IN3020/4020 – Database Systems
Spring 2021, Week 18.1

Labeled Property Graphs (Neo4j)

Egor V. Kostylev (with M. Naci Akkøk)

Based upon slides by D. Roman from Spring 2019

(Labelled) graph databases

- For data that is natural to describe and traverse as graphs
  - Each node has an inner structure describing its properties
  - The edges indicate relationships between the nodes
  - The edges can carry information in the same way as the nodes

- Can be schema-free
  - New nodes and edges (with new inner structures) can be introduced dynamically
  - Existing nodes and edges can be expanded with new properties

- Search: Specify how the graph should be navigated
  - The graph is traversed directly via pointers to neighboring nodes (the traverse requires no indexes and no join operations)

Neo4j

https://neo4j.com

- A native graph database
- “Whiteboard friendly”
- Schemaless – no need to define any structure in advance
- Query language: *Cypher*
  - Declarative, pattern-based
- Transaction support
- Scalability (support for clusters)
- Examples use: eBay, HP
- Open-source under GPL

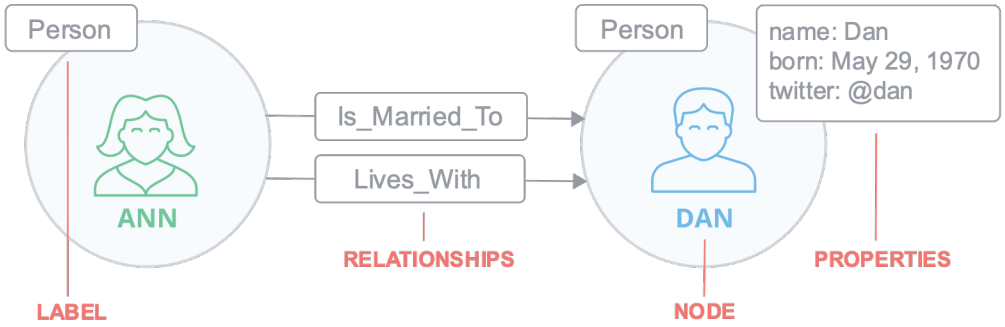
Getting started with Neo4j

- Easiest to get started is via the Neo4j Sandbox: https://neo4j.com/sandbox

- Alternatively, download and install locally the Neo4j desktop: https://neo4j.com/download/neo4j-desktop/

Data model

- **Node**
- **Label**: A type to a node
  - A node can have none or several labels
- **Relationship**:
  - Directed edge between two nodes
  - Each two nodes have have several relations
- **Relationship type**: Used to characterize a relationship
  - Each relationship has exactly one type
- **Property**: Key-value pair
  - Both nodes and relations can have properties
  - The name is a character string
  - The value is taken from a base datatype (int, char,…) or an array over a base datatype (int[], char[],…)



Cypher Graph patterns

https://neo4j.com/docs/cypher-manual

- The strength of the property graph lies in its ability to encode patterns of connected nodes and relationships

- Cypher is strongly based on patterns
  - Patterns are used to match desired graph structures
  - A simple pattern, with a single relationship, connects a pair of nodes

    a Person LIVES_IN *a City*
  - Complex patterns, using multiple relationships, can express arbitrarily complex concepts

    a Person LIVES_IN *a City is* PART_OF *a Country*

- Cypher represents graph-related patterns using clauses and keywords, for example MATCH, WHERE and DELETE are used to combine patterns and specify desired actions

Node syntax

- Nodes are represented using a pair of parentheses, e.g.: (), (foo)

```
()
(matrix)
(:Movie)
(matrix:Movie)
(matrix:Movie {title: "The Matrix"})
(matrix:Movie {title: "The Matrix", released: 1997})
```

Relationship syntax

- Undirected relationship uses a pair of dashes (--)
- Directed relationships have an arrowhead at one end (<--, -->)
- Bracketed expressions ([…]) can be used to add details

```
-->
-[role]->
-[:ACTED_IN]->
-[role:ACTED_IN]->
-[role:ACTED_IN {roles: ["Neo"]}]->
```

Pattern syntax

- Patterns are expressed by combining the syntax for nodes and relationships

```
(keanu:Person:Actor {name:  "Keanu Reeves"} )
-[role:ACTED_IN     {roles: ["Neo"] } ]->
(matrix:Movie       {title: "The Matrix"} )
```

Clauses

- Cypher statements typically have multiple **clauses**, each of which performs a specific task, for example:
  - Create and match patterns in the graph
  - Filter, project, sort, or paginate results
  - Compose partial statements

Creating data

- The simplest clause is **CREATE**

```
CREATE (:Movie { title:"The Matrix",released:1997 })
```

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 2
Labels added: 1
```

- To return the created data the **RETURN** clause is used (refers to the variable assigned to the pattern elements)

```
CREATE (p:Person { name:"Keanu Reeves", born:1964 })
RETURN p
```

```
+----------------------------------------+
| p                                      |
+----------------------------------------+
| Node[1]{name:"Keanu Reeves",born:1964} |
+----------------------------------------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

Creating data: more complex structures

- We can create more complex structures

```
CREATE (a:Person { name:"Tom Hanks",
  born:1956 })-[r:ACTED_IN { roles: ["Forrest"]}]->(m:Movie { title:"Forrest Gump",released:1994 })
CREATE (d:Person { name:"Robert Zemeckis", born:1951 })-[:DIRECTED]->(m)
RETURN a,d,r,m
```

- But in most cases, we want to connect new data to existing structures.

  This requires that we know how to find existing patterns in our graph data, which we will look at next.

Matching patterns

- Matching patterns is done using the **MATCH** statement, by passing the patterns describing what to look for
- A MATCH statement will search for the specified patterns and return one row per successful pattern match

```
MATCH (p:Person { name:"Tom Hanks" })-[r:ACTED_IN]->(m:Movie)
RETURN m.title, r.roles
```

```
+-----------------------------+
| m.title        | r.roles    |
+-----------------------------+
| "Forrest Gump" | ["Forrest"] |
+-----------------------------+
1 row
```

- It is possible to attach structures to the graph by combining MATCH and CREATE

```
MATCH (p:Person { name:"Tom Hanks" })
CREATE (m:Movie { title:"Cloud Atlas",released:2012 })
CREATE (p)-[r:ACTED_IN { roles: ['Zachry']}]->(m)
RETURN p,r,m
```

Completing patterns

- MERGE checks for the existence of data first before creating it
  - Define a pattern to be found or created (can provide additional properties to set ON CREATE)

```
MERGE (m:Movie { title:"Cloud Atlas" })
ON CREATE SET m.released = 2012
RETURN m
```

- MERGE can also assert that a relationship is only created once

```
MATCH (m:Movie { title:"Cloud Atlas" })
MATCH (p:Person { name:"Tom Hanks" })
MERGE (p)-[r:ACTED_IN]->(m)
ON CREATE SET r.roles =['Zachry']
RETURN p,r,m
```

Filtering results

- Filter conditions are expressed in a WHERE clause
  - Allows to use any number of Boolean expressions combined with AND, OR, XOR and NOT

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ "K.+" OR m.released > 2000 OR "Neo" IN r.roles
RETURN p,r,m
```

```
MATCH (p:Person)-[:ACTED_IN]->(m)
WHERE NOT (p)-[:DIRECTED]->()
RETURN p,m
```

Returning results

- The RETURN clause can return not only nodes and relations, but also expressions
- Simple expressions:
  - Values of Keys from Key-Value pairs: numbers, strings, arrays, etc.
  - Function evaluations: length(array), toInteger("12"), etc.
- Can be composed and concatenated to form more complex expressions
- Can use "expression AS alias" to improve readability
- To indicate unique results the DISTINCT keyword is used after RETURN

```
MATCH (p:Person)
RETURN p, p.name AS name, toUpper(p.name), coalesce(p.nickname,"n/a") AS nickname, { name: p.name,
  label:head(labels(p))} AS person
```

Aggregating information

- Aggregation happens in the RETURN clause while computing the final results
  - Many common aggregation functions are supported, e.g. count, sum, avg, min, and max

```
MATCH (:Person)
RETURN count(*) AS people
```

```
+--------+
| people |
+--------+
| 3      |
+--------+
1 row
```

Ordering and pagination, collecting aggregation

- Ordering works with the clause
  ORDER BY expression [ASC|DESC]
- Pagination works with the clause
  SKIP {offset} LIMIT {count}

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
RETURN a,count(*) AS appearances
ORDER BY appearances DESC LIMIT 10;
```

- A very helpful aggregation function is collect(): it collects all aggregated values into a list

```
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title AS movie, collect(a.name) AS cast, count(*) AS actors
```

Composing large statements

- UNION is used to combine the results of two statements that have the same result structure

```
MATCH (actor:Person)-[r:ACTED_IN]->(movie:Movie)
RETURN actor.name AS name, type(r) AS acted_in, movie.title AS title
UNION
MATCH (director:Person)-[r:DIRECTED]->(movie:Movie)
RETURN director.name AS name, type(r) AS acted_in, movie.title AS title
```

- WITH is used to combine fragments of statements and declare which data flows from one to the other
  - WITH is very much like RETURN with the difference that it doesn't finish a query but prepares the input for the next part (the only difference is that one must alias all columns as they would otherwise not be accessible)

```
MATCH (person:Person)-[:ACTED_IN]->(m:Movie)
WITH person, count(*) AS appearances, collect(m.title) AS movies
WHERE appearances > 1
RETURN person.name, appearances, movies
```

Constraints and indexes

- Constraints are used to guarantee uniqueness of a certain property on nodes with a specific label

```
CREATE CONSTRAINT ON (movie:Movie) ASSERT movie.title IS UNIQUE
```

- Indexes are mainly used to find the starting point in the graph as fast as possible

```
CREATE INDEX ON :Actor(name)
```

Removing and modifying data

- DELETE is used to delete nodes, relationships or paths
- DETACH DELETE is used to delete a node and any relationship going to or from it

```
MATCH (n:Person { name: 'UNKNOWN' })
DELETE n
```

```
MATCH (n { name: 'Andy' })
DETACH DELETE n
```

```
MATCH (n { name: 'Andy' })-[r:KNOWS]->()
DELETE r
```

- REMOVE is used to remove properties from nodes and relationships, and to remove labels from nodes

```
MATCH (a { name: 'Andy' })
REMOVE a.age
RETURN a.name, a.age
```

```
MATCH (n { name: 'Peter' })
REMOVE n:German
RETURN n.name, labels(n)
```

- SET clause is used to update labels on nodes and properties on nodes and relationships

```
MATCH (n { name: 'Andy' })
SET n.surname = 'Taylor'
RETURN n.name, n.surname
```

```
MATCH (n { name: 'George' })
SET n:Swedish:Bossman
RETURN n.name, labels(n) AS labels
```

Importing CSV files using LOAD CSV

persons.csv.

```
id,name
1,Charlie Sheen
2,Oliver Stone
3,Michael Douglas
4,Martin Sheen
5,Morgan Freeman
```

```
LOAD CSV WITH HEADERS FROM "https://neo4j.com/docs/developer-manual/3.4/csv/import/persons.csv" AS csvLine
CREATE (p:Person { id: toInteger(csvLine.id), name: csvLine.name })
```

movies.csv.

```
id,title,country,year
1,Wall Street,USA,1987
2,The American President,USA,1995
3,The Shawshank Redemption,USA,1994
```

```
CREATE INDEX ON :Country(name)
```

```
LOAD CSV WITH HEADERS FROM "https://neo4j.com/docs/developer-manual/3.4/csv/import/movies.csv" AS csvLine
MERGE (country:Country { name: csvLine.country })
CREATE (movie:Movie { id: toInteger(csvLine.id), title: csvLine.title, year:toInteger(csvLine.year)})
CREATE (movie)-[:MADE_IN]->(country)
```

roles.csv.

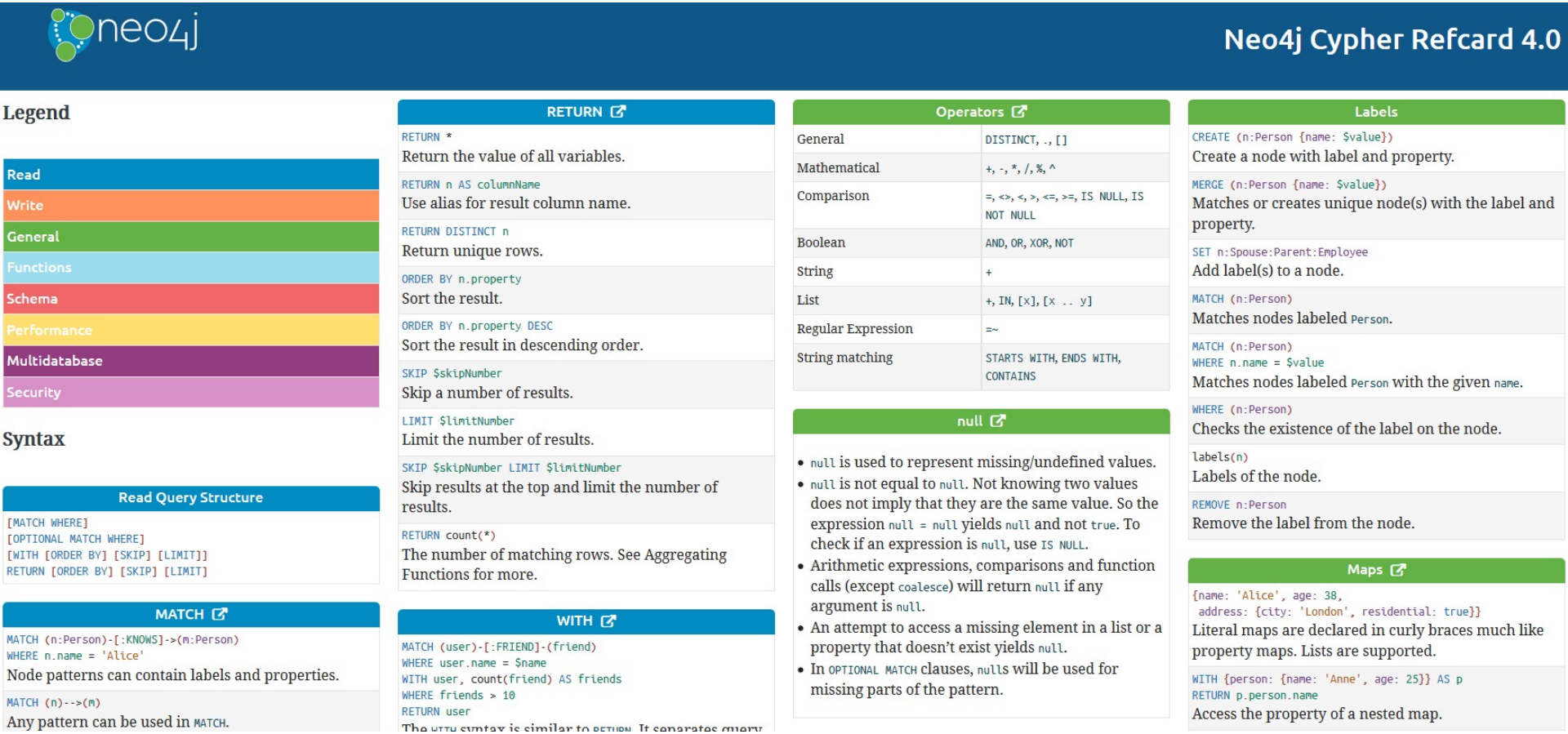
```
personId,movieId,role
1,1,Bud Fox
4,1,Carl Fox
3,1,Gordon Gekko
4,2,A.J. MacInerney
3,2,President Andrew Shepherd
5,3,Ellis Boyd 'Red' Redding
```

```
CREATE CONSTRAINT ON (person:Person) ASSERT person.id IS UNIQUE
CREATE CONSTRAINT ON (movie:Movie) ASSERT movie.id IS UNIQUE
```

```
USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "https://neo4j.com/docs/developer-manual/3.4/csv/import/roles.csv" AS csvLine
MATCH (person:Person { id: toInteger(csvLine.personId)}),
  (movie:Movie { id: toInteger(csvLine.movieId)})
CREATE (person)-[:PLAYED { role: csvLine.role }]->(movie)
```

Neo4j Cypher Refcard

https://neo4j.com/docs/cypher-refcard



neo4j

Neo4j Cypher Refcard 4.0

Legend

Read
Write
General
Functions
Schema
Performance
Multidatabase
Security

Syntax

Read Query Structure

```
[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

MATCH

```
MATCH (n:Person)-[:KNOWS]->(m:Person)
WHERE n.name = 'Alice'
```
Node patterns can contain labels and properties.

```
MATCH (n)-->(m)
```
Any pattern can be used in `MATCH`.

RETURN

```
RETURN *
```
Return the value of all variables.

```
RETURN n AS columnName
```
Use alias for result column name.

```
RETURN DISTINCT n
```
Return unique rows.

```
ORDER BY n.property
```
Sort the result.

```
ORDER BY n.property DESC
```
Sort the result in descending order.

```
SKIP $skipNumber
```
Skip a number of results.

```
LIMIT $limitNumber
```
Limit the number of results.

```
SKIP $skipNumber LIMIT $limitNumber
```
Skip results at the top and limit the number of results.

```
RETURN count(*)
```
The number of matching rows. See Aggregating Functions for more.

WITH

```
MATCH (user)-[:FRIEND]-(friend)
WHERE user.name = $name
WITH user, count(friend) AS friends
WHERE friends > 10
RETURN user
```
The `WITH` syntax is similar to `RETURN`. It separates query

Operators

General	`DISTINCT, ., []`
Mathematical	`+, -, *, /, %, ^`
Comparison	`=, <>, <, >, <=, >=, IS NULL, IS NOT NULL`
Boolean	`AND, OR, XOR, NOT`
String	`+`
List	`+, IN, [x], [x .. y]`
Regular Expression	`=~`
String matching	`STARTS WITH, ENDS WITH, CONTAINS`

null

- `null` is used to represent missing/undefined values.
- `null` is not equal to `null`. Not knowing two values does not imply that they are the same value. So the expression `null = null` yields `null` and not `true`. To check if an expression is `null`, use `IS NULL`.
- Arithmetic expressions, comparisons and function calls (except `coalesce`) will return `null` if any argument is `null`.
- An attempt to access a missing element in a list or a property that doesn't exist yields `null`.
- In `OPTIONAL MATCH` clauses, `null`s will be used for missing parts of the pattern.

Labels

```
CREATE (n:Person {name: $value})
```
Create a node with label and property.

```
MERGE (n:Person {name: $value})
```
Matches or creates unique node(s) with the label and property.

```
SET n:Spouse:Parent:Employee
```
Add label(s) to a node.

```
MATCH (n:Person)
```
Matches nodes labeled `Person`.

```
MATCH (n:Person)
WHERE n.name = $value
```
Matches nodes labeled `Person` with the given `name`.

```
WHERE (n:Person)
```
Checks the existence of the label on the node.

```
labels(n)
```
Labels of the node.

```
REMOVE n:Person
```
Remove the label from the node.

Maps

```
{name: 'Alice', age: 38,
 address: {city: 'London', residential: true}}
```
Literal maps are declared in curly braces much like property maps. Lists are supported.

```
WITH {person: {name: 'Anne', age: 25}} AS p
RETURN p.person.name
```
Access the property of a nested map.

Examples: Creation of nodes and edges

```
CREATE
  (a:Person {name:"Ann", born:1997}),
  (b:Person {name:"John", birthdate:191148}),
  (c:Person {name:"Carl", status: "married",
             interests:["ski", "diving"],
             email:"carl@gmail.com"}),
  (a)-[:RELATIVE {type:"daugther", status:"adopted"}]->(b),
  (a)-[:RELATIVE {type:"niece"}]->(c)


MATCH (x:Person {name:"John"}), (y:Person {name:"Carl"})
CREATE (x)-[r:RELATIVE {type:"brother"}]->(y)
RETURN r
```

Examples: Creation of nodes and edges (cont')

- Relatives of relatives of Ann:

```
MATCH (p:Person {name:"Ann"})-[:RELATIVE]-(s1),
      (s1)-[:RELATIVE]-(s2)
RETURN s2
```

- Common relatives of Ann and Carl:

```
MATCH (pers1)-[:RELATIVE]-(rel),
      (pers2)-[:RELATIVE]-(rel)
WHERE pers1.name = "Ann" AND pers2.name = "Carl"
RETURN rel
```

- (Undirected) Shortest path between Hilde and Geir (at most 5 relationships):

```
MATCH (p1:Person {name:"Hilde"}), (p2:Person {name:"Geir"}),
      path = shortestPath((p1)-[*..5]-(p2))
RETURN path
```

- Number of relatives (when the direction on the relationship is important):

```
MATCH (a:Person)-[:RELATIVE]->(b:Person)
RETURN a.name, count(*)
ORDER BY count(*) DESC
```

Graph model versus other data models

- Graph model vs. Relational model:
  - Traversing a graph is much cheaper than joins; uses direct pointers to neighboring nodes
  - Workload is shifted from query execution to data insertion and maintenance
  - “Dynamic” schema make it simpler to use for not-experts

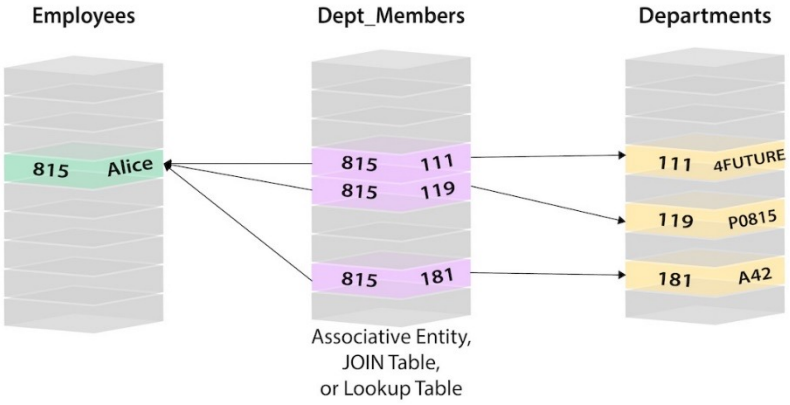
Signs of managing highly-connected data with a relational database

- Large number of JOINs
- Numerous self-JOINs (or recursive JOINs)
- Frequent schema changes
- Slow-running queries (despite extensive tuning)
- Pre-computing the results

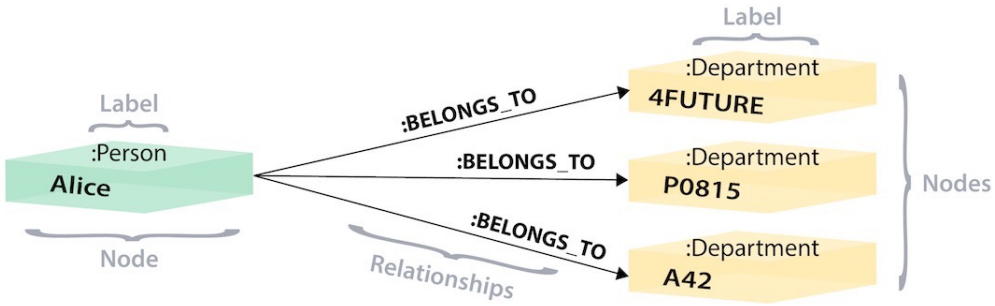
From relational to graph (Neo4j)

https://neo4j.com/developer/graph-db-vs-rdbms/

Relational



Graph



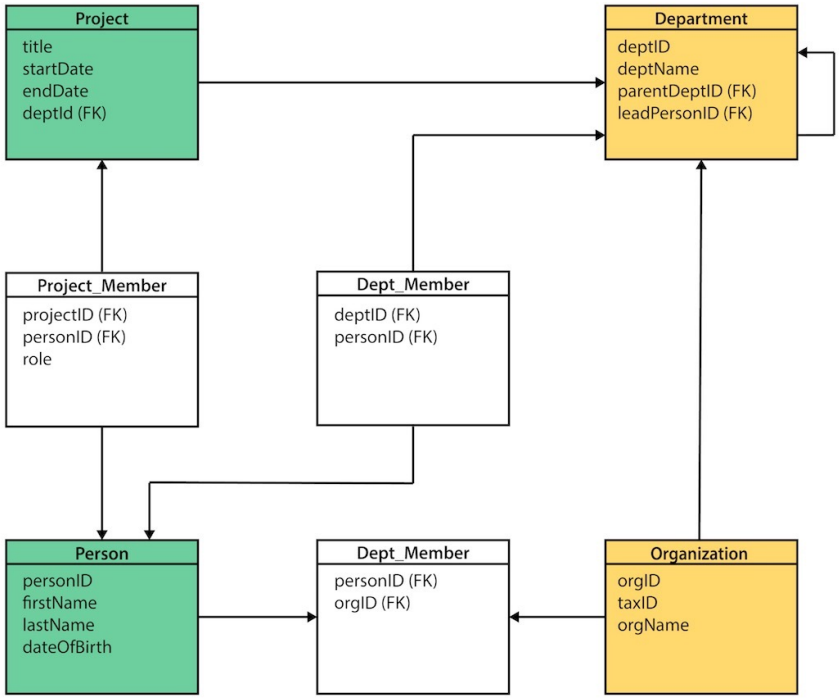
From relational to graph (Neo4j)
Tips for data model transformation

https://neo4j.com/developer/relational-to-graph-modeling

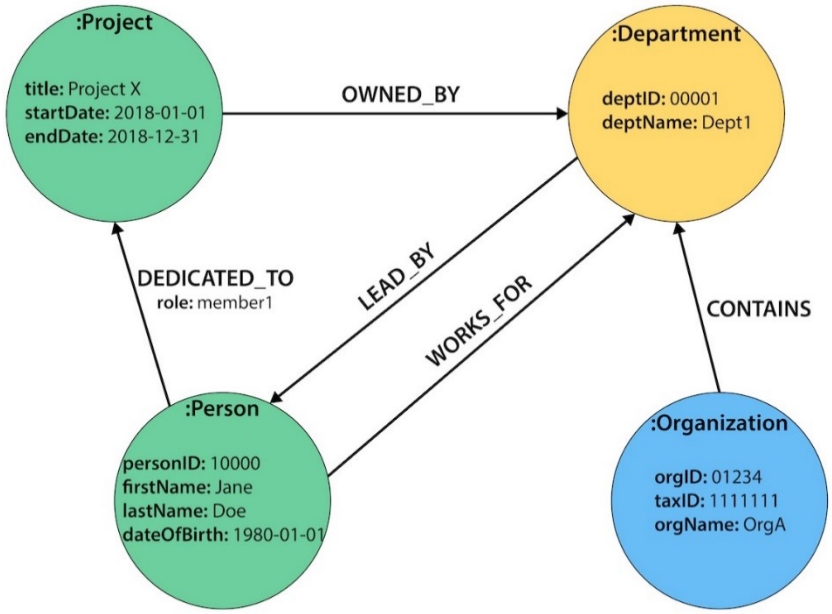
- **Table to Node Label** – each entity table in the relational model becomes a label on nodes in the graph model
- **Row to Node** – each row in a relational entity table becomes a node in the graph
- **Column to Node Property** – columns (fields) on the relational tables become node properties in the graph
- **Business primary keys only** – remove technical primary keys, keep business primary keys
- **Add Constraints/Indexes** – add unique constraints for business primary keys, add indexes for frequent lookup attributes
- **Foreign keys to Relationships** – replace foreign keys to the other table with relationships, remove them afterwards
- **No defaults** – remove data with default values, no need to store those
- **Clean up data** – duplicate data in denormalized tables might have to be pulled out into separate nodes to get a cleaner model
- **Index Columns to Array** – indexed column names (like email1, email2, email3) might indicate an array property
- **Join tables to Relationships** – join tables are transformed into relationships, columns on those tables become relationship properties

From relational to graph (Neo4j) Example

https://neo4j.com/developer/relational-to-graph-modeling



Relational (ER)



Graph (Neo4j)

From relational to graph (Neo4j) Example Query

https://neo4j.com/developer/graph-db-vs-rdbms

- Retrieve the employees in the “IT Department”

SQL

```
SELECT firstName, lastName FROM Person
LEFT JOIN Dept_Member
  ON Person.personId = Dept_Member.personId
LEFT JOIN Department
  ON Department.deptId = Dept_Member.deptId
WHERE Department.deptName = "IT Department"
```

Cypher

```
MATCH (p:Person)<-[:WORKS_FOR]-(d:Department)
WHERE d.name = "IT Department"
RETURN p.firstName,p.lastName
```

From relational to graph (Neo4j)
A More Extreme Query Example

https://neo4j.com/blog/sql-vs-cypher-query-languages

Cypher

```
MATCH (u:Customer {customer_id:'customer-one'})-
[:BOUGHT]->(p:Product)<- [:BOUGHT]-(peer:Customer)-
[:BOUGHT]->(reco:Product)

WHERE not (u)-[:BOUGHT]->(reco)

RETURN reco as Recommendation, count(*) as Frequency

ORDER BY Frequency DESC LIMIT 5;
```

SQL

```
SELECT product.product_name as Recommendation, count(1) as Frequency
FROM product, customer_product_mapping, (SELECT cpm3.product_id,
cpm3.customer_id
      FROM Customer_product_mapping cpm, Customer_product_mapping cpm2,
      Customer_product_mapping cpm3
      WHERE cpm.customer_id = 'customer-one'
      and cpm.product_id = cpm2.product_id
      and cpm2.customer_id != 'customer-one'
      and cpm3.customer_id = cpm2.customer_id
      and cpm3.product_id not in (select distinct product_id
      FROM Customer_product_mapping cpm
      WHERE cpm.customer_id = 'customer-one')
      ) recommended_products
WHERE customer_product_mapping.product_id = product.product_id
and customer_product_mapping.product_id in recommended_products.product_id
and customer_product_mapping.customer_id = recommended_products.customer_id
GROUP BY product.product_name
ORDER BY Frequency desc
```

Graph Algorithms

- Used to compute metrics for graphs, nodes, or relationships
- Provide insights on relevant entities in the graph (centralities, ranking), or inherent structures like communities (community-detection, graph-partitioning, clustering)
- Many of the approaches have high algorithmic complexity
  - Iterative approaches that frequently traverse the graph for the computation using random walks, breadth-first or depth-first searches, or pattern matching
  - Optimized algorithms utilize certain structures of the graph, recall already explored parts, and parallelize operations

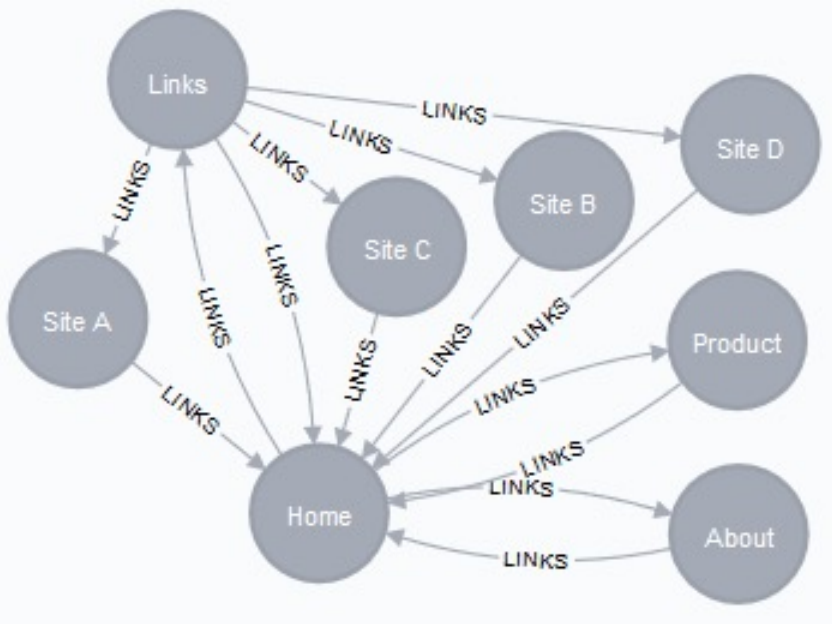
Neo4j graph algorithms

https://neo4j.com/docs/graph-data-science/current/algorithms

- **Centralities**: determine the importance of distinct nodes in a network (PageRank, Betweenness Centrality, Closeness Centrality)
- **Community detection**: evaluate how a group is clustered or partitioned, as well as its tendency to strengthen or break apart (Louvain, Label Propagation, Connected Components, Connected Components, Triangle Count / Clustering Coefficient)
- **Path finding**: find the shortest path or evaluate the availability and quality of routes (Minimum Weight Spanning Tree, All Pairs- and Single Source - Shortest Path, A* Algorithm, Yen's K-Shortest Paths, Random Walk)

Example: PageRank

- Measures the **transitive** influence or connectivity of nodes

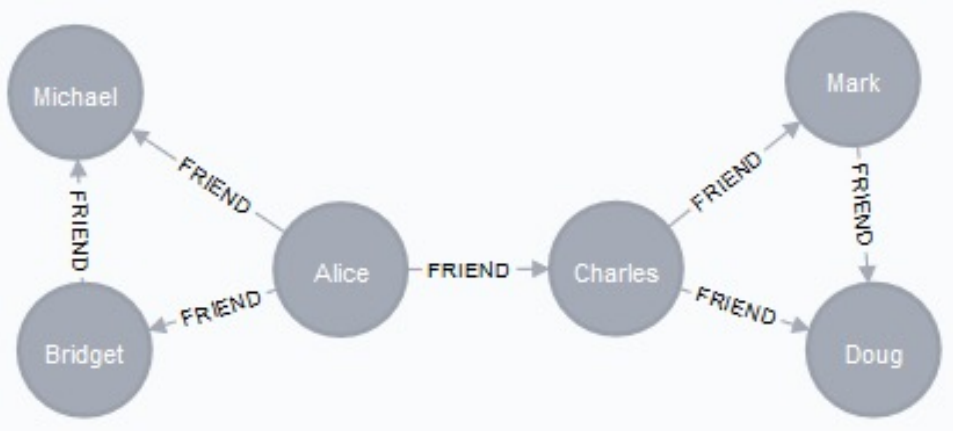


page	score
"Home"	3.236201617214829
"Product"	1.0611098274122923
"Links"	1.0611098274122923
"About"	1.0611098274122923
"Site A"	0.32922589540248737
"Site B"	0.32922589540248737
"Site C"	0.32922589540248737
"Site D"	0.32922589540248737

```
CALL algo.pageRank.stream('Page', 'LINKS', {iterations:20, dampingFactor:0.85})
YIELD nodeId, score

MATCH (node) WHERE id(node) = nodeId

RETURN node.name AS page,score
ORDER BY score DESC
```

Example: Louvain

- Used for detecting communities in networks
- Evaluates how much more densely connected the nodes within a community are, compared to how connected they would be in a random network



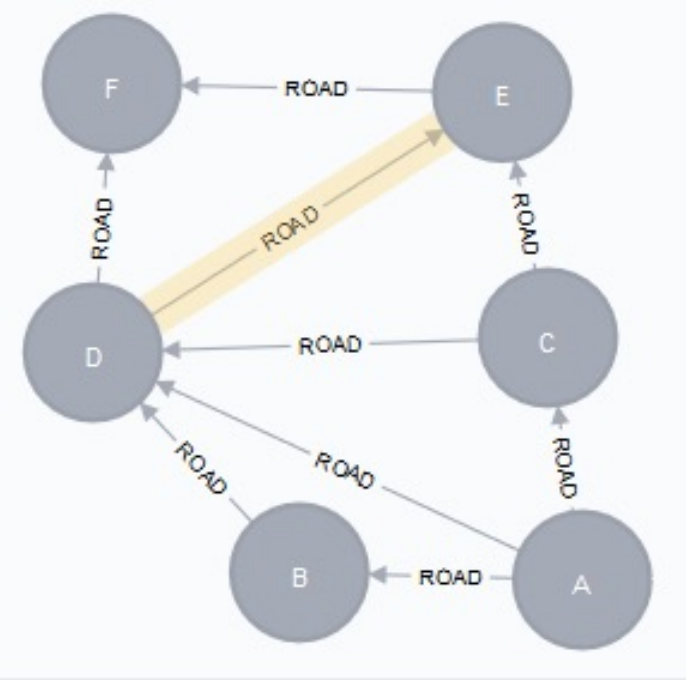

```
CALL algo.louvain.stream('User', 'FRIEND', {})
YIELD nodeId, community

MATCH (user:User) WHERE id(user) = nodeId

RETURN user.id AS user, community
ORDER BY community;
```

user	community
"Alice"	0
"Bridget"	0
"Michael"	0
"Charles"	1
"Doug"	1
"Mark"	1

Example: Shortest Path

- Calculates the shortest (weighted) path between a pair of nodes (Dijkstra's algorithm is the most well known)

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.shortestPath.stream(start, end, 'cost')
YIELD nodeId, cost
MATCH (other:Loc) WHERE id(other) = nodeId
RETURN other.name AS name, cost
```



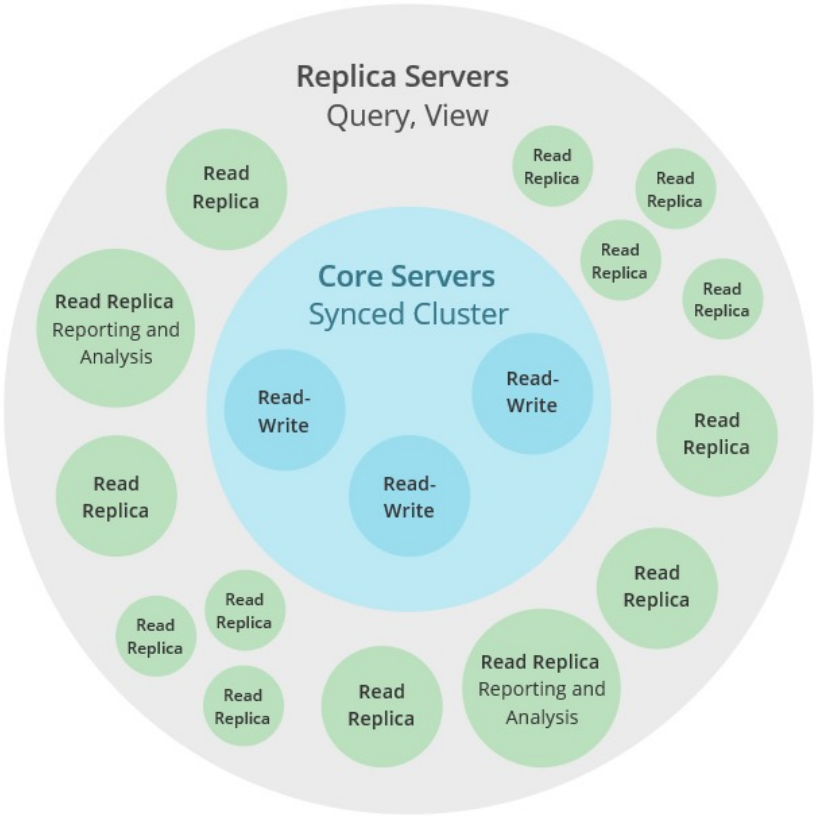
ROAD <id>: 6 cost: 30

name	cost
"A"	0.0
"B"	50.0
"D"	90.0
"E"	120.0
"F"	160.0

Replication in Neo4j

https://neo4j.com/docs/operations-manual/current/clustering/

- *Causal Clustering* architecture
  - Cores Replicas
  - Read Replicas

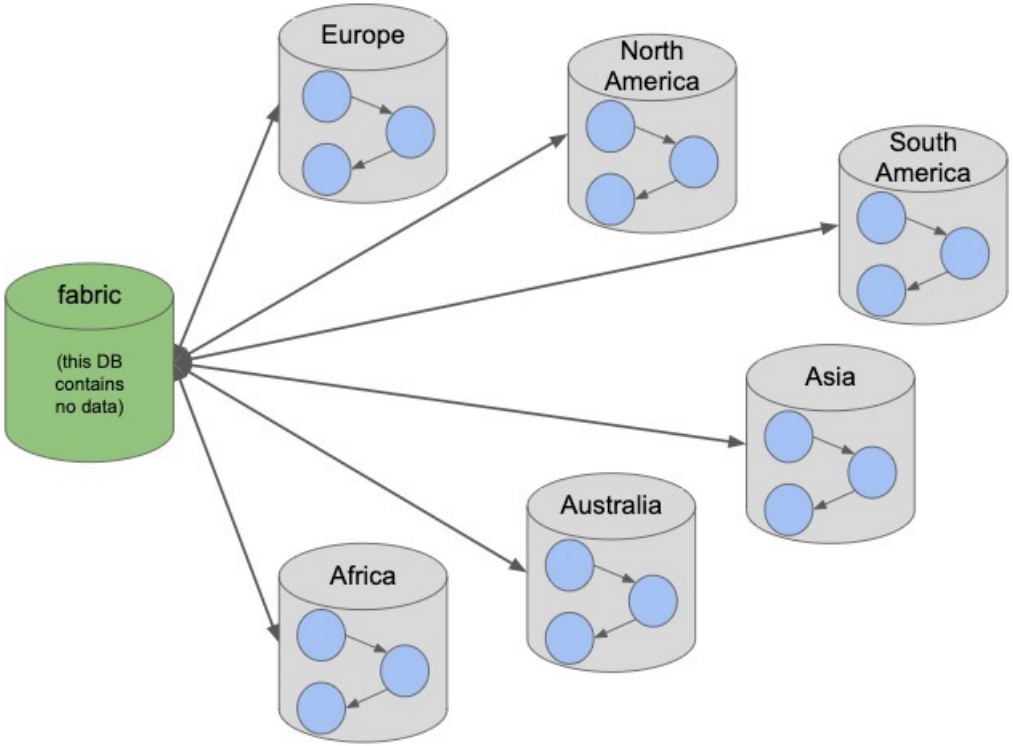


- *Safety*: Core Replicas provide a fault tolerant platform for transaction processing
- *Scale*: Read Replicas provide a scalable platform for graph queries
- *Causal consistency*: when invoked, a client application is guaranteed to read at least its own writes.

Sharding in Neo4j

https://neo4j.com/docs/operations-manual/current/fabric/

- The more data is connected, the more complicated it is to shard
- *Neo4j Fabric*: allows users to split a larger graph down into individual, smaller graphs and store them in separate databases
- The fabric database is a virtual database



Neo4j references

- Neo4j: https://neo4j.com
- Neo4j Cypher Refcard: https://neo4j.com/docs/cypher-refcard/current
- Neo4j documentation: https://neo4j.com/docs
  - Getting started: https://neo4j.com/docs/getting-started/current
  - Cypher Manual: https://neo4j.com/docs/cypher-manual/current
  - RDBMS to Graph: https://neo4j.com/developer/get-started/graph-db-vs-rdbms
  - Graph Data Science (incl graph algorithms): https://neo4j.com/docs/graph-data-science

Polyglot persistence

- Polyglot persistence: a variety of different database systems for different kinds of data

Picture taken from https://martinfowler.com/bliki/PolyglotPersistence.html

- Complexity cost
  - Each data storage mechanism introduces a new interface to be learned for each new data storage mechanism
  - Storage is usually a performance bottleneck
  - Multiple data silos
  - More complicated deployment, more frequent upgrades
  - Data consistency and duplication issues

Multi-model databases

- A database that consists of different data storage mechanisms (e.g. relational, document, key/value, graph database):
  - All in one database engine
  - With a unifying query language and API
  - That cover all data models and even allow for mixing them in a single query

- Next evolution of NoSQL technologies

- Multi-model vs Multi-modal
  - Multi-model: relational, key-value, document, graph, tree, etc.
  - Multi-modal: video, audio, image, text, etc.

Examples

- ArangoDB – document (JSON), graph, key-value
- Cosmos DB – document, table, key-value, JSON, SQL
- CouchBase – relational (SQL), document
- CrateDB – relational (SQL), document (Lucene)
- MarkLogic – document (XML and JSON), graph (RDF with OWL/RDFS), text, geospatial, binary, SQL
- OrientDB – document (JSON), graph, key-value, text, geospatial, binary, reactive, SQL
- Datastax – key-value, tabular, graph
- …

Hot topics in multi-model databases

- Benchmarking
- Extensions of existing query languages
- Cross-model schema languages and evolution
- Query processing
  - Cross-model complex joins
  - New index structures
- Model mapping
- Cross-model transaction and consistency