

# 튜토리얼: 세 언어, 하나의 그래프 — OWL/RDF · Cypher · TypeQL

## 이 문서가 답하는 질문

- OWL 온톨로지를 Turtle로 넣으면 무엇이 타입 계층이 되고, 매핑되지 않은 트리플은 어디로 가는가?
- 같은 그래프를 Cypher와 TypeQL로 읽고 쓸 때, 한 언어가 쓴 것을 다른 언어가 **실제로** 읽는가? 어디까지인가?
- "라벨 하나를 물으면 계층 전체가 답한다"는 세 언어 모두에서 성립하는가?
- 그래프를 다시 RDF로 내보낼 수 있는가? 무엇이 그대로 돌아오고 무엇이 바뀌는가?

처음이라면 데모 영상부터 보세요. Studio에서 OWL로 들어온 온톨로지를 Cypher와 TypeQL이 함께 읽고 쓰는 과정을 실제로 실행한 화면이고, 설명은 자막으로 들어 있습니다. 이 문서는 PDF 와 Word 문서 로도 받을 수 있습니다.

한빛물류에는 차량 데이터를 다루는 팀이 셋 있고, 각자 쓰는 언어가 다릅니다.

| 팀     | 하는 일                              | 언어        |
|-------|-----------------------------------|-----------|
| 표준 담당 | 차량 분류 온톨로지를 관리하고 Turtle 파일로 배포합니다 | OWL / RDF |
| 배차 앱  | 기사와 차량을 잇고 조회합니다. Neo4j 드라이버를 씁니다 | Cypher    |
| 정비팀   | TypeDB를 쓰던 팀입니다. 점검 기록을 남깁니다      | TypeQL    |

이 문서는 세 팀이 **그래프 하나**를 같이 쓰는 과정을 따라갑니다. 저장소는 하나이고, 세 언어는 그 위의 어댑터입니다. 그래서 서로의 데이터가 보이지만, 언어마다 데이터를 놓는 자리가 달라서 **보이는 방식**에는 규칙이 있습니다. 되는 것과 안 되는 것을 모두 실행 결과로 적었습니다. 처음 들어오는 데이터는 차량 3대, 기사 2명입니다.

| 차량 | 분류                        | 번호판     | 적재량(kg) | 기사  |
|----|---------------------------|---------|---------|-----|
| v1 | Truck                     | 81바1001 | 5000    | 김민수 |
| v2 | ElectricTruck (Truck의 하위) | 81바1002 | 3500    | 이서연 |
| v3 | Van                       | 81바1003 | 1200    | —   |

```
SELECT og_create_graph('fleet');
```

SQL

## 1단계. 온톨로지가 Turtle로 도착한다

표준 담당이 보낸 파일을 `og_load_rdf` 에 그대로 넘깁니다. 클래스 정의(TBox)와 인스턴스(ABox)가 한 문서에 있습니다.

SQL

```
SELECT jsonb_pretty(og_load_rdf('fleet', $rdf$
@prefix ex:    <http://example.org/fleet#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .

ex:Vehicle      a owl:Class .
ex:Truck         a owl:Class ; rdfs:subClassOf ex:Vehicle .
ex:Van           a owl:Class ; rdfs:subClassOf ex:Vehicle .
ex:ElectricTruck a owl:Class ; rdfs:subClassOf ex:Truck .
ex:Driver        a owl:Class .
ex:drives        a owl:ObjectProperty ; rdfs:domain ex:Driver ; rdfs:range ex:Vehicle .

ex:v1 a ex:Truck      ; ex:plate "81바1001" ; ex:payload "5000"^^xsd:integer .
ex:v2 a ex:ElectricTruck ; ex:plate "81바1002" ; ex:payload "3500"^^xsd:integer .
ex:v3 a ex:Van         ; ex:plate "81바1003" ; ex:payload "1200"^^xsd:integer .
ex:d1 a ex:Driver ; ex:name "김민수" .
ex:d2 a ex:Driver ; ex:name "이서연" .
ex:d1 ex:drives ex:v1 .
ex:d2 ex:drives ex:v2 .
ex:d2 ex:memberOf ex:union7 .
$rdf$)) AS report;
```

결과

report

```
{
  "note": "unmapped triples were preserved verbatim - see og_mapping_report(graph)",
  "facts": 10,
  "graph": "fleet",
  "classes": 5,
  "unmapped": 1,
  "instances": 5,
  "triples_read": 27,
  "inference_rules": 0,
  "subclass_axioms": 3,
  "object_properties": 1,
  "datatype_properties": 0
}
(1 row)
```

검산: 클래스 5개, `subClassOf` 3개, 객체 프로퍼티 1개(`drives`), 인스턴스 5개(차량 3 + 기사 2). `facts` 10은 리터럴 8개(번호판 3, 적재량 3, 이름 2)와 `drives` 2개입니다. `unmapped` 1은 마지막 줄입니다.

OWL 클래스는 별도의 트리플 저장소에 들어가지 않고, 이 데이터베이스의 **타입 계층**이 되었습니다.

SQL

```
SELECT name, kind, depth, parents, iri FROM og_type_view WHERE graph = 'fleet' ORDER BY lft;
```

결과

| name          | kind     | depth | parents   | iri                                    |
|---------------|----------|-------|-----------|----------------------------------------|
| Vehicle       | entity   | 0     | {}        | http://example.org/fleet#Vehicle       |
| Truck         | entity   | 1     | {Vehicle} | http://example.org/fleet#Truck         |
| ElectricTruck | entity   | 2     | {Truck}   | http://example.org/fleet#ElectricTruck |
| Van           | entity   | 1     | {Vehicle} | http://example.org/fleet#Van           |
| Driver        | entity   | 0     | {}        | http://example.org/fleet#Driver        |
| drives        | relation | 0     | {}        | http://example.org/fleet#drives        |

(6 rows)

`rdfs:subClassOf` 는 상속(`parents`)이 되고, `owl:ObjectProperty` 는 관계 타입이 되며, IRI는 타입에 남습니다. 마지막 줄 `ex:d2 ex:memberOf ex:union7` 은 `union7` 이 이 문서에 인스턴스로 없어서 관계로 만들 수 없습니다. 버려지지 않고, 사유와 함께 따로 보관됩니다.

SQL

```
SELECT s->>'predicate' AS predicate, s->>'reason' AS reason
FROM jsonb_array_elements(og_mapping_report('fleet')->'sample') AS s;
```

결과

| predicate                         | reason                                         |
|-----------------------------------|------------------------------------------------|
| http://example.org/fleet#memberOf | object IRI is not an instance in this document |

(1 row)

## 2단계. 배차 앱이 Cypher로 읽고 쓴다

OWL 클래스가 타입이 되었으므로 Cypher에서는 그냥 라벨입니다. `Vehicle` 을 물으면 하위 클래스의 인스턴스가 모두 답합니다. `Vehicle` 자체의 인스턴스는 하나도 없다는 점을 보세요.

CYPHER

```
MATCH (v:Vehicle) RETURN v.plate AS plate, labels(v) AS labels, v.iri AS iri ORDER BY plate
```

결과

| plate   | labels                                | iri                         |
|---------|---------------------------------------|-----------------------------|
| 81바1001 | ["Vehicle", "Truck"]                  | http://example.org/fleet#v1 |
| 81바1002 | ["Vehicle", "Truck", "ElectricTruck"] | http://example.org/fleet#v2 |
| 81바1003 | ["Vehicle", "Van"]                    | http://example.org/fleet#v3 |

(3 rows)

CYPHER

```
MATCH (d:Driver)-[:drives]->(v:Truck) RETURN d.name AS driver, v.plate AS plate ORDER BY plate
```

결과

```
driver | plate
-----+-----
김민수 | 81바1001
이서연 | 81바1002
(2 rows)
```

검산: `Truck` 은 v1과, 그 하위 `ElectricTruck` 인 v2입니다. Van인 v3은 빠집니다.

**주의 — 숫자 리터럴은 문자열로 들어옵니다.** Turtle에 `"5000"^^xsd:integer` 로 적었지만 프로퍼티는 `text` 로 만들어졌습니다. (`xsd:date` 는 `date` 가 됩니다.) 그래서 숫자 비교는 그대로는 실패합니다.

CYPHER

```
MATCH (v:Vehicle) WHERE v.payload >= 3000 RETURN v.plate
```

결과

```
ERROR: operator does not exist: text >= integer
HINT: No operator matches the given name and argument types. You might need to add explicit type casts.
```

지금은 Cypher에서 변환해서 씁니다.

CYPHER

```
MATCH (v:Vehicle) WHERE toInteger(v.payload) >= 3000
RETURN v.plate AS plate, toInteger(v.payload) AS kg ORDER BY kg DESC
```

결과

```
plate | kg
-----+-----
81바1001 | 5000
81바1002 | 3500
(2 rows)
```

이제 씁니다. 김민수 기사에게 새 밴을 배정합니다.

CYPHER

```
MATCH (d:Driver {name: '김민수'})
CREATE (v:Van {plate: '81바1004', payload: '1000'})
CREATE (d)-[:drives]->(v)
RETURN d.name AS driver, v.plate AS plate
```

결과

```
driver | plate
-----+-----
김민수 | 81바1004
(1 row)
```

온톨로지의 `rdfs:domain` / `rdfs:range` 는 `drives` 의 역할 제약이 되어, Cypher의 쓰기에도 적용됩니다. 기사가 기사를 운전할 수는 없습니다.

```
MATCH (a:Driver {name: '김민수'}), (b:Driver {name: '이서연'}) CREATE (a)-[:drives]->(b)
```

결과

```
ERROR: role 'object' of relation 'drives' requires a 'Vehicle', got 'Driver'
```

### 3단계. 다시 RDF로 내보낸다

표준 담당에게 현재 상태를 돌려줍니다. `og_dump_rdf` 는 Turtle 또는 N-Triples를 냅니다. 전체는 길어서 v2에 관한 줄과, 1단계에서 매핑되지 못한 트리플만 골라 봅니다.

SQL

```
SELECT line FROM regexp_split_to_table(og_dump_rdf('fleet', 'ntriples'), E'\n') AS line
WHERE line LIKE '%fleet#v2>%' OR line LIKE '%memberOf%';
```

결과

line

```
-----
-----
<http://example.org/fleet#v2> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://example.org/fleet#ElectricTruck> .
<http://example.org/fleet#v2> <urn:og:prop:payload> "3500" .
<http://example.org/fleet#v2> <urn:og:prop:plate> "81바1002" .
<http://example.org/fleet#d2> <http://example.org/fleet#drives> <http://example.org/fleet#v2> .
<http://example.org/fleet#d2> <http://example.org/fleet#memberOf> <http://example.org/fleet#union7>
.
(5 rows)
```

여기서 읽을 것이 셋입니다.

- 클래스, 인스턴스, 관계의 IRI는 들어온 그대로 나옵니다. 매핑되지 못한 `memberOf` 트리플도 다시 나옵니다.
- 데이터 프로퍼티의 술어는 원래 IRI(`ex:plate`)가 아니라 `urn:og:prop:plate` 로 나옵니다.
- 리터럴의 데이터타입(`^^xsd:integer`)과 언어 태그는 덤프에 붙지 않습니다.

즉 구조는 왕복하지만 무손실 왕복은 아닙니다. 덤프를 새 그래프에 다시 넣어 확인합니다. Cypher로 만든 밴(81바1004)까지 들어 있어서 차량은 4대입니다.

SQL

```
SELECT og_create_graph('fleet_copy');
SELECT og_load_rdf('fleet_copy', og_dump_rdf('fleet'));
```

```
MATCH (d:Driver)-[:drives]->(v:Vehicle)
RETURN d.`urn:og:prop:name` AS driver, v.`urn:og:prop:plate` AS plate, labels(v) AS labels ORDER BY
plate
```

결과

| driver | plate   | labels                                |
|--------|---------|---------------------------------------|
| 김민수    | 81바1001 | ["Truck", "Vehicle"]                  |
| 이서연    | 81바1002 | ["ElectricTruck", "Truck", "Vehicle"] |
| 김민수    | 81바1004 | ["Van", "Vehicle"]                    |

(3 rows)

계층과 관계는 그대로 답합니다. 다만 프로퍼티 이름이 `plate` 가 아니라 `urn:og:prop:plate` 가 되었습니다.

## 4단계. 정비팀이 TypeQL로 같은 그래프를 읽는다

TypeQL은 `og_typeql(graph, query)` 로 실행합니다. 정비팀은 아무것도 정의하지 않았지만, 스키마를 TypeQL로 뽑으면 OWL에서 온 타입 계층이 이미 있습니다.

SQL

```
SELECT og_typeql_schema('fleet') AS schema;
```

결과

| schema                                                                                                                                                                                    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>define entity Driver; entity ElectricTruck sub Truck; entity Truck sub Vehicle; entity Van sub Vehicle; entity Vehicle;  relation drives,   relates subject,   relates object;</pre> |

(1 row)

`isa` 도 하위 타입을 포함합니다. RDF로 들어온 3대와 Cypher로 만든 1대, 합쳐 4대입니다.

SQL

```
SELECT og_typeql('fleet', $$ match $v isa Vehicle; reduce $n = count; $$) AS answer;
```

결과

| answer              |
|---------------------|
| <pre>{"n": 4}</pre> |

(1 row)

여기서 경계가 나옵니다. Cypher·RDF가 쓴 `plate` 는 노드의 **프로퍼티**(열)이고, TypeQL의 `has` 는 **속성 타입**의 인스턴스를 찾습니다. 둘은 같은 것이 아니어서, TypeQL은 `plate` 를 읽지 못합니다.

```
SELECT og_typeql('fleet', $$ match $v isa Vehicle, has plate $p; $$);
```

SQL

```
ERROR: typeql error: type 'plate' is not defined in this graph
```

결과

관계도 마찬가지입니다. Cypher·RDF의 `drives` 는 두 끝점을 잇는 간선이고, TypeQL의 관계는 역할을 가진 인스턴스입니다. 아래 질의는 오류 없이 **0건**을 돌려줍니다. 간선 2개가 있는데도 그렇다는 점을 기억해 두세요.

```
SELECT og_typeql('fleet', $$ match drives (subject: $d, object: $v); $$) AS answer;
```

SQL

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

결과

정리하면 TypeQL은 다른 언어가 만든 **타입과 인스턴스**는 보지만, 그 **프로퍼티와 간선**은 보지 못합니다.

## 5단계. 정비팀이 TypeQL로 쓴다

정비팀은 OWL에서 온 타입 위에 자기 스키마를 엮습니다. 차대번호(`vin`)와 사번(`staff-no`)은 둘 다 `code` 의 하위 속성이고, 점검(`inspection`)은 차량·점검자·입회 기사, 세 역할을 가진 관계입니다. 끝점이 둘뿐인 간선으로는 적을 수 없는 모양입니다.

```
SELECT og_typeql('fleet', $$
define
  attribute code @abstract, value string;
  attribute vin sub code;
  attribute staff-no sub code;
  attribute depot, value string;
  attribute result, value string @values("pass", "fail");
  entity Vehicle, owns vin @key, owns depot, plays inspection:subject;
  entity Driver, owns staff-no @key, plays inspection:witness;
  entity Inspector, owns staff-no @key, plays inspection:inspector;
  relation inspection, relates subject, relates inspector, relates witness, owns result;
$$) AS answer;
```

SQL

```
answer
-----
{"defined": 9}
(1 row)
```

결과

`Vehicle` 과 `Driver` 는 새로 만든 것이 아니라 OWL에서 온 바로 그 타입입니다. `Vehicle` 에 선언한 `owns vin` 은 하위 타입이 물려받습니다.

주의 — `sub` 가 하나도 없는 `define` 은 지금 제대로 동작하지 않습니다. 위 묶음에는 `vin sub code` 가 있어서 문제가 없지만, `sub` 없이 관계와 `plays` 만 선언하면, 같은 묶음에서 선언한 역할을 찾지 못합니다.

```
SELECT og_typeql('fleet', $$ define relation handover, relates giver; entity Driver, plays
handover:giver; $$);
```

SQL

```
ERROR:  typeql error: plays 'handover:giver': relation 'handover' declares no role 'giver'
```

결과

데이터를 넣습니다. 전기 트럭과 밴 한 대씩(둘 다 인천 차고), 점검자 한 명, 신입 기사 한 명입니다.

```
SELECT og_typeql('fleet', $$
insert
  $e isa ElectricTruck, has vin "KMF-E-0005", has depot "인천";
  $n isa Van, has vin "KMF-V-0006", has depot "인천";
  $i isa Inspector, has staff-no "Q-17";
  $d isa Driver, has staff-no "D-03";
$$);
SELECT og_typeql('fleet', $$
match
  $v isa Vehicle, has vin "KMF-E-0005";
  $i isa Inspector, has staff-no "Q-17";
  $d isa Driver, has staff-no "D-03";
insert (subject: $v, inspector: $i, witness: $d) isa inspection, has result "pass";
$$);
```

SQL

`@key` 는 쓰는 시점에 검사됩니다. 같은 차대번호로 한 대를 더 넣어 봅니다.

```
SELECT og_typeql('fleet', $$ insert $v isa Truck, has vin "KMF-E-0005"; $$);
```

SQL

```
ERROR:  typeql error: 'vin' is declared @key but that value is already owned by instance
274877906946
```

결과

## 6단계. 서로가 쓴 것을 읽는다

차량은 RDF 3대 + Cypher 1대 + TypeQL 2대, 모두 6대입니다. 두 언어가 같은 수를 답합니다.

```
MATCH (v:Vehicle) RETURN count(v) AS vehicles
```

CYPHER

```
vehicles
-----
6
(1 row)
```

결과

```
SELECT og_typeql('fleet', $$ match $v isa Vehicle; reduce $n = count; $$) AS answer;
```

SQL

answer

```
-----
{"n": 6}
(1 row)
```

결과

TypeQL이 쓴 속성은 Cypher에서 **노드**로 보입니다. 소유는 내부 관계 타입 **\$has** 의 간선입니다. 속성에도 계층이 있어서 **code** 를 물으면 **vin** 과 **staff-no** 가 함께 답합니다.

```
MATCH (owner)-[:`$has`]->(c:code)
RETURN labels(owner) AS owner, labels(c) AS attribute, c.val AS value ORDER BY value
```

CYPHER

| owner                                 | attribute            | value      |
|---------------------------------------|----------------------|------------|
| ["Driver"]                            | ["code", "staff-no"] | D-03       |
| ["Vehicle", "Truck", "ElectricTruck"] | ["code", "vin"]      | KMF-E-0005 |
| ["Vehicle", "Van"]                    | ["code", "vin"]      | KMF-V-0006 |
| ["Inspector"]                         | ["code", "staff-no"] | Q-17       |

(4 rows)

결과

같은 질문을 TypeQL로 하면 **has code** 한 줄입니다. 같은 4건입니다.

```
SELECT og_typeql('fleet', $$ match $owner has code $c; select $c; sort $c asc; $$) AS answer;
```

SQL

answer

```
-----
{"c": "D-03"}
{"c": "KMF-E-0005"}
{"c": "KMF-V-0006"}
{"c": "Q-17"}
(4 rows)
```

결과

TypeQL의 속성은 값마다 인스턴스가 하나입니다. 두 차량의 **depot "인천"** 은 **같은 노드**라서, Cypher에서는 문자열 비교가 아니라 패턴으로 "같은 차고의 차량"을 찾습니다.

```
MATCH (a:Vehicle)-[:`$has`]->(d:depot)<-[:`$has`]->(b:Vehicle), (a)-[:`$has`]->(x:vin), (b)-[:`$has`]->(y:vin)
WHERE x.val < y.val
RETURN d.val AS depot, x.val AS vehicle_a, y.val AS vehicle_b
```

CYPHER

| depot | vehicle_a  | vehicle_b  |
|-------|------------|------------|
| 인천    | KMF-E-0005 | KMF-V-0006 |

(1 row)

결과

이번에는 Cypher가 TypeQL의 인스턴스에 덧붙입니다. 차대번호로 전기 트럭을 찾아 번호판을 달고, 이서연 기사에게 배정합니다. 한 노드에 TypeQL의 속성과 Cypher의 프로퍼티·간선이 함께 있게 됩니다.

CYPHER

```
MATCH (d:Driver {name: '이서연'}), (v:Vehicle)-[:`$has`]->(:vin {val: 'KMF-E-0005'})
SET v.plate = '81바1005'
CREATE (d)-[:drives]->(v)
```

CYPHER

```
MATCH (d:Driver)-[:drives]->(v:Truck)-[:`$has`]->(n:vin)
RETURN d.name AS driver, v.plate AS plate, n.val AS vin, labels(v) AS labels
```

결과

| driver | plate   | vin        | labels                                |
|--------|---------|------------|---------------------------------------|
| 이서연    | 81바1005 | KMF-E-0005 | ["Vehicle", "Truck", "ElectricTruck"] |

(1 row)

반대 방향은 되지 않습니다. 4단계에서 본 대로 TypeQL은 `plate` 를 읽지 못하므로, RDF·Cypher로 만든 차량 가운데 **특정 한 대**를 TypeQL에서 집어낼 수 없습니다. TypeQL로 다룰 인스턴스에는 TypeQL 속성이 있어야 합니다.

점검 관계는 Cypher에서 `inspection` 라벨의 노드이고, 그 속성도 `$has` 로 읽힙니다. 그러나 누가 어떤 역할로 참여했는지는 간선이 아니라 역할 배정으로 저장됩니다. Cypher 패턴으로는 따라갈 수 없고, SQL 뷰 `og_typeql_role` 로 읽거나 TypeQL로 묻습니다.

SQL

```
SELECT relation_type, role, player_type FROM og_typeql_role ORDER BY role;
```

결과

| relation_type | role      | player_type   |
|---------------|-----------|---------------|
| inspection    | inspector | Inspector     |
| inspection    | subject   | ElectricTruck |
| inspection    | witness   | Driver        |

(3 rows)

SQL

```
SELECT og_typeql('fleet', $$
match $r isa inspection, links (subject: $v, inspector: $i, witness: $w), has result $res;
fetch { "vin": $v.vin, "inspector": $i.staff-no, "witness": $w.staff-no, "result": $res };
$$) AS answer;
```

결과

| answer                                                                            |
|-----------------------------------------------------------------------------------|
| { "vin": "KMF-E-0005", "result": "pass", "witness": "D-03", "inspector": "Q-17" } |

(1 row)

두 언어는 **트랜잭션도 공유**합니다. 한 트랜잭션에서 Cypher와 TypeQL로 한 대씩 넣으면 TypeQL이 8대를 세고, 되돌리면 둘 다 사라집니다.

```
BEGIN;
SELECT og_cypher('fleet', $$ CREATE (:Truck {plate: '81바1009'}) $$);
SELECT og_typeql('fleet', $$ insert $v isa Van, has vin "KMF-V-0010"; $$) AS inserted;
SELECT og_typeql('fleet', $$ match $v isa Vehicle; reduce $n = count; $$) AS inside_transaction;
ROLLBACK;
SELECT og_typeql('fleet', $$ match $v isa Vehicle; reduce $n = count; $$) AS after_rollback;
```

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

inserted
-----
{"rows": 1, "operations": 1}
(1 row)

inside_transaction
-----
{"n": 8}
(1 row)

after_rollback
-----
{"n": 6}
(1 row)
```

감사 로그도 하나입니다. 어느 언어로 들어온 질의인지가 함께 남습니다.

```
SELECT lang, rows_out, left(regexp_replace(query, '\s+', ' ', 'g'), 58) AS query
FROM og_data.og_audit ORDER BY audit_id DESC LIMIT 3;
```

| lang   | rows_out | query                                                        |
|--------|----------|--------------------------------------------------------------|
| typeql | 1        | [fleet] match \$v isa Vehicle; reduce \$n = count;           |
| typeql | 1        | [fleet] match \$r isa inspection, links (subject: \$v, inspe |
| cypher | 1        | [fleet] MATCH (d:Driver)-[:drives]->(v:Truck)-[:`\$has`]->(  |

(3 rows)

## 7단계. 하지 않는 것

**SPARQL은 없습니다.** RDF는 들어오고 나가지만, 질의는 Cypher나 TypeQL로 합니다. SPARQL 함수는 존재하지 않습니다. SHACL도 없습니다.

```
SELECT pronom FROM pg_proc WHERE pronom ILIKE 'og%sparql%' OR pronom ILIKE 'og%shacl%';
```

SQL

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

결과

**Turtle**은 부분집합입니다. 접두사, `a`, `;` 와 `,` 목록, 타입·언어 태그가 붙은 리터럴은 읽습니다. 빈 노드 속성 목록 `[ ... ]` 은 파싱 오류입니다.

```
SELECT og_load_rdf('fleet', '@prefix ex: <http://example.org/fleet#> . ex:v1 ex:garage [ ex:city "인천" ] .');
```

SQL

```
ERROR:  RDF parse error: unexpected '[' at offset 58 (expected ';', ',' or '.')
```

결과

**TypeQL** 함수는 선언만 됩니다. `fun` 은 저장되고 `og_typeql_schema()` 에 다시 나오지만, 호출하면 오류입니다.

```
SELECT og_typeql('fleet', $$ define fun fleet_size() -> integer: match $v isa Vehicle; return count; $$) AS declared;
SELECT og_typeql('fleet', $$ match let $n = fleet_size(); $$);
```

SQL

```
ERROR:  typeql error: function 'fleet_size' is not available in expressions. built-ins: round, abs, floor, ceil, length. user-defined functions are spec 010 phase 6
```

결과

**TypeDB 2.x** 문법과 `undefine` 은 이름을 대며 거부합니다. 조용히 다른 뜻으로 실행하지 않습니다.

```
SELECT og_typeql('fleet', $$ match $v isa Vehicle; get $v; $$);
SELECT og_typeql('fleet', $$ undefine entity Inspector; $$);
```

SQL

```
ERROR:  typeql error: line 1:24: 'get' is TypeDB 2.x syntax. this engine implements TypeQL 3.x - use 'select'/'fetch' instead of 'get'
ERROR:  typeql error: 'undefine' is not implemented yet (spec 010 phase 5)
```

결과

**TypeQL** 데이터가 섞인 그래프의 덤프는 다시 적재되지 않습니다. 덤프 자체는 나옵니다. 속성 타입은 `urn:og:vin` 같은 클래스로, 속성 인스턴스와 관계 인스턴스는 `urn:og:node:...` 노드로 나가고, 역할 배정은 덤프에 없습니다. 이것을 `og_load_rdf` 에 넣으면 실패합니다. 3단계의 왕복은 TypeQL 데이터가 없을 때의 이야기입니다.

```
SELECT og_create_graph('fleet_again') IS NOT NULL AS created;
SELECT og_load_rdf('fleet_again', og_dump_rdf('fleet'));
```

SQL

```
ERROR:  'urn:og:inspection' is not an entity type
```

결과

## Neo4j 드라이버는 그대로 붙입니다

배차 앱은 코드를 바꾸지 않습니다. Bolt 게이트웨이가 Bolt 4.4를 말하므로, Neo4j 드라이버의 URI만 바꾸면 2단계의 Cypher가 그대로 실행됩니다. Neo4j의 database는 여기의 그래프이고, 사용자는 PostgreSQL 역할입니다. 아래 코드는 이 문서의 빌드가 실행하지 않습니다 (게이트웨이는 별도 프로세스입니다). 실행 방법과 지원 범위는 `bolt/README.md`에 있습니다. Bolt 5.x, `Path`, TLS는 아직 없습니다.

PYTHON

```
driver = GraphDatabase.driver("bolt://og-host:7687", auth=("pg_role", "pg_password")) # URI만 바꿉니다
driver.session(database="fleet").run("MATCH (d:Driver)-[:drives]->(v:Truck) RETURN d.name, v.plate")
```

TypeQL 쪽은 **언어** 호환입니다. TypeDB의 와이어 프로토콜과 드라이버는 지원하지 않으며, `og_typeql()`을 SQL로 호출합니다.

## 한 장으로 다시 보기

| 하려는 일                  | 문장                                                                                                          |
|------------------------|-------------------------------------------------------------------------------------------------------------|
| Turtle / N-Triples 적재  | <code>SELECT og_load_rdf('fleet', \$rdf\$ ... \$rdf\$)</code> — 결과는 매핑 보고서                                  |
| 매핑되지 못한 트리플 · RDF 덤프   | <code>SELECT og_mapping_report('fleet') · SELECT og_dump_rdf('fleet', 'ntriples')</code>                    |
| Cypher                 | <code>SELECT og_cypher('fleet', \$\$ MATCH (v:Vehicle) RETURN v \$\$)</code> 또는 Bolt                        |
| TypeQL 질의 · 스크립트       | <code>SELECT og_typeql('fleet', \$\$ match ... \$\$)</code> · <code>og_typeql_script('fleet', 파일 내용)</code> |
| 스키마를 TypeQL로 보기        | <code>SELECT og_typeql_schema('fleet')</code>                                                               |
| TypeQL 속성 · 역할을 밖에서 보기 | <code>MATCH (v)-[:`\$has`]-&gt;(a:vin) RETURN a.val</code> · <code>SELECT * FROM og_typeql_role</code>      |

| 누가 쓴 것을        | Cypher가 읽는가                          | TypeQL이 읽는가                               |
|----------------|--------------------------------------|-------------------------------------------|
| OWL 클래스 계층     | 예 — 라벨, 하위 타입 포함                     | 예 — <code>isa</code> , <code>sub</code>   |
| RDF·Cypher의 노드 | 예                                    | 타입과 개수까지 ( <code>isa</code> ). 프로퍼티는 못 읽음 |
| RDF·Cypher의 간선 | 예                                    | 아니오 (0건)                                  |
| TypeQL 엔터티     | 예 — 프로퍼티·간선을 덧붙일 수도 있음               | 예                                         |
| TypeQL 속성      | 예 — <code>\$has</code> 간선 건너의 노드     | 예 — <code>has</code>                      |
| TypeQL 관계의 역할  | 아니오 — <code>og_typeql_role</code> 뷰로 | 예 — <code>links</code> , 역할 이름            |

저장소·카탈로그·트랜잭션·감사 로그는 하나이고, 타입 계층은 세 언어가 공유합니다. Cypher의 프로퍼티와 TypeQL의 속성은 서로 다른 자리에 저장되므로, 두 언어로 함께 다룰 값은 어느 쪽에 둘지 먼저 정하세요.