Multiple graphs and composable queries in Cypher for Apache Spark

Max Kießling

openCypher Implementers Meeting V
Berlin, March 2019

neo4j

Outline

- Cypher for Apache Spark (CAPS) overview
  - Motivation
  - Architecture
  - Multiple Graphs
- SQL Property Graph Data Source and Graph DDL
  - Overview
  - SQL PGDS
  - Graph DDL
- Demo using LDBC social network

CAPS overview

For more details, have a look into our Spark+AI Summit talk
https://databricks.com/session/matching-patterns-and-constructing-graphs-with-cypher-for-apache-spark

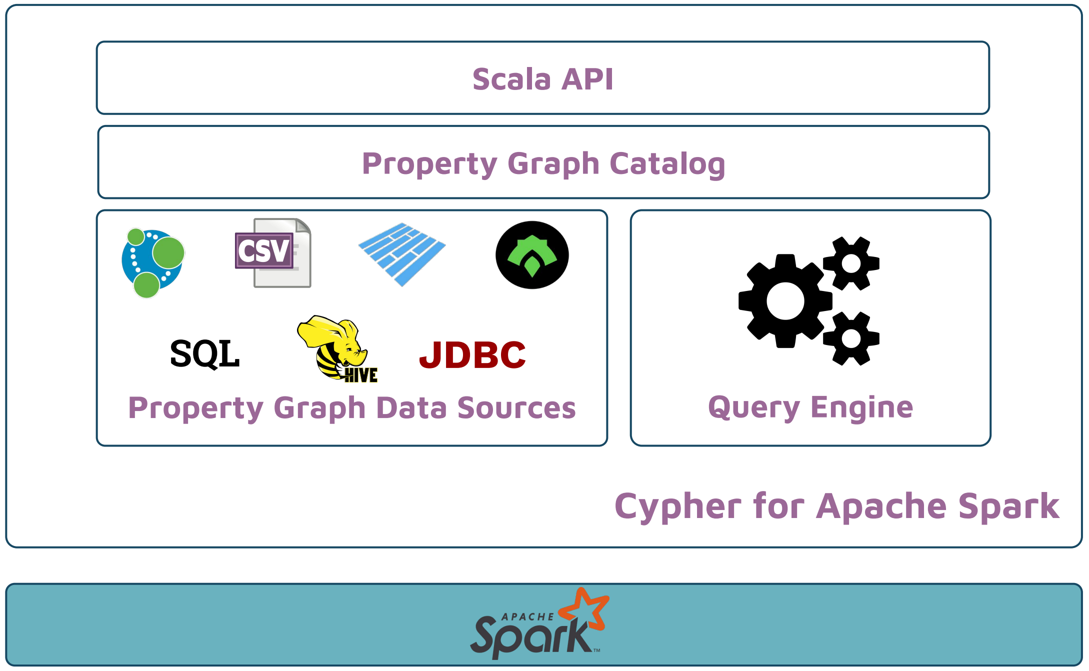
Motivation … What is Cypher for Apache Spark?

- Cypher implementation on top of Apache Spark
  - Apache Spark is the leading platform for distributed computations
  - Provides several APIs for relational querying (Spark SQL), machine learning (Spark ML) etc.
  - Already connects to many data sources (e.g. Parquet, Orc, CSV, JDBC, Hive, …)
- CAPS includes ...
  - A query engine to transform Cypher queries to relational operations over Spark SQL
  - Data source implementations for Neo4j and relational databases
  - A language (Graph DDL) to describe mappings between SQL DBs and property graphs

Motivation … What is CAPS good for?

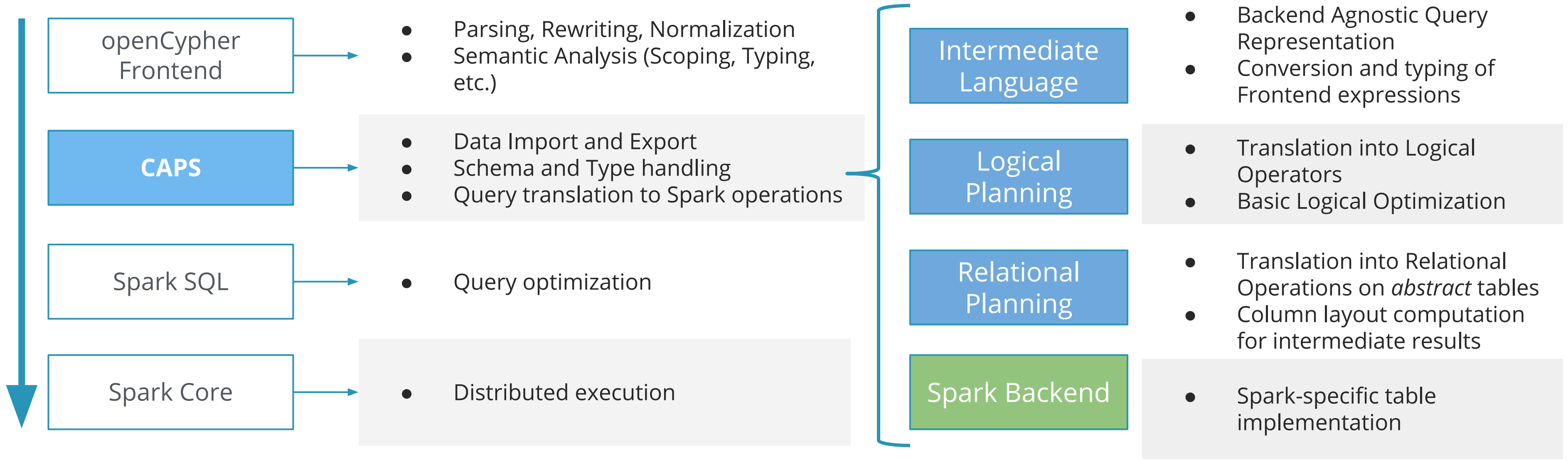
- Run Cypher queries in a distributed environment
- Support for multiple graphs and graph construction via Cypher (unlike Neo4j)
- Various data sources (File-based, JDBC, Neo4j)
- Support for merging graphs from CAPS into Neo4j
- Main use cases
  - Integrate non-graphy data from multiple heterogeneous data sources into one or more property graphs (i.e. ETL and graph transformations)
  - (Federated) data querying for distributed batch-style analytics
  - Integration with other Spark libraries (SQL, ML, …)

(Very) High-Level Architecture



Query engine architecture

```
MATCH (n:Person)-[:CAPTAIN]->(s:Ship)
WHERE n.name = ‘Morpheus’
RETURN n.name, s.name
```



- Parsing, Rewriting, Normalization
- Semantic Analysis (Scoping, Typing, etc.)

- Data Import and Export
- Schema and Type handling
- Query translation to Spark operations

- Query optimization

- Distributed execution

- Backend Agnostic Query Representation
- Conversion and typing of Frontend expressions

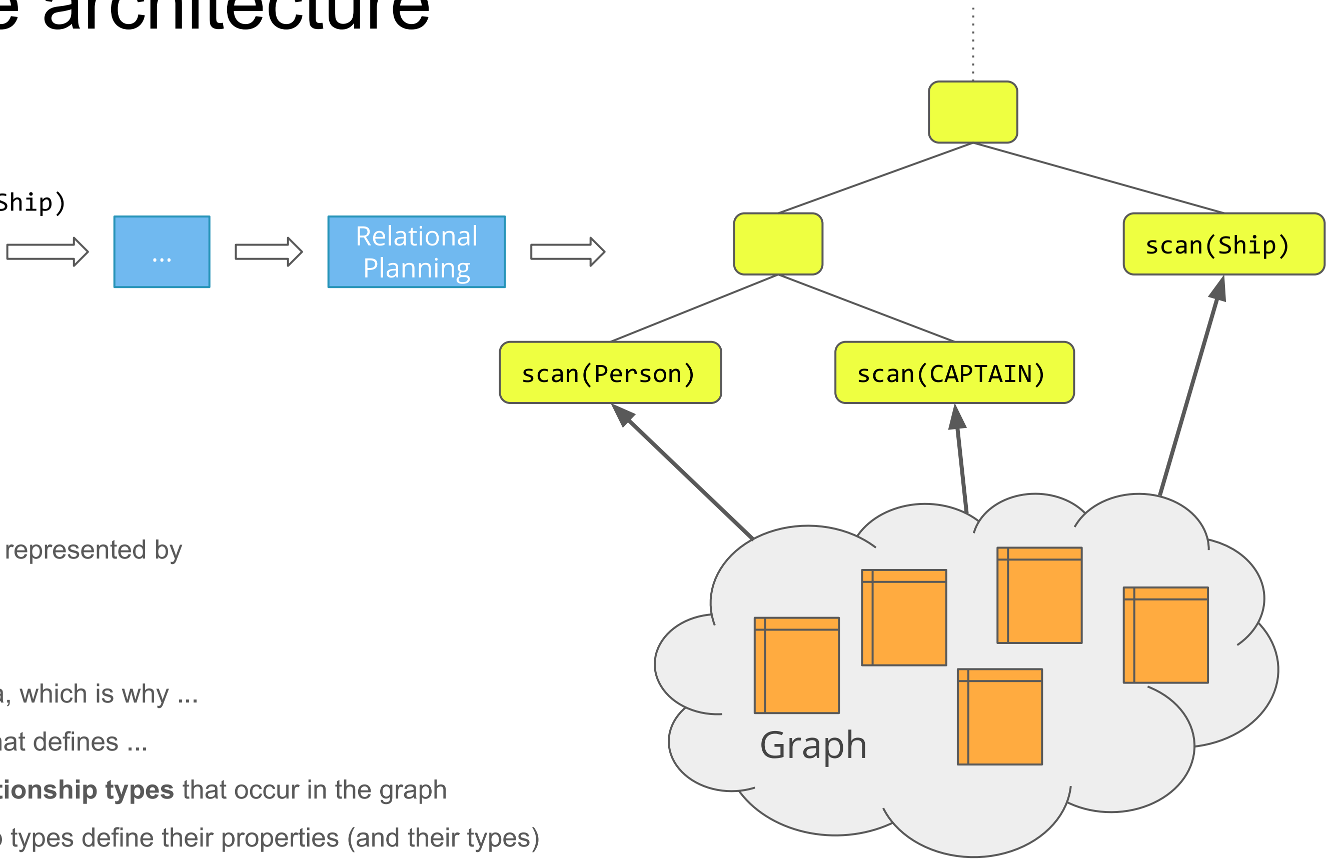
- Translation into Logical Operators
- Basic Logical Optimization

- Translation into Relational Operations on *abstract* tables
- Column layout computation for intermediate results

- Spark-specific table implementation

Query engine architecture

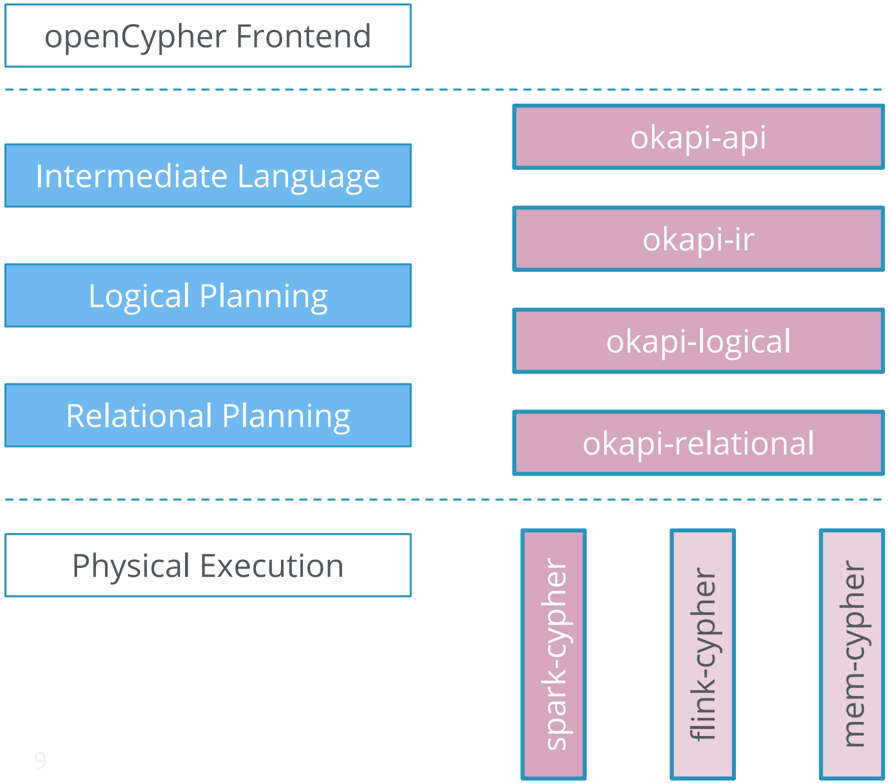
```
MATCH (n:Person)-[:CAPTAIN]->(s:Ship)
WHERE n.name = ‘Morpheus’
RETURN n.name, s.name
```



“Tables for Labels”

- In CAPS, property graphs are represented by
  - **Node tables**
  - **Relationship tables**
- Tables require a fixed schema, which is why ...
- Graphs have a **graph type,** that defines ...
  - **Node types** and **relationship types** that occur in the graph
  - Node and relationship types define their properties (and their types)

Query engine architecture



- Property Graph API
- Type System
- Property Graph Data Source API

- Intermediate Language, Typing
- Expressions

- Logical Planning

- Transformation into relational Operations on abstract table

- Session implementation
- Backend connector -> RelationalTable
- Data Source implementations

Cypher 10 - Multiple Graph Querying

- Combine data from multiple graphs in a single Cypher query
- Integrate data of different sources

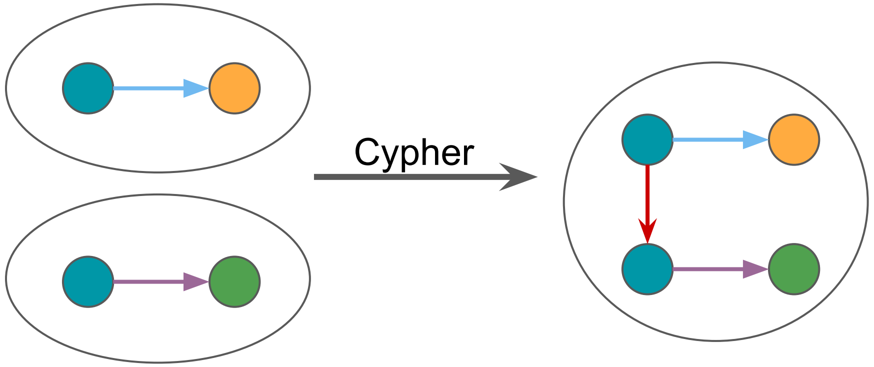
```
FROM social-net
MATCH (p:Person)
FROM products
MATCH (c:Customer)
WHERE p.email = c.email
RETURN p, c
```

Cypher 10 - Graph Construction

- Cypher 9
  - Input: Graph
  - Output: Table

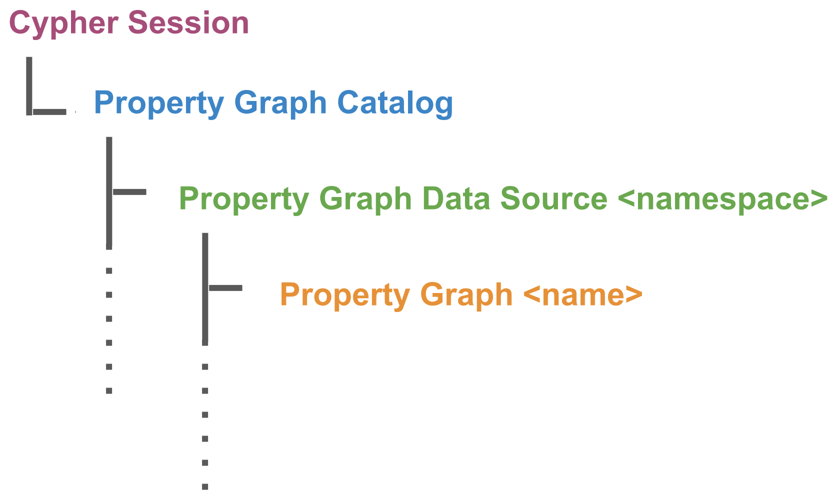
- Cypher 10
  - Input Graph
  - Ouput: Graph or Table

```
FROM social-net
MATCH (p:Person)
FROM products
MATCH (c:Customer)
WHERE p.email = c.email
CONSTRUCT ON social-net, products
  CREATE (c)
  CREATE (p)-[:SAME_AS]->(c)
RETURN GRAPH
```



Property Graph Catalog

- The Catalog manages Property Graph Data Sources (e.g. SQL, Neo4j, File-based)
- A Property Graph Data Source manages multiple Property Graphs
- Catalog functions (e.g. reading / writing a graph) can be executed via Cypher or Scala API



Property Graph Catalog

Cypher Session
- Property Graph Catalog
  - “social-net” (Neo4j PGDS)
    - “US” (Property Graph)

```
FROM social-net.US
MATCH (p:Person) RETURN p
```

Property Graph Catalog - Querying

- Cypher Session
  - Property Graph Catalog
    - “social-net” (Neo4j PGDS)
      - “US” (Property Graph)
      - “EU” (Property Graph)
    - “products” (SQL PGDS)
      - “2018” (Property Graph)
      - “2017” (Property Graph)

```
FROM social-net.US
MATCH (p:Person)
FROM products.2018
MATCH (c:Customer)
WHERE p.email = c.email
RETURN p, c
```

Property Graph Catalog - Construction

- Cypher Session
  - Property Graph Catalog
    - “social-net” (Neo4j PGDS)
      - “US” (Property Graph)
      - “EU” (Property Graph)
    - “products” (SQL PGDS)
      - “2018” (Property Graph)
      - “2017” (Property Graph)

```
CATALOG CREATE GRAPH social-net.US_new {
  FROM social-net.US
  MATCH (p:Person)
  FROM products.2018
  MATCH (c:Customer)
  WHERE p.email = c.email
  CONSTRUCT ON social-net.US
    CREATE (c)
    CREATE (p)-[:SAME_AS]->(c)
  RETURN GRAPH
}
```

Property Graph Catalog - Views

- Cypher Session
  - Property Graph Catalog
    - “social-net” (Neo4j PGDS)
      - “US” (Property Graph)
      - “EU” (Property Graph)
    - “products” (SQL PGDS)
      - “2018” (Property Graph)
      - “2017” (Property Graph)
  - Views
    - “youngPeople”

```
CATALOG CREATE VIEW youngPeople($sn) {
  FROM $sn
  MATCH (p:Person)-[r]->(n)
  WHERE p.age < 21
  CONSTRUCT
    CREATE (p)-[COPY OF r]->(n)
  RETURN GRAPH
}
```

```
FROM youngPeople(social-net.US)
MATCH (p:Person)
RETURN p
```

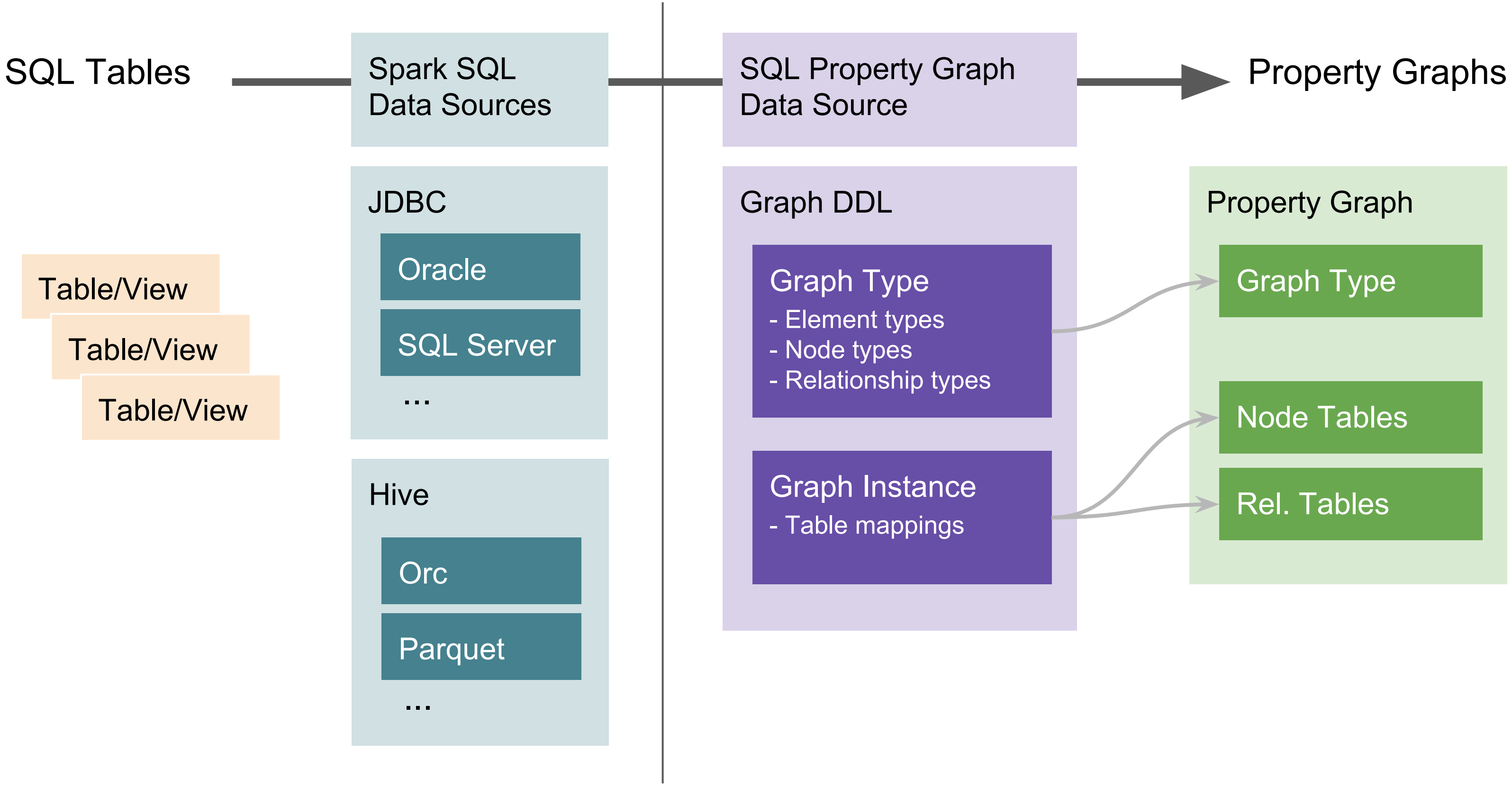
Property graph schema definition and table-to-graph mapping in CAPS

Martin Junghanns

openCypher Implementers Meeting V

Berlin, March 2019

Mapping SQL tables into a Property Graph

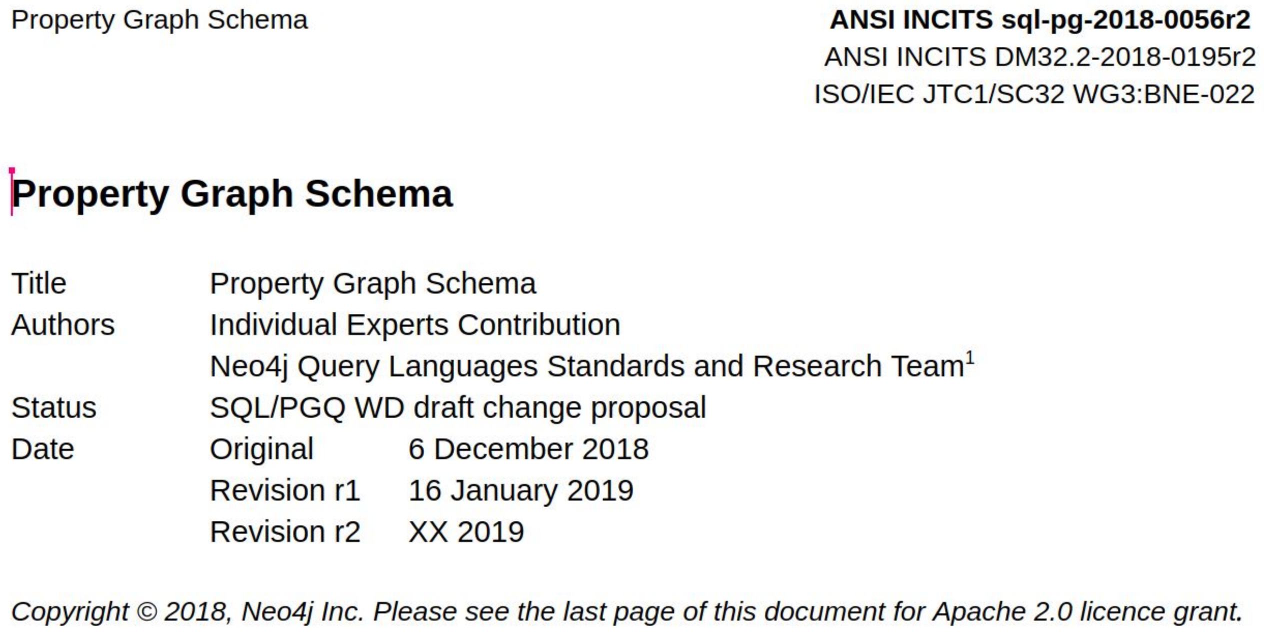


Graph Data Definition Language (DDL)

- A domain-specific language for expressing **property graph types** and **mappings** between those types and relational databases
- (Independent) Scala module within the Cypher-for-Apache-Spark project
- Provides “instructions” for the SQL Property Graph Data Source
- GitHub https://github.com/opencypher/cypher-for-apache-spark/tree/master/graph-ddl
- Maven: org.opencypher:graph-ddl:0.2.7

Graph Data Definition Language (DDL)

- Part of current a **standardization** discussion



Property Graph Schema

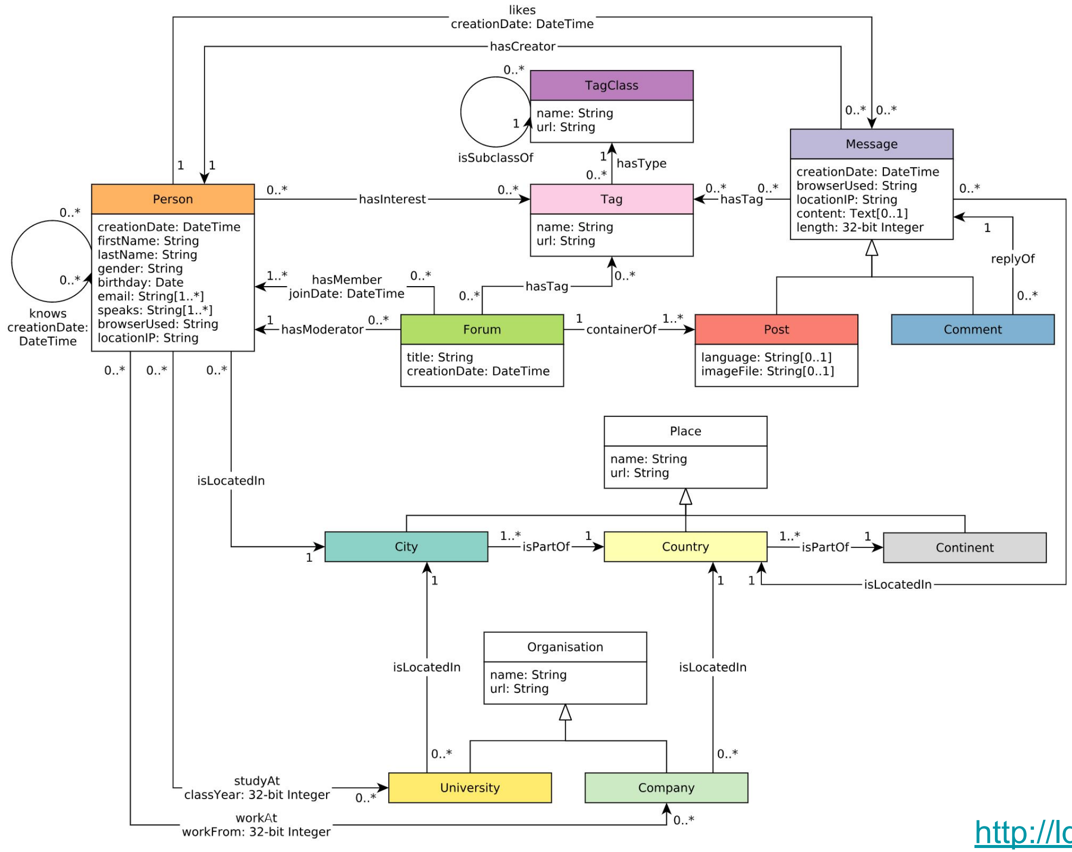
ANSI INCITS sql-pg-2018-0056r2
ANSI INCITS DM32.2-2018-0195r2
ISO/IEC JTC1/SC32 WG3:BNE-022

Property Graph Schema

Title	Property Graph Schema	
Authors	Individual Experts Contribution	
	Neo4j Query Languages Standards and Research Team[1]	
Status	SQL/PGQ WD draft change proposal	
Date	Original	6 December 2018
	Revision r1	16 January 2019
	Revision r2	XX 2019

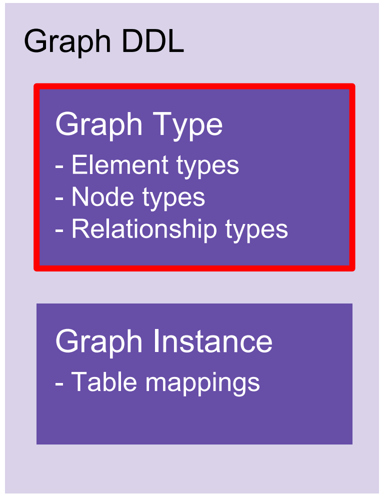
Copyright © 2018, Neo4j Inc. Please see the last page of this document for Apache 2.0 licence grant.

Running example: LDBC social network

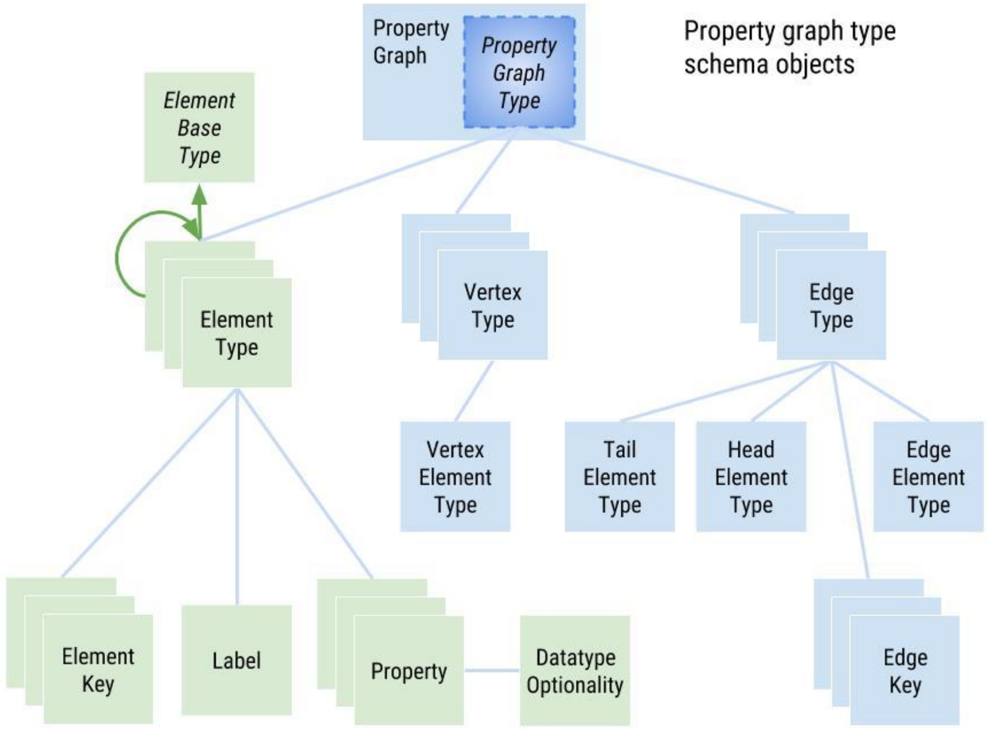


http://ldbcouncil.org/developer/snb

Graph DDL: Property graph type



Graph DDL: Property graph type



ANSI INCITS sql-pg-2018-0056r2

Element types

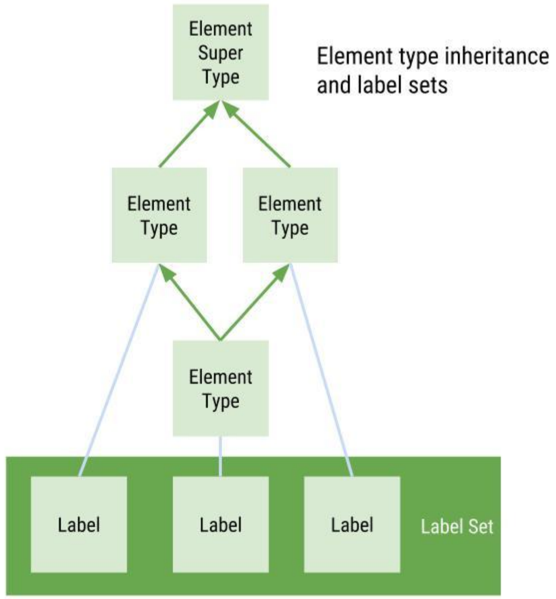
- We model the concepts / data types in our graph using element types
- Element types can have **properties** (i.e. name and data type pairs)
- They form the **basis** for node and relationship types

Name (i.e. label) Optional properties

```
Person            ( firstName STRING, lastName STRING, birthday DATE? ),

Place             ( name STRING ),

KNOWS             ( creationDate DATE ),

IS_LOCATED_IN,

...
```

Element types

- Element type support inheritance
- Similar to interface inheritance / mixin traits in programming languages



```
Place                 ( name String ),
City EXTENDS Place    ( districtCount INTEGER ),
Country EXTENDS Place ( language STRING ),
...
```

ANSI INCITS sql-pg-2018-0056r2

Node and relationship types

- We use **element types** to define a node type

```
(Person), -- resolves to label set (Person)

(City),   -- resolves to label set (City, Place)
```

- We use **two node types** and **one element type** to define a relationship type

```
(Person)-[KNOWS]->(Person),

(Person)-[IS_LOCATED_IN]->(City),
```

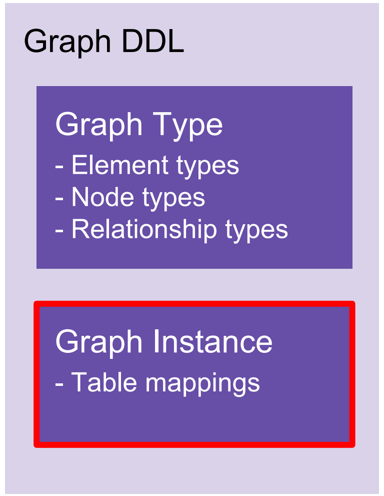
- Node / relationship types inherit all properties defined by the element types

Graph types

- All the preceding definitions are contained within a **graph type**
- A graph type is always **named** (e.g. social_network)

```
CREATE GRAPH TYPE social_network (
     Person                    ( firstName STRING, lastName String, birthday DATE? ),
     Place                     ( name STRING ),
     City EXTENDS Place        ( districtCount INTEGER ),
     Country EXTENDS Place     ( language STRING ),
     KNOWS                     ( creationDate DATE ),
     IS_LOCATED_IN,

     (Person),
     (City),
     (Country),

     (Person)-[KNOWS]->(Person),
     (Person)-[IS_LOCATED_IN]->(City),
     (City)-[IS_LOCATED_IN]->(Country)
)
```

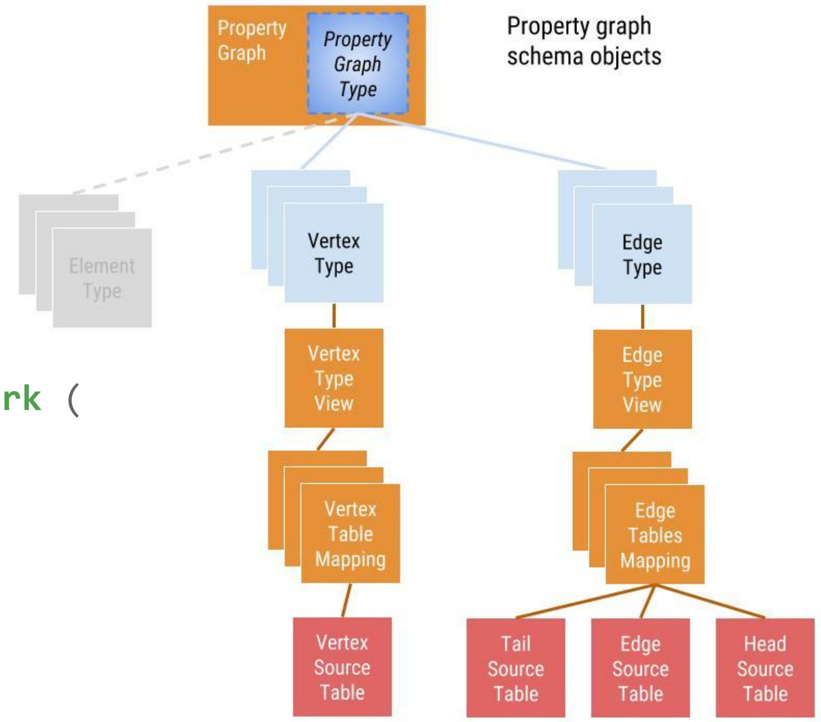
Graph DDL: Property Graph Instances



Property Graph Instances

- Graphs are instances of a **graph type**
- May define **additional element types**
- Define **node and edge type views**
- Graphs are always named

```
CREATE GRAPH social_network_US OF social_network (

    -- Additional element types

    -- Node type views / mappings

    -- Relationship type views / mappings

)
```



ANSI INCITS sql-pg-2018-0056r2

```
CREATE GRAPH social_network_US OF social_network (
    -- Node types views / mappings
                     Node source table          Optional column-property-mapping

    (Person)  FROM persons ( f_name AS firstName, l_name AS lastName ),
    (City)    FROM cities_east
              FROM cities_west,

    -- Relationship type views / mappings
                                                 Relationship source table

    (Person)-[KNOWS]->(Person) FROM person_knows_person edge
        START NODES (Person) FROM persons node JOIN node.id = edge.person1_id
        END   NODES (Person) FROM persons node JOIN edge.person2_id = node.id,

    (Person)-[IS_LOCATED_IN]->(City) FROM person_islocatedin_city edge
        START NODES (Person) FROM persons node JOIN node.id = edge.person_id
        END   NODES (City)   FROM cities  node JOIN edge.city_id = node.id
)
                     Head source table                     Tail source table

CREATE GRAPH social_network_EU OF social_network ( ... )
```

Configuring SQL data sources

```
# datasources.json

{
  "LDBC_H2" : {
    "type" :    "jdbc",
    "url" :     "jdbc:h2:mem:NORTH_AMERICA.db;INIT=CREATE SCHEMA IF NOT EXISTS NORTH_AMERICA;DB_CLOSE_DELAY=30;",
    "driver" :  "org.h2.Driver",
    "options" : {
      "user" :     "h2-user",
      "password" : "h2-password",
    }
  },
  "OTHER_DATASOURCE" : { ... }
}

# LDBC.ddl

CREATE GRAPH TYPE social_network ( ... )
SET SCHEMA        LDBC_H2.NORTH_AMERICA
CREATE GRAPH      social_network_US OF social_network ( … persons … cities … tableFoo … )
...
```

Configuring SQL data sources

```
# datasources.json

{
  "LDBC_H2" :   { ... },
  "LDBC_HIVE" : { ... }
}
```

```
# LDBC.ddl

CREATE GRAPH TYPE social_network ( ... )
SET SCHEMA        LDBC_H2.NORTH_AMERICA
CREATE GRAPH      social_network_US OF social_network (  … persons … cities … tableFoo …  )
SET SCHEMA        LDBC_HIVE.EUROPE
CREATE GRAPH      social_network_EU OF social_network (  … persons … cities … tableFoo …  )
...
```

Demo time!

https://github.com/tobias-johansson/graphddl-example-ldbc