

An In-Depth Analysis on Neo4j and Cypher

CS 4411 Final Report

Matthew Cheverie

December 8, 2022

1	Contents	
2	Background and Significance	3
2.1	What is Neo4j?	3
2.2	What is Cypher?	3
2.3	What is Neo4j's graph data science platform?	3
2.4	What is Neo4j Bloom?	3
2.5	Why use Neo4j as opposed to other databases?	3
3	Project Goals and Objectives	4
3.1	Project Goal	4
3.2	Project objective	4
4	In-depth Look into Neo4j and Cypher	4
4.1	Neo4j functionality	4
4.1.1	What is a graph database?	4
4.1.2	Comparison of RDBMS to Graph Database	6
4.2	Neo4j Graph Platform	7
4.2.1	Neo4j Database	7
4.2.2	Neo4j Desktop	8
4.2.3	Neo4j Browser	9
4.2.4	Neo4j Bloom	11
4.3	Cypher Query Language	13
4.3.1	Introduction to Cypher	13
4.3.2	Some query examples	14
4.3.3	Comparing cypher to SQL	19
5	Building a Neo4j Database	22
5.1	Creating a Proper Graph Model	23
5.2	CSV to Neo4j Database	23
5.2.1	Load CSV	23
5.2.2	Optimizing LOAD CSV	27
6	A simple Case study	28
6.1	Build a Graph model	28
6.2	Build a Neo4j Database from CSV	30
6.3	Visualize the Database with Bloom	32
7	Final Statements	36

2 Background and Significance

2.1 What is Neo4j?

Neo4j is an industry-leading NoSQL database platform. It stores data in a native graph database, using the Cypher query language to retrieve data from storage. A native graph database is a database that stores nodes connected in a graph data structure. These nodes are connected through direct pointers which represent all the relationships that a given node has with any other nodes. Neo4j offers tools such as their built-in graph data science and their graph visualization tool called Neo4j bloom. Neo4j also offers several plugins called connectors to get your database integrated with tools such as Apache Spark, Apache Kafka and business intelligence tools such as Tableau and Looker.

2.2 What is Cypher?

Cypher is Neo4j's native query language that is optimized for graph data structures. Cypher is inspired by SQL but has many improvements and advantages over SQL. Cypher implements pattern matching borrowed from the query language called SPARQL. Cypher makes use of basic ASCII characters to represent the graph nodes and the relationships between them, which makes queries easy to understand.

2.3 What is Neo4j's graph data science platform?

Neo4j's platform for data science is simply called Neo4j Graph Data Science. This platform offers a large selection of over 65 pre-tuned algorithms for analyzing relationships between nodes. It also offers in-graph machine-learning models ¹ that allow data scientists to extract new findings from the nodes and their relationships. Neo4j Graph Data Science is very powerful because it focuses on the relationships between nodes rather than having to extract these relationships through countless rows and columns as in a traditional database.

2.4 What is Neo4j Bloom?

Neo4j Bloom is Neo4j's graph visualization tool. It allows users to search the graph visually without needing to use any code. This can be very useful for businesses that wish to view their database quickly in an intuitive manner.

2.5 Why use Neo4j as opposed to other databases?

Neo4j's graph-style storage allows for relationships to be stored along with data which means data is connected at the time of storage, which differs from traditional style SQL databases where data is stored in tables. For example, since data is connected as it's stored in a Neo4j graph, a query can be completed using a deep graph traversal algorithm as opposed to an SQL query which might involve several join statements. This means queries can be completed at speeds that are orders of magnitude faster than an SQL query. Cypher statements are often much smaller and simpler than SQL statements making code much easier to maintain. Neo4j also offers more intuitive codeless visualization than an SQL database through Neo4j bloom, which can have advantages in the business field.

3 Project Goals and Objectives

3.1 Project Goal

The goal of my project is to gain an understanding of the Neo4j platform and demonstrate some of the advantages and disadvantages compared to a traditional SQL database.

3.2 Project objective

The project objectives are:

- Provide an In-depth look into how Neo4j functions
- Construct a Neo4j database on a sizeable dataset
- Perform some queries
- Demonstrate Neo4j Bloom visualization
- Discuss the advantages and disadvantages of Neo4j compared to an SQL database

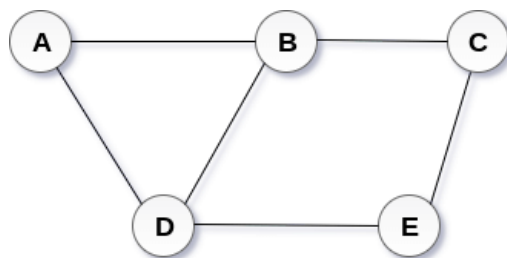
4 In-depth Look into Neo4j and Cypher

4.1 Neo4j functionality

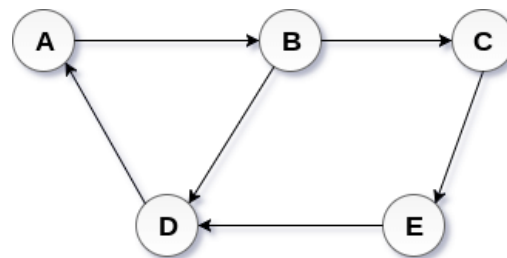
Unlike a traditional RDBMS like SQL which stores its data in tables, Neo4j stores its data in a connected graph. Due to the native graph structure, queries make use of optimized graph traversal algorithms that make complex queries very fast compared to traditional SQL queries. In order to understand the benefits of Neo4j, we must look further into what a graph database is.

4.1.1 What is a graph database?

In order to understand how a graph database works, we must understand what a graph is. A graph is a type of data structure where information is stored in nodes. These nodes can store anything from variable types to structures and classes. What makes a graph so powerful is that these nodes are connected through edges. Edges provide direct links between nodes which can be directed or undirected. However, in the case of a graph database, these edges are always directed. Because of the connected nature of nodes in a graph, search algorithms can take the optimal path in order to find the requested data. Let us consider the images below:



Undirected Graph



Directed Graph

Figure 1: An undirected graph

Figure 2: A directed graph

Now that we understand how graphs work, we must ask the question: how does this relate to databases? A graph database stores its information in the nodes of the graph which are linked together through edges that represent the relationship(s) between nodes in the graph.

A node is an entity in the graph. Nodes are tagged with labels that define what they represent in the database. For example, a label for a node could be "Person". This means that this node would represent a person object in the database. Nodes can also hold any number of key-value pairs. For example, when looking at our person object, they could have a key called "name" where the value stored in name would be the identifier (ID) of the object. Node labels can also contain metadata which can store information like constraints to certain nodes.

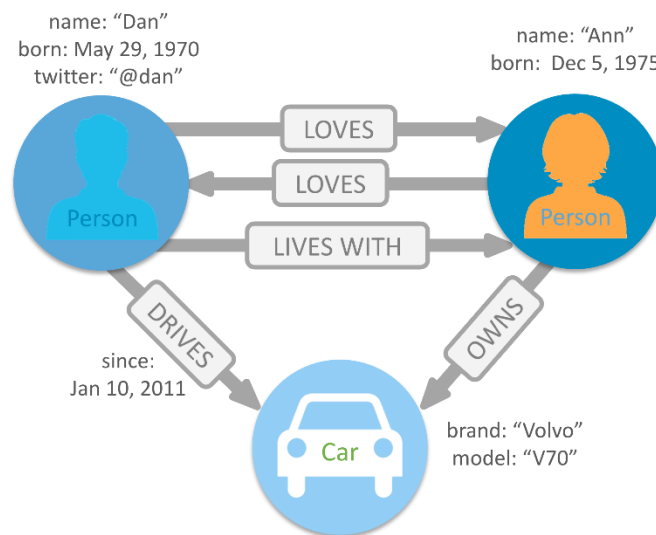


Figure 3: A diagram representing nodes and relationships

Relationships are the edges in our graph database. They provide directed, named connections between node entities. For example, we could have "Person" (node object) *loves* (relationship) "Person" (node object) as a connection in our database. Relationships always have a direction, a type, a start node and an end node. The type describes what the relationship represents. In the above example, *loves* would be the type of relationship between our two people nodes. The direction represents the connection between the start and the end node, with the direction following a path from the start node to the end node. Nodes can have any number of relationships without sacrificing performance which is due to the graph structure of the database. It is also important to note that although relationships are always directed, they can be traversed upon efficiently in any direction.

Neo4j offers constant time traversals in big graphs for both depth and breadth searches due to the efficiency between nodes and relationships, which allows databases on the scale of billions of nodes on moderate hardware. Because nodes and their relationships are already predefined and stored together, this makes writing queries very simple as opposed to traditional SQL. In fact, let's consider some comparisons between traditional RDBMSs like SQL and Neo4j.

4.1.2 Comparison of RDBMS to Graph Database

Relational databases such as SQL are known for their ability to store highly structured data in tables with pre-defined columns of specific types. Because of the strict organization of relational databases, developers must carefully structure the data used in their applications to fit the predetermined column types. In relational databases, references to other rows and tables are handles through the use of foreign key columns which contain primary key attributes of other tables. At query time, **joins** are done by matching foreign keys to the primary key specified. These operations are very heavy on the processor and memory which has an exponential cost relative to the number of joins needed. In Neo4j's graph database, joins are eliminated because the relationships between nodes are already stored with the specified nodes. Combine this with efficient graph traversal algorithms, and the speedups for complex queries are orders of magnitude faster with Neo4j. Observe the images below:

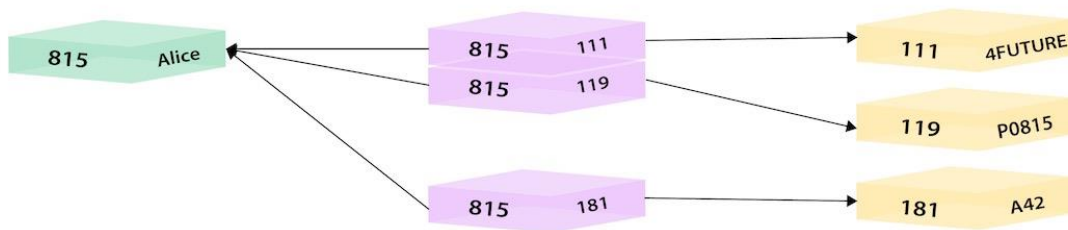


Figure 4: A data model for a Relational Database

This image represents a the structure of a traditional RDBMS for an Employee database. The table on the left is the person table which could potentially contain thousands of rows if we are dealing with a large data set. In SQL, we would have to search for the user of Alice and her person ID of 815. We would then have to go to the person-department table in the middle which stores pairs of person IDs and department numbers. We would then have to find all relevant rows that contain Alice's ID of 815 and the IDs for the departments she works in. We would then have to go to the table on the right and query the names of the departments based on the department IDs returned in the middle table. This query involves searching through many tables because of the references needed to map an employee to their department. Now consider the next image below:

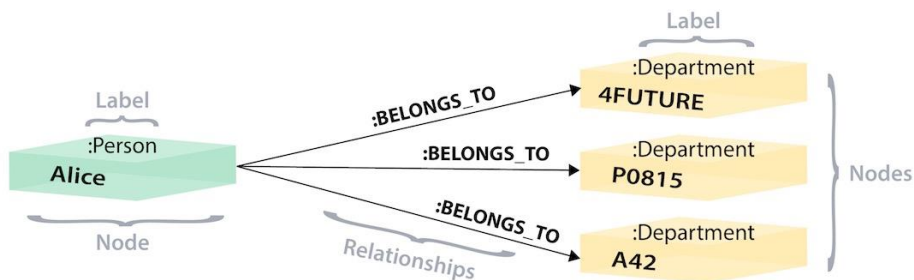


Figure 5: A data model for a graph database

This image represents a Neo4j graph databases structure for the same example above. We have a person node called Alice, which we would search the graph for. Due to the efficiency of graph traversal, this operation would be much faster than a lookup operation in an SQL table. To find out which departments she works in, we would just have to traverse all of the BELONGS_TO relationships from Alice to find the department nodes she is connected to. This method is far more efficient than an SQL lookup because there are no joins and searching multiple tables, just a single trip from Alice to her department nodes.

One might ask, how do I write queries for Neo4j? Do I need to understand graph theory and data structures? While an understanding of how graphs work is certainly helpful, Neo4j uses a query language called Cypher that is very similar to SQL, in fact, its syntax is based off of SQL. Cypher is a powerful query language that I will touch upon in detail further in the report.

Compared to traditional RDBMSs, visualizing your data in Neo4j is very intuitive. Neo4j uses Neo4j Bloom which is a graph visualization tool. This is a front-end tool that allows users to visually navigate and query the data without any query language or programming knowledge. I will touch on Bloom later in the report.

4.2 Neo4j Graph Platform

Neo4j offers a suite of tools and technologies that allow the user to use Neo4j in whatever manner suits them the best. Let us take a look at the core piece of Neo4j.

4.2.1 Neo4j Database

As previously explained, Neo4j's core piece is its graph database. It offers solutions to both transactional and analytical workloads on large data sets. It is optimized for traversing paths between the data using relationships between node entities.

Neo4j offers two types of database server solutions.

They offer a self-managed database server with two editions: community and Enterprise. Community is free and offers all the needed tools to build small scale projects. Some of the features offered are Open Source under GPLv3, fully featured native graph database, Cypher query language, fully ACID transactions and full compatibility with Neo4j Graph Data Science Community Edition. An enterprise server is meant for large scale server projects with the need for easy scaling. Can run in private cloud or public cloud infrastructure.

AuraDB is a fully managed graph database server specifically built for Neo4j offering a fully managed cloud service database that is always on and needs little administration. AuraDB is reliable and secure with a high level of autonomy which allows developers to focus on building graph databases without worrying about database management. There are three tiers to this service:

- AuraDB Free which is \$0 and is meant for small scale projects
- AuraDB professional which costs a minimum \$65 USD/Month and is meant for medium scale applications in advanced development environments
- AuraDB Enterprise whose cost varies on the scale of the project needed. For pricing information, direct contact with Neo4j is needed. This is meant for large scale, critical applications.

- Please note that these server solutions are meant for client server interactions and is different from the Neo4j desktop. Neo4j desktop is a IDE for local development only that include access to the Neo4j Enterprise features for free, but only on a local scale.

Please note that these server solutions are meant for client server interactions and is different from the Neo4j desktop. Neo4j desktop is a IDE for local development only that include access to the Neo4j Enterprise features for free, but only on a local scale. I will touch upon Neo4j Desktop later on in the report.

Neo4j offers multi-database technology that allows users to operate and manage multiple databases within a single installation of Neo4j DBMS. This technology allows data to be separated up by uses cases, sensitivity and application into various different databases. This is similar to how SQL allows you to manage multiple different database files. In Neo4j, developers can use Cypher query language to seamlessly switch between multiple databases within a single instance of Neo4j DBMS. The metadata for all Neo4j databases is stored in a special database called the system database. This is similar to how PostgreSQL uses the system catalogs and information schema to store metadata about databases.

Neo4j databases also offer a technology called Neo4j Fabric which is exclusive to the enterprise edition. This new feature introduced in the Neo4j 4.x edition allows developers to “shard” graph data by breaking down larger graphs into smaller graphs and store them in separate databases. Theses shards can be accessed individually or aggregated when required to see all of the data. Fabric also allows queries across multiple databases through the use of the Cypher programming language. Fabric makes it easy to abstract large datasets into smaller ones that are easier to work with but are still connected to the larger parent set they belong to. This technology could be revolutionary for large databases, and through my research have not found an equivalent technology in other DBMS.

Neo4j also offers reactive drives that allow users to implement Neo4j technology in other programming languages. This would be similar to sqlite3 for C++. The supported list of languages for which Neo4j drivers are supported is: Java, Spring, Neo4j-OGM, .NET, JavaScript, Python, Go, Ruby, PHP, Erlang/Elixir, Perl, C/C++, Clojure, Haskell and R. Neo4j also offers a native Database plugin for JetBrains IDEs such as PyCharm and IntelliJ.

Now that we have an understanding of how the core of Neo4j technology works, let us look into how to interface with it.

4.2.2 Neo4j Desktop

Neo4j Desktop is the native desktop IDE to develop and manage instances of Neo4j databases. It allows for the management of as many projects and local database servers as the user would like and even offers the ability to connect to remote Neo4j servers to run a local project. The remote Neo4j servers would be useful for a developer who lacks hardware sufficient enough to run a large-scale database server on their machine. Neo4j also comes with a free Developer Licensee of Neo4j Enterprise edition allowing full access to the suite of features Neo4j offers on a local scale.

It can also be integrated with the Neo4j self-managed database server solutions as well as the cloud database server solutions for larger scale applications that are not contained to the local machine.

Each server running a database can be configured and maintained through the desktop application, with no need to use the command line.

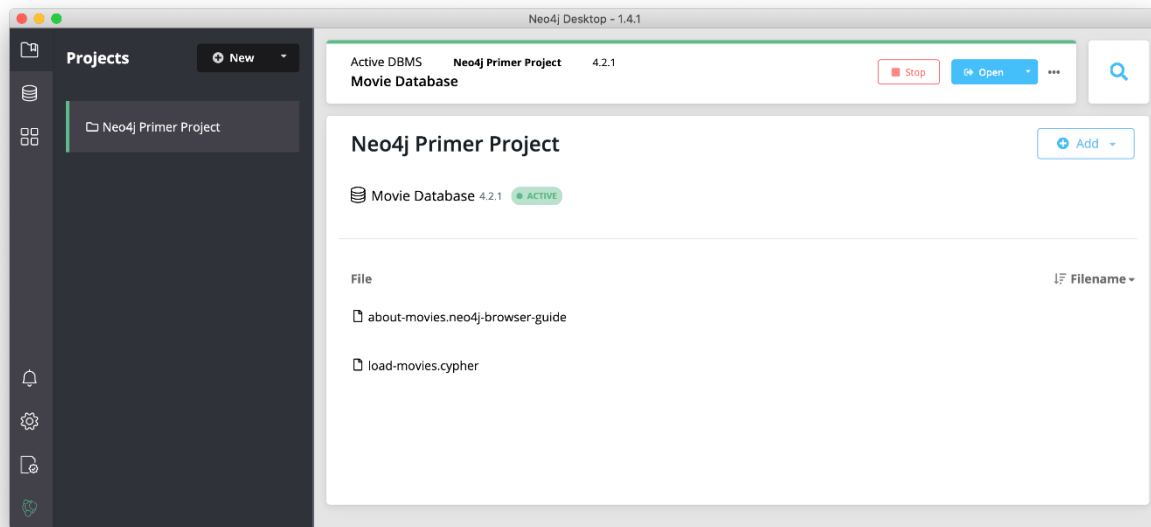


Figure 6: The landing screen of the Neo4j Desktop application

There is also a large number of extensions such as Awesome Procedures On Cypher (APOC) or the Graph Data Science Plugin. APOC offers optimized and streamlined versions of some of the most common procedures done using Cypher, making Cypher language even easier to write. The Graph Data Science plugin offers pre-made functions that can be used to perform data science on large data sets which allows developers to enable features that a data scientist requests without having to learn how to implement them.

Neo4j Desktop also offers Graph Apps, which are similar to IDE plugins. These allow Neo4j Browser and Bloom to be integrated right into the desktop application. Some of the most useful Graph Apps are tools that allow relational databases to be imported directly into Neo4j. I will touch upon this process later in the report. Now that we have an understanding of how the desktop application works, let's take a look at the web application for Neo4j.

4.2.3 Neo4j Browser

Neo4j browser is an interactive cypher command shell that allows the user to interact with their graph through queries and visualization. It comes standard with Neo4j and is available in all editions and versions of Neo4j. Neo4j browser can interact with a cloud database hosted on AuraDB and local databases running through Neo4j desktop. Neo4j browser contains built-in guides that can help users access the graph and manipulate it using Cypher. These guides are accessed through Cypher and an example can be seen below:

Title	Description	Command
Intro	A guided tour of Neo4j Browser	<code>:play intro</code>
Concepts	Graph database basics	<code>:play concepts</code>
Cypher	Neo4j's graph query language introduction	<code>:play cypher</code>
The Movie Graph	A mini graph model of connections between actors and movies	<code>:play movie graph</code>
The Northwind Database	A classic use case of RDBMS to graph with import instructions and queries	<code>:play northwind graph</code>
Custom Guides	Use <code>:play <url></code> to play a custom guide (custom guide documentation)	<code>:play</code> https://guides.neo4j.com/restaurant_recommendation

Figure 7: A table containing guides that can be run through Neo4j Browsers

Neo4j Browser allows the user to access the graph metadata. In the metadata, you can find things like currently used nodes, labels, relationship types and property keys. Clicking on any of these options will run a quick query to show a sample of the graph with those elements.

Any query run in the Neo4j browser app will show up in the area below the command line. There are also many viewing options like Graph view, table view and an ascii-table. An example of this can be seen below:

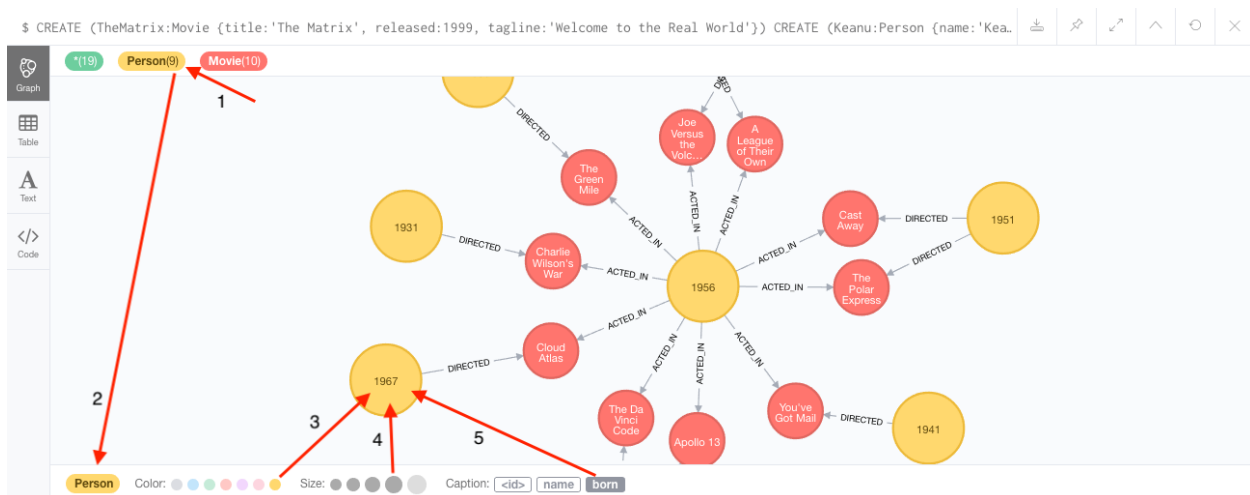


Figure 8: The result of a query done in Neo4j Browser

Overall, Neo4j browser offers a quick and lightweight solution to view your database through a browser. Let us move on to Neo4j's graph visualization tool.

4.2.4 Neo4j Bloom

Neo4j Bloom is a tool that allows users explore their graph database visually without any query programming knowledge. Users can write queries in a form similar to natural spoken language and bloom with retrieve data by traversing layers of the graph. Bloom also allows users who have the proper permissions to edit, update or correct the graph when needed.

An example of this can be seen below:

Suppose we had a database containing information about suppliers and their products. Lets say we wanted to find suppliers who make more than one product. We could run a palindrome search (search is the same if reversed) in the search bar and press enter to see the results. The search entered was: *Product Supplier Product*. The results are below

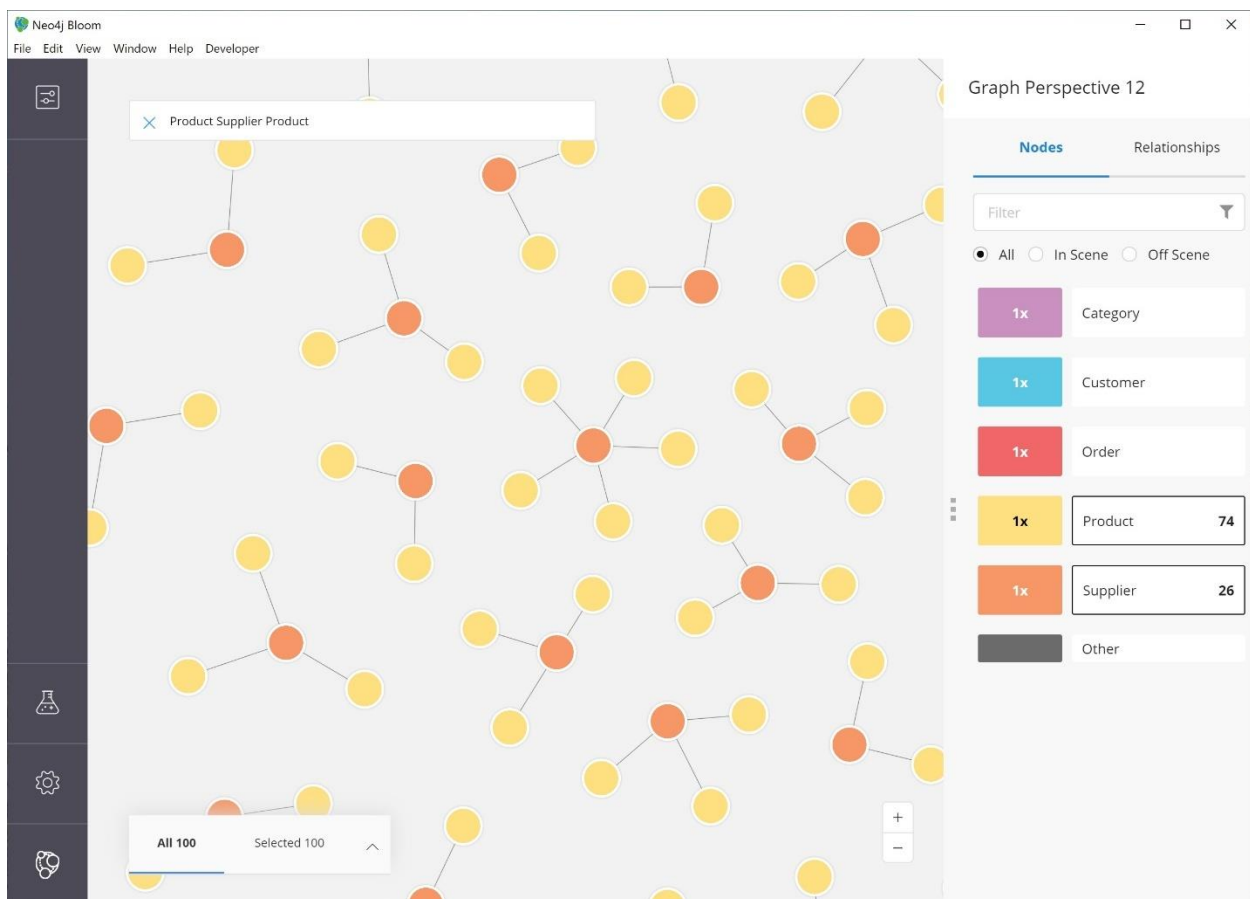


Figure 10: An example of a palindrome search done in Bloom

As we can see, with this simple search we now have all suppliers that supply more than one type of product. The suppliers are the orange nodes and the yellow nodes that are connected to each supplier are their products. This query required no programming and is simple to understand.

Bloom also provides a tool called *perspectives* that allow the user to create custom views of the graph to suit their viewing needs. These perspectives can be saved and viewed at any time through Bloom. This powerful tool allows for quick access to a databases through a variety of lenses. This is similar to SQL views in the sense that the user can create a custom way of viewing their data, but is much more intuitive. Lets take a look at some images below:

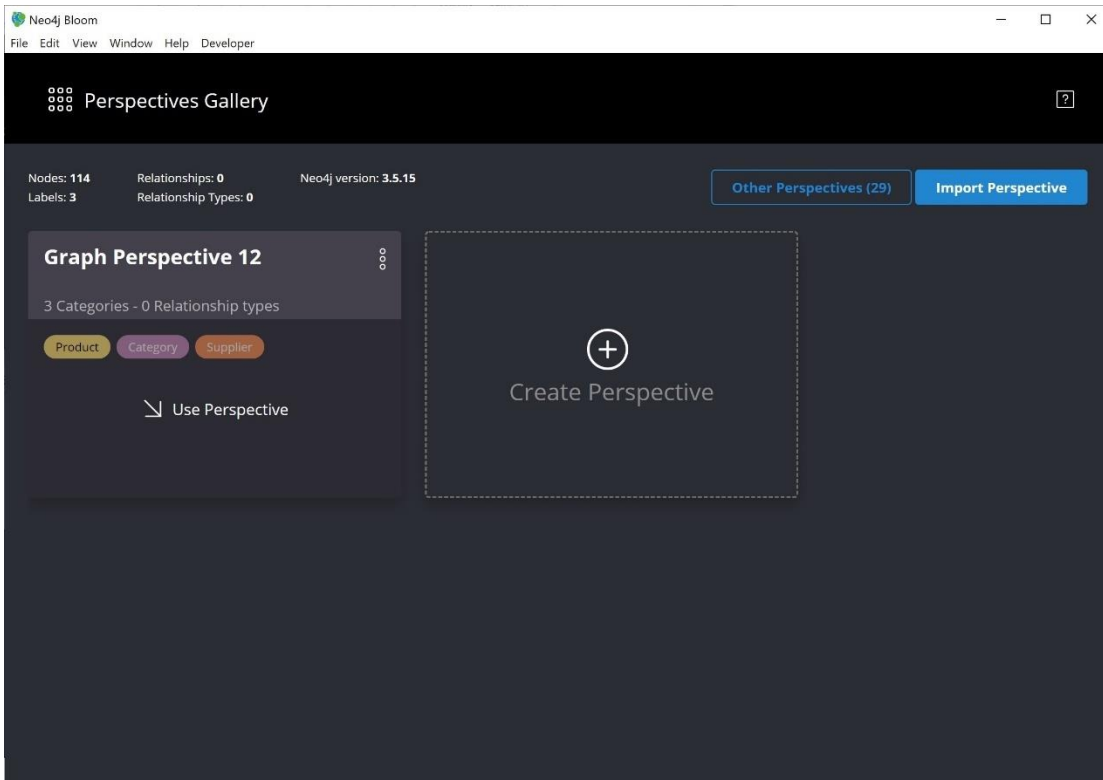


Figure 11: The perspective gallery

The perspective gallery shown above allows users to create and save custom views of their graph database. This can be very useful if company executives from different departments need different ways to view their database. For example, if a marketing executive wanted to view their database through a marketing lens, they could create a view that is centered around marketing information. In contrast, if an engineering executive wanted to view the database through an engineering lens, they could create a perspective centered around product properties and technical information.

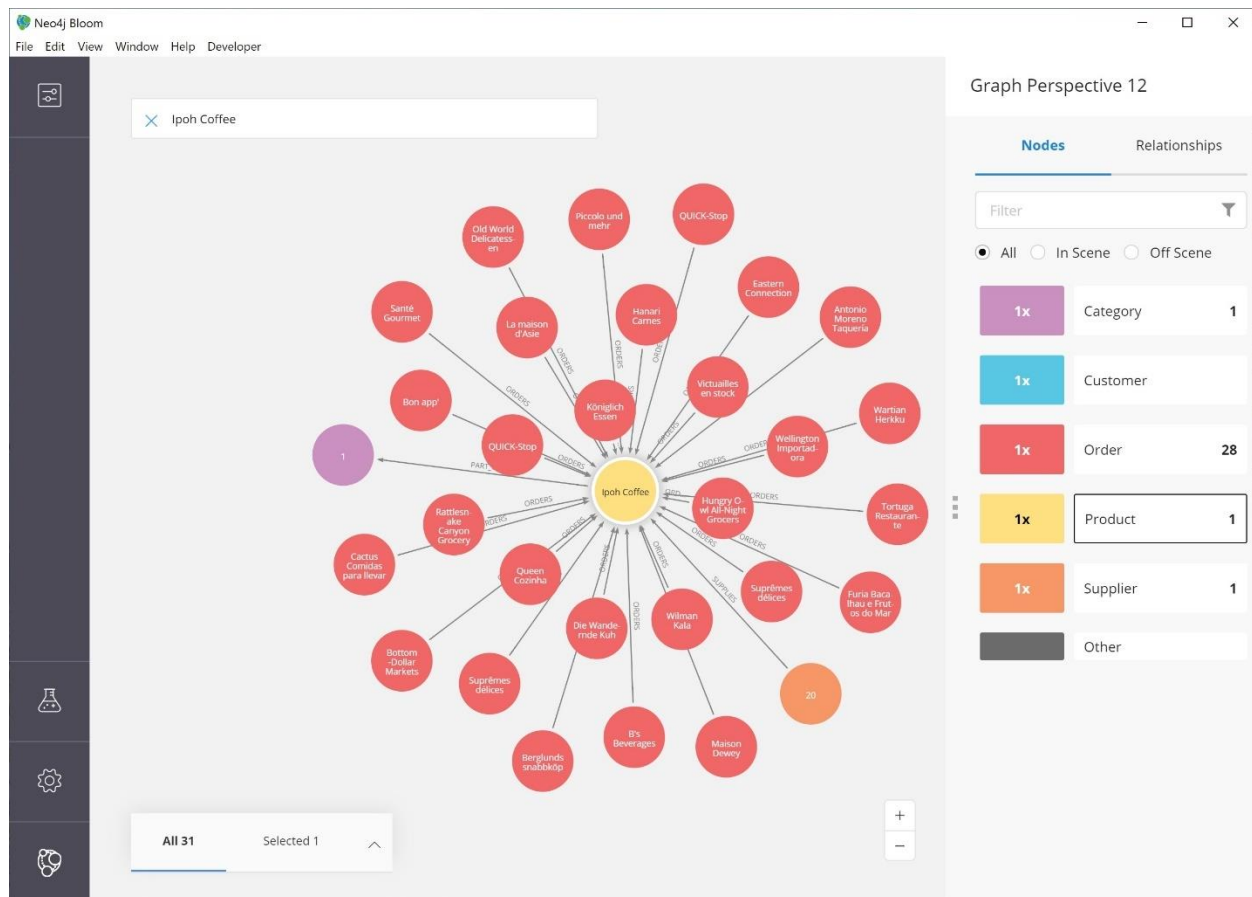


Figure 12: The result of a query done in bloom

In this image, we can see the layout of a bloom perspective. In this example, the user wanted to see order information related to their coffee product. In a database containing many products, having the ability to easily see all order information relating to one product without any programming required is quite impressive and demonstrates the power of Neo4j Bloom.

Neo4j bloom was developed designed for business analysts and other non-developers to interact with data without writing any code. It provides a simple way to view relationships between entities in the database as opposed to an SQL table where relationships between entities in separate tables are often difficult to query. Its kind of ironic that a relational database like SQL does not handle relationships between data intuitively.

Now that we have an overview of how to access data without programming queries, lets take a look at Cypher query language to understand to how powerful and simple it is compared to SQL.

4.3 Cypher Query Language

4.3.1 Introduction to Cypher

Cypher is Neo4j's native query language. It is basically SQL for graphs. The main difference between Cypher and SQL that cypher lets you focus on what data you wish to query the graph for without having

to worry about how the data is retrieved from the graph whereas in SQL you have to worry about how the data is queried like picking the right JOIN operation. The thing that really sets Cypher apart from SQL is the fact that there are no JOIN operations in Cypher because of the direct relationships stored in the graph. I will touch on direct comparisons between Cypher and SQL in section 4.4.3, but for now let's take a look at some examples of Cypher queries.

4.3.2 Some query examples

Before we take a look at some examples of queries, let's look at some keywords Cypher uses:

- **MATCH:** MATCH is the keyword used to tell Cypher that we want to search for something. It is analogous to SQL's SELECT. It can search for an existing node, relationship, label, property, or pattern in the database. Some use cases are:
 - Searching for all node labels in a database
 - Find all the nodes with a particular relationship
 - Look for patterns in nodes and relationships
- **RETURN:** The RETURN keyword specifies what values or results you might want to return from a query. RETURN is not needed when writing to the database, but is needed when reading the database. In order to return things in cypher queries you need to have variables specified in your MATCH statement for the data you want to return. An example can be seen below.

Cypher's queries are built in a unique manner that allows patterns and relationships to be matched visually. To illustrate what I mean, let us take a look at the following query:

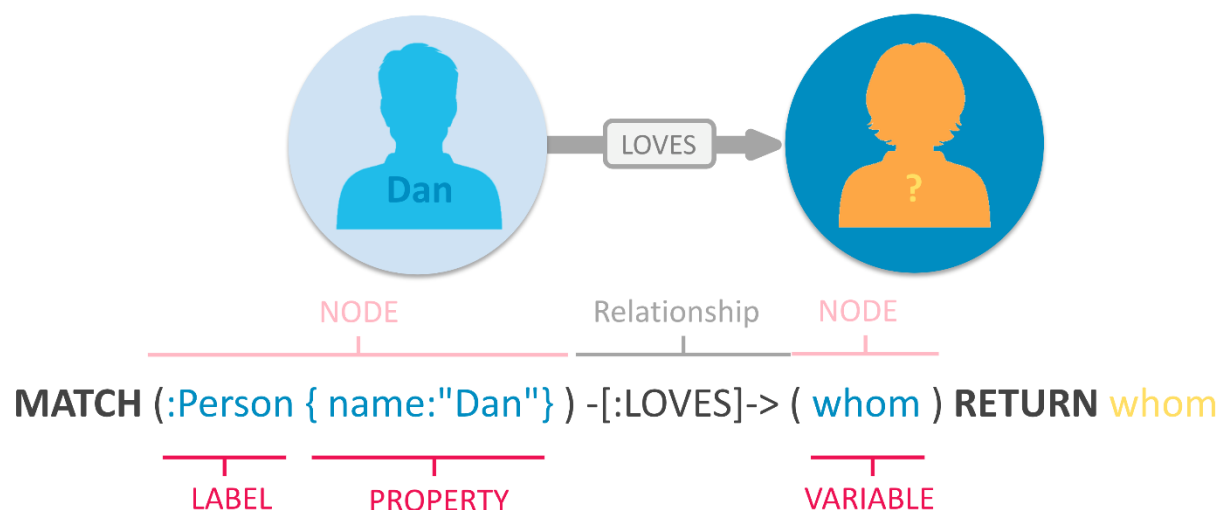


Figure 13: An image describing a Cypher Query

While at first, the syntax may look unique, it is quite easy to understand once we break it down. Let us consider the MATCH statement. This specific MATCH statement follows the syntax: `MATCH (node) -[:relationship]-> (node)`. Nodes are contained within rounded brackets. We can see within `(:Person {name:"Dan"})` we are referencing a node of Label (type) person where the name property is Dan. The most interesting part of this query is the relationship statement. The syntax for the relationship in this query is `-[:LOVES]->`, which means we are looking for the node that is connected to Dan's node by the relationship :LOVES. The next part of the query is another node with the syntax

(whom) where whom is the name of variable. Because we do not know who dan loves, we need to specify a variable for which the query result will be returned in. We can then see the RETURN keyword followed by whom. This specifies the variable that the query should return. All in all, this query does the following: return the node that is connected to Dan's through the *LOVES* relationship.

Let us take a look at some more queries varying in complexity. The following examples come from Neo4js documentation:

Example 1: Find person node with the name 'Tom Hanks'

```
MATCH (tom:Person {name: 'Tom Hanks'})  
RETURN tom
```

Figure 15: A cypher query

This query will find person nodes in the graph that name the name property 'Tom Hanks'. It stores the result in the variable tom and returns this variable. This query produced the following result:

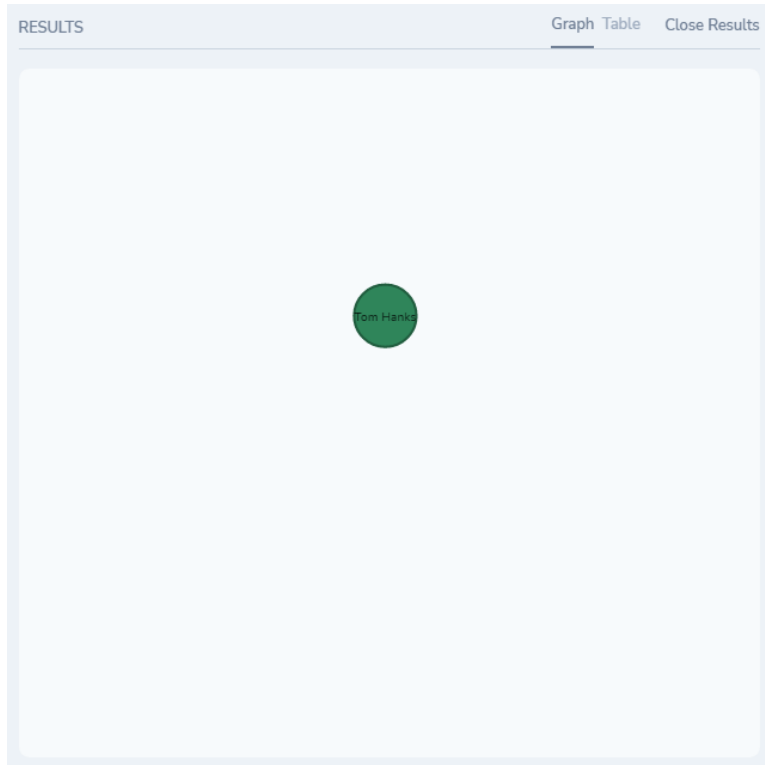


Figure 16: The query result

Example 2: Find the movies that Tom Hanks has directed

```
MATCH (:Person {name: 'Tom Hanks'})-[:DIRECTED]->(movie:Movie)
RETURN movie
```

Figure 17: A cypher query

We can see here that we need to access the person node with the name 'Tom Hanks'. We then follow the relationship called :DIRECTED to all nodes of type Movie. These get stored in the movie variable declared in (**movie**:Movie). We then return this variable called movie. The results are:



Figure 18: The query result

Example 3: Find the movie that Tom Hanks has directed, but return on the title of the movie

```
MATCH (:Person {name: 'Tom Hanks'})-[:DIRECTED]->(movie:Movie)
RETURN movie.title
```

Figure 19: A cypher query

The explanation is the same as the previous example. The only difference being that we don't return the whole node stored in the movie variable, but rather just its title property. This kind of query can be useful when you only want one property from a node rather than the whole node. The result is below:

RESULTS	Close Results
movie.title	
That Thing You Do	

Figure 20: The query result

Note: This result was returned in text format because we specified we only wanted a property rather than the whole node.

Notice that the above examples were fairly simple and only returned one node. What if we wanted to return multiple nodes with relationships between them. This is where subqueries come in. Subqueries are basically queries within a query. The closest SQL interpretation would be a JOIN statement that would allow us to return multiple results. Let us take a look at a type of subquery called an existential subquery. We can really see the power of Neo4j's joinless queries in the following example:

Suppose we wanted to find out the friends of someone who was an employee at Neo4j, the query would look something like this:

Example 4:

```
MATCH (p:Person)-[r:IS_FRIENDS_WITH]->(friend:Person)
WHERE EXISTS {
  MATCH (p)-[:WORKS_FOR]->(:Company {name: 'Neo4j'})
}
RETURN p, r, friend
```

Figure 21: The query result

The line `MATCH (p:Person)-[r:IS_FRIENDS_WITH]->(friend:Person)` creates a person variable called `p` and traverses `p`'s `IS_FRIENDS_WITH` relationship to find person nodes that get stored in the `friend` variable. We then have the following snippet:

```
WHERE EXISTS {
  MATCH (p)-[:WORKS_FOR]->(:Company {name: 'Neo4j'})
}
```

This snippet runs a subquery that evaluates `p` as a person who works for the company 'Neo4j'. This is an existential subquery because of the `WHERE EXISTS` clause`{}`. This transforms the query into just returning friends of the node `p`, into a query that returns friends of the node `p` where node `p` is a person who works for Neo4j. This type of pattern filtering allows for super fast queries that are not that costly. We return the result with the following line: `RETURN p, r, friend`. '`p`' is the variable that represents the person who works at Neo4j. '`r`' is the relationship that we want returned, in this case it is `IS_FRIENDS_WITH`. '`friend`' is the variable that contains all nodes traversed up from '`p`' using '`r`'. The result can be seen below:

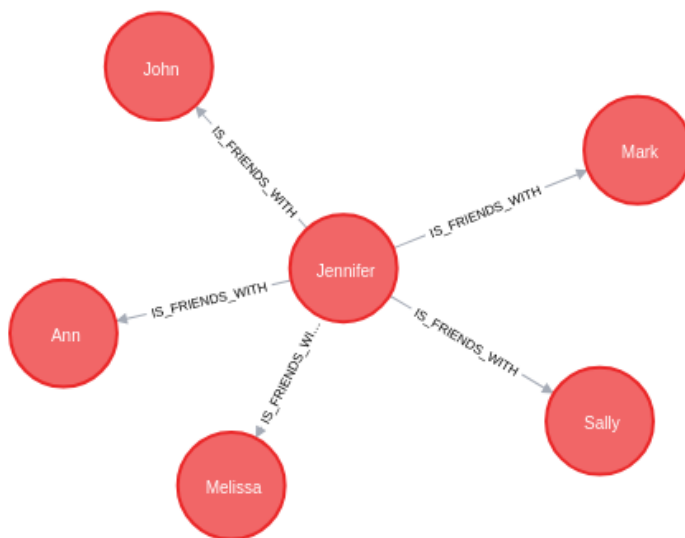


Figure 22: The query result

We can see that we are returned a smaller subgraph from the parent graph database. Jennifer is the person who works at Neo4j and the nodes connected to Jennifer through the IS_FRIENDS_WITH relationship are her friends.

We are only scratching the surface of what Cypher can do in terms of querying, but let us move on to some direct comparisons between Cypher and SQL to illustrate their similarities and differences.

4.3.3 Comparing cypher to SQL

Cypher and SQL are textual query languages where the programmer defines what data they would like returned in the query. The main difference is that Cypher is written to traverse graphs whereas SQL is written to traverse tables.

SQL and Cypher contain a lot of similar keywords, clauses and functions. Some the things they share are

- WHERE
- ORDER BY
- SKIP
- LIMIT
- Parameter accessing through the syntax `p.unitPrice > 10` where `p` is a product and `unitPrice` is the unit price of the product. `p.unitPrice > 10` would limit the returned product to those who have unit price greater than 10

This is where the similarities start to fade. Cypher is about easily expressing patterns within the data in the graph whereas in SQL, a lot of effort is required to analyze relationships between data entities. The largest advantage Cypher has over SQL is that Cypher has no need for JOINS because relationships are stored with the data, unlike SQL.

While Cypher shares many keywords and clauses with SQL, its syntax is slightly different. When expressing a relationship in a query, the programmer will type `–[:RELATIONSHIP_NAME]->`, where `RELATIONSHIP_NAME` would be the name of the relationship they are referencing. This unusual syntax where an arrow is visually written really emphasizes that this language is all about traversing relationships amongst data entities. To achieve the same thing in SQL, we would require Foreign Keys and JOIN statements which can be tricky to write and read. Let us compare some Cypher statements to their equivalent SQL statements:

Similar Queries between SQL and Cypher:

Find and Return all Products:

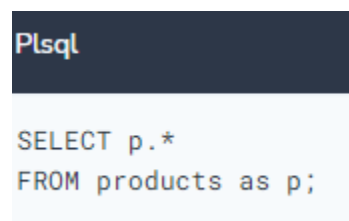


Image 23: Some SQL code

Cypher

```
MATCH (p:Product)
RETURN p;
```

Image 24: Equivalent Cypher code

As we can see, these queries are very similar. They both return all products stored in a database. The difference is in SQL we have to specify the table we wish to access whereas in Cypher we specify the type of node we wish to return.

Filtering By Equality:

In this query, we wish to return the name of a product and its price where the product has a certain name.

Plsql

```
SELECT p.ProductName, p.UnitPrice
FROM products AS p
WHERE p.ProductName = 'Chocolade';
```

Image 25: Some SQL code

Cypher

```
MATCH (p:Product)
WHERE p.productName = "Chocolade"
RETURN p.productName, p.unitPrice;
```

Image 26: Equivalent Cypher code

Again, we can see the similarities between SQL and Cypher as their syntaxes are very similar. In this query we wish to return the product name and unit price of a product that has the name "Chocolade".

Queries that are not similar in Cypher and SQL:

Joining Products and Customers

In this query, we wish to see the customers that are companies that ordered the product called “Chocolate”. We want this result to be distinct as well, meaning we only want the company name returned once, even if they have multiple orders of this product.

```
Plsql

SELECT DISTINCT c.CompanyName
FROM customers AS c
JOIN orders AS o ON (c.CustomerID = o.CustomerID)
JOIN order_details AS od ON (o.OrderID = od.OrderID)
JOIN products AS p ON (od.ProductID = p.ProductID)
WHERE p.ProductName = 'Chocolate';
```

Image 27: Some SQL code

```
Cypher                                Copy to Clipboard

MATCH (p:Product {productName:"Chocolate"})<-[:PRODUCT]-(:Order)<-[:PURCHASED]-(c:Customer)
RETURN distinct c.companyName;
```

Image 28: Equivalent Cypher code

This is where we really see the power behind Cypher and Neo4j. IN the above SQL example, we need three JOINS to match the customers to orders, then orders to order details, and then finally order details to products. These joins can be quite costly when compared to Cyphers relationship system. We can see that the Cypher query is much shorter and easier to read. If we follow the directions of the arrows, we can read the statement as find customers who have purchased an order that is of product “Chocolate”. Then return the company name in distinct fashion.

Top-Selling Employees

This querying involves aggregation of data from two tables and then grouping the data by some certain attribute.

```
Plsql

SELECT e.EmployeeID, count(*) AS Count
FROM Employee AS e
JOIN Order AS o ON (o.EmployeeID = e.EmployeeID)
GROUP BY e.EmployeeID
ORDER BY Count DESC LIMIT 10;
```

Image 29: Some SQL code

```
Cypher

MATCH (:Order)<-[ :SOLD]-(e:Employee)
RETURN e.name, count(*) AS cnt
ORDER BY cnt DESC LIMIT 10
```

Image 30: Equivalent Cypher code

In this example, we want to return the number of orders sold by the top 10 selling employees sorted in descending order. We can see in SQL that this requires a join between the order tables and the employee tables around the employee ID key. We then have to group the results around the Employee ID key and then order by count in descending order. In Cypher, we just specify the relationship we would like to return and then return employee name followed by the count function. We then specify how we would like the data ordered. Notice that in the Cypher query there was no GROUP BY statement. How does Cypher now how to group the data? In cypher, when using an aggregation function, grouping becomes implicit. As soon as an aggregation function is used, all non-aggregated columns become grouping keys. In this example, we aggregated the count of orders for each employee. Since we aggregated on orders, employee ID automatically becomes the grouping key.

Again, we are just looking at the power of Cypher compared to SQL from a pretty high level, there is much more that Cypher is capable of that SQL isn't. However, we have demonstrated some similarities and differences between Cypher and SQL that should help make clear the distinction between the two languages. Now that we have an understanding of how to interact with a Neo4j database, let us go over how to construct one in various manners.

5 Building a Neo4j Database

Let us now examine how to design a Neo4j database. We will look at creating a graph model from scratch and importing a CSV into a Neo4j Database to populate it. Note that Neo4j has some pretty cool technology that will allow developers to convert an existing relational database into a Neo4j graph database. It is documented thoroughly and explains the conceptual differences between relational DBs

and graph DBs. However, to keep the scale of the project manageable, I will not be looking into that technology in this report. If you would like to learn more about this technology, refer to this [Neo4j Docs](#).

5.1 Creating a Proper Graph Model

Creating a proper graph model is a fairly similar process to designing an object model or an entity relationship diagram for a RDBMS. Graph modeling is the process in which a developer describes some arbitrary domain as a connected graph of nodes to store information and relationships to connect these nodes together. A good starting point would be to figure out what the domain of your model is. The domain entails all of the types of nodes that will be stored along with their properties and relationships. While this might sound like a daunting task, its best to break up the domain into scenarios to figure out what is needed. Consider a database for a library; what kinds of information would we need to store in our database. If we break it down into a scenario of **a person borrowing a book**, we might node to represent a person and their properties, a node to represent the book and its properties and a relationship to connect the person to the book they just borrowed. While this is not an exhaustive list of everything a library database system would need, it is a good analogy to understand the process of breaking down a large domain into smaller scenarios to come up with everything needed for the database. This process can be followed until a sizeable structure has been built that will suffice for the domains needs. Let us now consider how to convert a database designed for a RDBMS into a graph architecture.

5.2 CSV to Neo4j Database

A CSV or file of comma separated values is the most common form of storing data in an organize manner. Neo4j offers different ways of importing CSV data into a Neo4j database which are:

1. **LOAD CSV Cypher command:** this command is a great starting point and handles small- to medium-sized data sets (up to 10 million records). *Works with any setup, including AuraDB.*
2. **neo4j-admin bulk import tool:** command line tool useful for straightforward loading of large data sets. *Works with Neo4j Desktop, Neo4j EE Docker image and local installations.*
3. **Kettle import tool:** maps and executes steps for the data process flow and works well for very large data sets, especially if developers are already familiar with using this tool. *Works with any setup, including AuraDB.*

While neo4j-admin and Kettle are useful, I will only talk about LOAD CSV in depth as it is the most common method of importing data into a Neo4j database and it what I will use in my case study. Note that section 5.2.1 will just explain how LOAD CSV functions. To see it in action, refer to the case study.

5.2.1 Load CSV

LOAD CSV is a clause that is part of the Cypher query language. It is not just a basic tool to ingest data but rather a mechanism that combines multiple functions into a single operation. LOAD CSV is capable of:

- Supports loading / ingesting CSV data from a URL
- Directly maps input data into complex graph/domain structure
- Handles data conversion

- Supports complex computations
- Creates or merges entities, relationships, and structure

CSV files can be read locally or through remotely through a URL access.

Here is an example of LOAD CSV command for a local file:

```
Cypher Examples

//Example 1 - file directly placed in import directory (import/data.csv)
LOAD CSV FROM "file:///data.csv"

//Example 2 - file placed in subdirectory within import directory (import/northwind/customers.csv)
LOAD CSV FROM "file:///northwind/customers.csv"
```

Figure 31: Some Cypher code that is used to load from a CSV

Here is an example of a LOAD CSV from a remote file:

```
Cypher Examples

//Example 1 - website
LOAD CSV FROM 'https://data.neo4j.com/northwind/customers.csv'

//Example 2 - Google
LOAD CSV WITH HEADERS FROM 'https://docs.google.com/spreadsheets/d/<yourFilePath>/export?format=csv'
```

Figure 32: Some Cypher code that is used to load from a CSV

An important note for web based file: Since these files can be directly accessed through a URL, it is important to make sure that the proper permissions are set on the file side of things so that it can be externally accessed.

The above examples are very simple load command that do not contain any data cleaning or conversions. Let's look at some tips about how we can prepare our data before importing it into Neo4j:

- All data from the CSV file is read as a string, so you need to use `toInteger()`, `toFloat()`, `split()` or similar functions to convert values.
- Check your Cypher import statement for typos. Labels, property names, relationship-types, and variables are **case-sensitive**.
- The cleaner the data, the easier the load. Try to handle complex cleanup/manipulation before load.

It is also important to have a solid understanding of the graph model for your database. This is because when creating your database through LOAD CSV, you need to specify how you want Cypher to create nodes and relationships from the CSV data. In order to do this, you must thoroughly understand your graph data model so you know how to deal with any deviations from the model should they arise.

Let us consider a more complex case where we need to deal with empty fields in our CSV. Neo4j does not store null values, so in this case null values can be skipped or replaced with a default value. For example:

Suppose we have a CSV file containing information about a company:

```
None companies.csv

Id,Name,Location,Email,BusinessType
1,Neo4j,San Mateo,contact@neo4j.com,P
2,AAA,,info@aaa.com,
3,BBB,Chicago,,G
```

Figure 33: A CSV file

We can see that some fields are missing like for ID 2, the location is missing. How does Cypher deal with this?

Below are three examples of how you could deal with null fields in a CSV with Cypher. Between each method is a function that will clear the data from the database

```
Cypher Examples

//skip null values
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
WITH row WHERE row.Id IS NOT NULL
MERGE (c:Company {companyId: row.Id});

// clear data
MATCH (n:Company) DELETE n;

//set default for null values
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
MERGE (c:Company {companyId: row.Id, hqLocation: coalesce(row.Location, "Unknown")})

// clear data
MATCH (n:Company) DELETE n;

//change empty strings to null values (not stored)
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
MERGE (c:Company {companyId: row.Id})
SET c.emailAddress = CASE trim(row.Email) WHEN "" THEN null ELSE row.Email END
```

Figure 34: Some Cypher code that is used to load from a CSV

The first example will skip rows in the CSV that are missing a company ID.

The second example will replace null location values with “Unknown” using the coalesce function.

The third example will simply remove parameters that are null from objects that have null values in the CSV.

Let us note the MERGE clause. This clause can match existing nodes and bind them or creates new data for a new node and binds that do the node.

Let us look at another case where we might have a CSV that stores information about employees. Lets say there is a field in the CSV that is a list separated by a delimiter such as a “:”. Lets say this list contains skills that an employee has. We can use Cypher’s split function to split up attributes and relate them to our node.

Consider the following CSV file:

```
None employees.csv

Id,Name,Skills,Email
1,Joe Smith,Cypher:Java:JavaScript,joe@neo4j.com
2,Mary Jones,Java,mary@neo4j.com
3,Trevor Scott,Java:JavaScript,trevor@neo4j.com
```

Figure 35: A CSV file

We could create an array of skills for each employee with the following cypher code:

```
Cypher Example

LOAD CSV WITH HEADERS FROM 'file:///employees.csv' AS row
MERGE (e:Employee {employeeId: row.Id, email: row.Email})
WITH e, row
UNWIND split(row.Skills, ':') AS skill
MERGE (s:Skill {name: skill})
MERGE (e)-[r:HAS_EXPERIENCE]->(s)
```

Figure 36: Some Cypher code that is used to load from a CSV

Using the line UNWIND split(row.Skills, ':') AS skill we split the Skills list from the row by the delimiter ‘:’ and reference each split skill as ‘skill’ variable.

On the line MERGE (s:Skill {name: skill}), we create a variable s that is of type skill and bind the name property of the skill to the ‘skill’ variable created on the previous line. This will create a skill node for every skill that an employee has.

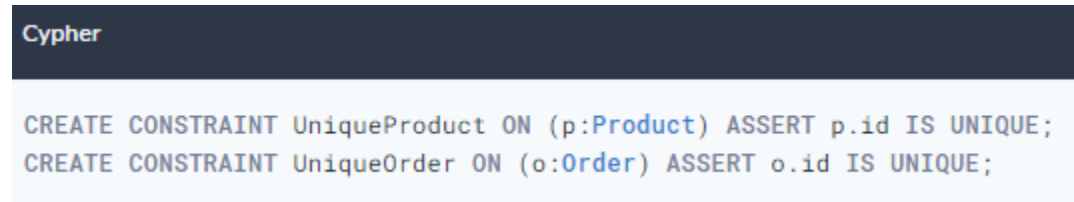
The following line MERGE (e)-[r:HAS_EXPERIENCE]->(s) will create a relationship called HAS_EXPERIENCE between an employee represented by ‘e’ and a skill represented by ‘s’

This covers the general skills needed to construct a Node4j database from a CSV file. Let us now talk about some optimization tips to help with efficient database construction

5.2.2 Optimizing LOAD CSV

There are many ways that we can improve the performance of our data loading sequence. One of the most important optimizations to make is to create indexes and constraints for each label and property you plan to MERGE or match on. A good rule of thumb is to always MATCH and MERGE on a single label with the indexed primary-key property.

To add a constraint on a primary key, let us look at the following cypher code:

A screenshot of a code editor with a dark blue header bar containing the word 'Cypher' in white. Below the header, the code is displayed on a light blue background. The code consists of two lines: 'CREATE CONSTRAINT UniqueProduct ON (p:Product) ASSERT p.id IS UNIQUE;' and 'CREATE CONSTRAINT UniqueOrder ON (o:Order) ASSERT o.id IS UNIQUE;'.

```
Cypher

CREATE CONSTRAINT UniqueProduct ON (p:Product) ASSERT p.id IS UNIQUE;
CREATE CONSTRAINT UniqueOrder ON (o:Order) ASSERT o.id IS UNIQUE;
```

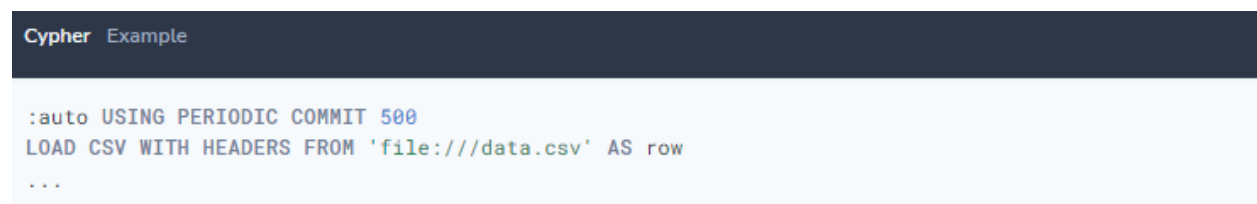
Figure 37: Some Cypher code creating key constraints

This ensures that each data integrity is not compromised forcing a no-duplicate rule while also indexing the product id and the order id with the constraint for faster queries. In the above example, these constraints would be created before any data is added in via CSV. This will ensure that whenever a new node of product type and order type is created, the entities are guaranteed to be unique and indexed.

Sometimes when a large amount of data is being loaded in via LOAD CSV, the data might be too large to fit into memory. If this is the case, here are some approaches that combat running out of memory on data load:

Batch import into sections with PERIODIC COMMIT to clear memory and transaction state

Some sample code for this is below:

A screenshot of a code editor with a dark blue header bar containing the words 'Cypher Example' in white. Below the header, the code is displayed on a light blue background. The code consists of three lines: ':auto USING PERIODIC COMMIT 500', 'LOAD CSV WITH HEADERS FROM 'file:///data.csv' AS row', and '...'.

```
Cypher Example

:auto USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM 'file:///data.csv' AS row
...
```

Figure 38: Some cypher code writing an optimization

Avoid using EAGER operator to reduce extra processing.

- To avoid EAGER loading, you can run PROFILE on your queries to check if they use it and modify queries or run multiple passes on the file to prevent it.

Adjust configuration for database on heap and memory to avoid page-faults

- dbms.memory.heap.initial_size and dbms.memory.heap.max_size: set to at least 4G.

- `dbms.memory.pagecache.size`: ideally, value large enough to keep the whole database in memory.

Now that we have covered most of the functionality needed to understand how a Neo4j database works. Let us move on to a simple case study demonstrating the capabilities of this DBMS.

6 A simple Case study

To demonstrate Neo4j, I have decided to build a database that will contain the information about the All-time Top 2000 songs on Spotify. The dataset is from Kaggle and can be found [here](#). I picked this dataset because it is large enough to showcase the power of a graph database but not so large that it is overwhelming. This dataset also had a very high useability rating on Kaggle meaning I would not have to do much prep to the data to make it useable. I only had to remove one column from the dataset because values were duplicates of another column. There are no null values, and all the data has a purpose. Another reason I picked this dataset is because it is easy to translate to a graph model right away. A lot of datasets are optimized for relational databases, which Neo4j has tools to help with. However, for this case study, I wanted to demonstrate how efficient Neo4j can be when using a well-designed graph model. We can now start building the graph model.

6.1 Build a Graph model

Before we can start building a model for our database, let us take a look at the rows and columns of this CSV. Each row represents a song. Each column represents a parameter for that song. We have 14 columns which go as follows:

- Index: ID
- Title: Name of the Track
- Artist: Name of the Artist
- Genre: Genre of the track
- Year: Release Year of the track
- Beats per Minute: The tempo of the song
- Energy: The energy of a song - the higher the value, the more energetic the song is
- Danceability: The higher the value, the easier it is to dance to this song.
- Loudness: The higher the value, the louder the song.
- Valence: The higher the value, the more positive mood for the song.
- Length: The duration of the song.
- Acoustic: The higher the value the more acoustic the song is.
- Speechiness: The higher the value the more spoken words the song contains
- Popularity: The higher the value the more popular the song is.

Now that we have an idea of the rows and columns in our database, we can create a graph model. The model is below

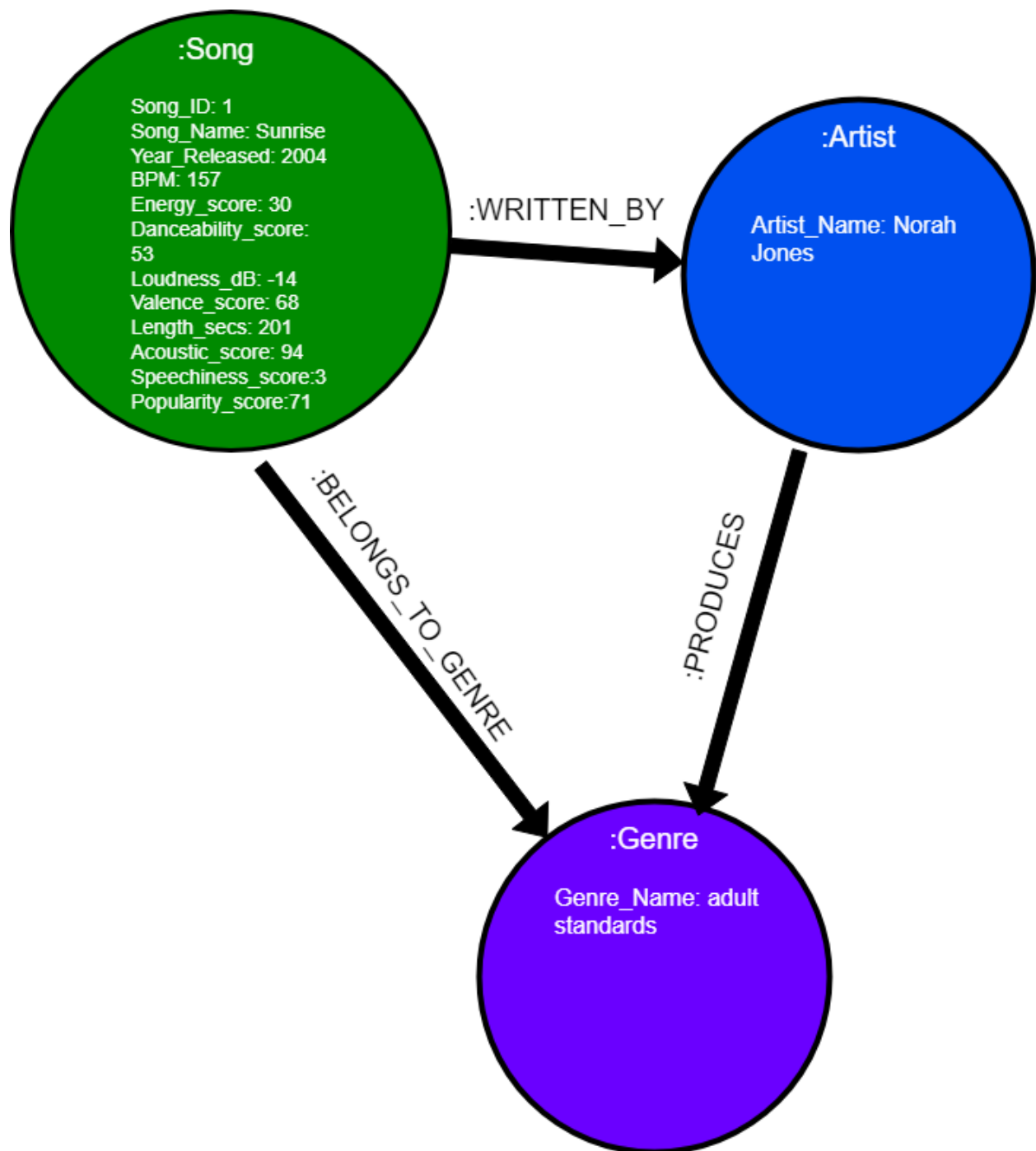


Figure 39: A graph model representing our dataset

We have 3 node objects:

- Song: which contains properties about the song
- Artist: which contains the artists name
- Genre: which contains the genre of music

We have 3 relationships:

- Song -WRITTEN_BY-> Artist: which maps a song to an artist
- Song -BELONGS_TO_GENRE-> which maps a song to a genre
- Artist -PRODUCES-> Genre which maps an artist to a genre of music the produce

Now that we have a strong graph model, we can start importing the data in Neo4j.

6.2 Build a Neo4j Database from CSV

For an in depth video of my case study, refer to the project deliverables. To be concise, I will just explain what I did through text

To start my study, I created a new project in the Neo4j Desktop application.

Then, I created a new instance of a Neo4j database system for this project.

The next task was to tell Neo4j where to find my csv file. This proved to be a bit tricky as some of the documentation was conflicting on how to do this. A quick check on StackOverflow pointed me in the right direction. I had to place my csv file in the following folder:

- C:\Program Files\Neo4j\relate-data\dbmss\dbms-91bb5ed0-c4e4-4b2d-a211-5baf74c1184f\import\
Where the red text specifies the specific instance of the Neo4j database we are working on.

After figure this problem out, I launched this instance of Neo4j through Neo4j browser which acts as a front end to interact with the backend running on Neo4j Desktop.

Once Neo4j could find my csv file, I used the following Cypher code to create the database:

```
LOAD CSV WITH HEADERS FROM 'file:///dataset.csv' AS row
MERGE (S:Song {
  Song_ID: row.Song_id,
  Song_Name: row.Title,
  Year_Released: row.Year,
  BPM: row.BPM,
  Energy_score: row.Energy,
  Danceability_score: row.Danceability,
  Loudness_dB: row.Loudness,
  Valence_score: row.Valence,
  Length_secs: row.Length,
  Acoustic_score: row.Acousticness,
  Speechiness_score: row.Speechiness,
  Popularity_score: row.Popularity
})
MERGE (a:Artist {Artist_Name: row.Artist})
MERGE (g:Genre {Genre_Name: row.Genre})
MERGE (S)-[r1:WRITTEN_BY]->(a)
MERGE (S)-[r2:BELONGS_TO_GENRE]->(g)
MERGE (a)-[r3:PRODUCES]->(g)
```

Figure 40: The Cypher code used to read the data from the CSV and build our database

The LOAD CSV line loads the dataset row by row and stores the row in AS row.

For each row in the csv we create:

- A song node with the MERGE (S:Song {attribute list}) where attribute list is all the attributes contained in each song node. Each song is unique along the song_id attribute which will allow for us to have songs that have the same title, but are not the same song.
- An artist node MERGE (a:Artist {Artist_Name: row.Artist}) command. Note, by using merge, we avoid creating duplicate artists if an artist's name is in multiple rows.
- A genre node created by MERGE (g:Genre {Genre_Name: row.Genre}). Similarly to the artist node, we avoid creating duplicate nodes due to merge.

Once all the nodes are created, we then create the relationship objects that will connect our nodes.

- The Song -WRITTEN_BY-> Artist relationship with MERGE (S)-[r1:WRITTEN_BY]->(a) which maps a song to an artist. This is a one-to-one relationship when considering song to artist because in this implementation there is only one artist per song. From the perspective of the artist, this becomes a one-to-many relationship as an artist can have many songs.
- The Song -BELONGS_TO_GENRE-> Genre relationship with MERGE (S)-[r2:BELONGS_TO_GENRE]->(g) that maps a song to a certain genre. In this implementation, this is a one-to-one relationship for the song as the song can only belong to one genre, but a one-to-many relationship for the genre as one genre can have many songs.
- The Artist -produces-> Genre relationship with MERGE (a)-[r3:PRODUCES]->(g) that maps an artist to a genre. This is a many to many relationship from the perspectives of both the artist and the genre.

Once this cypher query is executed, our database is built with nodes and relationships. To get an understanding of what the database looks like. Let us use Bloom in action.

6.3 Visualize the Database with Bloom

Bloom is Neo4j's graph visualization tool. To open bloom, I exited out of the Neo4j browser app, restarted the database on Neo4j Desktop to make sure it was updated and then launch Neo4j Bloom from the desktop application.

Let's take a look at what we see:

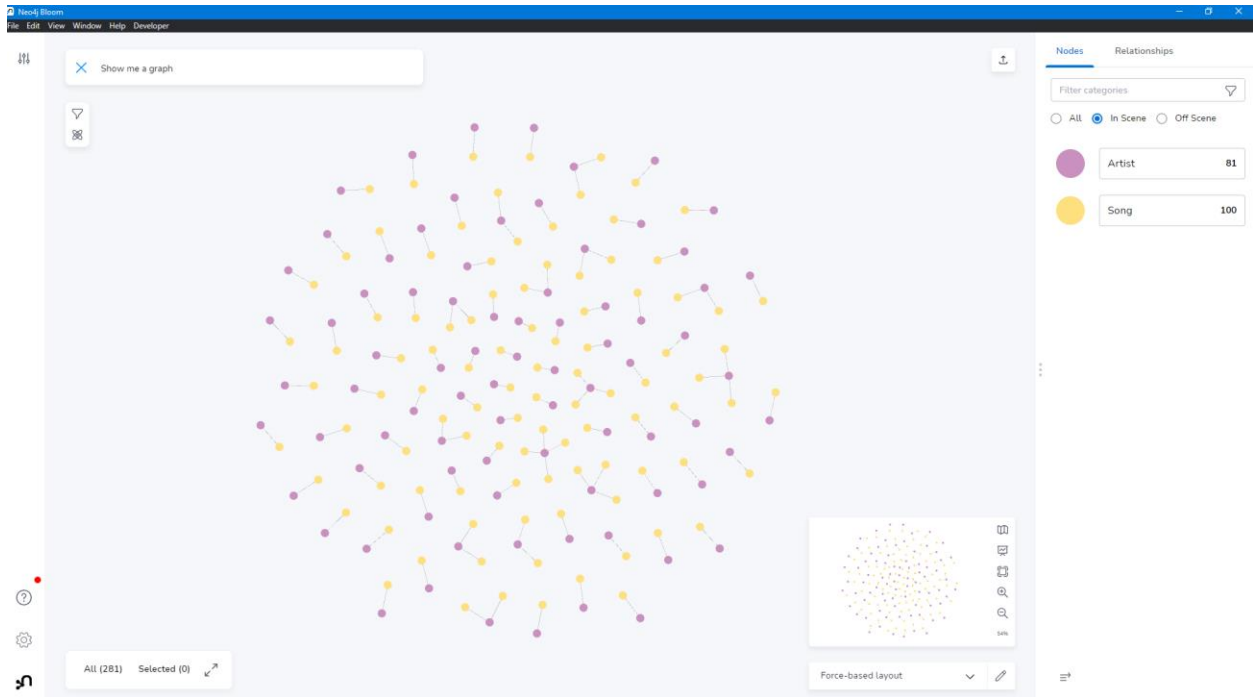


Figure 41: The home screen of Neo4j Bloom

Typing "show me a graph" in the search bar will return a sample of the data. In this case it returned 100 songs mapped to their artists. We can see that the total number of artists is less than the total number of songs meaning that some artist may have multiple songs. Refer below for a zoomed in example:

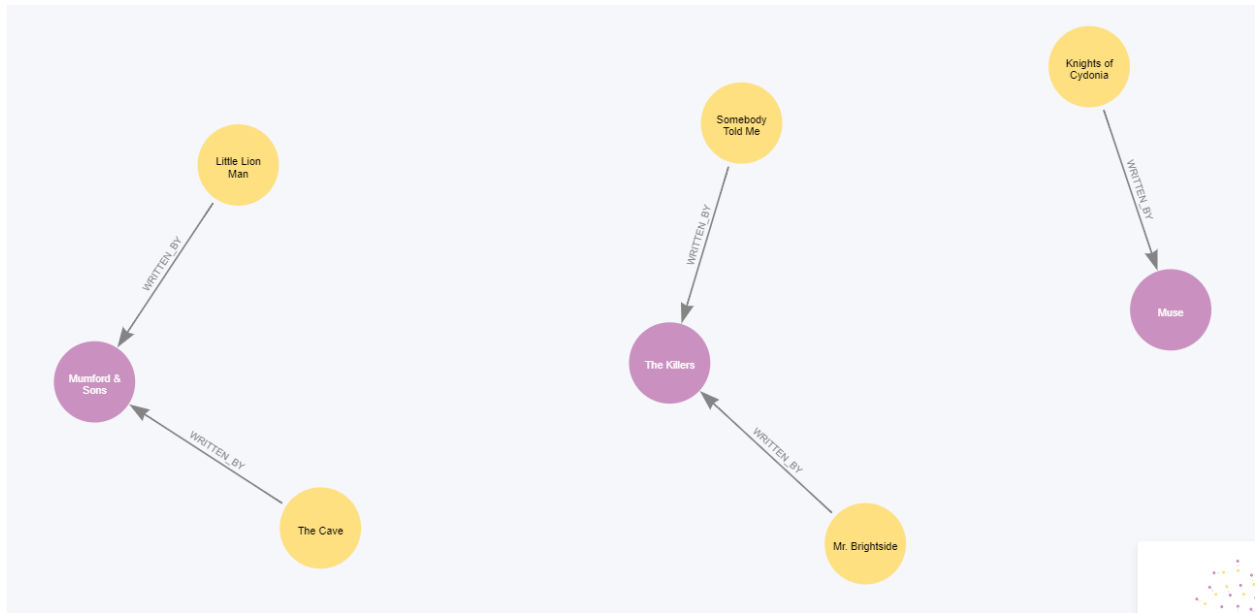


Figure 42: A no cypher query returned in Bloom

Here we can see various relationships. We can see the songs :

- “Little Lion man” and “The cave” written by Mumford and Sons
- “Mr. Brightside” and “Somebody told me” written by The killers
- “Knights of Cydonia” by Muse

Just from this simple textual query, we are able to see so much information related to Songs and artists without even having to write any Cypher. Lets see what happens when we write some cypher code in Bloom. I will show 3 cases in this report for conciseness. A full demo can be seen in the deliverables:

Using Bloom’s Search Phrase technology which allows users to map an English phrase (with dynamic user input) to an Cypher query, we will example the 3 cases below.

Case 1: Showing all songs written by an artist the user passes in:

Search Phrase: Show me all songs written by this artist: U2

Cypher query:

```
MATCH (s:Song) -[r:WRITTEN_BY]-> (a:Artist) where a.Artist_Name = $param
RETURN s,r,a
```

Note: \$param = U2

Result:

Case 2: Return the most popular song for an artist based on Popularity_Score

Search Phrase: Show me the most popular song from this artist: Eminem

Cypher Query:

```
MATCH (s:Song) -[r:WRITTEN_BY]-> (a:Artist) where a.Artist_Name = $param
```

```
RETURN s
```

```
ORDER BY s.Popularity_score DESC Limit 1
```

Note: \$param = Eminem

Result:

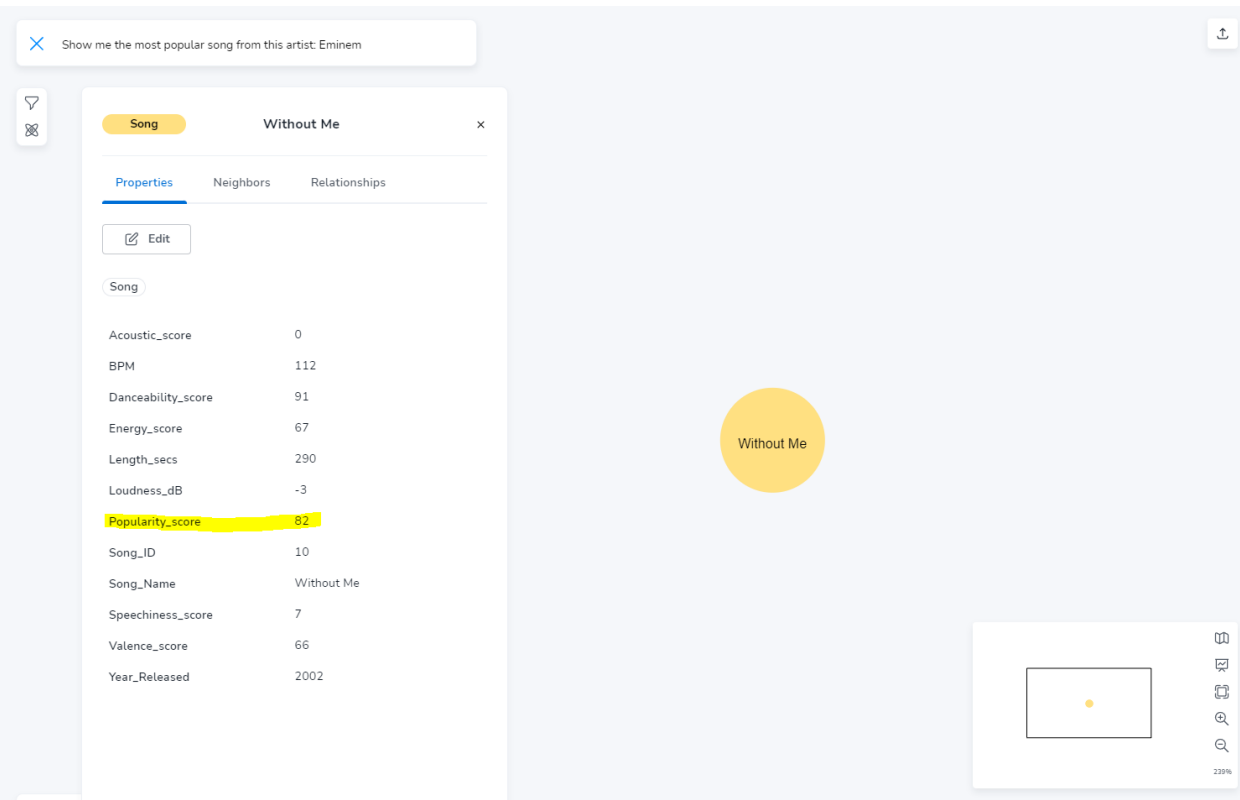


Figure 44: Query result 2 in Bloom

Neo4j is a popular graph database management system. It is known for its powerful querying capabilities and simple visualization options. The Cypher query language, which is used in Neo4j, has some advantages and disadvantages. One of the main advantages of Cypher is that it does not require the use of complex joins, which can make queries faster and more efficient. However, some people find the syntax for representing relationships in Cypher to be strange and difficult to understand.

If you are interested in exploring the capabilities of Neo4j and other data science tools, there are many resources available online that can help you get started. There are also tools and techniques available for converting an existing relational database management system (RDBMS) into a Neo4j database using extract, transform, and load (ETL) processes. These tools can help you take advantage of the power and flexibility of a graph database, while still leveraging your existing data and systems.

8 Resources

Some reading material used:

- Neo4j Product Overview: <https://neo4j.com/product/>
- Neo4j Getting Started: <https://neo4j.com/docs/getting-started/current/>
- Neo4j Bloom: <https://neo4j.com/product/bloom/?ref=product>
- Neo4j Graph Data Science: <https://neo4j.com/product/graph-data-science/>
- *Neo4j Developer Material: <https://neo4j.com/developer/get-started/>

* Note the majority of information and figures came from the development material.

Dataset used: <https://www.kaggle.com/datasets/paradisejoy/top-hits-spotify-from-20002019>