**OAEP-2023-02 DOI: 10.54285/ldbc.EPWQ6741**

Published on LDBC's website April 2023

Originally published privately within Neo4j Inc, October 2019
Published with permission by LDBC as document LEX-014, modified for public availability by Neo4j, and previously Made available to LEX WG members.

# Introduction to GQL Schema design

Neo4j Inc., October 2019, LDBC LEX-014

# LEX-014
# “Introduction to GQL Schema design”

This summary presentation was created by the Neo4j
Query Languages Standards and Research team in 2019 for discussion within Neo4j Inc.

The team’s members during the time this working design was produced were
Hannes Voigt, Petra Selmer, Stefan Plantikow, Tobias Lindaaker, Peter Furniss, and Alastair Green.

# Introduction to GQL Schema design

## Target: A UML class diagram in ASCII Art “schema patterns”

Node and relationship patterns that look like query patterns, gathered into a graph schema or type.
“Draw” the LDBC SNB data model with a keyboard.

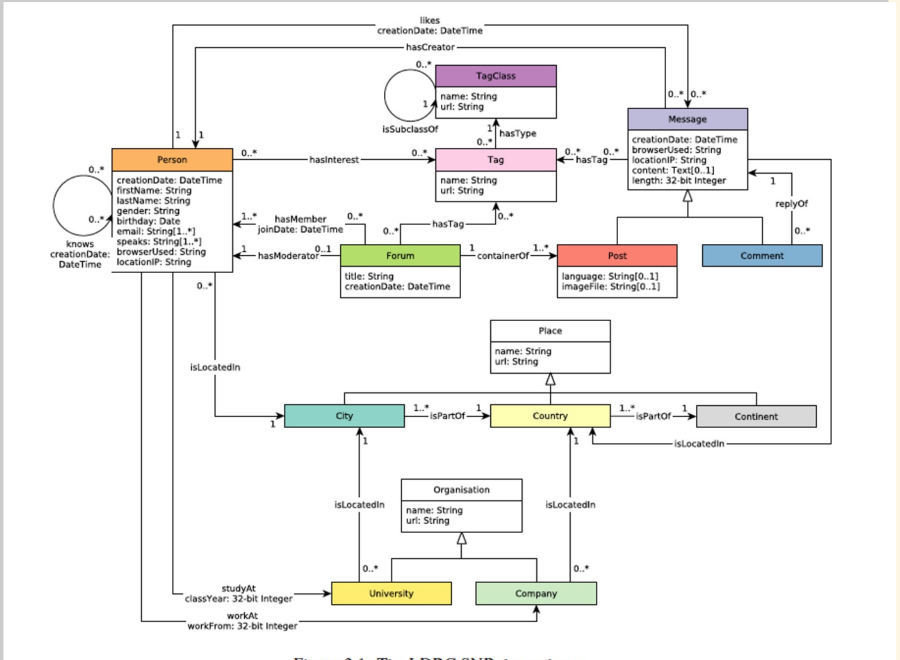
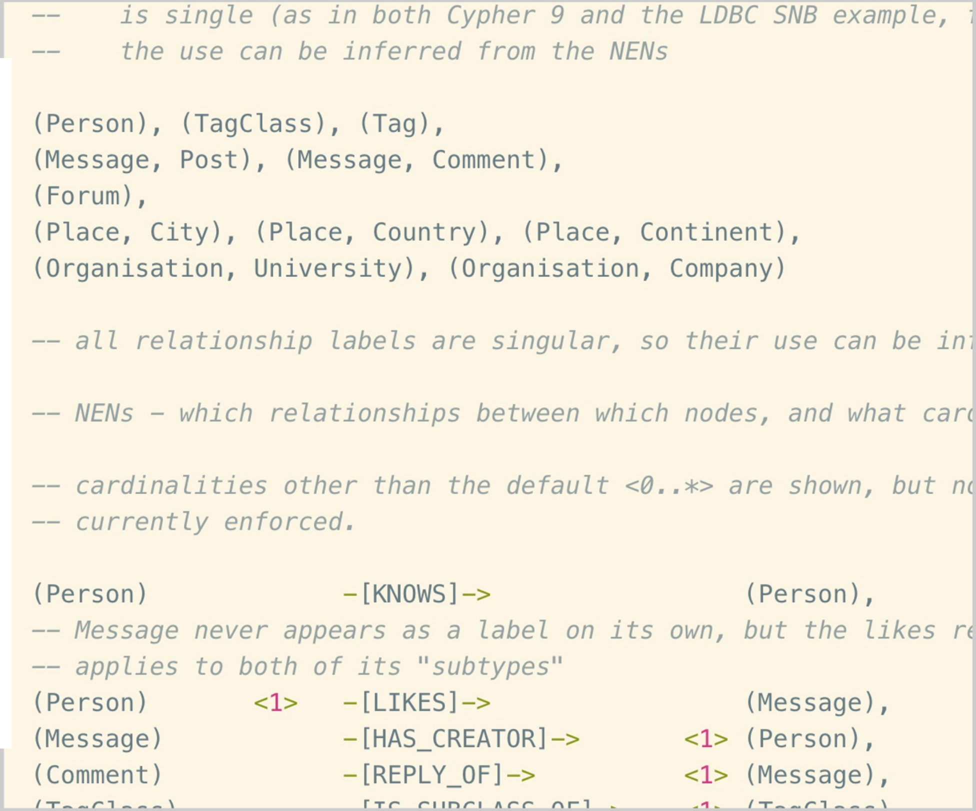


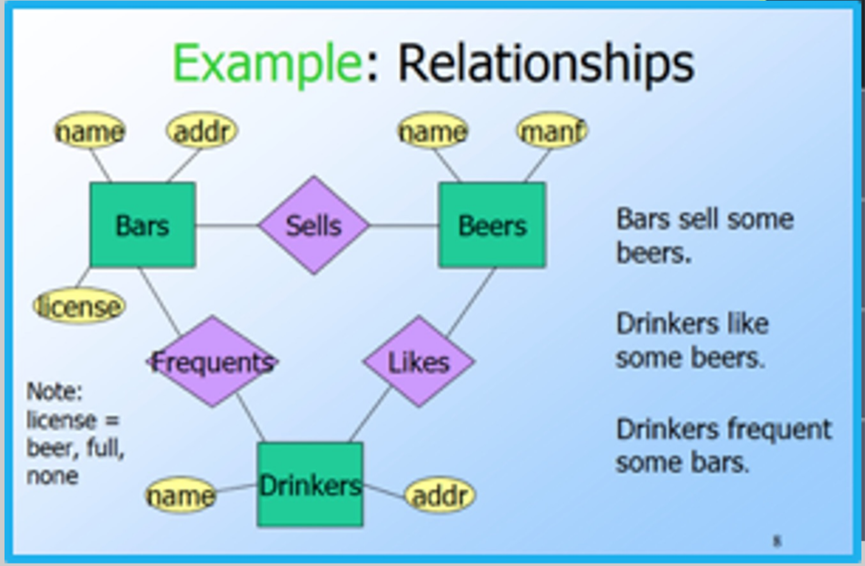
Figure 2.1: The LDBC SNB data schema


```
--    is single (as in both Cypher 9 and the LDBC SNB example,
--    the use can be inferred from the NENs

(Person), (TagClass), (Tag),
(Message, Post), (Message, Comment),
(Forum),
(Place, City), (Place, Country), (Place, Continent),
(Organisation, University), (Organisation, Company)

-- all relationship labels are singular, so their use can be in

-- NENs - which relationships between which nodes, and what car

-- cardinalities other than the default <0..*> are shown, but n
-- currently enforced.

(Person)                 -[KNOWS]->                (Person),
-- Message never appears as a label on its own, but the likes r
-- applies to both of its "subtypes"
(Person)         <1>     -[LIKES]->                (Message),
(Message)                -[HAS_CREATOR]->      <1> (Person),
(Comment)                -[REPLY_OF]->         <1> (Message),
(TagClass)               -[IS_SUBCLASS_OF]->   <1> (TagClass)
```

# An Entity-Relationship model and corresponding schema graph

The “Ullman drinking model” from the original version of Stanford’s class on data modelling, rendered as a graph model, where Neo4j node labels and “relationship types” are used as element type names. A schema graph.

[HTPS Querying Graphs]

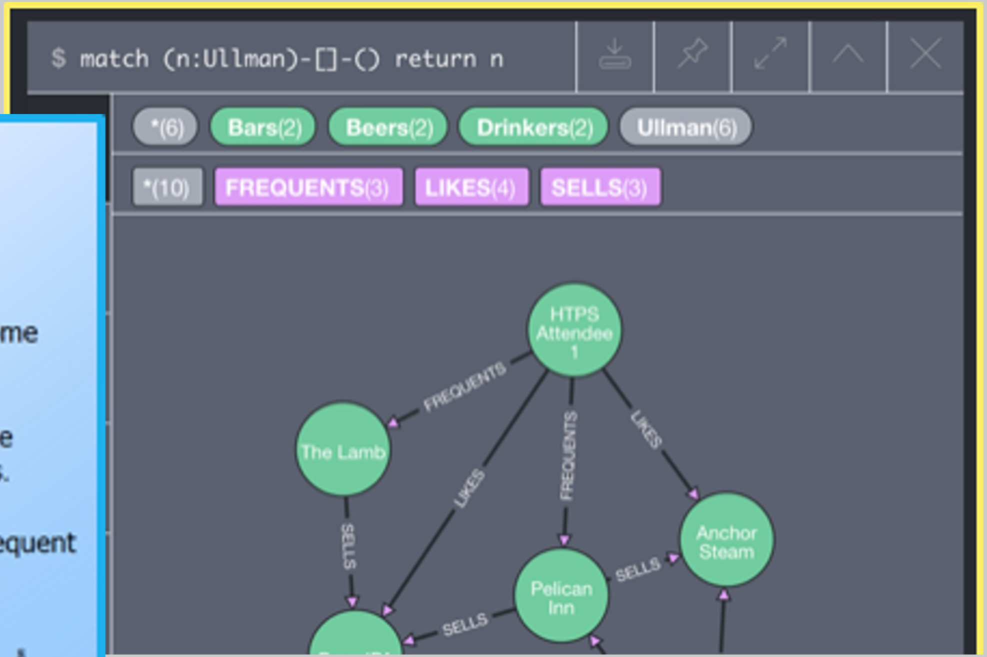


Bars sell some beers.

Drinkers like some beers.

Drinkers frequent some bars.

Note: license = beer, full, none



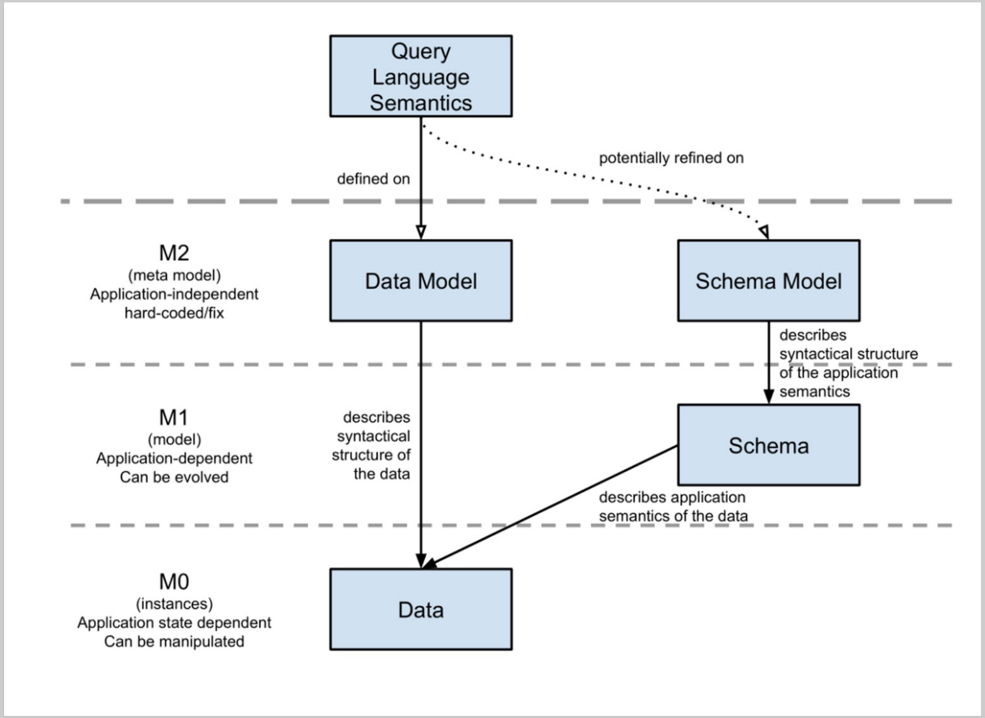
```
$ match (n:Ullman)-[]-() return n
```

## Model, schema and instance

A **data model** describes the kinds of values that can be held in a collection of information, and the way those values can be named, structured and related. It states general rules about information content, which are independent of business domain and application. *A query language is defined with respect to a data model.*

A **schema** is a set of rules which constrain the data model to cater for a specific business domain or application. It is like a "set builder": it defines the *kinds* of data *allowed* or *occurring* in a specific collection of information. *A query language may take advantage of schema information.*

An **instance** is a a specific collection of information, which conforms to the data model and is described by a schema. *It can be an operand or product of query language programs.*



## On the term “schema”

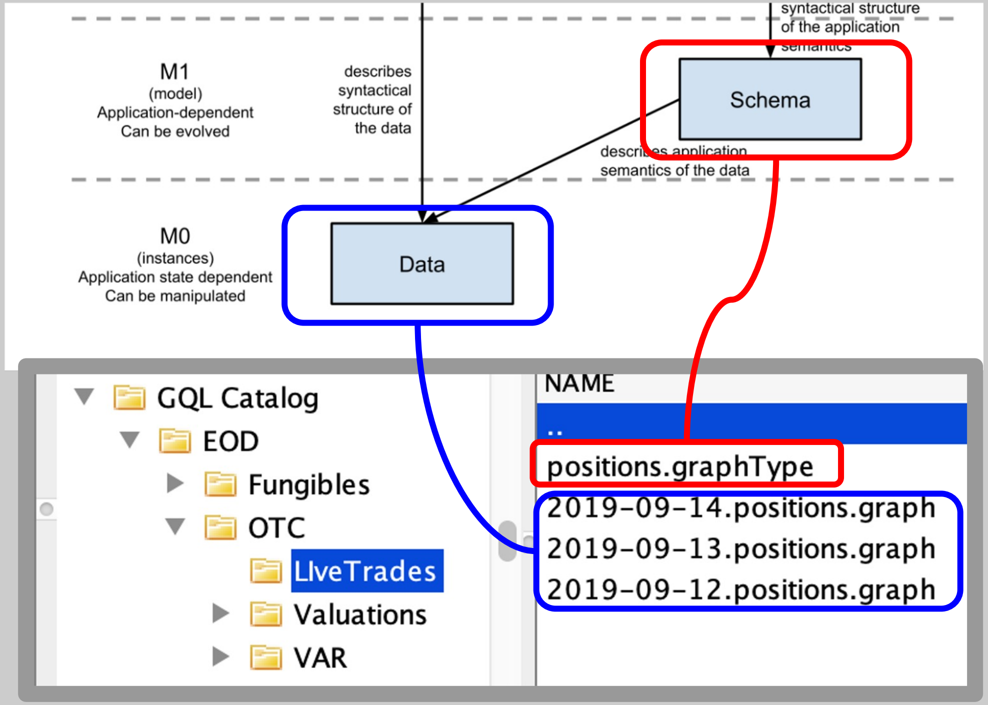
We will use the term **schema** in two ways

Generically, as a concept, as already stated

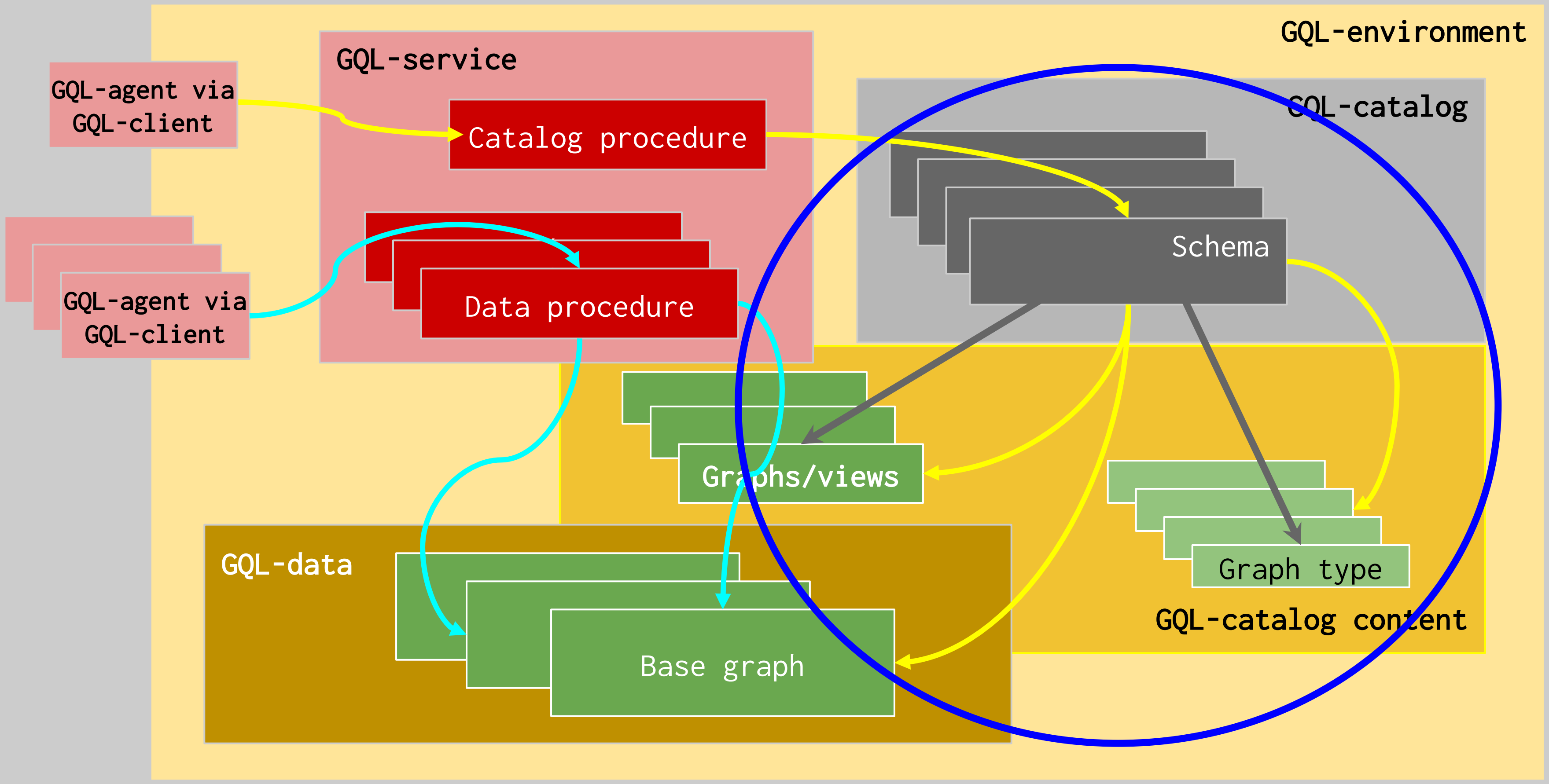
A **schema** is a set of rules which constrain the data model to cater for a specific business domain or application. It is like a “set builder”: it defines the *kinds* of data *allowed* or *occurring* in a specific collection of information. It is a predicate over the domain of the data model.

Specifically, in a way that resembles SQL:

A **GQL schema** is a directory in a GQL catalog which contains one to many named graphs (data, possibly accessed and maintained by one or more views) and one to many graph types, which are used to partially or wholly constrain those graphs. Such a directory plays the role of a generic schema, but also catalogs instances.


syntactical structure of the application semantics
M1
(model)
Application-dependent
Can be evolved
describes syntactical structure of the data
Schema
describes application semantics of the data
M0
(instances)
Application state dependent
Can be manipulated
Data
GQL Catalog
EOD
Fungibles
OTC
LIveTrades
Valuations
VAR
NAME
..
positions.graphType
2019-09-14.positions.graph
2019-09-13.positions.graph
2019-09-12.positions.graph

## The GQL-environment, and a schema's place within it



## GQL is a property Graph Query Language: elements and attributes

A property graph attaches **attributes (data values)** to **topological values** (nodes and edges), and the whole graph (graph attributes). Collectively, graph attributes, nodes and edges form a set of graph **elements**.

Attributes are the fields of **content records**: the attributes encode the information content associated with each topological element.

An attribute has a **name**, a **type** and a **value**. The type of an attribute determines the set of its possible values.

There is a set of types called **property types** (atomic types like integer, float, string, date, boolean, and parametric collection types which can be instantiated with an atomic type), each of which defines multiple possible values. An attribute whose type comes from this set is a **property**.

An attribute can also be of the `UNIT` type. That marks an attribute as a **label** or tag, with a single, unmodifiable, inaccessible value.

The topology of a graph can be viewed without reference to data values (an “unattributed graph”). Tabular representations of collections of data values can be extracted from a graph, thereby losing their structural context.

A mixed view of structure and data can also be taken (for example identifying or retrieving paths with associated attribution). Property graphs therefore straddle the pure graphs of RDF and the tuple-centric model of SQL.

## The GQL property graph data model

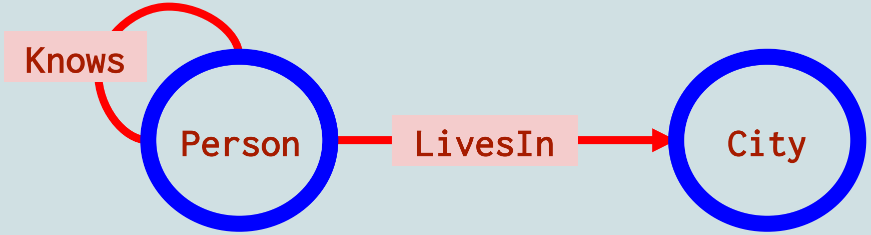
The **property graph data model** determines the kinds of types that users can define or use in a graph schema.

The GQL data model is an **optionally directed, attributed multigraph**

- A graph, and each of its nodes and edges, has a possibly empty set of typed attributes
- A set of attributes has two disjoint subsets, each possibly empty: its labels and its properties, which are divided by their types, as detailed in the previous slide
- An edge may be directed (one endpoint plays the role of tail, the other of head), or an edge may be undirected (neither endpoint has such a role)
- There can be many edges between two nodes

## GQL Schema proposal at a glance

```
CREATE GRAPH SocialNetwork (
     (Person {name::STRING, dob::DATE}),
     (City {name::STRING}),

     (Person)-[LivesIn]->(City),
     (Person)=[Knows]=(Person)
)
```



Undirected edges are envisioned as being supported vendor-optional in GQL

## GQL Schema proposal in one slide

Create a ***graph type*** with a name.

Define zero to many labels, and zero to many properties, using a similar pattern to one you would use in a Cypher `MATCH` or `CREATE` clause. That defines a ***content type***.

You can optionally give that content type a name, but you don't have to. If your content type has one label then it gets the same type name automatically. A graph itself can have a content type ("graph attributes").

You can also plug your content type declaration (or the name of one) inside `()` for **node types**, or inside one of the three parts of a relationship pattern `()-[]->()` to define a **relationship type.**

You can define some **keys**, if you want. Then you can create a **named graph** of that type.

```
CREATE GRAPH TYPE SocialNetwork (
    CONTENT TYPES
       Person :Person {name::STRING, dob::DATE}
    NODE TYPES
       (p::Person) KEY K_Person_personId (p.personId)
       (:City {name::STRING})
    EDGE TYPES
       (Person)-[:LIVES_IN]->(City),
       (Person)-[:Knows]->(Person)
)
```

```
CREATE DATABASE GlobalSocialNetwork

CREATE GRAPH
   GlobalSocialNetwork.EuropeanSocialNetwork
       OF SocialNetwork
```

```
CREATE DATABASE GlobalSocialNetwork
    GRAPHS
        GlobalSocialNetwork, // default graph type
        EuropeanSocialNetwork::SocialNetwork
```

## GQL Schema proposal in five slides (1) Node types and relationship types

If you plug a content type into an element type (node type, relationship type) with a name, but no label, then a label is automatically induced. If you specify a label, but no name, then you automatically get a named type.

Types are defined by a combination of properties and labels. You can have zero to many labels on nodes; in Neo4j you could restrict to exactly one "label" (reltype) on relationships. In either case you can have zero to many properties. *The combination of properties and labels is the type*. Names are just aliases for these *structural types*.

```
CREATE GRAPH TYPE SocialNetwork (
    CONTENT TYPES
       Person :Person {name::STRING, dob::DATE}
    NODE TYPES
       (p::Person) KEY K_Person_personName (p.name)
       (:City {name::STRING})
    RELATIONSHIP TYPES
       (Person)-[:LIVES_IN]->(City),
       (Person)-[:Knows]->(Person)
)
```

Explicitly named type `Person` is characterized by its structural contents, a label and a property

Label `:City` induces named type `City` if we set up a default in the session or the product to work that way, or mark the type as `LABEL IMPLIED BY TYPE NAME`.

Relationship types define triplet patterns. But if you want a type that only uses the arc between the endpoints, you can say `()-[ArcType]->()`. That is the equivalent of using a reltype as a type in e.g. schema security definitions.

## GQL Schema proposal in five slides (2) Subtyping

A content type, an element type (node or relationship type), and a graph type can all be subtyped. Having subtypes means that we can use the supertype to make schema statements that are true for all the subtypes.

All managers, employees, directors, consultants and officers of Neo4j are bound to take GDPR seriously. That statement can be made once with reference to the supertype “Neo4j agents”.

```
CREATE GRAPH TYPE CompanyGraph (
    Neo4jAgent {name::STRING}, // is not instantiable, just a content type
    (GDPRRegulations {date::DATE}), // these are instantiable node types
    (Employee <: Neo4jAgent), (Manager <: Neo4jAgent),
    (Director <: Neo4jAgent), (Officer <: Neo4jAgent),
    (Consultant <: Neo4jAgent), (ProfessionalAdvisor <: Neo4jAgent),
    // this is one relationship type declaration, implying many
    // instantiable relationship types, one per subtype of Neo4jagent
    (/* defaults to SUBTYPE OF */Neo4jAgent)-[TRAINED_ON {date::DATE}]->(GDPRRegulations)
)
```

Cypher queries operate in the same way: if you `MATCH (p:Person)` then you get the nodes with that label, but also all the ones with the `:Natural` and `:Corporate` labels. *We want schema patterns to behave like query patterns*.

This is good for familiarity (symmetry) and it’s good for conciseness.

You can also **subtype graphs**, so one extends another. That is used to define the different levels of schema.

## GQL Schema proposal in five slides (3) Strict, lax, no … schema

Schema is designed so that you can have *sealed* (or *final*) graph types that cannot be extended: this gives you closed schema like SQL. You can also have graph types that are not sealed, and can be extended. Unsealed or "open graphs" have to be the default because the default graph has to be extensible, so *Neo4j's schema-free design is the GQL default* (modulo the presence of undirected edges in the GQL data model).

Unsealed (non-final, extensible), "open" graph types let us specify the data model of the graph type that a user is defining. (Vendors are able to choose their default graph type, so Neo4j would be able to stick with our existing data model.)

```
CREATE GRAPH TYPE NEO4J_GRAPH (
    (),
    ()-[]->() // any directed relationships
)
```

This is not precisely how one would necessarily define a default model, but it illustrates the point about extensible, "open" types.

This "open graph" = "schema free" (it's water). A sealed graph = "schema governed" (a block of ice). A graph type that contains edge and relationship types *but is not closed* is "partial schema" (water and ice at 0°C).

There are two kinds of partial schema: ones that let you elaborate on the properties and labels, but enforces a rigid structure, and ones that let you add anything. Whole graphs or any element of a graph can be closed or finalized.

Neo4j is not compelled to implement any point on that spectrum. See type lattice for graph types for another view.

## GQL Schema proposal in five slides (4) Subgraph views

Every node or relationship type induces some ***built-in subgraph views*** that the user does not have to define.

There is an ***induced subgraph view for every element type*** *(edge type or node type)*, as well as a view over each whole graph.

So if we define a node type `(Employee :Person & Employee)` then that names a subgraph view for all nodes with the two labels `:Person` and `:Employee`.

A relationship type `(Person)-[HAS]->(SalaryHistory)` again induces a subgraph view of that name for all relationships of that pattern. (A type that describes all relationships with a particular Neo4j reltype would look like this `()-[HAS]->()`).

You can also create a user-defined subgraph view, that explicitly includes these induced views (and indeed could include other user-defined views).

```
CREATE VIEW HRGraph SUBGRAPH OF CompanyGraph (
    INCLUDE (Person), (SalaryHistory)
    INCLUDE (Person)-[HAS]->(SalaryHistory)
)
```

You can also `EXCLUDE` subset views from an included larger view.

You can also say `CREATE VIEW` and `INCLUDE` subgraph views from multiple graphs. See the slide on Fabric.

## GQL Schema proposal in five slides (5) Catalog and schema

The GQL-**catalog** plays the role of our system graphs. It has a concept of hierarchical naming, which derives from a hierarchy of directories, containing objects, aliases and other directories. It is deliberately analogous to a filesystem, which is a) a very well understood pattern, and b) is flexible.

Any ***directory*** can contain ***objects***. Catalog objects include content and graph types, graphs and views. Any object or directory in the catalog can be aliased, by creating an ***alias*** in a directory (symlink).

Some directories are distinguished as schema directories. A schema directory must contain at least one graph and one graph type. A user or role could be given access to a schema: there is no inbuilt relationship of that kind.

```
SET DIRECTORY Security.Roles
CREATE ROLES HR, FunctionalAdmins

GRANT CREATE ON Company.HR TO Security.Roles.FunctionalAdmins
SET DIRECTORY Conpany.HR
CREATE VIEW FunctionalGraph SUBGRAPH OF Company.CompanyGraph (
    INCLUDE (Person), (SalaryHistory)
    INCLUDE (Person)-[HAS]->(SalaryHistory)
)

GRANT MATCH ON GRAPH Company.HR.FunctionGraph TO Security.Roles.HR
```

Becomes a schema when it has at least one graph in it

## System-defined graph types

There are four abstract base four graph types:

A `NODES_GRAPH` has a node type `()`

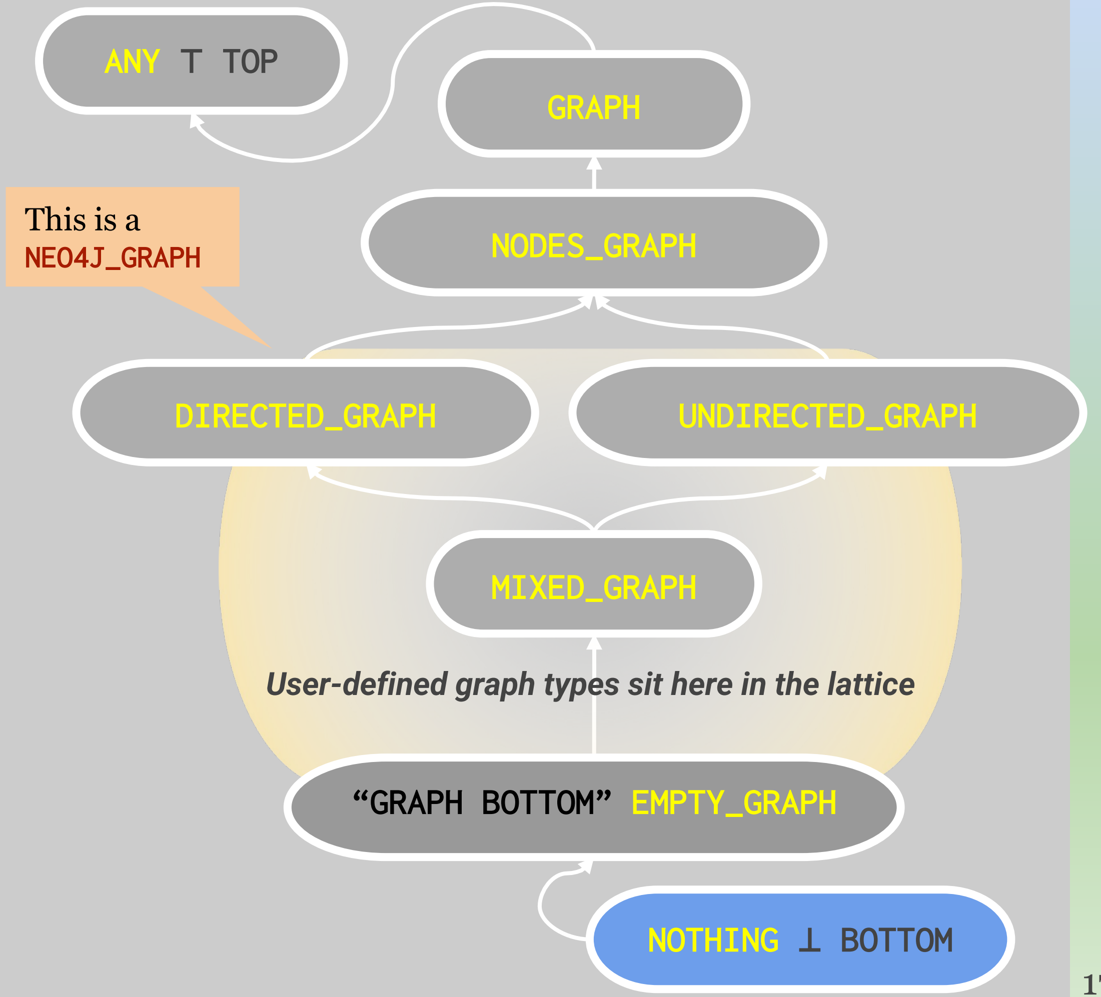
`DIRECTED_GRAPH` adds a type `()-[]->()`

`UNDIRECTED_GRAPH` adds a type `()=[]=()`

A `MIXED_GRAPH` combines the last two.

A **user-defined graph type** *subtypes* one of these types. The default supertype for a vendor implementation will be one of the latter three. This will determine the type of a graph whose type is not specified explicitly.

In a system that allows only strict schema, user-defined graph types are sealed by default. In a system that has lax schema, user-defined types are open by default.


ANY ⊤ TOP
GRAPH
This is a NEO4J_GRAPH
NODES_GRAPH
DIRECTED_GRAPH
UNDIRECTED_GRAPH
MIXED_GRAPH
User-defined graph types sit here in the lattice
"GRAPH BOTTOM" EMPTY_GRAPH
NOTHING ⊥ BOTTOM