Non-Standard-Datenbanken und Data Mining

Graphdatenbanken

Prof. Dr. Ralf Möller
Universität zu Lübeck
Institut für Informationssysteme
Acknowledgements for slides 2-36

Graph databases and graph querying

Advances in Data Management, 2019

Dr. Petra Selmer
Query languages standards & research group, Neo4j
Property graph

Node
- Represents an entity within the graph
- Has zero or more labels
- Has zero or more properties (which may differ across nodes with the same label(s))

Edge
- Adds structure to the graph (provides semantic context for nodes)
- Has one type
- Has zero or more properties
- Relates nodes by type and direction
- Must have a start and an end node

Property
- Name-value pair (map) that can go on nodes and edges
- Represents the data: e.g. name, age, weight etc
- String key; typed value (string, number, bool, list)

Graph databases and graph querying, Petra Selmer
Relational vs. graph models
Relationship-centric querying

Query complexity grows with need for JOINs

Graph patterns not easily expressible in SQL

Recursive queries
Variable-length relationship chains
Paths cannot be returned natively
Data Integration

Graph databases and graph querying, PetraSelmer
Introducing Cypher

Declarative **graph pattern matching** language

SQL-like syntax

- **DQL** for reading data
- **DML** for creating, updating and deleting data
- **DDL** for creating constraints and indexes
Searching for (matching) graph patterns

**Nodes:**
- () or (n)
  - Surround with parentheses
  - Use an alias n to refer to our node later in the query
- (n:Label)
  - Specify a Label starting with a colon :
  - Used to group nodes by roles or types (similar to tags)
- (n:Label {prop: ‘value’})
  - Nodes can have properties

**Edges/Relationships:**
- --
  - Wrapped in hyphens and square brackets
- [:KNOWS {since: 2010}]->
  - Relationships can have properties

**Example Query:**
```
MATCH (:Person { name: "Dan" }) -[:LOVES]-> (whom)
RETURN whom
```
Cypher: patterns

Used to query data

\[(n:\text{Label} \{\text{prop:} \text{‘value’}\})-[:\text{TYPE}]->(m:\text{Label})\]

Find Alice who knows Bob In otherwords:

- find Person with the name ‘Alice’
- who KNOWS
- a Person with the name ‘Bob’

\[(p1:\text{Person} \{\text{name:} \text{‘Alice’}\})-[:\text{KNOWS}]->(p2:\text{Person} \{\text{name:} \text{‘Bob’}\})\]
DML: Creating and updating data

// Data creation and manipulation
CREATE(you:Person)
SET you.name = 'Jill Brown'
CREATE(you)-[:FRIEND]->(me)

// Either match existing entities or create new entities.
// Bind in either case
MERGE(p:Person {name: 'Bob Smith'})
  ONCREATESET p.created = timestamp(), p.updated = 0
  ONMATCHSET p.updated = p.updated + 1
RETURN p.created, p.updated
// Pattern description (ASCII art)
MATCH (me:Person)-[:FRIEND]->(friend)
// Filtering with predicates
WHERE me.name = 'Frank Black'
AND friend.age > me.age
// Projection of expressions
RETURN toUpper(friend.name) AS name, friend.title AS title
// Order results
ORDER BY name, title DESC

**Input:** a propertygraph  
**Output:** a table

Queries are graphs

Multiple pattern parts can be defined in a single match clause (i.e. conjunctive patterns); e.g:
MATCH (a)-(b)-(c), (b)-(f)
Cypher patterns

Node patterns

MATCH(), (node), (node:Node), (:Node), (node {type:"NODE"})

Relationship patterns

MATCH()-->(), ()<--(), ()--() // Single relationship
MATCH()-[edge]->(), (a)-[edge]->(b) // With binding
MATCH()[:RELATES]->() // With specific relationship type
MATCH()-[edge {score:5}]->() // With property predicate
MATCH()-[r:LIKES|:EATS]->() // Union of relationship types
MATCH()-[r:LIKES|:EATS {age: 1}]->() // Union with property predicate
(axplies to all relationship types specified)
Cypher patterns

Variable-length relationship patterns

MATCH(me)-[:FRIEND*]-(foaf)  // Traverse 1 or more FRIEND relationships
MATCH(me)-[:FRIEND*2..4]-(foaf)  // Traverse 2 to 4 FRIEND relationships
MATCH(me)-[:FRIEND*0..]-(foaf)  // Traverse 0 or more FRIEND relationships
MATCH(me)-[:FRIEND*2]-(foaf)  // Traverse 2 FRIEND relationships
MATCH(me)-[:LIKES|HATES*]-(foaf)  // Traverse union of LIKES and HATES1 or more times

// Path binding returns all paths (p)
MATCH p = (a)-[:ONE]-()-[:TWO]-()-[:THREE]-()
// Each path is a list containing the constituent nodes and relationships, in order
RETURN p

// Variation: return all constituent nodes of the path
RETURN nodes(p)
// Variation: return all constituent relationships of the path
RETURN relationships(p)
Cypher: linear composition and aggregation

1: `MATCH (me:Person {name: $name})-[[:FRIEND]]-(friend)`  
2: `WITH me, count(friend) AS friends`  
3: `MATCH (me)-[:ENEMY]-(enemy)`  
4: `RETURN friends, count(enemy) AS enemies`

**WITH** provides a horizon, allowing a query to be subdivided:
- Further matching can be done after a set of updates
- Expressions can be evaluated, along with aggregations
- Essentially acts like the pipe operator in Unix

**Linear composition**
- Query processing begins at the top and progresses linearly to the end
- Each clause is a function taking in a table \( T \) (line 1) and returning a table \( T' \)
- \( T' \) then acts as a driving table to the next clause (line 3)
Example query: epidemic!

Assume a graph $G$ containing doctors who have potentially been infected with a virus....
Example query

The following Cypher query returns the name of each doctor in G who has perhaps been exposed to some source of a viral infection, the number of exposures, and the number of people known (both directly and indirectly) to their colleagues.

1. `MATCH (d:Doctor)
2.  OPTIONAL MATCH (d)-[:EXPOSED_TO]->(v:ViralInfection)
3.  WITH d, count(v) AS exposures
4.  MATCH (d)-[:WORKED_WITH]->(colleague:Person)
5.  OPTIONAL MATCH (colleague)<[:KNOWS*]-(p:Person)
6.  RETURN d.name, exposures, count(DISTINCT p) AS thirdPartyCount
Example query

1: \textbf{MATCH} (d:Doctor)
2: \textbf{OPTIONAL MATCH} (d)-[:EXPOSED_TO]->(v:ViralInfection)

Matches all :Doctors, along with whether or not they have been :EXPOSED_TO a :ViralInfection
\textbf{OPTIONAL MATCH} analogous to outer join in SQL
Produce rows provided entire pattern is found
If no matches, a single row is produced in which the binding for v is null

<table>
<thead>
<tr>
<th>d</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>SourceX</td>
</tr>
<tr>
<td>Sue</td>
<td>PatientY</td>
</tr>
<tr>
<td>Alice</td>
<td>SourceX</td>
</tr>
<tr>
<td>Bob</td>
<td>\textit{null}</td>
</tr>
</tbody>
</table>

Although we show the \textit{name} property (for ease of exposition), it is actually the \textit{node} that gets bound
3: \textbf{WITH} \texttt{d, count(v) AS exposures}

\textbf{WITH} projects a subset of the variables in scope - \texttt{d} - and their bindings onwards (to 4).
\textbf{WITH} also computes an aggregation:

- \texttt{d} is used as the grouping key implicitly (as it is not aggregated) for \texttt{count()}
- All non-null values of \texttt{v} are counted for each unique binding of \texttt{d}
- Aliased as \textit{exposures}

The variable \texttt{v} is no longer in scope after 3

<table>
<thead>
<tr>
<th></th>
<th>exposures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>2</td>
</tr>
<tr>
<td>Alice</td>
<td>1</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
</tr>
</tbody>
</table>

This binding table is now the driving table for the \textbf{MATCH} in 4
Example query

4: MATCH(d)-[:WORKED_WITH]->(colleague:Person)

Uses as driving table the binding table from 3
Finds all the colleagues (:Person) who have :WORKED_WITH our doctors

<table>
<thead>
<tr>
<th>d</th>
<th>exposures</th>
<th>colleague</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>2</td>
<td>Chad</td>
</tr>
<tr>
<td>Sue</td>
<td>2</td>
<td>Carol</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
</tr>
</tbody>
</table>
5: **OPTIONAL MATCH** (colleague)<-[[:KNOWS*]-(p:Person)

Finds all the people (:Person) who :KNOW our doctors’ colleagues (only in the one direction), both directly and indirectly (using :KNOWS* so that one or more relationships are traversed)

<table>
<thead>
<tr>
<th>d</th>
<th>exposures</th>
<th>colleague</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>2</td>
<td>Chad</td>
<td>Carol</td>
</tr>
<tr>
<td>Sue</td>
<td>2</td>
<td>Carol</td>
<td><em>null</em></td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
<td>Will</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
<td>Chad</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
<td>Carol*</td>
</tr>
<tr>
<td>Bob</td>
<td>0</td>
<td>Sally</td>
<td>Carol*</td>
</tr>
</tbody>
</table>

No (Carol)<-[[:KNOWS]-()] pattern in G

* This is due to the :KNOWS* pattern: Carol is reachable from Sally via Chad and Will (Carol :KNOWS Will and Chad)
Example query results

1: MATCH (d:Doctor)
2: OPTIONAL MATCH (d)-[:EXPOSED_TO]->(v:ViralInfection)
3: WITH d, count(v) AS exposures
4: MATCH (d)-[:WORKED_WITH]->(colleague:Person)
5: OPTIONAL MATCH (colleague)<-[:KNOWS*]-(p:Person)
6: RETURN d.name, exposures, count(DISTINCT p) AS thirdPartyCount

+--------+--------+-----------------+
| d.name | exposures | thirdPartyCount |
+--------+--------+-----------------+
| Bob    | 0      | 3 (Will, Chad, Carol) |
| Sue    | 2      | 1 (Carol)         |
+--------+--------+-----------------+
Other functionality

Aggregating functions

\[ \text{count, max, min, avg, ...} \]

Operators

Mathematical, comparison, string-specific, boolean, list

Map projections

Construct a map projection from nodes, relationships and properties

\text{CASE} expressions

Functions (scalar, list, mathematical, string, UDF, procedures)
Property graphs are everywhere

Many implementations

Amazon Neptune, Oracle PGX, Neo4j Server, SAP HANA Graph, AgensGraph (over PostgreSQL), Azure CosmosDB, Redis Graph, SQL Server 2017 Graph, Cypher for Apache Spark, Cypher for Gremlin, SQL Property Graph Querying, TigerGraph, Memgraph, JanusGraph, DSE Graph, ...

Multiple languages

| ISO SC32.WG3 | SQLPGQ (Property Graph Querying) |
| Neo4j        | openCypher                      |
| LDBC         | G-CORE (augmented with paths)   |
| Oracle       | PGQL                            |
| W3C          | SPARQL (RDF data model)         |
| Tigergraph   | GSQL                            |

...also imperative and analytics-based languages

SQL 2020
Participation from major DBMS vendors. Neo4j’s contributions freely available*.

* [http://www.opencypher.org/references#sql-pg](http://www.opencypher.org/references#sql-pg)
Graph Query Language (GQL)

A new stand-alone / native query language for graphs

Targets the labelled PG model

Composable graph query language with support for updating data

Based on

- “Ascii art” pattern matching
- Published formal semantics (Cypher, G-CORE)
- SQLPG extensions and SQL-compatible foundations (some data types, some functions, ...)

https://www.gqlstandards.org

GQL Documents also available at: http://www.opencypher.org/references#sql-pg

Graphs first, not graphs “extra”
Example GQL Query

//from graph or view ‘friends’ in the catalog
FROM friends

//match persons ‘a’ and ‘b’ who travelled together
MATCH (a:Person)-[:TRAVELLED_TOGETHER]-(b:Person)
WHERE a.age = b.age
   AND a.country = $country
   AND b.country = $country

//from view parameterized by country
FROM census($country)

//find out if ‘a’ and ‘b’ at some point moved to or were born in a place ‘p’
MATCH (a)-[:BORN_IN|MOVED_TO*]-(p)<[:BORN_IN|MOVED_TO*]-(b)

//that is located in a city ‘c’
MATCH (p)-[:LOCATED_IN]->(c:City)

//aggregate the number of such pairs per city and age group
RETURN a.age AS age, c.name AS city, count(*) AS num_pairs
GROUP BY age

Illustrative syntax only!

Regular path queries
Complex path patterns

Regular path queries (RPQs)

\[ X, (\text{likes.hates})^*(\text{eats|drinks})^+, Y \]

Find a path whose edge labels conform to the regular expression, starting at node \( X \) and ending at node \( Y \)

(X and Y are node bindings)

Plenty of research in this area since 1987!

SPARQL 1.1 has support for RPQs: “property paths”

I. F. Cruz, A. O. Mendelzon, and P. T. Wood
A graphical query language supporting recursion
Complex paths in the property graph data model

Property graph data model:

- Properties need to be considered
- Node labels need to be considered

Specifying a cost for paths (ordering and comparing)

Path patterns (e.g., GXPATH)

L. Libkin, W. Martens, and D. Vrgoč
Querying Graphs with Data
Composition of Path Patterns

Sequence / Concatenation: \((\cdots) -/ \alpha \beta / -()\)

Alternation / Disjunction: \((\cdots) -/ \alpha \mid \beta / -()\)

Transitive closure:

- 1 or more \((\cdots) -/ \alpha^* / -()\)
- 2 or more \((\cdots) -/ \alpha^+ / -()\)
- n or more \((\cdots) -/ \alpha^n.. / -()\)
- At least n, at most m \((\cdots) -/ \alpha^n..m / -()\)

Overriding direction for sub-pattern:

- Left to right direction \((\cdots) -/ \alpha > / -()\)
- Right to left direction \((\cdots) -/ < \alpha / -()\)
- Any direction \((\cdots) -/ < \alpha> / -()\)
Path Pattern: example

```
PATH PATTERN
older_friends = (a)-[:FRIEND]-(b) WHERE b.age > a.age
MATCH p=(me)-/\~older_friends+/-((you)
WHERE me.name = $myName AND you.name = $yourName
RETURN p AS friendship
```
PATH PATTERN
older_friends = (a)-[:FRIEND]-(b) \textbf{WHERE} b.age > a.age

PATH PATTERN
same_city = (a)-[:LIVES_IN]->(:City)<-[:LIVES_IN]-(b)

PATH PATTERN
older_friends_in_same_city = (a)/~older_friends/~(b)

\textbf{WHERE EXISTS} \{ (a)/~same_city/~(b) \}
Cost function for cheapest path search

PATH PATTERN road = (a)-[r:ROAD_SEGMENT]-(b) COST r.length

MATCH route = (start)-/~road*/-(end)

WHERE start.location = $currentLocation
   AND end.name = $destination

RETURN route

ORDER BY cost(route) ASC LIMIT 3
“Cyphermorphism”

Pattern matching today uses **edge isomorphism** (no repeated relationships)

MATCH (p:Person {name: 'Jack'})-[r1:FRIEND]-(node)-[r2:FRIEND]-(friend_of_a_friend)
RETURN friend_of_a_friend.name AS fofName

**Rationale** was to avoid potentially returning infinite results for varlength patterns when matching graphs containing cycles (this would have been different if we were just checking for the existence of a path).

- **r1** and **r2** may not be bound to the same relationship **within the same pattern**

Usefulness proven **in practice** over multiple industrial verticals: we have not seen any worst-case examples.
Graph projection

Sharing elements in the projected graph
Deriving new elements in the projected graph
Shared edges always point to the same (shared) endpoints in the projected graph
Projection is the inverse of pattern matching

Graph databases and graph querying, PetraSelmer
Queries are composable procedures

- Use the output of one query as input to another to enable abstraction and views
- Applies to queries with tabular output and graph output
- Support for nested subqueries
- Extract parts of a query to a view for re-use
- Replace parts of a query without affecting other parts
- Build complex workflows programmatically
Implications

Pass both multiple graphs and tabular data into a query

Return both multiple graphs and tabular data from a query

Select which graph to query

Construct new graphs from existing graphs

based on slide by S. Plantikow
Acknowledgements for slides 38-48

- Slides are taken from the following Presentation
  - Emerging Graph Queries in Linked Data
    - Arijit Khan, Yinghui Wu, Xifeng Yan
    - Department of Computer Science
    - University of California, Santa Barbara

- All errors are mine
Graph Search Queries

- **Containment Query**
- **Similarity Query**
- **Matching Query**

Retrieves all graphs from a graph database, such that they **contain** a given query graph (exact and approximate).
Graph Search Queries

- **Containment Query**
- **Similarity Query**
- **Matching Query**

Retrieves all graphs from a graph database, that are similar to the query graph (exact and approximate).
Graph Search Queries

- **Containment Query**
- **Similarity Query**
- **Matching Query**

Find all occurrences of a query graph in a large target network (exact and approximate).
Containment Query

Subgraph Isomorphism Problem is \textbf{NP-hard}.

Filtering and Verification

\textbf{Filtering Phase:}
Feature-based index is used to filter out the negative results and generate candidate sets.

\textbf{Verification Phase:}
Precise Subgraph Isomorphism Testing to generate final results from the candidate set.
Similarity Query

Graph Isomorphism is **neither known to be Polynomial or NP-Complete**

Graph Edit Distance **NP-hard**

**Maximum Common Subgraph (MCS) based approach.**

\[
\Delta = |d(Q, \text{MCS}(Q,G_1))| + |d(G_1, \text{MCS}(Q,G_2))| = 4
\]

\[
|d(G_2, \text{MCS}(Q,G_2))| = 10
\]

\[
|d(G_1, \text{MCS}(Q,G_1))| = 10
\]

Efficiently Finding MCS of two large networks (Approximate) 
- Zhu et al., CIKM ’11

Indexing based on MCS in Filtering Phase 
- Zhu et al., EDBT ’12
Kernel Based Approach.

Measure similarity of two graphs by comparing their substructures.

Map two graphs $G_1$ and $G_2$ via mapping $\varphi$ into feature space $H$.

$\varphi \equiv \text{length of all walks between every ordered pair of labels.}$

\[
\begin{align*}
\varphi_{(c, a)} &= \varphi_{(a, r)} = \varphi_{(r, t)} = 1 \\
\varphi_{(a, t)} &= 1 + 2 = 3 \\
\varphi_{(c, t)} &= 2 + 3 = 5 \\
\varphi_{(c, c)} &= 0 \text{ etc.}
\end{align*}
\]

Measure their similarity in $H$ as scalar product $<\varphi(G_1), \varphi(G_2)>$.

**Kernel Trick:** Compute inner product in $H$ as kernel in input space $k(G_1, G_2) = <\varphi(G_1), \varphi(G_2)>$; e.g., compute walks in the product graph $G_1 \times G_2$.

- Positive Definite.
**Complete Graph Kernel:** Let $k(G_1, G_2) = \langle \phi(G_1), \phi(G_2) \rangle$ be a graph kernel. If $\phi$ is injective, $k$ is called a complete graph kernel.

**Example:** The graph kernel that has one feature $\Phi_H$ for each possible graph $H$, each feature $\Phi_H(G)$ measuring how many subgraphs of $G$ have the same structure as graph $H$.

The above example of Complete Graph Kernel is NP-hard.

**Theorem:** Computing any complete graph kernel is at least as hard as deciding whether two graphs are isomorphic [Gärtner et. al., COLT ’03]
Graph Kernels

**Polynomial Time Computable Graph Kernels:**

- **Random Walk** - Kashima et al., *ICML ’03*
  - Gaertner et al., *COLT ’03*
  - Mahe et al., *ICML ’04*
  - Vishwanathan et al., *NIPS ’06*

- **Shortest Path** - Borgwardt et. al., *ICDM ’05*

- **Optimal assignment kernel** - Froehlich et al, *ICML ’05*
  [NOT Positive definite, Vert, ’08]

- **Weighted Decomposition Kernel** - Menchetti et al., *ICML ’05*

- **Edit-Distance Kernel** - Neuhaus et. al., *SSPR/SPR ’06*

- **Subtree Kernel** - Ramon et. al., *Mining Graphs, Trees and Sequences ’04*
  - Shervashidze et. al., *NIPS ’09*

- **Cyclic Pattern Kernel** - Horvath et al., *KDD ’04*

- **Neighborhood Kernel** - Wang et. al., *EDBT ’09*
Graph Pattern Mining

Given a graph dataset $D$, find all subgraphs $g$, s.t. \[ \text{freq}(g) \geq \theta \]
Where $\text{freq}(g)$ is the (relative) number of graphs that contain $g$.

$\Theta = 3$

Emerging Graph Queries in Linked Data, Arijit Khan, Yinghui Wu, Xifeng Yan
Why Mine Graph Patterns?

**Direct Use:**

- Mining over-represented sub-structures in chemical databases.
- Mining conserved sub-networks.
- Program control flow analysis.

**Indirect Uses:**

- Index the data graph and query graph using local features.
- Building block of further analysis, i.e., Classification, Clustering, Similarity Searches, Indexing
Why is Graph Mining Hard?

**Apriori Property**

If a graph is frequent, all of its subgraphs are frequent.

Pattern Search Tree
Summary

• Graph database language
  – Cypher and others → GQL

• Hardness results

• Implementation issues
  – Containment and matching
    • Indexing / filtering (still false positive, no false negatives)
    • Verification (eliminate false positives)
  – Similarity
    • Mapping into feature space with polynomial graph kernels

• Graph mining