Lecture 10

Graph Databases, Neo4J

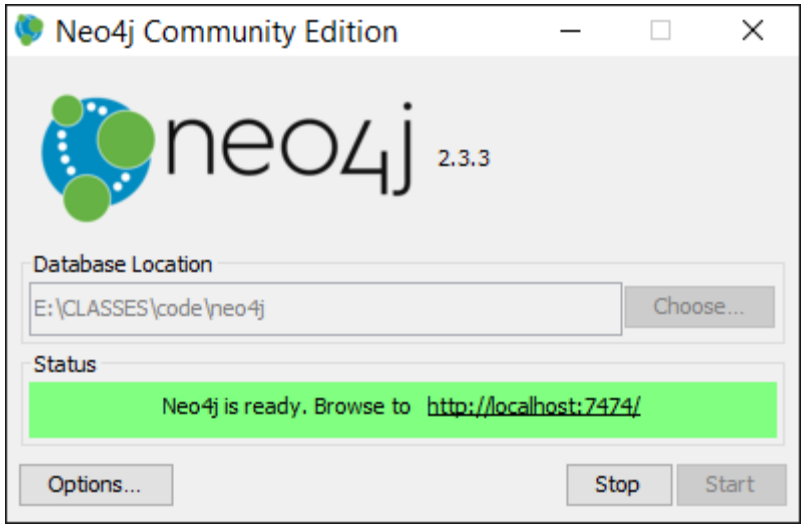
Zoran B. Djordjević

References

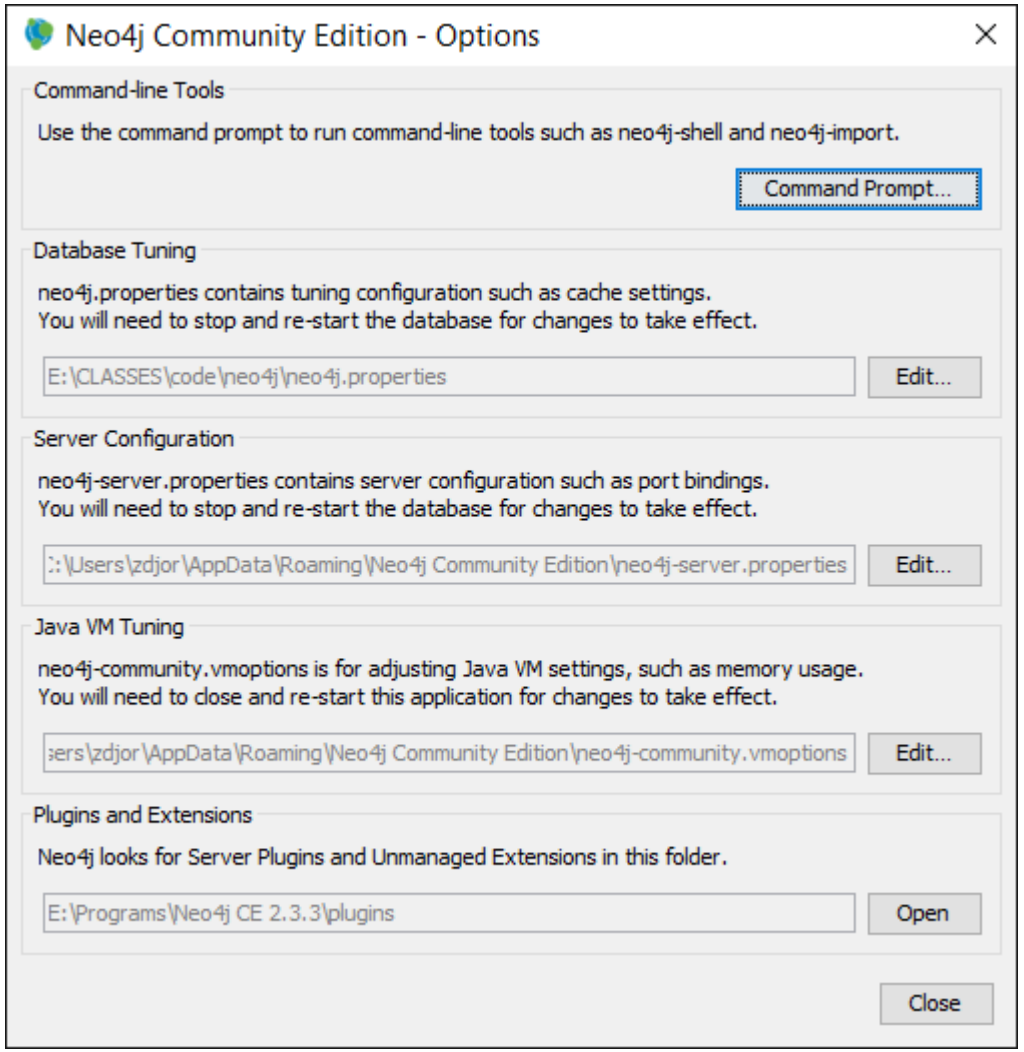
- These slides follow to a great measure Neo4J v2.3.3 User Manual:
  http://neo4j.com/docs/stable/

Installing Neo4J Community Edition

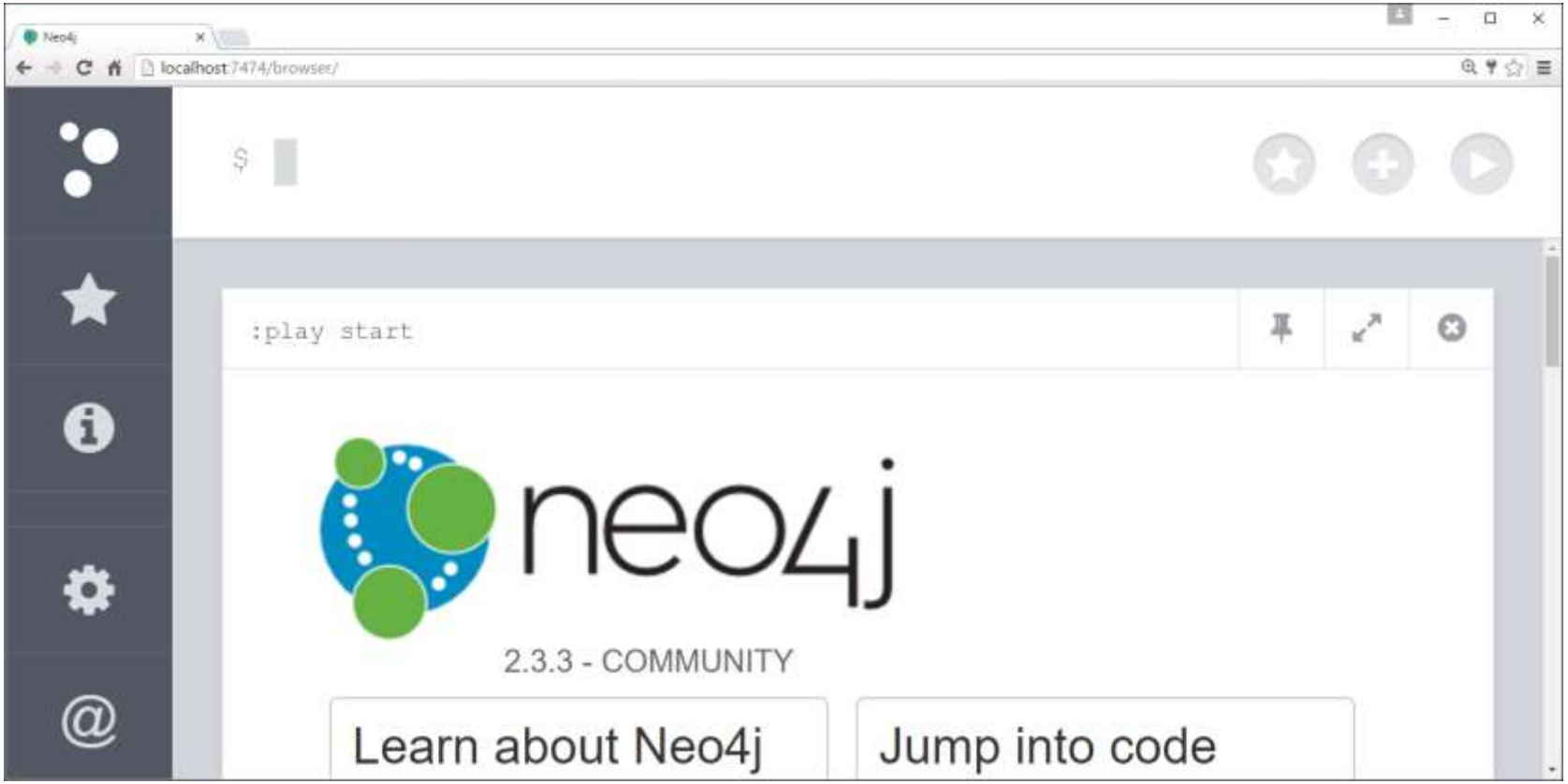
- Go to http://neo4j.com/download-thanks/?edition=community
- Select your operating system: Mac OS, Linux, Windows.
- Download neo4j-community_windows-x64_2_3_3.exe or similar
- Run the installation
- Select the directory
- Start



Properties files



Port 7474, neo4j/admin



Neo4J

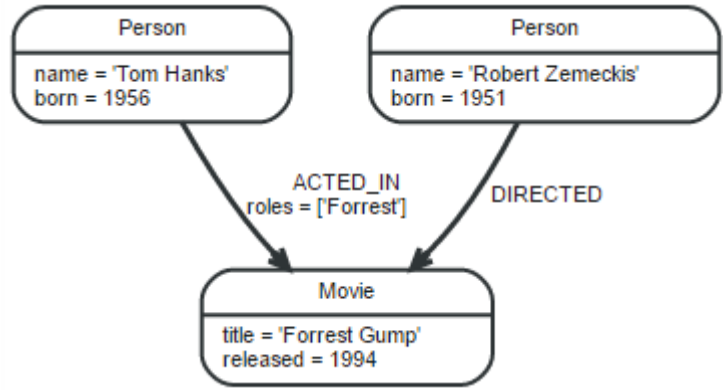
- As a robust, scalable and high-performance database, Neo4j is suitable for full enterprise deployment.
- Neo4J features:
  - true ACID transactions,
  - high availability,
  - scales to billions of nodes and relationships,
  - high speed querying through traversals,
  - declarative graph query language.
- Proper ACID behavior is the foundation of data reliability. Neo4j enforces that all operations that modify data occur within a transaction, guaranteeing consistent data. This robustness extends from single instance embedded graphs to multi-server high availability installations.
- Reliable graph storage can easily be added to any application. A graph can scale in size and complexity as the application evolves, with little impact on performance. Whether starting new development, or augmenting existing functionality, Neo4j is only limited by physical hardware.
- A single server instance can handle a graph of billions of nodes and relationships. When data throughput is insufficient, the graph database can be distributed among multiple servers in a high availability configuration
- The graph database storage shines when storing richly-connected data. Querying is performed through traversals, which can perform millions of traversal steps per second. A traversal step resembles a *join* in a RDBMS.

Key Concepts

- Key concepts in a graph database are:
  - Nodes
  - Relationships
  - Properties
  - Labels
  - Traversal
  - Paths
  - Schema

- A graph records data in nodes and relationships. Both can have properties.
- This is sometimes referred to as the *Property Graph Model*.

- A graph database stores data in a graph, the most generic of data structures, capable of elegantly representing any kind of data in a highly accessible way. The following is an example of a simple graph:



- This graph contains three nodes, two with label Person and one with Label Movie.
- One person is related to the move since he DIRECTED it and the other ACTED_IN it.
- Nodes and relationships have properties.

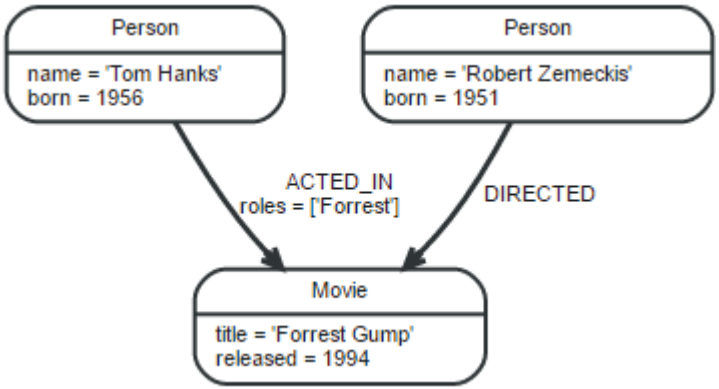
Nodes

- The fundamental units that form a graph are nodes and relationships. In Neo4j, both nodes and relationships can contain properties.
- Nodes are often used to represent entities, but depending on the domain relationships may be used for that purpose as well.
- Apart from properties and relationships, nodes can also be labeled with zero or more labels.
- The simplest possible graph is a single Node. A Node can have zero or more named values referred to as properties. Let's start out with one node that has a single property named title:

- More complex graphs could have two or more nodes. We could extend the previous graph with two more nodes and one more property on the first node:

Relationships

- Relationships organize the nodes by connecting them. A relationship connects two nodes — a start node and an end node. Just like nodes, relationships can have properties.
- Relationships between nodes are a key part of a graph database. They allow for finding related data. Just like nodes, relationships can have properties.
- A relationship connects two nodes, and is guaranteed to have valid start and end nodes.
- Relationships organize nodes into arbitrary structures, allowing a graph to resemble a list, a tree, a map, or a compound entity — any of which can be combined into yet more complex, richly inter-connected structures.



- Graph to the left uses ACTED_IN and DIRECTED as relationship types.
- The roles property on the ACTED_IN relationship has an array value with a single item in it.
- ACTED_IN relationship has Tom Hanks node as start node and Forrest Gump as end node.
- Tom Hanks node has an outgoing relationship, while the Forrest Gump node has an incoming relationship

- Relationships are equally well traversed in either direction.
- There is little need to add duplicate relationships in the opposite direction.

Relationships

- While relationships always have a direction, you can ignore the direction where it is not useful in your application. A node could have relationships to itself as well.
- We use relationship direction and type to extract information:

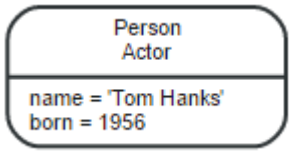
What we want to know	Start from	Relationship type	Direction
get actors in movie	movie node	ACTED_IN	incoming
get movies with actor	person node	ACTED_IN	outgoing
get directors of movie	movie node	DIRECTED	incoming
get movies directed by	person node	DIRECTED	outgoing

Properties

- Both nodes and relationships can have properties.
- Properties are named values where the name is a string. The supported property values are:
  - Numeric values,
  - String values,
  - Boolean values,
  - Collections of any other type of value.
- NULL is not a valid property value.
- NULLs can instead be modeled by the absence of a key.

Labels

- Labels assign roles or types to nodes.
- A label is a named graph construct that is used to group nodes into sets; all nodes labeled with the same label belongs to the same set. Many database queries can work with these sets instead of the whole graph, making queries easier to write and more efficient to execute. A node may be labeled with any number of labels, including none, making labels an optional addition to the graph.
- Labels are used when defining constraints and adding indexes for properties
- An example would be a label named User that you label all your nodes representing users with. With that in place, you can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.
- However, you can use labels for much more. Labels can be added and removed during runtime, and can be used to mark temporary states for your nodes. You might create an Offline label for phones that are offline, a Happy label for happy pets, and so on.
- Person and Movie are two labels in our graph.
- A node could have two or more labels.
- Tom Hanks node could be labeled with Person and Actor



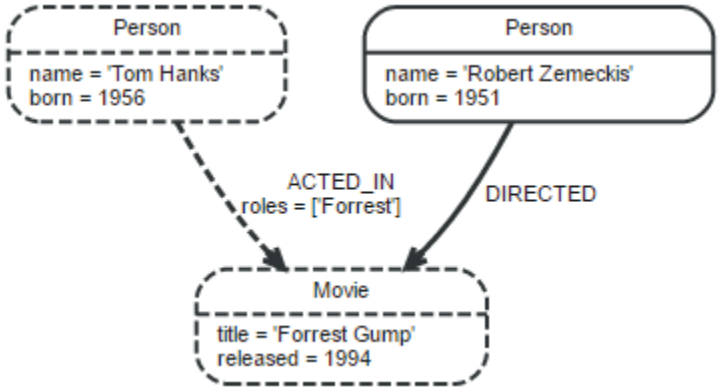
Label Names

- Any non-empty Unicode string can be used as a label name.
- In Cypher, you may need to use the backtick (`) syntax to avoid clashes with Cypher identifier rules or to allow non-alphanumeric characters in a label.
- By convention, labels are written with CamelCase notation, with the first letter in upper case. For instance, User or CarOwner.

- Labels have an id space of an `int`, meaning the maximum number of labels the database can contain is roughly 2 billion.

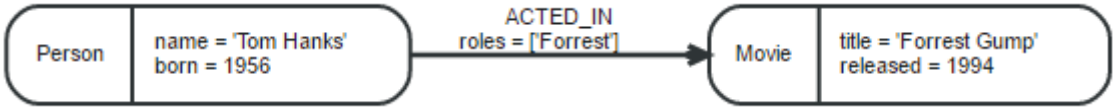
Traversal

- A traversal navigates through a graph to find paths.
- A traversal is how you query a graph, navigating from starting nodes to related nodes, finding answers to questions like “what music do my friends like that I don’t yet own,” or “if this power supply goes down, what web services are affected?”
- Traversing a graph means visiting its nodes, following relationships according to some rules. In most cases only a subgraph is visited, as you already know where in the graph the interesting nodes and relationships are found.
- Cypher provides a declarative way to query the graph powered by traversals and other techniques. See Cypher Query Language for more information.
- When writing server plugins or using Neo4j embedded, Neo4j provides a callback based traversal API which lets you specify the traversal rules. At a basic level there’s a choice between traversing breadth- or depth-first.
- To find out which movies Tom Hanks acted in the traversal would start from the Tom Hanks node, follow any ACTED_IN relationships connected to the node, and end up with Forrest Gump as the result (see the dashed lines)

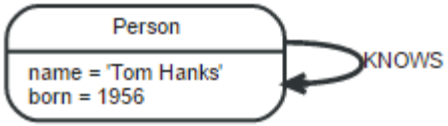


Paths

- A path is one or more nodes with connecting relationships, typically retrieved as a query or traversal result.
- In the previous example, the traversal result could be returned as a path



- The path above has length one.
- The shortest possible path has length zero — that is it contains only a single node and no relationships.
- Self referenced node has path of length one.



Schema

- Neo4j is a schema-optional graph database.

- You can use Neo4j without any schema. Optionally you can introduce it in order to gain performance or modeling benefits. This allows a way of working where the schema does not get in your way until you are at a stage where you want to reap the benefits of having one.

Indexes

- Performance is gained by creating indexes, which improve the speed of looking up nodes in the database.
- Once you've specified which properties to index, Neo4j will make sure your indexes are kept up to date as your graph evolves. Any operation that looks up nodes by the newly indexed properties will see a significant performance boost.

- Indexes in Neo4j are eventually available. That means that when you first create an index the operation returns immediately. The index is populating in the background and so is not immediately available for querying. When the index has been fully populated it will eventually come online. That means that it is now ready to be used in queries.

- If something should go wrong with the index, it can end up in a failed state. When it is failed, it will not be used to speed up queries. To rebuild it, you can drop and recreate the index. Look at logs for clues about the failure.

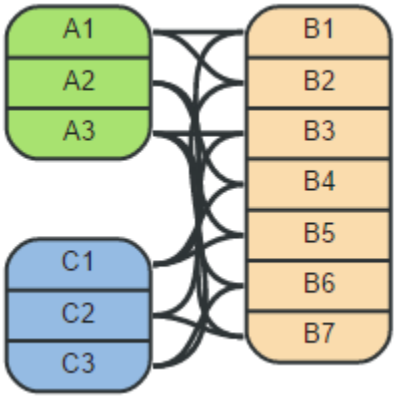
- You can track the status of your index by asking for the index state through the API you are using. Note, however, that this is not yet possible through Cypher.

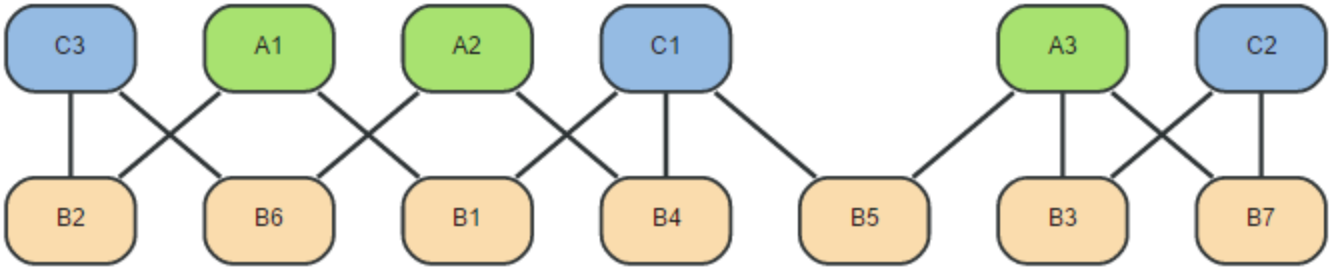
Constraints

- Neo4j can help you keep your data clean. It does so using constraints, that allow you to specify the rules for what your data should look like. Any changes that break these rules will be denied.
- In most recent version 2.3.3, unique constraints are the only available constraint type.
- One can impose constraints through the different APIs.

Graph DBs vs. RDBMS

- Relational databases are optimized for aggregate data.
- Graph databases are optimized for connected data.
- Extraction of information in relational databases depends on so called primary-foreign key relationships. Those relationships could also have properties.
- Deep navigation through RDBMS models require multiple joins which are expensive performance wise.
- Graph databases are optimized for localizing and returning deep and branched graph objects.

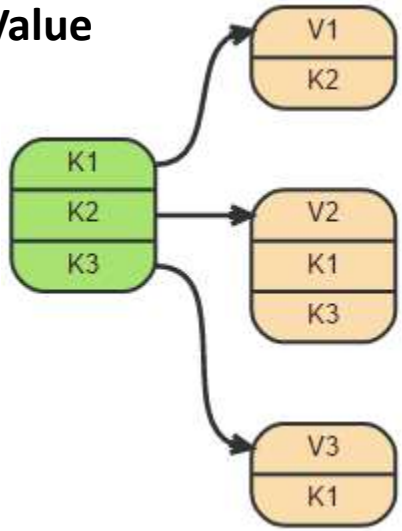


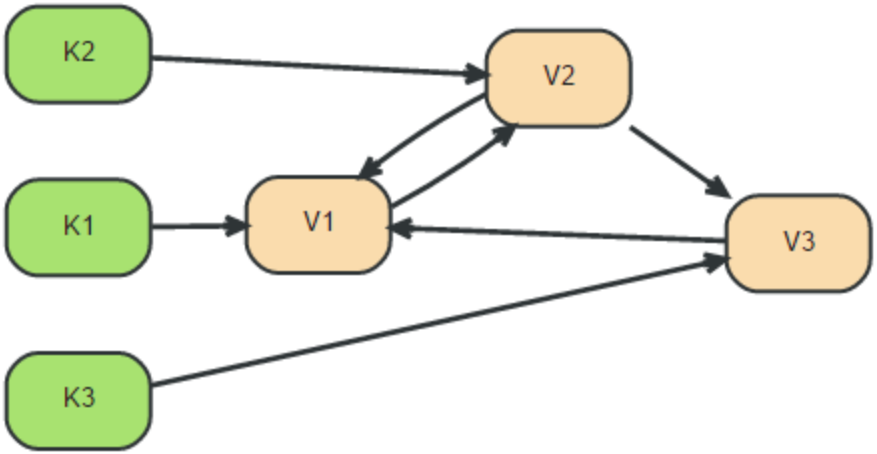


Graph DBs vs. Key-Value Stores

- A Key-Value model is great for lookups of simple values or lists. When the values are themselves interconnected, you've got a graph. Neo4j lets you elaborate the simple data structures into more complex, interconnected data.
- K* represents a key, V* a value. Note that some keys point to other keys as well as plain values.

Key-Value Store





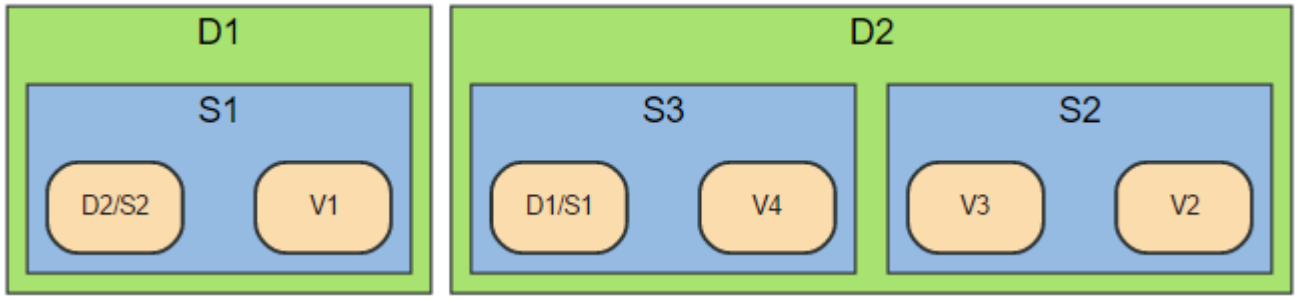
Graph Database as Key-Value Store

A Graph Database relates Column-Family

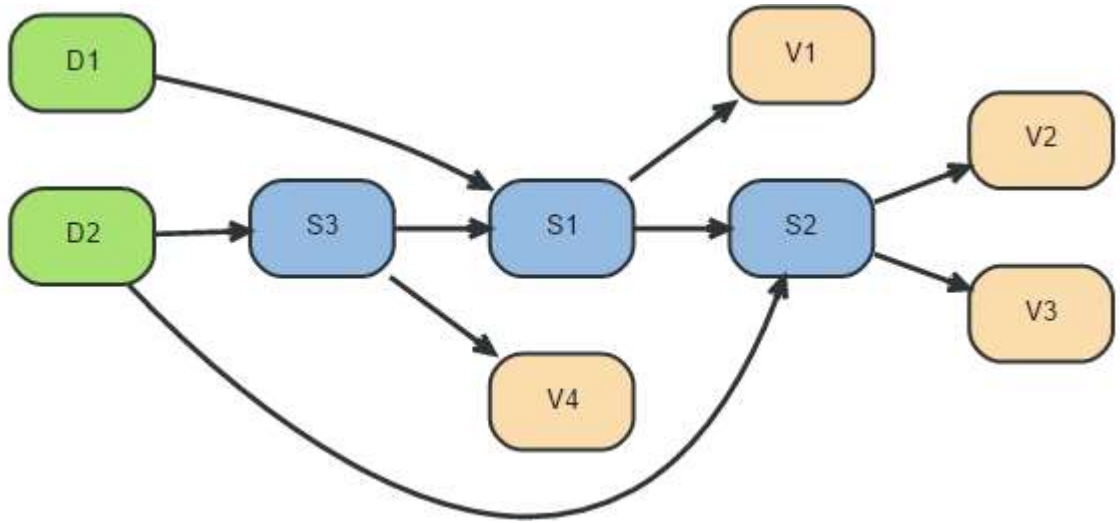
Column Family (BigTable-style) databases are an evolution of key-value, using "families" to allow grouping of rows. Stored in a graph, the families could become hierarchical, and the relationships among data becomes explicit.

Graph Database vs. Document Store

- The container hierarchy of a document database accommodates nice, schema-free data that can easily be represented as a tree. Which is of course a graph. Refer to other documents (or document elements) within that tree and you have a more expressive representation of the same data. When in Neo4j, those relationships are easily navigable.



D=Document,
S=Subdocument,
V=Value,
D2/S2 = reference to subdocument in (other) document



Graph Database as Document Store

Cypher

- Users of Neo4J, besides programming API-s need a convenience tool for inspection and manipulation of objects (graphs) stored in the database. Cypher is a language and a tool for ad hoc analysis.
- Cypher provides a convenient way to express queries and other Neo4j actions. Although Cypher is particularly useful for exploratory work, it is fast enough to be used in production.
- Cypher performs the following:
  - Parse and validate the query.
  - Generate the execution plan.
  - Locate the initial node(s).
  - Select and traverse relationships.
  - Change and/or return values.
- Query Preparation is accomplished by Parsing and validating the query. Subsequent steps include generating an optimal search strategy what can be far more challenging.
- The execution plan must tell the database how to locate initial node(s), select relationships for traversal, etc. This involves tricky optimization problems (eg, which actions should happen first),

Cypher Concepts

- Like SQL (used in relational databases), Cypher is a textual, declarative query language. It uses a form of ASCII art to represent graph-related patterns.
- SQL-like clauses and keywords (eg, MATCH, WHERE, DELETE) are used to combine these patterns and specify desired actions.
- This combination tells Neo4j which patterns to match and what to do with the matching items (e.g., nodes, relationships, paths, collections).
- As a declarative language, Cypher does *not* tell Neo4j how to find nodes, traverse relationships, etc. (This level of control is available from Neo4j's Java APIs.

Locate Initial node, Traverse

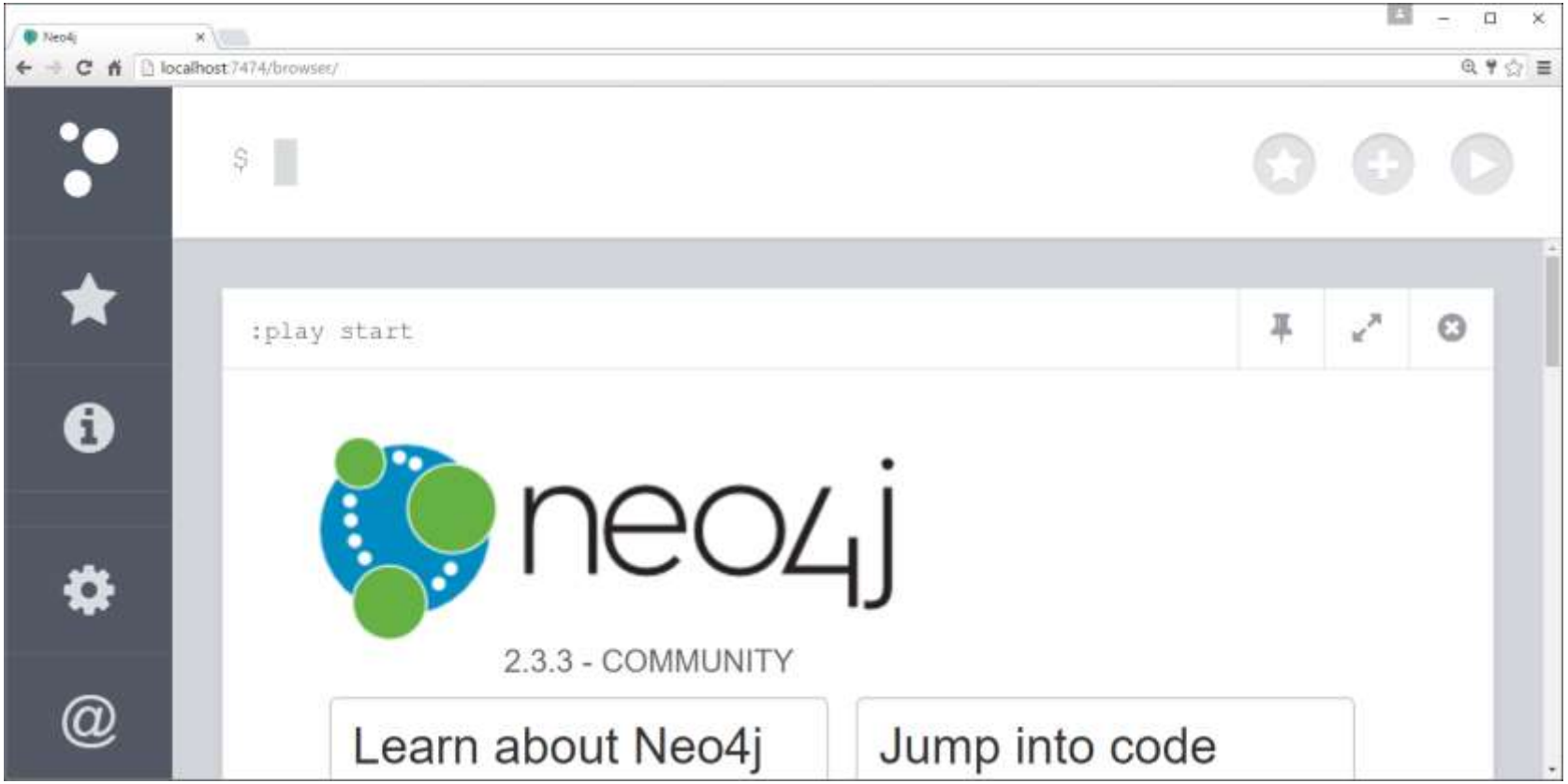
Location of Initial Node

- Neo4j is highly optimized for traversing property graphs. Under ideal circumstances, it can traverse millions of nodes and relationships per second, following chains of pointers in the computer's memory.
- However, before traversal can begin, Neo4j must know one or more starting nodes. Unless the user (or, more likely, a client program) can provide this information, Neo4j will have to search for these nodes.
- A "brute force" search of the database can be *very* time consuming. Every node must be examined to see if it has the property, then to see if the value meets the desired criteria.
- To avoid this effort, Neo4j creates and uses indexes. Neo4j uses a separate index for each label/property combination.

Traversal and actions

- Once the initial nodes are determined, Neo4j can traverse portions of the graph and perform any requested actions.
- The execution plan helps Neo4j to determine which nodes are relevant, which relationships to traverse, etc.

Port 7474, neo4j/admin



Nodes, Relationships and Patterns

- Nodes and relationships are simply low-level building blocks. The real strength of the Property Graph lies in its ability to encode *patterns* of connected nodes and relationships.
- A single node or relationship typically encodes very little information, but a pattern of nodes and relationships can encode arbitrarily complex ideas.
- Cypher, Neo4j's query language, is strongly based on patterns. Patterns are used to match desired graph structures.
- A simple pattern, which has only a single relationship, connects a pair of nodes (or, occasionally, a node to itself). For example, *a Person* LIVES_IN *a City* or *a City is* PART_OF *a Country*.
- Complex patterns, using multiple relationships, can express arbitrarily complex concepts and support a variety of interesting use cases.
- For example, we might want to match instances where *a Person* LIVES_IN *a Country*. The following Cypher code combines two simple patterns into a (mildly) complex pattern which performs this match:

```
(:Person) -[:LIVES_IN]-> (:City) -[:PART_OF]-> (:Country)
```

- Pattern recognition is fundamental to the way that the brain works. Humans are very good at working with patterns.
- When patterns are presented visually (eg, in a diagram or map), humans can use them to recognize, specify, and understand concepts. As a pattern-based language, Cypher takes advantage of this capability.

Node Syntax

- Cypher uses a pair of parentheses (usually containing a text string) to represent a node, eg: (), (foo).

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

- The simplest form, (), represents an anonymous, uncharacterized node. If we want to refer to the node elsewhere, we can add an identifier, eg: (matrix).
- Identifirrs are restricted (ie, scoped) to a single statement: an identifier may have different (or no) meaning in another statement.
- The Movie label (prefied in use with a colon) declares the node's type. This restricts the pattern, keeping it from matching (say) a structure with an Actor node in this position.
- Neo4j's node indexes also use labels: each index is specified to the combination of a label and a property.
- The node's properties (eg, title) are represented as a list of key/value pairs, enclosed within a pair of braces, eg: {...}. Properties
- can be used to store information and/or restrict patterns. For example, we could match nodes whose title is "The Matrix".

Relationship Syntax

- Cypher uses a pair of dashes (--) to represent an undirected relationship. Directed relationships have an arrowhead at one end (eg, <--, -->). Bracketed expressions (eg: [...]) can be used to add details. This may include identifiers, properties, and/or type information, eg:

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

- The syntax and semantics found within a relationship's bracket pair are very similar to those used between a node's parentheses.
- An identifier (eg, role) can be defined, to be used elsewhere in the statement.
- The relationship's type (eg, ACTED_IN) is analogous to the node's label.
- Like with node labels, the relationship type ACTED_IN is added as a symbol, pre xed with a colon: :ACTED_IN
- The properties (eg, roles) are entirely equivalent to node properties. (Note that the value of a property may be an array.) Identifiers (eg, role) can
- be used elsewhere in the statement to refer to the relationship.

Pattern Syntax, Clauses

- Combining the syntax for nodes and relationships, we can express patterns.

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

- Cypher allows patterns to be assigned to identifiers. This allow the matching paths to be inspected, used in other expressions, etc.

```
acted_in = (:Person)-[:ACTED_IN]->(:Movie)
```

- The acted_in variable would contain two nodes and the connecting relationship for each path that was found or created. There are a number of functions to access details of a path, including `nodes(path)`, `rels(path)` (same as relationships(path)), and `length(path)`.

Clauses

- Cypher statements typically have multiple *clauses*, each of which performs a task,
  - create and match patterns in the graph
  - filer, project, sort, or paginate results
  - connect/compose partial statements
- By combining Cypher clauses, we can compose more complex statements that express what we want to know or create. Neo4j then figures out how to achieve the desired goal in an efficient manner.

Creating Data

- You can create individual nodes and relationships:

```
CREATE (:Movie { title:"The Matrix",released:1997 })
Added 1 label, created 1 node, set 2 properties, statement executed in 353 ms
```

- We could ask for the node to be returned to us, as well:

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

- Cypher browser tool would return:

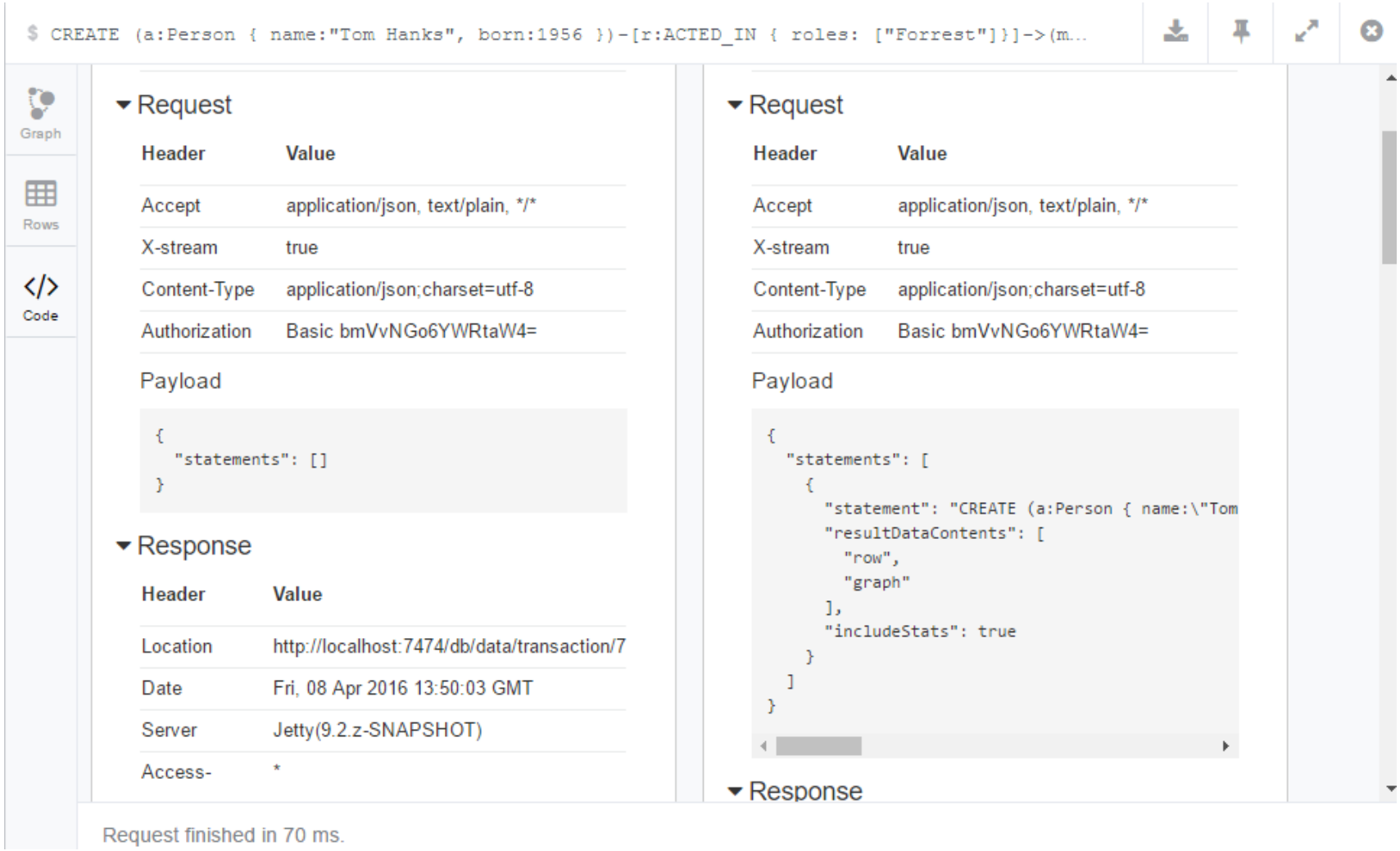
Creating Multiple Elements

- If we want to create more than one element, we can separate the elements with commas or use multiple CREATE statements without comma separation
- We can of course also create more complex structures, including an ACTED_IN relationship with information about the character, or DIRECTED relationship for the director.

```
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
```

</> Code Introspection

- If you click on </> symbol you will get an insight into traffic to the server and back



Fetching Values, `MATCH`-ing Patterns

- Matching patterns is a task for the MATCH statement. We pass the same kind of patterns we've used so far to MATCH to describe which we're looking for.
- A MATCH statement will search for the patterns we specify and return one row per successful pattern match.
- We can ask for all nodes labeled with the Movie label or a Person named Keanu Reeves

```
MATCH (m:Movie) RETURN m;
MATCH (p:Person { name:"Keanu Reeves" })  RETURN p;
```

- We can also find more interesting connections, like for instance the movies titles that *Tom Hanks* acted in and the roles he played.

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

m.title	r.roles
Forrest Gump	[Forrest]

- We returned the properties of the nodes and relationships that we were interested in. You can access them everywhere via a dot notation `identifer.property`

Extending the Graph, Attaching Structures

- To extend the graph with new information, we first match the existing connection points and then attach the newly created nodes to them with relationships.
- Adding *Cloud Atlas* as a new movie for *Tom Hanks* could be achieved like this:

```
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
```

- We can assign identifiers to both nodes and relationships and use them later on, no matter if they were created or matched.
- A tricky aspect of the combination of MATCH and CREATE is that we get *one row per matched pattern*. This causes subsequent CREATE statements to be executed once for each row. In many cases this is what you want. If that's not intended, move the CREATE statement before the MATCH, or change the cardinality of the query.

Completing Patterns, `MERGE` (update)

- Whenever we get data from external systems or are not sure if certain information already exists in the graph, we want to be able to express a repeatable (idempotent) update operation.
- In Cypher MERGE has this function. It acts like a combination of MATCH *or* CREATE, which checks for the existence of data before creating it.
- With MERGE you define a pattern to be found or created. Usually, as with MATCH you only want to include the key property to look for in your core pattern. MERGE allows you to provide additional properties you want to set ON CREATE.
- For example, even if graph already had *Cloud Atlas* we could merge it in again:

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

- Clause `SET m.realeased = …` acted as `UPDATE` statement of standard SQL
- MERGE makes sure that you can't create duplicate information or structures, but it comes with the cost of needing to check for existing matches. Especially on large graphs it can be costly to scan a large set of labeled nodes for a certain property. You can alleviate some of that by creating supporting indexes or constraints.

MERGE

- MERGE can also assert that a relationship is only created once. For that to work you *have to pass in* both nodes from a previous pattern match.

```
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
```

- In case the direction of a relationship is arbitrary, you can leave off the arrowhead. MERGE will then check for the relationship in either direction, and create a new directed relationship if no matching relationship was found.

Creating Branching Points

- If you choose to pass in only one node from a preceding clause, MERGE offers an interesting functionality. It will then only match within the direct neighborhood of the provided node for the given pattern, and, if not found create it. This can be used for creating for 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
```

Deleting Nodes and Relationships

- To delete unattached nodes with label Label, use DELETE clause:

```
MATCH (n:Label) DELETE n;
```

- To delete all nodes and relationships you can do this:

```
MATCH (n) DETACH DELETE n;
```

Deleted 6 nodes, deleted 6 relationships, statement executed in 49 ms.

Selecting Results we Need

- In normal SQL we perform many operations in order to restrict and select results we need. Cypher performs almost all of those operations:
  - Filtering
  - Returning
  - Aggregating
  - Ordering and Paginating
  - Collecting Aggregation
- Let us create (extend) a more elaborate graph, first:

```
CREATE (matrix:Movie { title:"The Matrix",released:1997 })
CREATE (cloudAtlas:Movie { title:"Cloud Atlas",released:2012 })
CREATE (forrestGump:Movie { title:"Forrest Gump",released:1994 })
CREATE (keanu:Person { name:"Keanu Reeves", born:1964 })
CREATE (robert:Person { name:"Robert Zemeckis", born:1951 })
CREATE (tom:Person { name:"Tom Hanks", born:1956 })
CREATE (tom)-[:ACTED_IN { roles: ["Forrest"]}]->(forrestGump)
CREATE (tom)-[:ACTED_IN { roles: ['Zachry']}]->(cloudAtlas)
CREATE (robert)-[:DIRECTED]->(forrestGump)
```

Filtering, `WHERE` Clause

- Quite often there are conditions in play for what we want to see. Similar to in *SQL* those filter conditions are expressed in a WHERE clause.
- WHERE clause allows use of any number of Boolean expressions (predicates) combined with AND, OR, XOR and NOT. The simplest predicates are comparisons.

```
MATCH (m:Movie)
WHERE m.title = "The Matrix‟ RETURN m
```

- Or in a slightly more compact notation

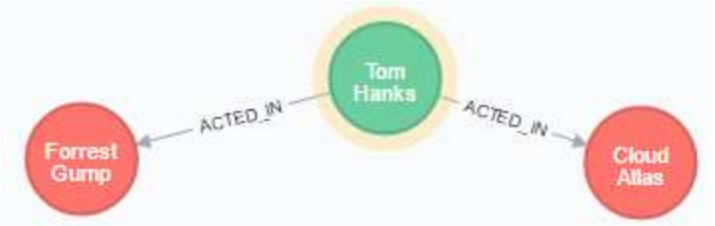
```
MATCH (m:Movie { title: "The Matrix" }) RETURN m
```

- `WHERE` clause could include numeric comparisons, matching regular expressions and checking the existence of values within a collection.

```
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
```

- We can use patterns as predicates. A pattern predicate restricts the current result set. It only allows the paths to pass that satisfy the additional patterns (or NOT)

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



`RETURN`-ing Results

- RETURN clause can actually return any number of expressions and not only nodes, relationships, or paths .
- What are actually expressions in Cypher? The simplest expressions are literal values like numbers, strings and arrays as `[1,2,3]`, and maps like `{name:"Tom Hanks", born:1964, movies:["Forrest Gump", ...], count:13}`.
- We can access individual properties of any node, relationship, or map with a dot-syntax like n.name. Individual elements or slices of arrays can be retrieved with subscripts like names[0] or movies[1..-1].
- Each function evaluation like `length(array)`, `toInt("12")`, `substring("2014-07-01",0,4)`, or `coalesce(p.nickname,"n/a")` is also an expression.
- By default the expression itself will be used as label for the column, in many cases you want to alias that with a more understandable name using `expression AS alias`. You can later on refer to that column using its alias.

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

- Here we used key word `DISTINCT` to eliminate duplicate rows

Grouping, Aggregating

- In many cases you want to aggregate or group the data that you encounter while traversing patterns in your graph. Aggregation happens in the RETURN clause while computing your final results.
- Many common aggregation functions are supported, e.g. `count, sum, avg, min, and max`, and several more.
- Counting the number of people in your database could be achieved by this:

```
MATCH (:Person) RETURN count(*) AS people
6
MATCH (p:Person) RETURN count(DISTINCT p.name) AS people
3
```

- NULL values are skipped during aggregation. For aggregating only unique values use DISTINCT, like in `count(DISTINCT role)`.
- *Cypher uses all non-aggregated columns as grouping keys*. Aggregation affects which data is still visible in ordering or later query parts.
- For example, to find out how often an actor and director worked together, you'd run this statement:

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)<-[:DIRECTED]-(director:Person)
RETURN actor,director,count(*) AS collaborations
```

actor	director	collaborations
Node[5]{name:"Tom Hanks",born:1956}	Node[4]{name:"Robert Zemeckis",born:1951}	1

Ordering and Pagination

- Ordering is imposed with `ORDER BY expression [ASC|DESC]` clause.
- If you return `person.name` you can still `ORDER BY person.age` as both are accessible from the person reference. You cannot order by things that you can't infer from the information you return. This is especially important with aggregation and DISTINCT return values as both remove the visibility of data that is aggregated.
- Pagination is achieved by use of `SKIP {offset} LIMIT {count}.`
- A common pattern is to aggregate for a count (score or frequency), order by it and only return the top-n entries.
- For example, the most prolific actors could be found as:

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

Collecting Aggregation

- The most helpful aggregation function is `collect()`, which, as the name says, collects all aggregated values into a *real* array or list. With COLLECT we don't loose the detail information while aggregating.
- Collect is well suited for retrieving the typical parent-child structures, where one core entity (parent, root or head) is returned per row with all it's dependent information in associated collections created with collect. This means there's no need to repeat the parent information per each child-row or even running 1+n statements to retrieve the parent and its children individually.
- To retrieve the cast of each movie in our database you could use this statement:

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

movie	cast	actors
Forrest Gump	[Tom Hanks]	1
Cloud Atlas	[Tom Hanks]	1

Composing Large Statements

- If you want to combine the results of two statements that have the same result structure, you can use UNION [ALL].
- For instance if you want to list both actors and directors without using the alternative relationship-type syntax `()-[:ACTED_IN|:DIRECTED]->()` we can do:

```
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
```

name	acted_in	title
Tom Hanks	ACTED_IN	Cloud Atlas
Tom Hanks	ACTED_IN	Forrest Gump
Robert Zemeckis	DIRECTED	Forrest Gump

Chain Statements with `WITH`

- It is possible to chain fragments of statements together, much like you would do within a data flow pipeline. Each fragment works on the output from the previous one and its results can feed into the next one.
- You use the WITH clause to combine the individual parts 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.
- You can use the same expressions, aggregations, ordering and pagination as in the RETURN clause. However, you *must* alias all columns as they would otherwise not be accessible. Only columns that you declare in your WITH clause is available in subsequent query parts.
- Below, we collect the movies someone appeared in, and then filter out (remove) those actors that appear in only one movie.

```
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
```

person.name	appearances	movies
Tom Hanks	2	[Cloud Atlas, Forrest Gump]

- In SQL, if you want to filter by an aggregated value, you would have to use HAVING. That's a single purpose clause for filtering aggregated information. In Cypher, WHERE can be used in both case

Labels, Constraints

- **Labels** are a convenient way to group nodes together. They are used to restrict queries, define constraints and create indexes.

- Unique constraints are used to guarantee uniqueness of a certain property on nodes with a specific label.
- These constraints are also used by the MERGE clause to make certain that a node only exists once.
- Let's start out adding a constraint. In this case we decided that all Movie node titles should be unique.

```
CREATE CONSTRAINT ON (movie:Movie) ASSERT movie.title IS UNIQUE
Added 1 constraint, statement executed in 314 ms.
```

- Note that adding the unique constraint will add an index on that property, so we won't do that separately. If we drop a constraint, and still want an index on the same property, we have to create such an index.
- Constraints can be added after a label is already in use, but that requires that the existing data complies with the constraints.

Indexes

- For a graph query to run fast, you don't need indexes, you only need them to find your starting points. The main reason for using indexes in a graph database is to find the starting points in the graph as fast as possible. After the initial index seek you rely on in-graph structures and the first class citizenship of relationships in the graph database to achieve high performance.
- In this case we want an index to speed up finding actors by name in the database:

```
CREATE INDEX ON :Actor(name)
Added 1 index, statement executed in 44 ms.
```

- If we add new data, indexes are applied automatically.

```
CREATE (actor:Actor { name:"Tom Hanks" }),(movie:Movie { title:'Sleepless
IN Seattle' }), (actor)-[:ACTED_IN]->(movie);
```

- Next time we ask for Tom Hanks node, the index will kick in behind the scenes to boost performance.

```
MATCH (actor:Actor { name: "Tom Hanks" })
RETURN actor;
```

Labels

- We can add more than one label to a node or relationship. For example, so far Tom Hanks was only an Actor. We could add another label, American:

```
MATCH (actor:Actor { name: "Tom Hanks" })
SET actor:American;
```

- We could remove Lables from nodes and relationships:

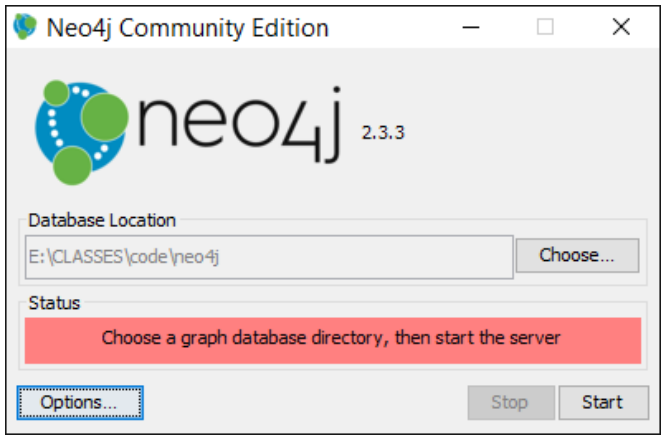
```
MATCH (actor:Actor { name: "Tom Hanks" })
REMOVE actor:American;
```

Loading Data, LOAD CSV

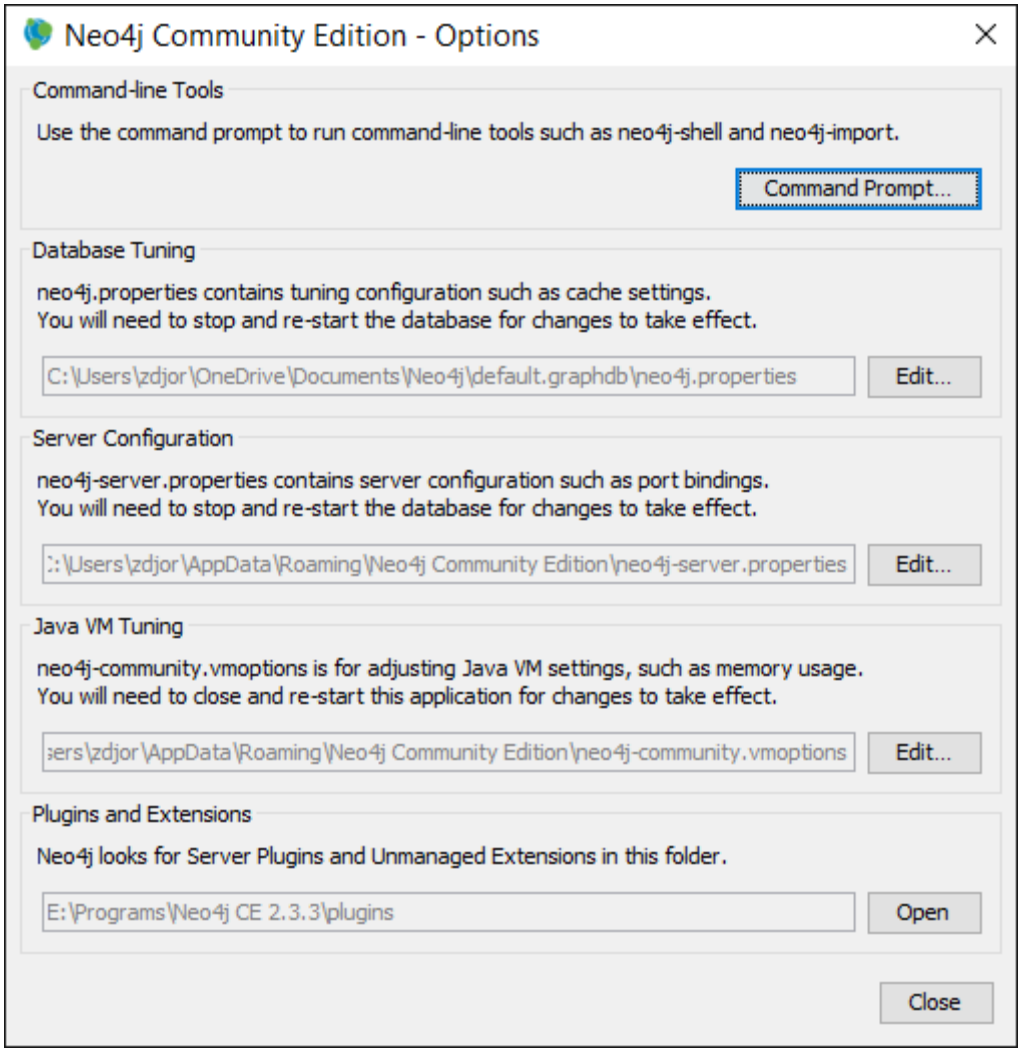
- Create statements we have seen so far are not practical if we have a very large number of objects. Often we need to use an existing data source that to drive the graph generation or modification process.
- Cypher provides an elegant built-in way to import tabular CSV data into graph structures.
- The LOAD CSV clause parses a local or remote file into a stream of rows which represent maps (with headers) or lists. Then you can use whatever Cypher operations you want to apply to either create nodes or relationships or to merge with existing graph structures.
- As CSV files usually represent either node- or relationship-lists, you run multiple passes to create nodes and relationships separately.
- We will use LOAD CSV command to import data from CSV files
- The URL of the CSV file is specified by using FROM followed by an arbitrary expression evaluating to the URL in question.
- It is required to specify an identifier for the CSV data using AS.
- LOAD CSV supports resources compressed with gzip, Deflate, as well as ZIP archives.
- CSV files can be stored on the database server and are then accessible using a file:/// URL. Alternatively, LOAD CSV also supports accessing CSV files via HTTPS, HTTP, and FTP.

Configure LOAD CSV

- Stop your server



- Select Options and then Edit
  Next to Server configuration file.
- Save edited file.
- Start the server.



neo4j-server.properties

- Server properties file needs new lines in red. Change dbms.security.auth_enabled as well to false

```
#*****************************************************************
# Server configuration
#*****************************************************************
# Require (or disable the requirement of) auth to access Neo4j
dbms.security.auth_enabled=false
# Allow CSV file loading
allow_file_urls=true
dbms.security.load_csv_file_url_root=csv-files
#
# HTTP Connector
#
# http port (for all data, administrative, and UI access)
org.neo4j.server.webserver.port=7474
# Let the webserver only listen on the specified IP. Default is localhost
(only
# accept local connections). Uncomment to allow any connection. Please see
the
# security section in the neo4j manual before modifying this.
#org.neo4j.server.webserver.address=0.0.0.0
#
# HTTPS Connector
#
# Turn https-support on/off
# org.neo4j.server.webserver.https.enabled=tru
```

Note on position of `csv-files` directory

- My database resides in `E:\CLASSES\code\neo4j`
- My configuration files in `c:\User\zdjor\AppData\Roaming\Neo4j Community Edition`
- My Neo4J application is in `E:\Programs\Neo4j CE 2.3.3`
- When specifying location of the directory for your CSV file, server (apparently) wants you to place it in `E:\Programs\Neo4j CE 2.3.3\bin` directory.
- My directory which I named `csv-files` has the full path

`E:\Programs\Neo4j CE 2.3.3\bin\csv-files`

- In that directory I placed files

persons.csv

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

movies.csv

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

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
```

movie_actor_role.csv

```
title;released;actor;born;characters
Back to the Future;1985;Michael J. Fox;1961;Marty McFly
Back to the Future;1985;Christopher Lloyd;1938;Dr. Emmet Brown
```

Loading data from CSV files

- To load data in those 4 files I run the following commands:

```
PERIODIC COMMIT LOAD CSV WITH HEADERS FROM "file:///persons.csv" AS line
MERGE (a:Person { id:line.id })
ON CREATE SET a.name=line.name;
```

Added 5 labels, created 5 nodes, set 10 properties, statement executed in 568 ms.

```
LOAD CSV WITH HEADERS FROM "file:///movies.csv" AS line
CREATE (m:Movie { id:line.id,title:line.title, released:toInt(line.year)});
```

Added 3 labels, created 3 nodes, set 9 properties, statement executed in 467 ms.

```
LOAD CSV WITH HEADERS FROM "file:///roles.csv" AS line
MATCH (m:Movie { id:line.movieId })
MATCH (a:Person { id:line.personId })
CREATE (a)-[:ACTED_IN { roles: [line.role]}]->(m);
```

(no changes, no rows)

```
LOAD CSV WITH HEADERS FROM "file:///movie_actor_roles.csv" AS line
FIELDTERMINATOR ";"
MERGE (m:Movie { title:line.title })
ON CREATE SET m.released = toInt(line.released)
MERGE (a:Person { name:line.actor })
ON CREATE SET a.born = toInt(line.born)
MERGE (a)-[:ACTED_IN { roles:split(line.characters,",")}]->(m);
```

Added 3 labels, created 3 nodes, set 8 properties, created 2 relationships, statement executed in 235 ms

Notes

- If you import a large amount of data (more than 10000 rows), it is recommended to prefix your LOAD CSV clause with a PERIODIC COMMIT hint. This allows Neo4j to regularly commit the import transactions to avoid memory issues.
- If you have your files on a remote location, you can load them through HTTP or HTTPS protocol, like:

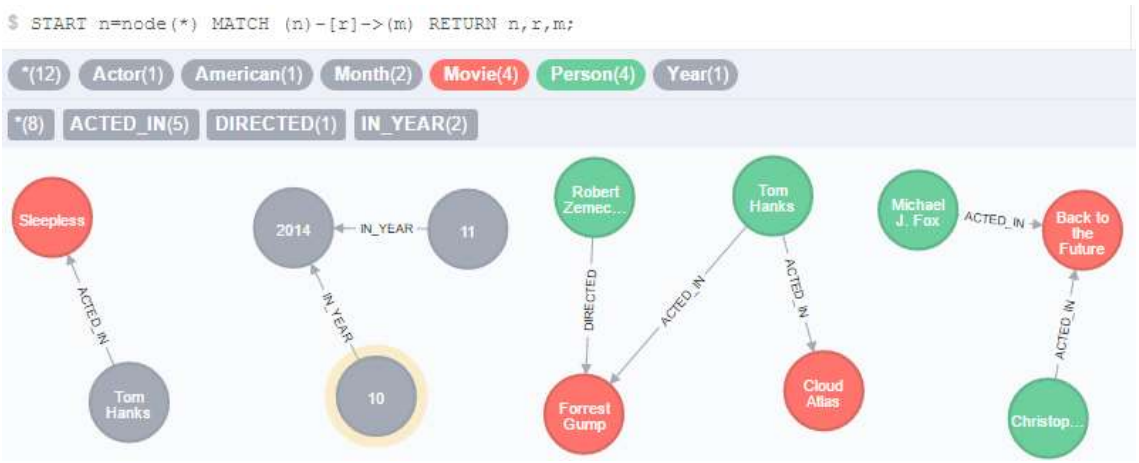
```
LOAD CSV WITH HEADERS FROM
"http://neo4j.com/docs/2.3.3/csv/intro/movies.csv" AS line
CREATE (m:Movie { id:line.id,title:line.title, released:toInt(line.year)});
```

- After all this work, we would like what we got. To display all nodes associated with some relationships we could ask:

```
START n=node(*) MATCH (n)-[r]->(m) RETURN n,r,m;
```

- If we want all the nodes whether they are in relationship or not, we could ask:

```
START n=node(*) RETURN n;
```



Data Structures

- Cypher can create and consume more complex data structures out of the box.
- You can create literal lists `( [1,2,3])` and maps `( {name: value})` within a statement.
- There are a number of functions that work with lists. For example, function `size(list)` returns the size of a list. Function `reduce()` runs an expression against the elements and accumulates the results.
- To begin with, collect the names of the actors per movie, and return two of them:

```
MATCH (movie:Movie)<-[:ACTED_IN]-(actor:Person)
RETURN movie.title AS movie, collect(actor.name)[0..2] AS two_of_cast;
```

movie	two_of_cast
Forrest Gump	[Tom Hanks]
Cloud Atlas	[Tom Hanks]
Back to the Future	[Christopher Lloyd, Michael J. Fox]

- In the last line we are accessing elements 0 to 2 of list collect(actor.name)[0..2].
- You can also access individual elements or slices of a list quickly with list[1] or list[5..-5]. Other functions one could use to access parts of a list are `head(list)`, `tail(list)` and `last(list)`

List Processing

- Oftentimes you want to process lists to filter, aggregate (reduce) or transform (extract) their values. Those transformations can be done within Cypher or in the calling code. This kind of list-processing can reduce the amount of data handled and returned, so it might make sense to do it within the Cypher statement.

```
WITH range(1,10) AS numbers
WITH extract(n IN numbers | n*n) AS squares
WITH filter(n IN squares WHERE n > 25) AS large_squares
RETURN reduce(a = 0, n IN large_squares | a + n) AS sum_large_squares;
```

sum_large_squares
330

- In a graph-query we can filter or aggregate collected values or work on array properties.

```
MATCH (m:Movie)<-[r:ACTED_IN]-(a:Person)
WITH m.title AS movie, collect({ name: a.name, roles: r.roles }) AS cast
RETURN movie, filter(actor IN cast WHERE actor.name STARTS WITH "M") Starts_M
```

Unwind Lists

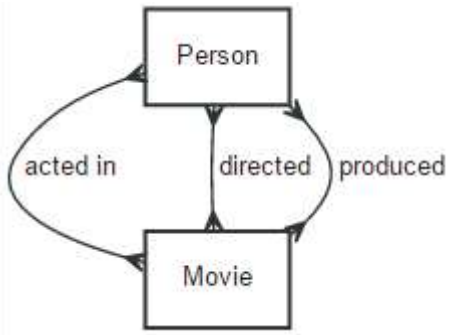
- Sometimes you have collected information into a list, but want to use each element individually as a row. For instance, you might want to further match patterns in the graph. Or you passed in a collection of values but now want to create or match a node or relationship for each element.
- You can use the UNWIND clause to unroll a list into a sequence of rows again.
- For instance, a query to find the top 3 co-actors and then follow their movies and again list the cast for each of those movies:

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)<-[:ACTED_IN]-
(colleague:Person)
WHERE actor.name < colleague.name
WITH actor, colleague, count(*) AS frequency, collect(movie) AS movies
ORDER BY frequency DESC LIMIT 3 UNWIND movies AS m
MATCH (m)<-[:ACTED_IN]-(a)
RETURN m.title AS movie, collect(a.name) AS cast
```

movie	cast
Back to the Future	[Christopher Lloyd, Michael J. Fox]

Comparison of RDBMS and Graph Database

- Relational Model for a person who participates in movies looks like this.
- We have Person and Movie entities, which are related in three different ways, each of which have many-to-many cardinality.
- In a RDBMS we would use tables for the entities as well as for the associative entities (join tables) needed. In this case we decided to go with the following tables: movie, person, acted_in, directed, produced. You'll find the SQL for this below.
- In Neo4j, the basic data units are nodes and relationships. Both can have properties, which correspond to attributes in a RDBMS.
- Nodes can be grouped by putting labels on them. In the example, we will use the labels Movie and Person.
- When using Neo4j, related entities can be represented directly by using relationships. There's no need to deal with foreign keys to handle the relationships, the database will take care of such mechanics. Also, the relationships always have full referential integrity. There are no constraints to enable for this, as it's not optional; it's really part of the underlying data model. Relationships always have a type, and we will differentiate the different kinds of relationships by using the types ACTED_IN, DIRECTED, PRODUCED.



Relational Tables

```
CREATE TABLE movie (
  id INTEGER,
  title VARCHAR(100),
  released INTEGER,
  tagline VARCHAR(100)
);
CREATE TABLE person (
  id INTEGER,
  name VARCHAR(100),
  born INTEGER
);
CREATE TABLE acted_in (
  role varchar(100),
  person_id INTEGER,
  movie_id INTEGER
);
CREATE TABLE directed (
  person_id INTEGER,
  movie_id INTEGER
);
CREATE TABLE produced (
  person_id INTEGER,
  movie_id INTEGER
);
```

```
INSERT INTO movie (id, title, released, tagline)
VALUES (
  (1, 'The Matrix', 1999, 'Welcome to the Real World'),
  (2, 'The Devil''s Advocate', 1997, 'Evil has its winning ways'),
  (3, 'Monster', 2003, 'The first female serial killer of America')
);
INSERT INTO person (id, name, born)
VALUES (
  (1, 'Keanu Reeves', 1964),   (2, 'Carrie-Anne Moss', 1967),
  (3, 'Laurence Fishburne', 1961),   (4, 'Hugo Weaving', 1960),
  (5, 'Andy Wachowski', 1967),   (6, 'Lana Wachowski', 1965),
  (7, 'Joel Silver', 1952),    (8, 'Charlize Theron', 1975),
  (9, 'Al Pacino', 1940),   (10, 'Taylor Hackford', 1944) );
INSERT INTO acted_in (role, person_id, movie_id)
VALUES (
  ('Neo', 1, 1),   ('Trinity', 2, 1),   ('Morpheus', 3, 1),
  ('Agent Smith', 4, 1),   ('Kevin Lomax', 1, 2),
  ('Mary Ann Lomax', 8, 2),    ('John Milton', 9, 2),
  ('Aileen', 8, 3) );
INSERT INTO directed (person_id, movie_id)
VALUES (
  (5, 1),  (6, 1),  (10, 2));
INSERT INTO produced (person_id, movie_id)
VALUES (   (7, 1),   (8, 3) );
```

Neo4J

- In Neo4j we won't create any schema up front. Labels can be used right away without declaring them. In other words, there is no predefine schema.
- In the CREATE statements below, we tell Neo4j what data we want to have in the graph. Simply put, the parentheses denote nodes, while the arrows (-->, or in our case with a relationship type included -[:DIRECTED]->) denote relationships.
- For the nodes we set identifiers like TheMatrix so we can easily refer to them later on in the statement. The identifiers are scoped to the statement, and not visible to other Cypher statements.

Create Statement in Neo4J

```
CREATE (TheMatrix:Movie { title:'The Matrix', released:1999, tagline:'Welcome to the
Real World' })
CREATE (Keanu:Person { name:'Keanu Reeves', born:1964 })
CREATE (Carrie:Person { name:'Carrie-Anne Moss', born:1967 })
CREATE (Laurence:Person { name:'Laurence Fishburne', born:1961 })
CREATE (Hugo:Person { name:'Hugo Weaving', born:1960 })
CREATE (AndyW:Person { name:'Andy Wachowski', born:1967 })
CREATE (LanaW:Person { name:'Lana Wachowski', born:1965 })
CREATE (JoelS:Person { name:'Joel Silver', born:1952 })
CREATE (Keanu)-[:ACTED_IN { roles: ['Neo']}]->(TheMatrix),
  (Carrie)-[:ACTED_IN { roles: ['Trinity']}]->(TheMatrix),
  (Laurence)-[:ACTED_IN { roles: ['Morpheus']}]->(TheMatrix),
  (Hugo)-[:ACTED_IN { roles: ['Agent Smith']}]->(TheMatrix),(AndyW)-[:DIRECTED]-
>(TheMatrix),
  (LanaW)-[:DIRECTED]->(TheMatrix),(JoelS)-[:PRODUCED]->(TheMatrix)
CREATE (TheDevilsAdvocate:Movie { title:"The Devil's Advocate", released:1997,
  tagline: 'Evil has its winning ways' })
CREATE (Monster:Movie { title: 'Monster', released: 2003,
  tagline: 'The first female serial killer of America' })
CREATE (Charlize:Person { name:'Charlize Theron', born:1975 })
CREATE (Al:Person { name:'Al Pacino', born:1940 })
CREATE (Taylor:Person { name:'Taylor Hackford', born:1944 })
CREATE (Keanu)-[:ACTED_IN { roles: ['Kevin Lomax']}]->(TheDevilsAdvocate),
  (Charlize)-[:ACTED_IN { roles: ['Mary Ann Lomax']}]->(TheDevilsAdvocate),
  (Al)-[:ACTED_IN { roles: ['John Milton']}]->(TheDevilsAdvocate),
  (Taylor)-[:DIRECTED]->(TheDevilsAdvocate),(Charlize)-[:ACTED_IN { roles:
['Aileen']}]->(Monster),
  (Charlize)-[:PRODUCED { roles: ['Aileen']}]->(Monster)
```

Read the Data

- In RDBMS:

```
SELECT movie.title
FROM movie;

SELECT movie.title
FROM movie
WHERE movie.released > 1998;

// names of actors and their movies
SELECT person.name, movie.title
FROM person
JOIN acted_in AS acted_in ON
acted_in.person_id = person.id
JOIN movie ON acted_in.movie_id =
movie.id;

// actors in a movie with K. Reeves
SELECT DISTINCT co_actor.name
FROM person AS keanu
  JOIN acted_in AS acted_in1 ON
acted_in1.person_id = keanu.id
  JOIN acted_in AS acted_in2 ON
acted_in2.movie_id = acted_in1.movie_id
  JOIN person AS co_actor
    ON acted_in2.person_id =
co_actor.id AND co_actor.id <> keanu.id
WHERE keanu.name = 'Keanu Reeves';
```

- In Neo4J

```
MATCH (movie:Movie)
RETURN movie.title;

MATCH (movie:Movie)
WHERE movie.released > 1998
RETURN movie.title;

// names of actors and their movies
MATCH (person:Person)-[:ACTED_IN]->
movie:Movie)
RETURN person.name, movie.title;

// actors in a movie with K. Reeves
MATCH (keanu:Person)-[:ACTED_IN]->
(movie:Movie),(coActor:Person)-[:ACTED_IN]->
(movie)
WHERE keanu.name = 'Keanu Reeves'
```

Read the Data

- In RDBMS:

```
//who both acted and produced
movies.
SELECT person.name
FROM person
WHERE person.id IN (SELECT person_id
FROM acted_in)
  AND person.id IN (SELECT person_id
FROM produced);

//directors of K. Reeves movies
SELECT director.name, count(*)
FROM person keanu
  JOIN acted_in ON keanu.id =
acted_in.person_id
  JOIN directed ON acted_in.movie_id
= directed.movie_id
  JOIN person AS director ON
directed.person_id = director.id
WHERE keanu.name = 'Keanu Reeves'
GROUP BY director.name
ORDER BY count(*) DESC
```

- In Neo4J

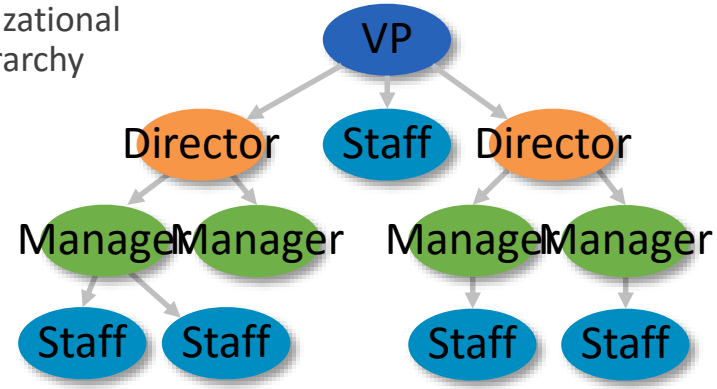
```
//who both acted and produced movies.
MATCH (person:Person)
WHERE (person)-[:ACTED_IN]->() AND (person)-
[:PRODUCED]->()
RETURN person.name;

//directors of K. Reeves movies
MATCH (keanu:Person { name: 'Keanu Reeves'
})-[:ACTED_IN]->(movie:Movie),
  (director:Person)-[:DIRECTED]->(movie)
RETURN director.name, count(*)
ORDER BY count(*) DESC
```

Use Case
Graphs for Master Data Management

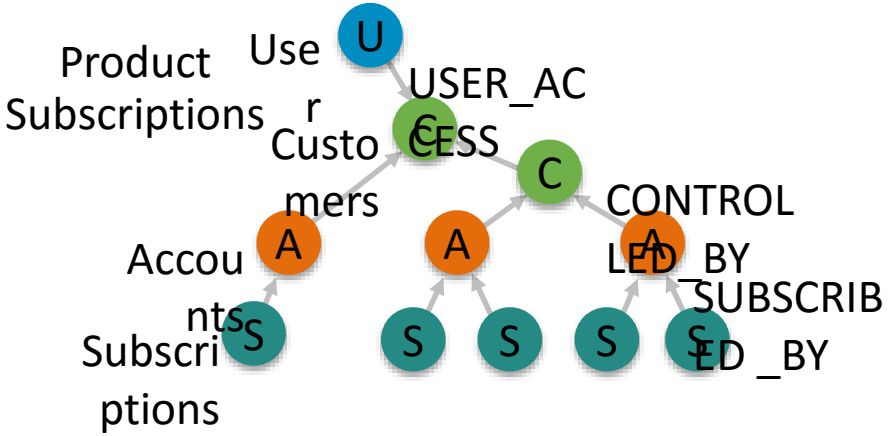
MDM Solutions with Graph Databases

Organizational Hierarchy

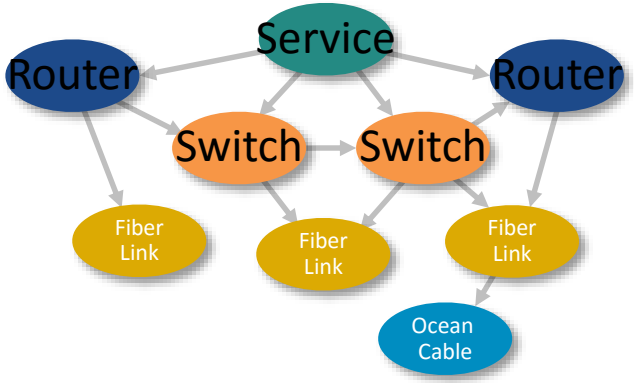


Social Networks

Product Subscriptions

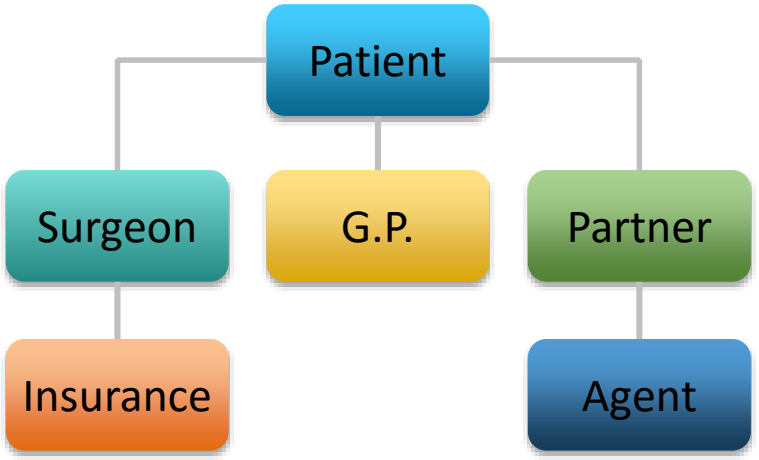


CMDB Network Inventory

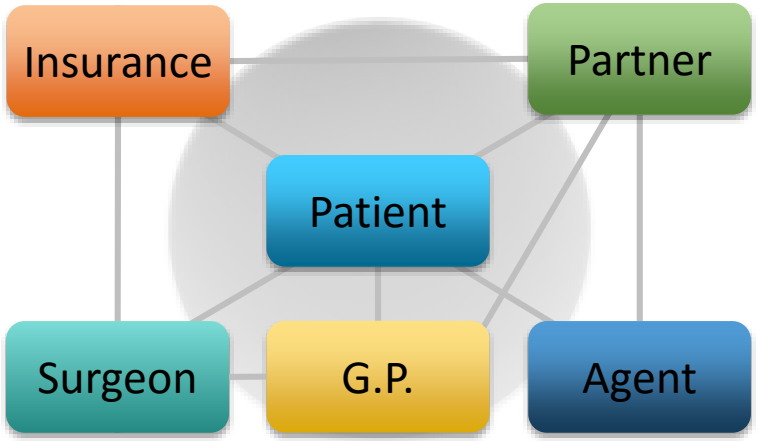


MDM Isn't Hierarchical

Typical MDM system structure



…but MDM is really a network



Gartner

Challenges with Current MDM Systems

Lack of support for non-hierarchical or matrix data relationships

- Master data is never strictly hierarchical
- Systems are designed for fixed top-down hierarchy
- Non-hierarchical data is not supported

Inability to unlock value from data relationships

- Systems store only very simple data relationships
- Complex relationships and links not stored

Inflexible and expensive to maintain

- Changes to the model are expensive and time-consuming

Use Case
Graphs for Network and IT Operations Management

Network Graphs – Telco Example

PROBLEM

Need: Instantly diagnose problems in networks of 1B+ elements

But: Basing diagnosis solely on streaming machine data severely limits accuracy and effectiveness

SOLUTION

Real-time graph analytics provide actionable insight for the largest complex connected networks in the world

- The entire network lives in a graph
- Analyzes dependencies in real time
- Highly scalable with carrier-grade uptime requirements

VIRTUAL INSTRUMENTS

DRAKER

hp

SERENA

StarHub

User Case
Graphs for Fraud Detection

Fraud Scenarios

Retail First Party Fraud

- Opening many lines of credit with no intention of paying back
- Accounts for $10B+ in annual losses at US banks[1]

Synthetic Identities and Fraud Rings

- Rings of synthetic identities committing fraud

Insurance – Whiplash for Cash

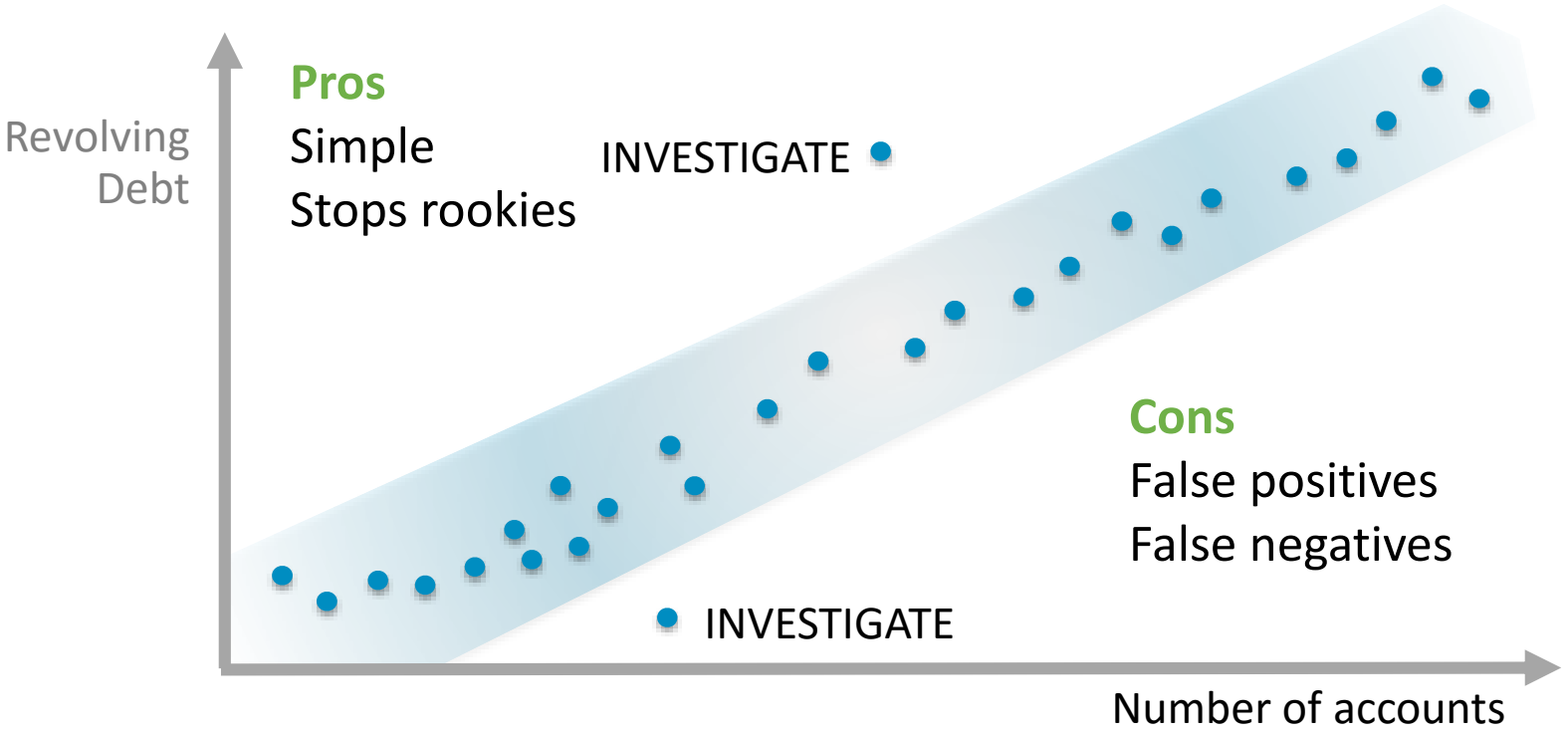
- Insurance scams using fake drivers, passengers and witnesses
- Increase network efficiency

eCommerce Fraud

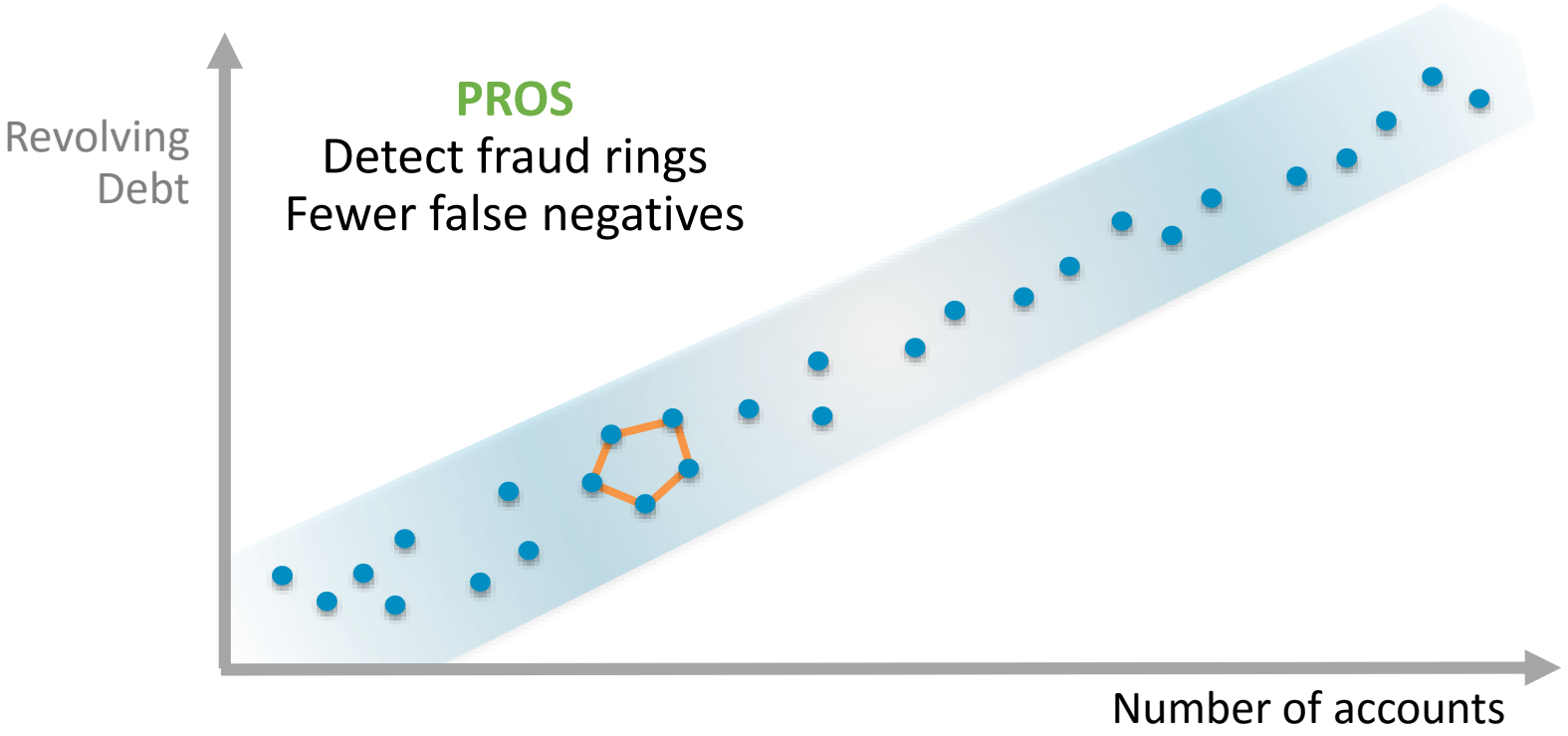
- Online payment fraud

(1) Business Insider: http://www.businessinsider.com/how-to-use-social-networks-in-the-fight-against-first-party-fr
2011-3

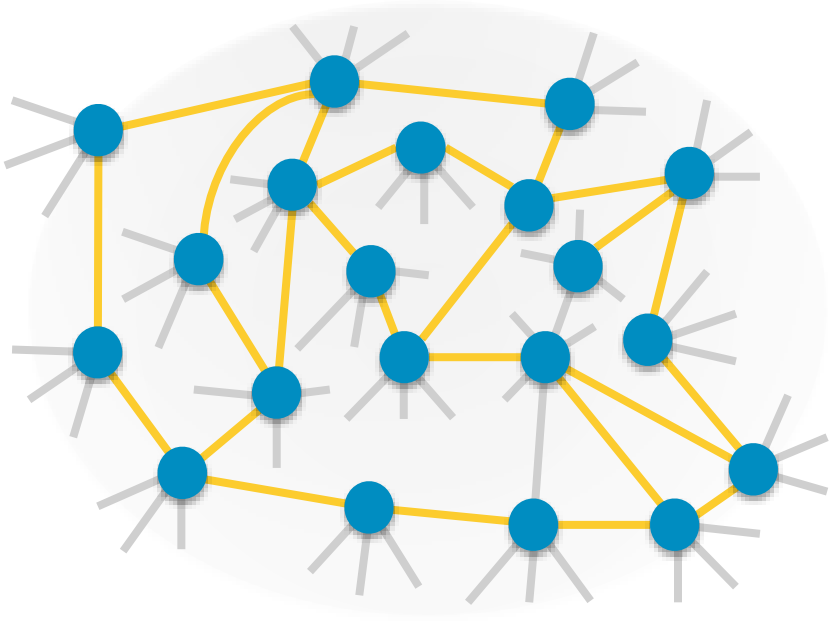
Discrete Data Analysis



Connected Analysis



Connected Analysis with Neo4j



Value

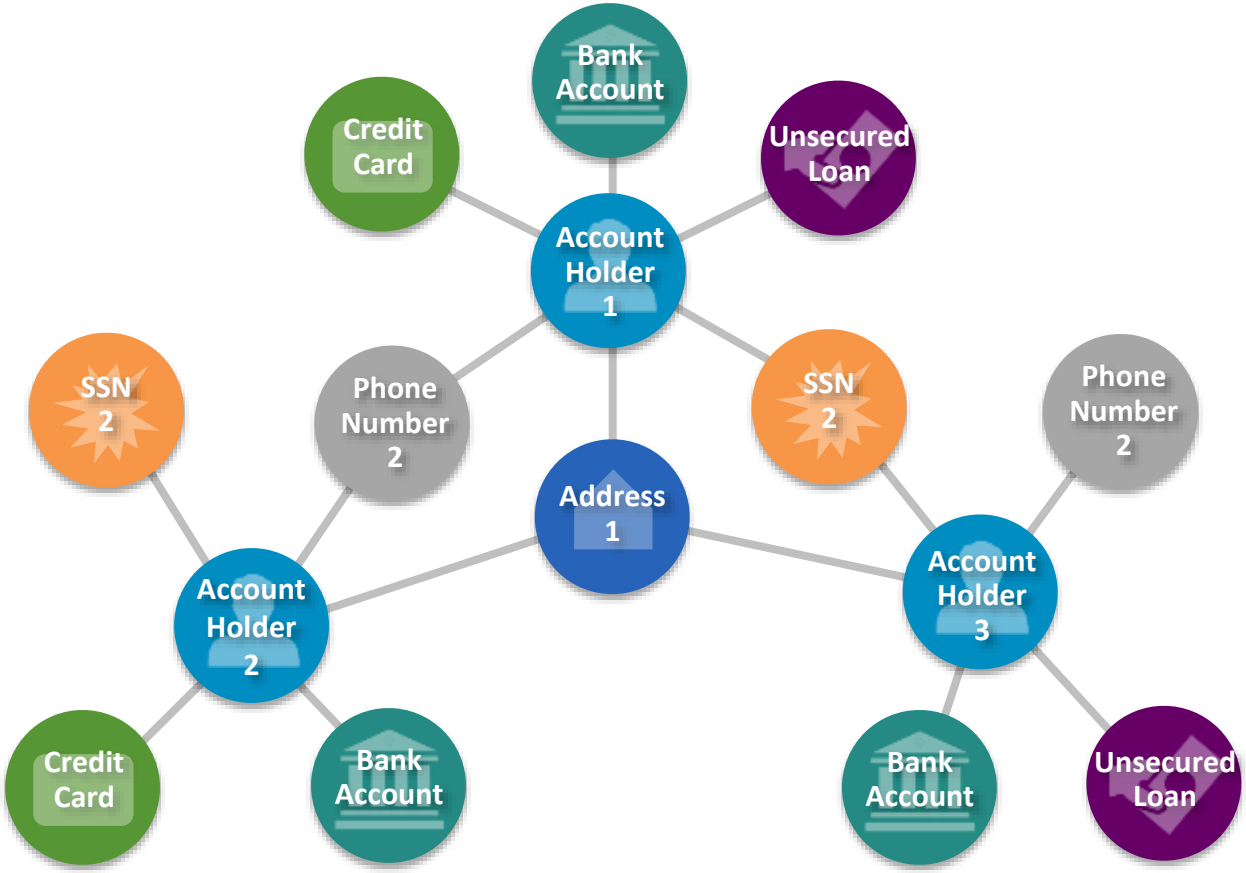
Effective in detecting some of the most impactful attacks, even from organized rings

Challenge

Extremely difficult with traditional technologies

*For example a **ten-person fraud bust-out is $1.5M**, assuming 100 false identities and 3 financial instruments per identity, each with a $5K credit limit*

Modeling a Fraud Ring as a Graph



Use Case
Graphs for Real-time Recommendations

Real-Time Recommendations - Benefits

Online Retail

- Suggest related products and services
- Increase revenue and engagement

Media and Broadcasting

- Create an engaging experience
- Produce personalized content and offers

Logistics

- Recommend optimal routes
- Increase network efficiency

Real-Time Recommendations - Challenges

Make effective real-time recommendations

- Timing is everything in point-of-touch applications
- Base recommendations on current data, not last night's batch load

Process large amounts of data and relationships for context

- Relevance is king: Make the right connections
- Drive traffic: Get users to do more with your application

Accommodate new data and relationships continuously

- Systems get richer with new data and relationships
- Recommendations become more relevant

Walmart – Retail Recommendations

- Needed online customer recommendations to keep pace with competition
- Data connections provided predictive context, but were not in a usable format
- Solution had to serve many millions of customers and products while maintaining superior scalability and performance

World's largest company by revenue

World's largest retailer and private employer

SF-based global e-commerce division manages several websites

Found in 1969
Bentonville, Arkansas