What is a Graph Database?

- A graph database is an online database management system with Create, Read, Update and Delete (CRUD) operations working on a graph data model.
- Unlike other databases, relationships take priority in graph databases. This means your application doesn't have to infer data connections using things like foreign keys or out-of-band processing, such as MapReduce.
- The data model for a graph database is also significantly simpler and more expressive than those of relational or other NoSQL databases.
- Graph databases are built for use with transactional (OLTP) systems and are engineered with transactional integrity and operational availability in mind.

Principles of graph databases:

Object-Oriented Thinking

This means very clear, explicit semantics for each query you write. There are no hidden assumptions, such as relational SQL where. You must know how the tables in FROM Clause will implicitly from cartesian products.

Performance

They have superior performance for querying related data, big or small. A graph is essentially an index data structure. It never needs to load or touch unrelated data for a given query. They're an excellent solution for real-time big data analytical queries.

Better Problem-Solving

Graph databases solve problems that are both impractical and practical for relational queries. Examples include iterative algorithms such as PageRank, gradient descent, and other data mining and machine learning algorithms. Research has proved that some graph query languages are Turing complete, meaning that you can write any algorithm on them. There are many query languages in the market that have limited expressive power, though. Make sure you ask many hypothetical questions to see if it can answer them before you lock in.

Update Data in Real-Time and Support Queries Simultaneously

Graph databases can perform real-time updates on big data while supporting queries at the same time. This is a major drawback of existing big data management systems such as Hadoop HDFS since it was designed for data lakes, where sequential scans and appending new data (no random seek) are the characteristics of the intended workload, and it is an architecture design choice to ensure fast scan I/O of an entire file. The assumption there was that any query will touch most of a file, while graph databases only touch relevant data, so a sequential scan is not an optimization assumption.

Flexible Online Schema Environment

Graph databases offer a flexible online schema evolvement while serving your query. You can constantly add and drop new vertex or edge types or their attributes to extend or shrink your data model. It's so convenient to manage explosive and constantly changing object types. The relational database just cannot easily adapt to this requirement, which is commonplace in the modern data management era.

Group by Aggregate Queries

Graph databases, in addition to traditional group-by queries, can do certain classes of group by aggregate queries that are unimaginable or impractical in relational databases. Due to the tabular model restriction, aggregate queries on a relational database are greatly constrained by how data is grouped together. In contrast, graph models are more flexible for grouping and aggregating relevant data.

Combine and Hierarchize Multiple Dimensions

Graph databases can combine multiple dimensions to manage big data, including time series, demographic, geo-dimensions, etc. with a hierarchy of granularity on different dimensions. Think about an application in which we want to segment a group of a population based on both time and geo dimensions. With a carefully designed graph schema, data scientists and business analysts can conduct virtually any analytical query on a graph database. This capability traditionally is only accessible to low-level programming languages such as C++ and Java.

AI Infrastructure

Graph databases serve as great AI infrastructure due to well-structured relational information between entities, which allows one to further infer indirect facts and knowledge. Machine learning experts love them. They provide rich information and convenient data accessibility that other data models can hardly satisfy. For example, the Google Expander team has used it for smart messaging technology. The knowledge graph was created by Google to understand humans better, and many more advances are being made on knowledge inference. The keys of a successful graph database to serve as a real-time AI data infrastructure are:

- Support for real-time updates as fresh data streams in,
- A highly expressive and user-friendly declarative query language to give full control to data scientists,
- Support for deep-link traversal (>3 hops) in real-time (sub-second), just like human neurons sending information over a neural network; deep and efficient
- Scale out and scale up to manage big graphs

Research Paper:

Use of Graph Database for the Integration of Heterogeneous Biological Data:

The paper is based in the complex relationships among heterogenous biological data which contributes functions of a living cell. But these findings are very difficult due to complex relationships among them. Traditional relational database systems, such as MySQL and Oracle, is limited as relational databases stores multiple tables.

A graph database uses a graph uses a graph structure. This database uses nodes and edges to represent and store data. Each node represents an entity (biological entity) and edge represents connection or relationship between two nodes. Graph database is more expressive and simpler. It is very useful for situations with heavily interconnected data.

In biological community, several researchers have adopted graph database for biological networks to establish various hypotheses. For example, Lysenko et al., Henkel et al., Mullen et al., Balaur et al. are some of the examples. In the work, authors set up a graph database and tested performance in storing and retrieving heterogenous and complex biological networks. They used Neo4j and compared the performance with MySQL in querying.

Selection of graph database engine

Neo4j is an open-source graph database has several advantages. There are 5 advantages for (i) use of graph model for intuitive information searches.; (2) full ACID (Access, Create, Insert, Delete); (3) billions of nodes, relationships and properties; (4) use of Java; (5) easy to use API.

Memory Configuration:

Large memory is needed and should also be set for Neo4j for usage of the system memory. Memory configuration includes 3 steps: (1) OS memory sizing; (2) page cache memory sizing; and (3) heap memory sizing.

OS memory sizing:

OS memory = 1 GB + (Size of graph.db/index) + (Size of graph.db/schema).

Thus, we allocated 768GB to page cache memory and heap memory.

According to Neo4j document:

Actual OS allocation = Available RAM − (Page cache + Heap size).

Allocated 100GB for system memory and rest to page cache and heap size.

Page cache sizing: When the size of the entire data is larger a swap occurs, which results into high disk access cost and reduced performance.

Heap sizing: Based on Java, Neo4j uses more memory as heap memory in a Java Virtual Machine (JVM) is increased. More heap memory increases the performance greatly, so they allocated 300Gb to heap sizing. They set dbms.memory.heap.initaial_size from 8gb to 300gb.

Disk access configuration: Logical transaction logs can occur in system and data recovery after an unexpected system shutdown. They are used for backup of at online status. The transaction log renewed when they exceed 25MB. The open file limit for most LINUX servers is 1024. Thus, they changed open file limit for accessing many files to 400,000.

Storage Engine: MySQL has variety of storage engines, where they used InnoDB and MyISAM. In InnoDb, table locking occurs when more than 5 million data are processed in indexed sites which deteriorates retrieval performance. InnoDB is slower than MyISAM, as it guarantees data integrity and loads indexes for retrieval.

Parameters of MySQL server: Disk searching is a huge performance bottleneck. It is more when gigantic data is used. To overcome, disks are used with low seek times. Innodb_buffer_pool_size parameter is being changed from default to 800GB. For maximum size of internal in-memory temporary tables, tmp_table_size is set from default to 64GB. For log I/O, innodb_log_file_size is set to 120gb from default.

Optimization of memory use: Over here, they changed key_buffer_size to 250GB, table_open_cache to maximum, join_buffer_size and sort_buffer_size to 4GB. The read_buffer_size, max_heap_table_size, and thread cache parameters were set from default to the maximum allowed value.

Collection of diverse information for graph DB: In this section, they classified each data into nodes and then into relationships and then removed redundant and ambiguous nodes and relationships. Then they integrated and expanded all the nodes and relationships. Later, they created data models for each node and relationships. Figure 2 illustrates the whole process.

Result:

Comparison of Neo4j after optimization: The authors compared two servers: optimized versus non-optimized servers. The non-optimized servers had settings for OS memory and environment. Where the optimized server had several modified settings in (1) page cache sizing; (2) OS memory; (3) heap memory; (4) number of open file limits.

They performed same query on each server. Observation was that the measured time to return results by performing a query to retrieve all data that transverse relationships. The optimized server returned results in 138ms whereas non optimized took 316ms to get results.

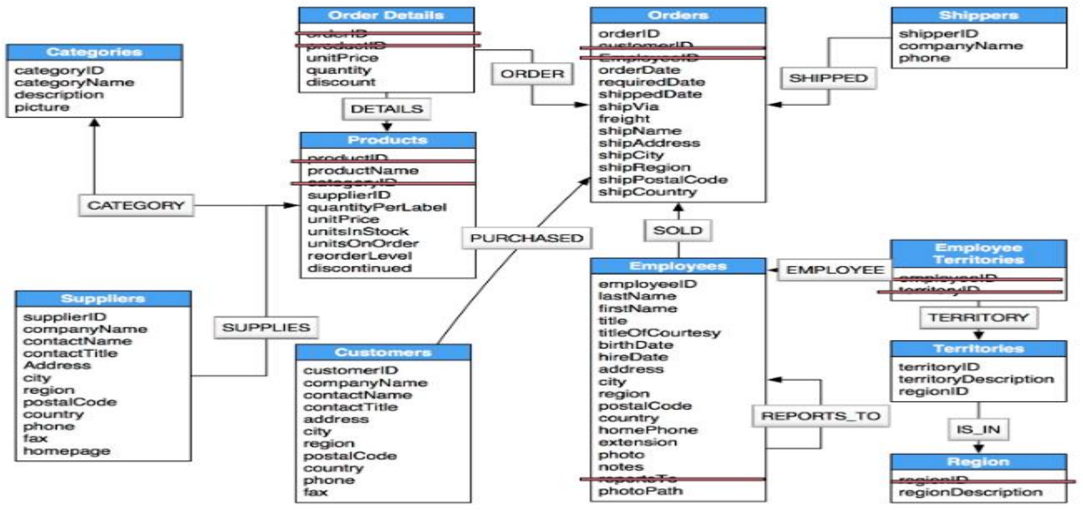
Comparison of search speed between two databases:

MySQL was tested with Neo4j with the same large data set. Neo4j outperformed MySQL in all tested cases. After running the same query in both MySQL and Neo4j, which was retrieving all data belonging to a particular gene-related path in a gene, disease and drug relationships through a 3-layer search. When the two databases were queried with the same search term, Neo4j took 2.128 sec while MySQL took 58.325 sec for a 3-layer search. For 4-layer, Neo4j took 20.128 sec while MySQL failed to return the results.

We also have annotated the whole paper which can be available in this link

2. Neo4j
small illustration of running Neo4j, use a small graph of your choice
We take small dataset which is currently present in prosql and then we transfer the whole dataset in neo4j which described here,

Nordwind Dataset



we need to extract the data from PostgreSQL so we can create it as a graph. The easiest way to do that is to export the appropriate tables in CSV format.

```
COPY (SELECT * FROM customers) TO '/tmp/customers.csv' WITH CSV header;
COPY (SELECT * FROM suppliers) TO '/tmp/suppliers.csv' WITH CSV header;
COPY (SELECT * FROM products)  TO '/tmp/products.csv' WITH CSV header;
COPY (SELECT * FROM employees) TO '/tmp/employees.csv' WITH CSV header;
COPY (SELECT * FROM categories) TO '/tmp/categories.csv' WITH CSV header;

COPY (SELECT * FROM orders
    LEFT OUTER JOIN order_details ON order_details.OrderID = orders.OrderID) TO '/tmp/orders.csv' WITH
CSV header;
```

After we've exported our data from PostgreSQL, we'll use Cypher's LOAD CSV command to transform the contents of the CSV file into a graph structure

```
// Create customers
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:customers.csv" AS row
CREATE (:Customer {companyName: row.CompanyName, customerID: row.CustomerID, fax: row.Fax, phone:
row.Phone});

// Create products
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:products.csv" AS row
```

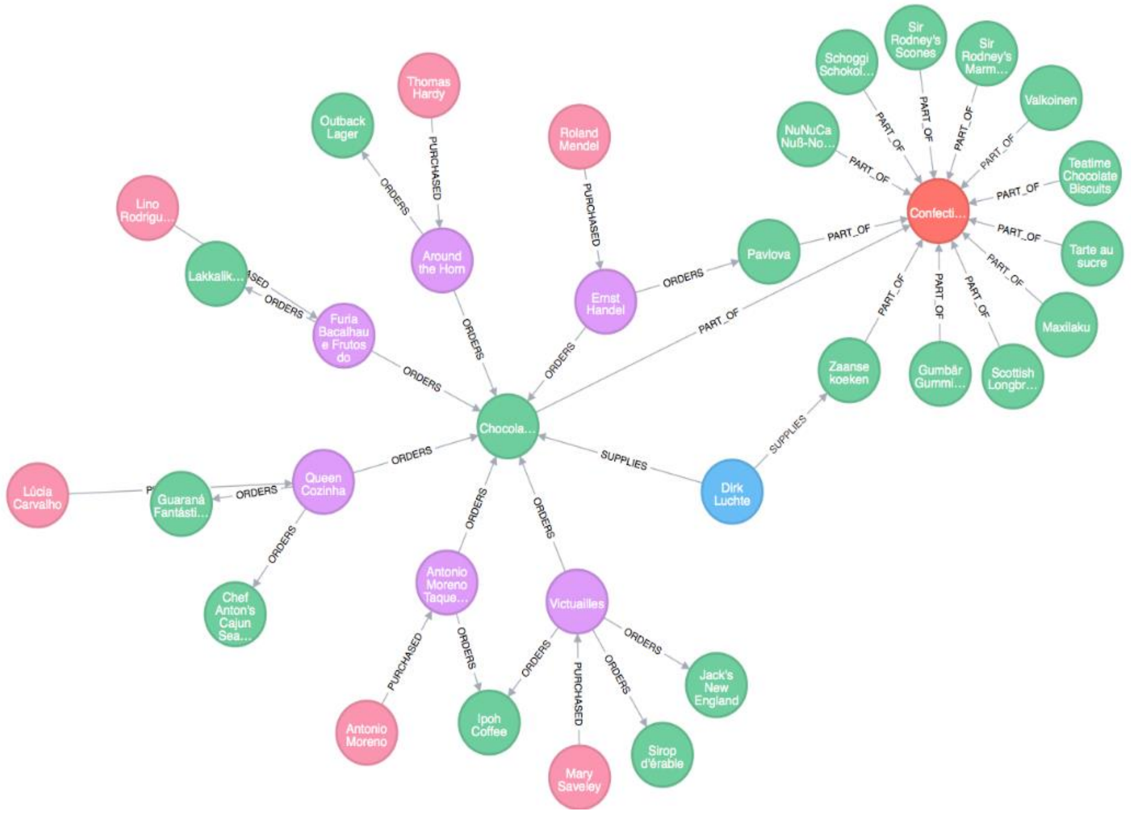
```
CREATE (:Product {productName: row.ProductName, productID: row.ProductID, unitPrice:
toFloat(row.UnitPrice)});

// Create suppliers
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:suppliers.csv" AS row
CREATE (:Supplier {companyName: row.CompanyName, supplierID: row.SupplierID});

// Create employees
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:employees.csv" AS row
CREATE (:Employee {employeeID:row.EmployeeID,  firstName: row.FirstName, lastName: row.LastName, title:
row.Title});

// Create categories
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:categories.csv" AS row
CREATE (:Category {categoryID: row.CategoryID, categoryName: row.CategoryName, description:
row.Description});

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:orders.csv" AS row
CREATE INDEX ON :Product(productID);
CREATE INDEX ON :Product(productName);
CREATE INDEX ON :Category(categoryID);
CREATE INDEX ON :Employee(employeeID);
CREATE INDEX ON :Supplier(supplierID);
CREATE INDEX ON :Customer(customerID);
CREATE INDEX ON :Customer(customerName);
CREATE CONSTRAINT ON (o:Order) ASSERT o.orderID IS UNIQUE;
```

Next, we’ll create indexes on the just-created nodes to ensure their quick lookup when creating relationships in the next step.

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:orders.csv" AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (product:Product {productID: row.ProductID})
MERGE (order)-[pu:PRODUCT]->(product)
ON CREATE SET pu.unitPrice = toFloat(row.UnitPrice), pu.quantity = toFloat(row.Quantity);

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:orders.csv" AS row
```

```
MATCH (order:Order {orderID: row.OrderID})
MATCH (employee:Employee {employeeID: row.EmployeeID})
MERGE (employee)-[:SOLD]->(order);

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:orders.csv" AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (customer:Customer {customerID: row.CustomerID})
MERGE (customer)-[:PURCHASED]->(order);
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:products.csv" AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (supplier:Supplier {supplierID: row.SupplierID})
MERGE (supplier)-[:SUPPLIES]->(product);
```

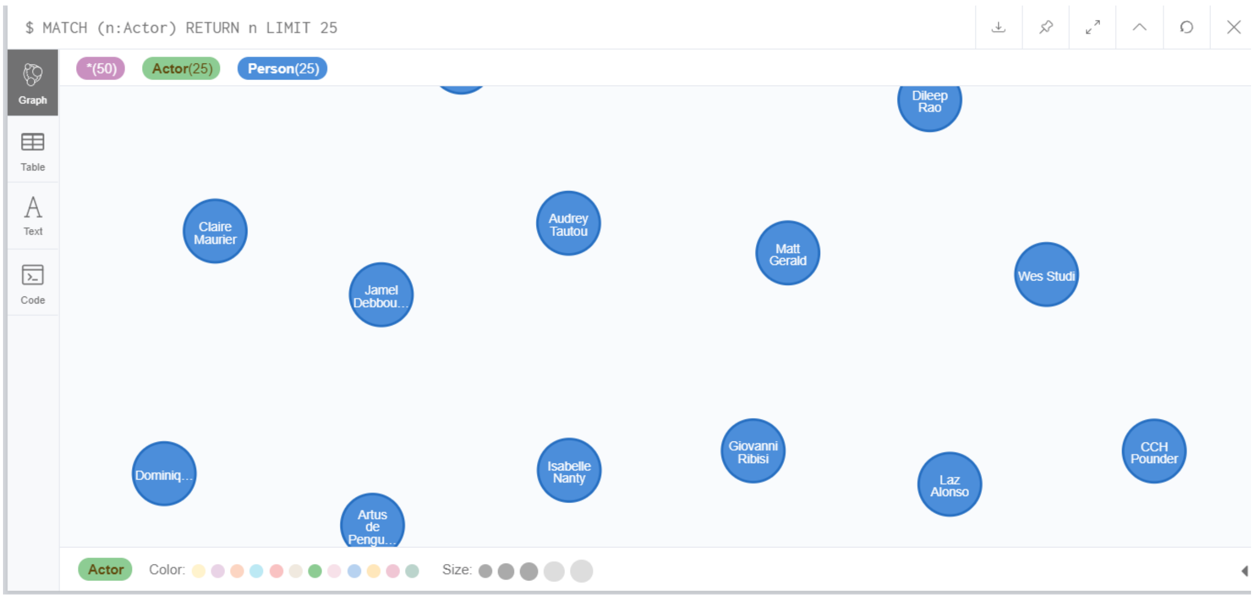
The resulting graph should look like this:



3.Movie dataset

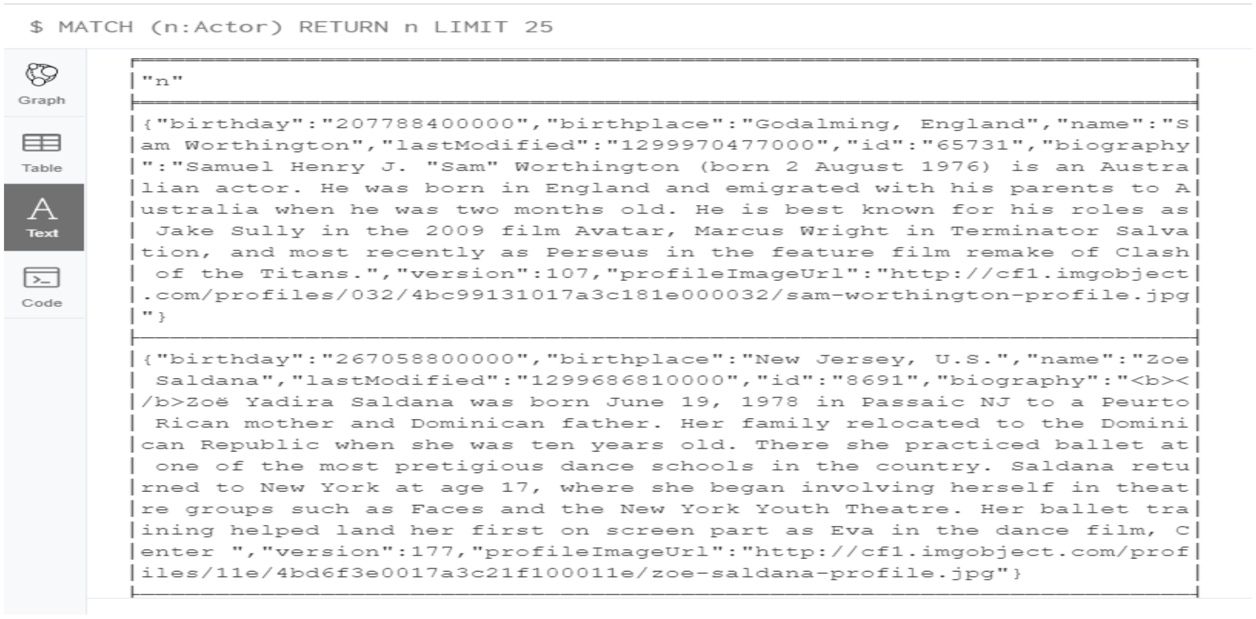
We import movie data set from csv file and create relationship between them which has more than 12K movies and 50K actors and current user which can view or rated movie info.

Actor:



```
$ MATCH (n:Actor) RETURN n LIMIT 25
```

Here Every actor node contains information about them.

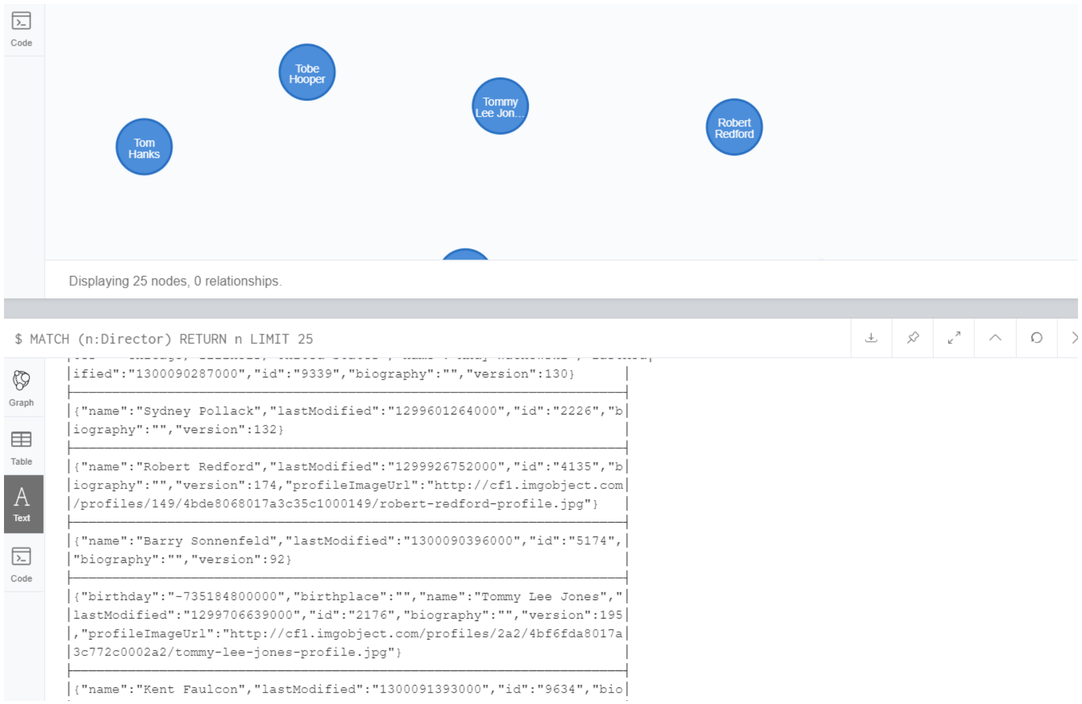


```
$ MATCH (n:Actor) RETURN n LIMIT 25
```

```
|"n"                                                                        |
|{"birthday":"207788400000","birthplace":"Godalming, England","name":"S|
|am Worthington","lastModified":"1299970477000","id":"65731","biography|
|":"Samuel Henry J. "Sam" Worthington (born 2 August 1976) is an Austra|
|lian actor. He was born in England and emigrated with his parents to A|
|ustralia when he was two months old. He is best known for his roles as|
| Jake Sully in the 2009 film Avatar, Marcus Wright in Terminator Salva|
|tion, and most recently as Perseus in the feature film remake of Clash|
| of the Titans.","version":107,"profileImageUrl":"http://cf1.imgobject|
|.com/profiles/032/4bc99131017a3c181e000032/sam-worthington-profile.jpg|
|"}                                                                     |
|{"birthday":"267058800000","birthplace":"New Jersey, U.S.","name":"Zoe|
| Saldana","lastModified":"1299686810000","id":"8691","biography":"<b><|
|/b>Zoë Yadira Saldana was born June 19, 1978 in Passaic NJ to a Peurto|
| Rican mother and Dominican father. Her family relocated to the Domini|
|can Republic when she was ten years old. There she practiced ballet at|
| one of the most pretigious dance schools in the country. Saldana retu|
|rned to New York at age 17, where she began involving herself in theat|
|re groups such as Faces and the New York Youth Theatre. Her ballet tra|
|ining helped land her first on screen part as Eva in the dance film, C|
|enter ","version":177,"profileImageUrl":"http://cf1.imgobject.com/prof|
|iles/11e/4bd6f3e0017a3c21f100011e/zoe-saldana-profile.jpg"}           |
```

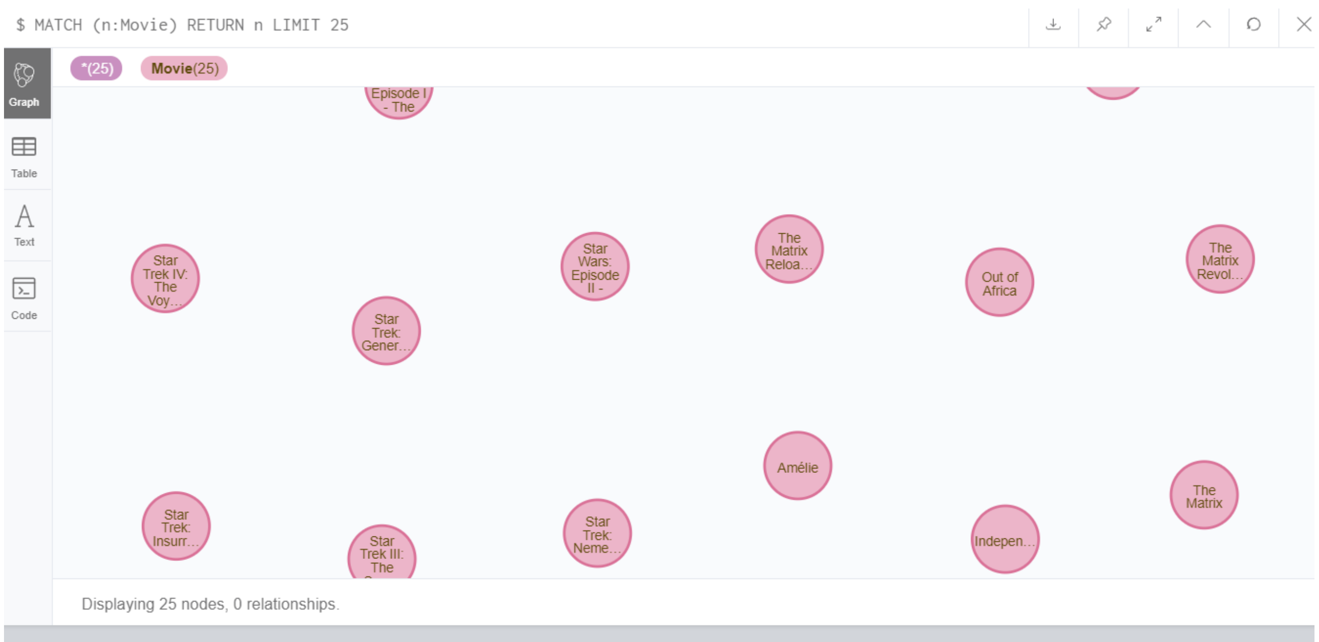
Also, we created director and movies data node which contains data about director’s personal info and movies basic info.

Director’s nodes.

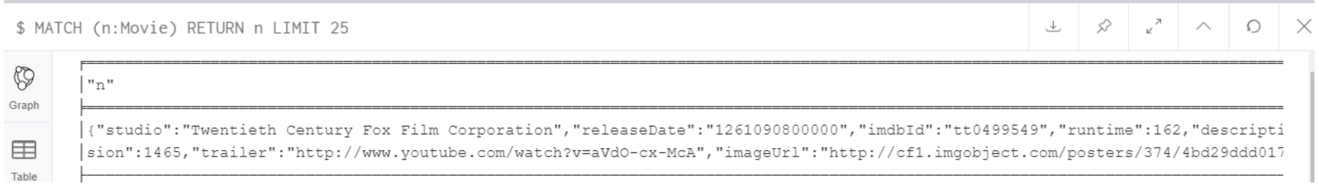


```
Displaying 25 nodes, 0 relationships.

$ MATCH (n:Director) RETURN n LIMIT 25
|ified":"1300090287000","id":"9339","biography":"","version":130}   |
├──────────────────────────────────────────────────────────────────┤
|{"name":"Sydney Pollack","lastModified":"1299601264000","id":"2226","b|
|iography":"","version":132}                                          |
├──────────────────────────────────────────────────────────────────┤
|{"name":"Robert Redford","lastModified":"1299926752000","id":"4135","b|
|iography":"","version":174,"profileImageUrl":"http://cf1.imgobject.com|
|/profiles/149/4bde8068017a3c35c1000149/robert-redford-profile.jpg"}  |
├──────────────────────────────────────────────────────────────────┤
|{"name":"Barry Sonnenfeld","lastModified":"1300090396000","id":"5174",|
|"biography":"","version":92}                                         |
├──────────────────────────────────────────────────────────────────┤
|{"birthday":"-735184800000","birthplace":"","name":"Tommy Lee Jones","|
|lastModified":"1299706639000","id":"2176","biography":"","version":195|
|,"profileImageUrl":"http://cf1.imgobject.com/profiles/2a2/4bf6fda8017a|
|3c772c0002a2/tommy-lee-jones-profile.jpg"}                           |
├──────────────────────────────────────────────────────────────────┤
|{"name":"Kent Faulcon","lastModified":"1300091393000","id":"9634","bio|
```

Here some of the movie's node with their info.

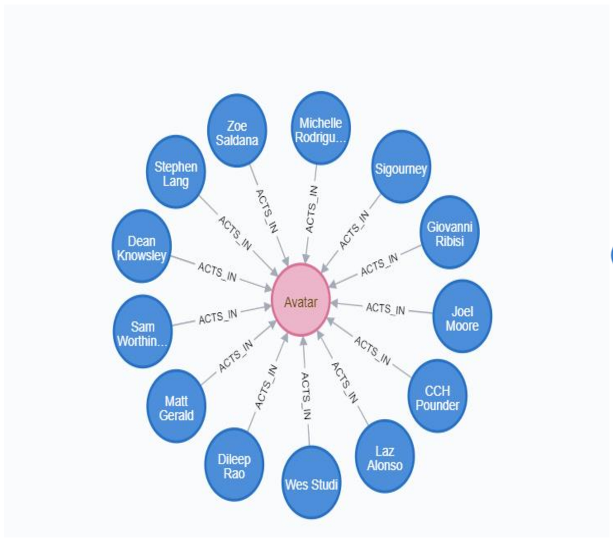


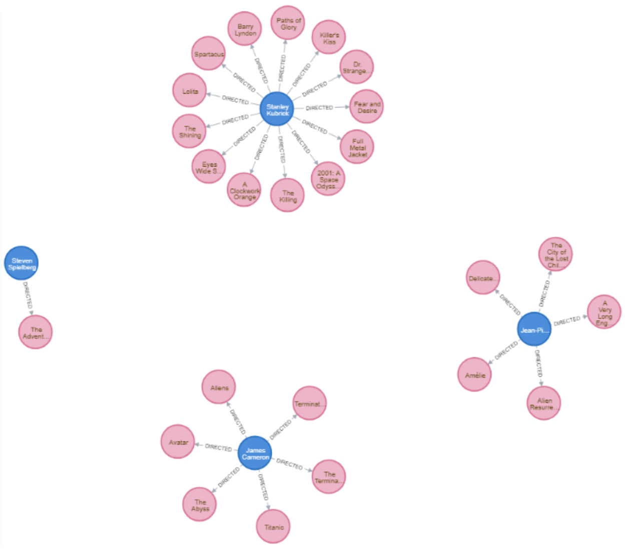
```
$ MATCH (n:Movie) RETURN n LIMIT 25

Displaying 25 nodes, 0 relationships.
```



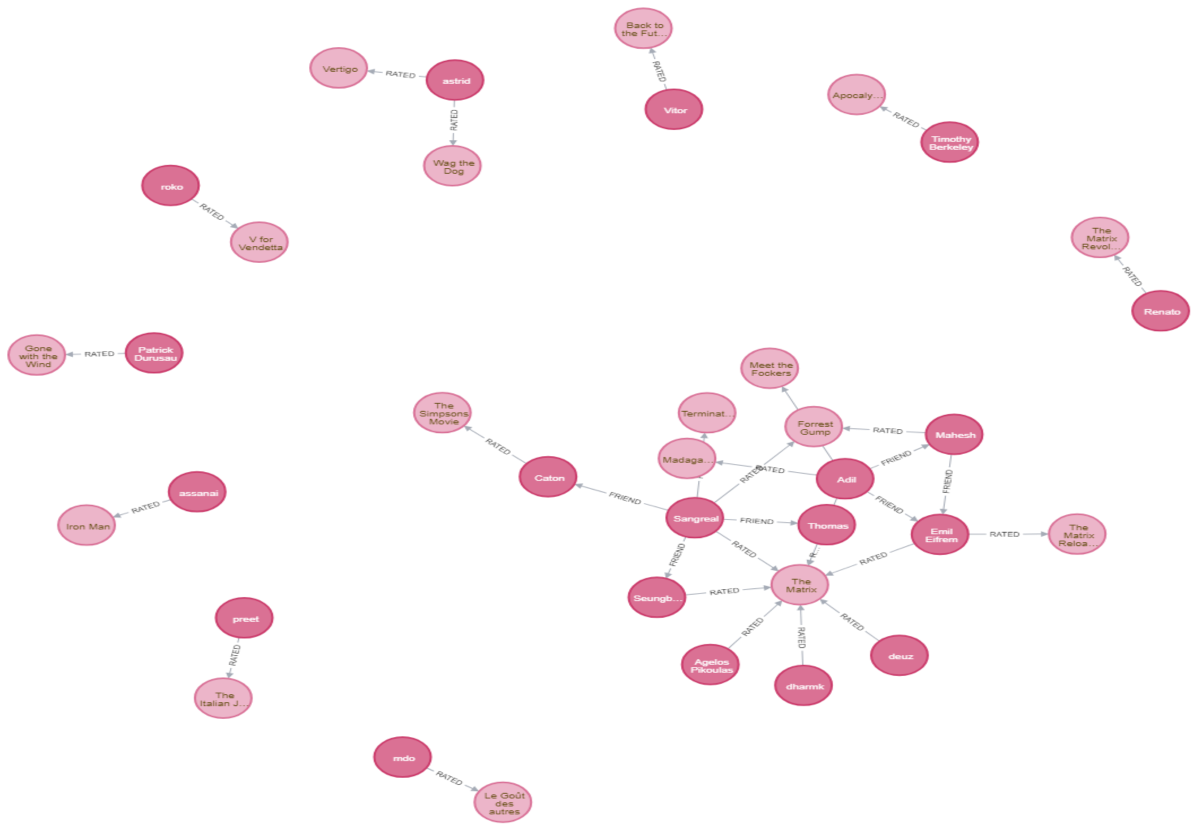
```
$ MATCH (n:Movie) RETURN n LIMIT 25
╒════════════════════════════════════════════════════════════════════════════════════════════
|"n"
╞════════════════════════════════════════════════════════════════════════════════════════════
|{"studio":"Twentieth Century Fox Film Corporation","releaseDate":"1261090800000","imdbId":"tt0499549","runtime":162,"descripti
|sion":1465,"trailer":"http://www.youtube.com/watch?v=aVdO-cx-McA","imageUrl":"http://cf1.imgobject.com/posters/374/4bd29ddd017
├────────────────────────────────────────────────────────────────────────────────────────────
```

Here is some relationship which we defined as act in between actor and movie then directed between director and movie and rated between user and movies.





RATED Relationship with friend

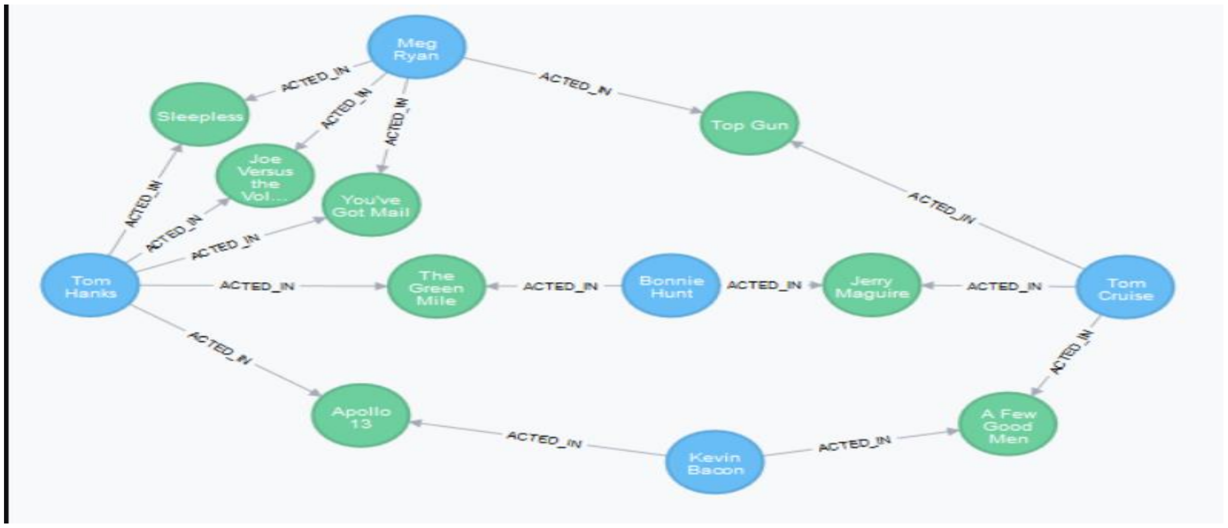


Some of interesting queries that we run,

- **This quire finds all relationship on Tom hanks and co actor**

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
(coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

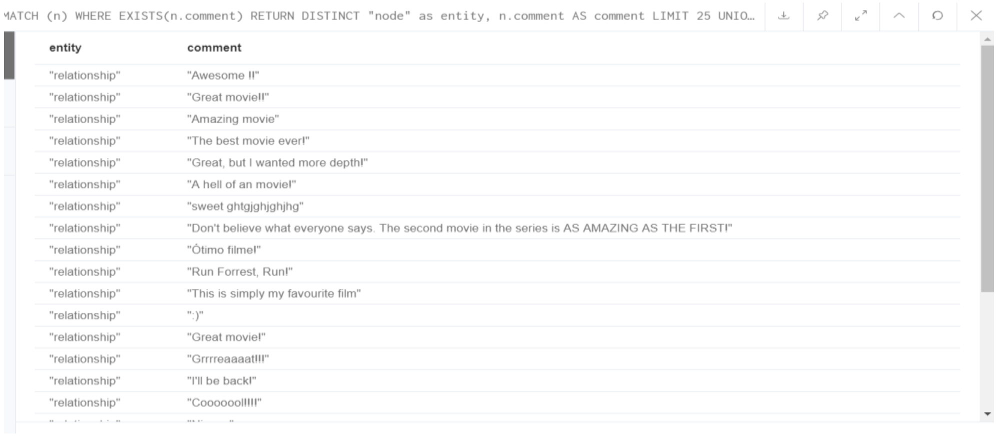
Result:



- **This quire finds all relationship on comment base**

MATCH (n) WHERE EXISTS(n.comment) RETURN DISTINCT "node" as entity, n.comment AS comment LIMIT 25 UNION ALL MATCH ()-[r]-() WHERE EXISTS(r.comment) RETURN DISTINCT "relationship" AS entity, r.comment AS comment LIMIT 25

Result:

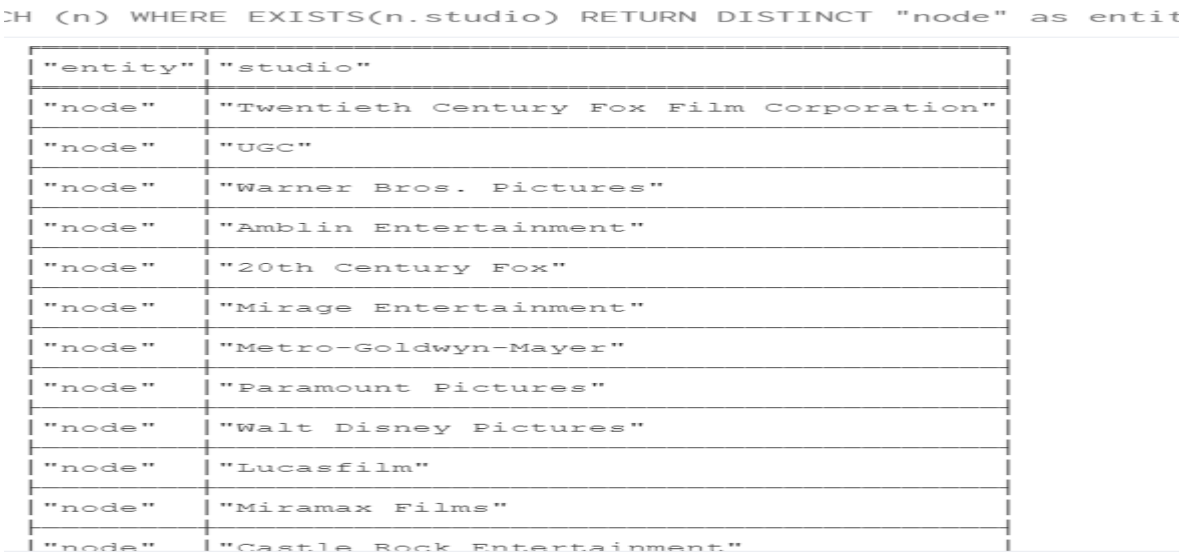


```
MATCH (n) WHERE EXISTS(n.comment) RETURN DISTINCT "node" as entity, n.comment AS comment LIMIT 25 UNIO...
```

entity	comment
"relationship"	"Awesome !!"
"relationship"	"Great movie!!"
"relationship"	"Amazing movie"
"relationship"	"The best movie ever!"
"relationship"	"Great, but I wanted more depth!"
"relationship"	"A hell of an movie!"
"relationship"	"sweet ghtgjghjghjhg"
"relationship"	"Don't believe what everyone says. The second movie in the series is AS AMAZING AS THE FIRST!"
"relationship"	"Ótimo filme!"
"relationship"	"Run Forrest, Run!"
"relationship"	"This is simply my favourite film"
"relationship"	":)"
"relationship"	"Great movie!"
"relationship"	"Grrrreaaaat!!!"
"relationship"	"I'll be back!"
"relationship"	"Coooooool!!!!"

- **This quire finds all nodes of studios:**

MATCH (n) WHERE EXISTS(n.studio) RETURN DISTINCT "node" as entity, n.studio AS studio LIMIT 25 UNION ALL MATCH ()-[r]-() WHERE EXISTS(r.studio) RETURN DISTINCT "relationship" AS entity, r.studio AS studio LIMIT 25



```
CH (n) WHERE EXISTS(n.studio) RETURN DISTINCT "node" as entit

╒════════╤═══════════════════════════════════════╕
│"entity"│"studio"                               │
╞════════╪═══════════════════════════════════════╡
│"node"  │"Twentieth Century Fox Film Corporation"│
├────────┼───────────────────────────────────────┤
│"node"  │"UGC"                                  │
├────────┼───────────────────────────────────────┤
│"node"  │"Warner Bros. Pictures"                │
├────────┼───────────────────────────────────────┤
│"node"  │"Amblin Entertainment"                 │
├────────┼───────────────────────────────────────┤
│"node"  │"20th Century Fox"                     │
├────────┼───────────────────────────────────────┤
│"node"  │"Mirage Entertainment"                 │
├────────┼───────────────────────────────────────┤
│"node"  │"Metro-Goldwyn-Mayer"                  │
├────────┼───────────────────────────────────────┤
│"node"  │"Paramount Pictures"                   │
├────────┼───────────────────────────────────────┤
│"node"  │"Walt Disney Pictures"                 │
├────────┼───────────────────────────────────────┤
│"node"  │"Lucasfilm"                            │
├────────┼───────────────────────────────────────┤
│"node"  │"Miramax Films"                        │
├────────┼───────────────────────────────────────┤
│"node"  │"Castle Rock Entertainment"            │
```

We can run endless quires for different purpose in this database which can be used for any general perspective also graph representation is more suitable for this type of data base because it runs fast and take less memory than other representation.