**Neo4j**

**Graph Modeling**

We usually fine **nodes** and **labels** by looking for **nouns**.

We can often find **relationships** by identifying **actions** or **verbs**.

**Convert relational to graph**

We don’t need associative entity tables or table joins any more.

Table to Node Label.

Row to Node.

Column to Node properties.

Business primary keys and no technical primary keys.

Foreign keys to relationships.

Remove data with default values, no need to store them.

Duplicate data in denormalized tables might have to be separated into new nodes.

Join tables to relationships and columns on those tables become relationship properties. Note that direction of the relationship is based on the where the FK resides.

**Cypher**

Nodes and relationships are the simple component that build -> **the Pattern.**

Patterns can express simple or complex **traversals** and **paths**.

Relationship variables -> if you forget the **colon** in front of a relationship like this –[LIKES->, it represents a variable (not a relationship type). Cypher searches all types of relationships since no type is declared.

**Matching patterns** -> what you are looking for? A match statement searches for the patterns you specify and return one row per successful pattern match.

**MATCH and CREATE** -> Subsequent CREATE statements are executed once for each row that are returned by matched pattern.

**MATCH or CREATE ->** MERGE

? MERGE -> if you choose to pass in only one node from preceding clause, MERGE offers an interesting functionality. It only matches within the **direct neighborhood** of the provided node for the given pattern, and if the pattern is not found it creates it. This can come in very handy for creating, for example, tree structures.

CREATE(y:Year{year: 2014})

**MERGE(y)**<-[:IN\_YEAR]-(m10:Month{month: 10})

**MERGE(y)**<-[:IN\_YEAR]-(m11:Month{month: 11})

RETURN y, m10, m11

**Pattern Predicate** -> it restricts the current result set. It only allows the paths to pass that satisfy the specified pattern.

MATCH (p:Person)-[:ACTED\_IN]->(m)

WHERE NOT (p)-[:DIRECTED]->()

RETURN p, m

**Filtering on patterns** -> filter results based on relationships or patterns. This allows you to test if a pattern also has a certain relationship or does not exist or another pattern exists.

//find which people are friends of someone who works for Neo4j

MATCH (p:Person)-[r:IS\_FRIENDS\_WITH]->(friend:Person)

WHERE exists((p)-[:WORKS\_FOR]->(:Company {name: 'Neo4j'}))

RETURN p, r, friend;

**OPTIONAL MATCH** -> acts like **outer join** in sql functions.

//Find all people whose name starts with J and who may work for a company.

MATCH (p:Person)

WHERE p.name STARTS WITH 'J'

OPTIONAL MATCH (p)-[:WORKS\_FOR]-(other:Company)

RETURN p.name, other.name;

**Chain** and **link** patterns ->

//find who likes graphs besides Jennifer that she is also friends with her

MATCH (j:Person {name: 'Jennifer'})-[:LIKES]->(:Technology {type: 'Graphs'})<-[:LIKES]-(p:Person), (j)-[:IS\_FRIENDS\_WITH]-(p)

RETURN p.name;

//Or

MATCH (j:Person {name: 'Jennifer'})-[:LIKES]->(:Technology {type: 'Graphs'})<-[:LIKES]-(p:Person)

MATCH (j)-[:IS\_FRIENDS\_WITH]-(p)

RETURN p.name;

**Return** clause -> can return any number of expressions like **Predicates** used in WHERE clauses (they are count are Boolean expressions).

**Aggregation** -> aggregation happens in the **RETURN clause** while computing final results. Aggregation works implicitly in Cypher. You specify which result columns you wish to aggregate. Cypher uses **all non-aggregated column** as grouping keys. Aggregation **removes** the **visibility** of data that is aggregated.

**COLLECT(expression)** -> returns a single aggregated list of the values. no information of details is lost while aggregating. It is well-suited for retrieving typical parent-child structures.

**Ordering** -> you can order by things that are part of returned information. Be careful with **aggregation** and **DISTINCT** return values, since both remove the visibility of data that is aggregated.

//for a list of techRequirements, look for people who have each skill

WITH ['Graphs','Query Languages'] AS techRequirements

UNWIND techRequirements AS technology

MATCH (p:Person)-[r:LIKES]-(t:Technology {type: technology})

WITH t.type AS technology, p.name AS personName

ORDER BY technology, personName

RETURN technology, collect(personName) AS potentialCandidates;

Notice that the first query has to order by Person name before collecting the values into a list. If you do not sort first (put the ORDER BY after the RETURN clause), you will sort based on the size of the list and not by the first letter of the values in the list.

To find a number of **relationship patterns** -> use the **COUNT{}** expression.

Using MERGE on a relationship -> the following statement will create duplicate nodes for Mark and Jennifer:

MERGE (j:Person {name: 'Jennifer'})-[r:IS\_FRIENDS\_WITH]->(m:Person {name: 'Mark'})

RETURN j, r, m

**WITH** -> It is similar to the RETURN clause. expressions, aggregations, ordering and pagination can be used in the same way as in the RETURN clause. The only difference is all columns must be aliased.

**Subqueries->** a set of Cypher statements execute within their **own scope**

EXISTS subquery expression

COUNT subquery expression

CALL { … } subquery clause

CALL { … } IN TRANSACTIONS subquery clause

. subqueries can be use to filter out results.

. subqueries can be use to return results as well.

//find people who work for a company whose name starts with 'Company' and who like at least one technology that’s liked by three or more people. You aren’t interested in knowing what those technologies are.

MATCH (person:Person)-[:WORKS\_FOR]->(company)

WHERE company.name STARTS WITH "Company"

AND EXISTS {

MATCH (person)-[:LIKES]->(t:Technology)

WHERE COUNT { (t)<-[:LIKES]-() } >= 3

}

RETURN person.name as person, company.name AS company

//find people who like Java or have more than one friend. you want to return the results ordered by name. return the technologies that these people like, and the friends that they have.

CALL {

MATCH (p:Person)-[:LIKES]->(:Technology {type:'Java'})

RETURN p

UNION

MATCH (p:Person)

WHERE COUNT {(p)-[:IS\_FRIENDS\_WITH]->() } > 1

RETURN p

}

WITH p,

[(p)-[:LIKES]->(t:Technology {type:'Java'}) | t.type ] AS technologies,

[(p)-[:IS\_FRIENDS\_WITH]->(friend:Person) | friend.name ] AS friends

RETURN DISTINCT p.name, technologies, friends

ORDER BY p.name DESC