSQL 부분은 빼고 반환합니다.

고려할 점

- `agversion` 은 읽기 전용 설정이기도 하므로 `SHOW agversion` 이 같은 문자열을 반환합니다. 함수 쪽은 표현식 안에서 쓸 수 있는 형태입니다.
- 그 값은 PostgreSQL의 버전이 아니라 이 포크 자신의 버전입니다. 둘 다 보려면 `version()` 을 읽으십시오.

```
SELECT agversion();
```

```
 agversion
------------
 2.18.6.1
```

ag_get_graphconstraintdef

```
ag_get_graphconstraintdef ( constraint oid ) -> text
```

OID를 주면 그래프 제약에 대한 `CREATE CONSTRAINT` 명령을 재구성합니다. `pg_get_constraintdef` 의 그래프 쪽 대응 함수이고, `pg_dump` 가 쓰는 것입니다.

ag_get_propindexdef

```
ag_get_propindexdef ( index oid ) -> text
```

OID를 주면 속성 인덱스에 대한 `CREATE PROPERTY INDEX` 명령을 재구성합니다.

고려할 점

- 속성 인덱스는 `properties` 컬럼에 대한 표현식 인덱스이므로 `pg_get_indexdef` 는 그것을 SQL로 출력합니다. 이 함수는 그 대신 Cypher 형태로 출력합니다.

coalesce

```
coalesce ( value [, value ] ... ) -> jsonb
```

null이 아닌 첫 인자를 반환합니다.

고려할 점

- 이것은 함수가 아니라 표현식 형태이므로, 카탈로그의 함수 목록에는 나타나지 않습니다.
- 속성 참조는 언제나 `jsonb` 입니다. 인자 중 하나라도 `jsonb` 이면 모든 인자가 `jsonb` 로 변환되고 결과도 `jsonb` 입니다. 그렇지 않으면 인자들이 공통 타입으로 정해져 그것을 유지하므로, 정점 둘에 대한 `coalesce(v1, v2)` 는 `vertex` 를 반환합니다.
- 그래서 그래프 요소를 다른 것과 섞으면 실패합니다. 그 요소를 감쌀 수 없기 때문입니다. 대신 `m::jsonb` 를 넘기십시오.
- null이 아닌 첫 인자 뒤의 인자들은 평가되지 않습니다.

© 2026 SKAI Worldwide Co., Ltd.

```
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;
```

```
 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
```

현재 트랜잭션의 타임스탬프, 지정한 시간대의 현재 시각, 또는 각 부분으로 만든 타임스탬프를 반환합니다.

이름	타입	설명
`year` , `month` , `mday`	`integer`	달력상의 날짜입니다.
`hour` , `min` , `sec`	`integer`	하루 중 시각입니다.
`millisec` , `nanosec`	`integer`	1초 미만 부분입니다. 마이크로초 해상도를 넘는 나노초는 버려집니다.
`timezone`	`"any"`	시간대 이름이나 약어입니다. 예를 들어 `'Asia/Seoul'` 입니다.

고려할 점

- 인자 없는 형태는 안정 함수이므로, 한 쿼리의 모든 행이 같은 시점을 보고 트랜잭션 사이에서는 값이 바뀝니다.
- 시간대 형태는 `timestamp without time zone` 을 반환합니다. 그 시간대의 벽시계 값이고 시간대 자체는 버려집니다.
- `RETURN` 은 결과를 ISO 8601 형태의 `jsonb` 문자열로 감쌉니다.

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

endNode

```
endNode ( e edge ) -> vertex
```

간선이 가리키는 정점을 반환합니다. `end_vertex`가 다른 이름으로 부른 같은 함수입니다. 경로 페이지의 **`end_vertex`**를 참고하십시오.

end_id

```
end_id ( e edge ) -> graphid
```

그 정점을 읽지 않고, 간선이 가리키는 정점의 `graphid`를 반환합니다.

고려할 점

- 간선 테이블은 `start`와 `end`를 컬럼으로 저장하므로, 이 함수는 간선 행 자체를 읽는 것 말고는 비용이 없습니다. `endNode()`는 그 정점을 가져와야 합니다.
- 식별자를 비교하거나 그룹화하기만 하면 될 때는 언제든 이것을 쓰십시오.

```
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
```

이 세션의 가장 최근 그래프 쓰기가 무엇을 바꿨는지 카운터 다섯 개로 된 한 행으로 보고합니다. `insertedvertices`, `insertededges`, `deletedvertices`, `deletededges`, `updatedproperties`입니다.

고려할 점

- 이 카운터는 세션 전체가 아니라 마지막 그래프 쓰기 문장만 설명합니다.
- 아무것도 쓰지 않은 문장은 앞 문장의 카운터를 0으로 만들지 않고 그대로 남기므로, 관심 있는 쓰기 바로 뒤에 그 행을 읽으십시오.
- SQL에서 호출하십시오. 이 함수는 레코드를 반환하고, Cypher에는 레코드를 펼칠 방법이 없습니다.

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

하루 중 현재 시각을 로컬 기준으로, 또는 지정한 시간대 기준으로 반환합니다.

고려할 점

- 두 형태 모두 불변으로 표시돼 있어서 플래너가 그 호출을 상수로 접습니다. 쿼리의 모든 행이 같은 시각을 보고, 준비된 문장의 모든 실행도 그렇습니다. 시각이 흘러가야 한다면 안정 함수인 `localdatetime()`이나 SQL의 `localtime`에서 읽으십시오.

```
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
```

레이블 id와 로컬 id로, 또는 `graphid`가 출력되는 형태인 `labid.locid` 숫자로 `graphid`를 만듭니다.

고려할 점

- 레이블 테이블의 `id` 컬럼 기본값이 이 함수로 정의돼 있습니다. **그래프가 어떻게 저장되는가**를 참고하십시오.
- 숫자 형태는 손으로 쓰는 SQL을 위한 편의입니다. 그 레이블이 존재하는지 확인하지 않습니다.

```
SELECT graphid(4, 1), graphid(4.1::numeric);
```

```
 graphid | graphid
---------+---------
 4.1     | 4.1
```

graph_labid

```
graph_labid ( qualified_label cstring ) -> integer
```

`graph.label` 형태로 지정한 레이블의 레이블 id를 반환합니다.

```
SELECT graph_labid('moviekb.person');
```

```
 graph_labid
-------------
           4
```

graphid_labid

```
graphid_labid ( id graphid ) -> integer
```

`graphid`의 레이블 부분, 즉 점 앞의 숫자를 반환합니다.

© 2026 SKAI Worldwide Co., Ltd.

고려할 점

- `graphid`는 상위 16비트에 레이블 id를, 나머지에 로컬 id를 담습니다. 그래서 하나의 레이블 전체가 `graphid` 값의 연속된 한 구간을 차지하고, 레이블 검사를 카탈로그를 건드리지 않고 인덱스 범위 스캔으로 답할 수 있습니다.

graphid_locid

```
graphid_locid ( id graphid ) -> bigint
```

`graphid`의 로컬 부분, 즉 점 뒤의 숫자를 반환하며, 그 값은 그 레이블의 시퀀스에서 나온 것입니다.

```
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
```

그래프 요소의 식별자를 반환합니다.

고려할 점

- 이것은 카탈로그 함수가 아닙니다. `vertex`는 복합 타입 `(id, properties)`이고 `edge`는 `(id, start, end, properties)` 입니다. 그래서 `id(v)`는 필드 선택을 PostgreSQL이 함수 형태로 표기한 것입니다(`parse_func.c:1961`). `properties(v)`도 같은 장치로 얻습니다.
- `n.id`는 같은 것이 **아닙니다.** 그것은 `id`라는 이름의 속성을 읽는데, 예제 데이터셋에는 그런 속성이 없으므로 null을 반환합니다. `id(n)`으로 쓰십시오.
- `RETURN`은 결과를 `jsonb`로 감싸지 않고 `graphid`로 둡니다. 집계하면 감싸집니다. `min(id(n))`은 `jsonb` 문자열 `"4.1"`로 돌아옵니다.

```
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
```

그래프 요소가 만들어질 때의 레이블 이름을 `jsonb` 문자열로 반환합니다.

© 2026 SKAI Worldwide Co., Ltd.

고려할 점

- `label` 과 **`labels`** 는 하나를 다르게 표기한 것이 아니라 서로 다른 함수입니다. `label` 은 문자열 하나, 그 요소 자신의 레이블을 반환합니다. `labels` 는 그 레이블 뒤에 그것이 상속하는 레이블들을 붙인 리스트를 반환합니다.
- `label` 은 정점뿐 아니라 간선에서도 동작하고, `labels` 는 그렇지 않습니다. 간선에 대해서는 `label(e)` 와 **`type(e)`** 가 같은 호출입니다.

```
MATCH (n:person {name:'Keanu Reeves'}) RETURN label(n) AS own, labels(n) AS own_and_inherited;
```

```
   own    | own_and_inherited
----------+-------------------
 "person" | ["person"]
```

두 답이 갈라지는 것은 레이블이 상속할 때뿐입니다. 부모가 있는 레이블을 하나 추가합니다:

```
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"]
```

그다음 그것을 다시 없앱니다:

```
MATCH (d:dragon) DETACH DELETE d;
DROP VLABEL dragon;
DROP VLABEL creature;
```

간선에는 `labels()` 가 없습니다:

```
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
```

정점 자신의 레이블과 그것이 상속하는 모든 레이블을 가까운 것부터 담은 리스트를 반환합니다.

© 2026 SKAI Worldwide Co., Ltd.

고려할 점

- 정점만 받습니다. `labels(edge)`는 없습니다.
- 부모가 없는 레이블에서는 리스트에 요소가 하나이고, 그것이 흔한 경우입니다.

length

```
length ( value jsonb ) -> jsonb
```

문자열의 글자 수, 리스트의 요소 수, 또는 맵의 키 수를 반환합니다.

고려할 점

- **`size`**가 이 함수의 Cypher 쪽 표기입니다. 파서가 그것을 다시 쓰므로 결과 컬럼의 이름이 `length`가 됩니다.
- 경로나 그래프 요소의 배열에 대한 `length`는 뜻이 다른 별개의 함수입니다. 그 경로의 간선 개수입니다. 경로 페이지의 **`length`**를 참고하십시오.
- 숫자를 주면 오류를 내지 않고 그 텍스트 형태의 길이를 반환합니다.

```
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
```

현재 트랜잭션의 타임스탬프, 또는 지정한 시간대의 현재 벽시계 값을 반환합니다.

고려할 점

- 인자 없는 형태는 타입까지 포함해 `datetime()`과 같은 값을 반환합니다. 두 이름이 있는 것은 GQL 호환을 위한 것이며, 어느 쪽도 시간대를 버리지 않습니다.
- 시간대 형태는 `datetime(zone)`과 마찬가지로 `timestamp without time zone`을 반환합니다.
- 두 형태 모두 안정 함수이므로, **`get_time`**과 달리 여기서 얻은 값은 트랜잭션 사이에서 흘러갑니다.

properties

```
properties ( v vertex ) -> jsonb
properties ( e edge ) -> jsonb
```

그래프 요소의 `properties` 전체를 맵으로 반환합니다.

고려할 점

- **`id`**와 마찬가지로 이것은 카탈로그 함수가 아니라 복합 타입에 대한 필드 선택이므로, 그 컬럼을 읽는 것 이상의 비용이 들지 않습니다.
- `vertex_to_jsonb`와 `edge_to_jsonb`가 같은 맵을 반환하며, 형 변환 `v::jsonb`가 쓰는 함수입니다.

© 2026 SKAI Worldwide Co., Ltd.

- `properties`를 읽으면 그것을 전부 detoast 합니다. 키 하나만 원한다면 그 키를 읽으십시오.

```
MATCH (n:person {name:'Keanu Reeves'}) RETURN properties(n) AS bag;
```

```
                  bag
----------------------------------------
 {"born": 1964, "name": "Keanu Reeves"}
```

regather_graphmeta

```
regather_graphmeta ( ) -> boolean
```

`ag_graphmeta`를 비우고, 현재 그래프의 모든 간선 레이블을 스캔해 다시 만듭니다. `ag_graphmeta`는 어떤 (시작 정점 레이블, 간선 레이블, 끝 정점 레이블) 조합이 나타나는지와 각 조합의 간선이 몇 개인지를 기록하며, 플래너는 그것으로 패턴이 일치시킬 수 없는 레이블 테이블을 건너뜁니다.

고려할 점

- 이 스캔은 그래프의 모든 간선을 읽으므로 비용이 그래프 크기와 함께 커집니다.
- `auto_gather_graphmeta`는 쓰기가 일어날 때마다 그 테이블을 최신으로 유지합니다. 그 설정을 끈 채로 대량 적재를 했다면 손으로 다시 만드십시오.
- `NOTICE: regathering graphmeta...`를 내고 참을 반환합니다.

```
SELECT regather_graphmeta();
```

```
NOTICE:  regathering graphmeta...
 regather_graphmeta
--------------------
 t
```

size

```
size ( value ) -> jsonb
size ( pattern ) -> bigint
```

값을 주면 그 길이를 반환합니다. 패턴을 주면 그 패턴이 일치하는 횟수를 반환합니다.

고려할 점

- 두 형태는 첫 글자로 구별됩니다. 패턴은 언제나 `(`로 시작하므로, 그럴 수 없는 것은 값으로 읽힙니다. 값 형태는 **`length`**가 되므로 결과 컬럼의 이름이 `length`입니다. 패턴 형태는 개수를 세는 서브쿼리가 되므로 이름이 `count`입니다.
- `shortestpath`나 `allshortestpaths`를 인자로 주면 패턴으로 읽히지 않고, 찾아낸 경로의 개수라는 기존 뜻을 유지합니다.

```
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
```

간선이 떠나는 정점을 반환합니다. `start_vertex`가 다른 이름으로 부른 같은 함수입니다. 경로 페이지의 **`start_vertex`**를 참고하십시오.

start_id

```
start_id ( e edge ) -> graphid
```

그 정점을 읽지 않고, 간선이 떠나는 정점의 `graphid`를 반환합니다. **`end_id`**의 고려할 점이 그대로 적용됩니다.

type

```
type ( e edge ) -> jsonb
```

간선의 레이블 이름을 `jsonb` 문자열로 반환합니다. `label(e)`와 같은 값이며, Cypher가 관계 타입에 쓰는 이름으로 부른 것입니다.

고려할 점

- 간선만 받습니다. 정점에 쓰면 거부됩니다. **`label`**을 쓰십시오.

```
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 type casts.
```

```
MATCH ()-[e:acted_in]->() WITH e LIMIT 1 RETURN type(e) AS relationship_type;
```

```
 relationship_type
-------------------
 "acted_in"
```

호환성

openCypher

`id()`는 openCypher가 반환하는 정수가 아니라 `graphid`, 즉 레이블 id와 시퀀스 번호를 반환합니다. 그 요소가 살아 있는 동안 변하지 않고 그래프 전체에서 고유합니다. 그러나 작고 촘촘한 정수가 아니며, 그것에 대한 산술은 뜻이 없습니다.

openCypher의 `labels()`는 정점이 지닌 모든 레이블을 반환합니다. 거기서는 정점이 여러 개를 지닐 수 있기 때문입니다. 여기서 정점은 정확히 하나의 레이블을 가지며, `labels()`는 그 레이블에 그것이 상속하는 레이블들을 더해 반환합니다.

© 2026 SKAI Worldwide Co., Ltd.

관련 항목

- 모든 함수 — 모든 함수를 한 표에
- 경로 함수 — 경로에 대한 `nodes`, `relationships`, `length`
- jsonb 연산자와 함수 — `properties()`가 반환하는 맵 읽기
- 그래프 메타데이터 함수 — `meta.*` 스키마

리스트 함수

리스트와 맵에 대한 함수들입니다. Cypher의 리스트는 `jsonb` 배열이므로, 이 함수들은 `jsonb`를 받고 `jsonb`를 반환합니다. 그래프 요소에 대한 `collect()`는 그 대신 네이티브 배열을 만들고, 배열 오버로드가 있는 함수는 둘 다에서 동작합니다.

Cypher는 리스트의 첨자를 0부터 세고, 슬라이스 `list[a..b]`는 상한을 포함하지 않습니다.

예제는 **예제 데이터셋**에서 실행됩니다.

array_size

```
array_size ( list jsonb ) -> integer
array_size ( list json ) -> integer
array_size ( list anyarray ) -> integer
```

배열의 요소 개수를 반환합니다.

고려할 점

- 배열만 받습니다. `cardinality`와 달리 맵의 키는 세지 않습니다.
- 오버로드 세 개가 있어서 같은 이름이 Cypher 리스트, `json` 값, PostgreSQL 배열에서 모두 동작합니다.

```
RETURN array_size([1,2,3]) AS n;
```

```
 n
---
 3
```

cardinality

2.18에 추가

```
cardinality ( value jsonb ) -> jsonb
cardinality ( p graphpath ) -> jsonb
```

리스트의 요소 수, 맵의 항목 수, 또는 경로의 정점과 간선 수를 반환합니다.

고려할 점

- 문자열은 거부됩니다. 문자열에는 `length`를 쓰십시오.

```
RETURN cardinality('abc');
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  cardinality(): list or map is expected but "abc"
```

- 경로에 대해서는 정점과 간선을 함께 세므로, 간선 하나인 경로는 3을 주고 같은 경로에 대한 `length`는 1을 줍니다.
- PostgreSQL 자신의 `cardinality(anyarray)`도 네이티브 배열에 그대로 적용되며, `jsonb`가 아니라 `integer`를 반환합니다.

```
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
```

리스트의 첫 요소를 반환합니다.

고려할 점

- 빈 리스트는 오류를 내지 않고 null을 반환합니다.
- 배열 오버로드는 요소 자신의 타입을 반환하므로, 정점에 대한 `head(collect(n))`은 `vertex`를 반환합니다.

```
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
```

맵의 키 이름 목록, 또는 그래프 요소의 속성 이름 목록을 반환합니다.

고려할 점

- 순서는 `jsonb` 자신의 키 순서입니다. 키 길이순, 그다음 바이트순입니다. 삽입 순서도, 알파벳 순서도 아닙니다. 고정된 순서가 필요하면 결과를 정렬하십시오.

© 2026 SKAI Worldwide Co., Ltd.

- 정점이나 간선을 인자로 줄 수 있습니다. 그것이 자기 속성 맵으로 변환되기 때문입니다.

```
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
```

리스트의 마지막 요소를 반환합니다. 빈 리스트는 null을 반환합니다. 실제 예제는 **head** 를 참고하십시오.

range

```
range ( start integer, stop integer ) -> jsonb
range ( start integer, stop integer, step integer ) -> jsonb
```

`start` 부터 `stop` 까지의 정수 목록을 반환하며, 양 끝을 포함합니다.

이름	타입	설명
`start`	`integer`	목록의 첫 값입니다.
`stop`	`integer`	목록이 도달할 수 있는 마지막 값입니다. `start` 에서 `step` 씩 더해 정확히 그 값에 닿을 때만 포함됩니다.
`step`	`integer`	증가량입니다. 생략하면 올라갈 때는 1, 내려갈 때는 -1입니다.

고려할 점

- 결과는 PostgreSQL 배열이 아니라 `jsonb` 이므로, 다른 Cypher 리스트와 마찬가지로 첨자를 붙이고 훑을 수 있습니다.
- 양 끝이 포함됩니다. `stop` 은 `start` 에서 `step` 씩 더해 정확히 그 값에 닿을 때만 목록에 나타납니다. `range(1,10,3)` 은 10에서 끝나지만 `range(1,9,3)` 은 7에서 끝납니다.
- `stop` 이 `start` 보다 작으면 음수 걸음 없이도 내려가며 셉니다.

© 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
```

리스트의 첫 요소를 뺀 나머지 전부를 반환합니다.

고려할 점

- `head`와 `last`와 달리, 빈 리스트는 null이 아니라 빈 리스트를 반환합니다.

```
RETURN head([]) AS first, tail([]) AS rest;
```

```
 first | rest
-------+------
       | []
```

unnest

2.18에 추가

```
unnest ( from_json jsonb ) -> setof jsonb
```

리스트를 요소마다 한 행으로 펼칩니다.

고려할 점

- 집합 반환 함수입니다. Cypher에서는 자기가 놓인 절의 행을 곱합니다. `FOR x IN list`가 같은 생각의 절 형태이고, 요소를 이름에 바인딩해야 할 때는 그쪽이 읽기 쉽습니다.
- 출력 컬럼의 이름은 `value`이므로, SQL에서는 `SELECT value FROM unnest(...)`로 씁니다.
- PostgreSQL의 `unnest(anyarray)`도 네이티브 배열에 그대로 적용됩니다.

```
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.

뒤집기와 쪼개기

리스트를 만드는 함수 두 개는 문자열 함수 페이지에 문서화돼 있습니다. 문자열뿐 아니라 리스트도 뒤집는 **`reverse`**, 그리고 문자열을 리스트로 자르는 **`split`** 입니다. 리스트 타입의 변환 세 가지는 타입 변환 페이지에 있습니다. **`toStringList`**, **`toIntegerList`**, **`toFloatList`** 입니다.

호환성

openCypher

여기서 `range()` 는 openCypher에서와 마찬가지로 `stop` 을 포함합니다. openCypher의 `reduce()` 는 지원되지 않습니다. 리스트 컴프리헨션이나 `FOR` 를 쓰십시오. 패턴 컴프리헨션은 지원되지 않습니다.

관련 항목

- 모든 함수 — 모든 함수를 한 표에
- 집계 함수 — 리스트를 만드는 `collect`
- 조건 함수 — `all`, `any`, `none`, `single`, `isEmpty`
- jsonb 연산자와 함수 — 리스트 아래에 있는 연산자들

문자열 함수

텍스트에 대한 함수들입니다. 속성은 `jsonb` 로 들어오고, 이 함수 중 몇 개는 `text` 오버로드와 `jsonb` 오버로드를 모두 갖고 있습니다. 파서가 어느 것을 고르는지는 무엇을 넘기는지에 달려 있으므로, 리터럴과 속성이 항상 같게 동작하지는 않습니다. 그것이 문제가 되는 항목마다 그 사실을 적어 두었습니다.

예제는 **예제 데이터셋**에서 실행됩니다.

left

```
left ( string jsonb, n jsonb ) -> jsonb
left ( string text, n integer ) -> text
```

문자열의 앞 `n` 글자를 반환합니다.

고려할 점

- 속성에 정수 리터럴을 함께 주면 **모호합니다.** `left(jsonb, jsonb)` 와 `left(text, integer)` 가 똑같이 잘 맞기 때문입니다:

```
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.
```

`left(m.title::text, 12)` 처럼 문자열을 형 변환하거나, 그런 충돌이 없는 **`substring`** 을 쓰십시오.

- `n` 이 문자열보다 크면 문자열 전체를 반환합니다.

```
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
```

문자열을 소문자로 반환합니다. PostgreSQL 자신의 함수이고, `jsonb` 문자열이 `text`로 변환되기 때문에 Cypher에서 닿을 수 있습니다. **`toLower`**가 Cypher 쪽 표기이고 `jsonb`를 곧바로 받습니다.

```
RETURN lower('AbCdeFG') AS lower, upper('AbCdeFG') AS upper;
```

```
   lower    |   upper
------------+-----------
 "abcdefg" | "ABCDEFG"
```

lTrim

```
lTrim ( string jsonb ) -> jsonb
```

문자열 앞쪽의 공백을 제거합니다.

고려할 점

- 공백만 제거합니다. 제거할 문자 집합을 받는 형태는 없습니다. 그것이 필요하면 PostgreSQL의 `ltrim(text, text)`을 쓰십시오.

```
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
```

어떤 부분 문자열이 나타나는 모든 곳을 다른 문자열로 바꿉니다.

이름	타입	설명
`string`	`jsonb`	검색할 문자열입니다.
`old`	`jsonb`	찾을 부분 문자열입니다. 첫 번째만이 아니라 나타나는 모든 곳이 바뀝니다.

© 2026 SKAI Worldwide Co., Ltd.

이름	타입	설명
`new`	`jsonb`	바꿔 넣을 문자열입니다. `old` 보다 길거나 짧아도 되고, 빈 문자열도 됩니다.

고려할 점

- `old` 와 `new` 의 길이가 같아야 하는 것은 아닙니다.
- 나타나지 않는 `old` 를 주면 문자열이 그대로 반환됩니다.

```
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
```

문자열이나 리스트를 뒤집습니다.

고려할 점

- 어느 오버로드를 받는지는 인자에 달려 있고, 그 둘은 문자열에 대해 서로 다르게 동작합니다. 문자열 **리터럴**은 `reverse(text)` 로 해석되어 뒤집힙니다. 문자열 **속성**은 이미 `jsonb` 이므로 `reverse(jsonb)` 로 해석되는데, 그것은 배열만 뒤집습니다:

```
RETURN reverse('Blade Runner 2049'::jsonb);
```

```
ERROR:  reverse(): list is expected but "Blade Runner 2049"
```

속성 참조는 이미 `jsonb` 이므로 그 형 변환과 같게 동작합니다. 글자를 뒤집으려면 반대로 형 변환을 하십시오. `reverse(m.title::text)` 입니다.

- 리스트 속성에는 형 변환이 필요하지 않습니다. `jsonb` 오버로드가 리스트를 뒤집습니다.

```
RETURN reverse('ABCDEFG') AS text, reverse([1,2,3]) AS list;
```

```
   text    |   list
-----------+-----------
 "GFEDCBA" | [3, 2, 1]
```

```
MATCH (m:movie {title:'Blade Runner 2049'}) RETURN reverse(m.title::text) AS backwards;
```


© 2026 SKAI Worldwide Co., Ltd.

```
     backwards
---------------------
 "9402 rennuR edalB"
```

right

```
right ( string jsonb, n jsonb ) -> jsonb
right ( string text, n integer ) -> text
```

문자열의 뒤 `n` 글자를 반환합니다. 속성에 정수 리터럴을 함께 줄 때의 모호함을 **`left`** 와 공유하며, 그것을 푸는 두 방법도 같습니다.

rTrim

```
rTrim ( string jsonb ) -> jsonb
```

문자열 뒤쪽의 공백을 제거합니다. 실제 예제는 **`lTrim`** 을 참고하십시오.

split

```
split ( string text, delimiter text ) -> anyarray
```

구분자가 나타나는 모든 곳에서 문자열을 잘라 리스트로 만듭니다.

고려할 점

- 그 값은 문자열의 배열입니다. 카탈로그가 반환 타입을 `anyarray` 로 선언하므로 SQL에서 `pg_typeof` 가 `anyarray` 로 보고합니다. Cypher의 `RETURN` 은 그것을 `jsonb` 리스트로 감쌉니다.
- 나타나지 않는 구분자를 주면 문자열 전체를 담은 요소 하나짜리 리스트가 반환됩니다.
- 구분자는 패턴이 아니라 리터럴 문자열입니다. 정규 표현식이 필요하면 PostgreSQL의 `regexp_split_to_array` 를 쓰십시오.

```
RETURN split('Action/Sci-Fi/Drama', '/') AS genres;
```

```
           genres
-------------------------------
 ["Action", "Sci-Fi", "Drama"]
```

str_size

```
str_size ( string text ) -> integer
```

문자열의 글자 수를 `integer` 로 반환합니다.

고려할 점

- **`size`** 와 **`length`** 가 Cypher 쪽 표기이고 `jsonb` 를 반환합니다. 이 함수는 네이티브 `integer` 가 필요한 호출자가 `jsonb` 를 벗겨 내지 않아도 되도록 제공됩니다. 드라이버, 또는 Cypher 쿼리를 감싸는 SQL이 그런 호출자입니다.

```
RETURN str_size('hello') AS n;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 n
---
 5
```

substring

```
substring ( string jsonb, start jsonb ) -> jsonb
substring ( string jsonb, start jsonb, count jsonb ) -> jsonb
```

문자열의 일부를 반환하며, `start`에서 시작해 끝까지 또는 `count` 글자만큼 이어집니다.

이름	타입	설명
`string`	`jsonb`	자를 문자열입니다.
`start`	`jsonb`	첫 글자의 위치이며 0부터 셉니다.
`count`	`jsonb`	취할 글자 수입니다. 생략하면 문자열의 나머지 전부입니다. 끝을 넘어가는 `count`는 그냥 끝에서 멈춥니다.

고려할 점

- `start`는 0부터 세며, 1부터 세는 SQL의 `substring`과 다릅니다. 파서가 1을 대신 더해 주므로 뜻하는 위치를 그대로 쓰십시오.
- **`left`**나 **`right`**와 달리 이 함수는 속성에 대해 모호하지 않으므로, 속성의 앞부분을 취하는 더 안전한 방법입니다.

```
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
```

문자열을 소문자로 반환합니다.

고려할 점

- `jsonb`를 받으므로 **`reverse`**와 달리 속성에 형 변환 없이 동작합니다.
- **`lower`**가 PostgreSQL 쪽 대응 함수이고 그것도 동작합니다. 둘은 받고 반환하는 타입만 다릅니다.

```
MATCH (m:movie {title:'The Fifth Element'}) RETURN toLower(m.title) AS lower,
toUpper(m.title) AS upper;
```


© 2026 SKAI Worldwide Co., Ltd.

```
        lower         |         upper
----------------------+----------------------
 "the fifth element" | "THE FIFTH ELEMENT"
```

toUpper

```
toUpper ( string jsonb ) -> jsonb
```

문자열을 대문자로 반환합니다. 실제 예제는 `toLower`를 참고하십시오.

trim

```
trim ( string jsonb ) -> jsonb
```

문자열 양쪽 끝의 공백을 제거합니다.

고려할 점

- `trim`은 함수 이름이면서 문법 형태이기도 하므로, 여기서는 인자를 정확히 하나 받습니다. SQL의 `trim(BOTH 'x' FROM str)` 표기는 Cypher에서 쓸 수 없습니다:

```
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
```

문자열을 대문자로 반환합니다. PostgreSQL 자신의 함수입니다. `lower`를 참고하십시오.

함수 없이 일치시키기

문자열 검사 네 개는 함수가 아니라 연산자이며, 각각 직접 입력하지 않는 이름으로 카탈로그에 항목이 있습니다:

연산자	구현 함수	참이 되는 조건
`STARTS WITH`	`string_starts_with`	왼쪽 문자열이 오른쪽 문자열로 시작할 때
`ENDS WITH`	`string_ends_with`	왼쪽 문자열이 오른쪽 문자열로 끝날 때
`CONTAINS`	`string_contains`	오른쪽 문자열이 왼쪽 문자열 어디에든 나타날 때
`=~`	`string_regex`	왼쪽 문자열이 오른쪽을 정규 표현식으로 삼아 일치할 때

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (m:movie) WHERE m.title STARTS WITH 'The Matrix' RETURN count(*) AS n;
```

```
 n
---
 4
```

넷 중 어느 것도 속성 인덱스를 쓸 수 없습니다. 각각 속성과의 비교가 아니라 속성에 대한 함수 호출이므로, 그중 하나로 걸러 내는 쿼리는 그 레이블을 스캔합니다.

호환성

openCypher

openCypher는 대소문자 함수를 `toLower` · `toUpper` 로, 공백 제거 함수를 `lTrim` · `rTrim` · `trim` 으로 표기하며, 다섯 개 모두 동작합니다. openCypher의 `substring` 도 여기서와 마찬가지로 0부터 셉니다. `left` 와 `right` 는 openCypher도 정의하는 PostgreSQL 함수입니다. 속성에 정수 리터럴을 함께 줄 때의 모호함은 표준에서 벗어난 점이고, 이식 가능한 형태는 `substring` 입니다.

관련 항목

- 모든 함수 — 모든 함수를 한 표에
- 타입 변환 함수 — `toString` , `toStringList`
- 리스트 함수 — `split` 이 반환하는 것

수치 함수

산술·로그·삼각 함수들입니다. 대부분 속성을 곧바로 받는 `jsonb` 오버로드와 PostgreSQL에서 온 `double precision` 오버로드를 갖고 있습니다. 어느 쪽이든 Cypher의 `RETURN` 은 결과를 `jsonb` 숫자로 출력합니다.

이름이 `d` 로 끝나지 않으면 각도는 라디안입니다.

예제는 예제 데이터셋에서 실행됩니다.

abs

```
abs ( x jsonb ) -> jsonb
abs ( x numeric ) -> numeric
```

숫자의 절댓값을 반환합니다.

```
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
```

`x` 의 아크코사인을 라디안으로 반환합니다. `x` 는 −1과 1 사이여야 합니다.

© 2026 SKAI Worldwide Co., Ltd.

asin

```
asin ( x jsonb ) -> jsonb
asin ( x double precision ) -> double precision
```

`x`의 아크사인을 라디안으로 반환합니다. `x`는 -1과 1 사이여야 합니다.

atan

```
atan ( x jsonb ) -> jsonb
atan ( x double precision ) -> double precision
```

`x`의 아크탄젠트를 라디안으로 반환합니다.

atan2

```
atan2 ( y jsonb, x jsonb ) -> jsonb
atan2 ( y double precision, x double precision ) -> double precision
```

`y / x`의 아크탄젠트를 라디안으로 반환하며, 두 인자의 부호를 써서 결과를 올바른 사분면에 놓습니다.

고려할 점

- `atan(y / x)`보다 이것을 쓰십시오. 그것은 -15/3과 15/-3을 구별할 수 없어 사분면을 잃습니다.

```
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
```

ceil

```
ceil ( x jsonb ) -> jsonb
ceil ( x numeric ) -> numeric
```

`x`보다 작지 않은 가장 작은 정수를 반환합니다.

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

cos

```
cos ( x jsonb ) -> jsonb
cos ( x double precision ) -> double precision
```

라디안으로 주어진 각의 코사인을 반환합니다.

cosd

```
cosd ( x double precision ) -> double precision
```

도로 주어진 각의 코사인을 반환합니다.

고려할 점

- 4분의 1 회전 지점에서 정확합니다. 각을 먼저 변환하는 쪽은 그렇지 않습니다. `cosd(90)`은 0이지만 `cos(radians(90))`은 0.00000000000000006123233995736766입니다.
- 도 단위 형태에는 `jsonb` 오버로드가 없으므로 속성이 `double precision`으로 변환됩니다.

```
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
```

라디안으로 주어진 각의 코탄젠트, 즉 탄젠트의 역수를 반환합니다. 실제 예제는 **`atan2`**를 참고하십시오.

degrees

```
degrees ( x jsonb ) -> jsonb
degrees ( x double precision ) -> double precision
```

라디안을 도로 변환합니다.

```
RETURN degrees(pi()) AS half_turn, radians(180) AS in_radians;
```

```
 half_turn |    in_radians
-----------+------------------
 180       | 3.141592653589793
```

e

```
e ( ) -> numeric
```

오일러 수를 `numeric`으로 반환합니다.

© 2026 SKAI Worldwide Co., Ltd.

고려할 점

- 결과가 `numeric` 이므로, `double precision` 인 `exp(1)` 보다 자릿수가 많습니다. `2.7182818284590452` 대 `2.718281828459045` 입니다.

```
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
```

e의 `x` 제곱을 반환합니다.

floor

```
floor ( x jsonb ) -> jsonb
floor ( x numeric ) -> numeric
```

`x` 보다 크지 않은 가장 큰 정수를 반환합니다. 실제 예제는 **`ceil`** 을 참고하십시오.

ln

```
ln ( x double precision ) -> double precision
ln ( x numeric ) -> numeric
```

`x` 의 자연로그를 반환합니다. Cypher에서 **`log`** 가 해석되는 함수가 이것이고, 이 이름으로 직접 호출할 수도 있습니다.

log

```
log ( x jsonb ) -> jsonb
```

`x` 의 **자연**로그를 반환합니다.

고려할 점

- Cypher 쿼리에서는 파서가 `log` 를 **`ln`** 으로 다시 쓰므로, 결과 컬럼의 이름이 `ln` 이 되고 **인자 두 개 형태는 없습니다**:

```
RETURN log(2, 8);
```

```
ERROR:  function ln(integer, integer) does not exist
LINE 1: RETURN log(2, 8);
               ^
```


© 2026 SKAI Worldwide Co., Ltd.

```
HINT:  No function matches the given name and argument types. You might need to add explicit t
ype casts.
```

- 밑이 10인 PostgreSQL의 `log` 와는 반대입니다. Cypher에서 밑 10을 쓰려면 **`log10`** 을 쓰십시오.

```
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
```

`x` 의 밑 10 로그를 반환합니다. 다시 쓰이지 않으므로 이 이름은 적힌 그대로의 뜻입니다.

pi

```
pi ( ) -> double precision
```

π를 반환합니다. 실제 예제는 **`degrees`** 를 참고하십시오.

radians

```
radians ( x jsonb ) -> jsonb
radians ( x double precision ) -> double precision
```

도를 라디안으로 변환합니다.

rand

```
rand ( ) -> jsonb
```

0 이상 1 미만의 반열린 구간에 있는 수를 반환합니다.

고려할 점

- `rand()` 는 불변으로 표시돼 있어서 플래너가 그것을 하나의 상수로 접습니다. 쿼리의 모든 행과 준비된 문장의 모든 실행이 **같은** 수를 봅니다. 대신 PostgreSQL의 `random()` 을 호출하십시오. 그것은 변동 함수이고 Cypher에서도 동작합니다.

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

round

```
round ( x jsonb ) -> jsonb
round ( x numeric [, scale integer ] ) -> numeric
```

`x`를 가장 가까운 정수로 반올림하며, 0.5는 0에서 먼 쪽으로 올립니다.

고려할 점

- `jsonb` 형태는 정수로만 반올림합니다. 소수 자리를 남기려면 형 변환을 해서 `numeric` 형태를 쓰십시오. `round(x::numeric, 2)` 입니다.

sign

```
sign ( x jsonb ) -> jsonb
sign ( x numeric ) -> numeric
```

`x`의 부호에 따라 −1, 0, 1을 반환합니다. 실제 예제는 **`ceil`**을 참고하십시오.

sin

```
sin ( x double precision ) -> double precision
```

라디안으로 주어진 각의 사인을 반환합니다.

고려할 점

- 삼각 함수 중 사인만 자기 이름으로 된 `jsonb` 오버로드가 없습니다. `jsonb` 사인은 `jsonb_sin`으로 카탈로그에 등록돼 있습니다. 잃는 것은 없습니다. 속성이 `double precision`으로 변환되고 `sin()`이 PostgreSQL 함수로 해석됩니다. 다만 `jsonb` 인자에 대한 결과 타입이 이웃 함수들과 다릅니다.

```
RETURN sin(0.5) AS sin, cos(0) AS cos, tan(0) AS tan;
```

```
        sin         | cos | tan
--------------------+-----+-----
 0.479425538604203 | 1   | 0
```

```
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
```

도로 주어진 각의 사인을 반환합니다. 실제 예제는 **`cosd`**를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

sqrt

```
sqrt ( x jsonb ) -> jsonb
sqrt ( x numeric ) -> numeric
```

`x`의 제곱근을 반환합니다.

고려할 점

- 음수 인자는 null이 아니라 오류입니다:

```
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
```

라디안으로 주어진 각의 탄젠트를 반환합니다. 실제 예제는 **`sin`**을 참고하십시오.

tand

```
tand ( x double precision ) -> double precision
```

도로 주어진 각의 탄젠트를 반환합니다. 실제 예제는 **`cosd`**를 참고하십시오.

속성에 대한 산술

`+`, `-`, `*`, `/`, `%`, `^` 연산자는 속성에 대해 동작하지만, `jsonb` 피연산자에서는 SQL의 규칙이 아니라 `jsonb` 자신의 규칙을 따릅니다. 정수끼리 나누면 정수 나눗셈이 됩니다.

```
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
```

각각 `jsonb`에 대한, 백틱으로 이름 붙은 연산자로 구현돼 있습니다. **jsonb에 대한 산술 연산자**를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

호환성

openCypher

여기서 `log()` 는 밑이 e이고 PostgreSQL에서는 밑이 10입니다. openCypher도 자연로그를 `log` 로 표기하므로, Cypher 쿼리는 이식 가능하고 SQL 쿼리는 그렇지 않습니다. openCypher의 `e()` 와 `rand()` 도 있습니다. `rand()` 는 상수로 접힌다는 점에서 벗어나 있습니다. `haversin()` 은 지원되지 않습니다.

관련 항목

- 모든 함수 — 모든 함수를 한 표에
- 집계 함수 — `avg` , `sum` , `stdev`
- 타입 변환 함수 — `toInteger` , `toFloat`

경로 함수

`graphpath` 에 대한 함수들과, 그것을 만들어 내는 세 가지 경로 탐색 형태입니다. `graphpath` 는 `MATCH p = (...)` 가 바인딩하는, 정점과 간선이 번갈아 오는 나열입니다.

경로는 이름을 붙여 바인딩합니다. `MATCH p = (a)-[r]->(b)` 처럼입니다. 이 함수들은 패턴이 아니라 그 **변수**를 받습니다.

예제는 **예제 데이터셋**에서 실행됩니다.

allshortestpaths

```
allshortestpaths ( pattern ) -> graphpath
```

두 정점 사이의 모든 최단 경로를 찾습니다. 경로 변수에 이름을 붙이면 경로마다 한 행을 반환하고, 표현식으로 쓰면 경로의 배열을 반환합니다.

고려할 점

- 패턴에는 관계가 정확히 하나 있어야 하며, 그렇지 않으면 파서가 `only one relationship is allowed` 를 냅니다. 홉을 더 적는 대신 그 관계에 `*..4` 같은 길이 범위를 주십시오.
- 여러 타입을 쓴 관계 패턴은 어디서도 지원되지 않으며 `allshortestpaths` 도 그렇습니다. 레이블을 지정하지 않은 간선을 일치시키고 `WHERE label(r) IN [...]` 으로 걸러 내십시오.
- 표현식으로 쓰면 `graphpath[]` 를 내놓으므로, 그것에 대한 `length()` 는 한 경로의 간선 수가 아니라 찾아낸 경로의 개수를 셉니다.

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

두 정점 사이의 가장 값싼 경로를 찾습니다. 각 간선의 비용은 `weight`가 평가된 값입니다.

이름	타입	설명
`pattern`	경로 사슬	두 정점 사이의 관계 정확히 하나입니다. 예를 들어 `(a)-[e:related_to]->(b)` 입니다. `weight`가 그 관계의 속성을 읽는다면 관계에 변수를 주십시오.
`weight`	수치 표현식	간선 하나를 지나는 비용입니다. 고려되는 모든 간선에서 0보다 커야 합니다.
`condition`	불리언 표현식	간선에 대한 선택적 필터입니다. 이것을 만족하는 간선만 지나갑니다.
`count`	정수	선택 사항입니다. 값싼 것부터 반환할 대안 경로의 개수입니다. 0보다 커야 합니다.

고려할 점

- 패턴의 일부로 적으면 경로를 바인딩합니다. `p = dijkstra(...)` 입니다. `(p, w) = dijkstra(...)`로 적으면 경로와 그 총 가중치를 바인딩하며, 그 값은 그 밖의 방법으로는 얻을 수 없습니다.
- `LIMIT`이 없으면 개수의 기본값이 1이므로 경로 하나가 돌아옵니다.
- 패턴은 속성 제약도, 가변 길이 관계도 받지 않습니다. 대신 `WHERE`에서 걸러 내십시오.

```
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...
                           ^
```

```
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
```

- 세 번째 위치 인자는 조건이므로, 그 자리에 숫자를 두면 `cannot cast 1 (jsonb type numeric) to boolean`이 나옵니다.
- 거부가 두 개 더 있는데 간선에 도달한 뒤 실행 시점에 나오므로, 쿼리만이 아니라 데이터에 달려 있습니다. 0 이하로 평가되는 가중치는 `WEIGHT must be larger than 0`을, `LIMIT 0`은 `LIMIT must be larger than 0`을 보고합니다.
- 할당하지 않고 표현식으로만 쓰면 `graphpath[]`를 내놓습니다.

가장 값싼 경로가 가장 짧은 경로는 아닙니다. Total Recall을 지나는 두 홉의 비용은 4 + 2이고, 이 그래프에서는 그것이 한 홉 대안들을 이깁니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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
```

먼 쪽 끝점을 제약하지 않으면 도달할 수 있는 모든 영화까지의 가장 값싼 경로가 나옵니다:

```
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"]
```

`ORDER BY`가 없으면 행은 특정한 순서 없이 돌아옵니다. 위의 정렬은 지면을 재현 가능하게 만들기 위한 것뿐입니다.

참고

`dijkstra`가 찾은 경로도 다른 것과 같은 `graphpath`이므로 `nodes()`·`relationships()`·`length()`가 모두 그것을 읽습니다. 다른 경로가 지니지 않는 것은 총 가중치이고, 그것을 드러내는 것은 `(p, w) =` 형태뿐입니다.

edges

```
edges ( p graphpath ) -> edge[]
```

경로의 간선을 순서대로 반환합니다. **`relationships`**가 Cypher가 쓰는 이름으로 부른 같은 함수입니다.

end_vertex

```
end_vertex ( e edge ) -> vertex
```

간선이 가리키는 정점을 반환합니다. `endNode`가 Cypher가 쓰는 이름으로 부른 같은 함수이며, 둘 다 **스칼라 페이지**에 나열돼 있습니다.

고려할 점

- 정점을 읽는다는 것은 그 행을 가져온다는 뜻입니다. 식별자만으로 충분하다면 간선 자신의 컬럼을 읽는 **`end_id`**를 쓰십시오.

length

```
length ( p graphpath ) -> jsonb
length ( vs vertex[] ) -> jsonb
```


© 2026 SKAI Worldwide Co., Ltd.

```
length ( es edge[] ) -> jsonb
length ( ps graphpath[] ) -> jsonb
```

경로에 대해서는 그 안의 간선 수를, 배열에 대해서는 요소 수를 반환합니다.

고려할 점

- 요소 수가 아니라 간선 수입니다. 간선 하나인 경로의 길이는 1이지만 같은 경로의 `cardinality`는 3입니다.
- `allshortestpaths()`와 할당하지 않은 `dijkstra()`가 `graphpath[]`를 내놓으므로, 그런 것에 대한 `length()`는 경로의 개수를 셉니다.
- 문자열·리스트·맵에 대한 `length`는 다른 함수입니다. 스칼라 페이지의 **`length`**를 참고하십시오.

```
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[]
```

경로의 정점을 순서대로 반환하며, 그 경로가 시작하는 정점부터 시작합니다.

고려할 점

- 결과는 네이티브 배열인 `vertex[]`이므로 Cypher의 `RETURN`이 그것을 `jsonb`로 감싸지 않습니다. 첨자는 여전히 0부터입니다.
- `vertices`가 다른 이름으로 부른 같은 함수입니다.
- 경로 **변수**를 넘기십시오. 그 자리에 적은 패턴은 표현식이 아니어서 파싱되지 않습니다.
- 정점 전체를 출력하는 것이 원하는 것인 경우는 드뭅니다. 하나하나가 자기 `properties`를 온전히 지니고 있기 때문입니다. 아래처럼 리스트 컴프리헨션으로 필요한 항목만 투영하십시오.

```
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"]
```

경로의 모든 정점이 투영되므로, 어떤 정점이 지니고 있지 않은 키는 null로 돌아옵니다.

relationships

```
relationships ( p graphpath ) -> edge[]
```

경로의 간선을 순서대로 반환합니다. **`edges`**와 동일합니다.

© 2026 SKAI Worldwide Co., Ltd.

```
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
```

두 정점 사이의 최단 경로 하나를 간선 수 기준으로 찾습니다. 똑같이 짧은 것이 여럿이면 그중 하나가 반환됩니다.

고려할 점

- 패턴에는 관계가 정확히 하나 있어야 하고, 길이 범위는 그 관계가 지닙니다. 홉을 늘여 쓰는 것은 거부됩니다:

```
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)...
                            ^
```

- 똑같이 짧은 경로가 여럿일 때 어느 것을 받는지는 정해져 있지 않습니다. 전부 필요하면 `allshortestpaths`를 쓰십시오.
- 경로가 없으면 null인 경로가 아니라 아무 행도 반환하지 않습니다.
- 여러 타입을 쓴 관계 패턴은 여기서도 지원되지 않습니다.

```
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
```

간선이 떠나는 정점을 반환합니다. `startNode`가 Cypher가 쓰는 이름으로 부른 같은 함수입니다. `startNode`를 참고하십시오.

정점은 복합 타입이므로, 그 결과에서 속성을 곧바로 읽을 수 있습니다:

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

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

vertices

```
vertices ( p graphpath ) -> vertex[]
```

경로의 정점을 순서대로 반환합니다. `nodes`와 동일합니다.

호환성

openCypher

`shortestPath`와 `allShortestPaths`는 이름의 대소문자를 구분하지 않으므로 openCypher의 대소문자 표기도 동작합니다. openCypher에는 `dijkstra`가 없습니다. 거기서 가중 최단 경로는 프로시저 라이브러리에서 옵니다.

openCypher는 `-[r:acted_in|directed]->`처럼 관계 패턴이 여러 타입을 이름으로 부르는 것을 허용합니다. 여기서 그것은 모든 길이에서, 모든 경로 탐색 형태 안에서 `multiple types for relationship not supported`를 냅니다. 대신 레이블을 지정하지 않은 관계를 일치시키고 레이블을 검사하십시오:

```
MATCH (p:person)-[r]->(m:movie) WHERE label(r) IN ['acted_in','directed'] RETURN count(r) AS
n;
```

```
 n
----
 46
```

관련 항목

- 모든 함수 — 모든 함수를 한 표에
- 스칼라 함수 — `startNode`, `endNode`, `start_id`, `end_id`
- 리스트 함수 — 경로에 대한 `cardinality`
- 구문 — 경로 패턴과 유연한 길이

타입 변환 함수

값을 다른 타입으로 바꾸는 함수들입니다. 두 계열이 있습니다. `to…` 함수는 값을 읽고 그것이 해당 타입의 값을 나타내지 않으면 null을 반환합니다. `to…OrNull` 변형은 같은 일을 하되 결코 오류를 내지 않습니다.

형 변환이 또 다른 변환 방법이고, Cypher는 대부분의 타입 이름에 대해 `::`를 받습니다:

```
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`와 `::time`이 예외입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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);
                                             ^
```

Cypher의 형 변환은 평범한 타입 이름을 받고, 수치·문자·비트·json·interval 표기에 대해서는 자기 규칙을 갖고 있습니다. 날짜와 시각 표기에 대한 규칙은 없습니다. `TIMESTAMP`와 `TIME`은 키워드이므로 그 둘로의 형 변환에는 그런 규칙이 필요한데 그것이 없습니다. `date`, `timestamptz`, `timetz`는 키워드가 아니어서 평범한 타입 이름으로 들어와 동작합니다. 흔한 우회 방법은 `::timestamptz`이고, 다른 하나는 그 Cypher를 SQL로 감싸는 것입니다.

예제는 **예제 데이터셋**에서 실행됩니다.

toBoolean

```
toBoolean ( value text ) -> boolean
toBoolean ( value "any" ) -> boolean
```

값이 나타내는 불리언을 반환하고, 아무것도 나타내지 않으면 null을 반환합니다.

고려할 점

- 대소문자와 무관하게 `true`, `false`, `t`, `f`, `on`, `off`, `1`, `0`을 받습니다. 그 밖의 것은 오류를 내지 않고 null을 반환합니다.
- 형 변환보다 엄격합니다. `toBoolean('yes')`는 null이지만 `'yes'::boolean`은 참입니다.
- `RETURN`은 불리언을 감싸지 않고 두므로 `t`나 `f`로 출력되고, null은 빈 칸으로 출력됩니다.

```
RETURN toBoolean('TRUE') AS t, toBoolean('false') AS f, toBoolean(true) AS already;
```

```
 t | f | already
---+---+---------
 t | f | t
```

toBooleanOrNull

```
toBooleanOrNull ( value text ) -> boolean
toBooleanOrNull ( value "any" ) -> boolean
```

값이 나타내는 불리언을 반환하거나 null을 반환합니다. **`toBoolean`**이 아예 읽을 수 없는 타입에서 오류를 낼 수 있는 자리에서, 이 함수는 null을 반환합니다.

```
RETURN toBooleanOrNull('yes') AS unrecognised, toBooleanOrNull('true') AS recognised;
```

```
 unrecognised | recognised
--------------+------------
              | t
```

toFloat

2.18에 추가

© 2026 SKAI Worldwide Co., Ltd.

```
toFloat ( value "any" ) -> double precision
```

값이 나타내는 부동소수점 수를 반환하거나 null을 반환합니다.

고려할 점

- `"any"` 를 받는 오버로드 하나이므로, 문자열·정수·`jsonb` 속성을 형 변환 없이 받습니다.
- 숫자가 아닌 값은 오류를 내지 않고 null을 반환합니다.
- 결과는 `double precision` 입니다. Cypher의 `RETURN` 은 그것을 `jsonb` 숫자로 감쌉니다.

```
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[]
```

리스트의 모든 요소를 부동소수점 수로 변환합니다.

고려할 점

- **어느 오버로드를 받는지가 반환 타입을 결정합니다.** Cypher의 리스트는 `jsonb` 이므로 `toFloatList([...])` 는 `jsonb` 를 반환합니다. `double precision[]` 을 주는 것은 PostgreSQL 배열 인자뿐이고, 그 형태는 SQL에서 닿을 수 있습니다.
- 숫자가 아닌 요소는 null이 됩니다. 리스트의 길이는 유지됩니다.

```
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

2.18에 추가

© 2026 SKAI Worldwide Co., Ltd.

```
toInteger ( value "any" ) -> bigint
```

값이 나타내는 정수를 반환하거나 null을 반환합니다.

고려할 점

- 반올림하지 않고 0쪽으로 버립니다. `toInteger(3.7)`은 3입니다.
- 불리언도 변환되며, 참은 1이 됩니다.
- 숫자가 아닌 값은 오류를 내지 않고 null을 반환합니다.
- 결과는 `bigint` 입니다.

```
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[]
```

리스트의 모든 요소를 정수로 변환합니다. 어느 오버로드가 적용되는지가 반환 타입을 결정하며, **`toFloatList`**와 정확히 같습니다. Cypher 리스트는 `jsonb`를, PostgreSQL 배열은 `bigint[]`를 줍니다.

```
RETURN toIntegerList(['1','2','x']) AS integers;
```

```
   integers
--------------
 [1, 2, null]
```

toString

```
toString ( value jsonb ) -> jsonb
```

값의 문자열 형태를 반환합니다.

고려할 점

- 이미 문자열인 값은 그대로 돌아옵니다.
- 숫자·불리언·리스트·맵 모두 변환됩니다. 결과는 `jsonb` 문자열이므로 `RETURN`은 그것을 따옴표와 함께 출력합니다.

```
RETURN toString(11.5) AS number, toString('already a string') AS string, toString(true) AS
boolean;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 number |       string       | boolean
--------+--------------------+---------
 "11.5" | "already a string" | "true"
```

toStringList

```
toStringList ( list jsonb ) -> jsonb
toStringList ( list anyarray ) -> text[]
```

리스트의 모든 요소를 문자열로 변환합니다. 어느 오버로드가 적용되는지가 반환 타입을 결정하며, **`toFloatList`**와 같습니다. Cypher 리스트는 `jsonb`를, PostgreSQL 배열은 `text[]`를 줍니다.

```
RETURN toStringList([1, 2.5, true]) AS strings;
```

```
       strings
----------------------
 ["1", "2.5", "true"]
```

toStringOrNull

```
toStringOrNull ( value "any" ) -> text
```

값의 문자열 형태를 반환하고, 그 값에 문자열 형태가 없으면 null을 반환합니다.

고려할 점

- **`toString`**과 달리 `jsonb`가 아니라 `text`를 반환합니다. Cypher의 `RETURN`이 그것을 다시 `jsonb` 문자열로 감싸므로 지면에서는 둘이 같아 보이지만, SQL에서는 그렇지 않습니다.
- 리스트는 null을 반환합니다. 리스트에는 하나의 문자열 형태가 없습니다.

```
RETURN toStringOrNull(11.5) AS number, toStringOrNull([1,2]) AS list,
       pg_typeof(toStringOrNull(1)) AS type;
```

```
 number | list |  type
--------+------+--------
 "11.5" |      | "text"
```

변환되지 않는 것

위에서 본 것처럼 `::timestamp`와 `::time`은 Cypher에 쓸 수 없습니다. 그리고 `toMap`도, `toList`도, `toDate`도 없습니다. 맵은 `{...}`로, 리스트는 `[...]`로 만들고, 날짜는 `::date`로 읽으십시오.

그래프 요소는 자기 `properties`로만 변환되고 그 밖으로는 변환되지 않습니다. `v::jsonb`가 그 맵을 주며, **`properties(v)`**와 같은 값입니다. 맵에서 정점으로 되돌리는 변환은 없습니다.

호환성

openCypher

`toInteger`, `toFloat`, `toString`, `toBoolean`은 openCypher의 이름이고 openCypher가 설명하는 대로 동작합니다. 예외 하나는, 읽을 수 없는 값에 대해 오류를 내지 않고 null을 반환한다는 점입니다. `…OrNull` 변형과 리스트 변환 세 개는 확장입니다.

© 2026 SKAI Worldwide Co., Ltd.

openCypher의 `toInteger`는 반올림하고, 이 함수는 버립니다.

관련 항목

- 모든 함수 — 모든 함수를 한 표에
- 문자열 함수 — `split`, `toLower`, `toUpper`
- 리스트 함수 — 이 변환들이 내놓는 리스트
- jsonb 연산자와 함수 — `to_jsonb`와 `#>> '{}'` 관용구

jsonb 연산자와 함수

속성은 `jsonb` **입니다.** 정점은 복합 타입 `(id graphid, properties jsonb)`이고, 간선에는 `start`와 `end`가 더해집니다. 그러므로 Cypher에서 읽는 모든 속성과 SQL에서 손대는 모든 속성 컬럼이 PostgreSQL `jsonb` 타입의 값입니다. 이 페이지의 다른 무엇보다 먼저 받아들여야 할 결과가 두 가지입니다. 문자열 속성은 따옴표와 함께 돌아오고, `jsonb` 연산자 전체가 그것에 적용됩니다.

예제는 **예제 데이터셋**에서 실행됩니다.

속성 접근 연산자

이 네 개는 왼쪽에 `jsonb` 값을 받습니다. 화살표가 겹친 둘은 `text`를 반환하고, 화살표가 하나인 둘은 `jsonb`를 반환합니다.

연산자	오른쪽 피연산자	반환 타입	결과
`->`	`text` 키 또는 `integer` 인덱스	`jsonb`	그 키나 인덱스에 있는 값
`->>`	`text` 키 또는 `integer` 인덱스	`text`	같은 값이며, 문자열은 따옴표가 벗겨진 상태
`#>`	`text[]` 경로	`jsonb`	그 경로에 있는 값
`#>>`	`text[]` 경로	`text`	같은 값이며, 문자열은 따옴표가 벗겨진 상태

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

따옴표 없는 스칼라로 꺼내기

Cypher의 `RETURN` 은 스칼라 출력 컬럼을 `jsonb` 로 감쌉니다. 그래서 `RETURN m.title` 은 따옴표까지 포함해 `"Inception"` 을 출력하고, `RETURN m.title::text` 도 그렇습니다. 그 따옴표는 `psql` 이 만든 것이 아니라 JSON의 일부입니다.

빠져나가는 길은 `#>> '{}'` 입니다. 빈 경로가 문서 전체를 고르고, `#>>` 이 그것을 `text` 로 반환합니다. Cypher를 SQL로 감싸고 거기서 적용하십시오:

```
SELECT t AS boxed, t #>> '{}' AS unboxed
FROM (MATCH (m:movie {title:'Inception'}) RETURN m.title AS t) q;
```

```
    boxed     |  unboxed
--------------+-----------
 "Inception" | Inception
```

Cypher 결과를 평범한 `text` 와 조인하거나 비교해야 할 때는 언제든 이 관용구를 꺼내 쓰십시오. 관계형 테이블과의 조인이 그런 경우입니다.

포함·존재 연산자

연산자	오른쪽 피연산자	참이 되는 조건
`@>`	`jsonb`	왼쪽 문서가 오른쪽 문서를 포함할 때
`<@`	`jsonb`	왼쪽 문서가 오른쪽 문서에 포함될 때
`?`	`text`	그 키나 문자열 요소가 있을 때
`?|`	`text[]`	그 키 중 하나라도 있을 때
`?&`	`text[]`	그 키가 모두 있을 때

```
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
```

속성 인덱스는 이 중 어느 것도 처리할 수 없습니다. 각각 키 하나와의 비교가 아니라 `properties` 전체에 대한 포함 검사이기 때문입니다. `properties` 에 걸린 GIN 인덱스는 처리할 수 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

변경 연산자

연산자	오른쪽 피연산자	결과
`||`	`jsonb`	두 문서를 병합한 것이며, 공유된 키에서는 오른쪽이 이깁니다
`-`	`text` 키 또는 `integer` 인덱스	그 키나 요소가 빠진 문서
`#-`	`text[]` 경로	그 경로의 값이 빠진 문서

```
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": {}}
```

Cypher에서는 이것들을 대신 절로 표기합니다. `SET n += {city:'Seoul'}`이 병합하고 `REMOVE n.city`가 키를 지웁니다.

jsonb에 대한 산술 연산자

PostgreSQL이 이미 `jsonb`에 대해 `-`와 `||`에 뜻을 주었으므로, AgensGraph는 그것을 산술에 다시 쓸 수 없었습니다. 그래서 이름을 백틱으로 감싼 연산자 여덟 개를 더했습니다. 속성에 `+`, `-`, `*`, `/`, `%`, `^`를 쓰면 Cypher 파서가 그것들을 내놓습니다.

연산자	구현 함수	OID	뜻
`` `+` ``	`jsonb_add`	7176	덧셈. 문자열 둘이나 리스트 둘이면 이어 붙이기
`` `-` ``	`jsonb_sub`	7178	뺄셈
`` `*` ``	`jsonb_mul`	7180	곱셈
`` `/` ``	`jsonb_div`	7182	나눗셈. 두 피연산자가 모두 정수면 정수 나눗셈
`` `%` ``	`jsonb_mod`	7184	나머지
`` `^` ``	`jsonb_pow`	7186	거듭제곱
`` `+` `` (접두)	`jsonb_uplus`	7190	단항 플러스
`` `-` `` (접두)	`jsonb_uminus`	7191	단항 마이너스

이 포크의 렉서에서 백틱은 연산자 문자이므로(`scan.l:369`), 이 이름들은 SQL에서도 입력할 수 있습니다. 두 피연산자 모두 `jsonb` 여야 합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT '"a"'::jsonb `+` '"b"'::jsonb AS strings, '[1]'::jsonb `+` '[2]'::jsonb AS lists;
```

```
 strings | lists
---------+--------
 "ab"    | [1, 2]
```

참고

Cypher의 산술은 평범한 연산자로 쓰고 — `n.a + n.b` — 파서가 이것들을 고르게 하십시오. 백틱 이름이 여기 있는 것은 `EXPLAIN` 계획이나 오류 메시지, `pg_operator` 목록에서 그것을 알아볼 수 있게 하려는 것입니다. 구현 함수를 직접 호출할 수도 있습니다. `jsonb_add('1'::jsonb, '2'::jsonb)`은 3을 반환합니다.

ag_property_text

2.18에 추가

```
ag_property_text ( properties jsonb, key text, expected_kind "char" ) -> text
```

속성의 값을 `text`로 반환하지만, 저장된 값이 요구한 JSON 종류일 때만 그렇습니다.

이름	타입	설명
`properties`	`jsonb`	읽을 속성 맵입니다.
`key`	`text`	속성의 이름입니다.
`expected_kind`	`"char"`	숫자는 `'n'`, 문자열은 `'s'`, 불리언은 `'b'`입니다.

고려할 점

- 타입 컬럼으로 승격한 속성을 읽을 때 이 함수가 쓰입니다. `->>`은 값의 텍스트를 반환하고 텍스트에는 종류가 없으므로, 그것을 통해서는 문자열 `"5"`와 숫자 `5`를 구별할 수 없습니다. 정수 컬럼이라면 문자열을 담은 속성에 대해 5라고 답해 버립니다. 이 함수는 종류를 먼저 묻습니다.
- 종류가 맞지 않으면 null을 반환하는 것이 아니라 **오류를 냅니다.** `property "n" holds a number`가 나오고 상세로 `A column promoting this property holds strings.`가 따라옵니다.
- 없는 속성, 또는 있으면서 JSON null인 속성은 null을 반환합니다.
- 손으로 쓰도록 만든 함수가 아닙니다. 속성은 Cypher에서 `n.key`로, SQL에서 `properties ->> 'key'`로 읽으십시오.

```
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      |
```


© 2026 SKAI Worldwide Co., Ltd.

cypher_to_jsonb

```
cypher_to_jsonb ( value anyelement ) -> jsonb
```

Cypher의 규칙으로 값을 `jsonb`로 감쌉니다. 그 규칙은 문자열 타입에 대해 `to_jsonb`의 규칙과 다릅니다.

고려할 점

- Cypher의 `RETURN`이 스칼라 출력 컬럼을 감쌀 때 넣는 함수가 이것입니다. 실행 계획에서 그 감싸기가 일어나는 지점에 나타납니다.
- 엄격 함수이므로 null은 null로 남습니다.
- 파서는 그래프 요소를 이 함수로 감싸기를 거부합니다. 그 거부 때문에 `coalesce(m, 0)`과 정점을 담은 `jsonb` 리스트가 실패합니다. 그 거부는 함수가 아니라 파서의 `coerce_to_jsonb()`에 있습니다. `vertex`에 대해 이름으로 직접 호출하면 이 함수는 `id`와 `tid`까지 포함한 복합 타입 전체를 반환하며, 그것은 `v::jsonb`가 주는 `properties`가 아닙니다. 그것을 원하면 **`vertex_to_jsonb`**나 형 변환을 쓰십시오.

```
SELECT cypher_to_jsonb('abc'::text) AS from_text, cypher_to_jsonb(42) AS from_integer;
```

```
 from_text | from_integer
-----------+--------------
 "abc"     | 42
```

edge_to_jsonb

```
edge_to_jsonb ( e edge ) -> jsonb
```

간선의 `properties`를 반환합니다. 형 변환 `e::jsonb`가 쓰는 함수가 이것이고, **`properties(e)`**와 같은 값을 반환합니다.

고려할 점

- 양 끝점은 포함되지 않고 속성만 포함됩니다. 끝점은 **`start_id`**와 **`end_id`**로 읽으십시오.

```
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
```

`graphid`가 `jsonb` 배열에 담긴 그래프 요소 중 하나이면 참을 반환합니다.

고려할 점

- 파서가 `id(n) IN [...]`에 대해 이 함수를 내놓아서, 요소 목록을 하나하나 벗겨 내지 않고도 검사할 수 있게 합니다. `graphid`는 `jsonb` 문자열로 감싸이고, 이 함수는 그 형태와 비교합니다.
- 손으로 쓰면 평범한 소속 검사입니다.

© 2026 SKAI Worldwide Co., Ltd.

```
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
```

`jsonb` 배열의 요소 개수를 `integer` 로 반환합니다.

고려할 점

- PostgreSQL 자신의 함수입니다. **`cardinality`** 와 **`length`** 가 Cypher 쪽 표기이고 `jsonb` 를 반환합니다. 이 함수는 네이티브 `integer` 를 반환하고, 배열이 아닌 값에 대해서는 오류를 냅니다.

```
SELECT jsonb_array_length('[1,2,3]');
```

```
 jsonb_array_length
--------------------
                  3
```

to_jsonb

```
to_jsonb ( value anyelement ) -> jsonb
```

어떤 SQL 값이든 `jsonb` 로 변환하며, 복합 타입은 객체로, 배열은 배열로 대응시킵니다.

고려할 점

- 관계형 행을 그래프로 들여오는 방법이 이것입니다.
  `LOAD FROM t AS r CREATE (:label = to_jsonb(row_to_json(r)))` 가 각 행을 `properties` 로 바꿉니다.
- 그래프 요소에 대해서는 `properties` 가 아니라 복합 타입 전체를 변환합니다. `properties` 를 주는 것은 `v::jsonb` 와 **`vertex_to_jsonb`** 입니다.

```
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
```

정점의 `properties` 를 반환합니다. 형 변환 `v::jsonb` 가 쓰는 함수가 이것이고, **`properties(v)`** 와 같은 값을 반환합니다.

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (n:person {name:'Keanu Reeves'}) RETURN vertex_to_jsonb(n) AS bag;
```

```
                  bag
-----------------------------------------
 {"born": 1964, "name": "Keanu Reeves"}
```

순서와 비교

`jsonb`에는 자기만의 전순서가 있고, 그것은 값들이 자기 타입에서 가질 순서가 아닙니다. 종류가 먼저 옵니다. null, 그다음 문자열, 그다음 숫자, 그다음 불리언, 그다음 배열, 그다음 객체입니다. 같은 종류의 값만 값으로 비교됩니다:

```
SELECT v FROM (VALUES ('1'::jsonb),('"a"'),('true'),('null'),('[1]'),('{"k":1}')) t(v) ORDER
BY v;
```

```
    v
----------
 null
 "a"
 1
 true
 [1]
 {"k": 1}
```

Cypher 결과를 정렬하거나 집계하는 곳이면 어디서나 이것이 문제가 됩니다. `jsonb` 컬럼에 대한 `min()`과 `max()`가 그 순서를 따르므로, 어떤 행에서는 숫자를 담고 어떤 행에서는 문자열을 담는 속성은 읽는 사람이 기대하는 대로 정렬되지 않습니다. 순서가 그 타입 자신의 것이어야 한다면 타입 컬럼을 통해 읽거나 형 변환을 하십시오.

정수끼리 나누면 정수 나눗셈이 됩니다. `jsonb`가 그 구별을 유지하기 때문입니다:

```
RETURN 7 / 2 AS integer_division, 7.0 / 2 AS real_division;
```

```
 integer_division |   real_division
------------------+--------------------
 3                | 3.5000000000000000
```

관련 항목

- 모든 함수 — 모든 함수를 한 표에
- 스칼라 함수 — `properties`, `keys`, `length`
- 타입 변환 함수 — `to…` 계열
- SQL과 Cypher 섞기 — `#>> '{}'`이 값어치를 하는 곳

서버 설정

AgensGraph는 PostgreSQL의 수백 개 설정에 열한 개를 더합니다. 이 페이지는 그 열한 개를 문서화합니다. 그 밖의 모든 설정은 PostgreSQL의 것이며 바뀌지 않았고, **PostgreSQL의 서버 설정** 장이 그것을 설명합니다. 무엇을 **어떻게** 그리고 **어디에** 설정하는지는 `postgresql.conf` · `ALTER SYSTEM` · `ALTER DATABASE` · `ALTER ROLE` · `SET`을 다루는 설정을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

열한 개 중 하나인 `graph_path`는 Cypher를 실행하는 모든 세션에 필요합니다. 나머지는 쓸 만한 기본값을 가지고 있으므로, 바꿀 이유가 생기기 전까지는 신경 쓰지 않아도 됩니다.

설정 목록

아래의 모든 기본값은 실행 중인 서버에서 읽은 `pg_settings.boot_val`입니다:

```
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
 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
```

두 가지 context 값은 이렇게 읽습니다:

`user`	어느 롤이든, 어느 세션에서든, 언제든 바꿀 수 있습니다. `postgresql.conf`에서, 또는 데이터베이스별· 롤별로도 설정할 수 있습니다.
`superuser`	세션에서 이것을 바꿀 수 있는 것은 슈퍼유저, 또는 그 매개변수에 `SET` 권한을 부여받은 롤뿐입니다. 그 밖의 모두는 `permission denied to set parameter`를 받습니다.

`postgresql.conf.sample`을 읽는 대신 `pg_settings`를 조회하십시오. 설치 디렉터리에 함께 들어 있는 샘플 파일은 `enable_eager`와 `enable_multiple_update`를 `off`로 적어 두었습니다. 둘의 실제 기본값은 `on`이므로, 그 파일을 옮겨 적은 표는 두 행이 틀리게 됩니다.

agversion — 설정이 아니라 프리셋

`agversion`은 AgensGraph 버전을 알려 줍니다. context가 `internal`이므로 이것은 **프리셋**입니다. 설정하는 것이 아니라 서버가 알려 주는 값입니다. `postgresql.conf`에 쓸 수 없고, 설정하려 하면 거부됩니다:

```
SET agversion = '1';
```

```
ERROR:  parameter "agversion" cannot be changed
```


© 2026 SKAI Worldwide Co., Ltd.

```
SELECT name, setting, context, category FROM pg_settings WHERE name = 'agversion';
```

```
   name    |  setting   | context  |    category
-----------+------------+----------+-----------------
 agversion | 2.18.6.1 | internal | Preset Options
```

무엇에 접속해 있는지 확인하는 보통의 방법인 `SHOW`로 읽습니다:

```
SHOW agversion;
```

```
 agversion
------------
 2.18.6.1
```

`version()` 과 `server_version`은 이 릴리스가 기반으로 삼은 PostgreSQL 버전을 알려 줍니다. 둘 다 알아야 하는 클라이언트는 둘 다 읽어야 합니다.

graph_path

```
string · default '' (empty) · context user
```

세션이 작업하는 그래프를 지정합니다. Cypher나 GQL 문장에서 한정하지 않은 모든 레이블 이름이 그 그래프에서 해석됩니다. 모든 그래프 DDL 문장이 거기에 레이블을 만들거나 변경합니다. 그래프판 `search_path`이며, 건너뛸 수 없는 유일한 설정입니다.

이것이 없으면 아무것도 해석되지 않습니다:

```
RESET graph_path;
MATCH (n:person) RETURN count(n);
```

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

설정하면 같은 문장이 실행됩니다:

```
SET graph_path = moviekb;
SHOW graph_path;
```

```
 graph_path
------------
 moviekb
```

`CURRENT_GRAPH`는 문장 안에서 키워드만으로 같은 값을 읽습니다:

```
SELECT CURRENT_GRAPH;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 current_graph
---------------
 moviekb
```

그래프 이름은 정확히 하나

`search_path`와 달리 `graph_path`는 목록이 아닙니다. 쉼표로 구분한 값을 이름 두 개로 해석하지 않습니다. 이름 하나로 받아들이며, 그런 이름의 그래프는 존재하지 않습니다:

```
SET graph_path = 'moviekb, twins';
```

```
ERROR:  invalid value for parameter "graph_path": "moviekb, twins"
DETAIL:  graph "moviekb, twins" does not exist.
```

그러므로 세션 하나는 그래프 하나를 읽고 씁니다. 한 문장에서 그래프 두 개에 닿으려면 둘을 스키마 두 개로 보는 SQL을 쓰십시오. 그 방법은 SQL과 Cypher 섞기에 있습니다.

값을 언제 검사하는가

존재하지 않는 그래프는 쿼리를 실행할 때가 아니라 `SET` 하는 시점에 거부됩니다:

```
SET graph_path = nosuchgraph;
```

```
ERROR:  invalid value for parameter "graph_path": "nosuchgraph"
DETAIL:  graph "nosuchgraph" does not exist.
```

이름은 식별자이므로 따옴표로 감싸지 않으면 소문자로 바뀝니다. 대소문자를 섞은 이름으로 만든 그래프에는 따옴표가 필요합니다:

```
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;
```

데이터베이스나 롤에 값 하나 지정하기

`SET`은 그 세션 동안만 유지됩니다. 더 넓은 범위 두 가지가 그 되풀이를 덜어 주며, 둘 다 지금 접속이 아니라 다음 접속부터 적용됩니다.

© 2026 SKAI Worldwide Co., Ltd.

어떤 데이터베이스에 접속하는 모두에게 적용하려면:

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

```
ALTER DATABASE
```

어느 데이터베이스에 접속하든 특정 롤 하나에 적용하려면:

```
ALTER ROLE analyst SET graph_path = moviekb;
```

```
ALTER ROLE
```

둘 다 `pg_db_role_setting`에서 보이며, 롤 설정이 데이터베이스 설정을 이깁니다:

```
SELECT r.rolname, s.setconfig FROM pg_db_role_setting s
JOIN pg_roles r ON r.oid = s.setrole WHERE r.rolname = 'analyst';
```

```
 rolname |      setconfig
---------+----------------------
 analyst | {graph_path=moviekb}
```

그 뒤에 접속하는 롤에는 이미 값이 들어 있습니다:

```
SHOW graph_path;
```

```
 graph_path
------------
 moviekb
```

`ALTER DATABASE … RESET graph_path`와 `ALTER ROLE … RESET graph_path`가 그것을 다시 없앱니다. `postgresql.conf`에 클러스터 전체 기본값을 두는 것도 되지만, 원하는 바인 경우는 드뭅니다. 클러스터의 모든 데이터베이스가 그 그래프를 가진 것처럼 말하게 되기 때문입니다.

값은 카탈로그와 대조해 검사하므로, `ALTER DATABASE db SET graph_path`는 `db`에 접속한 상태에서 실행해야 합니다. 검사는 **현재** 데이터베이스에서 그 그래프를 찾습니다. 다른 곳에서 실행하면 실패합니다:

```
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
```

null 속성 값을 저장할 것인지 버릴 것인지 결정합니다.

© 2026 SKAI Worldwide Co., Ltd.

꺼져 있으면 null을 대입하는 것이 속성을 지웁니다. 그러면 null로 설정한 속성과 애초에 없던 속성이 같은 것이 됩니다. 켜면 null이 JSON null로 저장되고 키가 남습니다.

```
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}
```

값이 없는 것과 null인 것의 뜻이 다르고 그 구별을 지켜야 하는 데이터를 적재할 때 켜십시오. 보통 관계형 원본에서 가져오는 경우입니다.

대가는 저장된 null이 값 없이 존재하는 키가 된다는 것입니다. 검사들이 서로 다르게 답합니다:

```
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`는 이 설정이 무엇이든 키를 지웁니다. 속성을 없애는 방법은 여전히 이것입니다.

```
RESET allow_null_properties;
DROP GRAPH nulls CASCADE;
SET graph_path = moviekb;
```

auto_gather_graphmeta

```
bool · default off · context superuser
```

모든 그래프 쓰기의 부수 효과로 `ag_graphmeta`를 최신 상태로 유지합니다.

`ag_graphmeta`는 그래프의 (시작 레이블, 간선 레이블, 끝 레이블) 조합마다 간선이 몇 개인지 기록합니다. 플래너는 그것으로 스캔을 쳐냅니다. 어떤 레이블과 도저히 일치할 수 없는 `MATCH`는 아예 스캔하지 않습니다. 이 설정이 꺼져 있으면 그 테이블은 마지막 `regather_graphmeta()` 호출 시점까지만 최신입니다.

켜면 즉시 전체 수집이 일어나므로, 그 시점부터 테이블이 정확해집니다:

```
SET auto_gather_graphmeta = on;
```

```
NOTICE:  regathering graphmeta...
SET
```


© 2026 SKAI Worldwide Co., Ltd.

```
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;
```

읽기가 대부분인 그래프에는 켜십시오. 쓰기마다 드는 약간의 일보다 정확한 메타데이터가 값어치가 있습니다. 쓰기가 많은 적재에는 꺼 둔 채로 두고 끝난 뒤에 `regather_graphmeta()`를 호출하십시오. `pg_upgrade`나 복원 뒤에도 반드시 호출해야 합니다. 그 테이블과 그것이 가능하게 하는 쳐내기는 **그래프 메타데이터**가 다룹니다.

cypher_allow_unsafe_ddl

```
bool · default off · context user
```

카탈로그와 더는 맞지 않는 그래프에 평소에는 접근하지 못하게 막는 카탈로그 검사 두 개를 건너뜁니다. 그래프를 고치기 위한 것이지, 그 상태로 작업하기 위한 것이 아닙니다.

`graph_path`가 존재하지 않는 그래프를 가리킬 수 있게 합니다. 검사가 `SET` 시점에서 쿼리 실행 시점으로 옮겨 가고, 그때의 메시지는 다릅니다:

```
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;
```

레이블을 잘못된 종류로 삭제할 수 있게 합니다. 간선 레이블에 `DROP VLABEL`을 쓰는 것은 평소에는 거부됩니다. 이 설정을 켜면 성공합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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;
```

둘이 쓸모 있는 상황은 정확히 하나입니다. 카탈로그와 테이블이 어긋난 그래프를 해체하려는 경우입니다. 그것이 필요한 문장에만 설정하고 곧바로 되돌리십시오.

eager_mem

```
integer · default 4096 · unit kB · context user · minimum 1024
```

eager 계획 노드 하나가 디스크로 넘기기 전에 쓸 수 있는 메모리 양입니다. eager 계획을 위한 `work_mem` 이며, `work_mem` 이 그렇듯 쿼리별이나 접속별이 아니라 노드별입니다.

```
SHOW eager_mem;
SET eager_mem = '64MB';
SHOW eager_mem;
```

```
 eager_mem
-----------
 4MB
(1 row)

SET
 eager_mem
-----------
 64MB
(1 row)
```

최솟값은 1 MB이며, 더 작은 값은 잘라 맞추는 것이 아니라 거부됩니다:

```
SET eager_mem = '512kB';
```

```
ERROR:  512 kB is outside the valid range for parameter "eager_mem" (1024 kB .. 2147483647 kB)
```


© 2026 SKAI Worldwide Co., Ltd.

```
RESET eager_mem;
```

자기가 쓴 것을 읽는 쓰기가 디스크로 넘어가고 있다면 올리십시오. `enable_eager`를 참고하십시오. 기본값 4 MB는 `work_mem`의 것과 같은 보수적인 출발점입니다. 큰 중간 결과를 구체화하는 문장은 이 값을 넘어섭니다.

enable_eager

```
bool · default on · context user
```

플래너가 eager 계획, 즉 뒤의 절이 읽기 전에 쓰기의 출력을 구체화하는 계획을 만들 수 있게 합니다.

어떤 문장에는 그것이 필요합니다. `CREATE` 다음에, 그 `CREATE`가 방금 만든 것과 일치할 수 있는 `MERGE`가 오는 경우를 보십시오. 둘을 행 단위로 번갈아 실행할 수 없습니다. 두 번째 절이 절반만 쓰인 그래프를 보게 되기 때문입니다. 그래서 플래너는 첫 절의 결과를 버퍼에 담습니다. `EXPLAIN`은 그 노드를 `eager`로 표시합니다:

```
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});
```

```
                                                 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>"
```

이것을 끈다고 해서 그런 문장이 다른 방식으로 실행되지는 않습니다. 실패하게 됩니다:

```
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;
```

그러므로 이것은 PostgreSQL의 `enable_hashjoin` 같은 비용 기반 토글이 아닙니다. 켜 두십시오. 어떤 문장이 eager 계획에 의존하는지 알아보려 할 때만 끄십시오. 이를 아는 것은 큰 쓰기를 더 싸게 만들려 할 때 중요해집니다.

enable_graph_ddl

```
bool · default off · context user
```


© 2026 SKAI Worldwide Co., Ltd.

SQL DDL이 그래프 레이블의 모양을 바꿀 수 있게 합니다.

레이블을 그래프 레이블이게 하는 것은 그 컬럼들이고, 그래프 카탈로그가 그것을 설명합니다. `ALTER TABLE`은 그 카탈로그를 갱신하지 않습니다. 그것으로 레이블의 모양을 바꾸면 그래프가 더는 사실이 아닌 것을 설명하게 될 수 있습니다. 꺼져 있으면 그런 시도는 거부되고, 카탈로그를 함께 맞춰 주는 문장을 힌트로 알려 줍니다:

```
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;
```

```
ERROR:  cannot remove a parent label from graph label "movie" with ALTER TABLE
HINT:  A label's parents are fixed when it is created.
```

거부되는 것의 전체 목록입니다. 컬럼을 더하거나 지우거나 타입을 바꾸는 것, 컬럼의 생성 표현식을 떼어 내거나 바꾸는 것, not-null 제약을 컬럼으로 지목하든 제약 이름으로 지목하든 지우는 것, 부모 레이블을 더하거나 없애는 것입니다. 같은 설정이 레이블의 컬럼 이름을 바꾸는 것과 레이블을 다른 스키마로 옮기는 것도 거부합니다.

켜면 그 모두가 풀립니다. 무슨 권한을 주는 것은 아닙니다. 그 레이블을 변경할 수 있는 사람이 여전히 그 레이블에 무슨 일이 일어날지 정합니다. 모양을 바꾸는 일이 의도된 것이기를 요구할 뿐입니다.

절대 풀리지 않는 것이 둘 있습니다. `id` · `start` · `end` · `properties`의 타입은 전혀 바꿀 수 없습니다. 모든 정점과 간선이 그 컬럼들로 이루어져 있기 때문입니다:

```
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;
```

그리고 테이블 자체의 이름을 바꾸거나 지우는 것은 각자의 메시지와 함께 계속 거부됩니다 (`cannot rename table "movie"`, `cannot drop table … because vlabel … requires it`). `ALTER VLABEL … RENAME TO`와 `DROP VLABEL`을 쓰십시오.

직접 설정하든 하지 않든 이 설정을 만나게 됩니다. 평범한 컬럼을 지닌 레이블에 대해 `pg_dump`는 복원이 되도록 `ALTER TABLE` 문장을 `SET enable_graph_ddl = on`으로 감쌉니다. **서버 측 도구**를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

enable_graph_dml

```
bool · default off · context superuser
```

그래프 레이블에 대한 SQL `INSERT` · `UPDATE` · `DELETE` · `MERGE` 를 허용합니다.

SQL로 레이블을 읽는 데는 설정이 필요 없습니다. 쓰는 데는 필요합니다. Cypher의 쓰기 경로가 그래프를 일관되게 지키기 때문입니다. 지키는 것은 존재하는 양 끝점, 레이블과 맞는 id, 그리고 그에 딸린 메타데이터입니다. 꺼져 있으면 모든 SQL 쓰기가 거부됩니다:

```
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;
```

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

켜면 그래프가 보이는 그대로 일관되다는 보장을 포기하게 됩니다. 간선 레이블에 대한 `INSERT` 는 양 끝점이 존재하는지 검사하지 않으므로, 끝점이 없는 간선을 만들 수 있습니다. Cypher로는 그런 것을 만들 수 없습니다:

```
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
```

그 결과는 서로 다른 두 쿼리가 다르게 답하는 간선입니다. 양 끝점을 모두 바인딩하는 패턴은 아무것도 찾지 못합니다. 없는 정점과의 조인이 실패하기 때문입니다:

```
MATCH (a)-[r:e]->(b) RETURN a, r, b;
```

```
 a | r | b
---+---+---
(0 rows)
```

양 끝점을 하나도 바인딩하지 않는 패턴은 그것을 셉니다. 조인할 것이 없기 때문입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH ()-[r:e]->() RETURN count(r) AS edges;
```

```
 edges
-------
 1
```

```
DROP GRAPH dangling CASCADE;
SET graph_path = moviekb;
```

조심해서 작성한 대량 적재를 할 때, 또는 Cypher가 닿지 못하는 행을 고칠 때 켰다가 다시 끄십시오. 이 설정은 `superuser` 설정 둘 중 하나입니다. 평범한 롤이 `SET enable_graph_dml = on`을 시도하면 `ERROR:  permission denied to set parameter "enable_graph_dml"`을 받으며, 스스로 켤 수 없습니다.

enable_graph_endpoint_elision

```
bool · default on · context user
```

쿼리가 전혀 쓰지 않는 패턴 끝점에 대한 스캔을 플래너가 없앨 수 있게 합니다.

`->()` 같은 익명 끝점은 조인 말고는 보태는 것이 없습니다. 그 끝점에 레이블도 속성 조건도 없다면 그 조인은 행을 걸러 내지도 못합니다. 플래너는 그것을 없앱니다. 이름이 레이블을 제한하는 데만 쓰인 경우에는 이름 있는 끝점도 없앱니다. 간선의 끝점 id에 대한 레이블 검사는 그 간선 자체에 대한 `graphid` 범위 검사이기 때문입니다:

```
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))
```

이것이 없으면 같은 쿼리가 릴레이션 세 개를 조인하고, 들여다보지도 않을 끝점을 위해 정점 계층 전체를 스캔합니다:

```
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"
```


© 2026 SKAI Worldwide Co., Ltd.

```
                      ->  Seq Scan on movie "<0000000001>_2"
                      ->  Seq Scan on person "<0000000001>_3"
```

```
RESET enable_graph_endpoint_elision;
```

열한 개 중 `EXPLAIN (SETTINGS)`가 보고하는 것은 이것뿐이므로, 이 값을 바꾼 세션에서 뽑은 계획은 그 사실을 함께 알려 줍니다. 계획을 비교할 때만 끄십시오.

enable_multiple_update

```
bool · default on · context user
```

한 문장에서 같은 요소를 두 번 이상 쓰는 것을 조용히 넘길지 경고할지 결정합니다.

패턴이 어떤 요소와 여러 번 일치하는 문장은 일치할 때마다 그것을 씁니다. 켜져 있으면 그 일이 조용히 일어납니다. 꺼져 있으면 서버가 되풀이마다 경고합니다. 쓰기는 그대로 일어나므로, 이것은 제한이 아니라 진단 장치입니다:

```
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}
```

크로스 조인이 행 네 개를 만들어 각 정점에 두 번씩 닿았지만, `n`은 1만 증가했습니다. 설정을 끄면 같은 문장이 그 사실을 알려 줍니다:

```
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;
```

`SET`이 기대한 대로 하지 않은 이유를 따질 때 꺼 두십시오. 되풀이된 쓰기는 대개 패턴이 의도보다 많이 일치한다는 신호입니다. 경고는 레이블 id와 로컬 id를 알려 주는데, 그것이 그 요소의 `graphid`입니다.

© 2026 SKAI Worldwide Co., Ltd.

enable_property_promotion

```
bool · default on · context user
```

승격된 타입 컬럼이 있을 때 Cypher의 속성 읽기를 그 컬럼에서 답할지 결정합니다.

속성은 실제 타입을 가진 실제 컬럼으로 구체화할 수 있습니다. **속성을 컬럼으로 승격하기**를 참고하십시오. 켜져 있으면 `n.key`가 그 컬럼을 읽고 그 컬럼의 타입으로 돌아옵니다. 꺼져 있으면 모든 읽기가 `jsonb` 컬럼인 `properties`로 가서 `jsonb`로 돌아옵니다:

```
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;
```

```
 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"
```

경고

비용에만 관여하는 플래너 토글과 달리, 이것을 끄면 쿼리 **결과**가 달라질 수 있습니다. 해석된 `n.key`는 그 타입 컬럼의 고유한 의미 — 수치 또는 콜레이션 정렬, 그리고 타입을 아는 비교 — 를 쓰는 반면, `jsonb` 대체 경로는 `jsonb`의 정렬과 비교를 씁니다. 정렬이나 범위 조건, 타입이 다른 값끼리의 비교가 다르게 나올 수 있습니다.

```
RESET enable_property_promotion;
DROP GRAPH promoted CASCADE;
SET graph_path = moviekb;
```

켜 두십시오. 승격된 컬럼이 쿼리의 답을 바꿨는지 확인할 때 끄십시오. 그리고 다시 켠 뒤에 쿼리나 컬럼 쪽을 고치십시오.

관련 항목

- 설정 — 설정을 어디에 지정할 수 있는지, 그리고 그래프 워크로드에 중요한 PostgreSQL 설정
- 구문 — `graph_path`의 값이 따르는 어휘 규칙
- 데이터 타입 — `allow_null_properties`와 `enable_property_promotion`이 속성의 타입에 관해 바꾸는 것
- 그래프 메타데이터 — `auto_gather_graphmeta`가 유지하는 테이블
- 서버 측 도구 — `pg_dump` · `pg_upgrade`와 그것들이 기대는 설정
- PostgreSQL: 서버 설정 — 그 밖의 모든 설정

© 2026 SKAI Worldwide Co., Ltd.

agens 클라이언트

`agens`는 대화형 터미널 클라이언트입니다. PostgreSQL의 `psql`을 AgensGraph 이름으로 부르는 것이며, 여기에 그래프를 설명하는 메타 명령 다섯 개가 더해져 있습니다.

`psql`이므로 모든 옵션과 모든 변수와 그 밖의 모든 메타 명령은 PostgreSQL의 **`psql` 레퍼런스**가 문서화합니다. 그것을 여기서 되풀이하지 않습니다. 이 페이지가 다루는 것은 둘입니다. 다른 어디에도 없는 그래프 메타 명령, 그리고 AgensGraph의 것인 환경 세부 두 가지입니다.

이름 둘을 가진 같은 프로그램

빌드는 같은 소스에서 바이너리 두 개를 만들어 둘 다 설치합니다. 차이는 브랜딩뿐입니다. 각자가 출력하는 이름, 그리고 도움말의 버그 신고 줄과 홈페이지 줄입니다.

```
agens --version
psql --version
```

```
agens (PostgreSQL) 18.6 (AgensGraph 2.18.6.1)
psql (PostgreSQL) 18.6 (AgensGraph 2.18.6.1)
```

두 도움말을 diff 하면 차이 전부가 드러납니다:

```
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/>
```

여기서 결론 둘이 따라오고, 둘 다 일을 덜어 줍니다.

`psql`에 대해 아는 것이 그대로 적용됩니다. 같은 옵션, 같은 `~/.psqlrc`, 같은 `PG*` 환경 변수, 같은 `\copy`, `\watch`, `\gexec`, `\timing`, `\x`, `\pset`, 같은 탭 완성과 변수 치환입니다.

```
agens -d docs -c "SHOW agversion;"
```

```
 agversion
------------
 2.18.6.1
```

그래프 메타 명령은 두 바이너리에 다 있습니다. 둘은 같은 오브젝트로 빌드되므로, 이 설치본의 `psql`도 `agens`와 똑같이 그래프를 나열합니다. 이미 `psql`이 경로에 있다면 그래프를 설명하려고 클라이언트를 바꿀 필요가 없습니다.

© 2026 SKAI Worldwide Co., Ltd.

서버 도구도 같은 방식을 따르며, 이름이 바뀐 쌍이 하나 있습니다:

AgensGraph 이름	PostgreSQL 이름	차이
`agens`	`psql`	브랜딩뿐입니다. 둘 다 설치됩니다.
`ag_ctl`	`pg_ctl`	브랜딩뿐입니다. 둘 다 설치됩니다.

그 밖의 모든 도구는 PostgreSQL 이름을 그대로 지니며 그 이름뿐입니다. `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`와 나머지가 그렇습니다. `ag_dump`도 `ag_basebackup`도 없습니다. 이 도구들이 그래프를 어떻게 다루는지는 **서버 측 도구**가 다룹니다.

그래프 메타 명령

메타 명령 다섯 개가 그래프를 설명합니다. `+`는 각각에 컬럼을 더하고, 패턴은 나열 대상을 좁힙니다.

명령	나열하는 것
`\dG[+] [PATTERN]`	그래프입니다.
`\dGl[+] [PATTERN]`	레이블 두 종류 모두입니다.
`\dGv[+] [PATTERN]`	정점 레이블, 그리고 각각의 설명입니다.
`\dGe[+] [PATTERN]`	간선 레이블, 그리고 각각의 설명입니다.
`\dGi[+] [PATTERN]`	속성 인덱스입니다.

대소문자가 중요합니다. `\dG`는 그래프를 나열합니다. `\dg`는 PostgreSQL 자신의 명령이며 롤을 나열합니다.

이 다섯이 전부이고, 그중 어느 것도 `\?`에 나오지 않습니다. 다른 `\d` 명령이 받는 `x`와 `S` 수식자는 `\dG` 바로 뒤에서는 받아들여지지 않습니다:

```
\dGx moviekb
```

```
invalid command \dGx
```

`x`는 하위 명령 네 개에서는 동작하므로, `\dGvx`는 목록을 확장 모드로 출력합니다.

아래 예제는 모두 **예제 데이터셋**을 상대로 실행합니다.

데이터베이스의 그래프 나열하기

```
\dG moviekb
```


© 2026 SKAI Worldwide Co., Ltd.

```
 List of graphs
  Name   | Owner
---------+-------
 moviekb | agens
```

`+`는 그래프의 접근 권한과 주석을 더합니다. 그래프는 스키마이므로 그 권한은 스키마 권한입니다. 무엇인가 부여하기 전에는 비어 있으므로, 컬럼이 채워진 것을 보려면 하나를 부여하십시오:

```
GRANT USAGE ON SCHEMA moviekb TO PUBLIC;
```

```
\dG+ moviekb
```

```
                 List of graphs
  Name   | Owner | Access privileges | Description
---------+-------+-------------------+-------------
 moviekb | agens | agens=UC/agens   +|
         |       | =U/agens          |
```

예제 데이터셋을 원래대로 두기 위해 다시 회수합니다:

```
REVOKE USAGE ON SCHEMA moviekb FROM PUBLIC;
```

패턴을 주지 않으면 `\dG`는 데이터베이스의 모든 그래프를 나열합니다.

그래프의 레이블 나열하기

`\dGl`은 두 종류를 함께 나열하며, 그래프·종류·이름 순으로 정렬합니다:

```
\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
```

모든 그래프가 `ag_vertex`와 `ag_edge`를 가지므로 그 둘이 함께 나옵니다. **레이블 상속**을 참고하십시오.

`+`는 레이블의 디스크 크기와 주석을 더합니다. 크기는 레이블 테이블의 `pg_table_size`이므로, 적재한 만큼을 반영합니다:

```
\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
```


© 2026 SKAI Worldwide Co., Ltd.

```
 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      |
```

정점 레이블 설명 보기

`\dGv`는 목록을 정점 레이블로 좁힌 다음, `\d`가 테이블의 꼬리말을 출력하듯 각각에 대한 블록을 출력합니다. 블록에는 그 레이블의 속성 인덱스와 제약, 그 레이블이 상속하는 레이블과 그 레이블을 상속하는 레이블이 담깁니다:

```
\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
```

인덱스와 제약이 있는 레이블은 둘 다 보여 줍니다:

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

블록이 보여 주지 **않는** 것에 유의하십시오. 레이블의 컬럼입니다. `id` 와 `properties` , 그리고 승격된 컬럼을 보려면 레이블이 테이블이므로 그 테이블에 `\d` 를 쓰십시오:

```
\d moviekb.movie
```

```
                                                  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
```

간선 레이블 설명 보기

`\dGe` 는 간선에 대한 `\dGv` 이며, 블록의 모양도 같습니다:

```
\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
```

이 블록을 출력하지 않는 것이 `\dGl` 입니다. 그것은 두 종류의 요약일 뿐, 그 이상이 아닙니다.

속성 인덱스 나열하기

`\dGi` 는 `ag_property_indexes` 를 읽어 각 인덱스의 Cypher 정의, 즉 다시 만들 때 입력할 그 문장을 보여 줍니다. 속성 인덱스가 없는 그래프는 아무것도 나열하지 않습니다:

```
\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)
```


© 2026 SKAI Worldwide Co., Ltd.

`+`는 인덱스의 크기와 주석을 더합니다:

```
\dGi+ demo.*
```

```
                                                                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 |
```

고유성 제약은 속성 인덱스가 아니므로 여기에 나오지 않습니다. 그것은 자기 레이블의 `\dGv`나 `\dGe` 블록에, 그리고 `\d`에서는 배제 제약으로 나타납니다.

```
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
```

패턴으로 목록 좁히기

패턴은 PostgreSQL의 `\d` 패턴 언어이며, 점으로 구분한 이름 두 부분에 적용됩니다. 그 언어에는 `*`와 `?` 와일드카드, 문자 클래스, 그리고 대소문자 변환을 막는 큰따옴표가 있습니다.

그 두 부분이 무엇인지는 명령마다 다르고, 그 차이에서 자주 헷갈립니다:

명령	첫째 `.` 둘째의 뜻
`\dG`	그래프 이름입니다. 둘째 부분은 없습니다.
`\dGl`, `\dGv`, `\dGe`	그래프, 그다음 **레이블** 이름입니다.
`\dGi`	그래프, 그다음 **인덱스** 이름입니다.

그래서 `\dGi moviekb.movie`는 아무것도 찾지 못합니다. `movie`가 인덱스 이름과 대조되기 때문입니다:

```
\dGi moviekb.movie
```

```
No matching property indexes found.
```

대신 인덱스 이름을 대거나, `moviekb.*`를 쓰고 `LabelName` 컬럼을 읽으십시오. 아무것도 일치하지 않는 패턴은 그렇다고 말하며, 오류가 아닙니다:

© 2026 SKAI Worldwide Co., Ltd.

```
\dGv nosuchgraph.*
```

```
No matching labels found.
```

AGDATA

`AGDATA` 는 `PGDATA` 가 그러하듯 데이터 디렉터리를 가리킵니다. 이것을 읽는 프로그램은 정확히 셋입니다. `initdb` 와 `pg_ctl` 과 `ag_ctl` 입니다. 다른 도구는 읽지 않습니다. `agens` 도, `psql` 도, `pg_dump` 도, `pg_basebackup` 도 읽지 않습니다. 그러므로 모든 것이 존중하는 변수를 설정하고 싶다면 여전히 `PGDATA` 를 설정해야 합니다.

한 번 export 해 두면 그 셋은 `-D` 가 필요 없어집니다:

```
export AGDATA=/tmp/agensdata
ag_ctl status
```

```
ag_ctl: no server running
```

둘 다 읽는 두 프로그램 사이에서 우선순위가 다르므로, 둘 다 설정하기 전에 확인하십시오:

프로그램	순서
`initdb`	`-D` , 그다음 `AGDATA` , 그다음 `PGDATA` 입니다.
`pg_ctl` 과 `ag_ctl`	`-D` , 그다음 `PGDATA` , 그다음 `AGDATA` 입니다.

`initdb` 는 결정을 내린 뒤 자기 환경에도 `AGDATA` 를 설정하며, 마지막 힌트는 `ag_ctl` 을 가리킵니다:

```
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
```

두 변수 모두 설정하지 않고 `-D` 도 주지 않으면 메시지는 `PGDATA` 만 가리킵니다. `AGDATA` 도 똑같이 동작했을 것입니다:

```
ag_ctl status
```

```
ag_ctl: no database directory specified and environment variable PGDATA unset
Try "ag_ctl --help" for more information.
```

데이터 디렉터리를 어디에 두고 init 시스템에서 서버를 어떻게 기동하는지는 **배포**가 다룹니다.

관련 항목

- **서버 측 도구** — `ag_ctl` · `pg_dump` · `pg_upgrade` 와 그래프
- **예제 데이터셋** — 위의 모든 목록이 설명하는 그래프

© 2026 SKAI Worldwide Co., Ltd.

- **시스템 카탈로그** — 이 명령들이 조회하는 `ag_graph` · `ag_label` · `ag_property_indexes`
- **서버 설정** — 세션이 Cypher를 실행하기 전에 필요한 `graph_path`
- **PostgreSQL: psql** — 모든 옵션과 그 밖의 모든 메타 명령
- **PostgreSQL: pg_ctl** — `ag_ctl`이 무엇인가

© 2026 SKAI Worldwide Co., Ltd.

가이드

AgensGraph 위에서 무엇인가를 만들 때 작업별로 따라 하는 안내입니다.

데이터 적재

그래프에 데이터를 넣는 방법은 셋이며, 무엇이 맞는지는 데이터의 양에 달려 있습니다.

경로	알맞은 경우	속도
Cypher `CREATE` / `MERGE`	요소 수십만 개까지. Cypher의 검사를 원하는 모든 경우	행당 가장 느리고 가장 안전합니다
테이블에서 `LOAD FROM`	이미 테이블에 있는 데이터, 또는 외부 테이블로 노출한 파일	빠르며 Cypher 안에 머무릅니다
레이블 테이블에 `COPY`	직접 다루는 파일에서 오는 수백만 개의 요소	큰 차이로 가장 빠릅니다

어느 것을 쓰든 **대량 적재 중의 인덱스** 절이 적용되고, **적재를 마친 뒤** 점검 목록도 적용됩니다. 갓 적재한 그래프에서 쿼리가 나쁘다면 대개 두 번째를 건너뛰었기 때문입니다.

Cypher CREATE

가장 뻔한 경로이며, 너무 느려지기 전까지 쓸 방법입니다. 묶어서 하십시오. 요소를 여러 개 만드는 문장 하나가, 하나씩 만드는 문장 여러 개보다 낫습니다:

```
CREATE (:person {name: 'Ann', born: 1990}),
       (:person {name: 'Bo',  born: 1985}),
       (:person {name: 'Cy',  born: 1979});
```

간선은 양 끝점을 먼저 찾아야 하므로 같은 문장에서 함께 일치시키십시오:

```
MATCH (a:person {name: 'Ann'}), (b:person {name: 'Bo'})
CREATE (a)-[:knows {since: 2019}]->(b);
```

간선을 `MATCH` 하나씩 적재하는 것이 이 경로가 느린 이유입니다. 문장마다 양 끝점을 다시 찾기 때문입니다. 간선을 대량으로 적재한다면 다른 두 경로 중 하나를 써서 서버가 그 탐색을 조인으로 하게 하십시오.

적재가 두 번 돌 수 있고 그것이 멱등이기를 바란다면 `CREATE` 대신 `MERGE`를 쓰십시오. 다만 요소마다 탐색 비용이 든다는 점을 알아두십시오.

테이블에서 LOAD FROM

`LOAD FROM`은 평범한 테이블에서 행을 읽어 Cypher 쓰기 절에 넘기므로, 끝점 탐색이 행마다의 쿼리가 아니라 조인 하나가 됩니다. 데이터가 이미 관계형일 때 자연스러운 경로입니다. `file_fdw`로 외부 테이블로 노출한 파일도 여기에 들어갑니다.

행이 이미 스테이징 테이블 두 개에 들어와 있다고 합시다:

```
CREATE TABLE staging_people (name text, born int);
INSERT INTO staging_people VALUES ('Ann', 1990), ('Bo', 1985);
```


© 2026 SKAI Worldwide Co., Ltd.

```
CREATE TABLE staging_knows (from_name text, to_name text, since int);
INSERT INTO staging_knows VALUES ('Ann', 'Bo', 2019);
```

그러면 문장 하나가 각 행을 정점으로 바꿉니다:

```
LOAD FROM staging_people AS row
CREATE (:person {name: row.name, born: row.born});
```

또는 행 전체를 속성 맵으로 받을 수도 있습니다:

```
LOAD FROM staging_people AS row
CREATE (:person = to_jsonb(row_to_json(row)));
```

중요한 경우는 간선입니다. 속도가 나오는 곳이 그 조인이기 때문입니다:

```
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);
```

그것을 실행하기 전에 일치에 쓰는 속성에 인덱스를 만드십시오. 아래를 참고하십시오.

파일을 테이블로 읽기

`file_fdw` 는 서버 파일시스템의 파일을 읽기 전용 테이블로 만듭니다. contrib 모듈이므로 빌드할 때 `make install-world` 가 필요합니다(AgensGraph 설치 참고):

> **참고**
>
> 아래 두 블록은 `file_fdw` 가 필요하므로, contrib 모듈을 빌드하지 않았다면 이 페이지에서 유일하게 실행할 수 없는 부분입니다.

```
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');
```

그다음은 위와 같이 `LOAD FROM staging_people AS row …` 입니다. 파일은 서버에서 읽으므로 경로는 서버의 경로이고, 서버를 실행하는 사용자가 그것을 읽을 수 있어야 합니다.

파일이 서버가 아니라 자기 장비에 있다면, 먼저 agens 클라이언트의 `\copy` 로 평범한 테이블에 적재한 뒤 그것을 `LOAD FROM` 하십시오.

레이블 테이블에 COPY

가장 빠른 경로이며, 요소가 수백만 개일 때 쓸 방법입니다. 레이블은 테이블이므로 거기에 곧바로 `COPY` 할 수 있습니다. `properties` 컬럼을 직접 쓰고, `id` 컬럼은 기본값이 스스로 채우게 두는 것입니다.

SQL로 레이블에 쓰는 것은 막혀 있으므로, 적재 동안 그 관문을 여십시오:

```
SET enable_graph_dml = on;

COPY moviekb.person (properties) FROM STDIN;
{"name": "Ann", "born": 1990}
```


© 2026 SKAI Worldwide Co., Ltd.

```
{"name": "Bo", "born": 1985}
\.
```

간선에는 양 끝점의 id도 `labid.locid` 형태로 함께 주십시오:

```
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"}
\.
```

그러므로 일은 그 id를 만들어 내는 데 있습니다. 두 가지 방법이 있습니다:

산술로 만드십시오. 번호 매김을 직접 다룬다면 `graphid()` 가 레이블 id와 로컬 id로 id를 조립합니다:

```
SELECT graphid(graph_labid('moviekb.person'::cstring), 100::int8);
```

```
 made_id
---------
 4.100
```

그러므로 행 번호를 매기는 적재기는 정점 id와 그것을 가리키는 간선 끝점을 탐색 없이 함께 내보낼 수 있습니다. 가능한 가장 빠른 적재입니다.

또는 조인으로 풀어내십시오. 정점을 먼저 적재한 다음, 업무 키로 조인해 스테이징 테이블에서 간선 행을 만듭니다:

```
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;
```

간선마다의 쿼리가 아니라 집합 단위 문장 하나이며, 요점이 바로 그것입니다. 조인에 쓰는 키에 먼저 인덱스를 만드십시오.

```
DROP TABLE staging_roles;
```

경고

`COPY` 와 `INSERT` 는 Cypher가 하는 검사를 건너뜁니다. `start` 가 존재하지 않는 정점을 가리키는 간선을 쓰는 것을 아무것도 막지 않으며, 그 결과는 어떤 쿼리도 제대로 순회할 수 없는 그래프입니다. 적재하기 전에 양 끝점을 확인하고, 끝난 뒤에는 `enable_graph_dml` 을 다시 끄십시오.

대량 적재 중의 인덱스

행 수백만 개가 들어오는 동안 인덱스를 유지하는 것은 비쌉니다. 레이블은 인덱스를 꺼 둔 채로 만들 수 있습니다:

```
CREATE VLABEL bulk_person DISABLE INDEX;
```


© 2026 SKAI Worldwide Co., Ltd.

나중에 끌 수도 있습니다:

```
ALTER VLABEL bulk_person DISABLE INDEX;
```

데이터를 적재한 뒤 다시 만듭니다:

```
REINDEX VLABEL bulk_person;
```

> 참고
>
> `ENABLE INDEX`는 없습니다. `ALTER VLABEL person ENABLE INDEX`는 구문 오류이고, 인덱스를 되살리는 것은 `REINDEX`입니다. 그것을 실행하기 전까지 레이블의 인덱스는 존재하지만 쓸 수 없으므로, 쿼리는 순차 스캔으로 물러나고 고유 제약도 적용되지 않습니다.

큰 적재에 도움이 되는 것 두 가지가 더 있습니다:

속성 인덱스는 데이터보다 먼저가 아니라 나중에 만드십시오. 다 찬 레이블에 인덱스를 한 번 만드는 것이 행마다 유지하는 것보다 훨씬 쌉니다. 속성을 컬럼으로 승격하는 것도 마찬가지입니다. `ALTER`가 레이블 전체를 다시 쓰므로, 적재 전이 아니라 적재 뒤에 한 번 승격하십시오.

다시 적재할 수 있는 데이터라면 **unlogged 레이블을 고려하십시오.** WAL을 통째로 건너뜁니다:

```
CREATE UNLOGGED VLABEL staging_person;
```

unlogged 레이블은 크래시를 견디지 못하고 복제되지도 않습니다. 그러므로 이것은 스테이징과 다시 만들 수 있는 그래프를 위한 것이지, 최종 데이터를 위한 것이 아닙니다.

재인덱싱 전에 `maintenance_work_mem`을 올리십시오. 한 번에 명령 하나만 쓰므로 임시로 큰 값을 두는 것은 안전합니다. 설정을 참고하십시오.

적재를 마친 뒤

세 단계이며, 어느 것도 선택 사항이 아닙니다.

인덱스를 꺼 두었다면 위와 같이 **다시 만드십시오.**

승격할 생각이라면 **걸러 내기와 정렬에 쓰는 속성을 승격하십시오.** `ALTER VLABEL person ADD COLUMN age int GENERATED`는 jsonb `properties` 옆에 속성 `age`를 위한 실제 `int` 컬럼을 만들고 쓰기 시점에 채우며, 그것에 대한 인덱스는 평범한 컬럼 인덱스입니다. 쿼리가 계속 이름을 대는 것은 속성 키입니다. `p.age`는 여전히 `age`를 읽고, 컬럼 이름은 Cypher에 나타나지 않습니다. 컬럼 이름을 키와 다르게 하려면 `GENERATED (sourceKey)`를 쓰십시오. 속성을 컬럼으로 승격하기를 참고하십시오.

`ANALYZE` 하십시오. 갓 적재한 레이블에는 플래너가 쓸 통계가 없고, 그래프 순회 계획은 통계에 매우 민감합니다:

```
ANALYZE moviekb.person;
ANALYZE moviekb.acted_in;
```

그래프 메타데이터를 수집하십시오. `ag_graphmeta`는 그래프에 어떤 (시작 레이블, 간선 레이블, 끝 레이블) 조합이 나타나고 각각이 간선 몇 개로 이루어지는지 기록합니다. 그것이 패턴이 닿을 수 없는 레이블의 스캔을 플래너가 쳐낼 수 있게 합니다:

```
SELECT regather_graphmeta();
```

```
SELECT graphname, graphmeta_valid FROM pg_catalog.ag_graph;
```

그런 다음 적재한 것의 모양을 확인하십시오. 잘못된 적재를 잡아내는 가장 빠른 방법이기도 합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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
```

기대하지 않은 간선 모양이나 있어야 할 값의 몇 배인 개수가 여기서 곧바로 드러납니다.

적재가 끝나면 스테이징 테이블과 임시 레이블을 정리하십시오:

```
DROP TABLE staging_people, staging_knows;
```

```
DROP VLABEL bulk_person;
DROP VLABEL IF EXISTS staging_person;
```

내보내기

`pg_dump`는 그래프를 담으며, 스키마를 지정하면 그래프 하나를 내보냅니다. **백업과 복구**를 참고하십시오. 구조가 아니라 데이터를 원한다면 레이블 테이블에 `COPY … TO`가 동작하고, SQL로 감싼 Cypher 쿼리는 원하는 모양 그대로를 내보냅니다:

```
\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
```

벡터 검색

개요

AgensGraph는 PostgreSQL 위에 만들어졌으므로, **pgvector** 확장이 그래프에 벡터 유사도 검색을 곧바로 더합니다. 그러면 임베딩이 그래프 데이터 옆에 놓이고, 데이터베이스 하나가 의미(벡터) 검색과 Cypher 그래프 순회를 결합할 수 있습니다. 검색 증강 생성(RAG)과 GraphRAG 응용의 토대가 그것입니다.

벡터 검색은 단어가 아니라 **의미**로 일치시킵니다. 두 글이 어휘를 하나도 공유하지 않아도 벡터 공간에서는 가까울 수 있습니다. 키워드로 대조하는 **전문 검색**에 견준 강점이 그것입니다.

둘을 결합하는 것은 **하이브리드 검색**이 다룹니다. 세 페이지의 예제가 모두 같은 영화 집합을 쓰므로 동작을 직접 견주어 볼 수 있습니다.

임베딩은 정점이나 간선의 수치 배열 **속성**으로 저장됩니다. 속성은 `jsonb`이므로 배열은 JSON 배열로 보관됩니다. 벡터 연산이 필요할 때마다 `::vector(n)` 형 변환이 그것을 pgvector의 `vector` 타입으로 바꿉니다.

그 형 변환은 읽을 때마다 일어납니다. 행이 몇천 개일 때는 괜찮지만 규모가 커지면 낭비입니다. 2.18부터는 대신 **그 속성을 실제 `vector` 컬럼으로 승격**할 수 있습니다.

그러면 형 변환은 쓰기 시점에 한 번 일어나고, HNSW 인덱스를 그 컬럼 자체에 만들 수 있습니다. 승격된 컬럼에 남아 있는 `::vector(n)`은 접혀 사라지므로, 이 페이지의 쿼리는 그대로 동작하고 인덱스도 계속 씁니다.

승격하면 두 가지가 달라집니다.

값의 **타입**입니다. 승격된 `d.emb`는 `vector`로 돌아오고 승격하지 않은 것은 `jsonb`로 돌아오므로, JSON 형태를 파싱하는 클라이언트는 고쳐야 합니다.

© 2026 SKAI Worldwide Co., Ltd.

인덱스의 모양입니다. 승격된 컬럼 위에서는 형 변환 표현식에 대한 인덱스가 아니라 평범한 컬럼 인덱스가 되며, 그래서 덤프하고 복원할 인덱스로는 이 형태가 낫습니다.

pgvector 설치

pgvector는 별도의 확장이며, 쓰고 있는 AgensGraph 설치본에 맞춰 빌드해 설치해야 합니다. 쓸 수 있게 된 뒤에는 데이터베이스마다 켜십시오:

```
CREATE EXTENSION vector;
```

예제 데이터

예제는 **예제 데이터셋**을 씁니다. 영화 18편이 각각 `plot`과 그 줄거리의 4차원 `embedding`을 가지고 있습니다. 더 가기 전에 그것을 적재하십시오.

실제 임베딩은 `text-embedding-3-small`이나 로컬 `sentence-transformers` 모델 같은 모델에서 나오며 차원이 수백 개입니다. 여기서는 지면에서 기하 구조가 읽히도록 넷으로 줄였습니다.

유사도 검색

pgvector는 거리 연산자와 그에 대응하는 함수를 제공합니다. 두 피연산자를 `vector(n)`으로 변환한 뒤 거리로 정렬하십시오.

연산자	함수	거리	HNSW 연산자 클래스
`<=>`	`cosine_distance`	코사인 거리	`vector_cosine_ops`
`<->`	`l2_distance`	유클리드(L2)	`vector_l2_ops`
`<#>`	`inner_product`	(음의) 내적	`vector_ip_ops`
`<+>`	`l1_distance`	맨해튼(L1)	`vector_l1_ops`

줄거리가 쿼리 벡터에 가장 가까운(코사인 거리) 영화를 찾습니다. 여기서 쿼리는 The Matrix의 줄거리 임베딩이므로, 검색은 의미가 가장 비슷한 영화를 돌려줍니다:

```
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)
```


© 2026 SKAI Worldwide Co., Ltd.

처음 세 결과는 뻔한 속편이지만, 흥미로운 것은 네 번째입니다. Total Recall의 줄거리는 "a construction worker discovers that his memories have been implanted" 입니다. The Matrix의 "a hacker … a simulated world controlled by machines" 와 **키워드를 하나도** 공유하지 않습니다.

그런데도 벡터 검색은 그것을 네 번째로 올립니다. 두 영화 모두 자기 현실이 꾸며진 것임을 알아차리는 인물의 이야기이기 때문입니다. 키워드 검색은 둘을 결코 잇지 못합니다.

의미 검색은 바로 그것을 잡아내고 어휘 검색은 그러지 못합니다. 같은 데이터에 대한 **전문 검색** 결과와 견주어 보고, 둘을 결합하는 것은 **하이브리드 검색**을 참고하십시오.

> `ORDER BY` 는 거리 표현식이 `RETURN` 목록에 없어도 그것을 참조할 수 있습니다(AgensGraph 2.17 이상). `title` 만 반환하면 출력이 깔끔해지고, 더 중요하게는 HNSW 인덱스가 그 정렬을 맡을 수 있습니다. 거리를 그대로 반환하면 `jsonb` 로 변환되어 인덱스를 쓰지 못하게 됩니다.

함수 형태는 「The Terminator와 가장 비슷한 영화」처럼 저장된 다른 벡터와 견줄 때 편리합니다. 쿼리 벡터를 손으로 쓰지 않아도 됩니다:

```
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)
```

HNSW로 인덱싱하기

인덱스가 없으면 유사도 쿼리마다 모든 행을 스캔하며 각각의 거리를 계산합니다. 데이터가 크다면 플래너가 인덱스 스캔을 쓸 수 있도록 HNSW 인덱스를 만드십시오. 인덱싱하는 값이 형 변환 표현식이므로 **속성 인덱스**를 쓰십시오:

```
CREATE PROPERTY INDEX ON movie
    USING hnsw ((embedding::vector(4)) vector_cosine_ops);
```

연산자 클래스는 쿼리에 쓰는 거리에 맞추십시오(`<=>` 에는 `vector_cosine_ops` , `<->` 에는 `vector_l2_ops` 하는 식입니다). 그러면 앞의 유사도 쿼리가 인덱스를 씁니다:

```
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))
```


© 2026 SKAI Worldwide Co., Ltd.

행이 18개뿐인 이 예제에서는 순차 스캔이 더 싸므로 플래너가 여전히 그것을 고를 수 있습니다. 인덱스 계획을 보려면 `SET enable_seqscan = off`를 더하십시오. 실제 말뭉치에서는 플래너가 알아서 인덱스를 고릅니다.

`::vector(n)` 형 변환에 대한 HNSW 속성 인덱스는 AgensGraph 2.17 이상이 필요합니다.

승격된 컬럼 인덱싱하기

위의 인덱스는 형 변환 표현식 위에 만들어집니다. 속성을 승격하면 형 변환이 사라지므로, 인덱스는 컬럼에 대한 평범한 인덱스가 됩니다.

승격은 jsonb `properties` 옆에 그 속성을 위한 실제 타입 컬럼을 만들고 쓰기 시점에 채웁니다. 선언하는 것은 컬럼 이름이고, 그것이 비추는 속성은 같은 단어로 지칭됩니다:

```
DROP PROPERTY INDEX movie_embedding_idx;
ALTER VLABEL movie ADD COLUMN embedding vector(4) GENERATED;
```

`GENERATED`만 쓰면 컬럼 이름과 같은 이름의 속성, 여기서는 `embedding`을 비춥니다. 컬럼을 다른 이름으로 부르려면 `GENERATED (sourceKey)`라고 쓰십시오 — `ALTER VLABEL movie ADD COLUMN emb vector(4) GENERATED (embedding)`은 같은 속성을 `emb`이라는 컬럼에 저장합니다. 어느 쪽이든 **쿼리는 컬럼이 아니라 속성의 이름을 계속 댑니다.** 해석은 속성 키로 이루어지고 컬럼 이름은 Cypher에 나타나지 않습니다. 자세한 것은 속성을 컬럼으로 승격하기에 있습니다.

그러면 인덱스는 형 변환도 괄호도 없는, 컬럼에 대한 평범한 인덱스가 됩니다:

```
CREATE PROPERTY INDEX ON movie USING hnsw (embedding vector_cosine_ops);
```

이 페이지의 모든 쿼리는 고치지 않아도 그대로 동작합니다. 승격된 컬럼에 남은 `::vector(4)`는 접혀 사라지므로 `m.embedding`과 `m.embedding::vector(4)`는 같은 계획을 만듭니다:

```
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;
```

```
Limit
  ->  Index Scan using movie_embedding_idx on movie m
        Order By: (embedding <=> '[-0.07594558,0.04081754,0.2959212,-0.11921061]'::vector(4))
```

중요

위에서처럼, 표현식 인덱스가 읽는 키를 승격하기 전에 그 인덱스를 지우십시오. 표현식에 대한 속성 인덱스는 jsonb `properties`를 상대로 만들어지고 승격된 키에 대한 쿼리는 컬럼을 읽으므로, 둘은 더는 만나지 않습니다. **표현식 속성 인덱스는 `properties`에 걸린다** 절을 참고하십시오.

승격하면 값의 타입이 달라집니다. 이제 `m.embedding`은 `vector`로 돌아오며, 승격하기 전에는 `jsonb`로 돌아왔습니다.

GraphRAG

그래프 데이터베이스 안의 벡터 검색에는 남다른 강점이 있습니다. 의미 일치가 그래프 순회의 **출발점**이 될 수 있다는 것입니다.

전형적인 GraphRAG 검색은 쿼리 하나로 두 가지를 합니다. 벡터 유사도로 질문에 가장 관련 있는 항목을 찾고, 그다음 관계를 따라 뻗어 나가 언어 모델에 줄 연결된 맥락을 모읍니다.

© 2026 SKAI Worldwide Co., Ltd.

예제 데이터셋에는 영화 사이의 `related_to` 간선이 이미 들어 있습니다. 시리즈 속편과 주제로 이어진 작품들입니다. 그래서 아래 순회가 따라갈 것이 있습니다.

쿼리할 때는 줄거리에 쓴 것과 같은 모델로 사용자의 질문을 임베딩합니다. 그런 다음 가장 가까운 영화를 가져와 그 이웃으로 뻗는 Cypher 쿼리 하나를 실행합니다. 여기서 질문 임베딩은 The Matrix의 벡터입니다:

```
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)
```

`WITH … ORDER BY … LIMIT 2`가 벡터 유사도로 질문에 가장 잘 맞는 영화 둘을 고릅니다. 그것이 **시드**입니다. 두 번째 `MATCH`는 각 시드에서 `related_to` 간선을 따라가 이웃 영화를 모읍니다.

합쳐진 집합은 시드와 그 그래프 이웃입니다. 벡터 검색만으로 얻는 것보다 풍부한 맥락을 이루며, 언어 모델에 넘겨 최종 답을 만들게 할 수 있습니다.

이 방식은 일반화됩니다. 시드를 여러 홉으로 뻗거나, 간선 종류나 속성으로 거르거나, 다른 레이블(사람·스튜디오·출처)에 조인해 RAG 파이프라인에 필요한 맥락을 정확히 조립할 수 있습니다.

프레임워크가 그 일을 대신 해 주기를 바란다면, **AI·LLM 연동**이 그런 라이브러리를 나열합니다.

전문 검색

개요

AgensGraph는 PostgreSQL 위에 만들어졌고 그 전문 검색 엔진을 그대로 물려받으므로, 그래프 속성에 저장한 텍스트를 상대로 언어적 키워드 검색을 곧바로 할 수 있습니다. 전문 검색은 문서와 쿼리를 어휘소로 정규화합니다(어간 추출·불용어 제거·대소문자 변환). 그래서 사용자가 입력한 단어를 대조하는 **어휘** 검색에 잘 맞습니다.

의미로 대조하는 **벡터 검색**과 자연스럽게 짝을 이룹니다. 둘을 결합하는 것은 **하이브리드 검색**이 다룹니다. 세 페이지가 같은 영화를 쓰므로 결과를 직접 견주어 볼 수 있습니다.

그래프 속성은 `jsonb`로 저장되므로, 텍스트 속성은 `tsvector`로 바꾸기 전에 (`::text`로) `text`로 형 변환합니다.

예제 데이터

예제는 **예제 데이터셋**을 씁니다. 영화 18편이 각각 `plot`과 그 줄거리의 4차원 `embedding`을 가지고 있습니다. 더 가기 전에 그것을 적재하십시오.

실제 임베딩은 `text-embedding-3-small`이나 로컬 `sentence-transformers` 모델 같은 모델에서 나오며 차원이 수백 개입니다. 여기서는 지면에서 기하 구조가 읽히도록 넷으로 줄였습니다.

텍스트 속성 검색하기

`to_tsvector`로 속성을 `tsvector`로 바꾸고, `*_to_tsquery` 함수 중 하나로 사용자의 입력을 파싱한 뒤, `@@` 연산자로 대조하십시오. `websearch_to_tsquery`는 너그러운 웹 스타일 구문을 받습니다. 따옴표로 감싼 구절, `OR`, 그리고 단어를 빼는 `-`입니다. 그래서 사용자 입력에는 이것이 좋은 기본값입니다.

© 2026 SKAI Worldwide Co., Ltd.

```
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)
```

쿼리는 원문 텍스트가 아니라 어휘소와 대조됩니다. `machines`는 어간 `machin`으로 줄어드므로 단수형 `machine`도 정확히 같은 행과 일치합니다.

반환되지 않은 것도 보십시오. Total Recall과 The Terminator는 주제로 보면 기계가 지배하는 미래를 다루지만 줄거리에 그 단어를 쓰지 않았으므로 키워드 검색이 찾지 못합니다. 그 재현율의 빈틈이 바로 **벡터 검색**이 돕는 지점입니다.

기대하는 입력에 따라 다른 쿼리 파서도 쓸 수 있습니다:

함수	입력 형태
`websearch_to_tsquery`	웹 스타일: `machines "virtual reality" -documentary`
`plainto_tsquery`	평문이며 모든 단어를 `AND`로 묶습니다
`to_tsquery`	연산자를 직접 씁니다: `machines & reality`

결과 순위 매기기

`ts_rank`는 각 문서가 쿼리에 얼마나 잘 맞는지 점수를 매기므로 결과를 관련도로 정렬할 수 있습니다. 쿼리 단어를 더 많이 맞힌 줄거리가 더 높은 순위를 받습니다:

```
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 Matrix가 속편들보다 위에 오는 것은 그 줄거리가 `machines`와 `reality`를 **둘 다** 담고 있는 반면 속편들은 `machines`만 맞히기 때문입니다.

© 2026 SKAI Worldwide Co., Ltd.

정확한 단어 대조

쿼리가 정확히 맞아야 하는 특정 단어 — 이름, 코드, 드문 키워드 — 일 때는 전문 검색이 알맞은 도구입니다. dinosaurs에 관한 영화를 물으면 정확히 하나를 돌려줍니다:

```
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)
```

벡터 검색에는 `dinosaurs`라는 글자 그대로의 단어라는 개념이 없습니다. Jurassic Park와 가장 비슷한 영화를 물으면 가장 가까운 이웃으로 The Avengers를 돌려줍니다. 둘 다 대규모 볼거리 블록버스터이니 의미로는 그럴듯한 짐작이지만, "dinosaurs"를 입력한 사람이 원하는 것은 아닙니다:

```
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)
```

그러므로 두 방법은 서로 반대쪽에 사각지대를 가집니다. 전문 검색은 정확하지만 동의어와 의도를 놓치고, 벡터 검색은 의미를 잡아내지만 정확한 단어에서 흔들립니다. **하이브리드 검색**은 둘 다 실행하고 결과를 합쳐 각각의 강점을 취합니다.

GIN으로 인덱싱하기

검색할 때마다 모든 행에 대해 `to_tsvector`를 다시 계산하는 것은 규모를 감당하지 못합니다. 인덱스로 일치를 찾도록 같은 `to_tsvector` 표현식에 GIN 인덱스를 만드십시오. 인덱싱하는 값이 속성에 대한 표현식이므로 **속성 인덱스**를 쓰고 표현식을 괄호로 감싸십시오:

```
CREATE PROPERTY INDEX ON movie
    USING gin ((to_tsvector('english', plot::text)));
```

그러면 검색이 모든 행을 스캔하는 대신 인덱스를 씁니다:

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

```
         Index Cond: (to_tsvector('english'::regconfig, (properties.'plot'::text)::text) @@ ''
'machin'''::tsquery)
```

인덱스가 쓰이려면 쿼리 시점에 쓰는 텍스트 검색 구성(위의 `'english'`)이 인덱스 표현식의 것과 같아야 합니다.

> 행이 적은 이 예제에서는 순차 스캔이 더 싸므로 플래너가 여전히 그것을 고를 수 있습니다. 인덱스 계획을 보려면 `SET enable_seqscan = off`를 더하십시오. 실제 말뭉치에서는 플래너가 알아서 인덱스를 고릅니다.
>
> `to_tsvector(...)` 표현식에 대한 GIN 속성 인덱스는 AgensGraph 2.17 이상이 필요합니다.

승격된 컬럼 인덱싱하기

2.18부터는 속성을 **실제 타입 컬럼으로 승격**할 수 있으며, 그러면 행마다의 `jsonb` 읽기와 `::text` 형 변환이 사라집니다. `ALTER VLABEL movie ADD COLUMN plot text GENERATED`는 컬럼 이름이 가리키는 속성, 여기서는 `plot`을 비춥니다. 이름이 다른 키를 비추려면 `GENERATED (sourceKey)`를 씁니다. 쿼리는 컬럼이 아니라 속성의 이름을 계속 댑니다 — 해석은 속성 키로 이루어집니다.

이 페이지의 인덱스는 속성에 대한 **표현식**이므로 승격할 때 다시 만들어야 합니다. 표현식에 대한 속성 인덱스는 jsonb `properties`를 상대로 만들어지고 승격된 키에 대한 쿼리는 컬럼을 읽으므로, 둘은 더는 만나지 않게 됩니다.

이 인덱스는 컬럼을 대상으로 SQL `CREATE INDEX`로 만드십시오. `\dGi`는 여전히 그것을 나열하고, 쿼리도 그것을 씁니다:

```
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)
```

이제 `m.plot`에는 `::text`가 필요 없습니다. 승격된 컬럼이 이미 `text`이기 때문입니다. **표현식 속성 인덱스는 `properties`에 걸린다** 절을 참고하십시오.

GraphRAG

전문 검색도 그래프 순회의 시드가 될 수 있습니다. 질문에 텍스트가 가장 잘 맞는 영화를 찾은 다음, 관계를 따라 뻗어 나가 언어 모델에 줄 연결된 맥락을 모읍니다.

벡터 검색의 GraphRAG 예제에 대응하는 어휘 쪽 예제입니다. 임베딩보다 키워드 검색이 잘 다루는 특정 단어(이름·코드·정확한 구절)가 질문에 들어 있을 때 빛을 발합니다.

예제 데이터셋에는 영화 사이의 `related_to` 간선이 이미 들어 있습니다. 시리즈 속편과 주제로 이어진 작품들입니다. 그래서 아래 순회가 따라갈 것이 있습니다.

Cypher 쿼리 하나가 질문(여기서는 단어 `resistance`)에 대한 전문 검색 관련도로 영화의 순위를 매기고, 가장 잘 맞는 것을 남긴 뒤 그 이웃으로 뻗어 나갑니다:

```
MATCH (seed:movie)
WHERE to_tsvector('english', seed.plot::text) @@ websearch_to_tsquery('english', 'resistance')
```


© 2026 SKAI Worldwide Co., Ltd.

```
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;
```

```
             seed             |               related_context
------------------------------+-----------------------------------------------
 "Terminator 2: Judgment Day" | ["The Terminator"]
 "The Terminator"             | ["Terminator 2: Judgment Day", "The Matrix"]
(2 rows)
```

`WHERE` 조건과 `WITH … ORDER BY rank LIMIT`가 쿼리에 텍스트가 가장 잘 맞는 영화를 고릅니다. 그것이 **시드**이며, 여기서는 Terminator 두 편입니다. 두 번째 `MATCH`는 `related_to` 간선을 따라가 관련 작품을 모읍니다.

시드와 그 이웃이 모델에 넘길 맥락을 이룹니다. 순회가 The Matrix까지 닿는 것에 유의하십시오. 키워드 `resistance`는 그 영화와 일치한 적이 없지만, 기계와의 전쟁이라는 공통 주제로 Terminator 영화들과 이어져 있습니다. 뻗어 나가기 전에 이 어휘 검색을 의미(벡터) 검색과 결합하려면 **하이브리드 검색**을 참고하십시오.

같은 검색을 손으로 짜는 대신 프레임워크로 하려면 **AI·LLM 연동**을 참고하십시오.

하이브리드 검색

개요

벡터 검색은 **의미**로 대조합니다. **전문 검색**은 **단어**로 대조합니다. 각각은 상대가 약한 곳에서 강합니다.

벡터 검색은 바꿔 쓴 표현과 관련 개념을 찾아내지만, 제품 코드나 이름, 드문 키워드 같은 정확한 단어를 놓칠 수 있습니다. 전문 검색은 정확한 단어를 정확히 잡지만 동의어와 의도를 놓칩니다.

하이브리드 검색은 둘 다 실행하고 결과를 합치므로, 검색이 의미를 알면서도 어휘에서 정확해집니다. AgensGraph는 **pgvector**와 PostgreSQL 전문 검색을 한 데이터베이스에서 제공하므로, 하이브리드 쿼리는 같은 그래프에 대한 문장 하나입니다.

> 이 페이지는 **검색 결과의 융합**에 관한 것입니다. 한 문장에서 SQL과 Cypher를 섞는 것을 다루는 **SQL과 Cypher 섞기** 페이지와는 관계가 없습니다.

상호 순위 융합(RRF)

두 검색은 척도가 다른 점수(코사인 거리 대 `ts_rank`)를 내므로 더할 수 없습니다. **상호 순위 융합**(RRF)은 점수가 아니라 **순위**를 합쳐 그 문제를 피합니다.

각 결과는 자기가 나타난 검색마다 `1 / (k + rank)`를 보태고, 그 값들을 모두 더합니다:

```
score(d) = 1/(k + rank_vector(d)) + 1/(k + rank_fulltext(d))
```

`k`는 최상위 순위가 얼마나 지배하는지를 누르는 평활 상수입니다. 흔히 쓰는 기본값은 60입니다. 두 검색 **어느 한쪽**에서 순위가 높은 문서는 그럭저럭 점수를 얻고, **둘 다**에서 높은 문서가 가장 높은 점수를 얻습니다.

예제 데이터

예제는 **예제 데이터셋**을 씁니다. 영화 18편이 각각 `plot`과 그 줄거리의 4차원 `embedding`을 가지고 있습니다. 더 가기 전에 그것을 적재하십시오.

실제 임베딩은 `text-embedding-3-small`이나 로컬 `sentence-transformers` 모델 같은 모델에서 나오며 차원이 수백 개입니다. 여기서는 지면에서 기하 구조가 읽히도록 넷으로 줄였습니다.

© 2026 SKAI Worldwide Co., Ltd.

하이브리드 쿼리

상호 순위 융합에는 각 검색의 **순위**가 필요하고, 그것을 만들어 내는 것이 SQL 윈도 함수입니다. 그래서 융합은 그래프 데이터를 Cypher `MATCH` 서브쿼리에서 받는 SQL 쿼리입니다.

한 서브쿼리는 벡터 거리로, 다른 하나는 전문 검색 관련도로 영화의 순위를 매깁니다. `row_number()`가 정렬된 결과를 순위로 바꾸고, 바깥 쿼리가 역수 순위를 더합니다. 각 서브쿼리는 정점 id만 반환하고, 반환하지 않는 순위 표현식으로 정렬합니다. 그래서 벡터와 전문 검색 속성 인덱스가 정렬과 걸러 내기를 곧바로 맡을 수 있습니다.

아래 쿼리는 The Matrix 계열이면서 `machines`를 글자 그대로 언급하는 영화를 찾습니다. 벡터 쿼리는 그 영화의 줄거리 임베딩입니다:

```
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)
```

여기서 융합은 두 가지를 동시에 합니다. Total Recall(벡터 순위 4)은 `machines`라는 단어를 쓰지 않으므로 전문 검색 쪽은 그것을 아예 놓칩니다. 그런데도 결과에 살아남습니다. 벡터 검색이 그것을 The Matrix와 같은 **가짜 현실** 이야기로 알아보았기 때문입니다.

The Matrix Revolutions는 반대 경우입니다. 벡터로는 7번째로 가까울 뿐이라 벡터만 쓰는 상위 다섯 안에 들지 못하지만, 줄거리가 machines를 글자 그대로 말하므로 전문 검색 쪽이 3위로 올립니다. 융합이 그것을 다시 끌어올립니다.

양쪽 **모두**를 만족하는 영화(The Matrix·Reloaded·Resurrections)가 최상위를 차지합니다. 벡터 검색은 재현율을, 전문 검색은 정확도를 보태고, RRF가 둘을 섞습니다.

© 2026 SKAI Worldwide Co., Ltd.

양쪽 모두 인덱싱하기

쿼리의 각 갈래는 평범한 유사도 검색과 전문 검색이므로, **벡터 검색**과 **전문 검색**에서 설명한 것과 같은 **속성 인덱스**가 그것을 빠르게 합니다. 임베딩에 대한 HNSW 인덱스와 평문 줄거리에 대한 GIN 인덱스입니다:

```
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)));
```

각 Cypher 서브쿼리는 id만 반환하고 가공하지 않은 순위 표현식으로 정렬하므로, 플래너가 각 갈래를 인덱스로 처리할 수 있습니다. HNSW 인덱스가 벡터 후보를 정렬하고 GIN 인덱스가 전문 검색 일치를 걸러 내므로, 어느 검색도 레이블 전체를 스캔하지 않습니다.

융합 쿼리에 대한 `EXPLAIN`에서 관련 있는 부분은 다음과 같습니다:

```
 ...
 ->  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
```

> 아주 작은 예제 데이터에서는 행이 몇 개뿐이라 인덱스보다 순차 스캔이 싸므로 플래너가 여전히 그것을 고를 수 있습니다. 인덱스 계획을 보려면 `SET enable_seqscan = off`를 더하십시오. 실제 말뭉치에서는 플래너가 알아서 인덱스를 고릅니다.

위의 두 인덱스는 모두 속성에 대한 표현식입니다. 두 속성 중 어느 하나라도 **타입 컬럼으로 승격**한다면 그쪽 갈래의 인덱스를 다시 만드십시오. 표현식 속성 인덱스는 jsonb `properties` 위에 만들어지는데, 승격된 키에 대한 쿼리는 컬럼을 읽기 때문입니다. 그러면 임베딩 갈래는 컬럼에 대한 평범한 인덱스가 되고, 전문 검색 갈래는 그 컬럼에 대한 SQL 인덱스가 됩니다:

```
ALTER VLABEL movie ADD COLUMN embedding vector(4) GENERATED;
CREATE PROPERTY INDEX ON movie USING hnsw (embedding vector_cosine_ops);
```

융합 쿼리 자체는 바뀌지 않습니다. **벡터 검색**과 **전문 검색**이 각자의 쪽을 보여 줍니다.

조정

- `k` **상수** — 값이 크면 최상위 순위의 기여가 평평해지고, 작으면 앞의 몇 결과가 지배합니다. 60이 무난한 출발점입니다.
- **가중치** — 한쪽 방법에 무게를 실으려면 그 항에 곱하십시오. 의미 쪽으로 기울이려면 예를 들어 `1.5 * COALESCE(1.0/(60 + vec.rnk), 0)` 처럼 씁니다.
- **후보 깊이** — 각 CTE 안의 `LIMIT`이 융합 전에 각 검색이 얼마나 깊이 가는지를 정합니다. 지연을 조금 치르고 재현율을 높이려면 이 값을 키우십시오.

GraphRAG

하이브리드 검색은 그래프 순회의 가장 강한 출발점입니다. 의미 유사도와 키워드 관련도 둘 다로 영화의 순위를 매기고(RRF), 상위 결과에서 관계를 따라 뻗어 나가 언어 모델에 줄 맥락을 조립합니다.

© 2026 SKAI Worldwide Co., Ltd.

상호 순위 융합에는 SQL 윈도 함수가 필요하지만, 그래프에 손대는 부분은 모두 `FROM` 절의 Cypher `MATCH` 서브쿼리로 쓰여 있습니다. 두 검색과 뻗어 나가기가 다 그렇습니다. 그것이 **SQL과 Cypher 섞기**의 장치입니다. 쿼리는 그래프에 머무르며 위에서 만든 속성 인덱스를 그대로 씁니다.

예제 데이터셋에는 영화 사이의 `related_to` 간선이 이미 들어 있습니다. 시리즈 속편과 주제로 이어진 작품들입니다. 그래서 아래 순회가 따라갈 것이 있습니다.

여기서 벡터 갈래는 The Matrix 같은 영화를 가져오고, 전문 검색 갈래는 키워드 `memories`를 대조합니다. SQL `row_number()`가 정렬된 결과를 순위로 바꿉니다. `seeds` CTE가 RRF로 둘을 융합해 상위 둘을 남기고, 마지막 Cypher 순회가 각 시드의 이웃을 모읍니다:

```
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)
```

융합은 어떤 영화가 순회의 시드가 되는지를 바꿉니다. Total Recall은 The Matrix와 벡터로는 4번째로 가까울 뿐이므로, 벡터만으로 시드를 뽑았다면 The Matrix와 The Matrix Reloaded를 뽑았을 것입니다. 그러나 키워드 `memories`가 Total Recall과 정확히 일치하고, RRF가 그것을 상위 둘로 끌어올립니다.

Total Recall이 시드에 들어오면 순회는 그 `related_to` 간선을 따라 Inception에 닿습니다. 어느 한쪽만으로는 떠오르지 않았을 맥락입니다.

결과는 하이브리드로 순위를 매긴 시드와 그 그래프 이웃입니다. 의미 유사도와 정확한 키워드, 그리고 명시적 관계를 한꺼번에 끌어오므로 세 방법 중 가장 풍부한 맥락입니다.

© 2026 SKAI Worldwide Co., Ltd.

AI·LLM 연동의 프레임워크들이 이와 같은 형태를 대신 만들어 주며, 그중 몇몇은 그것을 리트리버 하나로 노출합니다.

AI·LLM 연동

개요

지식 그래프와 그 임베딩을 한 데이터베이스에 함께 둘 수 있으므로, AgensGraph는 검색 증강 생성의 저장소로 흔히 쓰입니다. 그래프는 테이블이고 속성은 `jsonb`이므로, 정점에 저장한 임베딩을 `pgvector`로 인덱싱할 수 있습니다.

그러므로 쿼리 하나가 의미로 가장 가까운 문서를 찾고 거기서 그래프를 따라 뻗어 나갈 수 있습니다. GraphRAG에 필요한 것이 정확히 그것입니다.

연동은 별도 매뉴얼 **AgensGraph AI**에 있습니다. 연동마다 설치와 사용 페이지가 거기에 있으며, 이 페이지는 각 연동이 어디에 쓰이는지 설명합니다.

이들은 모두 **AgensGraph Python 드라이버** 위에서 동작합니다. 그 드라이버는 **psycopg 3** 위에 만들어졌고 그것을 통해 Cypher를 실행합니다. 그래서 어느 프레임워크를 고르든 정점·간선·경로가 텍스트가 아니라 객체로 프레임워크에 전달됩니다. 0.3.1 버전의 연동 패키지에는 드라이버 2.0 이상이 필요합니다. 드라이버 자체는 **드라이버와 클라이언트 라이브러리**를 참고하십시오.

임베딩은 `pgvector`가 다루는데, `pgvector`는 함께 배포되지 않으므로 데이터베이스에 한 번 설치해야 합니다. **벡터 검색**을 참고하십시오.

모든 연동의 소스는 **agensgraph-ai 저장소**에 있으며, 실행할 수 있는 데모도 함께 들어 있습니다.

LangChain

그래프 저장소, pgvector 기반 벡터 저장소, 채팅 메시지 이력, LangGraph 체크포인터, 그리고 LLM 그래프 트랜스포머입니다.

LangChain 연동

LlamaIndex

`PropertyGraphIndex`와 `VectorStoreIndex` 저장소이며, 하이브리드 검색·메타데이터 걸러 내기· AgensGraph에 맞춘 Text2Cypher를 갖췄습니다.

LlamaIndex 연동

LightRAG

LightRAG의 저장소 역할 네 가지를 한 데이터베이스에서 맡으며, 워크스페이스 단위 멀티테넌시를 지원합니다.

LightRAG 연동

cognee

cognee AI 메모리 프레임워크를 위한 그래프·벡터 어댑터입니다.

cognee 연동

MCP 서버

Cypher·메모리·데이터 모델링, 세 가지 Model Context Protocol 서버이며, Claude Desktop 같은 클라이언트에 AgensGraph를 노출합니다.

MCP 서버

프레임워크 없이

프레임워크를 거치지 않고 검색을 직접 만든다면 검색 가이드 세 편에서 시작하십시오. **벡터**·**전문**· **하이브리드** 검색은 각각 그 검색과 순회를 한 문장에 결합하는 GraphRAG 절로 끝납니다.

© 2026 SKAI Worldwide Co., Ltd.

드라이버와 클라이언트 라이브러리

AgensGraph는 PostgreSQL 통신 프로토콜을 쓰므로, **어떤 PostgreSQL 드라이버로도 접속해 Cypher를 실행할 수 있습니다.** 프로토콜이 보기에 Cypher 문장은 그저 문장 하나입니다. AgensGraph를 쓰는 데 그래프 전용 클라이언트가 필요하지 않습니다.

드라이버가 도움 없이 하지 못하는 것은 그래프 타입을 해석하는 일입니다. `m.title` 을 반환하는 쿼리는 모든 드라이버가 아는 `jsonb` 를 돌려주지만, `m` 을 반환하는 쿼리는 `vertex` 를, 경로 패턴의 `p` 는 `graphpath` 를 돌려줍니다. 보통의 드라이버는 그것들을 텍스트로 봅니다. AgensGraph를 위해 쓰인 드라이버는 타입 처리를 더하므로, 정점이 자기 레이블과 id와 속성을 지닌 객체로 도착합니다.

그래서 방법은 두 가지입니다:

이미 쓰고 있는 PostgreSQL 드라이버를 쓰십시오. 요소 전체가 아니라 속성을 반환하고, `jsonb` 대신 평범한 스칼라를 원하는 곳에서는 Cypher를 SQL로 감싸십시오. **SQL과 Cypher 섞기**를 참고하십시오. 코드도 의존성도 가장 적고, Python이 아닌 언어에서는 이 길이 맞습니다.

정점과 간선과 경로를 네이티브 객체로 받고 싶을 때, 또는 AI 연동을 기반으로 만들 때는 **Python 드라이버를 쓰십시오.**

Python 드라이버

기반으로 삼을 드라이버는 **AgensGraph for Python**입니다. psycopg 3 위에 그래프 타입 처리를 더하며, 모든 **AI 연동**이 이 드라이버 위에서 동작합니다. 설치와 사용법은 그 매뉴얼에 있습니다.

LangChain·LlamaIndex·LightRAG·cognee와 MCP 서버가 모두 이 드라이버를 기반으로 합니다. **AI·LLM 연동**을 참고하십시오.

다른 언어

Java·Node.js·Go에서는 보통의 PostgreSQL 드라이버 — pgjdbc, node-postgres, pgx — 를 쓰고, 위에서 말한 대로 요소 전체가 아니라 속성을 읽으십시오. 지금 기반으로 삼기에 알맞은 조합은 이것입니다.

그 세 언어를 위한 드라이버도 있기는 하며, 새로 시작하는 자리가 아니라 이미 쓰고 있는 프로젝트를 위해 여기에 남겨 둡니다:

- Java / JDBC legacy — pgjdbc의 포크이며 2023년 1월에 `net.bitnine:agensgraph-jdbc:1.4.3` 으로 배포됐습니다.
- Node.js legacy — node-postgres를 감싸며 `pg >= 6.1.2` 를 대상으로 합니다. 2025년 9월에 `@skaiworldwide/ag-driver` 1.1.0으로 배포됐습니다.
- Go legacy — `lib/pq` 위에 만들어졌고 태그 `v0.1.1` 의 Go 모듈로 배포됩니다.

접속과 매개변수

접속에 `graph_path` 를 설정하십시오. 세션별이므로, 풀에서 꺼내 주는 접속은 꺼낼 때마다 설정해야 합니다. 아니면 롤이나 데이터베이스에 한 번 설정하십시오:

```
ALTER ROLE appuser SET graph_path = moviekb;
```

Python 드라이버의 `ConnectionPool` 은 `graph=` 로 받은 그래프를 접속을 열 때 한 번만 설정하고 꺼내 줄 때마다 다시 설정하지는 않으므로, 접속을 빌려 간 쪽이 `graph_path` 를 바꾸고 커밋하면 그 값이 다음에 빌려 가는 쪽으로 넘어갑니다.

Cypher 매개변수에는 타입을 주십시오. 드라이버가 타입을 정해 주지 않은 매개변수는 JSON으로 파싱되므로, 평범한 문자열은 실패합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  invalid input syntax for type json
DETAIL:  Token "The" is invalid.
CONTEXT:  JSON data, line 1: The...
```

드라이버의 API로 매개변수의 타입을 선언하거나, 문장 안에서 형 변환하십시오:

```
MATCH (m:movie {title: $1::text}) RETURN m.year;
```

형 변환이 이식성 있는 형태이고 비용도 들지 않습니다. 값이 어느 쪽으로도 읽힐 수 있는 곳에서 가장 중요합니다. 타입이 없으면 `'123'` 이 숫자 123으로 도착해, 문자열 `"123"` 을 담은 속성과 일치하지 않게 됩니다.

Python 드라이버는 타입을 대신 선언해 줍니다. 2.0부터는 Python `str` 을 `text` 로 보내므로 문자열 매개변수는 따로 형 변환할 필요가 없습니다. 다만 문자열을 다른 타입으로 받아야 하는 자리에서는 여전히 `%s::date` 처럼 형 변환해야 합니다.

© 2026 SKAI Worldwide Co., Ltd.

운영

AgensGraph를 운영 환경에서 돌리는 일입니다. 배포, 설정, 접근 제어, 백업, 그리고 서버 측 도구입니다.

배포

AgensGraph 서버가 운영 트래픽을 받기 전에 정해야 할 것들입니다. 소프트웨어를 장비에 올리는 절차는 **AgensGraph 설치**에 있고, 이 페이지는 그 주변의 선택을 다룹니다.

플랫폼

AgensGraph는 Linux와 Windows에서 빌드·실행되며, 개발용으로는 macOS에서도 됩니다. 배포는 대부분 Linux에서 이루어집니다. 시험한 배포판은 Rocky Linux·RHEL·Debian·Ubuntu의 현재 릴리스입니다. **컨테이너 이미지**는 Debian trixie와 bookworm, Rocky 9와 10, 그리고 Alpine 위에서 빌드됩니다.

AgensGraph의 어떤 점도 PostgreSQL의 플랫폼 지원 범위를 좁히지 않으므로, **PostgreSQL의 지원 플랫폼 안내**가 그대로 적용됩니다.

용량 산정

권장값 표가 아니라 데이터에서 출발하십시오.

가장 크게 작용하는 지렛대는 메모리입니다. 그래프 순회는 인덱스를 따라 흩어진 페이지로 간선을 따라가므로 임의 접근 작업입니다. 캐시 밖의 임의 접근은 느립니다. 작업 집합 — 쿼리가 훑는 레이블과 그 인덱스 — 이 운영체제 캐시에 들어가는 것을 목표로 삼으십시오. `shared_buffers`와 `effective_cache_size`를 그에 맞춰 잡으십시오. 방법은 **설정**에 있습니다.

무엇을 상대하고 있는지 보려면:

```
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;
```

간선 레이블은 대개 그래프에서 가장 큰 객체이고 자체 인덱스를 셋 지니므로, 행뿐 아니라 인덱스까지 계산에 넣으십시오.

스토리지는 플래시여야 합니다. 처리량 때문이 아니라 임의 읽기 지연 때문입니다. 순회가 시간을 쓰는 곳이 거기입니다. 플래시 스토리지라면 플래너 설정 방법도 달라집니다. **설정**의 `random_page_cost`를 참고하십시오.

CPU는 순회에서는 생각보다 덜 중요하고, 레이블 전체를 훑는 분석 쿼리에서는 생각보다 더 중요합니다. 앞은 대체로 직렬이고 뒤는 병렬화되기 때문입니다.

쓸 만한 무엇이든 하려면 RAM은 최소 4 GB입니다. 디스크는 데이터와 인덱스, WAL, 그리고 인덱스를 만들 여유까지 잡으십시오. 기본 설치 자체에 약 2.5 GB가 듭니다.

처음 기동하기 전에

서버는 그 자신의 권한 없는 사용자로 실행하십시오. root도 아니고, 애플리케이션과 함께 쓰는 로그인 계정도 아닙니다:

```
useradd --system --home /var/lib/agensgraph --shell /usr/sbin/nologin agens
```

가능하다면 **데이터 디렉터리를 자기 파일시스템에 두십시오.** 그래야 그것이 가득 차도 장비의 나머지를 함께 끌고 가지 않고, 따로 스냅샷을 뜰 수 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

```
install -o agens -g agens -m 0700 -d /var/lib/agensgraph/18/data
sudo -u agens initdb -D /var/lib/agensgraph/18/data
```

`initdb`는 데이터 디렉터리에 0700이나 0750 모드를 요구합니다. PostgreSQL 18에서는 데이터 페이지 체크섬도 기본으로 켭니다. 그대로 두는 편이 좋고, 나중에 이 클러스터로 업그레이드할 때 플래그 하나를 좌우하므로 알아 둘 만합니다. **2.18로 업그레이드하기**를 참고하십시오.

커널 자원은 손댈 일이 드뭅니다. AgensGraph는 9.3 이후의 PostgreSQL이 그렇듯 주 공유 메모리에 익명 mmap을 쓰고 System V 세그먼트는 아주 작게만 쓰므로, 예전의 `SHMMAX` 조정은 필요 없습니다. 여전히 발목을 잡을 수 있는 것은 메모리가 빠듯한 호스트에서의 `vm.overcommit_memory`와 OOM 킬러, 그리고 접속이 많은 서버에서의 열린 파일 제한입니다. **PostgreSQL의 커널 자원** 절이 레퍼런스입니다.

systemd에서 운영하기

인스턴스 하나를 위한 유닛입니다:

```
[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
```

`postgres`를 직접 기동하는 `Type=notify` 구성이 systemd가 가장 잘 다루는 형태입니다. 서버가 실제로 접속을 받기 시작한 시점을 systemd에 알려 주고, `systemctl stop`이 깨끗한 종료를 하도록 유지합니다. `OOMScoreAdjust`는 커널이 postmaster보다 백엔드를 먼저 죽이도록 만듭니다.

임시 작업에는 `ag_ctl`을, 서비스에는 systemd를 쓰십시오. 손으로 기동하고 systemd로 정지하는 식으로 둘을 섞으면 postmaster가 유령처럼 남게 됩니다.

접속

`listen_addresses`는 `initdb` 직후 `localhost`입니다. 넓힐 때는 의도를 가지고, 아래의 인증 규칙과 함께해야 합니다.

`max_connections`의 기본값은 100입니다. 접속마다 메모리가 들고 그래프 쿼리는 `work_mem` 할당을 여러 개 동시에 쓸 수 있으므로, 이 값을 올리는 것은 공짜가 아닙니다. 짧게 살다 가는 접속을 많이 여는 애플리케이션 앞에는 커넥션 풀러를 두십시오. AgensGraph는 PostgreSQL 프로토콜을 쓰므로 PgBouncer와 Pgpool-II가 그대로 동작합니다.

인증은 `pg_hba.conf`에서 설정하며, 로컬 접속의 기본값인 `trust`는 그대로 둘 것이 아닙니다. **보안**을 참고하십시오.

복제와 고가용성

그래프는 테이블이고 WAL은 그 차이를 알지 못하므로, 스트리밍 복제·논리 복제·동기 커밋이 모두 그래프에서 동작합니다. 따라서 대기 서버는 추가 설정 없이 그래프 데이터를 담고, 읽기 전용 그래프 쿼리는 핫 스탠바이에서 실행됩니다.

© 2026 SKAI Worldwide Co., Ltd.

AgensGraph에만 해당하는 것 둘입니다:

- 그래프 쓰기는 병렬화되지 않으므로, 쓰기가 많은 그래프 워크로드는 주 서버에서 문장 하나에 코어 하나를 넘겨 쓰지 않습니다. 이는 복제의 동작이 아니라 쓰기 처리량을 어떻게 계획할지에 영향을 줍니다.
- 대기 서버를 승격한 뒤, 또는 어떤 물리 복원 뒤에도 `ag_graph.graphmeta_valid`를 확인하고 거짓이면 `regather_graphmeta()`를 실행하십시오. **그래프 메타데이터**를 참고하십시오.

자동 장애 조치에는 보통 Patroni를 씁니다. PostgreSQL에서와 마찬가지로 AgensGraph에서도 동작합니다.

운영 투입 전 점검 목록

- `postgresql.conf`를 **설정**에 비추어 검토했는가. 특히 `shared_buffers`·`work_mem`·`effective_cache_size`·`random_page_cost`.
- `pg_hba.conf`에 `trust` 규칙이 없고, `listen_addresses`가 필요 이상으로 넓지 않은가.
- 백업이 돌고 있고 복원을 시험했는가. **백업과 복구**에 따라 레이블별 정점·간선 개수까지 확인했는가.
- 최근 쓰기를 잃는 것이 용납되지 않는다면 WAL 아카이빙을 켰는가.
- 느린 쿼리가 흔적을 남기도록 `log_min_duration_statement`를 설정하고 `auto_explain`을 올렸는가.
- 접속 수·복제 지연·캐시 적중률·디스크 여유를 감시하고 있는가.
- `autovacuum`을 켜 둔 채로 두었는가. 간선 레이블은 자주 바뀌며, autovacuum을 끄는 것이 그래프 데이터베이스가 데이터의 두 배 크기인 테이블을 갖게 되는 경로입니다.
- `auto_gather_graphmeta`를 의도를 가지고 정했는가. 켜서 플래너의 스캔 쳐내기를 위해 쓰기마다 비용을 조금씩 치르거나, 꺼 두고 정해진 일정에 다시 수집하거나.
- 간선 레이블의 통계 목표치를 확인했는가. 특히 클러스터를 2.17에서 업그레이드했다면. **2.18로 업그레이드하기**를 참고하십시오.

설정

AgensGraph의 설정은 PostgreSQL의 설정에 자체 설정 열한 개를 더한 것입니다. 이 페이지는 설정하는 방법과 그래프 워크로드에 중요한 설정을 다룹니다. AgensGraph가 더한 설정은 **서버 설정**을 참고하십시오. 이 페이지가 나열하지 않는 수백 개의 PostgreSQL 설정은 **PostgreSQL의 서버 설정** 장을 참고하십시오. 그대로 적용됩니다.

설정을 어디에 지정할 수 있는가

범위는 넓은 쪽에서 좁은 쪽 순이며, 적용되는 것 가운데 가장 좁은 범위가 이깁니다.

방법	범위	비고
`postgresql.conf`	클러스터 전체	파일을 고친 뒤 `ag_ctl reload`를 합니다. context가 `postmaster`인 설정은 재기동해야 합니다.
`ALTER SYSTEM SET`	클러스터 전체	`postgresql.auto.conf`에 씁니다. 마찬가지로 reload나 재기동이 필요합니다.
`ALTER DATABASE db SET`	데이터베이스 하나	다음 접속부터 적용됩니다.
`ALTER ROLE r SET`	롤 하나	다음 접속부터 적용됩니다.
`SET`	현재 세션	접속을 끊으면 되돌아갑니다. `SET LOCAL`은 트랜잭션이 끝날 때 되돌아갑니다.

© 2026 SKAI Worldwide Co., Ltd.

가장 먼저 만나게 될 설정은 `graph_path`이며, 본디 세션별입니다. 모든 세션이 같은 그래프를 쓴다면 데이터베이스에 지정하십시오:

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

설정이 지금 무엇이고 그 값이 어디서 왔는지 보려면:

```
SELECT name, setting, source, sourcefile, sourceline
FROM pg_settings WHERE name = 'work_mem';
```

`SHOW ALL`은 전부를 나열합니다. `pg_settings`는 같은 정보를 걸러 볼 수 있는 테이블로 제공하므로 더 쓸모 있습니다. `boot_val` 컬럼은 내장 기본값이고, `reset_val`은 `RESET` 했을 때 얻는 값이며, `context`는 변경에 재기동이 필요한지 알려 줍니다.

> **참고**
>
> 설정의 내장 기본값은 설치 디렉터리의 주석 처리된 `postgresql.conf.sample`이 아니라 `pg_settings.boot_val`에서 읽으십시오. `boot_val`이 서버가 실제로 기동할 때 쓰는 값입니다.

메모리

이 넷이 서버 동작의 대부분을 좌우하며, PostgreSQL의 기본값은 어디서든 기동되도록 일부러 작게 잡혀 있습니다.

`shared_buffers` — 서버 자신의 페이지 캐시입니다. 기본값은 128 MB입니다. 전용 장비라면 RAM의 25%가 무난한 출발점입니다. 40%를 크게 넘기면 대개 역효과가 납니다. 운영체제의 캐시도 그 페이지들을 들고 있기 때문입니다. 재기동이 필요합니다.

`work_mem` — 정렬이나 해시 하나가 디스크로 넘기기 전에 쓸 수 있는 메모리 양입니다. 기본값은 4 MB입니다. 접속별이 아니라 연산별입니다. 그래프 쿼리는 해시를 여러 개 동시에 들고 있을 수 있고, 조인이 많은 쿼리는 이 값의 몇 배를 쓸 수 있습니다. 그러니 조금씩 올리고, 서버가 메모리를 소진하는지 지켜보십시오.

`maintenance_work_mem` — `VACUUM` · `ANALYZE` · `CREATE INDEX` · `REINDEX`를 위한 값입니다. 한 번에 명령 하나만 쓰므로, 대량 적재 뒤 인덱스를 만들 때 임시로 크게 올릴 만합니다.

`effective_cache_size` — 할당이 아니라 힌트입니다. 장비가 캐시에 쓸 수 있는 메모리가 대략 얼마인지 플래너에게 알려 줍니다. 그 추정이 인덱스 스캔을 감당할 만해 보이게 만듭니다. RAM의 50~75% 정도로 잡으십시오. 큰 장비에서 기본값 4 GB로 두는 것이, 인덱스 스캔이 이길 상황에서 그래프 순회가 순차 스캔을 고르는 가장 흔한 이유입니다.

순회에 중요한 플래너 설정

`random_page_cost` — 임의 페이지 읽기를 순차 읽기에 견준 플래너의 추정값이며 기본값은 4입니다. 그 비율은 회전 디스크를 설명합니다. SSD나 NVMe에서 실제 비율은 1에 훨씬 가깝고, 그래프 순회는 거의 전부가 임의 접근입니다. 인덱스를 따라 흩어진 힙 페이지로 간선을 따라가기 때문입니다. 플래시 스토리지에서 이 값을 1.1 정도로 낮추는 것이 순회 계획을 좋게 만들 가능성이 가장 큰 단 하나의 변경입니다.

그 말을 곧이곧대로 믿지도, 극단까지 밀어붙이지도 마십시오. 재어 보십시오. 가장 느린 순회를 4일 때와 1.1일 때 `EXPLAIN (ANALYZE, BUFFERS)`로 돌려 비교하고, 자기 데이터에서 이기는 값을 쓰십시오. 0에 가까운 값은 임의 I/O가 공짜라고 플래너에게 말하는 것인데, 메모리 안에서도 사실이 아니며, 플래너는 기꺼이 스래싱하는 계획을 고를 것입니다.

`max_parallel_workers_per_gather` — 기본값 2입니다. 그래프 쓰기는 병렬로 실행되지 않고 순회는 대체로 직렬이므로, 이득의 대부분은 `MATCH`가 컴파일된 순차 스캔과 비트맵 스캔에서 옵니다. 이 값을 올리면 레이블 전체를 훑는 분석 쿼리에 도움이 됩니다.

`jit` — 기본으로 켜져 있습니다. 오래 도는 분석 쿼리에서는 제값을 하고, 짧은 쿼리에서는 컴파일 시간을 잡아먹습니다. 워크로드가 작은 순회의 연속이라면 꺼 놓고 재어 보십시오.

PostgreSQL 18의 비동기 I/O

PostgreSQL 18은 비동기 I/O를 들여왔고, 그래프 쿼리도 그중 일부를 그냥 얻습니다. 이 이득이 쿼리 수정이 아니라 설정 변경에서 오므로 알아 둘 만합니다.

© 2026 SKAI Worldwide Co., Ltd.

길이가 고정된 `MATCH` 는 레이블 테이블에 대한 평범한 순차·비트맵·인덱스 스캔으로 컴파일되고, PostgreSQL 18의 미리 읽기는 **순차 스캔과 비트맵 힙 스캔**이 씁니다. 따라서 그것들은 저절로 이득을 봅니다. 그렇지 않은 것이 둘 있습니다. 평범한 인덱스 스캔의 힙 가져오기는 PostgreSQL 18에서 미리 읽기를 얻지 못하고, 가변 길이·최단 경로·데이크스트라 엔진은 설계상 정점을 하나씩 따라갑니다.

이를 제어하는 설정이 둘 있습니다:

`effective_io_concurrency` — PostgreSQL 18에서 기본값이 1에서 16으로 올랐습니다. 이 기법 전체의 관문이므로, 예전 릴리스의 `postgresql.conf` 를 그대로 가져와 이 값을 1로 못 박아 두었다면 비동기 I/O를 꺼 놓은 것입니다.

`io_method` — 기본값은 `worker` 이며, 읽기를 백그라운드 워커에서 수행합니다. Linux에서 차가운 스캔에는 `io_uring` 이 더 빠르고 때로는 큰 차이가 나지만, 서버를 liburing과 함께 빌드한 경우에만 쓸 수 있습니다. 설정하기 전에 확인하십시오:

```
SELECT enumvals FROM pg_settings WHERE name = 'io_method';
```

```
 enumvals
---------------
 {sync,worker}
```

이 빌드에는 `io_uring` 이 없습니다. 그것과 함께 빌드한 서버는 값 세 개를 나열합니다. `io_method` 를 바꾸려면 재기동해야 합니다.

이득은 데이터가 캐시에 들어가지 않을 때 가장 크고, 들어갈 때는 거의 없습니다. 그러니 확인하려면 캐시가 차가운 상태에서 시험하십시오. 최단 경로 쿼리에는 도움을 기대하지 마십시오. 그것은 I/O가 아니라 직렬 탐색에 매여 있습니다.

로깅

`log_statement` — 기본값은 `none` 입니다. 2.18의 변경에 유의하십시오. 이제 `mod` 가 Cypher 쓰기를 기록합니다. 2.18 이전에는 Cypher 문장이 `all` 에서만 기록 대상이었으므로, 데이터를 바꾸는 문장을 남기라고 한 서버가 SQL `INSERT` 는 남기고 그 옆의 `CREATE` 는 조용히 건너뛰었습니다.

`log_min_duration_statement` — 그래프 워크로드가 느린데 어느 쿼리인지 모를 때 가장 먼저 살펴볼 설정입니다. 임계값을 넘는 모든 문장을 남기며, Cypher도 포함합니다.

`auto_explain` — contrib 모듈이며, 그래프 워크로드에서 가장 쓸모 있는 진단 수단입니다. 느린 순회는 대개 데이터가 늘어난 것이 아니라 계획이 바뀌었다는 뜻이기 때문입니다:

```
shared_preload_libraries = 'auto_explain'
auto_explain.log_min_duration = '3s'
auto_explain.log_analyze = on
```

AgensGraph가 더하는 설정

열한 개이며 서버 설정이 다룹니다. `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` 입니다.

운영에 관계되는 것은 다음과 같습니다:

- **`graph_path`** — 세션이 작업하는 그래프입니다. 이것이 없으면 아무것도 해석되지 않습니다.
- **`auto_gather_graphmeta`** — 플래너가 스캔을 쳐낼 수 있도록 `ag_graphmeta` 를 최신으로 유지합니다. 쓰기마다 비용이 들므로 기본으로 꺼져 있습니다. 그 득실과 `regather_graphmeta()` 는 **그래프 메타데이터**를 참고하십시오.
- **`enable_graph_dml`** 과 **`enable_graph_ddl`** — 기본으로 꺼져 있으며, SQL로 레이블에 쓰는 것과 레이블의 모양을 바꾸는 것을 막습니다. 정비 작업이 아닐 때는 꺼 두십시오.
- **`eager_mem`** — eager 그래프 쓰기 계획이 `work_mem` 과 별개로 쓸 수 있는 메모리입니다.

다음에 볼 것

- **배포** — 용량 산정, 데이터 디렉터리, systemd에서의 운영.

© 2026 SKAI Worldwide Co., Ltd.

- **보안** — 인증, 롤, 그래프에 대한 권한.
- **쿼리 성능** — 계획을 상대로 하는 튜닝이며, 이 페이지와는 다른 문제입니다.

보안

AgensGraph의 보안 모델은 PostgreSQL의 것입니다. 롤·`pg_hba.conf`·TLS·`GRANT`·행 수준 보안이 모두 거기서와 똑같이 동작하며, **PostgreSQL의 클라이언트 인증** 장과 **데이터베이스 롤** 장이 레퍼런스입니다.

이 페이지가 더하는 것은 그 장치들이 그래프에 어떻게 내려앉는가입니다. 그래프는 스키마이고 레이블은 테이블이므로, Cypher 쿼리에 필요한 권한이 짐작한 것과 늘 같지는 않습니다.

인증

데이터 디렉터리의 `pg_hba.conf`가 누가 어디서 어느 데이터베이스에 어떻게 접속할 수 있는지 정합니다. 규칙은 위에서 아래로 대조하며 먼저 맞는 것이 이깁니다.

```
# 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
```

고친 뒤에는 reload 하십시오:

```
ag_ctl -D $AGDATA reload
```

서버가 외부에서 닿을 수 있게 되기 전에 바로잡아야 할 것 셋입니다:

기본 `trust` 규칙을 바꾸십시오. `initdb`는 로컬 접속에 `trust`를 써 두는데, 이는 셸 계정이 있는 사람이면 누구나 슈퍼유저를 포함해 어떤 롤로도 접속할 수 있다는 뜻입니다. 로컬 유닉스 소켓 접속에는 `peer`를, 네트워크를 지나는 것에는 `scram-sha-256`을 쓰십시오.

`md5`가 아니라 `scram-sha-256`을 쓰십시오. 비밀번호를 만들거나 바꾸기 전에 `password_encryption = scram-sha-256`을 설정해, 저장되는 검증자가 강한 쪽이 되게 하십시오.

네트워크 접속에는 TLS를 요구하십시오. 규칙에는 `hostssl`을, `postgresql.conf`에는 `ssl = on`과 인증서·키를 둡니다. 가로채기를 실제로 막는 것은 클라이언트의 `sslmode=verify-full`입니다. `require`는 암호화만 합니다.

쓸 수 있는 방식은 PostgreSQL의 전체 목록 — `peer`·`scram-sha-256`·`cert`·`ldap`·`gss`·`sspi`·`pam`·`radius`·`ident`·`trust`·`reject` — 이며 옵션도 같습니다. **PostgreSQL의 인증 방식** 장을 참고하십시오.

접속이 거부되면 서버 로그가 이유를 말해 주며, 그 메시지는 클라이언트가 받는 것보다 구체적입니다. 규칙을 고치기 전에 그것을 확인하십시오.

롤

롤은 클러스터 전체의 것입니다. 데이터베이스가 아니라 서버에 속합니다. `LOGIN`을 가진 롤이 다른 시스템에서 말하는 사용자입니다.

```
CREATE ROLE appuser LOGIN PASSWORD 'secret';
CREATE ROLE readonly;                       -- a group, no LOGIN
GRANT readonly TO appuser;
```

운영상 중요한 속성은 `SUPERUSER`·`CREATEDB`·`CREATEROLE`·`REPLICATION`·`BYPASSRLS`입니다. 애플리케이션 롤에는 그중 무엇도 주지 마십시오. `ALTER ROLE … SET`은 롤에 설정을 고정하는데, `graph_path`를 강제하거나 정비용 롤 말고는 `enable_graph_dml`을 꺼 두는 깔끔한 방법입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
ALTER ROLE appuser SET graph_path = moviekb;
ALTER ROLE appuser SET enable_graph_dml = off;
```

그래프에 대한 권한

그래프는 스키마이고 각 레이블은 테이블이므로 권한도 평범한 것들입니다. 놀랄 일은 다른 데서 옵니다. Cypher 쿼리는 자기가 이름을 댄 레이블보다 많은 객체에 손을 댑니다.

읽기

읽기에는 그래프 스키마에 대한 `USAGE`와, 손대는 각 레이블에 대한 `SELECT`가 필요합니다. 어느 지점에서 어떤 권한이 필요해지는지 다음 순서로 확인하십시오. 스키마만 준 상태에서 시작합니다:

```
CREATE ROLE analyst;
GRANT USAGE ON SCHEMA moviekb TO analyst;
```

```
SET ROLE analyst;
```

이제 Cypher 읽기가 레이블 이름과 함께 테이블 권한 오류로 실패합니다:

```
MATCH (m:movie) RETURN count(m);
```

```
ERROR:  permission denied for table movie
```

레이블에 권한을 주면 동작합니다:

```
RESET ROLE;
GRANT SELECT ON moviekb.movie TO analyst;
SET ROLE analyst;
```

```
MATCH (m:movie) RETURN count(m);
```

```
 count
-------
 18
```

레이블 없는 패턴에는 `ag_vertex`도 필요합니다. `MATCH (n)`은 그래프의 부모인 `ag_vertex`와 그 아래의 모든 레이블을 스캔하므로, `movie`는 읽을 수 있지만 `ag_vertex`는 읽지 못하는 롤은 이것을 실행하지 못합니다:

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

```
ERROR:  permission denied for table ag_vertex
```

레이블 없는 간선에 대해서도 `ag_edge`에 같은 것이 적용됩니다. 그러므로 읽기 전용 분석가 롤에는 레이블을 하나씩 주기보다 스키마 전체에 주십시오:

```
RESET ROLE;
```


© 2026 SKAI Worldwide Co., Ltd.

```
GRANT SELECT ON ALL TABLES IN SCHEMA moviekb TO analyst;
ALTER DEFAULT PRIVILEGES IN SCHEMA moviekb GRANT SELECT ON TABLES TO analyst;
```

다음 달에 만들어질 레이블까지 읽히게 하는 것이 `ALTER DEFAULT PRIVILEGES` 줄입니다.

쓰기

정점이나 간선을 만들려면 레이블에 대한 `INSERT`와 **그 레이블 id 시퀀스에 대한** `USAGE` **권한**이 필요합니다. 두 번째 것을 빠뜨리기 쉽습니다. `INSERT`만 준 경우입니다:

```
RESET ROLE;
```

```
GRANT INSERT ON moviekb.movie TO analyst;
```

```
SET ROLE analyst;
```

```
CREATE (:movie {title: 'X'});
```

```
ERROR:  permission denied for sequence movie_id_seq
```

시퀀스를 더하면 쓰기가 통과합니다:

```
RESET ROLE;
```

```
GRANT USAGE ON SEQUENCE moviekb.movie_id_seq TO analyst;
```

`SET`과 `REMOVE`에는 레이블에 대한 `UPDATE`가 필요합니다. `DELETE`에는 `DELETE`가 필요합니다. 그리고 `DETACH DELETE`에는 `ag_edge`에 대한 권한이 필요합니다. 지울 간선을 그것으로 찾기 때문입니다:

```
ERROR:  permission denied for table ag_edge
```

`ag_edge`나 `ag_vertex`를 가리키는 오류는 행 보안의 결과가 아니라 그래프의 부모 레이블에 대한 권한이 없다는 뜻입니다. 행 보안 때문으로 오해하기 쉬우므로 알아 둘 만합니다.

그래프 전체를 읽고 쓰는 롤에 쓸 만한 권한 묶음입니다:

```
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;
```

레이블 만들기

레이블을 만들려면 그래프 스키마에 대한 `CREATE`가 필요합니다. 그래프를 만들려면 데이터베이스에 대한 `CREATE`가 필요합니다.

© 2026 SKAI Worldwide Co., Ltd.

행 수준 보안

행 수준 보안은 레이블에서 동작하므로, 정책으로 어떤 롤이 어떤 정점이나 간선을 보고 쓰는지 제한할 수 있습니다:

```
ALTER TABLE moviekb.movie ENABLE ROW LEVEL SECURITY;

CREATE POLICY movie_visible ON moviekb.movie
  FOR SELECT TO analyst
  USING ((properties->>'genre') = 'Sci-Fi');
```

경고

2.17에서 쓰던 RLS 정책은 다시 시험하십시오. 2.18 이전에는 그래프 쓰기에 정책이 제대로 적용되지 않았습니다. `UPDATE`나 `DELETE` 정책의 `USING` 절이 결코 스캔되지 않는 릴레이션에 붙어 있어 아무것도 거르지 못했고, `DELETE` 정책이 아예 없는 롤이 행을 지울 수 있었습니다. `INSERT` 정책은 엉뚱한 행을 대상으로 검사됐고, 간선 생성은 아예 검사되지 않아 행 보안이 걸린 간선 레이블이 모든 삽입을 받아들였습니다.

2.18은 그 전부를 적용합니다. 그것이 올바른 동작이며, 지금 애플리케이션이 하고 있는 쓰기가 거부되기 시작할 수 있다는 뜻입니다. 업그레이드한 뒤가 아니라 전에 시험하십시오. **릴리스 노트**를 참고하십시오.

실무에서 알아 둘 것 둘입니다:

- 정책 표현식은 레이블의 컬럼을 읽으므로 `properties`에서 동작하고, **승격된 컬럼**에서도 동작합니다. 뒤쪽이 더 빠르고 정책이 인덱스를 쓸 수 있게 합니다.
- `BYPASSRLS`와 테이블 소유권은 둘 다 정책을 건너뜁니다. 애플리케이션 롤에는 둘 중 어느 것도 주지 마십시오.

보안 태세의 일부인 설정 둘

`enable_graph_dml`과 `enable_graph_ddl`은 기본으로 꺼져 있으며, Cypher가 아니라 SQL로 레이블에 쓰는 것과 레이블의 모양을 바꾸는 것을 막습니다. 꺼 두면 애플리케이션이 Cypher 경로에 머무르고, 그래프 자신의 검사 — 이를테면 간선의 양 끝점이 있는지 — 가 그대로 동작합니다.

다만 이 설정이 그래프 정합성의 전부는 아니며, 그렇게 읽어서는 안 됩니다. `COPY`는 두 설정을 보지 않고 레이블에 씁니다. 설정과 무관하게 모든 경로에서 지켜지는 것은 행의 id가 그 행을 담은 레이블을 가리킨다는 것입니다. 이를 깨뜨리는 쓰기는 Cypher든 `INSERT`든 `UPDATE`든 `COPY`든 복제 반영이든 거부됩니다.

정비 작업 때 켰다가 다시 끄거나, `ALTER ROLE … SET`으로 롤마다 꺼 두십시오.

`cypher_allow_unsafe_ddl`도 마찬가지입니다. **서버 설정**을 참고하십시오.

저장 데이터 암호화

AgensGraph는 데이터 파일 자체를 암호화하지 않습니다. 파일시스템이나 블록 장치 암호화를 쓰거나, 투명 데이터 암호화가 요구되는 곳에는 **Agens TDE**를 쓰십시오. 컬럼 수준 암호화는 `pgcrypto`가 다루며, 속성은 평범한 `jsonb` 값이므로 다른 컬럼과 마찬가지로 `pgcrypto` 암호문을 담을 수 있습니다.

백업과 복구

그래프는 PostgreSQL 스키마와 테이블로 저장되므로, 모든 PostgreSQL 백업 방법이 그래프를 담으며, 관계형 데이터와 같은 시점으로 담습니다. 따로 받아야 할 그래프 백업도, 따로 맞춰야 할 복원도 없습니다.

그러므로 **PostgreSQL의 백업** 장이 레퍼런스이고, 이 페이지는 선택지와 그래프에만 해당하는 세부를 다룹니다.

© 2026 SKAI Worldwide Co., Ltd.

어느 방법을 쓸 것인가

방법	얻는 것	언제 쓰는가
`pg_dump` / `pg_dumpall`	논리적 사본입니다. SQL 문장 또는 아카이브이며, 메이저 버전과 플랫폼을 넘어 이식됩니다.	데이터베이스를 옮길 때, 그래프 하나를 복사할 때, 버전에 매이지 않는 사본을 둘 때.
`pg_basebackup`	클러스터 전체의 물리적 사본입니다.	시점 복구 구성의 기반, 그리고 복제본을 처음 채울 때.
연속 아카이빙	베이스 백업에 그 뒤의 모든 WAL 세그먼트를 더한 것이며, 어느 순간으로든 복구할 수 있습니다.	운영 환경. 한 시간 치 쓰기를 잃는 것이 용납되지 않는 곳.
파일시스템 스냅샷	스토리지 계층이 뜨는 물리적 사본입니다.	클러스터가 쓰는 모든 파일시스템에 걸쳐 스냅샷이 원자적일 때만.

서버가 도는 중에 `cp`로 데이터 디렉터리를 복사하는 것은 백업이 아닙니다. 테이블 파일을 하나씩 복사하는 것은 결코 백업이 아닙니다. 커밋 로그는 클러스터 전체의 것이므로, 홀로 떠 온 파일은 자기 행 중 무엇이 커밋됐는지 말할 방법이 없습니다.

논리 백업

```
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
```

복원:

```
pg_restore --exit-on-error -d mydb mydb.dump
psql -d postgres -f cluster.sql
```

경고

두 도구 모두 기본값이 오류를 만나도 계속 진행하는 것이고, `psql`은 그러고도 종료 코드 0을 냅니다. 그래서 객체가 빠진 채로 복원이 "성공"이라고 보고될 수 있습니다 — 서버가 거부한 제약이 있으면 그 테이블은 제약 없이 만들어지고, 그다음부터 존재하지 않는 행을 참조하는 행도 그대로 들어갑니다. `--exit-on-error`를 주면 `pg_restore`가 첫 오류에서 멈춥니다.

`psql`에는 `ON_ERROR_STOP`이 있지만 `pg_dumpall` 산출물에는 쓸 수 없습니다. 그 파일은 이미 존재하는 부트스트랩 슈퍼유저를 다시 만들기 때문에, 14행에서 멈추고 아무것도 적재하지 않습니다. 그 경우에는 출력을 직접 읽거나, 데이터베이스마다 `pg_dump -Fc`로 따로 덤프해서 `pg_restore --exit-on-error`로 복원하십시오.

그래프가 곧 스키마이므로, 그래프 하나를 덤프하는 방법은 그 스키마를 지정하는 것입니다. 덤프에는 데이터와 함께 `CREATE GRAPH` · `CREATE VLABEL` · `CREATE ELABEL` 문장이 담기므로, 복원하면 그래프 구조가 다시 만들어집니다.
그래프에만 해당하는 것 네 가지이며, 자세한 내용은 모두 **서버 측 도구**에 있습니다:

- 두 도구 모두 AgensGraph 버전을 검사하지 않고 PostgreSQL 버전만 검사합니다.
- 평범한 컬럼을 지닌 레이블의 덤프는 복원할 때 `enable_graph_ddl`이 필요하며, 덤프가 자기 문장을 그것으로 감쌉니다.
- 어떤 그래프든 승격된 컬럼을 쓰고 있으면 2.18 덤프는 2.17에 복원되지 않습니다.
- Cypher 쿼리로 만든 뷰는 그 Cypher가 컴파일된 SQL로 덤프됩니다.

© 2026 SKAI Worldwide Co., Ltd.

어떤 복원이든 끝난 뒤에는 그래프 메타데이터와 통계를 다시 만드십시오:

```
SET graph_path = <your graph>;
SELECT regather_graphmeta();
```

```
vacuumdb -d mydb --analyze-in-stages
```

물리 백업과 시점 복구

동작 방식은 PostgreSQL의 것 그대로입니다. 요약하면:

서버를 아카이빙하도록 설정하고 재기동합니다:

```
wal_level = replica
archive_mode = on
archive_command = 'test ! -f /mnt/archive/%f && cp %p /mnt/archive/%f'
```

`wal_level = minimal`은 아카이브에서 복구할 만큼을 기록하지 않으므로, 이 방법을 쓸 수 없게 만듭니다.

베이스 백업을 받습니다:

```
pg_basebackup -D /mnt/backup/base -Ft -z -P -X stream
```

복구하려면 베이스 백업을 빈 데이터 디렉터리에 풀고, 아카이브된 WAL을 어디서 가져올지 서버에 알려 줍니다:

```
restore_command = 'cp /mnt/archive/%f %p'
recovery_target_time = '2026-09-02 14:30:00'
```

이 값들은 `postgresql.conf`에 넣고, 데이터 디렉터리에 둔 `recovery.signal`이라는 빈 파일이 서버를 복구 모드로 들어가게 합니다. 복구가 끝나면 서버는 그 신호 파일을 지우고 접속을 받기 시작합니다.

> 참고
>
> 예전 AgensGraph나 PostgreSQL 문서를 보고 작업한다면 자리를 옮긴 것이 셋 있습니다. 복구 설정은 `recovery.conf`가 아니라 `recovery.signal` 파일과 함께 `postgresql.conf`에 있습니다. `recovery.conf`는 PostgreSQL 12에서 없어졌고, 서버는 그것을 발견하면 기동을 거부합니다. WAL 디렉터리는 `pg_xlog`가 아니라 `pg_wal`이고, 커밋 로그는 `pg_clog`가 아니라 `pg_xact`입니다. 둘 다 PostgreSQL 10에서 이름이 바뀌었습니다.

베이스 백업에 기대기 전에 그것을 검증하십시오:

```
pg_verifybackup /mnt/backup/base
```

더 높은 수준의 도구

`pgBackRest`와 `Barman`은 보관 기간·병렬 압축·증분 백업·복원 절차를 관리하며, 둘 다 그래프에만 해당하는 설정 없이 AgensGraph 클러스터에서 동작합니다. 서버를 둘 이상 운영한다면 손으로 쓴 `archive_command`보다 어느 쪽이든 더 나은 답입니다.

무엇을 확인할 것인가

복원해 보지 않은 백업은 백업이 아니라 바람입니다. 특히 그래프라면, 복원 시험은 서버가 뜬다는 것 이상을 확인해야 합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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;
```

원본과 일치하는 개수, 그리고 인덱스와 제약이 제자리에 있다는 것이 그래프가 온전히 돌아왔음을 말해 줍니다.

서버 측 도구

AgensGraph에 딸려 오는 도구는 PostgreSQL의 것이므로 **PostgreSQL의 해당 레퍼런스**가 그대로 적용되며 여기서 되풀이하지 않습니다. 이 페이지는 다른 점, 그리고 가장 많이 쓰는 도구에 그래프가 더하는 것을 다룹니다.

대화형 클라이언트와 그 그래프 메타 명령은 **agens 클라이언트**를 참고하십시오.

AgensGraph가 도구 모음에 더하는 것

도구 두 개가 두 번째 이름으로도 배포됩니다:

AgensGraph 이름	PostgreSQL 이름	차이
`agens`	`psql`	브랜딩뿐입니다. 둘 다 설치되며, 그래프 메타 명령은 어느 쪽에서도 동작합니다.
`ag_ctl`	`pg_ctl`	브랜딩뿐입니다. 둘 다 설치됩니다.

```
agens --version
```

```
agens (PostgreSQL) 18.6 (AgensGraph 2.18.6.1)
```

> **참고**
>
> 개발 브랜치에서 빌드한 것은 대신 `-devel` 접미사를 붙여 알려 줍니다.

```
psql --version
```

```
psql (PostgreSQL) 18.6 (AgensGraph 2.18.6.1)
```

그 밖의 모든 것 — `initdb` · `pg_dump` · `pg_dumpall` · `pg_restore` · `pg_upgrade` · `pg_basebackup` · `pg_controldata` · `pg_checksums` · `vacuumdb` · `reindexdb` · `createdb` 와 나머지 — 은 PostgreSQL 이름과 PostgreSQL 동작을 그대로 지닙니다.

© 2026 SKAI Worldwide Co., Ltd.

`AGDATA`는 기본 데이터 디렉터리로서 `PGDATA`와 나란히 동작하므로, `export AGDATA=/path/to/data`를 해 두면 `-D`를 생략할 수 있습니다. 이것을 읽는 바이너리는 정확히 셋 — `initdb` · `pg_ctl` · `ag_ctl` — 이고, 우선순위가 서로 다릅니다. `initdb`는 `PGDATA`보다 `AGDATA`를 앞세우고, `pg_ctl`과 `ag_ctl`은 `PGDATA`를 앞세웁니다. 그 밖의 모든 도구는 `PGDATA`만 받습니다.

기동과 정지

```
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
```

systemd에서는 PostgreSQL 자신의 유닛 관례를 쓰고 그것이 AgensGraph 바이너리를 가리키게 하십시오. 여기에 그래프에만 해당하는 것은 없습니다. **배포**를 참고하십시오.

그래프 덤프하기

그래프는 스키마이고 그 레이블은 테이블이므로, `pg_dump`와 `pg_dumpall`은 특별한 옵션 없이 그래프를 다룹니다. 따라서 덤프에는 `CREATE GRAPH` · `CREATE VLABEL` · `CREATE ELABEL` 문장과 레이블 데이터가 담깁니다.

```
pg_dump -d mydb -f mydb.sql
pg_dump -d mydb -Fc -f mydb.dump
```

그래프 하나만 덤프하려면 그 스키마를 지정하십시오:

```
pg_dump -d mydb --schema=moviekb -f moviekb.sql
```

그래프에만 해당하는 알아 둘 것 네 가지입니다:

- **두 도구 모두 AgensGraph 버전을 검사하지 않습니다.** PostgreSQL 자신의 `server_version` 범위 검사만 적용되므로, 더 새 덤프를 더 낡은 서버에 복원하면 처음부터 거부되지 않고 복원 도중에 파싱 실패로 드러납니다.
- **평범한 컬럼을 지닌 레이블의 덤프는 그 `ALTER TABLE` 문장을 `SET enable_graph_ddl = on`으로 감쌉니다.** 그 설정은 어느 롤이든 쓸 수 있으므로 평범한 사용자도 복원할 수 있습니다. 그 롤에 `ALTER ROLE … SET`으로 강제로 꺼 두지 않았다면 말입니다.
- **어떤 그래프든 승격된 컬럼을 쓰고 있으면 2.18 덤프는 2.17에 복원되지 않습니다.** 내보내진 `CREATE VLABEL` 형태를 2.17 파서가 받아들이지 못하며, 평문 스트림에서는 그 문장 뒤의 모든 것이 유실됩니다.
- **Cypher 쿼리로 정의한 뷰는 그 Cypher가 컴파일된 SQL로 덤프됩니다.** 사용자가 쓴 Cypher가 아닙니다. 복원되고 동작하지만, 되돌려 받는 정의는 사용자가 쓴 정의가 아닙니다. **SQL과 Cypher 섞기**를 참고하십시오.

복원한 뒤에는 그래프 메타데이터를 다시 만들고 다시 `ANALYZE` 하십시오. 정확한 명령은 **2.18로 업그레이드하기**에 있으며, 업그레이드뿐 아니라 모든 복원에 적용됩니다.

메이저 버전 간 업그레이드

`pg_upgrade`는 특별한 처리 없이 그래프에서 동작하며, AgensGraph에만 해당하는 조건이 둘 있습니다:

- `pg_upgrade`는 레이블에 `id` · `start` · `end` · `properties` 말고 다른 컬럼이 있는 그래프를, 물리적 컬럼 배치를 망가뜨리는 대신 거부합니다.
- 끝난 뒤에는 `ag_graphmeta`가 비어 있고 그래프가 무효로 표시되므로, `regather_graphmeta()`를 실행하기 전까지 플래너의 스캔 쳐내기가 꺼져 있습니다.

전체 절차는 **2.18로 업그레이드하기**에 있고, **업그레이드 실습**은 실제 출력을 곁들인 예제입니다.

© 2026 SKAI Worldwide Co., Ltd.

베이스 백업과 WAL

`pg_basebackup` ·WAL 아카이빙· `pg_receivewal` · `pg_verifybackup` · `pg_rewind` 는 모두 그래프를 있는 그대로 — 같은 클러스터 안의 스키마와 테이블로 — 다루므로, 물리 백업은 그래프 데이터와 관계형 데이터를 같은 시점에 함께 담습니다. **백업과 복구**를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

시스템 카탈로그

그래프를 설명하는 카탈로그와 뷰, 그리고 AgensGraph가 그래프에 대해 수집하는 메타데이터입니다.

그래프 카탈로그

AgensGraph는 `pg_catalog`에 테이블 일곱 개를 더합니다. 그중 넷이 그래프 구조를 설명하며, 조회하게 될 것은 그 넷입니다. 셋은 모든 레이블이 상속하는 공용 부모 릴레이션입니다. PostgreSQL 자신의 카탈로그는 바뀌지 않았고 **그 카탈로그 장**에 문서화돼 있습니다.

카탈로그	담는 것
`ag_graph`	그래프마다 한 행입니다.
`ag_label`	레이블마다 한 행이며 정점과 간선을 가리지 않습니다.
`ag_label_property`	타입 컬럼으로 승격된 속성마다 한 행입니다.
`ag_graphmeta`	(시작 레이블, 간선 레이블, 끝 레이블) 조합마다의 간선 수입니다.
`ag_vertex`, `ag_edge`	그래프 안 모든 정점·간선 레이블의 부모 릴레이션입니다. 각 그래프의 스키마 안에 한 쌍씩 있으며 `pg_catalog`에 있지 않습니다.
`ag_graphpath`	`graphpath` 타입을 떠받치는 복합 타입입니다.

같은 정보를 읽기 좋은 형태로 보여 주는 뷰가 둘 있습니다. **그래프 뷰**를 참고하십시오.

ag_graph

그래프마다 한 행입니다. 그래프는 스키마이므로 `nspid`가 그 레이블들을 담은 스키마를 가리킵니다.

컬럼	타입	설명
`oid`	`oid`	행 식별자입니다.
`graphname`	`name`	그래프의 이름이며 데이터베이스 안에서 고유합니다.
`nspid`	`oid`	그래프 스키마에 해당하는 `pg_namespace` 행입니다.
`graphmeta_valid`	`boolean`	이 그래프에 대해 `ag_graphmeta`가 최신인지 여부입니다. 거짓이면 플래너가 그것을 스캔 쳐내기에 쓸 수 없습니다.

© 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
```

메타데이터가 낡았음이 분명할 때, 이를테면 바이너리 업그레이드 뒤에 `graphmeta_valid`가 거짓으로 설정됩니다. `SELECT regather_graphmeta();` 가 그것을 다시 만들고 플래그를 세웁니다. **그래프 메타데이터**를 참고하십시오.

ag_label

레이블마다 한 행입니다. 정점 레이블과 간선 레이블이 함께 있으며 `labkind`로 구분합니다.

컬럼	타입	설명
`oid`	`oid`	행 식별자입니다.
`labname`	`name`	레이블의 이름이며 그래프 안에서 고유합니다.
`graphid`	`oid`	이 레이블이 속한 `ag_graph` 행입니다.
`labid`	`integer`	그래프 안에서의 레이블 id이며, 그 안 모든 `graphid`의 상위 16비트입니다.
`relid`	`oid`	레이블 테이블에 해당하는 `pg_class` 행입니다.
`labkind`	`"char"`	정점 레이블이면 `v`, 간선 레이블이면 `e`입니다.

```
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` 1과 2는 언제나 그래프와 함께 만들어지는 그 그래프의 `ag_vertex`와 `ag_edge`입니다. `labid`가 16비트이고 0은 유효하지 않으므로, 그래프 하나는 최대 65,535개의 레이블을 담습니다. **제한**을 참고하십시오.

`relid`를 `pg_class`에 조인하면 레이블에서 그 테이블로 갈 수 있습니다. 평범한 PostgreSQL 인트로스펙션이 그래프에서도 통하게 하는 것이 그 조인입니다:

© 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

타입 컬럼으로 승격된 속성마다 한 행입니다. 이 카탈로그는 2.18에서 새로 생겼습니다.

컬럼	타입	설명
`laboid`	`oid`	승격된 컬럼이 속한 `ag_label` 행입니다.
`propname`	`name`	그 컬럼이 담는 속성 키입니다.
`attnum`	`smallint`	레이블 테이블에서 그 컬럼의 속성 번호입니다.
`semantics`	`"char"`	그 컬럼이 속성과 어떤 관계인지입니다.

기본 키가 `(laboid, propname)`이므로, 레이블 하나는 같은 키를 많아야 한 번 승격합니다.

속성 읽기가 승격된 컬럼으로 해석되는 것은 여기에 행이 있을 때뿐입니다. 흔히 혼동을 부르는 다음 승격 방식도 이 장치 때문입니다. `GENERATED ALWAYS AS (expr) STORED`는 컬럼을 만들지만 이 카탈로그에 행을 등록하지 않습니다. 그래서 그 컬럼은 SQL에서는 읽히고 Cypher의 속성 읽기에는 보이지 않습니다.

ag_graphmeta

그래프에 실제로 나타나는 (시작 레이블, 간선 레이블, 끝 레이블) 조합마다 한 행이며, 그것을 이루는 간선이 몇 개인지가 함께 있습니다. 플래너는 이것을 두 가지로 씁니다. 순회를 추정하는 데, 그리고 패턴에 참여할 수 없는 레이블의 스캔을 쳐내는 데 씁니다.

컬럼	타입	설명
`graph`	`oid`	`ag_graph` 행입니다.
`edge`	`smallint`	간선 레이블의 `labid` 입니다.
`start`	`smallint`	시작 정점 레이블의 `labid` 입니다.
`end`	`smallint`	끝 정점 레이블의 `labid` 입니다.
`edgecount`	`bigint`	그 모양의 간선이 몇 개인지입니다.

레이블 id를 이름으로 풀어 주는 **`ag_graphmeta_view`**로 읽으십시오. 이 테이블은 `auto_gather_graphmeta`가 켜져 있을 때만 유지되며, `regather_graphmeta()`가 그것을 처음부터 다시 만듭니다. 둘 다 **그래프 메타데이터**가 다룹니다.

© 2026 SKAI Worldwide Co., Ltd.

ag_vertex와 ag_edge

이 둘은 `pg_catalog`에 없습니다. 모든 그래프의 스키마가 자기 것 한 쌍을 담고 있고, 사용자가 만드는 모든 레이블이 그중 하나를 상속합니다. 레이블 없는 패턴에 뜻을 주는 것이 그 상속입니다. **그래프가 어떻게 저장되는가**를 참고하십시오.

`<graph>.ag_vertex`는 정점의 컬럼 쌍 `(id graphid, properties jsonb)`를 가집니다. `<graph>.ag_edge`는 거기에 양 끝점을 더해 `(id, start, end, properties)`를 가집니다.

ag_graphpath

`graphpath` 뒤에 있는 복합 타입입니다. 구현 세부입니다. 이것을 직접 건드리지 말고 패턴 구문과 경로 함수로 경로를 만들고 읽으십시오.

그래프 뷰

`pg_catalog`의 뷰 두 개가 oid와 레이블 id를 이름으로 풀어 그래프 카탈로그를 읽기 좋은 형태로 보여 줍니다. 둘 다 평범한 뷰이므로 조인하고 걸러 낼 수 있습니다.

ag_property_indexes

속성 인덱스마다 한 행이며, 그것을 다시 만들 문장이 함께 있습니다.

컬럼	타입	설명
`graphname`	`name`	인덱스가 속한 그래프입니다.
`labelname`	`name`	인덱스가 걸린 레이블입니다.
`indexname`	`name`	인덱스의 이름입니다.
`tablespace`	`name`	테이블스페이스이며, 기본값이면 null입니다.
`unique`	`boolean`	인덱스가 고유한지 여부입니다.
`indexdef`	`text`	인덱스를 다시 만드는 `CREATE PROPERTY INDEX` 문장입니다.
`owner`	`name`	인덱스를 소유한 롤입니다.
`size`	`text`	인덱스의 크기이며 이미 서식이 갖춰져 있습니다.
`description`	`text`	인덱스에 달린 주석이 있으면 그것입니다.

예제 데이터셋에는 속성 인덱스가 없으므로, 볼 것을 만들기 위해 두 개를 만듭니다:

```
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
```

다 보았으면 `DROP PROPERTY INDEX movie_title_idx;` 로 다시 지우십시오.

쓸모 있는 것은 `indexdef` 컬럼입니다. 서버가 만든 표현식 인덱스가 아니라 속성 수준의 문장을 돌려주는데, 스키마를 스크립트로 만들 때 원하는 형태가 그것입니다. agens 클라이언트의 `\dGi` 도 같은 정보를 출력합니다.

이 뷰는 `properties` 에 대한 인덱스뿐 아니라 **승격된 컬럼**에 대한 인덱스도 나열합니다. 그러므로 「이 레이블에는 무엇이 인덱싱돼 있는가」를 볼 곳은 여기 하나입니다.

ag_graphmeta_view

그래프에 실제로 나타나는 간선 모양마다 한 행이며, 레이블 id 대신 이름이 들어 있습니다.

컬럼	타입	설명
`graphname`	`name`	그래프입니다.
`start`	`name`	간선 시작 정점의 레이블입니다.
`edge`	`name`	간선 레이블입니다.
`end`	`name`	간선 끝 정점의 레이블입니다.
`edgecount`	`bigint`	그 모양의 간선이 몇 개인지입니다.

```
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
```

이것이 AgensGraph가 줄 수 있는, 그래프의 실제 모양에 대한 가장 또렷한 그림입니다. 쿼리를 튜닝하기 전에 읽으십시오. 어떤 순회가 존재하고 각각이 얼마나 큰지 알려 줍니다. 플래너도 같은 행을 씁니다.

메타데이터를 수집하지 않았다면 이 뷰는 비어 있습니다. `auto_gather_graphmeta` 와 `regather_graphmeta()` 는 **그래프 메타데이터**를 참고하고, 수치가 이상해 보이면 `ag_graph.graphmeta_valid` 를 확인하십시오.

© 2026 SKAI Worldwide Co., Ltd.

그래프 메타데이터

소개

meta 확장은 그래프 메타데이터를 다루는 일을 간단하게 해 주는 AgensGraph용 보조 함수 모음입니다. 이 함수들은 그래프 구조를 살피고, 레이블을 확인하고, 크기를 추정하고, 관계를 분석하고, Cypher 타입을 다룹니다.

AgensGraph의 그래프를 더 적은 되풀이와 더 적은 수고로 들여다보고, 감시하고, 관리할 수 있게 해 줍니다.

설치

meta 확장은 AgensGraph의 contrib 디렉터리에 들어 있습니다. 설치하려면 먼저 contrib 모듈을 켠 채로 AgensGraph를 빌드해 설치하십시오.

설치한 뒤에는 다음으로 어느 데이터베이스에나 확장을 추가할 수 있습니다:

```
CREATE EXTENSION meta;
```

그러면 그 데이터베이스에서 모든 meta 함수를 쓸 수 있습니다.

함수

meta 확장에는 다음 함수들이 들어 있습니다:

meta.graphs

설명
쓸 수 있는 모든 그래프의 목록을 반환합니다.

인수

- 없음

반환

- 그래프 이름의 집합

예제

```
SELECT * FROM meta.graphs();

 graphname
-----------
 social
 company
 test
```

meta.labels

설명
주어진 그래프의 모든 레이블을 반환합니다. 시스템 레이블인 `ag_vertex`와 `ag_edge`는 결과에 들어가지 않습니다. `graph` 인수를 주지 않으면 현재 `graph_path` 설정을 씁니다.

인수

- `graph` (name, 선택): 그래프의 이름입니다. 기본값은 현재 `graph_path`입니다.

© 2026 SKAI Worldwide Co., Ltd.

반환

- `label_name` (name): 지정한 그래프의 모든 레이블 이름이며, 시스템 레이블 `ag_vertex`와 `ag_edge`는 뺍니다.

예제

```
SELECT * FROM meta.labels('social');

 label_name
------------
 Person
 Product
 Transaction
```

meta.vertex_labels

설명

주어진 그래프에 정의된 모든 정점 레이블을 반환합니다. 시스템 레이블 `ag_vertex`는 뺍니다. `graph` 인수를 주지 않으면 현재 `graph_path` 설정으로 그래프를 정합니다.

인수

- `graph` (name, 선택): 그래프의 이름입니다. 기본값은 현재 `graph_path` 입니다.

반환

- `label_name` (name): 지정한 그래프의 모든 정점 레이블 이름입니다.

예제

```
SELECT * FROM meta.vertex_labels('social');

 label_name
------------
 Person
 Product
 Comment
```

meta.edge_labels

설명 주어진 그래프에 정의된 모든 간선 레이블을 반환합니다. 시스템 레이블 ag_edge는 뺍니다. graph 인수를 주지 않으면 현재 graph_path 설정으로 그래프를 정합니다.

인수

- `graph` (name, 선택): 그래프의 이름입니다. 기본값은 현재 `graph_path` 입니다.

반환

- `label_name` (name): 지정한 그래프의 모든 간선 레이블 이름입니다.

예제

```
SELECT * FROM meta.edge_labels('social');

 label_name
------------
 Follows
```


© 2026 SKAI Worldwide Co., Ltd.

```
 Likes
 Purchases
```

meta.label_id

설명

주어진 그래프에서 특정 레이블의 내부 id를 반환합니다. `graph` 인수를 주지 않으면 현재 `graph_path` 설정을 씁니다. 그 레이블이 그래프에 없으면 예외를 냅니다.

인수

- `label` (name, 필수): id를 얻으려는 레이블의 이름입니다.
- `graph` (name, 선택): 그래프의 이름입니다. 기본값은 현재 `graph_path` 입니다.

반환

- `labid` (int): 지정한 레이블의 내부 id입니다.

예제

```
SELECT meta.label_id('Person', 'social');

 labid
-------
     5
```

meta.label_kind

설명

주어진 그래프에서 레이블의 종류(정점이면 `v`, 간선이면 `e`)를 반환합니다. `graph` 인수를 주지 않으면 현재 `graph_path` 설정을 씁니다. 그 레이블이 없으면 `NULL`을 반환합니다.

인수

- `label` (name, 필수): 종류를 얻으려는 레이블의 이름입니다.
- `graph` (name, 선택): 그래프의 이름입니다. 기본값은 현재 `graph_path` 입니다.

반환

- `kind` (char): 레이블 종류를 나타내는 한 글자(`v` 또는 `e`)입니다.

예제

```
SELECT meta.label_kind('Person', 'social');

 kind
------
 v
```

meta.label_name

설명

특정 그래프에서 ID(`labid`)나 OID로 레이블의 이름을 반환합니다. `graph` 인수를 주지 않으면 현재 `graph_path` 설정을 씁니다. 그 레이블이 없으면 notice를 냅니다.

© 2026 SKAI Worldwide Co., Ltd.

인수

- `labelid` (int, 필수): 레이블의 ID 또는 OID입니다.
- `graph` (name, 선택): 그래프의 이름입니다. 기본값은 현재 `graph_path` 입니다.

반환

- `label_name` (text): 레이블의 이름입니다.

예제

```
SELECT meta.label_name(5, 'social');

 label_name
------------
 Person
```

meta.graph_size

설명

그래프의 전체 크기를 모든 레이블을 포함해 반환합니다. `graph_name` 인수를 주지 않으면 현재 `graph_path` 설정을 씁니다. 크기는 사람이 읽기 좋은 형태로도, 바이트 수로도 반환할 수 있습니다.

인수

- `graph_name` (name, 선택): 그래프의 이름입니다. 기본값은 현재 `graph_path` 입니다.
- `pretty` (boolean, 선택): `true` (기본값)이면 사람이 읽기 좋은 크기(예: `42 MB`)를 반환하고, `false` 이면 바이트 수를 text로 반환합니다.

반환

- `size` (text): 그래프의 전체 크기입니다.

예제

```
SELECT meta.graph_size('social');

 graph_size
------------
 123 MB

SELECT meta.graph_size('social', false);

 graph_size
------------
 129394816
```

meta.label_size

설명

특정 레이블의 전체 크기를 인덱스를 포함해 반환합니다. `graph` 인수를 주지 않으면 현재 `graph_path` 설정을 씁니다. 크기는 사람이 읽기 좋은 형태로도, 바이트 수로도 반환할 수 있습니다.

인수

- `labelname` (text): 레이블의 이름입니다.

© 2026 SKAI Worldwide Co., Ltd.

- `graph` (name, 선택): 그 레이블이 든 그래프입니다. 기본값은 현재 `graph_path` 입니다.
- `pretty` (boolean, 선택): `true` (기본값)이면 사람이 읽기 좋은 크기(예: `42 MB`)를 반환하고, `false` 이면 바이트 수를 text로 반환합니다.

반환

- `size` (text): 인덱스를 포함한 레이블의 전체 크기입니다.

예제

```
SELECT meta.label_size('person');

 label_size
------------
 42 MB

SELECT meta.label_size('person', pretty => false);

 label_size
------------
 44040192
```

meta.count

설명

지정한 레이블의 정점 또는 간선 개수를 정확히 반환합니다. `graph` 인수를 주지 않으면 현재 `graph_path` 설정을 씁니다.

인수

- `labelname` (text): 요소 수를 셀 레이블입니다.
- `graph` (name, 선택): 그 레이블이 든 그래프입니다. 기본값은 현재 `graph_path` 입니다.

반환

- `count` (bigint): 그 레이블의 요소(정점 또는 간선) 개수입니다.

예제

```
-- 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');
```

meta.estimated_count

설명

PostgreSQL 통계(`pg_class.reltuples`)를 바탕으로 주어진 레이블의 요소(정점 또는 간선) 개수를 추정해 반환합니다. `graph` 인수를 주지 않으면 현재 `graph_path` 설정을 씁니다. `meta.count` 보다 빠르지만 정확하지 않을 수 있습니다.

> 참고: 통계가 없으면 NULL을 반환할 수 있습니다. `ANALYZE "graph"."label"` 로 통계를 수집하십시오.

인수

- `labelname` (text): 추정 요소 수를 얻을 레이블입니다.
- `graph` (name, 선택): 그 레이블이 든 그래프입니다. 기본값은 현재 `graph_path` 입니다.

© 2026 SKAI Worldwide Co., Ltd.

반환

- `estimated_count` (bigint): 그 레이블의 추정 요소 개수입니다.

예제

```
-- 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

설명

지정한 이름의 그래프가 있는지 확인합니다.

인수

- `graph` (name): 확인할 그래프의 이름입니다.

반환

- `boolean` : 그래프가 있으면 `true` , 없으면 `false` 입니다.

예제

```
SELECT meta.graph_exists('social');

 ?column?
----------
 t
```

meta.label_exists

설명

지정한 그래프에 레이블이 있는지 확인합니다. 그래프를 주지 않으면 현재 `graph_path` 설정을 씁니다.

인수

- `labelname` (text): 확인할 레이블의 이름입니다.
- `graph` (name, 선택): 그래프의 이름입니다. 주지 않으면 `graph_path` 를 씁니다.

반환

- `boolean` : 레이블이 있으면 `true` , 없으면 `false` 입니다.

예제

```
SELECT meta.label_exists('Person', 'social');

 ?column?
----------
 t
```


© 2026 SKAI Worldwide Co., Ltd.

meta.vertex_label_exists

설명

지정한 그래프에 정점 레이블이 있는지 확인합니다. 그래프를 주지 않으면 현재 `graph_path` 설정을 씁니다.

인수

- `labelname` (text): 확인할 정점 레이블의 이름입니다.
- `graph` (name, 선택): 그래프의 이름입니다. 주지 않으면 `graph_path`를 씁니다.

반환

- `boolean`: 정점 레이블이 있으면 `true`, 없으면 `false`입니다.

예제

```
SELECT meta.vertex_label_exists('Person', 'social');

 ?column?
----------
 t
```

meta.edge_label_exists

설명

지정한 그래프에 간선 레이블이 있는지 확인합니다. 그래프를 주지 않으면 현재 `graph_path` 설정을 씁니다.

인수

- `labelname` (text): 확인할 간선 레이블의 이름입니다.
- `graph` (name, 선택): 그래프의 이름입니다. 주지 않으면 `graph_path`를 씁니다.

반환

- `boolean`: 간선 레이블이 있으면 `true`, 없으면 `false`입니다.

예제

```
SELECT meta.edge_label_exists('Follows', 'social');

 ?column?
----------
 t
```

meta.get_child_labels

설명

지정한 부모 레이블을 상속하는 모든 자식 레이블을 반환합니다. 그래프를 주지 않으면 현재 `graph_path` 설정을 씁니다.

인수

- `parent_label` (text): 부모 레이블의 이름입니다.
- `graph` (name, 선택): 그래프의 이름입니다. 주지 않으면 `graph_path`를 씁니다.

반환

- `table(labelname name)`: 자식 레이블 이름의 집합입니다.

© 2026 SKAI Worldwide Co., Ltd.

예제

```
SELECT * FROM meta.get_child_labels('Person', 'social');

 labelname
-----------
 Employee
 Customer
```

meta.get_parent_labels

설명

지정한 자식 레이블이 상속하는 모든 부모 레이블을 반환합니다. 그래프를 주지 않으면 현재 `graph_path` 설정을 씁니다.

인수

- `child_label` (text): 자식 레이블의 이름입니다.
- `graph` (name, 선택): 그래프의 이름입니다. 주지 않으면 `graph_path`를 씁니다.

반환

- `table(labelname name)` : 부모 레이블 이름의 집합입니다.

예제

```
SELECT * FROM meta.get_parent_labels('Employee', 'social');

 labelname
-----------
 Person
```

meta.cypher_type

설명

주어진 값에 대응하는 Cypher 타입을 판정합니다. 표준 타입과 JSON/JSONB, 그리고 수치 구분을 지원합니다.

인수

- `element` (anyelement): Cypher 타입을 판정할 값입니다.

반환

- `text` : 문자열로 된 Cypher 타입입니다. `INTEGER` · `FLOAT` · `STRING` · `BOOLEAN` · `MAP` · `LIST` · `NUMBER` 등이 나올 수 있습니다.

예제

```
SELECT meta.cypher_type(42) AS result;
result
-------
INTEGER

SELECT meta.cypher_type('{"key": "value"}'::jsonb) AS result;
result
-------
MAP
```


© 2026 SKAI Worldwide Co., Ltd.

meta.cypher_types

설명

입력 JSONB 객체의 각 키를 그에 대응하는 Cypher 타입에 대응시킨 JSONB 객체를 반환합니다. 입력이 JSON 객체가 아니면 오류를 냅니다.

인수

- `properties` (jsonb): 키를 Cypher 타입에 대응시킬 JSONB 객체입니다.

반환

- `jsonb` : 각 키가 입력의 키에 대응하고 값이 판정된 Cypher 타입(`INTEGER` · `FLOAT` · `STRING` · `BOOLEAN` · `MAP` · `LIST` · `NULL` · `NUMBER`)인 JSONB 객체입니다.

예제

```
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

설명

주어진 값이 지정한 Cypher 타입과 맞는지 확인합니다. 지원하는 타입은 `NULL` · `INTEGER` · `FLOAT` · `STRING` · `BOOLEAN` · `MAP` · `LIST` · `NUMBER` 입니다.

인수

- `value` (anyelement): 확인할 값입니다.
- `expected_type` (text): 기대하는 Cypher 타입입니다.

반환

- `boolean` : 값이 기대한 타입과 맞으면 `true` , 아니면 `false` 입니다.

예제

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

meta.count_self_loops

설명

지정한 그래프의 셀프 루프(시작 정점과 끝 정점이 같은 간선) 개수를 반환합니다.
관계를 주지 않으면 간선 레이블과 무관하게 모든 간선을 셉니다. 그래프를 주지 않으면 `graph_path` 설정을 씁니다.

인수

- `relationship` (text, 선택): 확인할 관계/간선 레이블의 이름입니다. 기본값은 `ag_edge` 입니다.
- `graph` (text, 선택): 그래프 이름입니다. 기본값은 `graph_path` 설정입니다.

반환

- `bigint` : 셀프 루프 간선의 개수입니다.

예제

```
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)
```

meta.edge_density

설명

그래프의 간선 밀도를 계산합니다. 실제 간선 수를 가능한 최대 간선 수로 나눈 비율입니다.
그래프를 주지 않으면 `graph_path` 설정을 씁니다.

인수

- `graph` (name, 선택): 그래프 이름입니다. 기본값은 `graph_path` 설정입니다.

반환

- `float` : 그래프의 간선 밀도입니다.

예제

```
-- 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;
```


© 2026 SKAI Worldwide Co., Ltd.

meta.estimated_edge_density

설명

정확한 개수 대신 PostgreSQL 통계(`pg_class.reltuples`)를 써서 그래프의 간선 밀도를 추정합니다.
큰 그래프에서는 더 빠르지만 정확하지 않을 수 있습니다. 그래프 이름을 주지 않으면 `graph_path` 를 씁니다.

인수

- `graph` (name, 선택): 그래프 이름입니다. 기본값은 `graph_path` 설정입니다.

반환

- `float` : 그래프의 추정 간선 밀도입니다.

예제

```
-- 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.

릴리스 노트

릴리스마다 무엇이 바뀌었는지, 그리고 돌고 있는 설치본을 어떻게 앞으로 옮기는지입니다.

AgensGraph 2.18

2.18.6.1

2.18.6.1 릴리스는 쿼리·카탈로그·저장 동작을 바꾸지 않습니다. 2.18.6.0과 같이 PostgreSQL 18.6 위에 만들어졌으며, 2.18.6.0 데이터 디렉터리에서 그대로 기동합니다.

- `agens --help`와 `ag_ctl --help`가 버그 신고 주소로 **inquiry@skaiworldwide.com**을, 홈페이지로 **https://skaiworldwide.com/**을 안내합니다. 2.18.6.0은 회사가 더 이상 보유하지 않은 `bitnine.net`을 안내했습니다.

2.18.6.0

2.18.6.0 릴리스는 AgensGraph를 PostgreSQL 18 위로 옮기고 ISO/IEC 39075(GQL) 표준의 상당 부분을 더합니다. 또한 타입 컬럼 속성 승격을 들여와, 자주 쓰이는 `jsonb` 속성이 쿼리를 고치지 않고도 실제로 인덱싱할 수 있는 컬럼이 되게 합니다.

잘 돌던 2.17 애플리케이션을 망가뜨릴 변경이 여럿 있습니다. 업그레이드하기 전에 아래 **2.18로 옮겨 가기**를 읽고, **2.18로 업그레이드하기**의 절차를 따르십시오. 2.17 데이터 디렉터리는 2.18 바이너리로 기동할 수 없습니다.

개요

- **PostgreSQL 18 기반.** 2.17은 PostgreSQL 17.10 위에, 2.18.6.0은 18.6 위에 만들어졌으므로 PostgreSQL 18의 새로운 것이 모두 함께 옵니다.
- **GQL.** 새 문장 `LET` · `NEXT` · `FILTER` · `FINISH` · `INSERT` · `FOR`, `CALL { }` 서브쿼리와 `CALL … YIELD`, `OPTIONAL CALL`, 서브쿼리 값 표현식 `EXISTS { }` · `COUNT { }` · `COLLECT { }` · `ARRAY { }` · `VALUE { }` · `x IN { }`, 독립 절이 된 `ORDER BY` · `SKIP` · `OFFSET` · `LIMIT`, `XOR` 연산자, 그리고 `CURRENT_GRAPH` / `CURRENT_PROPERTY_GRAPH` 입니다. **GQL 적합성**을 참고하십시오.
- **타입 컬럼 속성 승격.** `ALTER VLABEL movie ADD COLUMN year int GENERATED`는 속성에 선언한 타입의, 인덱싱할 수 있는 실제 컬럼을 주고, Cypher의 속성 읽기가 쿼리를 고치지 않고도 그것으로 해석됩니다. **속성을 컬럼으로 승격하기**를 참고하십시오.
- **네이티브 배열에 담기는 그래프 요소.** 정점·간선·경로에 대한 `collect()`가 jsonb 사본이 아니라 `vertex[]` · `edge[]` · `graphpath[]`를 만들므로, 모은 요소를 그대로 쓸 수 있습니다.
- **순회에 대한 플래너 작업.** `ag_graphmeta`에 기반한 스캔 쳐내기, 쓰이지 않는 패턴 끝점 제거, 가변 길이 간선의 추정 개선이 들어 있습니다.
- **크래시 15종 수정.** 그중 몇은 읽기 권한만으로도 닿을 수 있었습니다.

새 기능

GQL 문장

문장	하는 일
`LET`	범위 안의 것을 하나도 버리지 않고 계산한 값을 행에 더합니다.

© 2026 SKAI Worldwide Co., Ltd.

문장	하는 일
`NEXT`	한 쿼리를 다른 쿼리의 결과에 이어 붙입니다.
`FILTER`	행을 걸러 냅니다. `FILTER`와 `FILTER WHERE` 형태가 있습니다.
`FINISH`	아무것도 투영하지 않는 쿼리를 끝냅니다.
`INSERT`	요소를 만듭니다. `CREATE`의 GQL 표기입니다.
`FOR`	리스트를 순회하며, 인덱스는 `WITH OFFSET`으로 받습니다.
`CALL { }`	서브쿼리이며, 가져올 변수를 명시적으로 적습니다.
`CALL … YIELD`	함수를 호출하고 그 출력 컬럼의 이름을 정합니다.
`OPTIONAL CALL`	`CALL`과 같지만, 아무것도 내놓지 않는 호출도 행 하나를 만듭니다.
서브쿼리 표현식	서브쿼리에 대한 `EXISTS { }`와 `VALUE { }` 입니다. `COUNT { }` · `COLLECT { }` · `ARRAY { }` · `x IN { }`도 같은 일을 하지만 GQL이 아니라 AgensGraph 확장입니다.

`ORDER BY` · `SKIP` · `OFFSET` · `LIMIT`이 `RETURN`과 `WITH`의 일부가 아니라 그 자체로 절이 되었으므로 파이프라인 어디에나 놓을 수 있습니다. 쿼리를 끝맺지는 못합니다. 문장은 여전히 `RETURN` · `FINISH`나 쓰기 절로 끝나야 합니다.

그 밖에 새로 생긴 것은 `XOR` 연산자, `NULLIF`, `IS [NOT] UNKNOWN`, `RETURN ALL` / `WITH ALL`, `exists(property)`, `size(value)`, `cardinality()`, `toInteger()`, `toFloat()`, 그리고 Cypher 안에서 쓰는 SQL 값 함수입니다.

이제 형 변환에 키워드 타입을 쓸 수 있습니다. `::int` · `::numeric` · `::varchar` · `::bit` · `::interval` · `::boolean` · `::date` · `::timestamptz` 등입니다. 그중 몇은 전에는 통하는 표기가 아예 없었습니다. `::timestamp`와 `::time`은 여전히 Cypher에서 쓸 수 없습니다. 그 둘은 표현식을 SQL로 감싸서 쓰십시오.

타입 컬럼 속성 승격

속성은 요소마다의 `jsonb` 문서 안에 있으므로, 읽기는 JSON 문서를 훑게 되고 값 자신의 타입에 대한 인덱스를 쓸 수 없습니다. 승격은 속성 하나에 선언한 타입의, 생성되어 저장되는 컬럼을 줍니다:

```
ALTER VLABEL movie ADD COLUMN year int GENERATED;
CREATE PROPERTY INDEX ON movie (year);
```

쿼리는 바뀌지 않습니다. 이제 `MATCH (m:movie) WHERE m.year > 2010`은 `m.year`를 그 컬럼으로 해석하고 인덱스를 쓸 수 있습니다. 승격은 레이블을 만들 때도 되며, 컬럼이 이름이 다른 속성에서 값을 받을 수도 있습니다:

```
CREATE VLABEL doc (words int GENERATED (word_count), score numeric GENERATED);
```

컬럼은 괄호 안에 적은 속성을 담으므로, `words`로 해석되는 읽기는 `d.word_count`입니다. 컬럼 이름이 아니라 속성 키입니다.

확장 타입도 됩니다. 검색에서 중요해지는 것이 바로 이 경우입니다. `pgvector`를 설치했다면 `CREATE VLABEL doc (emb vector(1024) GENERATED)`가 임베딩에, 읽을 때마다 형 변환하는 `jsonb` 배열이 아니라 인덱싱할 수 있는 실제 컬럼을 줍니다.

승격된 키는 더 이상 스키마 없는 값이 아닙니다. 값이 컬럼의 타입에 맞지 않는 쓰기는 중단됩니다. 그 득실은 **속성을 컬럼으로 승격하기**가 다룹니다.

© 2026 SKAI Worldwide Co., Ltd.

그 밖에 더해진 것

- `ALTER VLABEL … ADD COLUMN` / `DROP COLUMN` 입니다.
- 어느 컬럼이 어느 속성을 담는지 기록하는 새 카탈로그 `ag_label_property`, 그리고 스캔 쳐내기의 관문이 되는 새 컬럼 `ag_graph.graphmeta_valid` 입니다. **시스템 카탈로그**를 참고하십시오.
- 새 설정 세 개 — `enable_graph_ddl` · `enable_property_promotion` · `enable_graph_endpoint_elision` 입니다. **서버 설정**을 참고하십시오.
- 이제 `log_statement = mod` 가 Cypher 쓰기를 기록합니다. 전에는 데이터를 바꾸는 문장을 남기라고 한 서버가 그중 어느 것도 남기지 않았습니다. SQL `INSERT` 는 기록되고 그 옆의 Cypher `CREATE` 는 그러지 않았습니다.

2.18로 옮겨 가기

지금 잘 돌고 있는 것을 망가뜨릴 가능성이 가장 큰 변경들입니다. 전체 목록과 절차는 **2.18로 업그레이드하기**에 있습니다.

새 예약어

이제 `xor` · `current_graph` · `current_property_graph` 가 예약어입니다. 그 이름을 가진 식별자· 레이블·변수·별칭은 따옴표로 감싸야 하며, 여기서 따옴표란 SQL에서와 마찬가지로 Cypher에서도 **큰따옴표**입니다 — `MATCH (n:"xor")`. 백틱은 여기서 따옴표가 아니므로 `` `xor` `` 는 구문 오류입니다.

`insert` · `finish` · `let` 도 예약어이지만, 2.16과 2.17에서 통하던 쓰임 두 가지는 일부러 되살렸습니다. 변수로 쓰는 `insert`, 그리고 **패턴 안에서** 레이블로 쓰는 `insert` 나 `finish` 입니다. 그런 레이블을 만들 때는 여전히 따옴표가 필요합니다. `CREATE VLABEL insert;` 는 구문 오류이고 `CREATE VLABEL "insert";` 는 되며, 그러고 나면 따옴표 없는 패턴 표기가 그것을 찾습니다. 어느 자리에서든 여전히 따옴표가 필요한 것은 **변수** 이름으로 쓰는 `let` 이나 `finish`, 레이블로 쓰는 `let`, 그리고 `let` 이나 `finish` 라는 별칭으로 정렬하는 것입니다.

어느 단어가 어느 자리에서 실패하는지의 전체 표는 **키워드**에 있습니다.

`CALL` 은 예약어로 만들지 않았으므로 `call` 은 컬럼·테이블·별칭·레이블·속성 키·변수로 계속 쓸 수 있습니다. 뜻이 달라져 따옴표가 필요한 표기는 정확히 둘입니다. `call(...)` — 이름이 글자 그대로 `call` 인 함수를 인수와 함께 호출하는 것 — 과, 변수 이름이 `call` 이고 레이블 없이 속성 맵만 지닌 정점 패턴 `(call { ... })` 입니다.

키워드를 참고하십시오.

조건은 불리언만 받는다

`WHERE n.count` 와 `FILTER n.name` 은 속성을 읽어 0이 아닌지, 비어 있지 않은지로 행을 골랐습니다. 이제 형 변환 오류를 냅니다:

```
ERROR:  cannot cast "Jack" (jsonb type string) to boolean
```

`WHERE` · `FILTER` · `AND` · `OR` · `NOT` · `XOR` · `CASE WHEN` 과 `dijkstra()` 의 한정자에 적용됩니다. 명시적인 `::boolean` 은 여전히 변환하며, 진짜 불리언과 jsonb null, 없는 키는 영향을 받지 않습니다. 의도한 변경입니다. ISO/IEC 39075는 검색 조건을 불리언 값 표현식으로 정의합니다.

달라진 결과 타입

- **정점·간선·경로에 대한 `collect()`** 는 사본을 담은 jsonb 배열이 아니라 `vertex[]` · `edge[]` · `graphpath[]` 를 반환합니다. 모든 요소에서 속성을 읽는 것이 이제 됩니다. 전에는 조용히 빈 값을 답했습니다. **속성**에 대한 `collect` 는 여전히 jsonb 리스트를 반환하므로 저장된 데이터의 모양은 그대로입니다.
- **정점이나 간선에 대한 `min()` 과 `max()` 는 이제 오류입니다**:

  ```
  ERROR:  function min(vertex) is not unique
  ```

  PostgreSQL 18이 `min(record)` 와 `max(record)` 를 더했고, 그것이 암시적 `vertex → jsonb` 형 변환을 거쳐 `min(jsonb)` / `max(jsonb)` 와 경쟁하게 됐습니다. 명시적으로 변환하거나 — `min(v::jsonb)` — 식별자를 집계하십시오. `min(id(v))` 입니다.
- **`range()`** 는 정수 배열이 아니라 jsonb 배열을 반환합니다.

© 2026 SKAI Worldwide Co., Ltd.

- `toStringList` · `toIntegerList` · `toFloatList` 는 다시 작성했습니다. 각각 오버로드가 둘입니다. SQL에서 실제 배열에 대해 부르면 `text[]` · `bigint[]` · `double precision[]` 을 반환하고, `jsonb` 인 Cypher 리스트에 대해 부르면 `jsonb` 를 반환합니다. 예전 것들은 여러 가지를 조용히 틀리게 답했습니다. bigint가 상위 비트를 잃었고, 숫자가 아닌 문자열이 0으로 변환됐으며, 원소가 하나인 리스트가 빈 채로 돌아왔고, 64자를 넘는 원소가 고정 버퍼를 넘어갔습니다.

달라진 결과 집합

`SKIP` 이나 `LIMIT` 이 붙은 `WITH … WHERE` 는 이제 걸러 낸 뒤에 나눕니다. 전에는 나누기가 먼저 일어나서 `WITH x WHERE x > 1 ORDER BY x LIMIT 2` 가 행 하나를 반환할 수 있었습니다. 이제 순서는 `WHERE` , 그다음 `ORDER BY` , 그다음 `SKIP` / `LIMIT` 입니다. 결과 집합이 달라집니다.

`RETURN <expr> AS d ORDER BY d` 는 jsonb 형태가 아니라 값 자체를 정렬합니다. 유한하지 않은 실수를 빼면 순서가 있는 모든 타입에서 행과 정렬은 그대로입니다. 그 실수는 전에 문자열 사이에 정렬됐습니다. 얻는 것은 인덱스가 그 정렬을 맡을 수 있다는 점입니다.

쓰기 절의 변경이 같은 문장의 뒤쪽 읽기에 보입니다. 전에는 `MATCH (p) SET p.age = 99 WITH p MATCH (q) RETURN q.age` 는 쓰기 이전 값을 반환했고, `CREATE (:Q) WITH 1 MATCH (x:Q) RETURN count(x)` 는 생성을 조용히 버리면서 0을 반환했습니다.

`ALL` · `ANY` · `NONE` · `SINGLE` 이 3값 논리를 따릅니다. 이제 `any([null] WHERE p)` 는 거짓이 아니라 null입니다. 빈 리스트는 공허하게 참인 결과를 그대로 유지합니다.

엄격해진 검사

아래는 모두 예전에는 성공했고, 대부분의 경우 틀린 답을 내놓던 것들입니다:

- `ALTER TABLE` 은 더는 그래프 레이블의 모양을 바꿀 수 없습니다. 컬럼을 더하거나 지우는 것, 상속을 바꾸는 것, 레이블 컬럼의 이름을 바꾸는 것입니다. 레이블 DDL이 같은 일을 하며, `enable_graph_ddl` 이 그 제한을 풉니다. 그러고도 `id` · `start` · `end` · `properties` 의 타입을 바꾸는 것은 거부됩니다.
- 레이블에 대한 SQL `MERGE` 는 이제 `enable_graph_dml` 이 필요합니다. `INSERT` · `UPDATE` · `DELETE` 는 이미 그랬습니다.
- **행 수준 보안이 이제 그래프 쓰기에 실제로 적용됩니다.** `UPDATE` 나 `DELETE` 정책의 `USING` 절이 결코 스캔되지 않는 범위 테이블 항목에 붙어 있어 아무것도 거르지 못했고, `DELETE` 정책이 아예 없는 롤도 행을 지웠습니다. `INSERT` 정책은 엉뚱한 행을 대상으로 검사됐고 간선 생성은 검사된 적이 없어, 행 보안이 걸린 간선 레이블이 모든 삽입을 받아들였습니다. 그래프에 RLS를 기대고 있다면 2.18에서 다시 시험하십시오. 전에는 통과하던 쓰기가 이제 거부됩니다.
- `WITH` 는 같은 이름을 두 번 담을 수 없고, `ORDER BY` 키는 휘발성 항목으로 계산할 수 없으며, 패턴 변수는 이미 범위에 있는 컬럼과 같은 이름을 쓸 수 없고, 그래프 쓰기는 속성 맵을 스칼라로 바꿔칠 수 없으며, `CALL` 서브쿼리는 바깥에 이미 바인딩된 이름을 반환할 수 없습니다.

진단

그래프 요소에 대한 `count()` , 속성 읽기, 그리고 가변 길이 순회 노드에서 `EXPLAIN` 출력이 바뀌었습니다. 가변 길이 순회 노드는 이제 자기가 따라가는 간선 레이블의 이름을 표시합니다. 계획 텍스트를 비교하는 회귀 기준이 있다면 갱신하십시오.

수정

크래시 15종을 고쳤습니다. 그중 몇은 읽기 권한밖에 없는 사용자도 닿을 수 있었고, 그 인스턴스의 모든 세션을 함께 쓰러뜨렸습니다:

- `head('{}'::text[])` 가 계획을 짜는 도중 서버를 죽였습니다.
- `RETURN cardinality(['a','b'])` 가 파싱 시점에 죽었습니다.
- `MATCH ()-[r:e*1..2 = 5]->()` — 가변 길이 간선에 맵이 아닌 조건을 붙인 것입니다.
- `CREATE PROPERTY INDEX ON doc (tags[0])` 입니다.
- 리스트 컴프리헨션 안의 집계입니다.
- 64자를 넘는 리스트 원소가 고정 버퍼를 넘어갔습니다.

© 2026 SKAI Worldwide Co., Ltd.

오류 없이 틀린 답을 내던 문제를 고친 수정이 넷 더 있습니다. 2.17.0에서 `vertex`의 순서와 `edge`의 같음 판정이 틀려서, 정점에 대한 `<` · `<=` · `>` · `>=`와 간선에 대한 `=`가 말하는 그대로를 뜻하지 않았습니다. `min(jsonb)`와 `max(jsonb)`는 아무 원소나 반환했습니다. `DELETE`는 자기가 지운 요소에 대해 빈 값을 반환했고, 그것에 바인딩된 다른 변수도 비웠습니다. 그리고 그래프 요소에 대한 `x IN <list>`가 동일성이 아니라 직렬화 전체를 비교했습니다.

이 릴리스 안에서 PostgreSQL 18 회귀도 하나 고쳤습니다. 속성 읽기에 대한 `DISTINCT` · `GROUP BY` · `ORDER BY` · `min()` · `max()`가 "could not determine which collation to use"로 실패했습니다. 이제 그런 키는 데이터베이스 기본값, 즉 jsonb 자신이 쓰는 것과 같은 콜레이션을 취하므로 결과는 그대로입니다.

이 릴리스에 없는 것

문의가 있었으므로 분명히 적어 둡니다:

- **`CALL { }` 안의 쓰기 절.** 파서가 서브쿼리 본문의 `CREATE` · `SET`과 그 밖의 쓰기 절을 거부합니다. `CALL { } IN TRANSACTIONS`는 없습니다.
- **GQL의 세션과 스키마** — `SESSION_SET` · `USE` — 그리고 타입 있는 그래프와 그래프 타입입니다.
- **`CREATE OR REPLACE CONSTRAINT`.** 파싱은 되지만 그다음 거부됩니다.
- **저장되지 않거나 생성되지 않는 승격 컬럼.** 승격 컬럼은 `GENERATED … STORED`여야 합니다.

2.18로 업그레이드하기

AgensGraph 2.18은 PostgreSQL 18 위에 만들어졌으므로 그리로 옮기는 것은 메이저 버전 업그레이드입니다. 데이터 디렉터리 형식이 바뀌었고, 2.17 데이터 디렉터리는 2.18 바이너리로 기동할 수 없습니다. 경로는 둘입니다. 기존 데이터 파일을 재사용하는 빠른 `pg_upgrade`, 또는 덤프 후 다시 적재하는 것입니다.

어느 쪽이든 하기 전에 **무엇이 바뀌었는지**를 읽고 자기 애플리케이션을 그에 비추어 확인하십시오. 2.17에서 잘 도는 쿼리를 망가뜨릴 변경이 여럿 있고, 업그레이드 도구는 그중 어느 것도 잡아 주지 않습니다.

애플리케이션부터 확인하기

이 목록을 자기 코드베이스에 대고 하나씩 짚어 보십시오. 각 항목은 2.17에서 잘 돌던 코드를 2.18에서 오류나 다른 답으로 바꾸는 변경입니다.

- **`xor` · `current_graph` · `current_property_graph`라는 이름의 식별자** — 이제 예약어입니다. 이름을 바꾸거나 **큰따옴표**로 감싸십시오(`MATCH (n:"xor")`). 여기서 백틱은 따옴표가 아닙니다. 변수 이름으로 쓰는 `let`과 `finish`, 레이블로 쓰는 `let`도 거부되며, 패턴 표기 `MATCH (n:insert)`는 여전히 되지만 `CREATE VLABEL insert;`에는 따옴표가 필요합니다. 어느 단어가 어느 자리에서 실패하는지의 표는 **키워드**를 참고하십시오.
- **`call(...)` 또는 `(call { ... })`** — 이름이 글자 그대로 `call`인 함수, 또는 변수가 `call`이고 레이블 없이 속성 맵을 지닌 정점 패턴입니다. 둘 다 뜻이 달라지므로 이름을 따옴표로 감싸십시오.
- **불리언이 아닌 조건** — `WHERE n.count`, `FILTER n.name`입니다. 전에는 0이 아닌지, 비어 있지 않은지로 골랐지만 이제 형변환 오류를 냅니다. `::boolean`을 붙이거나 명시적으로 비교하십시오.
- **정점·간선·경로에 대한 `collect()`** — 이제 사본을 담은 jsonb 배열이 아니라 네이티브 배열을 반환합니다. jsonb 형태를 파싱하던 클라이언트 코드는 고쳐야 하고, 모은 요소에서 속성을 읽던 코드는 이제 동작하기 시작합니다.
- **정점이나 간선에 대한 `min()` / `max()`** — 이제 모호해져 오류입니다. `min(v::jsonb)`이나 `min(id(v))`를 쓰십시오.
- **`range()` · `toStringList` · `toIntegerList` · `toFloatList`** — 반환 타입이 바뀌었습니다.
- **`SKIP`이나 `LIMIT`이 붙은 `WITH … WHERE`** — 이제 걸러 낸 뒤에 나누므로 결과 집합이 달라집니다.
- **그래프에 대한 행 수준 보안** — 이제 RLS가 그래프 쓰기에 실제로 적용됩니다. 레이블에 정책이 걸려 있다면 다시 시험하십시오. 전에는 빠져나가던 쓰기가 거부됩니다.

© 2026 SKAI Worldwide Co., Ltd.

- **레이블에 대한 `ALTER TABLE`** — 구조를 바꾸는 것은 모두 거부됩니다. 레이블 DDL을 쓰거나 `enable_graph_ddl`을 설정하십시오.
- **`EXPLAIN` 회귀 기준** — 요소에 대한 `count()`, 속성 읽기, 가변 길이 순회 노드에서 계획 텍스트가 바뀌었습니다.
- **양쪽 콜레이션이 다른 외래키** — 둘 중 하나라도 비결정 콜레이션이면 거부됩니다(`42P21`). 2.17은 만들어 주었습니다. 양쪽이 같은 비결정 콜레이션이면 지금도 됩니다. 서로 다른 결정 콜레이션 둘도 됩니다. `NOT VALID`로도 미룰 수 없고, 레이블 열에 건 외래키도 마찬가지로 거부됩니다. 이 항목은 애플리케이션만의 문제가 아닙니다. 2.17 덤프를 복원하면 그 테이블이 **제약 없이** 만들어집니다. 업그레이드 전에 해당 쌍을 찾으십시오 — **비결정 콜레이션이 낀 외래키**를 보십시오.

비결정 콜레이션이 낀 외래키

덤프하거나 업그레이드하기 전에, **2.17** 클러스터의 데이터베이스마다 다음을 실행하십시오.

```
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);
```

나오는 행 하나하나가 2.18이 거부할 제약입니다. 2.17에서 두 열의 콜레이션을 같게 맞춘 다음 덤프하십시오. 조건은 서버가 적용하는 조건 그대로이고 서버가 읽는 카탈로그 열을 그대로 읽으므로, 업그레이드가 거부할 제약보다 더 많이도 더 적게도 고르지 않습니다.

> 경고
>
> 어느 경로도 미리 알려 주지 않습니다. `pg_upgrade --check`는 `*Clusters are compatible*`로 끝나고, 업그레이드는 그 제약을 다시 만드는 자리에서 실패합니다. 덤프·복원은 더 나쁩니다. 복원이 성공했다고 보고하고 테이블은 **외래키 없이** 들어오며, 그 뒤로는 존재하지 않는 행을 참조하는 행도 통과합니다.

경로 1. pg_upgrade 쓰기

`pg_upgrade`는 새 클러스터의 카탈로그를 만든 다음 기존 사용자 데이터 파일을 재사용하므로 데이터를 다시 쓰지 않습니다. 그래프는 평범한 스키마와 테이블로 저장되므로, 그래프 데이터도 관계형 데이터와 똑같이 옮겨 갑니다.

시작하기 전에

- **2.18을 별도의 위치에 설치하십시오.** 2.17 설치본을 덮어쓰지 마십시오. `pg_upgrade`에는 두 바이너리가 모두 필요하고, 2.17을 그대로 두는 것이 되돌아갈 길을 만들어 줍니다.
- **디스크 공간.** 기본적으로 데이터 파일을 복사하므로 두 번째 사본을 둘 자리가 필요합니다. `--link`는 대신 하드 링크를 겁니다. 거의 즉시 끝나고 공간도 거의 들지 않지만, 두 데이터 디렉터리가 한 파일시스템에 있어야 하고 새 서버가 기동한 뒤에는 되돌릴 수 없게 됩니다.
- **2.17 클러스터가 쓰는 외부 모듈을 2.18 설치본에도 똑같이 설치하십시오.** 예를 들어 `pgvector`입니다. AgensGraph 자신의 그래프 기능은 서버에 내장돼 있으므로 다시 만들 그래프 확장은 없습니다.
- **평범한 컬럼을 지닌 레이블이 업그레이드를 막습니다.** `pg_upgrade`는 `pg_dump --binary-upgrade`에 일을 넘기는데, 그것은 `id` · `start` · `end` · `properties` 말고 다른 컬럼을 가진 그래프 레이블을, 물리적 컬럼 배치를 조용히 망가뜨리는 대신 거부합니다. 시작하기 전에 찾아 두십시오:

© 2026 SKAI Worldwide Co., Ltd.

```
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');
```

업그레이드하기 전에 그 컬럼들을 지우거나 데이터를 다른 곳으로 옮기십시오.

- 모든 명령은 데이터 디렉터리를 소유한 운영체제 사용자로 실행하십시오.

아래 명령은 다음 변수를 씁니다:

```
export OLD=/usr/local/agensgraph-2.17
export NEW=/usr/local/agensgraph-2.18
export OLDDATA=$OLD/data
export NEWDATA=$NEW/data
```

1단계. 옛 서버 정지

`pg_upgrade`가 도는 동안 두 클러스터 모두 내려가 있어야 합니다.

```
$OLD/bin/ag_ctl -D $OLDDATA stop
```

2단계. 새 클러스터 초기화

옛 클러스터와 같은 슈퍼유저·인코딩·로케일을 쓰고, 데이터 체크섬 설정도 맞추십시오.

```
$NEW/bin/initdb -D $NEWDATA -U postgres --encoding=UTF8 --no-data-checksums
```

> 경고
>
> **대부분의 업그레이드에서 `--no-data-checksums`는 선택 사항이 아닙니다.** PostgreSQL 18은 데이터 페이지 체크섬을 기본으로 켜고 PostgreSQL 17은 껐습니다. 그러므로 2.17 클러스터를 `-k`로 초기화하지 않았다면 두 클러스터가 어긋나고, `pg_upgrade`는 아무것도 하기 전에 멈춥니다:
>
> ```
> Performing Consistency Checks
> -----------------------------
> Checking cluster versions                                   ok
>
> old cluster does not use data checksums but the new one does
> Failure, exiting
> ```
>
> 초기화하기 전에 어느 쪽인지 확인하십시오:
>
> ```
> $OLD/bin/pg_controldata -D $OLDDATA | grep checksum
> ```
>
> `Data page checksum version: 0`으로 나오면 `--no-data-checksums`를 주십시오. `1`로 나오면 그 플래그를 빼십시오. 체크섬을 켜는 것은 별개의 작업이며 (`pg_checksums` 참고) 할 만한 일이지만, 업그레이드의 일부로 하지 말고 그 뒤에 하십시오.

3단계. 호환성 확인

`--check`는 두 클러스터를 검사하며 아무것도 바꾸지 않습니다.

© 2026 SKAI Worldwide Co., Ltd.

```
$NEW/bin/pg_upgrade \
  -b $OLD/bin -B $NEW/bin \
  -d $OLDDATA -D $NEWDATA \
  -U postgres --check
```

검사가 성공하면 `*Clusters are compatible*` 로 끝납니다.

> 경고
>
> 검사를 통과했다고 업그레이드가 성공한다는 뜻은 아닙니다. 이 검사는 **비결정 콜레이션이 낀 외래키**를 찾지 않으며, 4단계는 그 제약을 다시 만드는 자리에서 실패합니다. 위 질의를 직접 실행하십시오.

데이터베이스 슈퍼유저를 `-U` 로 주십시오. 기본적으로 `pg_upgrade` 는 운영체제 사용자로 접속하는데, 그것은 보통 데이터베이스 롤이 아닙니다.

4단계. 업그레이드 실행

`--check` 만 뺀 같은 명령입니다:

```
$NEW/bin/pg_upgrade \
  -b $OLD/bin -B $NEW/bin \
  -d $OLDDATA -D $NEWDATA \
  -U postgres
```

`Upgrade Complete` 로 끝납니다. 데이터 파일을 복사하는 대신 하드 링크하려면 검사와 업그레이드 양쪽에 `--link` 를 더하십시오.

5단계. 새 서버 기동

```
$NEW/bin/ag_ctl -D $NEWDATA -l $NEW/logfile start
```

새 서버에 붙어 있는지, 그리고 그래프가 넘어왔는지 확인하십시오:

```
SELECT version();
```

```
SET graph_path = <your graph>;
MATCH (n) RETURN count(n);
```

6단계. 업그레이드 뒤에 할 일

이 중 셋은 AgensGraph에만 해당하며, 어느 것도 선택 사항이 아닙니다.

통계를 다시 만드십시오. `pg_upgrade` 는 옵티마이저 통계를 가져오지 않으므로, 다시 만들기 전까지 계획이 나쁩니다:

```
$NEW/bin/vacuumdb -U postgres --all --analyze-in-stages
```

그래프 메타데이터를 다시 수집하십시오. `ag_graphmeta` 가 플래너의 스캔 쳐내기를 이끕니다. 바이너리 업그레이드는 그것을 빈 채로 두고 그래프를 무효로 표시하므로, 그래프마다 한 번씩 다시 만들기 전까지 쳐내기가 꺼져 있습니다:

```
SET graph_path = <your graph>;
SELECT regather_graphmeta();
```

반영됐는지 확인하십시오:

© 2026 SKAI Worldwide Co., Ltd.

```
SELECT graphname, graphmeta_valid FROM pg_catalog.ag_graph;
```

간선 레이블의 통계를 다시 조정하십시오. 2.18은 새 간선 레이블의 `start` · `end` 컬럼에 대한 통계 목표치를 10000에서 1000으로 낮췄습니다. 2.17에서 만든 레이블은 옛 값을 그대로 지녀 간선 조인 때마다 큰 계획 비용 — 큰 레이블에서는 수백 밀리초 — 을 치릅니다. 간선 레이블마다 이렇게 하십시오:

```
ALTER TABLE <graph>.<elabel> ALTER COLUMN start SET STATISTICS 1000;
ALTER TABLE <graph>.<elabel> ALTER COLUMN "end" SET STATISTICS 1000;
ANALYZE <graph>.<elabel>;
```

그런 다음 업그레이드한 클러스터가 잘 도는 것을 확인했으면, `pg_upgrade` 를 실행한 디렉터리에 그것이 써 둔 스크립트로 옛 데이터 파일을 지우십시오:

```
./delete_old_cluster.sh
```

되돌리기

복사 모드에서, 그리고 `delete_old_cluster.sh` 를 실행하기 전이라면 2.17 데이터 디렉터리는 그대로 있습니다. 새 서버를 정지하고 옛 서버를 기동하십시오:

```
$NEW/bin/ag_ctl -D $NEWDATA stop
$OLD/bin/ag_ctl -D $OLDDATA -l $OLD/logfile start
```

`--link` 업그레이드 뒤에는 두 클러스터가 데이터 파일을 공유하므로, 새 서버가 기동한 뒤에는 되돌릴 수 없습니다. 백업에서 복원하십시오.

경로 2. 덤프하고 다시 적재하기

더 느리지만 모든 것을 다시 쓰고, 실행 중에 바이너리 사본을 둘씩 두지 않아도 됩니다.

```
$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
```

> 경고
>
> `psql` 은 오류가 나도 멈추지 않고, 그러고도 종료 코드 0 을 냅니다. 새 서버가 거부하는 것은 — 예를 들어 **비결정 콜레이션이 낀 외래키** — 출력 한 줄로 지나가고 복원은 그것 없이 계속됩니다. 복원을 믿기 전에 로그를 읽으십시오.

복원이 멈추게 하려면 롤과 데이터를 따로 덤프하십시오. 롤 부분은 `ON_ERROR_STOP` 으로 복원할 수 없습니다. `pg_dumpall` 은 이미 새 클러스터에 있는 부트스트랩 슈퍼유저의 `CREATE ROLE` 을 쓰고, `--clean --if-exists` 는 접속 중인 롤의 `DROP ROLE` 로 실패 지점을 옮길 뿐입니다.

```
$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
```


© 2026 SKAI Worldwide Co., Ltd.

알아 둘 것 둘입니다:

- **복원에 `enable_graph_ddl`이 필요할 수 있습니다.** 평범한 컬럼을 지닌 레이블의 덤프는 자기 `ALTER TABLE … ADD COLUMN` 문장 둘레에 `SET enable_graph_ddl = on`을 내보냅니다. 그 설정은 어느 롤이든 쓸 수 있으므로 평범한 사용자도 복원할 수 있습니다. 그 롤에 `ALTER ROLE … SET`으로 강제로 꺼 두지 않았다면 말입니다.
- **위의 그래프 메타데이터와 통계 작업은 여기서도 그대로 적용됩니다.**

다운그레이드는 지원하지 않는다

어떤 그래프든 승격된 컬럼을 쓰고 있으면 2.18 덤프는 2.17에 복원되지 않습니다. 덤프에 2.17 파서가 거부하는 `CREATE VLABEL` 형태가 들어 있고, 평문 스트림에서는 그 문장 뒤의 모든 것이 유실됩니다. 되돌아갈 길이 필요하다면 업그레이드 전에 받아 둔 2.17 백업을 보관하십시오.

`pg_dump`와 `pg_restore`가 AgensGraph 버전을 검사하지 않는다는 점도 유의하십시오. PostgreSQL의 `server_version` 범위 검사만 적용됩니다. 그래서 어긋남은 처음부터 거부되지 않고 복원 도중의 파싱 실패로 드러납니다.

실제 예제

업그레이드 실습은 작은 클러스터에서 실제 명령과 출력으로 전체 절차를 돌려 봅니다.

업그레이드 실습

작은 클러스터에서 2.17을 2.18로 업그레이드한 전 과정이며, 실행한 명령과 그 출력을 함께 담았습니다. 레퍼런스는 2.18로 업그레이드하기이고, 이 페이지는 실제로 해 보면 어떤 모습인지를 보여 줍니다.

여기 클러스터에는 그래프 하나와 평범한 테이블 하나가 있어, 둘 다 넘어오는 것을 볼 수 있습니다:

```
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');
```

2.17에서 그것을 읽으면 이렇습니다:

```
MATCH (n:person)-[r:knows]->(m:person) RETURN n.name, r.since, m.name;
```

```
 name  | since | name
-------+-------+------
 "Ann" | 2019  | "Bo"
```

1. 옛 클러스터의 체크섬 설정 확인

이것이 다음 단계의 플래그 하나를 좌우하며, 틀리면 업그레이드가 시작도 하기 전에 멈춥니다.

```
$OLD/bin/pg_controldata -D $OLDDATA | grep checksum
```


© 2026 SKAI Worldwide Co., Ltd.

```
Data page checksum version:           0
```

0은 체크섬이 꺼져 있다는 뜻이며, PostgreSQL 17의 기본값입니다.

2. 옛 서버 정지

```
$OLD/bin/ag_ctl -D $OLDDATA stop
```

3. 2.18 클러스터 초기화

옛 클러스터가 `0`으로 나왔으므로 새 클러스터는 `--no-data-checksums`로 초기화해야 합니다. PostgreSQL 18은 체크섬을 기본으로 켜므로, 이 플래그를 빼는 것이 다음 절의 실패를 만듭니다.

```
$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
```

이제 두 클러스터가 서로 맞습니다.

맞지 않으면 어떻게 되는가

그 플래그 없이 초기화한 2.18 클러스터를 상대로 `pg_upgrade --check`를 돌린 전체 출력입니다:

```
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok

old cluster does not use data checksums but the new one does
Failure, exiting
```

두 번째 검사에서 멈추고 아무것도 바꾸지 않으므로, 새 클러스터를 다시 초기화하기만 하면 됩니다.

4. 호환성 확인

```
$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
```


© 2026 SKAI Worldwide Co., Ltd.

```
Checking for prepared transactions                              ok
Checking for new cluster tablespace directories                 ok

*Clusters are compatible*
```

5. 업그레이드 실행

```
$NEW/bin/pg_upgrade -b $OLD/bin -B $NEW/bin -d $OLDDATA -D $NEWDATA -U postgres
```

출력의 끝부분입니다:

```
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
----------------
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. 새 서버를 기동하고 무엇이 넘어왔는지 확인

```
$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 ...
```

간선 속성까지 포함한 그래프입니다:

```
SET graph_path = social;
MATCH (n:person)-[r:knows]->(m:person) RETURN n.name, r.since, m.name;
```

```
 name  | since | name
-------+-------+------
 "Ann" | 2019  | "Bo"
```

속성 인덱스입니다:

© 2026 SKAI Worldwide Co., Ltd.

```
\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'))
```

그리고 평범한 테이블입니다:

```
SELECT * FROM audit;
```

```
 id |      note
----+----------------
  1 | before upgrade
```

7. 업그레이드 뒤에 할 일 셋

통계

```
$NEW/bin/vacuumdb -U postgres --all --analyze-in-stages
```

그래프 메타데이터

바이너리 업그레이드는 `ag_graphmeta`를 빈 채로 두고 그래프를 무효로 표시하므로, 플래너의 스캔 쳐내기가 꺼져 있습니다:

```
SELECT graphname, graphmeta_valid FROM pg_catalog.ag_graph;
```

```
 graphname | graphmeta_valid
-----------+-----------------
 social    | f
```

그래프마다 한 번씩 다시 만드십시오:

```
SET graph_path = social;
SELECT regather_graphmeta();
```

```
NOTICE:  regathering graphmeta...
 regather_graphmeta
--------------------
 t
```

```
SELECT graphname, graphmeta_valid FROM pg_catalog.ag_graph;
```


© 2026 SKAI Worldwide Co., Ltd.

```
 graphname | graphmeta_valid
-----------+-----------------
 social    | t
```

간선 레이블 통계

업그레이드는 옛 통계 목표치를 그대로 가져오며, 그것은 2.17이 쓰던 값입니다:

```
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
```

10000은 큰 간선 레이블에서 실제로 계획 시간을 잡아먹을 만큼 높은 값이고, 2.18은 새 간선 레이블의 목표치를 1000으로 정합니다. 업그레이드한 레이블도 그에 맞추십시오:

```
ALTER TABLE social.knows ALTER COLUMN start SET STATISTICS 1000;
ALTER TABLE social.knows ALTER COLUMN "end" SET STATISTICS 1000;
ANALYZE social.knows;
```

8. 옛 클러스터 제거

새 클러스터가 잘 돈다고 확신한 뒤에만 하십시오:

```
./delete_old_cluster.sh
```

그것을 실행하기 전까지 2.17 데이터 디렉터리는 그대로이므로, 새 서버를 정지하고 옛 서버를 기동해 되돌릴 수 있습니다. `--link`를 쓰지 않았다면 말입니다.

AgensGraph 2.17.0

이 AgensGraph 릴리스는 PostgreSQL 17.10과의 상류 호환, Cypher 쿼리 지원 개선, 빌드 인프라 갱신을 비롯한 상당한 개선과 버그 수정을 담고 있습니다.

상류 변경

- **상류 머지:** AgensGraph가 이제 17.10 릴리스까지의 **PostgreSQL 17**과 호환되며, 최신 PostgreSQL 개선과 보안 패치를 담고 있습니다.
- **증분 백업:** 이제 `pg_basebackup`이 증분 백업을 받을 수 있고 나중에 `pg_combinebackup`으로 합칠 수 있어, 정기 베이스 백업의 크기와 시간이 줄어듭니다.
- **MERGE 개선:** 이제 `MERGE`가 (`merge_action()` 함수와 함께) `RETURNING` 절과 `WHEN NOT MATCHED BY SOURCE` 동작을 지원합니다.
- **SQL/JSON `JSON_TABLE`:** 표준을 따르는 JSON 처리를 위해 `JSON_TABLE()`과 SQL/JSON 쿼리 함수 `JSON_EXISTS()` · `JSON_QUERY()` · `JSON_VALUE()`를 더했습니다.

© 2026 SKAI Worldwide Co., Ltd.

- **VACUUM 메모리와 성능:** 죽은 튜플 ID를 담는 새 내부 자료구조가 VACUUM의 메모리 사용을 크게 줄이고 큰 테이블에서의 성능을 개선합니다.
- **쿼리 성능:** `IS [NOT] NULL`의 처리 개선과 순차 스캔의 스트리밍 I/O를 비롯해 플래너와 실행기의 개선이 많습니다.

개선

- **DELETE / DETACH DELETE 성능 최적화 — 최대 약 30배 (#781):**
  - 정점을 지울 때 더는 모든 간선 레이블을 순차로 스캔하지 않으므로, 내부 시험에서 연결이 많은 정점의 삭제가 대략 **30배 빨라졌습니다.**
- **새 설정 매개변수 `enable_graph_dml` (#786):**
  - 그래프 객체(그 아래의 정점·간선 테이블)에 SQL DML 문장(`INSERT` / `UPDATE` / `DELETE`)을 직접 실행할 수 있게 하는 `enable_graph_dml` GUC를 더했습니다. 그래프 저장소에 날 SQL DML을 실행하면 그래프 데이터가 망가질 수 있으므로 기본값은 `off`이고 슈퍼유저만 쓸 수 있습니다.
- **그래프 속성에 대한 pgvector·표현식 속성 인덱스 (#804):**
  - 이제 `CREATE PROPERTY INDEX`가 그래프 속성에 대해 `::vector(n)` 형 변환 위의 pgvector HNSW 인덱스와 `to_tsvector(...)` 위의 GIN 인덱스를 만들고 쓸 수 있습니다. Cypher 안에서의 벡터·전문·하이브리드 검색의 토대입니다(새로 생긴 벡터 / 전문 / 하이브리드 검색 가이드를 참고하십시오).

AI 라이브러리:

이 제품들은 AI 저장소에서 볼 수 있습니다

- **LangChain AgensGraph**
  - `langchain`에서 GraphStore와 VectorStore 연동을 지원하는 Python 플러그인입니다.
- **LlamaIndex AgensGraph**
  - `llamaindex`에서 AgensGraphStore·PropertyGraphIndex·AgensgraphVectorStore 연동을 지원하는 Python 플러그인입니다.
- **AgensGraph LightRag**
  - `LightRAG`로 AgensGraph에 지식 그래프를 저장하고 조회하는 것을 지원하는 Python 플러그인입니다.
- **Cognee Community Graph Adapter - AgensGraph**
  - `Cognee` 프레임워크를 위한 AgensGraph 그래프 데이터베이스 어댑터를 제공하는 Python 플러그인입니다.
- **AgensGraph MCP Server**
  - 데이터베이스와의 상호작용을 제공하고 AgensGraph를 통한 그래프 탐색을 가능하게 하는 Model Context Protocol(MCP) 서버 구현입니다.

버그 수정

- **리스트 컴프리헨션에서 그래프 객체를 쓸 수 없던 문제 (#788)**
  - 그래프 객체와 그것을 다루는 함수 — 예를 들어 `nodes()` · `relationships()` · `startNode()` · `type()` · `label()` — 를 이제 `[n IN nodes(p) | n]`이나 `[r IN relationships(p) WHERE type(r) = 'knows']` 같은 리스트 컴프리헨션 안에서 쓸 수 있습니다. 전에는 그래프 객체가 타입 없는 JSONB 배열로 모여 틀린 결과를 반환했습니다. 이제는 제대로 타입이 있는 배열로 모입니다.

© 2026 SKAI Worldwide Co., Ltd.

- **`IN` 연산자의 `CypherList is expected` 오류 (#789)**
  - `IN` 연산자의 오른쪽이 표현식으로 만들어진 리스트일 때, 예를 들어 `x IN split(s, ',')`이나 `x IN ['1', '2']` 일 때 잘못 나던 `CypherList is expected` 오류를 고쳤습니다.
- **`SET +=` 연산자에서 NULL로 인한 어서션 실패 (#791)**
  - `SET +=` (맵 병합) 연산자에 `NULL`을 써도 더는 어서션 실패가 나지 않습니다. NULL인 경우를 `COALESCE`로 안전하게 처리합니다.
- **`ORDER BY`가 `RETURN` 밖의 변수를 참조하지 못하던 문제 (#805)**
  - 투영에 없는 컬럼으로 `ORDER BY` 하는 Cypher `RETURN` — 예를 들어 `MATCH (p:person) RETURN p.name ORDER BY p.age` — 이 실패하는 대신 이제 동작합니다.

문서:

- **벡터·전문·하이브리드 검색 가이드:**
  - 그래프 속성에 대한 의미(pgvector)·전문·하이브리드 검색을 다루는 개발자 매뉴얼 페이지가 새로 생겼으며, 인덱싱과 GraphRAG 방식도 담았습니다.
- **문서 포털:**
  - 문서 포털이 AgensGraph 2.17 릴리스에 맞춰 갱신됐습니다.

© 2026 SKAI Worldwide Co., Ltd.

부록

GQL 적합성

AgensGraph는 ISO/IEC 39075:2024 Information technology — Database languages — GQL의 일부를 구현합니다. 이 부록은 적합성을 남김없이 선언하는 문서가 아닙니다. 주요 주제를 알맞고 쓸모 있는 만큼 자세히 다룹니다.

아래 표는 표준의 세부 조항을 기준으로 삼습니다. 표준 자신이 구문 조각에 적합성 규칙을 붙일 때 쓰는 단위가 그것이기 때문입니다. 모든 행은 실제로 도는 2.18 서버에서 확인했습니다. 어떤 행이 **Yes**라고 말하는 것은 그 조항이 정의하는 구문이 받아들여졌고 조항이 요구하는 결과를 냈을 때뿐입니다. 조항의 구문 중 일부는 받아들여지고 나머지는 오류를 내면 **Partial**이고, 어느 것도 파싱되지 않으면 **No**입니다. 표준이 같은 생각을 여러 표기로 제시하는 곳에서는 AgensGraph가 어느 쪽을 취하는지를 비고에 적었습니다.

모든 행에 두루 해당하는 것 셋이 있으며, 비고마다 되풀이하지 않습니다.

GQL은 별도의 모드가 아닙니다. GQL 문장과 Cypher 문장은 같은 파서를 지나며 한 파이프라인에서 섞일 수 있으므로, `LET`이 `WITH` 옆에 오고 `INSERT`가 `CREATE` 옆에 옵니다. **Cypher, GQL, SQL**을 참고하십시오.

그래프는 PostgreSQL 스키마이고 레이블은 테이블입니다. 그래서 디렉터리·스키마·그래프 타입의 카탈로그를 설명하는 표준의 부분들에는 GQL이 아니라 PostgreSQL 쪽 대응물이 있습니다. 작업 그래프는 `USE` 절이 아니라 `graph_path` 설정에서 오고, 권한은 PostgreSQL의 권한입니다. **그래프가 어떻게 저장되는가**를 참고하십시오.

오류는 GQLSTATUS가 아니라 다섯 글자의 SQLSTATE를 지닙니다. **오류 코드**를 참고하십시오.

지원하는 기능

아래 세부 조항은 전부 또는 일부가 구현돼 있습니다. **Partial**인 비고는 무엇이 빠졌는지를 가리키므로, 표준 GQL을 옮겨 오는 독자가 무엇을 살펴야 할지 알 수 있습니다.

세부 조항	이름	지원	비고
8.1–8.4	`<start transaction command>`, `<transaction characteristics>`, `<rollback command>`, `<commit command>`	Yes	PostgreSQL 자신의 트랜잭션 문장이 이 조항들에 답합니다. `START TRANSACTION READ ONLY` · `COMMIT` · `ROLLBACK`이 모두 동작합니다.
9.2	`<procedure body>`	Partial	`NEXT`가 한 문장을 앞 문장의 결과에 이어 붙입니다. `NEXT`는 `YIELD` 절을 지닐 수 없고, 9.2가 함께 허용하는 바인딩 변수 정의 블록은 없습니다. 아래 10.1–10.3을 참고하십시오.
11.1	`<graph expression>`	Partial	`CURRENT_GRAPH`와 `CURRENT_PROPERTY_GRAPH`를 Cypher에서도 SQL에서도 받습니다. 키워드만으로 쓰며, `current_graph()`는 구문 오류입니다. 그 밖의 그래프 표현식 형태는 없습니다.
12.4	`<create graph statement>`	Partial	`CREATE GRAPH name`과 `CREATE GRAPH IF NOT EXISTS name`이 동작합니다. 표준이 이름 뒤에 요구하는 그래프 타입(`ANY` · `LIKE` ·그래프 타입 참조)은 받지 않으며, `OR REPLACE` · `PROPERTY GRAPH` · `AS COPY OF`도 마찬가지입니다.

© 2026 SKAI Worldwide Co., Ltd.

세부 조항	이름	지원	비고
12.5	`<drop graph statement>`	Partial	`DROP GRAPH name`과 `DROP GRAPH IF EXISTS name`이 동작하고, 표준에 없는 `CASCADE`도 있습니다. 새 그래프에 이미 `ag_vertex`와 `ag_edge`가 들어 있어 그것이 필요합니다. `DROP PROPERTY GRAPH`는 구문 오류입니다.
13.1	`<linear data-modifying statement>`	Yes	문장 하나가 읽기 절과 쓰기 절을 한 파이프라인에 엮으며, 쓰기 절의 변경이 그 뒤의 읽기 절에 보입니다.
13.2	`<insert statement>`	Yes	`INSERT`는 Cypher `CREATE`와 같은 패턴을 받으며 그것의 동의어입니다.
13.3	`<set statement>`	Partial	속성 대입이 동작하며, 표준에 없는 `+=` 병합 형태도 있습니다. 이 조항이 함께 정의하는 레이블 집합 대입은 없습니다. 정점은 레이블을 정확히 하나 가집니다.
13.4	`<remove statement>`	Partial	`REMOVE n.prop`이 속성을 지웁니다. 같은 까닭으로 `REMOVE n:Label`은 없습니다.
13.5	`<delete statement>`	Yes	`DELETE`와 `DETACH DELETE`입니다. 간선이 남아 있는 정점을 지우는 것은 `DETACH`를 주지 않으면 거부됩니다.
14.2	`<composite query expression>`	Partial	`UNION` · `UNION ALL` · `EXCEPT` · `INTERSECT`가 쿼리 문장 둘을 엮습니다. `OTHERWISE`는 구문 오류입니다.
14.3	`<linear query statement>` and `<simple query statement>`	Yes	쿼리는 아래의 기본 쿼리 문장을 이어 놓은 것이며, 기본 결과 문장으로 끝납니다.
14.4	`<match statement>`	Partial	`MATCH`와 `OPTIONAL MATCH`가 동작합니다. 중괄호 형태 `OPTIONAL { MATCH … MATCH … }`는 구문 오류입니다.
14.5	`<call query statement>`	Partial	두 호출 형태 모두 동작합니다. 각각이 무엇을 요구하는지는 15.2와 15.3을 참고하십시오.
14.6	`<filter statement>`	Yes	`FILTER expr`과 `FILTER WHERE expr` 둘 다 됩니다.
14.7	`<let statement>`	Yes	기존 바인딩에 변수를 더하는 방식이고, 행마다 평가되며, 쿼리 맨 앞에 올 수도 있습니다. `LET` 표현식 안의 집계는 거부됩니다.
14.8	`<for statement>`	Partial	리스트 순회가 동작하며 `WITH OFFSET AS name`이 0부터 시작하는 인덱스를 줍니다. `WITH ORDINALITY`는 구문 오류이고, `FOR`는 바인딩 테이블을 순회하지 못합니다.
14.9	`<order by and page statement>`	Yes	`ORDER BY` · `SKIP` · `OFFSET` · `LIMIT`이 그 자체로 문장이며 파이프라인 어디에나 놓입니다. 표준과 마찬가지로 AgensGraph도 이것들이 쿼리를 끝맺게 하지 않습니다. 14.3이 기본 결과 문장으로 끝날 것을 요구하므로, `ORDER BY`로 끝나는 쿼리는 `ERROR: Cypher query must end with RETURN, FINISH or update clause`를 냅니다.

© 2026 SKAI Worldwide Co., Ltd.

세부 조항	이름	지원	비고
14.10	`<primitive result statement>`	Yes	`RETURN`, `RETURN … ORDER BY …`, 그리고 `FINISH` 입니다.
14.11	`<return statement>`	Partial	`RETURN` · `RETURN DISTINCT` · `RETURN ALL` · `RETURN *` 과 명시적 별칭입니다. `RETURN NO BINDINGS` 는 구문 오류입니다. 그룹화(grouping)는 암시적입니다. 16.15를 참고하십시오.
15.1	`<call procedure statement>` and `<procedure call>`	Yes	`CALL` 과 `OPTIONAL CALL` 이며 두 호출 형태 모두에 적용됩니다.
15.2	`<inline procedure call>`	Partial	`CALL (v) { … }` · `CALL () { … }` · `CALL (*) { … }` · `CALL { WITH v … }` 가 모두 바깥 변수를 올바르게 가져옵니다. 그냥 쓴 `CALL { … }` 은 표준과 다릅니다. 구문 규칙 3은 변수 범위 절이 없으면 들어오는 작업 레코드의 **모든** 필드를 가리키는 것으로 다시 쓰라고 하므로, 표준대로라면 `CALL (*) { … }` 처럼 전부를 가져옵니다. AgensGraph는 대신 아무것도 가져오지 않으면서 오류도 내지 않으므로, 쿼리가 겉보기와 다른 결과를 내게 됩니다. 언제나 가져올 변수 목록을 적으십시오. 서브쿼리 본문의 쓰기 절은 거부되고 `IN TRANSACTIONS` 는 없습니다.
15.3	`<named procedure call>`	Partial	`CALL routine(args) YIELD item [AS alias]` 와 `YIELD *` 가 동작하며, 인수가 이미 범위에 있는 변수를 참조하면 상관 호출이 됩니다. 조항은 `YIELD` 를 선택으로 두지만 여기서는 필수이고, 이 절은 쿼리를 이끌 수도 끝맺을 수도 없습니다. 이름은 17.5가 설명하는 카탈로그의 프로시저가 아니라 PostgreSQL 함수로 해석됩니다. 기본 서버에는 프로시저가 등록돼 있지 않고, 프로시저를 부르면 `<name>() is a procedure` 가 납니다.
16.3	`<graph pattern binding table>`	Partial	그래프 패턴이 바인딩 테이블을 만듭니다. 이 조항이 함께 허용하는 패턴 `YIELD` 절은 구문 오류입니다.
16.4	`<graph pattern>`	Partial	쉼표로 구분한 경로 패턴이 동작합니다. 일치 모드(`DIFFERENT EDGES` · `REPEATABLE ELEMENTS`)와 `KEEP` 절은 구문 오류입니다.
16.5	`<insert graph pattern>`	Yes	`INSERT` 나 `CREATE` 뒤의 패턴은 정점을, 간선을, 또는 이미 바인딩된 정점 사이의 간선을 만들 수 있습니다.
16.7	`<path pattern expression>`	Partial	이어 붙이기가 동작합니다. 경로 다중집합 선택과 경로 패턴 합집합 — `|` 로 구분한 경로 패턴 둘 — 은 구문 오류입니다.
16.8	`<label expression>`	Partial	레이블 이름 하나만 됩니다. 논리합(`:a|b`)·논리곱(`:a&b`)·부정(`:!a`)·와일드카드(`:%`)는 모두 구문 오류입니다. 대신 `WHERE` 에서 `label()` 함수로 레이블을 검사하십시오.
16.9	`<path variable reference>`	Yes	`MATCH p = (…)` 가 경로 전체를 바인딩하고, `nodes()` · `relationships()` · `length()` 가 그것을 읽습니다.
16.10	`<element variable reference>`	Yes	패턴 변수는 그 뒤의 모든 절에서 범위 안에 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

세부 조항	이름	지원	비고
16.11	`<graph pattern quantifier>`	Partial	간선 수량자 `-[r*1..3]->`가 홉 수가 정해지지 않은 순회를 합니다. 괄호로 묶은 경로 패턴에 붙이는 표준의 수량자 `(…)->{1,3}`은 구문 오류이고, `*` 대신 `+`를 쓰는 것도 그렇습니다.
16.13	`<where clause>`	Partial	`MATCH` 뒤, 그리고 `FILTER` · `WITH` ·리스트 컴프리헨션 안의 `WHERE`입니다. 정점 패턴 안의 `WHERE`는 구문 오류입니다. 검색 조건은 불리언이어야 하며 속성 읽기는 변환되지 않습니다.
16.16	`<order by clause>`	Yes	정렬 키 여럿, `ASC` / `DESC`, `NULLS FIRST` / `NULLS LAST`, 표현식을 정렬 키로 쓰는 것, `RETURN` 별칭을 정렬 키로 쓰는 것입니다.
16.17	`<sort specification list>`	Yes	16.16과 같습니다. 정수 정렬 키는 출력 컬럼 번호가 아니라 상수입니다. **Cypher 호환성**을 참고하십시오.
16.18	`<limit clause>`	Yes	`LIMIT`이며 개수로 표현식을 쓰는 것도 됩니다.
16.19	`<offset clause>`	Yes	조항이 정의하는 두 표기 `OFFSET`과 `SKIP` 모두입니다.
19.1	`<search condition>`	Yes	조항이 요구하는 대로, 조건은 불리언만 받고 그 밖의 것은 받지 않습니다.
19.2	`<predicate>`	Partial	아래 네 가지 조건입니다. 19.6–19.13이 정의하는 나머지 아홉은 지원하지 않습니다.
19.3	`<comparison predicate>`	Yes	`=` · `<>` · `<` · `<=` · `>` · `>=`이며 스칼라·그래프 요소·리스트에 적용됩니다.
19.4	`<exists predicate>`	Partial	조항이 정의하는 다섯 형태 중 넷입니다. `EXISTS { pattern }` (`WHERE`를 붙일 수 있음), `EXISTS ( pattern )` (붙일 수 없으며 괄호 안의 `WHERE`는 구문 오류), `EXISTS { MATCH … }`, `EXISTS { <read statement> }`입니다. `EXISTS ( MATCH … )`는 구문 오류이고, `MATCH`가 둘 이상인 일치 문장 블록은 파싱되려면 `RETURN`이 필요하므로 기능 GQ22는 충족하지 못합니다.
19.5	`<null predicate>`	Yes	`IS NULL`과 `IS NOT NULL`입니다.
20.1	`<value expression>`	Yes	연산자·함수 호출· `CASE` ·서브쿼리 표현식·생성자가 자유롭게 엮입니다.
20.2	`<value expression primary>`	Yes	리터럴, 변수, 속성 참조, 괄호로 묶은 표현식입니다.
20.3	`<value specification>`	Partial	`SESSION_USER`와 `CURRENT_USER`입니다. 매개변수는 GQL의 일반 매개변수 선언이 아니라 드라이버가 넘기는 PostgreSQL 매개변수 `$1`입니다.
20.6	`<value query expression>`	Yes	이 조항이 정의하는 전부인 `VALUE { … }`입니다. 본문의 값 하나를 자기 타입으로 돌려주고, 본문이 비면 null입니다. 구문 규칙 10은 본문에 `LIMIT 1`이나 집계를 요구하지만, AgensGraph는 어떤 본문이든 받고 대신 실행 시점에 `more than one row returned by a subquery used as an expression`을 냅니다. `EXISTS { }`는 19.4이고, `COUNT { }` · `COLLECT { }` · `ARRAY { }` · `x IN { }`은 확장입니다. 아래를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

세부 조항	이름	지원	비고
20.7	`<case expression>`	Yes	검색형과 단순형 둘 다이며, `NULLIF` 와 `COALESCE` 도 있습니다.
20.8	`<cast specification>`	Partial	`::` 연산자가 형 변환을 하며 키워드 타입 이름(`::int` · `::numeric` · `::interval`)을 받습니다. `CAST(x AS t)` 는 Cypher에서 구문 오류이고 SQL에서는 됩니다.
20.9	`<aggregate function>`	Partial	`COUNT(*)` , 그리고 일반 집합 함수 `AVG` · `COUNT` · `MAX` · `MIN` · `SUM` · `STDDEV_SAMP` · `STDDEV_POP` 이며 각각 `DISTINCT` 나 `ALL` 을 받습니다. 정점이나 간선에 대한 `MIN` 과 `MAX` 는 모호하므로 형 변환이나 식별자를 주어야 합니다. `COLLECT_LIST` 의 표기는 `collect` 입니다. 이항 집합 함수 `PERCENTILE_CONT` 와 `PERCENTILE_DISC` 는 Cypher에서 닿을 수 없습니다. 둘 다 `WITHIN GROUP` 이 필요한데 그것이 Cypher에서는 구문 오류입니다.
20.11	`<property reference>`	Yes	`n.prop` , 중첩한 `n.a.b` , 첨자를 붙인 `n.list[0]` 입니다.
20.12	`<binding variable reference>`	Yes	패턴· `WITH` · `LET` · `FOR` · `YIELD` 가 바인딩한 이름입니다.
20.13	`<path value expression>`	Partial	`MATCH p = (…)` 가 바인딩한 경로는 `graphpath` 타입의 일급 값입니다.
20.15	`<list value expression>`	Partial	`[a, b, c]` , 첨자, `IN` , 리스트 컴프리헨션입니다. `LIST[…]` 는 구문 오류입니다.
20.17	`<list value constructor>`	Partial	20.15와 같습니다.
20.18	`<record constructor>`	Partial	`{a: 1, b: 2}` 가 맵을 만들고 `.a` 가 필드를 읽습니다. `RECORD { … }` 표기는 구문 오류입니다.
20.19	`<field>`	Yes	맵의 필드 접근 `({a: 1}).a` 입니다.
20.20	`<boolean value expression>`	Yes	`AND` · `OR` · `NOT` · `XOR` 과 `IS [NOT] UNKNOWN` 입니다.
20.21	`<numeric value expression>`	Yes	`+` · `-` · `*` · `/` · `%` · `^` 입니다.
20.22	`<numeric value function>`	Yes	`abs` · `ceil` · `floor` · `round` · `sign` · `sqrt` · `exp` · `ln` · `log` · `log10` · `power` · `mod` · `pi` , 삼각 함수, `degrees` · `radians` · `char_length` · `octet_length` · `rand` 입니다.
20.23	`<string value expression>`	Yes	`||` 로 잇는 것, 그리고 `+` 로 잇는 것입니다.
20.24	`<character string function>`	Partial	`upper` · `lower` · `btrim` · `ltrim` · `rtrim` · `trim` · `substring(s, from, count)` · `left` · `right` · `replace` · `reverse` · `split` 입니다. `NORMALIZE` 와 `SUBSTRING(s FROM a FOR b)` 표기는 Cypher에서 구문 오류입니다.

© 2026 SKAI Worldwide Co., Ltd.

세부 조항	이름	지원	비고
20.26	`<datetime value expression>`	Partial	PostgreSQL의 타입을 통한 날짜·타임스탬프 연산입니다.
20.27	`<datetime value function>`	Partial	`CURRENT_DATE` · `CURRENT_TIME` · `CURRENT_TIMESTAMP` · `LOCALTIME` · `LOCALTIMESTAMP` 입니다. `LOCAL_TIME` 처럼 표준이 쓰는 밑줄 표기는 인식하지 않습니다.
21.1	Names and variables	Partial	따옴표 없는 식별자는 소문자로 바뀌고, 큰따옴표로 감싼 식별자는 대소문자를 지킵니다. 표준의 백틱 식별자는 구문 오류입니다.
21.2	`<literal>`	Partial	16진·8진·2진과 밑줄로 끊은 형태를 포함한 정수, 실수, 문자열, `true` · `false` · `null` , 리스트, 맵입니다. 타입이 붙은 리터럴 — `DATE '…'` · `DURATION '…'` ·바이트 문자열 — 은 구문 오류이며, 값으로 쓴 `UNKNOWN` 도 그렇습니다.
21.3	`<token>` , `<separator>` , and `<identifier>`	Partial	표준이 요구하는 대로 `--` 와 `/* */` 가 주석을 구분합니다. 그래서 openCypher의 익명 화살표 `-->` 와 `--` 를 쓸 수 없습니다. **Cypher 호환성** 을 참고하십시오.

지원하지 않는 기능

아래 세부 조항이 정의하는 구문은 하나도 받아들여지지 않습니다. 같은 결과에 이르는 AgensGraph 쪽 길이 있으면 비고에 적었습니다.

세부 조항	이름	지원	비고
7.1	`<session set command>`	No	`SESSION SET` 은 구문 오류입니다. 작업 그래프는 `SET graph_path = name` 으로 정하며, 그것은 PostgreSQL 설정이므로 PostgreSQL의 세션·트랜잭션 범위를 따릅니다.
7.2	`<session reset command>`	No	`RESET graph_path` 나 `RESET ALL` 을 쓰십시오.
7.3	`<session close command>`	No	접속을 끊으십시오.
7.4	`<session parameter specification>`	No	없습니다.
10.1	`<graph variable definition>`	No	없습니다. 쿼리는 `graph_path` 가 가리키는 그래프 하나를 상대합니다.
10.2	`<binding table variable definition>`	No	`WITH` 나 `LET` 바인딩, 또는 Cypher를 감싼 SQL 공통 테이블 표현식이 중간 결과를 이어 나릅니다.
10.3	`<value variable definition>`	No	14.7이며 지원되는 `LET` 을 쓰십시오.

© 2026 SKAI Worldwide Co., Ltd.

세부 조항	이름	지원	비고
11.2	`<binding table expression>`	No	없습니다.
11.3	`<object expression primary>`	No	없습니다.
12.1	`<linear catalog-modifying statement>`	No	카탈로그 변경은 PostgreSQL DDL 문장이며 쿼리 파이프라인에 엮이지 않습니다.
12.2	`<create schema statement>`	No	그래프가 곧 스키마이며 `CREATE GRAPH`가 그것을 만듭니다. SQL `CREATE SCHEMA`도 있지만 그것으로는 그래프가 되지 않습니다.
12.3	`<drop schema statement>`	No	`DROP GRAPH` 입니다.
12.6	`<create graph type statement>`	No	없습니다. 레이블에는 타입이 없습니다. 컬럼으로 승격한 속성만 그 컬럼의 타입을 얻습니다. **`CREATE VLABEL`**을 참고하십시오.
12.7	`<drop graph type statement>`	No	없습니다.
12.8	`<call catalog-modifying procedure statement>`	No	없습니다.
13.6	`<call data-modifying procedure statement>`	No	`CALL` 본문 안의 쓰기 절은 거부됩니다.
14.1	`<composite query statement>`	No	복합 쿼리 문장이 지니는 선택적 `USE` 접두에 대응하는 것이 없습니다. 복합 쿼리 표현식 자체인 14.2는 지원합니다.
14.12	`<select statement>`	No	`SELECT … FROM MATCH (…)`는 구문 오류입니다. Cypher 서브쿼리에 대한 SQL `SELECT`가 같은 일을 합니다. **SQL과 Cypher**를 참고하십시오.
16.1	`<at schema clause>`	No	문장은 접속해 있는 데이터베이스에서 실행됩니다.
16.2	`<use graph clause>`	No	`SET graph_path = name` 입니다.
16.6	`<path pattern prefix>`	No	경로 모드(`WALK` · `TRAIL` · `SIMPLE` · `ACYCLIC`)도, 경로 탐색 접두(`ALL` · `ANY` · `ANY SHORTEST` · `ALL SHORTEST` · `SHORTEST k` · `SHORTEST k GROUPS`)도 파싱되지 않습니다. 최단 경로 탐색은 `shortestpath()` · `allshortestpaths()` · `dijkstra()`가 맡습니다.
16.12	`<simplified path pattern expression>`	No	`-/ … /->` 형태는 구문 오류입니다.
16.15	`<group by clause>`	No	명시적 `GROUP BY`는 Cypher에서 구문 오류입니다. 그룹화는 암시적이며, `RETURN`이나 `WITH`에서 집계하지 않은 항목이 그룹화 키가 됩니다. SQL에서는 SQL `GROUP BY`를 쓸 수 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

세부 조항	이름	지원	비고
17.1–17.8	`<schema reference>` and the other object references	No	객체는 PostgreSQL의 스키마 한정 이름으로 가리킵니다. 표준의 디렉터리 경로 `/dir/schema/graph` 는 파싱되지 않습니다.
18.1–18.10	`<nested graph type specification>` and the other type elements	No	없습니다.
19.6	`<value type predicate>`	No	`IS TYPED` 와 `IS ::` 는 구문 오류입니다. 형 변환해 비교하거나 `pg_typeof` 를 쓰십시오.
19.7	`<normalized predicate>`	No	없습니다.
19.8	`<directed predicate>`	No	모든 간선은 방향이 있습니다.
19.9	`<labeled predicate>`	No	`IS LABELED` 는 구문 오류이고, 표현식 안의 `n:Label` 도 그렇습니다. `label(n) = 'person'` 이라고 쓰십시오.
19.10	`<source/destination predicate>`	No	`IS SOURCE OF` 와 `IS DESTINATION OF` 는 구문 오류입니다. 식별자를 비교하십시오. `id(a) = id(startNode(r))` 입니다.
19.11	`<all_different predicate>`	No	식별자를 둘씩 비교하십시오.
19.12	`<same predicate>`	No	`id(a) = id(b)` 입니다.
19.13	`<property_exists predicate>`	No	`PROPERTY_EXISTS(n, key)` 는 구문 오류입니다. `exists(n.key)` 라고 쓰십시오.
20.4	`<dynamic parameter specification>`	No	드라이버의 매개변수 자리표시자 `$1` 이 그 자리를 대신합니다.
20.5	`<let value expression>`	No	표현식 안의 `LET … IN` 은 구문 오류입니다. 14.7인 `LET` 문장은 지원합니다.
20.10	`<element_id function>`	No	`element_id()` 는 없습니다. `id()` 가 식별자를 `graphid` 로 반환합니다.
20.14	`<path value constructor>`	No	경로는 생성자가 아니라 패턴에서 나옵니다.
20.16	`<list value function>`	No	`trim(list, n)` 과 `elements()` 는 없습니다. `head` · `last` · `tail` · `size` · `cardinality` 는 있습니다.
20.25	`<byte string function>`	No	PostgreSQL의 `bytea` 함수를 SQL에서 쓸 수 있습니다.
20.28	`<duration value expression>`	No	PostgreSQL의 `interval` 타입이 기간을 다루지만 SQL에서만 됩니다. `INTERVAL '1 day'` 는 Cypher에서 구문 오류입니다.

© 2026 SKAI Worldwide Co., Ltd.

세부 조항	이름	지원	비고
20.29	`<duration value function>`	No	`duration()` 은 없습니다.
21.4	`<GQL terminal character>`	No	렉서는 Cypher를 위해 확장한 PostgreSQL의 것이며, 표준의 문자 집합 규칙을 구현하지 않습니다.
22.1–22.20	Additional common rules	No	이 조항들은 구문이 아니라 표준의 평가 방식을 정의합니다. AgensGraph는 PostgreSQL의 플래너와 실행기로 쿼리를 평가하므로 적힌 그대로 구현하지 않습니다.
23.1	GQLSTATUS	No	오류는 SQLSTATE를 지닙니다. **오류 코드**를 참고하십시오.
23.2	Diagnostic records	No	진단 정보는 PostgreSQL의 오류 보고 필드에서 옵니다.

표준을 넘어선 확장

AgensGraph는 ISO/IEC 39075가 정의하지 않는 것도 상당히 받아들입니다. 그중 일부는 이 제품이 먼저 지원했던 openCypher입니다. 또 일부는 여기서 그래프가 PostgreSQL에 저장되고 양쪽 모두에 손이 닿기 때문에 있는 것입니다.

확장	더하는 것
Cypher 서브쿼리에 대한 SQL 문장, 그리고 레이블 테이블에 대한 Cypher	Cypher 쿼리는 테이블 표현식이고 레이블은 테이블이므로, 문장 하나가 그래프와 관계형 데이터를 조인할 수 있습니다. **SQL과 Cypher**를 참고하십시오.
`LOAD FROM table AS r CREATE (…)`	외부 테이블을 포함한 릴레이션의 행에서 그래프 요소를 만듭니다. **데이터 적재**를 참고하십시오.
레이블 상속	`CREATE VLABEL child INHERITS (parent)` 가 레이블의 계층을 만들고, 부모에 대한 일치가 자식도 함께 반환합니다. `MATCH (n:parent ONLY)` 는 일치를 부모로 제한합니다. **레이블 상속**을 참고하십시오.
`SET n += map`	속성 전체를 바꾸는 대신 맵을 속성에 병합합니다.
`dijkstra()` , `shortestpath()` , `allshortestpaths()`	표준의 경로 탐색 접두를 대신하는, 가중치가 있는·없는 최단 경로 탐색입니다.
승격된 속성	`ALTER VLABEL movie ADD COLUMN year int GENERATED` 가 속성에 타입이 있고 인덱싱할 수 있는 컬럼을 주며, 속성 읽기가 그것으로 해석됩니다. **`ALTER VLABEL`** 을 참고하십시오.
그래프 DDL	`CREATE VLABEL` · `CREATE ELABEL` · `CREATE PROPERTY INDEX` · `CREATE CONSTRAINT` · `REINDEX VLABEL` · `ALTER VLABEL` 입니다. **데이터 정의**를 참고하십시오.
openCypher 절	`UNWIND` , `ON CREATE` · `ON MATCH` 가 붙은 `MERGE` , 그리고 `WITH` 입니다.

© 2026 SKAI Worldwide Co., Ltd.

확장	더하는 것
`COUNT { }`, `COLLECT { }`, `ARRAY { }`, `x IN { }`	표준에 대응 형태가 없는 서브쿼리 값 표현식 넷입니다. 20.9는 `COUNT(*)` · `COUNT(expr)` 와 행에 대한 `COLLECT_LIST` 집계만 정의합니다. `COUNT { }` 와 `COLLECT { }` 는 openCypher에서, `ARRAY { }` 는 SQL의 `ARRAY(subquery)` 에서, `x IN { }` 은 SQL의 `IN (subquery)` 에서 왔습니다. **서브쿼리 값 표현식**을 참고하십시오.
openCypher 연산자와 함수	`STARTS WITH` · `ENDS WITH` · `CONTAINS` · `=~` , 리스트 컴프리헨션, `all` / `any` / `none` / `single` , `keys` · `properties` · `id` · `label` · `labels` · `type` · `startNode` · `endNode` · `nodes` · `relationships` · `length` · `range` · `head` · `last` · `tail` · `toString` · `toInteger` · `toFloat` 입니다.
PostgreSQL의 모든 것	모든 SQL 타입·함수·인덱스 방식·확장·설정을 그래프 쿼리에서 쓸 수 있으며, 벡터 검색과 전문 검색이 그렇게 동작합니다. **벡터 검색**을 참고하십시오.

관련 항목

- Cypher, GQL, SQL — 한 문장 안에서 세 언어가 어떻게 맞물리는가
- Cypher 호환성 — AgensGraph가 다른 Cypher 구현과 다른 곳
- 키워드 — 표준의 새 문장들이 예약한 단어
- AgensGraph 2.18.6.1 릴리스 노트 — 이 릴리스가 더한 GQL 문장

Cypher 호환성

AgensGraph의 Cypher는 다른 구현과 충분히 비슷해서 대부분의 쿼리가 고치지 않고 넘어옵니다. 그러면서도 몇몇은 실패하거나, 더 나쁘게는 다른 답을 내며 성공할 만큼 다릅니다. 이 부록은 그 차이를, 서버가 내는 오류나 돌려주는 결과와 함께 나열합니다. 옮겨 온 쿼리가 이상하게 동작할 때 그 원인을 알아볼 수 있게 하려는 것입니다.

모든 예제는 `graph_path` 를 `moviekb` 로 둔 채 **예제 데이터셋**을 상대로 실행합니다.

아래의 대부분은 구조에 관한 두 사실로 설명됩니다. 정점은 레이블을 정확히 하나 가집니다. 레이블이 테이블이고 행은 테이블 하나에 살기 때문입니다. 그리고 Cypher 쿼리는 PostgreSQL 쿼리이므로, 렉서와 타입 체계와 오류 코드가 PostgreSQL의 것입니다.

존재하지 않는 구문

관계 패턴 하나에 여러 타입

관계 패턴은 타입을 많아야 하나 받습니다:

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

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

가변 길이(`-[r:acted_in|directed*1..2]->`)에서도, `shortestpath()` 안에서도 같은 오류가 돌아옵니다. `-[r:acted_in|:directed]->` 표기는 그냥 구문 오류입니다. 대신 레이블을 지정하지 않은 간선을 일치시킨 뒤 레이블을 검사하십시오:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (p:person)-[r]->(m:movie)
WHERE label(r) IN ['acted_in','directed']
RETURN count(r);
```

```
 count
-------
 46
```

정점 레이블 선택도 같은 식으로 실패합니다. `MATCH (n:person|movie)` 는 `|` 에서 구문 오류이고, GQL 표준이 정의하는 다른 레이블 표현식 연산자 `:person&movie` · `:!person` · `:%` 도 마찬가지입니다.

정점의 레이블 집합

레이블의 집합이란 것이 없고, 정점이 가진 레이블을 바꿀 방법도 없습니다. 아래는 모두 두 번째 콜론에서 구문 오류입니다:

```
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 ":"
```

그 자리를 레이블 상속이 대신합니다. `CREATE VLABEL actor INHERITS (person)` 은 별도의 레이블을 만들고, 부모 레이블과 일치시키면 이 레이블의 행도 함께 반환됩니다. 그래서 `MATCH (n:person)` 은 배우도 찾고, `MATCH (n:person ONLY)` 는 그들을 뺍니다. 정점은 배우로 승격되는 것이 아니라 처음부터 배우로 만들어집니다. **레이블 상속**을 참고하십시오.

FOREACH

```
FOREACH (x IN [1,2] | CREATE (:zz {v:x}));
```

```
ERROR:  syntax error at or near "FOREACH"
```

`UNWIND` 와 GQL의 `FOR` 문장이 둘 다 리스트를 순회합니다:

```
UNWIND [1,2] AS x RETURN x;
```

```
 x
---
 1
 2
```

둘 중 어느 것 뒤에도 쓰기 절이 올 수 있으므로, `UNWIND list AS x CREATE (…)` 가 `FOREACH` 본문이 담는 것을 대신합니다.

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"
```


© 2026 SKAI Worldwide Co., Ltd.

패턴을 일치시키고 일치가 없을 때만 만드는 `MERGE` 를 쓰십시오. 여기서는 간선이 이미 있으므로 `MERGE` 가 하나를 더 만드는 대신 있는 것을 반환합니다:

```
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"
```

가져오기 경로는 테이블을 거칩니다. `COPY` 나 외부 테이블이 CSV를 릴레이션에 넣고, `LOAD FROM` 이 그 행을 그래프 요소로 바꿉니다:

```
CREATE TABLE people_src (name text, age int);
INSERT INTO people_src VALUES ('Ann', 30), ('Bo', 41);
```

```
LOAD FROM people_src AS r CREATE (:person {name: r.name, age: r.age});
```

```
UPDATE 2
```

```
DROP TABLE people_src;
```

외부 테이블 래퍼를 포함한 전체 절차는 **데이터 적재**를 참고하십시오.

익명 화살표 `-->` 와 `--`

`--` 는 SQL에서와 마찬가지로 주석을 시작하므로, 익명 화살표는 문장을 조용히 끝내 버립니다:

```
MATCH (a:movie)-->(b) RETURN count(*);
```

```
ERROR:  Cypher query must end with RETURN, FINISH or update clause
```

`<--` 는 다르게 실패합니다. 주석이 시작되기 전에 `<` 가 먼저 소비되기 때문입니다:

```
MATCH (a:movie)<--(b) RETURN count(*);
```

```
ERROR:  syntax error at end of input
```

대괄호를 쓴 형태로 적으십시오. `-[]->` · `<-[]-` · `-[]-` 가 모두 동작합니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (a:movie)-[]->(b) RETURN count(*);
```

```
 count
-------
 11
```

맵 투영

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

```
ERROR:  syntax error at or near "{"
```

맵을 직접 만드십시오. 맵 생성자는 평범한 표현식입니다:

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

```
             ?column?
-----------------------------------
 {"year": 2009, "title": "Avatar"}
```

`properties(m)`이 맵 전체를 반환하므로 `m {.*}`를 대신합니다.

패턴 컴프리헨션

```
MATCH (m:movie {title:'Avatar'}) RETURN [(m)<-[:acted_in]-(p) | p.name];
```

```
ERROR:  syntax error at or near ":"
```

서브쿼리 표현식이 같은 일을 하며, 언어의 나머지 부분과 더 비슷하게 읽힙니다:

```
MATCH (m:movie {title:'Avatar'})
RETURN COLLECT { MATCH (m)<-[:acted_in]-(p) RETURN p.name };
```

```
        array
---------------------
 ["Sam Worthington"]
```

`reduce()` · `extract()` · `filter()`

셋 다 없습니다.

```
RETURN reduce(acc = 0, x IN [1,2,3] | acc + x);
```

```
ERROR:  variable does not exist
```


© 2026 SKAI Worldwide Co., Ltd.

`extract` 와 `filter` 는 이미 다른 데 쓰이고 있으므로 `extract()` 와 `filter()` 는 대신 구문 오류입니다. `extract` 는 SQL의 `EXTRACT(field FROM source)` 가, `filter` 는 GQL의 `FILTER` 문장이 차지하고 있습니다.

리스트 컴프리헨션이 `extract()` 와 `filter()` 를 대신하며, 세 가지 표기가 모두 동작합니다:

```
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]
```

접기(fold)가 필요하면 Cypher 결과 위에서 SQL로 집계하거나, 그 접기가 합이라면 `sum()` 을 쓰십시오.

패턴 자체를 조건으로 쓰기

패턴은 불리언 표현식이 아닙니다:

```
MATCH (m:movie) WHERE (m)<-[:acted_in]-() RETURN count(m);
```

```
ERROR:  syntax error at or near ":"
```

`exists()` 로, 또는 GQL의 `EXISTS { }` 서브쿼리 표현식으로 감싸십시오:

```
MATCH (m:movie) WHERE exists((m)<-[:acted_in]-()) RETURN count(m);
```

```
 count
-------
 18
```

백틱으로 감싼 식별자

백틱은 따옴표 문자가 아닙니다:

```
MATCH (m:`movie`) RETURN count(m);
```

```
ERROR:  syntax error at or near "`"
```

식별자는 SQL 방식대로 큰따옴표로 감쌉니다. `MATCH (m:"movie")` 입니다. 2.18에서 예약어가 된 단어들에 이것이 중요합니다. 키워드를 참고하십시오.

시간·공간·거리 생성자

`duration()` · `distance()` · `haversin()` 은 없습니다:

```
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.
```


© 2026 SKAI Worldwide Co., Ltd.

`point()` 와 `timestamp()` 는 더 헷갈리게 실패합니다. 각각 PostgreSQL의 이름과 부딪히기 때문입니다.
`point({x:1, y:2})` 는 PostgreSQL의 기하 타입 `point` 를 찾는데, 그 오버로드는 실수 쌍과 다른 기하 타입을 받으므로 호출이 없는 것이 아니라 모호해집니다:

```
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()` 는 구문 오류입니다. `timestamp` 가 타입 이름이기 때문입니다.

`datetime()` 은 있지만 다른 함수이며, 특히 조심해야 할 것입니다. 시간대나 날짜 구성 요소의 목록을 받고 ISO-8601 문자열은 결코 받지 않습니다. 타임스탬프 문자열을 넘기면 그것을 시간대로 읽습니다:

```
RETURN datetime('2020-01-01T00:00:00');
```

```
ERROR:  time zone "2020-01-01T00:00:00" not recognized
```

시간대 이름이나 날짜 구성 요소 아홉 개는 받습니다:

```
RETURN datetime(2020, 1, 1, 0, 0, 0, 0, 0, 'UTC');
```

```
       datetime
-----------------------
 "2020-01-01T00:00:00"
```

문자열에서 타임스탬프를 얻으려면 형 변환하십시오. `'2020-01-01'::date` , `'2020-01-01 00:00'::timestamptz` 입니다.
`CURRENT_DATE` 와 `CURRENT_TIMESTAMP` 는 키워드만으로 동작합니다. 기간은 PostgreSQL의 interval이며 SQL에서 `::interval` 로 닿습니다. 공간 관련 작업은 PostgreSQL 확장을 거치며, 벡터 검색과 같은 방식으로 그래프 쿼리에서 닿습니다. **벡터 검색**을 참고하십시오.

`db.*` 프로시저

`db` 네임스페이스는 없습니다:

```
CALL db.labels();
```

```
ERROR:  schema "db" does not exist
```

같은 물음에는 카탈로그가 답합니다. `ag_label` 이 그래프의 레이블을 나열합니다. **그래프 카탈로그**를 참고하십시오.

PROFILE

```
PROFILE MATCH (m:movie) RETURN count(m);
```

```
ERROR:  syntax error at or near "PROFILE"
```

`EXPLAIN` 이 PostgreSQL의 옵션과 함께 동작합니다. `EXPLAIN (ANALYZE, BUFFERS) MATCH …` 입니다.

© 2026 SKAI Worldwide Co., Ltd.

동작이 다른 것

RETURN이 내놓는 모든 컬럼은 jsonb다

속성 읽기는 `jsonb` 값이고 투영을 거쳐도 그대로입니다. 그래서 문자열은 JSON 따옴표와 함께 찍힙니다:

```
MATCH (m:movie {title:'Avatar'}) RETURN m.title, pg_typeof(m.title);
```

```
  title   | pg_typeof
----------+-----------
 "Avatar" | "jsonb"
```

`::text`는 따옴표를 없애지 않습니다. 따옴표를 포함해 jsonb 값을 그려 낼 뿐입니다. 평범한 스칼라를 얻으려면 Cypher를 감싼 SQL에서 뽑아내십시오:

```
SELECT t #>> '{}' AS title
FROM (MATCH (m:movie) RETURN m.title AS t) q
ORDER BY title LIMIT 1;
```

```
  title
----------
 Avatar
```

SQL과 Cypher를 참고하십시오.

`ORDER BY 2`는 아무것도 정렬하지 않는다

정렬 키의 정수는 출력 컬럼 번호가 아니라 상수입니다. 모든 행을 같은 상수로 정렬하면 순서가 그대로 남고, 아무것도 경고해 주지 않습니다:

```
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
```

키의 이름을 대십시오. `ORDER BY y`는 기대한 대로 정렬합니다. 정수 형태는 SQL에서는 동작하며, 거기서 정수 정렬 키는 그 자리의 출력 컬럼을 뜻합니다. 그러므로 Cypher를 `SELECT … ORDER BY 2`로 감싸도 정렬됩니다.

```
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
```


© 2026 SKAI Worldwide Co., Ltd.

`n.id` 는 식별자가 아니다

`id` 는 평범한 속성 이름입니다. 무엇인가 값을 써 두지 않았다면 없습니다:

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

```
 id | id
----+-----
    | 3.7
```

`id(m)` 이 식별자, 즉 labid `.` locid로 찍히는 `graphid` 를 반환합니다. 그 두 부분이 무엇을 담는지는 제한을 참고하십시오.

그냥 쓴 `CALL { }` 은 다른 물음에 답한다

서브쿼리 본문은 바깥의 변수를 보지 못합니다. 그 이름이 아무 제약 없는 새 변수로 다시 바인딩되므로, 본문은 지금 행이 아니라 그래프 전체를 상대로 실행됩니다:

```
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
```

25는 그래프에 있는 `acted_in` 간선 전체의 개수입니다. 가져올 변수 목록을 적으면 같은 쿼리가 올바릅니다:

```
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
```

> 중요
>
> `CALL` 서브쿼리에는 언제나 가져올 변수 목록을 주십시오. `CALL (v) { … }` · `CALL (*) { … }` · `CALL { WITH v … }` 셋 다 올바르고 서로 일치합니다. 이 차이를 보이는 형태 가운데 오류를 내지 않고 엉뚱한 결과를 내는 것은 그냥 쓴 형태뿐입니다.

`shortestpath` 의 양 끝점은 이름 있는 변수여야 한다

끝점에 패턴을 직접 적는 것은 거부됩니다:

```
MATCH p = shortestpath((:movie {title:'The Matrix'})-[:related_to*..4]->(:movie
{title:'Inception'}))
RETURN length(p);
```


© 2026 SKAI Worldwide Co., Ltd.

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

양 끝을 먼저 바인딩하십시오:

```
MATCH (a:movie {title:'The Matrix'}), (b:movie {title:'Inception'})
MATCH p = shortestpath((a)-[:related_to*..4]->(b))
RETURN length(p);
```

```
 length
--------
 2
```

같은 규칙이 `allshortestpaths()` 와 `dijkstra()` 에도 적용됩니다.

따옴표 없는 레이블은 소문자로 바뀐다

식별자는 SQL의 대소문자 변환 규칙을 따르므로, 대소문자를 섞어 쓴 레이블은 소문자로 찾습니다:

```
MATCH (m:Movie) RETURN count(m);
```

```
 count
-------
 18
```

`Movie` · `MOVIE` · `movie` 는 같은 레이블입니다. 따옴표로 감싼 대소문자 혼용 이름으로 만든 레이블은 쓸 때마다 따옴표로 감싸야 합니다.

조건은 불리언만 받는다

속성을 참·거짓처럼 읽는 것은 비어 있지 않은지 검사하는 대신 형 변환 오류를 냅니다:

```
MATCH (m:movie {year: 1999}) WHERE m.title RETURN count(m);
```

```
ERROR:  cannot cast "The Matrix" (jsonb type string) to boolean
```

메시지는 실행기가 처음 닿은 값을 가리키므로, 여러 행에 걸친 제약 없는 `MATCH` 는 마침 먼저 읽은 값을 알려 줍니다.

이 규칙은 `WHERE` · `FILTER` · `AND` · `OR` · `NOT` · `XOR` · `CASE WHEN` 과 `dijkstra()` 의 조건 인자에 모두 적용됩니다. 뜻하는 검사를 그대로 쓰십시오. `WHERE m.title IS NOT NULL` 이나 `WHERE m.count <> 0` 처럼 말입니다. 아니면 `::boolean` 으로 명시적으로 변환하십시오.

없는 레이블은 오류가 아니다

존재하지 않는 레이블을 가리키는 패턴은 아무것과도 일치하지 않고 행 수 0을 반환합니다:

```
MATCH (n:no_such_label) RETURN count(n);
```

```
 count
-------
 0
```

그래서 레이블 이름의 오타는 빈 결과처럼 보입니다. 기대한 것이 하나도 나오지 않으면 `ag_label` 의 레이블 목록을 확인하십시오.

© 2026 SKAI Worldwide Co., Ltd.

`size()` 와 `length()` 는 같은 함수다

`size()` 는 `length()` 의 다른 표기이며, 어느 쪽을 쓰든 출력 컬럼의 이름은 `length` 입니다:

```
RETURN size([1,2,3]), size('abcd'), length('abcd');
```

```
 length | length | length
--------+--------+--------
 3      | 4      | 4
```

패턴에 대한 `size()` 는 일치 수를 세며, 컬럼 이름은 `count` 입니다:

```
MATCH (m:movie {title:'Avatar'}) RETURN size((m)<-[:acted_in]-());
```

```
 count
-------
 1
```

`labels()` 는 언제나 레이블 하나를 반환한다

`labels()` 는 있고 리스트를 반환하지만, 정점이 레이블을 하나 가지므로 그 리스트의 원소는 정확히 하나입니다:

```
MATCH (n:person) RETURN labels(n) LIMIT 1;
```

```
   labels
------------
 ["person"]
```

`label(n)` 은 그 하나의 레이블을 스칼라로 반환하며, 이쪽이 더 나은 형태입니다. 간선에는 `type(r)` 과 `label(r)` 이 모두 동작합니다.

그래프 요소에 대한 `collect()` 는 네이티브 배열을 반환한다

정점을 모으면 사본의 목록이 아니라 `vertex[]` 가 나오므로, 모은 요소를 그대로 쓸 수 있습니다:

```
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"
```

속성을 모으는 것은 여전히 jsonb 리스트를 반환합니다.

© 2026 SKAI Worldwide Co., Ltd.

간선이 달린 정점은 지워지지 않는다

`DELETE`는 끝점이 없는 간선을 남기는 것을 거부합니다:

```
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.
```

쿼리에는 상대할 그래프가 있어야 한다

기본 그래프란 것이 없습니다. `graph_path`가 설정돼 있지 않으면 모든 그래프 문장이 거부됩니다:

```
RESET graph_path;
MATCH (m:movie) RETURN count(m);
```

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

세션에 설정하거나, `postgresql.conf`에 또는 롤마다 설정하십시오. 설정을 참고하십시오:

```
SET graph_path = moviekb;
```

AgensGraph가 더한 것

이것들은 다른 Cypher 구현에 대응하는 것이 없으므로, 이를 쓰는 쿼리는 밖으로 옮겨 가지 못합니다.

`MATCH (n:label ONLY)`는 일치를 레이블 하나로 제한하며, 그것을 상속하는 레이블을 뺍니다. `ONLY`가 없으면 부모 레이블에 대한 일치가 자식들도 함께 반환합니다.

`SET n += map`은 속성 전체를 바꾸는 대신 맵을 속성에 병합합니다:

```
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 (…)`는 릴레이션의 행에서 그래프 요소를 만들며, 대량 적재가 이렇게 이루어집니다.

`dijkstra((a)-[e:related_to]->(b), e.weight)`는 가중치가 있는 최단 경로를 찾습니다. 가중치를 따르므로 가장 싼 길이 홉 수로도 가장 짧은 길일 필요는 없습니다:

```
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"]
```

Cypher 쿼리는 테이블 표현식이고 레이블은 테이블이므로, 한 문장에서 SQL과 Cypher를 섞을 수 있습니다:

© 2026 SKAI Worldwide Co., Ltd.

```
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
```

관련 항목

- GQL 적합성 — ISO/IEC 39075의 어느 부분을 지원하는가
- 키워드 — 2.18이 예약한 단어와 그로 인해 깨지는 것
- SQL과 Cypher — 두 언어를 넘나드는 방법 전부
- Cypher, GQL, SQL — 세 언어가 어떻게 맞물리는가

키워드

AgensGraph는 PostgreSQL의 키워드 목록을 물려받고 거기에 자기 그래프 문장에 필요한 단어를 더합니다. 키워드의 범주가 그 단어를 여전히 이름으로 쓸 수 있는 자리를 정합니다. **비예약**어는 무엇의 이름으로도 쓸 수 있습니다. **예약**어는 그럴 수 없고 먼저 큰따옴표로 감싸야 합니다.

물려받은 SQL 단어와 그 범주는 PostgreSQL의 Key Words 부록이 나열합니다. 이 페이지는 AgensGraph가 더하고 바꾸는 것만 다룹니다. PostgreSQL이 쓰는 범주 중 타입/함수 이름과 컬럼 이름 둘에는 AgensGraph의 항목이 없습니다. 아래의 모든 단어는 비예약어이거나 완전한 예약어입니다.

식별자는 SQL 방식대로 큰따옴표로 감쌉니다. 백틱은 구문 오류이므로 `` `let` `` 은 회피 표기로 통하지 않습니다. `"let"` 이라고 쓰십시오.

AgensGraph가 더하는 단어

아래 스물네 단어는 AgensGraph의 키워드 목록에 있고 PostgreSQL 18의 목록에는 없습니다. 비고는 그 단어가 속한 절이나 문장을 가리킵니다.

단어	범주	비고
`allshortestpaths`	비예약	경로 함수. 두 정점 사이의 모든 최단 경로입니다.
`assert`	비예약	`CREATE CONSTRAINT … ASSERT` 입니다.
`collect`	비예약	`COLLECT { }` 서브쿼리 표현식입니다.
`contains`	비예약	`CONTAINS` 문자열 조건입니다.

© 2026 SKAI Worldwide Co., Ltd.

단어	범주	비고
`count`	비예약	`COUNT { }` 서브쿼리 표현식입니다.
`current_graph`	**예약**	키워드만으로 쓰는 작업 그래프입니다.
`current_property_graph`	**예약**	`CURRENT_GRAPH`의 동의어입니다.
`dijkstra`	비예약	경로 함수. 가중치가 있는 최단 경로입니다.
`elabel`	비예약	`CREATE ELABEL` · `ALTER ELABEL` · `DROP ELABEL` 입니다.
`ends`	비예약	`ENDS WITH` 문자열 조건입니다.
`finish`	**예약**	아무것도 투영하지 않는 쿼리를 끝내는 `FINISH` 절입니다.
`graph`	비예약	`CREATE GRAPH` · `DROP GRAPH`, 그리고 `graph_path` 설정의 문장들입니다.
`let`	**예약**	`LET` 절입니다.
`optional`	비예약	`OPTIONAL MATCH`와 `OPTIONAL CALL` 입니다.
`property`	비예약	`CREATE PROPERTY INDEX` · `CURRENT_PROPERTY_GRAPH` 입니다.
`remove`	비예약	`REMOVE` 절입니다.
`shortestpath`	비예약	경로 함수. 최단 경로 하나입니다.
`single`	비예약	`single()` 리스트 조건입니다.
`size`	비예약	`size()` 함수입니다.
`starts`	비예약	`STARTS WITH` 문자열 조건입니다.
`unwind`	**예약**	`UNWIND` 절입니다.
`vlabel`	비예약	`CREATE VLABEL` · `ALTER VLABEL` · `DROP VLABEL` 입니다.
`xor`	**예약**	`XOR` 불리언 연산자입니다.
`yield`	비예약	`CALL … YIELD` 입니다.

AgensGraph가 물려받은 단어 넷은 여기서 예약어이고 PostgreSQL에서는 비예약어입니다. 각각이 Cypher 절을 이끌기 때문입니다. `insert` · `match` · `merge` · `return` 입니다.

© 2026 SKAI Worldwide Co., Ltd.

`CALL` 은 일부러 비예약어로 두었으므로 `call` 은 여전히 컬럼·테이블·별칭·레이블·속성 키·변수의 이름이 됩니다. 뜻이 달라지는 표기는 정확히 둘입니다. 렉서가 `CALL` 뒤에 `(` 나 `{` 가 올 때 그것을 특별하게 다루기 때문입니다:

- `call(...)` — 이름이 글자 그대로 `call` 인 함수를 인수와 함께 호출하는 것입니다. 따옴표로 감싸십시오. `"call"(1)` 입니다.
- `(call { ... })` — 변수 이름이 `call` 이고 레이블 없이 속성 맵을 지닌 정점 패턴입니다. 따옴표로 감싸십시오. `("call" { ... })` 입니다.

2.18에서 예약어가 된 단어

2.18에서 예약어이고 2.17에서는 아니었던 단어가 여섯입니다. `insert` · `finish` · `let` · `xor` · `current_graph` · `current_property_graph` 입니다. 단어를 예약하면 그것을 이름으로 쓰던 애플리케이션이 망가질 수 있으므로, 각각을 문법이 허락하는 만큼 좁게 만들었습니다. 그래서 결과가 한결같지 않습니다. 여전히 통하는 자리가 있으며, 아래 표는 일반 규칙이 아니라 실제로 재어 본 결과입니다.

각 칸은 그 자리를 2.18 서버에서 실제로 실행해 본 결과입니다. **된다**는 따옴표 없는 단어가 받아들여졌다는 뜻입니다.

단어	패턴 안의 레이블	`CREATE VLABEL`	변수	`RETURN … AS` 별칭	정렬 키로 쓴 별칭	속성 키
`insert`	된다	안 된다	된다	된다	된다	된다
`finish`	된다	안 된다	안 된다	된다	안 된다	된다
`let`	안 된다	안 된다	안 된다	된다	안 된다	된다
`xor`	안 된다	안 된다	안 된다	된다	안 된다	된다
`current_graph`	안 된다	안 된다	안 된다	된다	된다	된다
`current_property_graph`	안 된다	안 된다	안 된다	된다	된다	된다

그 표에서 두 열이 이상하게 읽히므로, 둘 다 말로 적어 둡니다.

`insert` 와 `finish` 는 레이블로 쓸 때 패턴에서는 되고 레이블 DDL에서는 안 됩니다. `CREATE (:insert)` 와 `MATCH (n:insert)` 는 받아들여집니다. `CREATE VLABEL insert` 는 그렇지 않습니다:

```
CREATE VLABEL insert;
```

```
ERROR:  syntax error at or near "insert"
```

레이블은 따옴표로 감싼 이름으로 만드십시오. `CREATE VLABEL "insert"` 입니다. 그 뒤에도 따옴표 없는 패턴 표기가 그것을 찾습니다. 따옴표 없는 식별자는 소문자로 바뀌기 때문입니다.

별칭은 언제나 되지만 그 별칭으로 정렬하는 것은 늘 되지는 않습니다. `RETURN n.k AS let` 은 받아들여지고 그 뒤의 `ORDER BY let` 은 그렇지 않습니다:

© 2026 SKAI Worldwide Co., Ltd.

```
MATCH (n:t) RETURN n.k AS let ORDER BY let;
```

```
ERROR:  syntax error at or near "let"
```

`current_graph`와 `current_property_graph`가 예외입니다. 이 두 이름은 별칭으로 붙인 뒤 정렬 키로 써도 됩니다.

이름을 큰따옴표로 감싸면 실패하는 칸이 모두 해결됩니다. 정의할 때와 쓸 때 모두 감싸야 합니다:

```
MATCH (n:t) RETURN n.k AS "let" ORDER BY "let";
```

속성 키에는 따옴표가 결코 필요하지 않습니다. 키는 식별자가 아니라 `jsonb` 문서 안의 문자열이므로, `CREATE (:t {let: 1})`과 `RETURN n.let`은 이 모든 것에 영향을 받지 않습니다.

SQL 쪽에서도 같은 여섯 단어는 컬럼 이름으로 쓸 때 따옴표가 필요합니다. `CREATE TABLE t (xor int)`는 구문 오류이고 `CREATE TABLE t ("xor" int)`는 됩니다.

예약어 집합은 그래프 단어보다 넓다

그래프 절을 위해 단어를 예약하는 것은 전체 그림의 일부일 뿐입니다. AgensGraph는 PostgreSQL의 예약어도 물려받습니다. Cypher의 레이블이나 변수도 같은 문법의 식별자이므로, 그중 무엇이든 따옴표 없이는 거부됩니다. `both`가 좋은 예입니다. 그래프와 아무 상관이 없는 PostgreSQL 예약어입니다:

```
CREATE VLABEL both;
```

```
ERROR:  syntax error at or near "both"
```

`all` · `and` · `any` · `array` · `as` · `case` · `cast` · `end` · `from` · `in` · `not` · `null` · `or` · `select` · `table` · `to` · `union` · `when` · `where`와 PostgreSQL 예약어 목록의 나머지도 마찬가지입니다. 레이블· 변수·속성 인덱스의 이름을 고르기 전에 PostgreSQL의 **Key Words** 부록에 비추어 확인하십시오. AgensGraph가 지닌 모든 SQL 단어의 범주가 거기에 있습니다.

> 참고
>
> 키워드 목록은 AgensGraph 소스의 파일 하나이며, 단어마다 범주와 함께 한 줄씩 적혀 있습니다. 어떤 이름이 거부되는데 이 페이지가 그 까닭을 말해 주지 않으면 그것을 보십시오.

관련 항목

- **구문** — 식별자 대소문자 변환을 포함한 어휘 규칙
- **Cypher 호환성** — 옮겨 온 쿼리가 부딪히는 그 밖의 차이
- **2.18로 업그레이드하기** — 업그레이드 전 점검 목록 전부
- **AgensGraph 2.18.6.1 릴리스 노트** — 이 단어들이 쓰이는 절

오류 코드

AgensGraph 서버가 내는 모든 메시지는 PostgreSQL과 똑같이 다섯 글자의 SQLSTATE 코드를 지닙니다. 앞 두 글자가 오류의 **부류**를, 뒤 세 글자가 그 안의 구체적인 상황을 가리킵니다. 그래서 코드를 모르는 애플리케이션도 그 부류를 보고 대응할 수 있습니다.

© 2026 SKAI Worldwide Co., Ltd.

AgensGraph는 자체 오류 코드를 정의하지 않습니다. 오류 코드 목록이 PostgreSQL 18의 것과 바이트 단위로 같으며, 실패한 그래프 문장은 PostgreSQL에 이미 있는 코드 중 하나를 냅니다. PL/pgSQL이 각각에 쓰는 상황 이름을 곁들인 전체 코드 목록은 PostgreSQL의 Error Codes 부록에 있습니다. 여기서는 옮겨 적지 않습니다.

클라이언트에게 이것이 뜻하는 바는, 그래프에만 해당하는 실패를 코드로 알아볼 수 없다는 것입니다. `multiple types for relationship not supported` 와 `Cypher query must end with RETURN, FINISH or update clause` 는 `0A000` 과 `42601` 로 옵니다. 지원하지 않는 SQL 기능과 SQL 구문 오류가 쓰는 것과 같은 코드입니다. 그 둘을 갈라야 한다면 메시지 텍스트로 대조하고, 믿을 것은 부류라고 여기십시오.

코드 읽기

`psql` 에서 `\set VERBOSITY verbose` 를 실행하면 그 뒤의 모든 오류가 자기 SQLSTATE와 그것을 낸 소스 위치를 함께 보여 줍니다:

```
\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
```

`VERBOSITY` 를 `sqlstate` 로 두면 코드만 출력합니다. 드라이버에서는 그 코드가 오류 객체의 필드이며, 대부분의 PostgreSQL 클라이언트에서 `sqlstate` 입니다.

그래프 문장이 흔히 내는 코드

그래프 작업에서 만나게 될 코드들이며, 각각에 실제 메시지를 곁들였습니다. 상황 이름은 PL/pgSQL과 대부분의 드라이버가 쓰는 것입니다.

코드	상황 이름	예를 들어 이런 것이 냅니다
`0A000`	`feature_not_supported`	`multiple types for relationship not supported` — 타입을 둘 이상 적은 관계 패턴입니다.
`22023`	`invalid_parameter_value`	`labid out of range: 65536` — 제한의 범위를 벗어난 `graphid` 입니다.
`3F000`	`invalid_schema_name`	`graph_path is NULL` — 작업할 그래프가 설정돼 있지 않습니다. 그래프가 스키마이므로 부류가 이것입니다.
`42601`	`syntax_error`	`Cypher query must end with RETURN, FINISH or update clause` , 그리고 모든 파싱 실패입니다.
`42703`	`undefined_column`	`variable does not exist` — 범위 안에 없는 이름입니다.
`42712`	`duplicate_alias`	`WITH passes on "t" more than once` , 그리고 그 밖의 이름 충돌 거부입니다.
`42846`	`cannot_coerce`	`cannot cast "…" (jsonb type string) to boolean` — 조건에 불리언이 아닌 것을 준 경우입니다.

© 2026 SKAI Worldwide Co., Ltd.

코드	상황 이름	예를 들어 이런 것이 냅니다
`42P07`	`duplicate_table`	`label "movie" already exists` — 레이블이 테이블이므로 레이블 중복은 테이블 중복입니다.
`54000`	`program_limit_exceeded`	`no more new labels are available` — 그래프의 레이블 id를 다 썼습니다.
`55000`	`object_not_in_prerequisite_state`	`Vertex with connected edges cannot be removed` — 끝점이 없는 간선을 남기게 될 `DELETE` 입니다.

그중 세 행이 예상해야 할 방식을 보여 줍니다. 그래프는 스키마이고, 레이블은 테이블이며, 속성은 `jsonb` 값입니다. 그래서 그래프의 오류는 그것이 실제로 관계된 PostgreSQL 객체의 부류로 보고됩니다.

관련 항목

- Cypher 호환성 — 다른 Cypher에서 옮겨 온 쿼리가 부딪히는 오류와 그 문구
- 제한 — `22023` 과 `54000` 을 내는 두 제한
- 용어집 — 이 메시지들이 쓰는 용어

제한

AgensGraph의 제한 중 둘은 모든 정점과 간선에 주는 식별자에서 나옵니다. 나머지는 PostgreSQL의 것입니다. 그래프는 스키마이고 레이블은 테이블이기 때문입니다.

그래프의 제한

모든 정점과 간선은 `graphid` 타입의 식별자를 지니며, 그 배치가 그래프가 닿을 수 있는 두 제한을 정합니다. `graphid` 는 둘로 나뉜 64비트 부호 없는 정수입니다. 상위 16비트가 **레이블 id**이고, 하위 48비트가 그 레이블 안에서 행의 일련번호인 **로컬 id**입니다. 그래서 식별자는 숫자 두 개로 찍힙니다. `movie[3.7]` 은 레이블 3의 7번 행입니다.

제한	값	유래
그래프당 레이블 수	65,535	16비트 레이블 id입니다. 0은 무효 레이블 id로 예약돼 있어 쓸 수 있는 범위는 1부터 65,535까지입니다.
레이블당 행 수	281,474,976,710,655	48비트 로컬 id, 즉 2^{48} − 1입니다.
레이블 이름 길이	63바이트	PostgreSQL의 식별자 길이입니다.

그래프 자체에는 따로 제한이 없습니다. 그래프는 스키마이고 PostgreSQL은 데이터베이스당 스키마 수에 제한을 두지 않습니다. 다만 그래프가 많으면 릴레이션을 소모합니다. 그래프마다 자기 시퀀스 하나와 루트 레이블 둘을 만들기 때문입니다.

`graphid` 의 두 부분은 모두 범위 검사를 받습니다. 어느 쪽이든 범위를 벗어나면 생성자가 거부합니다:

```
SELECT graphid(65536, 1);
```


© 2026 SKAI Worldwide Co., Ltd.

```
ERROR:  labid out of range: 65536
```

```
SELECT graphid(1, 281474976710656);
```

```
ERROR:  locid out of range: 281474976710656
```

`graphid` 리터럴을 파싱할 때도 문제가 된 값만 빠진 같은 메시지 둘이 돌아옵니다. `SELECT '65536.1'::graphid`는 `labid out of range`를 냅니다.

이 번호를 나눠 주는 계수기는 평범한 시퀀스이므로 들여다볼 수 있습니다. 그래프마다 레이블 id를 위한 `ag_label_seq`가 하나씩, 레이블마다 로컬 id를 위한 `<label>_id_seq`가 하나씩 있습니다:

```
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
```

`ag_label_seq`는 순환하고 레이블 생성은 이미 쓰이는 번호를 건너뛰므로, 레이블을 지우면 그 id가 다시 쓸 수 있게 풀립니다. 남은 id가 없으면 레이블 생성이 실패합니다:

```
ERROR:  no more new labels are available
```

새 그래프마다 레이블 둘이 함께 생깁니다. 레이블 id 1인 `ag_vertex`와 id 2인 `ag_edge`입니다. 모든 정점·간선 레이블이 상속하는 두 뿌리입니다. 그래서 직접 만드는 레이블에는 65,533개의 id를 쓸 수 있습니다.

63바이트를 넘는 레이블 이름은 잘린다

식별자는 63바이트입니다. 그보다 긴 레이블 이름은 notice와 함께 63바이트로 잘립니다. 그러므로 **앞 63바이트가 같은 레이블 이름 둘은 한 레이블입니다.** 두 번째 `CREATE VLABEL`은 첫 번째 레이블과 이름이 겹쳐 실패합니다:

```
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
```

패턴에서 이것이 발목을 잡습니다. 패턴은 찾지 못한 레이블을 만들고 아무것도 실패하지 않기 때문입니다. 위의 두 이름에 쓰면 두 행이 같은 레이블에 들어갑니다:

```
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;
```


© 2026 SKAI Worldwide Co., Ltd.

```
                                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"
```

레이블 이름은 63바이트 안에 두고, 앞 63바이트가 서로 다르게 하십시오. 제한은 글자 수가 아니라 바이트 수이므로, 여러 바이트를 쓰는 문자로 된 이름은 더 빨리 그 제한에 닿습니다.

그래프가 부딪히는 PostgreSQL의 제한

레이블은 테이블이므로 그래프도 데이터베이스의 나머지와 같은 제한을 받습니다. 아래는 그래프 작업이 닿는 것들입니다. 전체 목록은 PostgreSQL의 **Limits** 부록에 있습니다.

제한	값	그래프에 뜻하는 것
릴레이션 크기	32 TB	기본 8 kB 블록 크기에서 레이블 하나의 저장 공간입니다. 이를 넘는 레이블은 파티셔닝하거나 자식 레이블로 나눠야 합니다.
필드 크기	1 GB	속성이 `jsonb` 필드 하나이므로, 정점이나 간선 하나의 속성 맵 전체입니다. 긴 값은 먼저 압축되고, 그래도 크면 행 밖(TOAST)에 저장됩니다.
릴레이션당 컬럼 수	1600	속성을 컬럼으로 승격할 때에만 닿습니다. 정점 레이블은 컬럼 둘로, 간선 레이블은 넷으로 시작합니다.
식별자 길이	63바이트	레이블 이름·그래프 이름·속성 인덱스 이름·변수 이름입니다. 속성 키는 식별자가 아니므로 여기에 매이지 않습니다.
테이블당 행 수	릴레이션 크기에 묶입니다	48비트 로컬 id가 훨씬 크므로 저장 공간 제한에 먼저 닿습니다.
테이블당 인덱스 수	제한 없음	레이블 하나에 대한 속성 인덱스입니다.

참고

1 GB 필드 크기 말고는 요소 하나가 속성을 몇 개 지닐 수 있는지도, 속성이 얼마나 깊이 중첩될 수 있는지도 제한하지 않습니다. 다만 속성 읽기에는 맵 크기에 따라 커지는 비용이 있습니다. 키 하나에 닿으려고 맵 전체를 가져오기 때문입니다. 그 비용을 벗어나는 길은 `ALTER VLABEL`로 그 키를 컬럼으로 승격하는 것입니다.

관련 항목

- **그래프가 어떻게 저장되는가** — 이 제한들이 설명하는 저장 배치
- **레이블 상속** — 큰 레이블 하나를 여럿으로 나누기
- **데이터 정의** — 레이블과 인덱스를 만드는 문장

© 2026 SKAI Worldwide Co., Ltd.

용어집

이 매뉴얼이 쓰는 용어이며, 각 용어를 제대로 다루는 페이지로 가는 링크를 함께 두었습니다. 여럿이 서로 나란히 놓여야만 뜻이 통하므로, 용어를 가나다순이 아니라 무엇을 설명하는지에 따라 묶었습니다.

모델

속성 그래프(property graph)

데이터가 간선으로 이어진 정점의 집합이고, 그 둘 모두 이름 붙은 값을 지니는 데이터 모델입니다. AgensGraph는 속성 그래프 하나와 관계형 테이블 한 벌을 같은 데이터베이스에 담으며, 문장 하나가 둘 다 읽을 수 있습니다. **속성 그래프 모델**을 참고하십시오.

그래프(graph)

이름이 붙은 속성 그래프입니다. 레이블 집합을 담는 그릇이자 `graph_path`가 고르는 단위입니다. 그래프는 `CREATE GRAPH`로 만드는 PostgreSQL 스키마입니다. **그래프가 어떻게 저장되는가**를 참고하십시오.

정점(vertex)

그래프 안의 개체 하나입니다. 사람, 영화, 계정 같은 것입니다. 정점은 식별자 하나와 정확히 하나의 레이블, 그리고 속성의 집합을 가집니다. `label[graphid] {properties}` 형태로 찍히며, 예를 들어 `movie[3.7]{"title": "Avatar"}` 입니다. **속성 그래프 모델**을 참고하십시오.

간선(edge)

정점 둘 사이의 방향 있는 연결 하나이며, 자기 레이블과 속성을 가집니다. 간선은 양 끝점 없이는 존재할 수 없습니다. 모든 간선은 방향이 있고, 패턴은 어느 방향으로든 그것을 지날 수 있습니다. **속성 그래프 모델**을 참고하십시오.

속성(property)

정점이나 간선에 붙은, 이름이 있는 값 하나입니다. `title` · `year` · `weight` 같은 것입니다. 속성에는 스키마가 없습니다. 레이블이 같은 정점 둘이 같은 키를 지녀야 할 이유가 없습니다. 속성들은 요소마다 하나씩 있는 `jsonb` 문서 안에 함께 삽니다. 속성 키는 식별자가 아니므로 키워드 예약이나 대소문자 변환의 영향을 받지 않습니다. **그래프가 어떻게 저장되는가**를 참고하십시오.

레이블(label)

정점이나 간선을 분류하는 이름이자 저장의 단위입니다. 레이블은 PostgreSQL 테이블입니다. 정점은 레이블을 정확히 하나 가지며, 정점을 만든 뒤에는 바꿀 수 없습니다. 레이블의 집합이란 것은 없습니다. **그래프가 어떻게 저장되는가**를 참고하십시오.

vlabel

정점 레이블이며 `CREATE VLABEL`로 만드는 것입니다. 그 테이블은 `(id graphid, properties jsonb)`이고 그래프의 `ag_vertex`를 상속합니다. **데이터 정의**를 참고하십시오.

elabel

간선 레이블이며 `CREATE ELABEL`로 만드는 것입니다. 그 테이블은 vlabel의 컬럼에 `start`와 `end`를 더하고 그래프의 `ag_edge`를 상속합니다. **데이터 정의**를 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

레이블 상속(label inheritance)

레이블은 다른 레이블을 상속할 수 있으며, 그래서 부모에 대한 일치가 모든 자식의 행도 함께 반환합니다. 레이블 집합 대신 개체 하나가 여러 역할을 맡는 방법이 이것입니다. `MATCH (n:parent ONLY)`가 그것을 끕니다. **레이블 상속**을 참고하십시오.

식별과 지칭

graphid

정점이나 간선의 식별자이자 일급 타입입니다. 레이블 id `.` 로컬 id라는 숫자 둘로 찍히는 64비트 값입니다. 그래서 `3.7`은 레이블 3의 7번 행입니다. 그것을 반환하는 것은 `id(n)` 입니다. `n.id`는 아닙니다. `id`는 평범한 속성 이름일 뿐이기 때문입니다. **제한**을 참고하십시오.

graph_path

문장이 상대할 그래프를 가리키는 설정입니다. 기본값은 없습니다. `graph_path`가 설정돼 있지 않으면 모든 그래프 문장이 `ERROR: graph_path is NULL`로 거부됩니다. 세션에는 `SET graph_path = name`으로, 그 밖에 롤마다 또는 클러스터 전체에 설정할 수 있습니다. **설정**을 참고하십시오.

쿼리

경로(path)

정점과 간선이 번갈아 이어진 것이며, 생성자가 아니라 패턴이 만들어 냅니다. `MATCH p = (a)-[e]->(b)`가 그것을 `graphpath` 타입의 값으로 바인딩하고, 그다음 `nodes()` · `relationships()` · `length()`가 그것을 읽습니다. **그래프 읽기**를 참고하십시오.

순회(traversal)

한 정점에서 간선을 따라 다른 정점에 닿는 것입니다. 그래프 쿼리는 키로 조인하는 대신 이것을 합니다. 깊이를 모르는 순회는 수량이 붙은 간선 `-[r*1..3]->`으로 씁니다. **그래프 읽기**를 참고하십시오.

속성 인덱스(property index)

속성 표현식에 대한 인덱스이며 `CREATE PROPERTY INDEX ON label (key)`로 만듭니다. 키가 `properties` 문서에 대한 표현식인 PostgreSQL 인덱스이고, 평범한 `CREATE INDEX`로는 그 키를 적을 수 없습니다. **그래프가 어떻게 저장되는가**를 참고하십시오.

승격된 컬럼(promoted column)

선언한 타입의 실제 컬럼을 받은 속성이며, `properties` 문서에서 생성되어 저장됩니다. 속성 읽기가 그 컬럼으로 해석되므로 그것에 대한 평범한 인덱스가 쿼리를 처리하고 값도 자기 타입을 지킵니다. 예제 데이터셋의 `movie.year`가 그런 것입니다. **`CREATE VLABEL`**과 **`ALTER VLABEL`**을 참고하십시오.

© 2026 SKAI Worldwide Co., Ltd.

언어

Cypher

AgensGraph가 처음부터 받아 온 그래프 쿼리 언어입니다. 선언형이고, 패턴을 ASCII로 그려 일치시킵니다. AgensGraph의 방언은 다른 구현과 비슷하지만 같지는 않습니다. **Cypher, GQL, SQL과 Cypher 호환성**을 참고하십시오.

GQL

그래프 쿼리 언어의 국제 표준인 ISO/IEC 39075:2024입니다. 2.18 릴리스가 그 상당 부분을 구현했으며, `LET` · `NEXT` · `FILTER` · `FINISH` · `INSERT` · `FOR` 문장이 거기에 들어갑니다. GQL은 별도의 모드가 아닙니다. GQL의 문장과 Cypher의 문장은 한 파이프라인에서 자유롭게 섞입니다. **GQL 적합성**을 참고하십시오.

관련 항목

- **속성 그래프 모델** — 예제를 곁들인 모델 설명
- **그래프가 어떻게 저장되는가** — 위의 용어들이 어떻게 저장되는가
- **예제 데이터셋** — 모든 예제가 쓰는 그래프

© 2026 SKAI Worldwide Co., Ltd.