Chapter 1

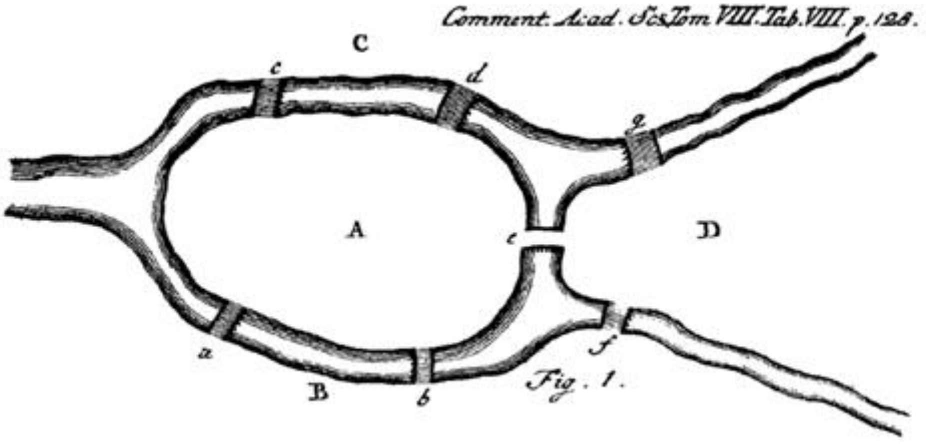
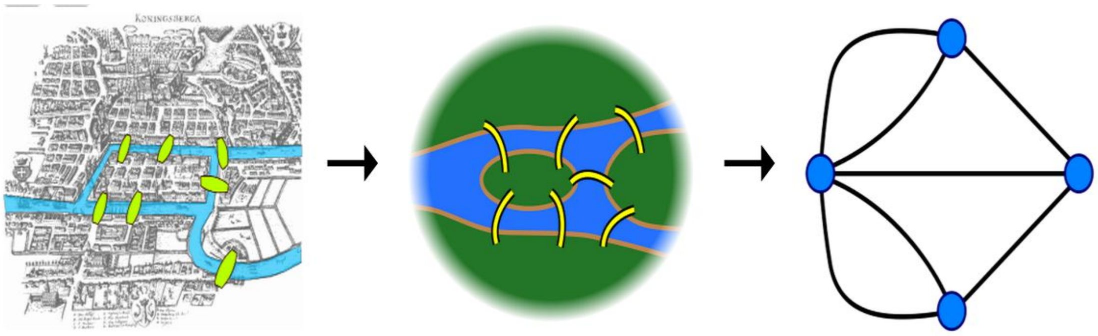
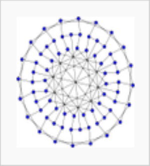
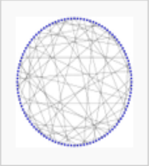
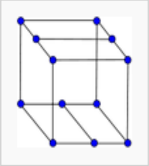


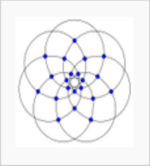
Illustration of the mentioned problem as mentioned by Euler in his paper in 1736

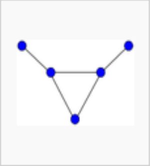


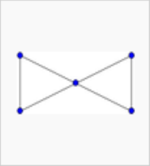

Balaban 10-cage

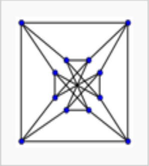

Balaban 11-cage

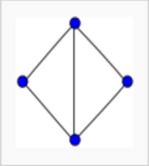

Bidiakis cube

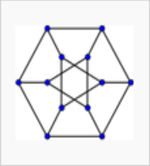

Brinkmann graph

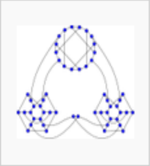

Bull graph

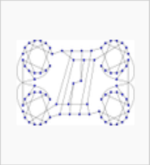

Butterfly graph

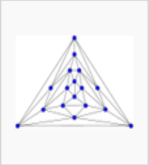

Chvátal graph

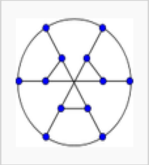

Diamond graph

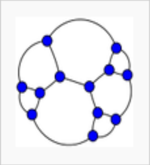

Dürer graph

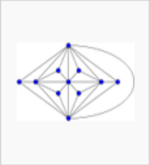

Ellingham–Horton 54-graph

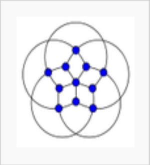

Ellingham–Horton 78-graph

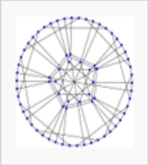

Errera graph

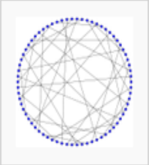

Franklin graph

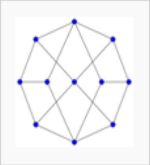

Frucht graph

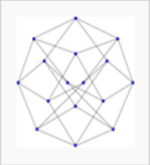

Goldner–Harary graph

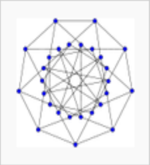

Grötzsch graph

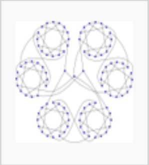

Harries graph

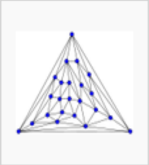

Harries–Wong graph

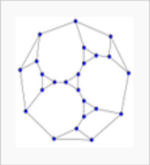

Herschel graph

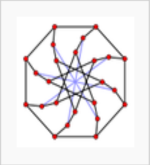

Hoffman graph

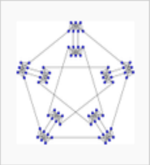

Holt graph

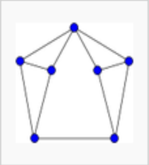

Horton graph

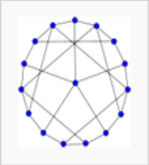

Kittell graph

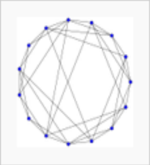

Markström graph

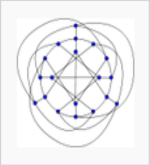

McGee graph

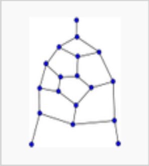

Meredith graph

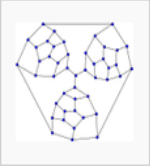

Moser spindle

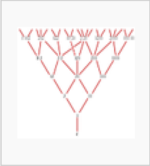

Sousselier graph

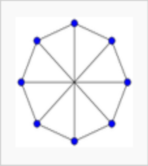

Poussin graph

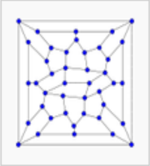

Robertson graph

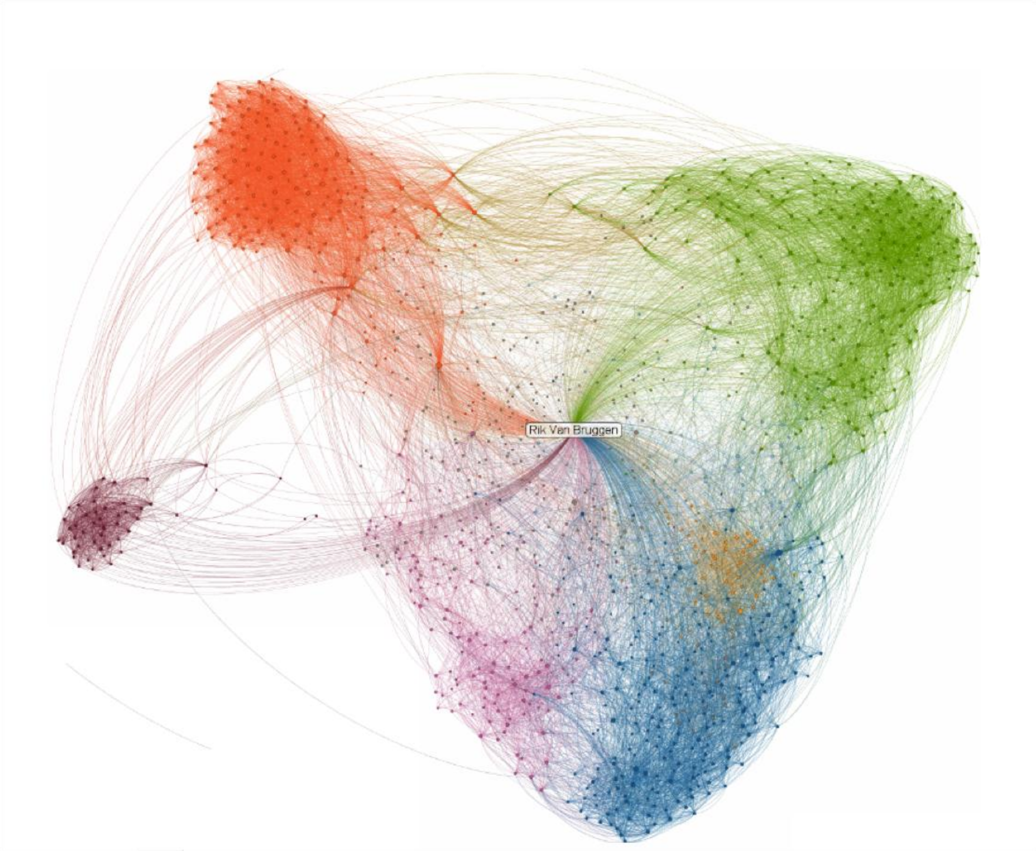

Tutte's fragment

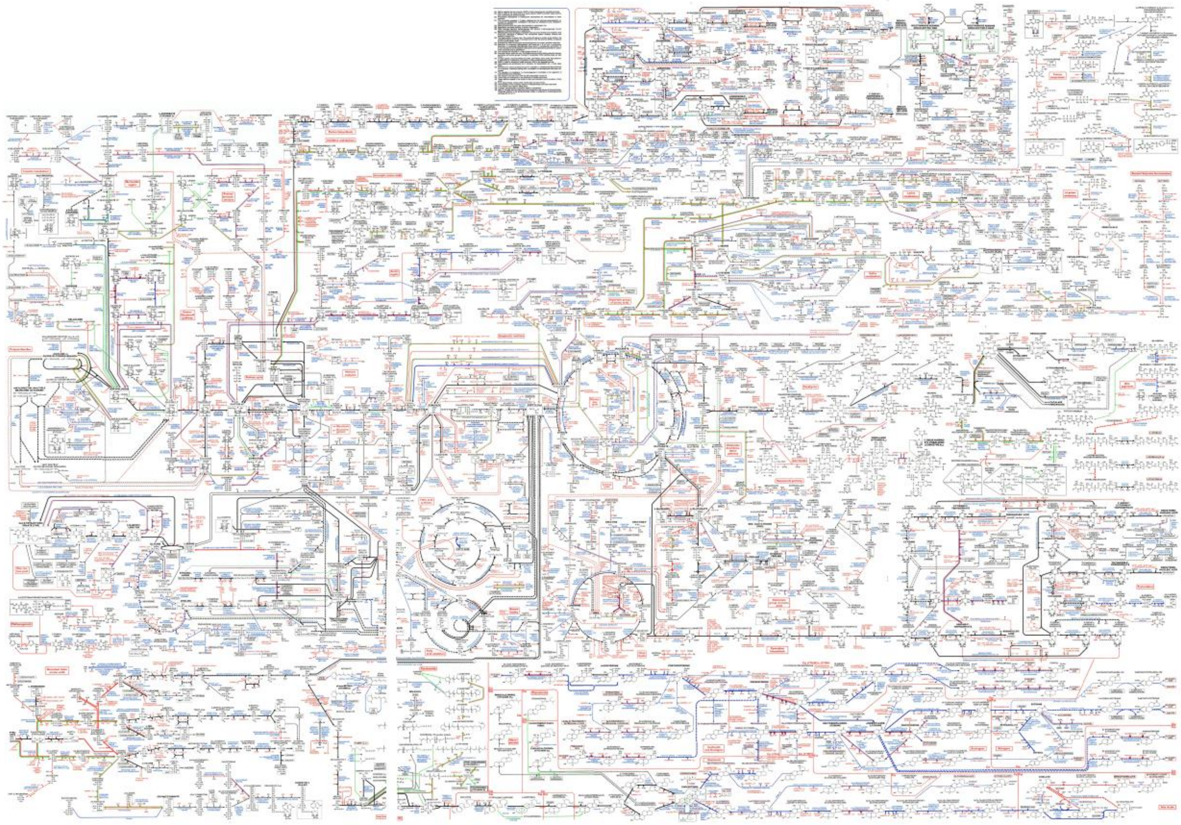

Tutte graph


Young–Fibonacci graph

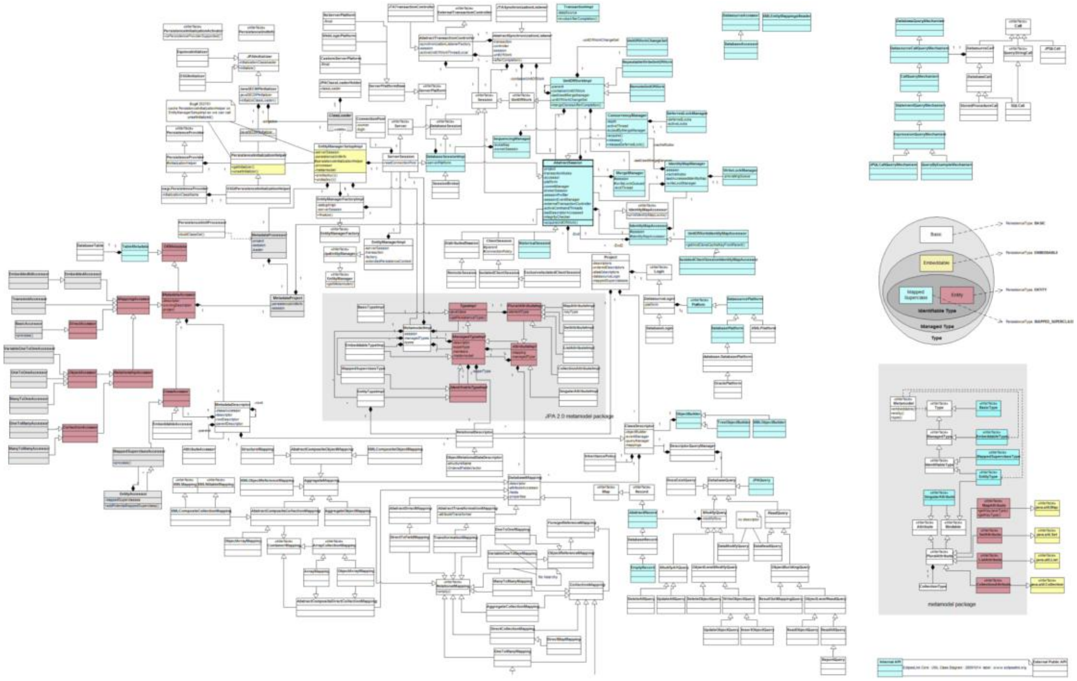

Wagner graph


Wiener–Araya graph

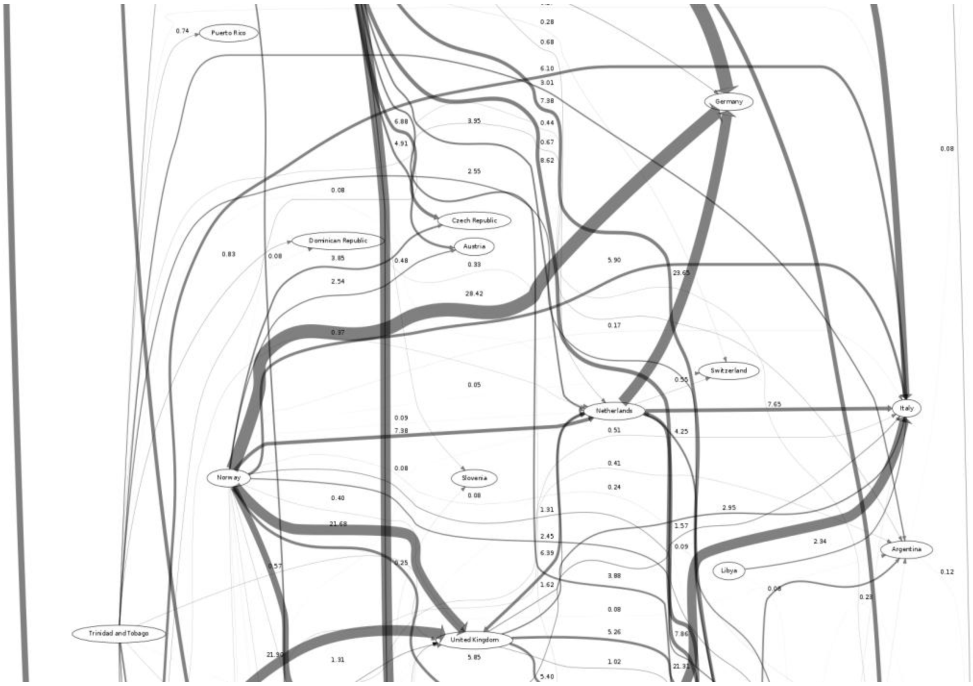

Rik Van Bruggen



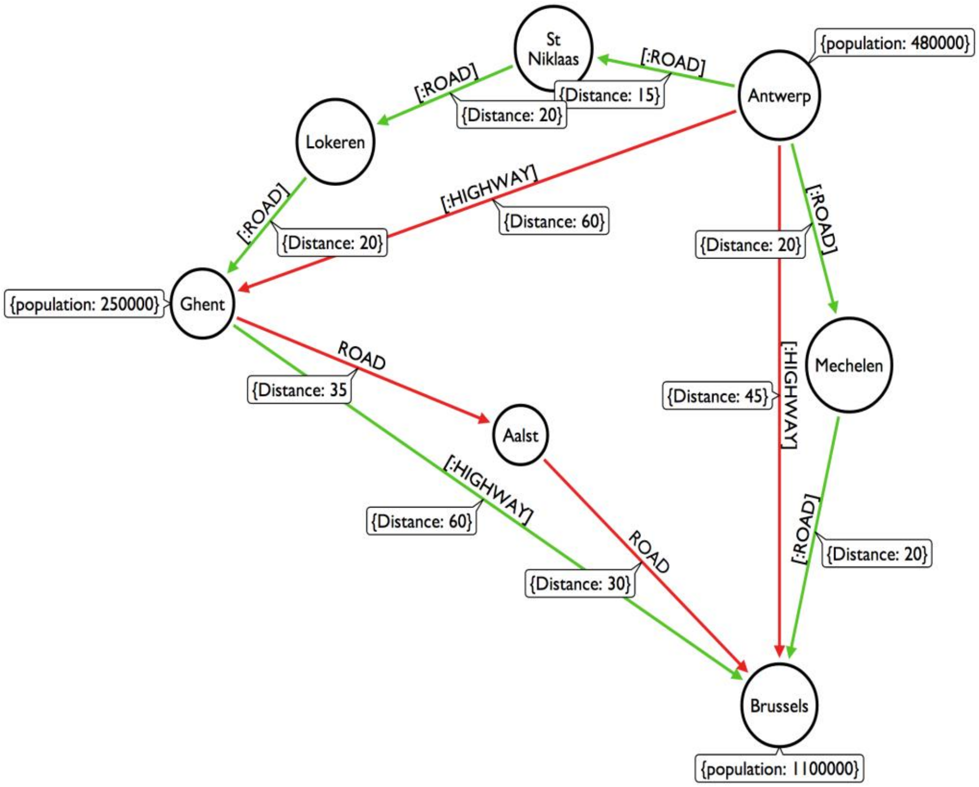
A diagram representing the human metabolic system



An example of a UML diagram

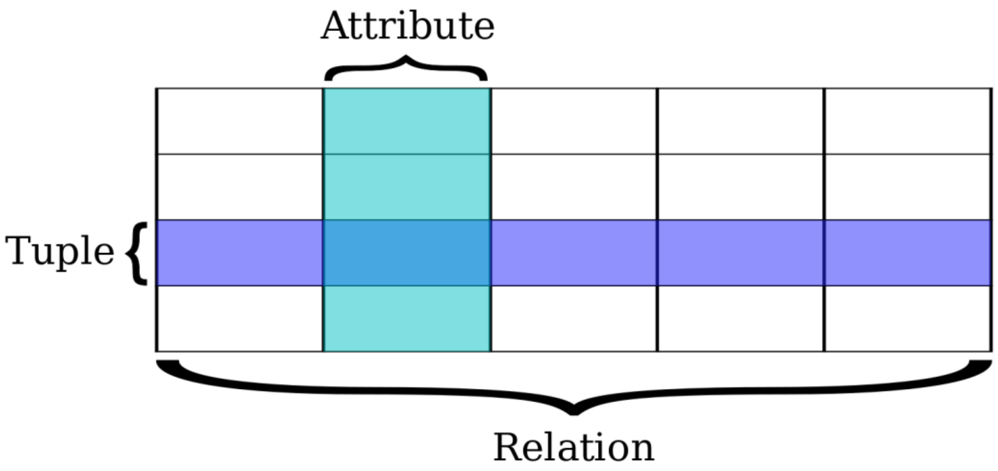


An example of a flow network

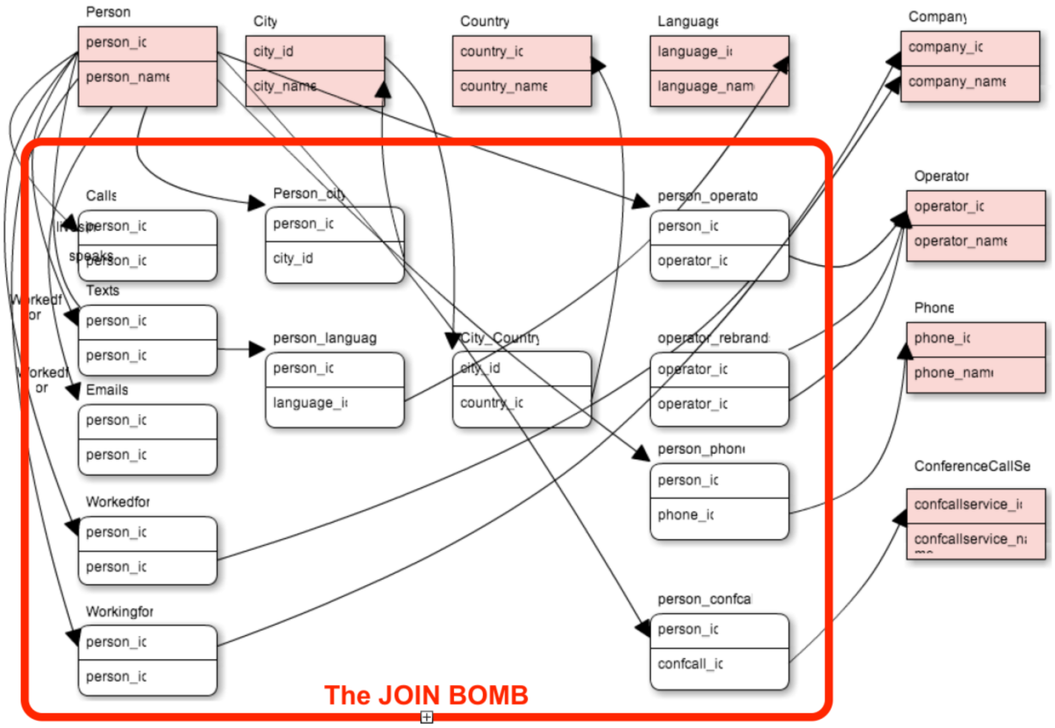


A simple route planning example between cities to choose roads versus highways

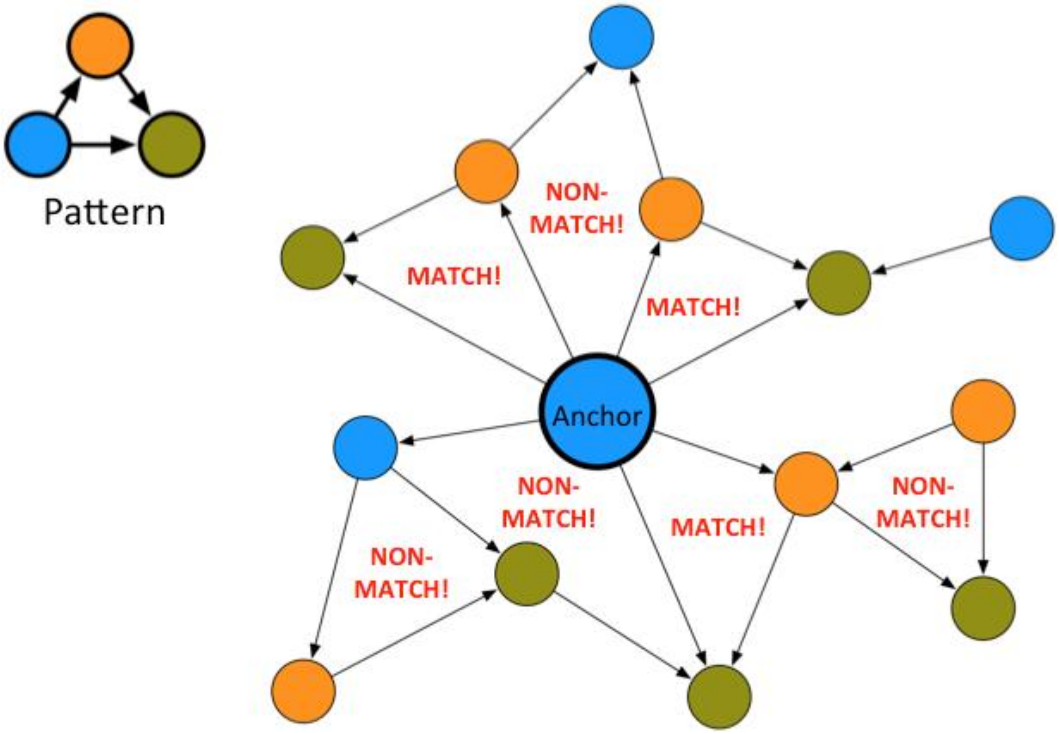
Chapter 2



Relational database terminology

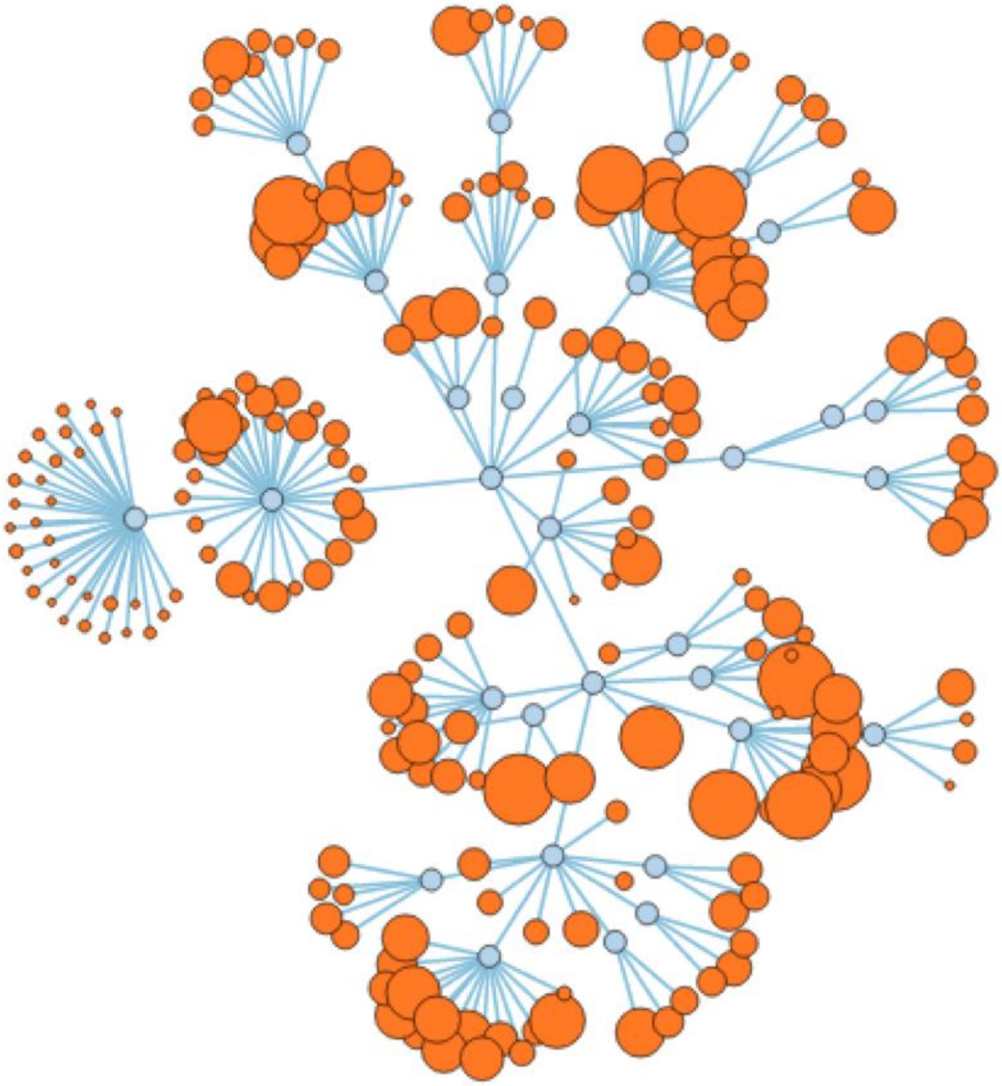


Relational database schema with explosive join tables

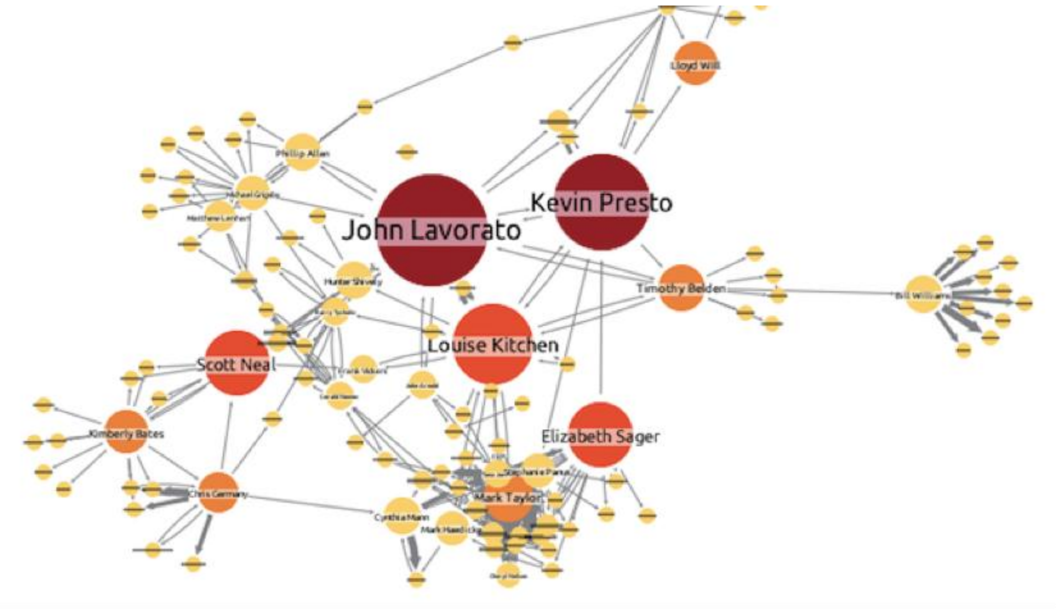


Matching patterns connected to an anchor node

Chapter 8

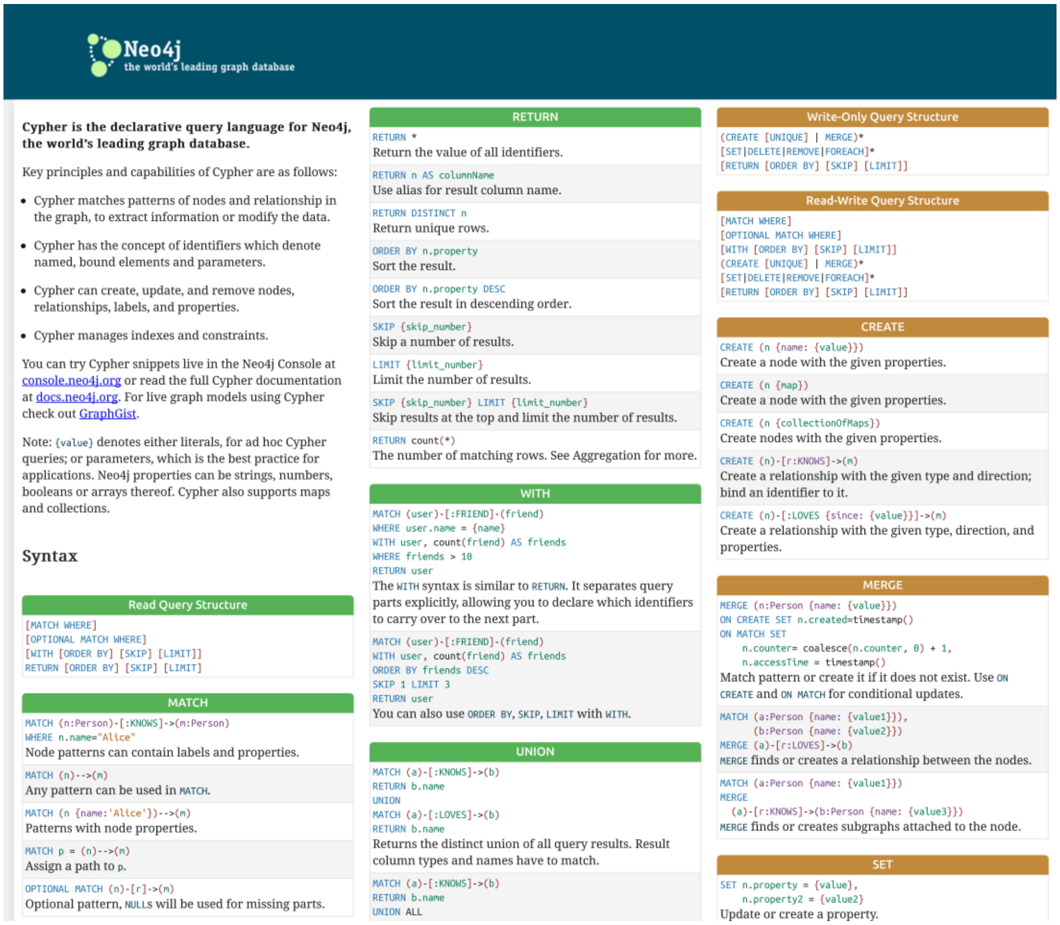


D3 visualization of a graph



Keylines graph visualization

Appendix B



Cypher is the declarative query language for Neo4j, the world's leading graph database.

Key principles and capabilities of Cypher are as follows:

- Cypher matches patterns of nodes and relationship in the graph, to extract information or modify the data.
- Cypher has the concept of identifiers which denote named, bound elements and parameters.
- Cypher can create, update, and remove nodes, relationships, labels, and properties.
- Cypher manages indexes and constraints.

You can try Cypher snippets live in the Neo4j Console at console.neo4j.org or read the full Cypher documentation at docs.neo4j.org. For live graph models using Cypher check out GraphGist.

Note: `{value}` denotes either literals, for ad hoc Cypher queries; or parameters, which is the best practice for applications. Neo4j properties can be strings, numbers, booleans or arrays thereof. Cypher also supports maps and collections.

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`.

```
MATCH (n {name:'Alice'})-->(m)
```

Patterns with node properties.

```
MATCH p = (n)-->(m)
```

Assign a path to `p`.

```
OPTIONAL MATCH (n)-[r]->(m)
```

Optional pattern, `NULL`s will be used for missing parts.

RETURN

```
RETURN *
```

Return the value of all identifiers.

```
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 {skip_number}
```

Skip a number of results.

```
LIMIT {limit_number}
```

Limit the number of results.

```
SKIP {skip_number} LIMIT {limit_number}
```

Skip results at the top and limit the number of results.

```
RETURN count(*)
```

The number of matching rows. See Aggregation 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 parts explicitly, allowing you to declare which identifiers to carry over to the next part.

```
MATCH (user)-[:FRIEND]-(friend)
WITH user, count(friend) AS friends
ORDER BY friends DESC
SKIP 1 LIMIT 3
RETURN user
```

You can also use `ORDER BY`, `SKIP`, `LIMIT` with `WITH`.

UNION

```
MATCH (a)-[:KNOWS]->(b)
RETURN b.name
UNION
MATCH (a)-[:LOVES]->(b)
RETURN b.name
```

Returns the distinct union of all query results. Result column types and names have to match.

```
MATCH (a)-[:KNOWS]->(b)
RETURN b.name
UNION ALL
```

Write-Only Query Structure

```
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

Read-Write Query Structure

```
[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

CREATE

```
CREATE (n {name: {value}})
```

Create a node with the given properties.

```
CREATE (n {map})
```

Create a node with the given properties.

```
CREATE (n {collectionOfMaps})
```

Create nodes with the given properties.

```
CREATE (n)-[r:KNOWS]->(m)
```

Create a relationship with the given type and direction; bind an identifier to it.

```
CREATE (n)-[:LOVES {since: {value}}]->(m)
```

Create a relationship with the given type, direction, and properties.

MERGE

```
MERGE (n:Person {name: {value}})
ON CREATE SET n.created=timestamp()
ON MATCH SET
    n.counter= coalesce(n.counter, 0) + 1,
    n.accessTime = timestamp()
```

Match pattern or create it if it does not exist. Use `ON CREATE` and `ON MATCH` for conditional updates.

```
MATCH (a:Person {name: {value1}}),
      (b:Person {name: {value2}})
MERGE (a)-[r:LOVES]->(b)
```

`MERGE` finds or creates a relationship between the nodes.

```
MATCH (a:Person {name: {value1}})
MERGE
  (a)-[r:KNOWS]->(b:Person {name: {value3}})
```

`MERGE` finds or creates subgraphs attached to the node.

SET

```
SET n.property = {value},
    n.property2 = {value2}
```

Update or create a property.

WHERE

```
WHERE n.property <> {value}
```

Use a predicate to filter. Note that `WHERE` is always part of a `MATCH`, `OPTIONAL MATCH`, `WITH` or `START` clause. Putting it after a different clause in a query will alter what it does.

Operators

Mathematical	`+, -, *, /, %, ^`
Comparison	`=, <>, <, >, <=, >=`
Boolean	`AND, OR, XOR, NOT`
String	`+`
Collection	`+, IN, [x], [x .. y]`
Regular Expression	`=~`

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 expressoin is `NULL`, use `IS NULL`.
- Arithmetic expressions, comparisons and function calls (except `coalesce`) will return `NULL` if any argument is `NULL`.
- Missing elements like a property that doesn't exist or accessing elements that don't exist in a collection yields `NULL`.
- In `OPTIONAL MATCH` clauses, `NULL`s will be used for missing parts of the pattern.

CASE

```
CASE n.eyes
 WHEN 'blue' THEN 1
 WHEN 'brown' THEN 2
 ELSE 3
END
```

Return `THEN` value from the matching `WHEN` value. The `ELSE` value is optional, and substituted for `NULL` if missing.

```
CASE
 WHEN n.eyes = 'blue' THEN 1
 WHEN n.age < 40 THEN 2
 ELSE 3
END
```

Return `THEN` value from the first `WHEN` predicate evaluating to `TRUE`. Predicates are evaluated in order.

```
MATCH (a)-[:LOVES]->(b)
RETURN b.name
```

Returns the union of all query results, including duplicated rows.

Collections

```
['a','b','c'] AS coll
```

Literal collections are declared in square brackets.

```
length({coll}) AS len, {coll}[0] AS value
```

Collections can be passed in as parameters.

```
range({first_num},{last_num},{step}) AS coll
```

Range creates a collection of numbers (`step` is optional), other functions returning collections are: `labels`, `nodes`, `relationships`, `rels`, `filter`, `extract`.

```
MATCH (a)-[r:KNOWS*]->()
RETURN r AS rels
```

Relationship identifiers of a variable length path contain a collection of relationships.

```
RETURN matchedNode.coll[0] AS value,
       length(matchedNode.coll) AS len
```

Properties can be arrays/collections of strings, numbers or booleans.

```
coll[{idx}] AS value,
coll[{start_idx}..{end_idx}] AS slice
```

Collection elements can be accessed with `idx` subscripts in square brackets. Invalid indexes return `NULL`. Slices can be retrieved with intervals from `start_idx` to `end_idx` each of which can be omitted or negative. Out of range elements are ignored.

```
UNWIND {names} AS name
MATCH (n {name:name})
RETURN avg(n.age)
```

With `UNWIND`, you can transform any collection back into individual rows. The example matches all names from a list of names.

Performance

- Use parameters instead of literals when possible. This allows Cypher to re-use your queries instead of having to parse and build new execution plans.
- Always set an upper limit for your variable length patterns. It's easy to have a query go wild and touch all nodes in a graph by mistake.
- Return only the data you need. Avoid returning whole nodes and relationships — instead, pick the data you need and return only that.

```
SET n = {map}
```

Set all properties. This will remove any existing properties.

```
SET n += {map}
```

Add and update properties, while keeping existing ones.

```
SET n:Person
```

Adds a label `Person` to a node.

DELETE

```
DELETE n, r
```

Delete a node and a relationship.

REMOVE

```
REMOVE n:Person
```

Remove a label from `n`.

```
REMOVE n.property
```

Remove a property.

INDEX

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

Create an index on the label `Person` and property `name`.

```
MATCH (n:Person) WHERE n.name = {value}
```

An index can be automatically used for the equality comparison. Note that for example `lower(n.name) = {value}` will not use an index.

```
MATCH (n:Person) WHERE n.name IN [{value}]
```

An index can be automatically used for the IN collection checks.

```
MATCH (n:Person)
USING INDEX n:Person(name)
WHERE n.name = {value}
```

Index usage can be enforced, when Cypher uses a suboptimal index or more than one index should be used.

```
DROP INDEX ON :Person(name)
```

Drop the index on the label `Person` and property `name`.

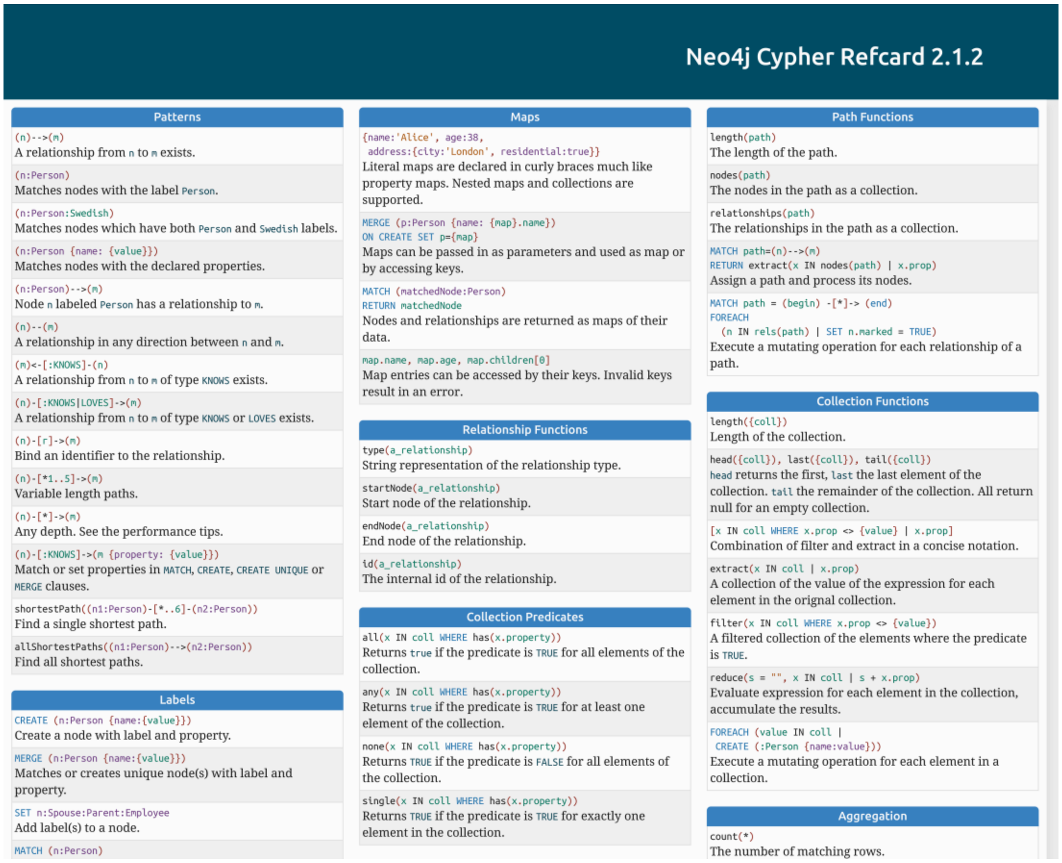
CONSTRAINT

```
CREATE CONSTRAINT ON (p:Person)
       ASSERT p.name IS UNIQUE
```

Create a unique constraint on the label `Person` and property `name`. If any other node with that label is updated or created with a `name` that already exists, the write operation will fail. This constraint will create an accompanying index.

```
DROP CONSTRAINT ON (p:Person)
       ASSERT p.name IS UNIQUE
```

Drop the unique constraint and index on the label `Person` and property `name`.



Neo4j Cypher Refcard 2.1.2

Patterns

```
(n)-->(m)
```
A relationship from `n` to `m` exists.

```
(n:Person)
```
Matches nodes with the label `Person`.

```
(n:Person:Swedish)
```
Matches nodes which have both `Person` and `Swedish` labels.

```
(n:Person {name: {value}})
```
Matches nodes with the declared properties.

```
(n:Person)-->(m)
```
Node `n` labeled `Person` has a relationship to `m`.

```
(n)--(m)
```
A relationship in any direction between `n` and `m`.

```
(m)<-[:KNOWS]-(n)
```
A relationship from `n` to `m` of type `KNOWS` exists.

```
(n)-[:KNOWS|LOVES]->(m)
```
A relationship from `n` to `m` of type `KNOWS` or `LOVES` exists.

```
(n)-[r]->(m)
```
Bind an identifier to the relationship.

```
(n)-[*1..5]->(m)
```
Variable length paths.

```
(n)-[*]->(m)
```
Any depth. See the performance tips.

```
(n)-[:KNOWS]->(m {property: {value}})
```
Match or set properties in `MATCH`, `CREATE`, `CREATE UNIQUE` or `MERGE` clauses.

```
shortestPath((n1:Person)-[*..6]-(n2:Person))
```
Find a single shortest path.

```
allShortestPaths((n1:Person)-->(n2:Person))
```
Find all shortest paths.

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 label and property.

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

```
MATCH (n:Person)
```

Maps

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

```
MERGE (p:Person {name: {map}.name})
ON CREATE SET p={map}
```
Maps can be passed in as parameters and used as map or by accessing keys.

```
MATCH (matchedNode:Person)
RETURN matchedNode
```
Nodes and relationships are returned as maps of their data.

```
map.name, map.age, map.children[0]
```
Map entries can be accessed by their keys. Invalid keys result in an error.

Relationship Functions

```
type(a_relationship)
```
String representation of the relationship type.

```
startNode(a_relationship)
```
Start node of the relationship.

```
endNode(a_relationship)
```
End node of the relationship.

```
id(a_relationship)
```
The internal id of the relationship.

Collection Predicates

```
all(x IN coll WHERE has(x.property))
```
Returns `true` if the predicate is `TRUE` for all elements of the collection.

```
any(x IN coll WHERE has(x.property))
```
Returns `true` if the predicate is `TRUE` for at least one element of the collection.

```
none(x IN coll WHERE has(x.property))
```
Returns `TRUE` if the predicate is `FALSE` for all elements of the collection.

```
single(x IN coll WHERE has(x.property))
```
Returns `TRUE` if the predicate is `TRUE` for exactly one element in the collection.

Path Functions

```
length(path)
```
The length of the path.

```
nodes(path)
```
The nodes in the path as a collection.

```
relationships(path)
```
The relationships in the path as a collection.

```
MATCH path=(n)-->(m)
RETURN extract(x IN nodes(path) | x.prop)
```
Assign a path and process its nodes.

```
MATCH path = (begin) -[*]-> (end)
FOREACH
  (n IN rels(path) | SET n.marked = TRUE)
```
Execute a mutating operation for each relationship of a path.

Collection Functions

```
length({coll})
```
Length of the collection.

```
head({coll}), last({coll}), tail({coll})
```
`head` returns the first, `last` the last element of the collection. `tail` the remainder of the collection. All return null for an empty collection.

```
[x IN coll WHERE x.prop <> {value} | x.prop]
```
Combination of filter and extract in a concise notation.

```
extract(x IN coll | x.prop)
```
A collection of the value of the expression for each element in the orignal collection.

```
filter(x IN coll WHERE x.prop <> {value})
```
A filtered collection of the elements where the predicate is `TRUE`.

```
reduce(s = "", x IN coll | s + x.prop)
```
Evaluate expression for each element in the collection, accumulate the results.

```
FOREACH (value IN coll |
 CREATE (:Person {name:value}))
```
Execute a mutating operation for each element in a collection.

Aggregation

```
count(*)
```
The number of matching rows.

Matches nodes labeled as `Person`.

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

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

`labels(n)`
Labels of the node.

`REMOVE n:Person`
Remove label from node.

Predicates

`n.property <> {value}`
Use comparison operators.

`has(n.property)`
Use functions.

`n.number >= 1 AND n.number <= 10`
Use boolean operators to combine predicates.

`n:Person`
Check for node labels.

`identifier IS NULL`
Check if something is `NULL`.

`NOT has(n.property) OR n.property = {value}`
Either property does not exist or predicate is `TRUE`.

`n.property = {value}`
Non-existing property returns `NULL`, which is not equal to anything.

`n.property =~ "Tob.*"`
Regular expression.

`(n)-[:KNOWS]->(m)`
Make sure the pattern has at least one match.

`NOT (n)-[:KNOWS]->(m)`
Exclude matches to `(n)-[:KNOWS]->(m)` from the result.

`n.property IN [{value1}, {value2}]`
Check if an element exists in a collection.

Functions

`coalesce(n.property, {defaultValue})`
The first non-`NULL` expression.

`timestamp()`
Milliseconds since midnight, January 1, 1970 UTC.

`id(node_or_relationship)`
The internal id of the relationship or node.

`toInt({expr})`
Converts the given input in an integer if possible; otherwise it returns `NULL`.

`toFloat({expr})`
Converts the given input in a floating point number if possible; otherwise it returns `NULL`.

Mathematical Functions

`abs({expr})`
The absolute value.

`rand()`
A random value. Returns a new value for each call. Also useful for selecting subset or random ordering.

`round({expr})`
Round to the nearest integer, `ceil` and `floor` find the next integer up or down.

`sqrt({expr})`
The square root.

`sign({expr})`
`0` if zero, `-1` if negative, `1` if positive.

`sin({expr})`
Trigonometric functions, also `cos`, `tan`, `cot`, `asin`, `acos`, `atan`, `atan2`, `haversin`.

`degrees({expr}), radians({expr}), pi()`
Converts radians into degrees, use `radians` for the reverse. `pi` for π.

`log10({expr}), log({expr}), exp({expr}), e()`
Logarithm base 10, natural logarithm, `e` to the power of the parameter. Value of `e`.

String Functions

`str({expression})`
String representation of the expression.

`replace({original}, {search}, {replacement})`
Replace all occurrences of `search` with `replacement`. All arguments are be expressions.

`substring({original}, {begin}, {sub_length})`
Get part of a string. The `sub_length` argument is optional.

```
left({original}, {sub_length}),
  right({original}, {sub_length})
```
The first part of a string. The last part of the string.

```
trim({original}), ltrim({original}),
  rtrim({original})
```
Trim all whitespace, or on left or right side.

`upper({original}), lower({original})`
UPPERCASE and lowercase.

`split({original}, {delimiter})`
Split a string into a collection of strings.

START

`START n=node(*)`
Start from all nodes.

`START n=node({ids})`
Start from one or more nodes specified by id.

`START n=node({id1}), m=node({id2})`
Multiple starting points.

`START n=node:nodeIndexName(key={value})`
Query the index with an exact query. Use `node_auto_index` for the automatic index.

`count(identifier)`
The number of non-`NULL` values.

`count(DISTINCT identifier)`
All aggregation functions also take the `DISTINCT` modifier, which removes duplicates from the values.

`collect(n.property)`
Collection from the values, ignores `NULL`.

`sum(n.property)`
Sum numerical values. Similar functions are `avg`, `min`, `max`.

`percentileDisc(n.property, {percentile})`
Discrete percentile. Continuous percentile is `percentileCont`. The `percentile` argument is from `0.0` to `1.0`.

`stdev(n.property)`
Standard deviation for a sample of a population. For an entire population use `stdevp`.

Upgrading

With Neo4j 2.0 several Cypher features in version 1.9 have been deprecated or removed.

- `START` is optional.
- `MERGE` will take `CREATE UNIQUE`'s role for the unique creation of patterns. Note that they are not the same, though.
- Optional relationships are handled by `OPTIONAL MATCH`, not question marks.
- Non-existing properties return `NULL`, `n.prop?` and `n.prop!` have been removed.
- The separator for collection functions changed from `:` to `|`.
- Paths are no longer collections, use `nodes(path)` or `rels(path)`.
- Parentheses around nodes in patterns are no longer optional.
- `CREATE a={property:’value’}` has been removed.
- Use `REMOVE` to remove properties.
- Parameters for index-keys and nodes in patterns are no longer allowed.
- To still use the older syntax, prepend your Cypher statement with `CYPHER 1.9`.

CREATE UNIQUE

```
CREATE UNIQUE
    (n)-[:KNOWS]->(m {property: {value}})
```
Match pattern or create it if it does not exist. The pattern can not include any optional parts.