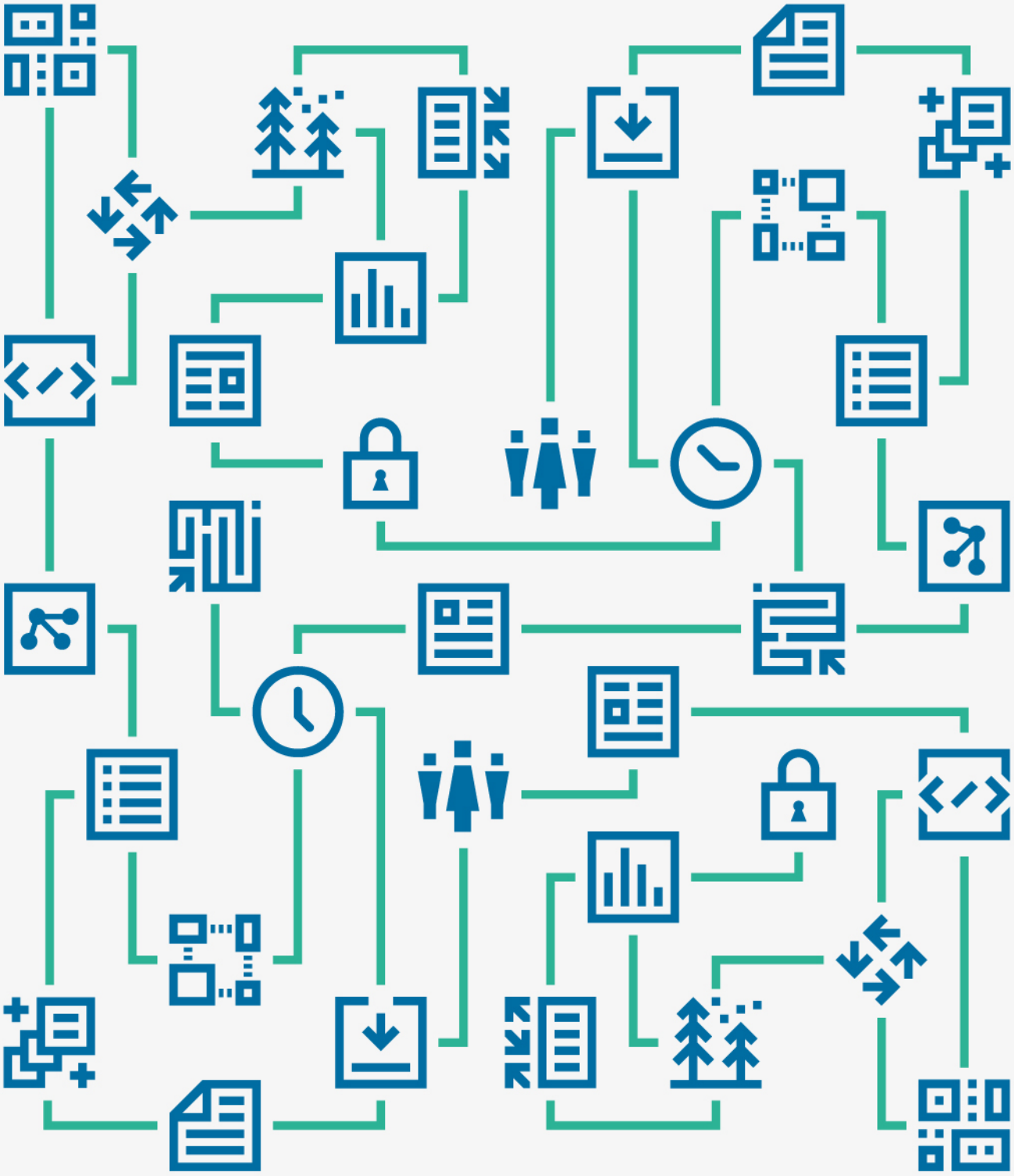
openCypher

Sparksee by Sparsity Technologies



Contents

Introduction **5**

Disclaimer . . . 5

Differences with Open Cypher . . . 5

What is currently NOT supported . . . 7

DataTypes and Expressions . . . 8

Operators and Functions . . . 8

CASE expressions . . . 9

Generic CASE expressions . . . 10

Pattern Matching **11**

Node homomorphism, Edge Isomorphism . . . 12

Matching Nodes . . . 12

Matching Edges . . . 13

Edge directions . . . 13

Matching Nodes and Edges . . . 14

Complex patterns with multiple paths and MATCH clauses . . . 14

Returning columns with * . . . 15

Clauses **17**

MATCH . . . 17

Matching nodes . . . 17

Matching nodes and edges . . . 17

OPTIONAL MATCH . . . 18

RETURN . . . 19

Grouping . . . 20

WITH . . . 21

WHERE . . . 21

ORDER BY . . . 21

SKIP . . . 22

LIMIT . . . 23

UNION . . . 23

Introduction

Disclaimer

This document is the reference of the Sparksee openCypher Language (SCL). SCL is inspired by the OpenCypher Query Language. Inspired means that although the goal is to make SCL as close as possible to OpenCypher, differences between Sparksee's property graph model and OpenCypher model and other technical issues make the two languages to divert in some aspects.

Differences with Open Cypher

In this section, we detail the main differences between SCL and OpenCypher

- The concept of a "label" does not exist in Sparksee, thus, it does not exist in SCL. In OpenCypher, one can attach more than one label to nodes (but not to edges), and use these labels in patterns. Alternatively, edges have one and only one type. Sparksee does not have labels, and treat nodes and edges similarly, with a single one and only one type per node/edge. These types can be used in patterns.
- In SCL, when a node or edge is retrieved, only the OID of the object in question is returned. This contrasts to OpenCypher, where all attributes of the node/edge are returned. In SCL, the returning of properties must be explicitly stated.
- OpenCypher assumes a very relaxed property-graph model where properties are not attached to node/edge types. Sparksee adopts, in general, a more restrictive model where properties are usually restricted to specific types, unless created specifically as NODES, EDGES or GLOBAL properties, which apply to all node types, edge types or both, respectively. Thus, we introduce a set of rules and constructs to understand when a "type-specific", NODES, EDGES or GLOBAL property will be used in a query The general rule is that always, the most restrictive property is searched until one is found, or an error will be reported if one cannot be found, as follows:
  - For nodes, if the type of the node is known: "type-specific" → NODES → GLOBAL.
    For example, in the following query, the type of n is known to be an "actor". Thus, Sparksee will first look for an "actor" "type-specific" property "name". If it exists, it will use that property in the query. If it does not exist, it will look for a NODES property named "name". If it exists, it will use that property in the query. If not, it will look for a GLOBAL property. If it exists, it will use that property in the query. If not, the query will throw an error.

```
MATCH (n:actor {name : 'Brad'})
RETURN *
```

Similarly, in the following query, the type of n is known to be "actor" because of the right path of the pattern, which allows the engine to ensure that the resulting nodes bound to n will be of type actor. Thus, Sparksee will first look for an "actor" "type-specific" property "name". If it exists, it will use that property in the query. If it does not exist, will look for a NODES property named "name". If it exists, it will use that property in the query. If not, it will look for a GLOBAL property. If it exists, it will use that property in the query. If not, the query will throw an error.

```
MATCH (n), (n:actor)-[]-()
RETURN n.name
```

– For nodes, if the type of the node is **NOT** known: NODES → GLOBAL.

  For example, in the following query, the type of n is not known. Thus, Sparksee will first look for a NODES property named "name". If it exists, it will use that property in the query. If not, it will look for a GLOBAL property. If it exists, it will use that property in the query. If not, the query will throw an error.

```
MATCH (n {name : 'Brad'})
RETURN *
```

– For edges, if the type of the edge is known: "type-specific" → EDGES → GLOBAL.

  For example, in the following query, the type of r is known to be "role". Thus, Sparksee will first look for a "role" "type-specific" property "type". If it exists, it will use that property in the query. If it does not exist, will look for a EDGES property named "type". If it exists, it will use that property in the query. If not, it will look for a GLOBAL property "type". If it exists, it will use that property in the query. If not, the query will throw an error.

```
MATCH ()-[r:role {type : 'actor'}]->()
RETURN *
```

– For edges, if the type of the edge is **NOT** known: EDGES → GLOBAL.

  For example, in the following query, the type of r is unknown. Thus, Sparksee will look for a EDGES property named "type". If it exists, it will use that property in the query. If not, it will look for a GLOBAL property "type". If it exists, it will use that property in the query. If not, the query will throw an error. MATCH ()-[r {type : 'actor'}]->() RETURN *

  Similarly, in the following query, the type of r unknown because of the union of the two edge collections of different types. Thus, Sparksee will look for a EDGES property named "timestamp". If it exists, it

will use that property in the query. If not, it will look for a GLOBAL property "timestamp". If it exists, it will use that property in the query. If not, the query will throw an error.

```
MATCH ()-[r:role]->()
UNION
MATCH ()-[r:is_located]->()
RETURN r.timestamp
```

– Finally, if nodes and edges are mixed in a query: GLOBAL
For example, in the following query, a union between nodes of type actor and edges of type role is done. In this case, the type of column is "unknown", since it contains objects of different types. When looking for the timestamp, Sparksee will always look for a GLOBAL property. If it does not exist, it will return an error

```
MATCH (n:actor)
UNION
MATCH ()-[n:role]->()
RETURN n.timestamp
```

- When adding properties to a node/edge from a map, the properties must exist in the schema and be of the correct DataType. The same rules for inferring the property of the values that are inserted to apply.
- When using CREATE, one and only one type must always be provided for nodes and edges. The creation of typeless nor multityped nodes and edges is not allowed.
- The DETACH DELETE clause is not supported. By default, when deleting a node, all its relationships are also deleted
- Setting a property to NULL with SET, does not remove the property of the node, it sets it to null. The Sparksee property-graph model does not allow nodes or edges without a property value if their type has that property. Instead, the property is NULL when they do not have it.
- The REMOVE clause is not supported, given that it is meant to remove properties from nodes. This does not make sense in Sparksee, since a property nor a label cannot be removed from a node. Instead, these are set to NULL, which can be already done with the SET clause.

What is currently NOT supported

With the current version, the following things are not still supported:

- Variable length path patterns, either bound and unbound
- Support for path binding
- Lists and Maps
- Dynamic parameters

- Graph modification clauses (CREATE, SET, DELETE)
- CALL procedures
- String Manipulation functions
- Statistical and Mathematical functions

DataTypes and Expressions

The following are the datatypes supported in expressions:

- Boolean: True or False
- Integer: 32 bit integer
- Long: 64 bit integer
- Double: 64 bit floating point
- String: Unicode character array string
- Timestamp: timestmap stored as a 64 bit integer

All expressions in SCL have a resulting expression datatype. Multiple expressions (typically one or two) can be combined with an operator to form another expression whose datatype will be that of the subexpressions. In general, only subexpressions of the same datatype can be combined, However, If the subexpressions are of different datatypes but these are numeric (Integer, Long, Double or Timestamp), the datatypes of the more restrictive type will be promoted to match the less restrictive datatype of the subexpressions. The promotion order is established as follows: Integer → Long → Timestamp → Double.

Operators and Functions

SCL supports the following unary and binary operators

Table 1Unary operators

Operator	Description
NOT X	Negates X
X IS NULL	Checks if X is null
X IS NOT NULL	Checks if X is not null
+ X	Unary positive arithmetic operator
- X	Unary negative arithmetic operator

Table 2Binary operators

Operator	Description
X AND Y	Binary AND
X OR Y	Binary OR
X = Y	X equals Y
X <> Y	X different than Y
X > Y	X greater than Y

Operator	Description
X >= Y	X greater or equal than Y
X < Y	X smaller than Y
X <= Y	X smaller or equal than Y
X + Y	X plus Y
X - Y	X minus Y
X * Y	X times Y
X / Y	X divided by Y
X % Y	X modulo Y

Additionally, the following functions can be used in expressions

Table 3Functions

Operator	Description
toBoolean(X)	Casts X into a boolean
toInteger(X)	Casts X into an Integer
toLong(X)	Casts X into a Long
toTimestamp(X)	Casts X into a Timestamp
toDouble(X)	Casts X into a Double
toString(X)	Casts X into a String
type(X)	Returns the type of object X as a String
date(X) or datetime(X)	Converts the string X (e.g. 2006-01-01) to a timestamp
timestamp()	Returns the current timestamp

CASE expressions

SCL supports case expressions in two forms: simple and generic.

Simple CASE expressions In simple case expressions, an expression is compared against different values until a match is found. If a match is not found, the default value is returned or NULL if such value has not been specified. The syntax is as follows

```
CASE test
 WHEN value THEN result
 [WHEN ...]
 [ELSE default]
END
```

where “test” is a valid expression, “value” is the value the expression is compared to, and result is the new value to set the column to. If no matches are found, the “default” value is set or NULL if it is not specified. All results must be of the same datatype. For example, a valid simple case expression is:

```
CASE n.color
 WHEN "Blue" THEN 1
 WHEN "Red"  THEN 2
 ELSE 3
END AS COLOR_CATEGORY
```

Generic CASE expressions

Generic case expressions are similar to simple ones, with the difference that the predicate can be an arbitrary expression. As in the simple CASE expression, all results must be of the same datatype.

```
CASE
 WHEN predicate THEN result1
 [WHEN ...]
 [ELSE default]
END
```

For instance, we could write:

```
CASE
 WHEN n.age < 13 THEN "kid"
 WHEN n.age < 18 THEN "teen"
 ELSE "adult"
END
```

Pattern Matching

SCL queries follow the same structure as in Open Cypher queries. In their simplest form, these to start with a MATCH clause that specifies a pattern to match, plus a RETURN clause returning all the bound variables. In the following query, all nodes of the graph are matched and bound to variable n.

```
MATCH (n)
RETURN *
```

Statements in a query are conceptually executed one after the other (although the engine is free to execute them in any order as long as the correct query result is preserved). When the star (*) operator is used in a RETURN statement, all bound variable are returned.

Patterns are expressed in terms of one or more paths. A path is a succession of steps connecting nodes of the graph through edges. For instance, the following pattern consist of a single path one step that matches all pairs of connected nodes through a directed edge. In this case, the first node in the pair is bound to variable n, while the second node is bound to variable m:

```
MATCH (n)->(m)
RETURN *
```

More complex patterns can be expressed by combining multiple paths like in the following query, which uses to paths with one step to look for pairs of nodes that are reciprocally connected through a directed edge.

```
MATCH (n)->(m), (m)->(n)
RETURN *
```

The previous query, could be expressed alternatively with a single path with two steps:

```
MATCH (n)->(m)->(n)
RETURN *
```

or alternatively, by reversing the direction of the matched edges

```
MATCH (n)<-(m)<-(n)
RETURN *
```

Finally, if we are interested in also knowing the ids of the edges, we can express the pattern in the following way:

```
MATCH (n)-[r]->(m)-[q]->(n)
RETURN *
```

Node homomorphism, Edge Isomorphism

In SCL, as in Open Cypher, node matching follows the rules of subgraph homomorphism, while edges follow the rules of subgraph isomorphism. In homomorphism, two variables can be bound to the same object, while in isomorphism two variables must be bound to different objects. For instance, in the following query:

```
MATCH (n)-[r]->(m)-[q]->(n)
RETURN *
```

n and **m** could be bound to the same node, but **r** and **q** must be bound to different edges. This means that if the graph contains a node with two self loops (a self loop pointing to itself), the pattern would find an occurrence. This would not be true if the node had just a single self loop, because then only one edge (**r** or **q**) could be bound in the pattern.

Matching Nodes

The following query matches and returns all nodes of the graph

```
MATCH (n)
RETURN *
```

Nodes can be filtered by their type when matched. The following query, returns all the nodes of the graph of type "person":

```
MATCH (n:person)
RETURN *
```

Finally, we can also match nodes with certain property values. For instance, the following query returns all nodes of type person whose name is "John" and are 18 years old:

```
MATCH (n:person { name : "John", age : 18} )
RETURN *
```

As described in ['Differences with Open Cypher'][doc:Introudction], the properties matched depend on whether the type of the node ("person" in this case) is specified or not. In the query above, if there exists both a "type-specific" "name" property for "person", and a global property called "name", the "type-specific" one will take preference given that we know that the type of the node is "person". However in the following query, the global property would take preference because we have not specified the type of n:

```
MATCH (n { name : "John"} )
RETURN *
```

Matching Edges

Similar to nodes, we can match edges. The following query matches and returns all the edges of the graph, regardless of the direction. This means that each edge will be matched twice, one for each orientation.

```
MATCH ()-[r]-()
RETURN *
```

The matching of edges can also be restricted by the edge type. For instance, the following query matches all the edges of type "role":

```
MATCH ()-[r:role]-()
RETURN *
```

Finally, edges can also have properties like nodes, and can be used to restrict the matching of the pattern:

```
MATCH ()-[r:role {type : 'actor'}]-()
RETURN *
```

Edge directions

When matching edges, we can specify the direction of the match. This is particularly useful when having directed edges and we are interested on particular endpoints. For instance, the following query retrieves all the nodes that are a "tail" of an edge of type "role":

```
MATCH (n)-[r:role]->()
RETURN *
```

Similarly, we could be interested in the nodes that are a "head" of an edge of type "role" as follows:

```
MATCH (n)<-[r:role]-()
RETURN *
```

or

```
MATCH ()-[r:role]->(n)
RETURN *
```

Matching Nodes and Edges

The matching of nodes and edges can be combined in a single query. For instance, the following query matches all pairs of "persons" and "movies" such that the person has exercised as an "actor" in the movie

```
MATCH (n:person)-[r:role {type : 'actor'}]-(m:movie)
RETURN *
```

Similarly, we can restrict more the pattern when looking for all the actors of a specific movie:

```
MATCH (n:person)-[r:role {type : 'actor'}]-(m:movie {title : 'The Thing'})
RETURN *
```

We can also specify matching a node or an edge without binging them to a variable. For instance, if in the previous query we are only interested in the person nodes, we can specify such query by removing the variable names as follows:

```
MATCH (n:person)-[:role {type : 'actor'}]-(:movie {title : 'The Thing'})
RETURN *
```

Complex patterns with multiple paths and MATCH clauses

We can build more complex patterns by combining multiple paths. When using multiple paths, occurrence of the same variable appearing in more than one path refers to the same matched object. For instance, the following query looks for pairs of movies where a person has participated.

```
MATCH (n:person)-[r1:role]-(m:movie),
      (n:person)-[r2:role]-(q:movie)
RETURN *
```

Note that due to the edge isomorphism rule, r1 and r2 must be different edges.

If we want to allow r1 and r2 to match the same edges, we can express the same query as follows:

```
MATCH (n:person)-[r1]-(m:movie),
MATCH (n:person)-[r2]-(q:movie)
RETURN *
```

In this case, two patterns are matched in the graph, and are combined. In other words, the edge-isomorphism rules only apply to multiple paths within the same pattern, but not between different patterns.

Returning columns with *

When returning columns with the * operator, all variables bound in the query are returned. However, the following aspects must be considered: * All variables bound in the query are returned * The order of the columns is the same as the order of occurrence of the variables in the query, reading from left to right first, and then from top to bottom. * Variables occurring more than once will only be returned once, and that of the first occurrence.

For instance, in the following query:

```
MATCH (n:person)-[r1:role {type : 'actor'}]-(m:movie),
MATCH (n:person)-[r2:role {type : 'director'}]-(q:movie)
RETURN *
```

the returned columns will be: n, r1, m, r2, q, in that order. Columns are always named after the expression name, unless an alias is specified. For more details, refer to the [‘RETURN’ clause][doc:Clauses].

Clauses

MATCH

Matching nodes

The MATCH clause is the cornerstone around which the Sparksee Cypher Language queries are constructed. With the MATCH clause, the user specifies the pattern to look for in the graph.

Patterns in a MATCH clause are composed of "Paths", separated by commas. Paths connect one or more nodes through edges. Restrictions such as the node or edge type, the direction of the connecting edges and node and edge properties can be specified in order to restrict the patterns to be matched.

For each pattern matched, one row is returned containing all the variables that have been bound in the pattern. These variables can then be used in subsequent clauses such as other MATCH, expressions in predicates or RETURN statements, grouping, etc.

The simplest use of the MATCH clause is to match all the nodes of the graph as follows:

```
MATCH (n)
RETURN *
```

This query pattern consists of a single path with a single node. Additionally, the type and the properties of the node can be specified as follows, where all the nodes of type person whose name is "John":

```
MATCH (n:person { name: 'John' })
RETURN *
```

Matching nodes and edges

We can specify more complex paths connecting nodes through edges. The following query matches all pairs of "person" and "movie" connected through a "role" edge:

```
MATCH (n:person)-[:role]->(m:movie)
RETURN *
```

Similarly to nodes, we can specify properties on edges as follows, where only the roles of property "type" equals "actor" are matched:

```
MATCH (n:person)-[:role {type : "actor"}]->(m:movie)
RETURN *
```

Paths, are not limited to just one step, longer paths can be expressed by chaining multiple edges:

```
MATCH (n:person)-[:role {type : "actor"}]->(:movie)<-[:role {type : "actor"}]-(q:
    person)
RETURN *
```

where all pairs of persons (bound to n and q) that exercised as an "actor" in the same movie are searched for. Note that due to the rules described in [Node homomorphism, Edge Isomorphism][doc:Introduction], both n and q could match the same node. The previous query, could also be expressed using two separate paths as follows:

```
MATCH (n:person)-[:role {type : "actor"}]->(m:movie), (m:movie)<-[:role {type : "
    actor"}]-(q:person)
RETURN *
```

with the difference that in this case we would also return the movie "m". Finally, we could use two MATCH clauses to enforce "n" and "q" to be different:

```
MATCH (n:person)-[:role {type : "actor"}]->(m:movie)
MATCH (m:movie)<-[:role {type : "actor"}]-(q:person)
RETURN *
```

where conceptually, two patterns are matched independently and joined through the common variables.

OPTIONAL MATCH

The OPTIONAL MATCH is a variant of MATCH clause that, as its name indicates, optionally checks for the presence of a pattern. It is the equivalent of an outer join in SQL, in this case when joining the result of two MATCH clauses. For instance, the following query matches all persons of the graph and "optionally", returns the movies where these have exercised as an "actor":

```
MATCH (n:person)
OPTIONAL MATCH (n:person)-[:role { type : "actor" }]->(m:movie)
RETURN *
```

In this query, all the persons would be returned, regardless whether these have been actors in a movie or not. If a person does not optionally match the second pattern, NULL will be returned for the "m" column. However, for each movie she participated as an "actor", a row will be returned. If, instead of using OPTIONAL MATCH, we used a regular MATCH in the previous query, only those pairs of "person" "n" acting in a "movie" "m" would be returned.

Note that a query cannot start with an OPTIONAL MATCH clause.

RETURN

The RETURN clause is used to decide what to project in the result. The easiest way to use the RETURN clause is by using the * operator, which will return all the variables bound, following the following rules:

- All variables bound in the query are returned
- The order of the columns is the same as the order of occurrence of the variables in the query, reading from left to right first, and then from top to bottom.
- Variables occurring more than once will only be returned once, and that of the first occurrence.

For instance, in the following query:

```
MATCH (n:person)-[r1:role {type : 'actor'}]-(m:movie),
MATCH (n:person)-[r2:role {type : 'director'}]-(q:movie)
RETURN *
```

the returned columns will be: n, r1, m, r2, q, in that order. Columns are always named after the expression name, unless an alias is specified.

Users can, however, customize what to project, including properties of nodes an edges. The following query will be equivalent to the previous one:

```
MATCH (n:person)-[r1:role {type : 'actor'}]-(m:movie),
MATCH (n:person)-[r2:role {type : 'director'}]-(q:movie)
RETURN n, r1, m, r2, q
```

To specify the projection of a property value, just use the "var.property" convention, as follows:

```
MATCH (n:person),
RETURN n, n.name, n.age
```

where both the id, the "name" and the "age" of the "person" nodes are returned as columns named "n", "n.name" and "n.age" respectively. One can also rename the columns using aliases, as fallows:

```
MATCH (n:person),
RETURN n AS ID, n.name AS NAME, n.age AS AGE
```

where the returned columns will be named as "ID", "NAME" and "AGE" respectively. Finally, arbitrary expressions can be returned:

```
MATCH (n:person),
RETURN n AS ID,
       CASE
          WHEN n.age >= 18 THEN "adult"
          ELSE "child"
```

```
        END AS CATEGORY,
        timestamp() - n.timestamp AS ELAPSED_TIME,
```

Grouping

In RETURN expressions, one can perform grouping operations and compute statistics on these groups. When a grouping operation is specified in a projection, the other projected columns become the grouping columns used to form the groups. For instance, we can write the following query to count the number of persons of each age as follows:

```
MATCH (n:person)
RETURN n.age, count(*) AS FREQUENCY
```

Instead, we can count the number of non-NULL values of a particular expression. For instance, the following query returns the number of persons with a non-NULL time.

```
MATCH (n:person)
RETURN n.age, count(n.time) AS COUNT_NON_NULL
```

Also, we can use the DISTINCT word to compute the number of distinct values in a group. For instance, the following query computes the number of distinct names on each age group:

```
MATCH (n:person)
RETURN n.age, count(DISTINCT n.name) AS COUNT_DISTINCT_NAMES
```

The following, are the different grouping functions available.

Table 4Aggregate Functions

Operator	Description
count(*)	Count the number of returned values
count(X)	Counts the number of values different than NULL
min(X)	Gets the mínimum of the values
max(X)	Gets the maximum of the values
avg(X)	Gets the average of the values
sum(X)	Gets the sum of the values

Additionally, all the aggregate functions can be modified with the DISTINCT modifier (e.g. sum(DISTINCT X), count(DISTINCT X), etc)

WITH

The WITH clause is used to chain query parts in such a way that the output of one part is piped into the input of the other. With the WITH clause you can manipulate the output of the query before being consumed further, for instance, by deciding what to project, limiting the number of outputs, filter on aggregate values, etc.

The WITH is used like the RETURN clause. It creates a projection for the data, and after it, only the projected elements can be referenced in the remaining of the query. The * operator can be used to return all variables, aliases can be assigned to columns, etc. For instance, the following query returns the age groups where the frequency is larger than a threshold:

```
MATCH (n:person)
WITH n.age, count(*) AS FREQUENCY
WHERE FREQUENCY > 10
RETURN *
```

WHERE

The WHERE clause is used to filter the rows using predicates. WHERE must appear in conjunction with a MATCH, OPTIONAL MATCH or WITH clause. With the WHERE clause, the user must specify a boolean expression to filter the rows. For instance, the following query filters the persons of the graph by their age:

```
MATCH (n:person)
WHERE n.age > 18
RETURN n
```

Similarly, we can express the same query using WHERE in conjunction with a WITH clause:

```
MATCH (n:person)
WITH n, n.age AS AGE
WHERE AGE > 18
return n
```

ORDER BY

The ORDER BY clause is used to sort the rows using different criteria. ORDER BY is a sub-clause of WITH and RETURN, and as such it must appear in conjunction with one of these two other clauses. Rows can be sorted by different columns, and the sorting direction (ascending or descending) can be specified independently for each of them. For instance, the following query sorts the persons by their “age” ascendingly:

```
MATCH (n:person)
RETURN n, n.age AS AGE
ORDER BY AGE ASC
```

Similarly, we can sort the edges descendingly:

```
MATCH (n:person)
RETURN n, n.age AS AGE
ORDER BY AGE DESC
```

We can sort rows by multiple columns using different criteria for each column. For instance, we can sort first by "age" descendingly and then by "name" ascendingly as follows:

```
MATCH (n:person)
RETURN n, n.age AS AGE, n.name AS NAME
ORDER BY AGE DESC, NAME ASC
```

Finally, we can use ORDER BY with the WITH clause. The previous query could be rewritten as fallows:

```
MATCH (n:person)
WITH n, n.age AS AGE, n.name AS NAME
ORDER BY AGE DESC, NAME ASC
RETURN *
```

SKIP

The SKIP clause is used to specify from which point the rows are being included in the result. SKIP clause is a sub-clause of RETURN and WITH, and as such it must appear in conjunction with one of those. Note that without an ORDER BY clause before the SKIP, no guarantees are given to which rows are being skipped. For instance, we can write a query to filter the first 10 persons sorted by name descendingly:

```
MATCH (n:person)
RETURN n, n.name AS NAME
ORDER BY NAME DESC
SKIP 10
```

Also, more complex expressions can be used in the SKIP clause, but always without referencing any variable and resolving into a result of INTEGER or LONG type:

```
MATCH (n:person)
RETURN n, n.age AS AGE, n.name AS NAME
ORDER BY AGE DESC, NAME ASC
SKIP toInteger(timestamp()/10000000)
```

LIMIT

The LIMIT clause is used to specify from which point the rows are being included in the result. LIMIT clause is a sub-clause of RETURN and WITH, and as such it must appear in conjunction with one of those. Note that without an ORDER BY clause before the LIMIT, no guarantees are given to which rows are conserved. For instance, we can write a query to only keep the first 10 persons sorted by name descendingly:

```
MATCH (n:person)
RETURN n, n.name AS NAME
ORDER BY NAME DESC
LIMIT 10
```

Also, more complex expressions can be used in the LIMIT clause, but always without referencing any variable and resolving into a result of INTEGER or LONG type:

```
MATCH (n:person)
RETURN n, n.age AS AGE, n.name AS NAME
ORDER BY AGE DESC, NAME ASC
LIMIT toInteger(timestamp()/10000000)
```

UNION

The UNION clause performs the union of multiple chained queries. Such queries must always contain the same number of columns and these need to have the same name. For example, the following query returns the union of nodes of type "person" and "movie", with a column containing their type

```
MATCH (n:person)
WITH n as ID, type(n) AS TYPE
UNION
MATCH (m:movie)
WITH m as ID, type(m) AS TYPE
RETURN *
```