Multiple Graphs

Updatable views & Choices

Presentation at oCIM 4 on May 22-24, 2018

Stefan Plantikow, Andrés Taylor, Petra Selmer (Neo4j)

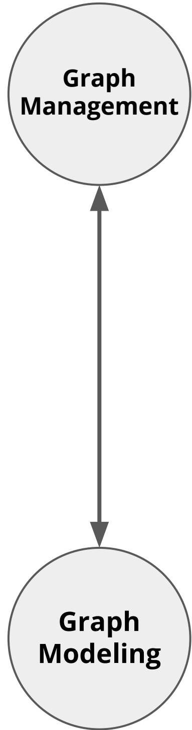
History

- Started working on multiple graphs early 2017
- Parallel work in LDBC query language task force
- Built first non-prototype proposal for oCIM 3 in Nancy (Nov 2017)
- Lots of great feedback but outcome incomplete
- Inspired by Nancy we started to reconstruct the MG proposal
  - Starting from first principles (data model, execution model, language semantics)
  - Incrementally add orthogonal concern
  - Aim for simple, minimal design
  - Reduce scope of work
- **Today: Primer for core of new proposal draft => Input to GQL**

Outline

- Multiple property graphs model
- Query processing model (Catalog and working graph)
- Working with multiple graphs (Matching, updating, returning graphs)
- Graph operations (Graph projection and graph union)
- Updating the catalog
- Examples
- Discussion (Status, timeline, next steps)

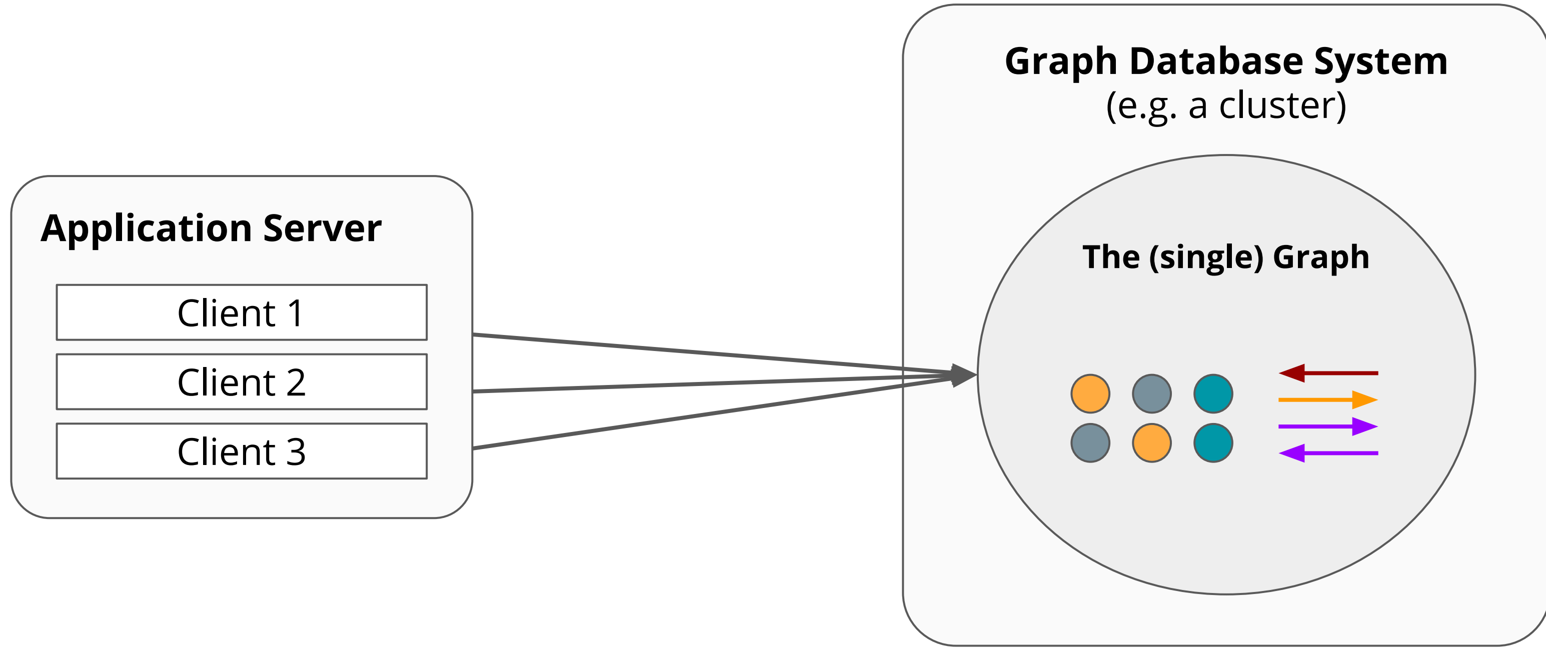
Why multiple graphs?



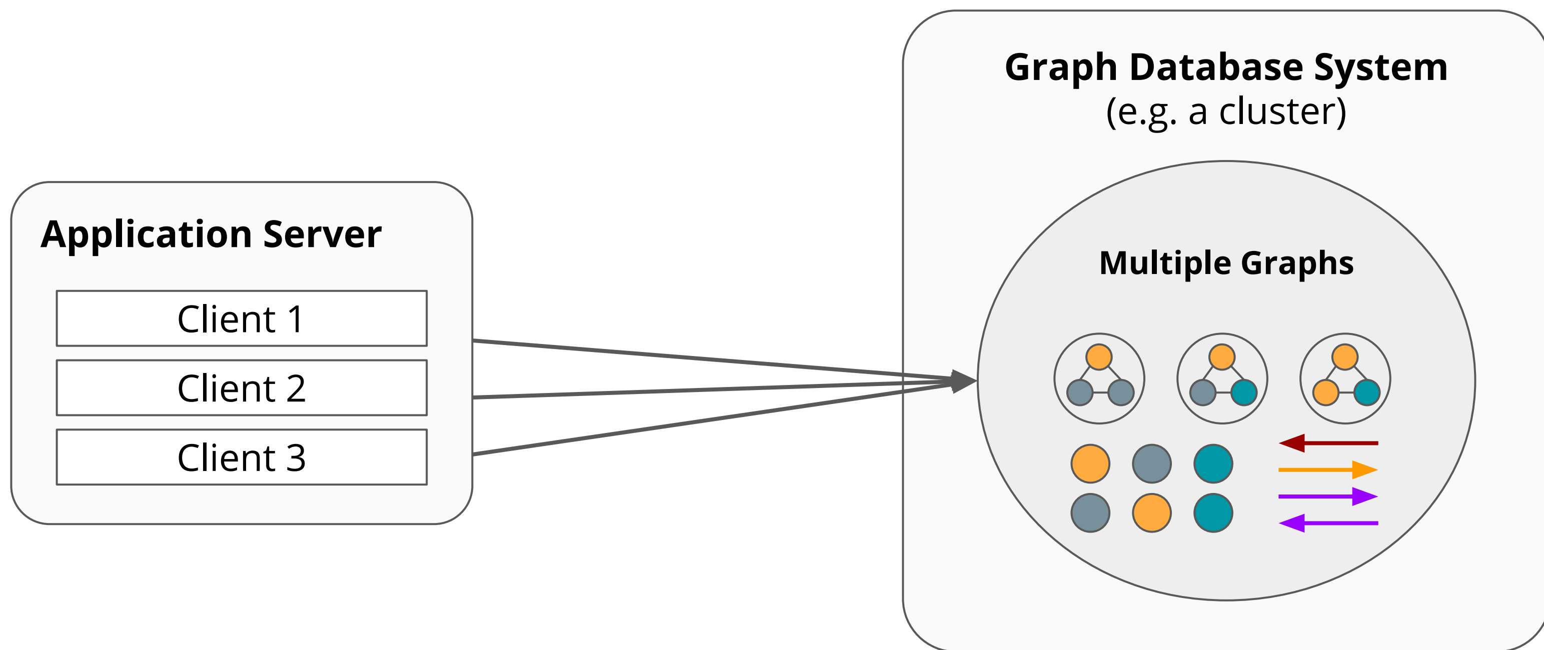
- Combining and transforming graphs from multiple sources
- Versioning, snapshotting, computing difference graphs
- Graph views for access control
- Provisioning applications with tailored graph views
- Shaping and integrating heterogeneous graph data
- Roll-up and drill-down at different levels of detail

Multiple Property Graphs Model

Cypher today: (Single) property graph model



Cypher 10: Multiple property graphs model



Single Property Graph Model

- A single graph
- Labeled nodes with properties
- Typed, directed relationships with properties

Multiple Property Graphs Model

- Multiple graphs
- Labeled nodes with properties and typed, directed relationships with properties *existing in a single graph*
- The nodes of a relationship R must be part of the same graph as R; i.e. each graph is *independent*

Graph and entity ids

- The `graph(entity)` function returns the **graph id** of the **entity**
  - An **entity** (a node or a relationship) in Cypher 10 needs to track its **graph** (to differentiate whether or not two entities are from the same graph)
  - `graph(entity)` is a value that **uniquely** identifies a **graph** within the implementation
  - This is a two-way process: a graph will also be aware of the entities it contains

- The `id(entity)` function returns the **(entity) id** of **entity** in **graph(entity)**
  - `id(entity)` is a value that **uniquely** identifies an **entity** within its graph and types (i.e. nodes and relationships do not share the id space)
  - The format of ids is implementation specific
  - This does not change between Cypher 9 and Cypher 10

Id validity and comparability

Validity of ids describes the duration for which an id is not assigned to another object.

Validity of graph and entity ids are only guaranteed for the duration of a query!

Validity of **entity ids** is only guaranteed from the matching of an entity to the consumption of a query result. In other words: for the duration of the query.
Validity, **entity ids** are guaranteed to not be re-used per graph for the duration of the query.

These are the minimal lifetime guarantees mandated by Cypher 10.
Implementations may provide extended validity guarantees as they see fit.

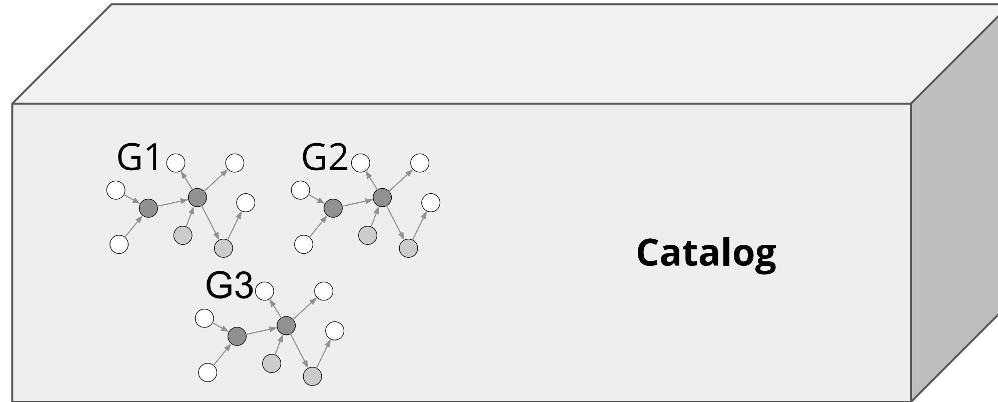
Comparability of ids

Graph and entity ids are suggested to be comparable values,
i.e. can be compared with `<, <=, =, >=, >`

Query Processing Model

Query processor

Query Processor = Query Processing Service (e.g. a single server or a cluster)



Graph catalog

- A **graph catalog** maps **graph names** to **graphs**
- A **qualified graph name** is an (optionally dotted) identifier

- A graph name may be contextual; i.e. only work inside a specific
  - server
  - database
  - ...

- There are no further restrictions placed on graphs; e.g. graphs may be
  - mutable or immutable
  - exist only for a limited time (session, transaction, ...)
  - visible to all or only some of the users of the system
  - ...

Syntax for qualified graph names

- Qualified graph name syntax follows function name syntax

- Parts
    - **graph namespace (optional)**
    - **graph name**

- Syntax
    - `foo.bar.baz`
    - `` `foo`.`bar` `` `` `foo.bar` ``
    - `` `foo.bar`.`baz.grok` `` (namespace `foo.bar`, graph `baz.grok`)
    - Number of periods can be 0 to *
    - Namespace and name separation is implementation-dependent
    - Last part is always a suffix of the graph name

Working graph

1. A clause is always operating within the context of a *working* graph
   a. by reading from it
   b. by updating it

2. During the course of executing a query
   a. the working graph may be assigned by name
      (i.e. assigned to a graph from the catalog)
   b. the working graph may be assigned by graph projection
      (i.e. assigned by creating a new working graph)
   c. the working graph may be returned as a query result
   d. the graph id of the current working graph may be obtained using the graph() function

Initial working graph

- Determined by Query Processor
- This allows for implementation freedom
  - This includes having no initial working graph
  - Queries that need an initial working graph fail if no working graph has been set

- Example
  - In Neo4j: Upon establishing a session via Bolt, an initial working graph to be used by subsequent queries is established (perhaps per user etc.)

  - In CAPS: Cypher queries are always executed explicitly against a graph which implicitly becomes the initial working graph

Working graph operations

- Assign working graph **by name** (from catalog lookup)
  - Access multiple graphs within one query
- Assign working graph **by graph projection** (next section)
  - Consolidate data from one or more data sets into one graph
- **Return (working) graph** from query
  - Construct temporary graph via multiple update queries (e.g. during interactive workflow)
  - Return (export) graphs (e.g. system graph)

Working with multiple graphs

Select working graph

```
// Set the working graph to the graph foo in the catalog for reading
FROM foo

// Set the working graph to the graph foo in the catalog for updating
UPDATE foo

// Change between reading and updating
FROM GRAPH
UPDATE GRAPH
```

In any case,

- Does not consume the current cardinality
- Retains all variable bindings

Example: Reading from multiple graphs

Which friends to invite to my next dinner party?

```
[1] FROM social_graph
[2] MATCH (a:Person {ssn: $ssn})-[:FRIEND]-(friend)
[3] FROM salary_graph
[4] MATCH (f:Entity {ssn: friend.ssn})<-[:INCOME]-(event)
[5] WHERE $startDate < event.date < $endDate
[6] RETURN friend.name, sum(event.amount) as incomes
```

Matching with multiple graphs

```
FROM persons
MATCH (a:Person)
FROM cars
MATCH (b:Car)-[:OWNED_BY]->(a) // Does NOT match
```

Updating multiple graphs

```
FROM persons
MATCH (a:Person)
UPDATE cars
CREATE (b:Car)-[:OWNED_BY]->(a) // Error
```

Handling multiple graphs: Copy patterns

How to copy data between graphs?

```
FROM persons
MATCH (a:Person)
UPDATE cars
CREATE (b:Car)-[:OWNED_BY]->(x COPY OF a) // This works
```

- `COPY OF node` Copies labels, properties
- `COPY OF rel` Copies relationship type, properties (ignores start, end!)

Also usable in matching: `MATCH (COPY OF a) ...`

Handling multiple graphs: Equivalence

How to work with data from different graphs?

```
FROM persons
MATCH (a {name: $name})
FROM people
MATCH (b {name: $name}) WHERE content(a) ~ content(b)
RETURN count(*)
```

New function for comparing by content `content` `// labels + properties`

New operator for equivalence `~` (`NULL ~ NULL` => `true`)

Returning a graph

- Cypher 10 queries may return a graph

- Three variations of consuming a graph returned by a query
    a. **Deliver graph** to client
    b. **Pass graph** to another query; e.g. within a subquery (not covered in these slides)
    c. **Store graph** in the catalog; i.e. registering a returned graph under a new name

- Delivering a **result graph** from the query to the client is always **by value**
    a. All nodes with their labels and properties
    b. All relationships with their relationship type, start node, end node, and their properties
    c. Result graph is completely independent from graphs managed by the query processor

Syntax for returning a graph

```
// Returns the working graph
...
RETURN GRAPH

// Removing the current cardinality while keeping the working graph
...
WITH GRAPH
...

// Return a graph from the catalog
FROM graphName
RETURN GRAPH
```

Graph operations

Graph operations

- Graph construction
- Common graph union
- Disjoint graph union

Graph construction

Graph construction dynamically constructs a **new working graph**

- for querying, storing in the catalog, later updating
- using entities from other graphs (this is called replication in this proposal)

Simple example

```
MATCH (a)-[:KNOWS {from: "Berlin"}]->(b)
CONSTRUCT
MERGE (a), (b) // replication, aka "shared entities"
CREATE (a)-[:MET_IN_BERLIN]->(b)
RETURN GRAPH
```

Compare and contrast

CAPS Cloning, shared entities have a different id

- precise
- too verbose, somewhat confusing for the user

G-Core Entities as references to base data but shared in views

- right model idea
- lack of updatable view semantics
- limited construction capabilities

Compare and contrast

Cypher 10/ GQL

Similar to G-Core but uses DML syntax for construction and pays attention to

- identity and equality semantics
- formulation of provenance tracking
- construction behavior
- updatable view behavior

Replicating nodes

Take an **original entity** and create a **representative replica** in the constructed graph

```
MATCH (a)
CONSTRUCT
MERGE (a)
```

Replicating the **same original** multiple times still only creates a **single replica**

```
MATCH (root)-[:PARENT_OF*]->(child)
CONSTRUCT
MERGE (root), (child)
CREATE (root)-[:ANCESTOR_OF]->(child)
...
```

Replicating relationships

Replicating a relationship r implicitly replicates `startNode(r)` and `endNode(r)`

Replication is powerful enough to reconstruct whole subgraphs!

```
FROM car_owners
MATCH (a:Person {city: "Berlin"})-[r:OWNERS]->(c:Car)
CONSTRUCT
MERGE r
// cars and their owners from berlin
RETURN GRAPH
```

Replicating paths and complex values

- Cloning a path *p* clones all nodes(*p*) and rels(*p*)
- More generally, cloning a value clones all contained entities!

```
MATCH (a:Person), (b:Person)
MATCH p = (a)-[:KNOWS*]-(b)
CONSTRUCT
MERGE a, b, p // must be variables (not any expr)!
CREATE (a)<-[:START]-(x:Path {steps: size(p)})-[:END]->(b)
RETURN GRAPH
```

Replicating graphs

Replicate all entities from given graphs into a new graph

```
CONSTRUCT                          // Sugar: CONSTRUCT ON social_graph
MERGE GRAPH social_graph
```

Replicate all entities from working graph into new graph

```
TRANSFORM                          // Alternative: CONSTRUCT ON GRAPH
```

Replicate a graph and a contained entity still only creates a **single replica** for the entity

```
MATCH (a) WITH * LIMIT 1
TRANSFORM
MERGE (a)
// creates only one a (not two) in the new graph
// by re-using replicas already constructed by TRANSFORM
```

Multi-step graph construction

So far, we've only looked at one step of graph construction. What about?

```
[1] MATCH (a)                  // single node a
[2] CONSTRUCT
[3] MERGE (a)                  // replicate node a at most once
[4] TRANSFORM
[5] MERGE (a)                  // replicate node a at most once
[6] ...
```

There is at most one replica per original. This is achieved via **provenance tracking**.

It's useful for **updatable views** and **graph union**.

Provenance tracking aka entity sharing

- **Data model**: Entities belong to one and only one graph
  `Node #1 in Graph #1`

- **Provenance graph**: Tracks entities across graph construction
  `Node #2 in Graph #2 is a replica of Node #1 in Graph #1`

- **Entity values:** References to a replica group with the same root
  `n references Node #1 in Graph #1 and all of its replicas (e.g. Node #2 in Graph #2)`

- ```
  graph(n)   - Graph of root, e.g. Graph #1
  id(n)      - id of root, e.g. #1
  a=b        - graph(a) = graph(b) && id(a) = id (b)
  ```
  ```

Updatable views

```
CONSTRUCT
...
// build the view
...
WITH GRAPH
UPDATE GRAPH
...
// update the view
...
```

Updates during graph construction

General rule: Maintain logical consistency regarding provenance tracking

CREATE	Create in the constructed graph
MERGE	Merge in the constructed graph
SET | REMOVE	Not allowed on replicas
[DETACH] DELETE	Delete from the constructed graph

Updates after graph construction

General rule: Only allow updates that can be propagated safely

`CREATE`	Not allowed
`MERGE`	Not allowed
`SET` | `REMOVE`	Update replica group
`[DETACH] DELETE`	Update all graphs

General form of graph construction

```
CONSTRUCT [ON graph1, graph2, ...]|TRANSFORM
[MERGE GRAPH [name]]
[MERGE variable, pattern, ... | * ]
<updating clause 1>
<updating clause 2>
...
WITH ... | WITH GRAPH | RETURN ... | RETURN GRAPH
```

Common graph union

Entities are always replicated

```
CONSTRUCT
...
RETURN GRAPH
UNION
CONSTRUCT
...
RETURN GRAPH
```

More graph operations: **INTERSECT**, **EXCEPT**

Disjoint graph union

All entities (including replicas) are copied (i.e. this breaks provenance tracking)

```
CONSTRUCT
...
RETURN GRAPH
UNION ALL
CONSTRUCT
...
RETURN GRAPH
```

Updating the catalog

Catalog side-effects

```
CREATE GRAPH foo   // Create new graph 'foo'
DELETE GRAPH foo   // Delete graph 'foo'

COPY foo TO bar    // Copy graph 'foo' with schema
RENAME foo TO bar // Rename graph 'foo' to 'bar'
TRUNCATE foo       // Remove data but keep schema in 'foo'


// Extensions
ALIAS foo TO bar        // Aliasing
... GRAPH foo TO bar   // Error if 'foo' is not a graph
... GRAPH TO bar        // Use working graph
```

Create new base graph (snapshot) in catalog

```
// create a new base graph in the catalog
CREATE GRAPH myGraph

// copy content of graph as new graph into the catalog
CREATE GRAPH bar { <some query that returns a graph> }
```

These are always top-level clauses; i.e. they may not be nested

Copy data from one base graph to another

We can accomplish this by using DML and hand crafting the queries

```
MATCH (a)-[r]->(b)
UPDATE foo
CREATE ...
MERGE ...
SET ...
```

global graphs & view definitions

```
CREATE GRAPH name { ... RETURN GRAPH }

CREATE QUERY name { ... RETURN GRAPH }

CREATE QUERY name(params) {
  WITH a, b
  MATCH (a)-[:X]->(b)<-[:Y]-(c {name: $param})
  RETURN GRAPH
}
```

local graphs & view definitions

```
GRAPH name1 { ... RETURN GRAPH }
QUERY name2 { ... RETURN GRAPH }
QUERY name3(params) { ... RETURN GRAPH }
FROM ...
MATCH ...
```

Schema updates

Need to select the graph

```
SETUP foo                  // Alternative: Use UPDATE
CREATE CONSTRAINT ...
CREATE INDEX ...
...
```

Catalog function

`catalog()`	Current working graph name in catalog or NULL
`catalog(graph(e))`	Name of graph of e in catalog or NULL

Nested subqueries

Anatomy of a Cypher query

```
WITH expected, variables
...
RETURN variables | RETURN GRAPH
```

Let's nest tabular subqueries

```
CALL {
  // without: input driving table ignored
  WITH expected, variables
  …
  // without: updating side-effect
  RETURN variables
}
```

Let's nest graph subqueries

```
FROM {
  WITH expected, variables
  …
  RETURN GRAPH
}
```

Let's parameterize

```
// call per grouping key $param
CALL PER foo+bar AS $param {
  WITH expected, variables
  …
  RETURN GRAPH
}
```

Summary

- Multiple graphs: select, construct, return
- Working with disjoint base graphs from the catalog: ~, content, COPY OF
- Working with updatable, overlapping, dynamically constructed graph views
- Graph operations like graph union based on provenance tracking in views
- Catalog management
- Full composition via parameterized nested subqueries

Status

- Working on three CIPs
  - Multiple graphs CIP (This talk, also: larger examples)
  - Equivalence operator CIP
  - Query composition CIP
  - Features: views, nested subqueries, equivalence, more graph operations

- Future work: More elaborate updatable views, graph aggregation

- Goal is to finalize draft of multiple graphs CIP for next oCIM or GQL draft

- Design heavily influenced by feedback from CAPS project and other implementation concerns

Feedback

Please feedback by commenting on

- slides or
- CIP or
- openCypher implementers slack

Thanks!