

1. Implementation Platform

Use of graphical representation for NoSQL, our team chose to utilize NEO4J with previous experience from A3. Our project topic is on the basis of Project 1, which was creating a Game database, it holds information of the games, its platform, genres and more key information about rating, price, summary, released date, etc..

2. Design Changes

As the project description suggested “you may remove IS-A relationships that you created in phase I”, our NoSQL database no longer incorporates from project 1 SQL design the IS A relationship that was referred to as **ActionGames**. Furthermore, we have according to the instructions removed weak entities such as **GameGenres** and **GamePlatforms** while maintaining their information on the basis of relationships.

3. Data Transfer and NoSQL Script

Steps for Data Transfer from SQL to NoSQL:

Step 1: Open Project 1 “**P1.sql**” on MySQL Workbench

Step 2: Run the database Create table script on MySQL

Step 3: Open up IDE to run Python Script, first run “RAWG.py” then run “IGDB.py” then finally run “main.py” all of this can be found from Project 1 submission.

Step 4: Now the database on MySQL should be set up and you should be able to see the data imported in each of the respective tables

Step 5: Now open up each entity table to see the data it stores, save each of the entity table as a CSV file.

Step 6: Open up NEO4J Database imports folder and transfer all the CSV files into it.

Step 7: Run the following Cypher Text on NEO4J:

This is the script for creating our relations and entities in terms of nodes and relationship among nodes in NEO4J:

// Create Game nodes

LOAD CSV WITH HEADERS FROM 'file:///Games.csv' AS row

```
CREATE (:Game {
  game_id: toInteger(row.game_id),
  nameOfGames: row.nameOfGames,
  released_date: date(row.released_date),
  rating: toFloat(row.rating),
  summary: row.summary
});
```

// Create Platform nodes

```
LOAD CSV WITH HEADERS FROM 'file:///Platforms.csv' AS row
CREATE (:Platform {
  platform_id: toInteger(row.platform_id),
  nameOfPlatform: row.nameOfPlatform
});
```

// Create Genre nodes

```
LOAD CSV WITH HEADERS FROM 'file:///Genres.csv' AS row
CREATE (:Genre {
  genre_id: toInteger(row.genre_id),
  nameOfGenres: row.nameOfGenres
});
```

// Create GamePlatforms relationships

```
LOAD CSV WITH HEADERS FROM 'file:///GamePlatforms.csv' AS row
MATCH (g:Game {game_id: toInteger(row.game_id)}), (p:Platform {platform_id:
toInteger(row.platform_id)})
CREATE (g)-[:AVAILABLE_ON]->(p);
```

// Create GameGenres relationships

```
LOAD CSV WITH HEADERS FROM 'file:///GameGenres.csv' AS row
MATCH (g:Game {game_id: toInteger(row.game_id)}), (gn:Genre {genre_id:
toInteger(row.genre_id)})
CREATE (g)-[:BELONGS_TO]->(gn);
```

Special note: Ensure the settings for NEO4J are as follows to handle a large dataset

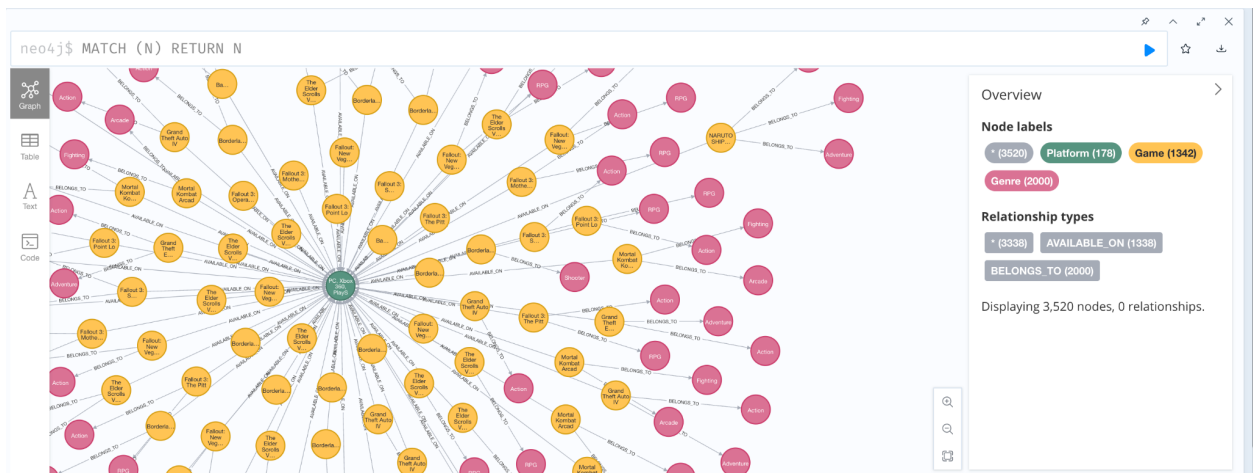
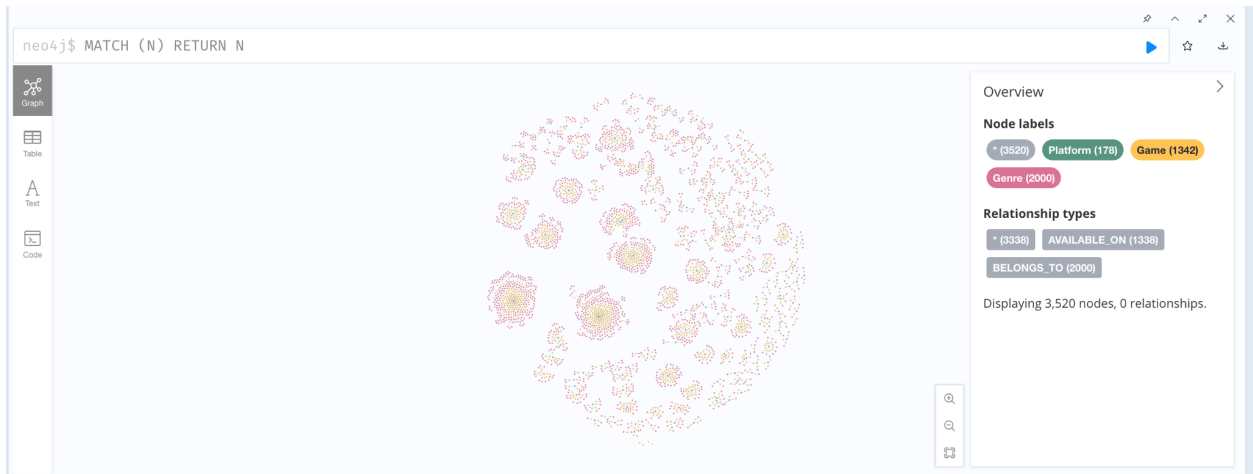
The image shows a settings panel for Neo4j Desktop. It contains several input fields and checkboxes. The 'Initial Node Display' field is set to 4000. The 'Max neighbours from vis interaction' field is set to 4000. The 'Result view max rows' field is set to 4000. The 'Max record fields' field is set to 4000. The 'Connect result nodes' checkbox is checked. The 'Show zoom interactions hint' checkbox is unchecked. Below these settings is a section titled 'Product Analytics' with a note: 'Product usage statistics are sent and crash reports are sent. These settings can be changed in Desktop.'

Setting	Value
Initial Node Display	4000
Max neighbours from vis interaction	4000
Result view max rows	4000
Max record fields	4000
Connect result nodes	<input checked="" type="checkbox"/>
Show zoom interactions hint	<input type="checkbox"/>

Product Analytics

Product usage statistics are sent and crash reports are sent. These settings can be changed in Desktop.

Result:

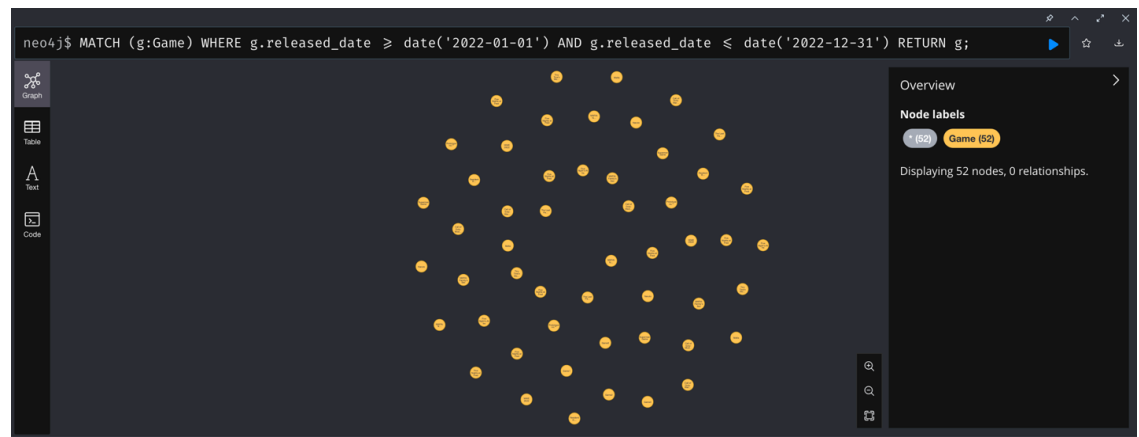


4. Query Implementation

Q1. Basic Search Query on an Attribute Value

- **Retrieve games released in a specific year:**
MATCH (g:Game)
WHERE g.released_date >= date('2022-01-01') AND g.released_date <= date('2022-12-31')
RETURN g;

Result Output:

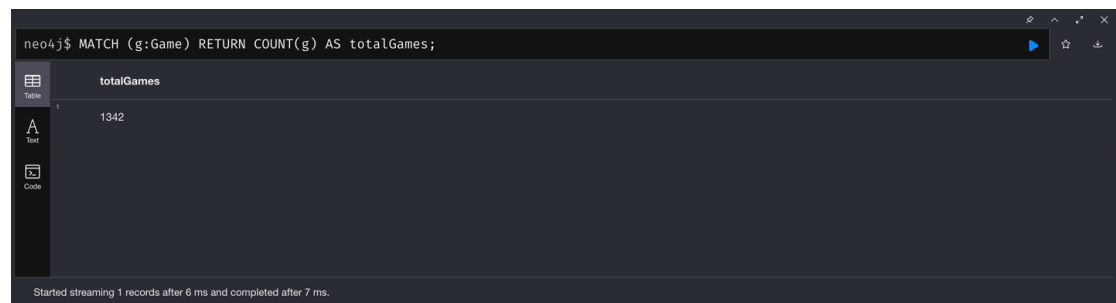


Q2. Aggregate Data Query

Query 1: (Without Criteria)

- **Count the total number of games in the database:**
MATCH (g:Game)
RETURN COUNT(g) AS totalGames;

Result Output:



Query 2: (With Criteria)

- Count the total number of games in the database that have a rating greater than 90:

```
MATCH (g:Game)
```

```
WHERE g.rating > 90
```

```
RETURN COUNT(g) AS numberOfGamesWithRatingGreaterThan90;
```

Result Output:



The screenshot shows the Neo4j query interface. The query entered is: `neo4j$ MATCH (g:Game) WHERE g.rating > 90 RETURN COUNT(g) AS numberOfGamesWithRatingGreaterThan90;`. The result is displayed in a table view with the title `numberOfGamesWithRatingGreaterThan90`. The table has one column and one row with the value 34. The status bar at the bottom indicates: "Started streaming 1 records after 7 ms and completed after 9 ms."

numberOfGamesWithRatingGreaterThan90
34

Q3. Find top n entities satisfying a criteria, sorted by an attribute:

- Retrieves top 5 games with a rating greater than 50, sorting them in descending order based on their ratings:

```
MATCH (g:Game)
```

```
WHERE g.rating > 50
```

```
RETURN g
```

```
ORDER BY g.rating DESC
```

```
LIMIT 5;
```

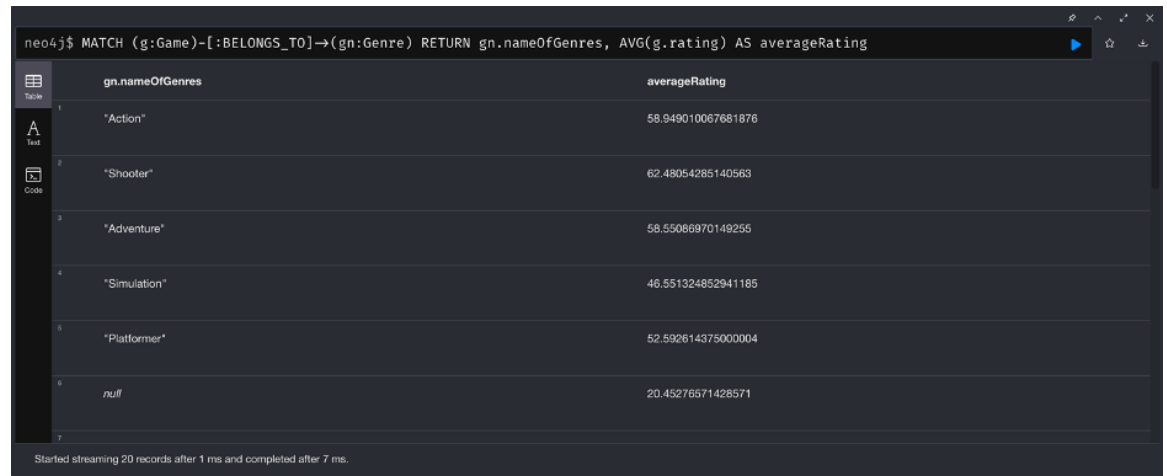
Result Output:



Q4. Simulate a Relational GROUP BY Query in NoSQL:

- **Calculate the average rating per genre:**
MATCH (g:Game)-[:BELONGS_TO]->(gn:Genre)
RETURN gn.nameOfGenres, AVG(g.rating) AS averageRating

Result Output:



```
neo4j$ MATCH (g:Game)-[:BELONGS_TO]->(gn:Genre) RETURN gn.nameOfGenres, AVG(g.rating) AS averageRating
```

	gn.nameOfGenres	averageRating
1	"Action"	58.949010067661876
2	"Shooter"	62.48054285140563
3	"Adventure"	58.55086970149255
4	"Simulation"	46.551324852941185
5	"Platformer"	52.592614375000004
6	null	20.45276571428571
7		

Started streaming 20 records after 1 ms and completed after 7 ms.

Q5. Build the appropriate indexes for previous queries, report the index creation statement and the query execution time before and after you create the index:

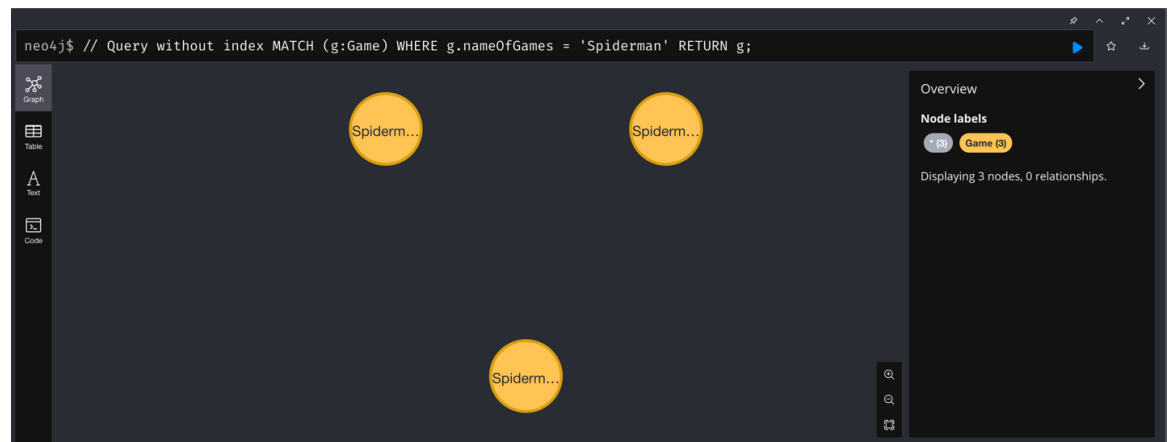
Query 1:

- **Demonstrate creating an index on the nameOfGames property for faster searches:**

// Query without index

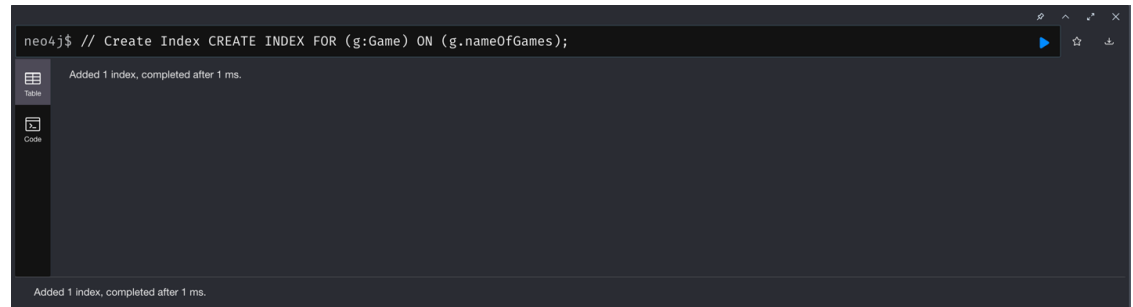
```
MATCH (g:Game)
WHERE g.nameOfGames = 'Spiderman'
RETURN g;
```

Result Output:



// Create Index

```
CREATE INDEX FOR (g:Game) ON (g.nameOfGames);
```



// Query with index

```
MATCH (g:Game)
USING INDEX g:Game(nameOfGames)
WHERE g.nameOfGames = 'Spiderman'
RETURN g;
```

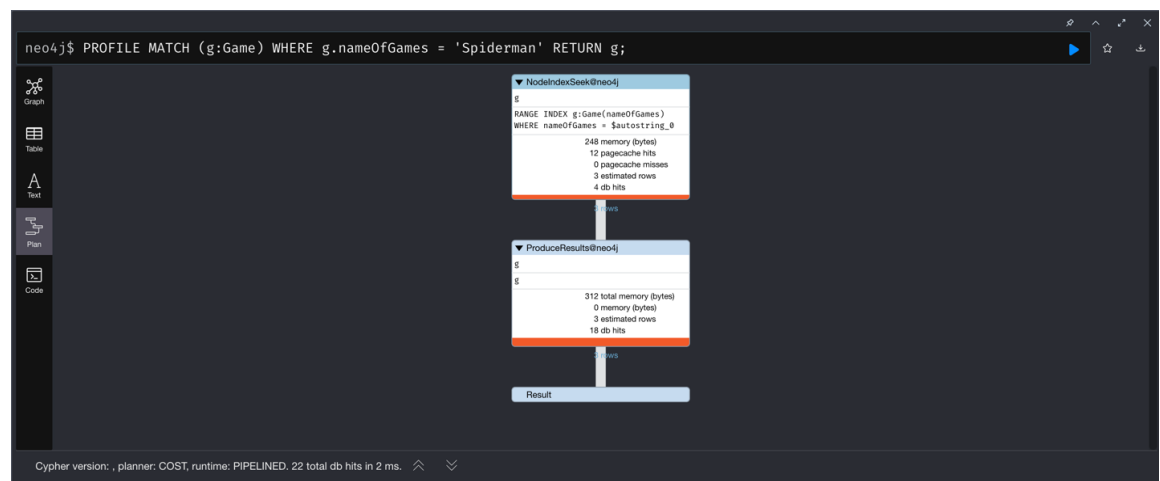
Result Output:



Using *Profile Match* to see performance difference between with and without index:

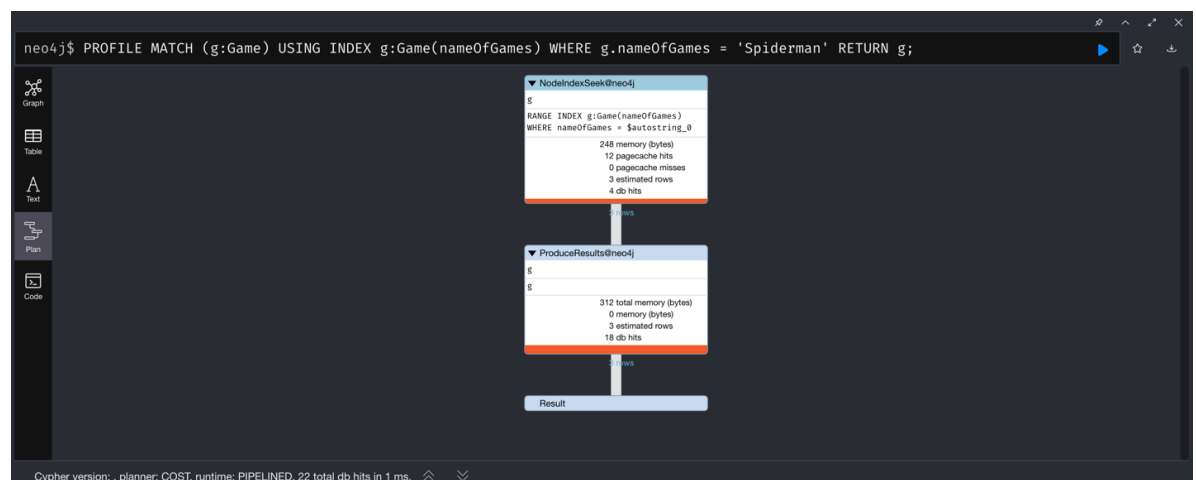
//Without Index

```
PROFILE MATCH (g:Game)
WHERE g.nameOfGames = 'Spiderman'
RETURN g;
```



//With Index

```
PROFILE MATCH (g:Game)
USING INDEX g:Game(nameOfGames)
WHERE g.nameOfGames = 'Spiderman'
RETURN g;
```



You can see that with the index the speed is slightly faster 1ms compared to 2ms without index, the speed would be drastically more significant depending on the complexity of query and number of instances.

Query 2 Based on Q1:

// Query without index

MATCH (g:Game)

WHERE g.released_date >= date('2022-01-01') AND g.released_date <=
date('2022-12-31')

RETURN g;

```
neo4j$ MATCH (g:Game) WHERE g.released_date >= date('2022-01-01') AND g.released_date <= date('2022-12-31') RETURN g;
```



The screenshot shows the Neo4j query interface with the query: `neo4j$ MATCH (g:Game) WHERE g.released_date >= date('2022-01-01') AND g.released_date <= date('2022-12-31') RETURN g;`. The results are displayed in a table view with 3 columns. The first column is the index, the second is the JSON representation of the game object, and the third is the element ID. The first record (index 1) is for game ID 0, with a summary of 'Awesome action game', a rating of 85.5, and a release date of '2022-01-01'. The second record (index 2) is for game ID 202, with a summary of 'Mafia: Family's Secret is a top-down shooting game. Kill your enemies, go through all the missions and discover the secrets of Guidice mafia family.', a rating of 0.0, and a release date of '2022-01-08'. The third record (index 3) is an empty object. The status bar at the bottom indicates 'Started streaming 52 records after 9 ms and completed after 12 ms.'

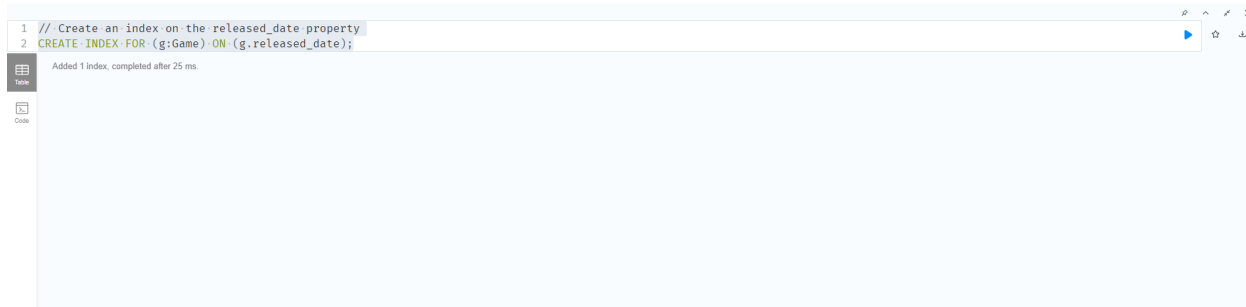
1	{ "identity": 0, "labels": ["Game"], "properties": { "summary": "Awesome action game", "rating": 85.5, "nameOfGames": "Game1", "released_date": "2022-01-01", "game_id": 1 }, "elementId": "0" }	0
2	{ "identity": 202, "labels": ["Game"], "properties": { "summary": "Mafia: Family's Secret is a top-down shooting game. Kill your enemies, go through all the missions and discover the secrets of Guidice mafia family.", "rating": 0.0, "nameOfGames": "MAFIA: Family's Secret", "released_date": "2022-01-08", "game_id": 203 }, "elementId": "202" }	202
3	{	

Started streaming 52 records after 9 ms and completed after 12 ms.

// Create index

CREATE INDEX FOR (g:Game) ON (g.released_date);

```
1 // Create an index on the released_date property
2 CREATE INDEX FOR (g:Game) ON (g.released_date);
```



The screenshot shows the Neo4j query interface with the query: `1 // Create an index on the released_date property 2 CREATE INDEX FOR (g:Game) ON (g.released_date);`. The results are displayed in a table view with 3 columns. The first column is the index, the second is the JSON representation of the game object, and the third is the element ID. The first record (index 1) is for game ID 0, with a summary of 'Awesome action game', a rating of 85.5, and a release date of '2022-01-01'. The second record (index 2) is for game ID 202, with a summary of 'Mafia: Family's Secret is a top-down shooting game. Kill your enemies, go through all the missions and discover the secrets of Guidice mafia family.', a rating of 0.0, and a release date of '2022-01-08'. The third record (index 3) is an empty object. The status bar at the bottom indicates 'Added 1 index, completed after 25 ms.'

1	{ "identity": 0, "labels": ["Game"], "properties": { "summary": "Awesome action game", "rating": 85.5, "nameOfGames": "Game1", "released_date": "2022-01-01", "game_id": 1 }, "elementId": "0" }	0
2	{ "identity": 202, "labels": ["Game"], "properties": { "summary": "Mafia: Family's Secret is a top-down shooting game. Kill your enemies, go through all the missions and discover the secrets of Guidice mafia family.", "rating": 0.0, "nameOfGames": "MAFIA: Family's Secret", "released_date": "2022-01-08", "game_id": 203 }, "elementId": "202" }	202
3	{	

Added 1 index, completed after 25 ms.

// Query with index

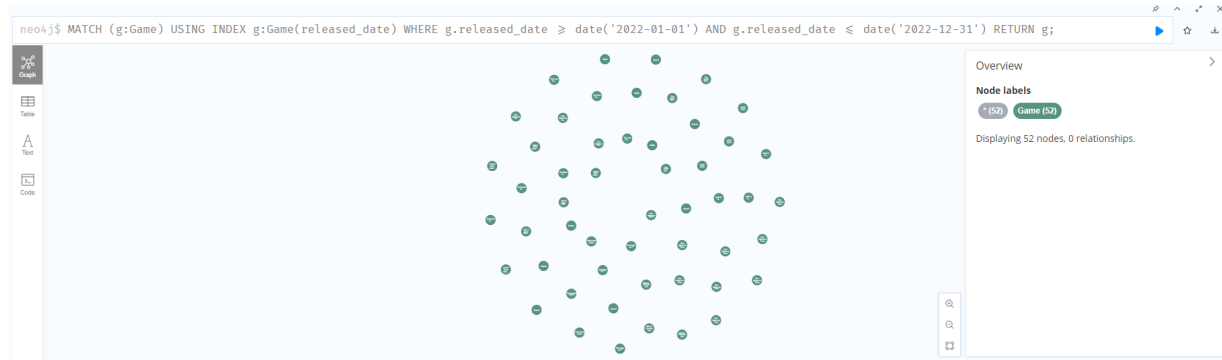
```
MATCH (g:Game)
```

```
USING INDEX g:Game(released_date)
```

```
WHERE g.released_date >= date('2022-01-01') AND g.released_date <=
```

```
date('2022-12-31')
```

```
RETURN g;
```



Using *Profile Match* to see performance difference between with and without index:

//Without index



// With index

Graph

Table

Test

Plan

Code

neo4j\$ PROFILE MATCH (g:Game) USING INDEX g:Game(released_date) WHERE g.released_date >= date('2022-01-01') AND g.released_date <= date('2022-12-31') RETURN g;

▼ NodeIndexScan[Range@neo4j]

E

RANGE INDEX g:Game(released_date)
WHERE released_date >
RuntimeConstant(Variable(anon.0),
ResolveFunctionInvocation(date,
Some(dateInput = "DEFAULT",
TEMPORAL_ARGUMENT :: ANY?) :: DATE
?),VectorAutoExtractedParameter(
autoextracting_0.any_0MemorySize)))
AND released_date <=
RuntimeConstant(Variable(anon.1),
ResolveFunctionInvocation(date,
Some(dateInput = "DEFAULT",
TEMPORAL_ARGUMENT :: ANY?) :: DATE
?),VectorAutoExtractedParameter(
autoextracting_1.any_0MemorySize)))

240 memory (bytes)
130 pages/row hit
0 pages/row miss
23 estimated rows
83.00 ms

▼ ProduceResults[neo4j]

E

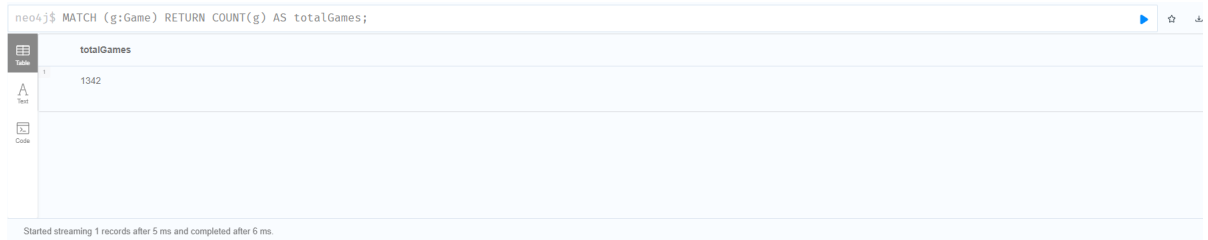
E

Cypher version: , planner: COST, runtime: PIPELINED, 365 total db hits in 14 ms

Query 3 Based on Q2:

1. Query (Without Criteria):

// Query without index



The screenshot shows the Neo4j query interface. The query entered is `neo4j$ MATCH (g:Game) RETURN COUNT(g) AS totalGames;`. The result is displayed in a table with one column, `totalGames`, and one row with the value `1342`. The interface includes a sidebar with icons for Table, Text, and Code, and a status bar at the bottom indicating the query execution time.

totalGames
1342

// Create index

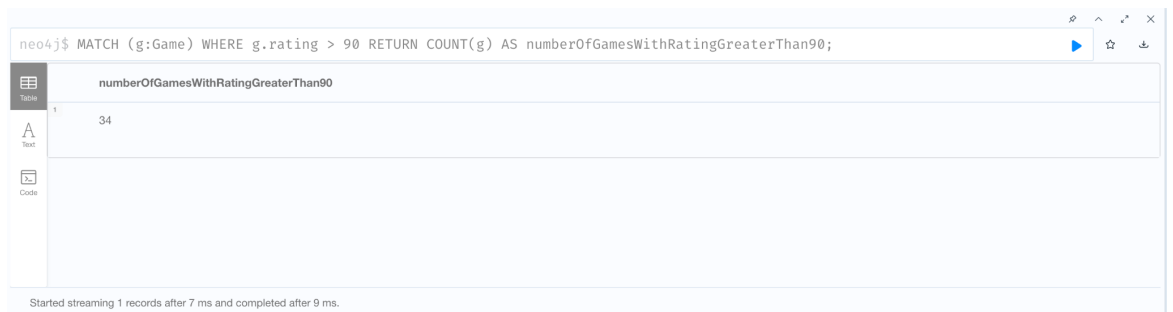
NOTE: In this case, creating an index won't have a significant impact on the performance of this specific query because it doesn't involve filtering based on a specific property. Indexes are generally useful for improving the performance of queries that involve filtering or sorting based on specific properties. In your current query, you are counting all nodes with the label `Game`, and indexing won't change the way the count is calculated.

2. Query (With Criteria):

Count the total number of games in the database that have a rating greater than 90:

// Query without index

```
MATCH (g:Game)
WHERE g.rating > 90
RETURN COUNT(g) AS numberOfGamesWithRatingGreaterThan90;
```

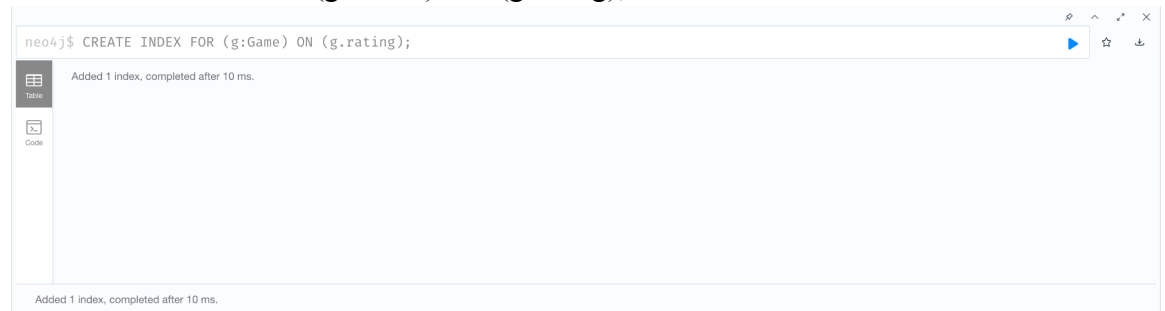


The screenshot shows the Neo4j query interface. The query entered is `neo4j$ MATCH (g:Game) WHERE g.rating > 90 RETURN COUNT(g) AS numberOfGamesWithRatingGreaterThan90;`. The result is displayed in a table with one column, `numberOfGamesWithRatingGreaterThan90`, and one row with the value `34`. The interface includes a sidebar with icons for Table, Text, and Code, and a status bar at the bottom indicating the query execution time.

numberOfGamesWithRatingGreaterThan90
34

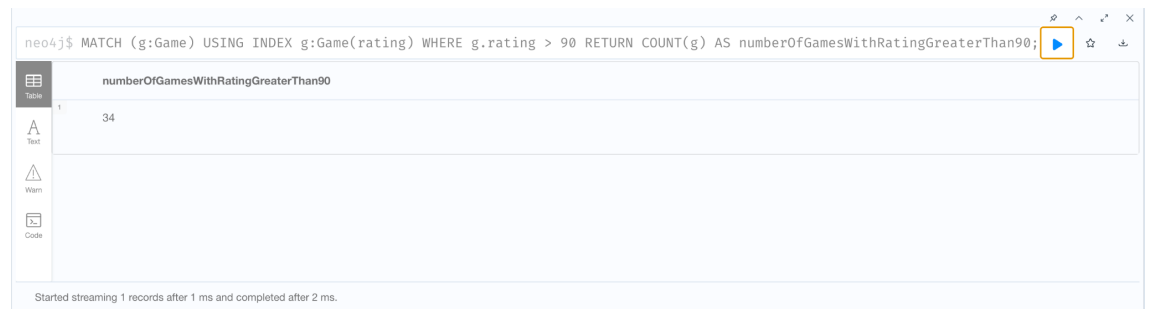
// Create index

```
CREATE INDEX FOR (g:Game) ON (g.rating);
```



// Query with index

```
MATCH (g:Game)
USING INDEX g:Game(rating)
WHERE g.rating > 90
RETURN COUNT(g) AS numberOfGamesWithRatingGreaterThan90;
```



As you can see there is a difference of 2ms vs 9ms.

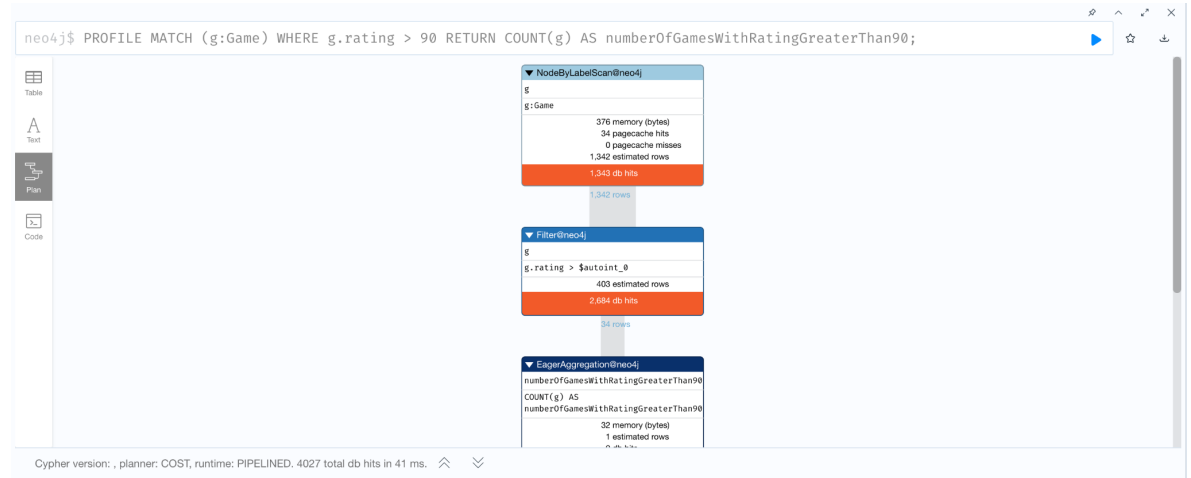
Using Profile Match to see performance difference between with and without index:

//Without Index:

PROFILE MATCH (g:Game)

WHERE g.rating > 90

RETURN COUNT(g) AS numberOfGamesWithRatingGreaterThan90;



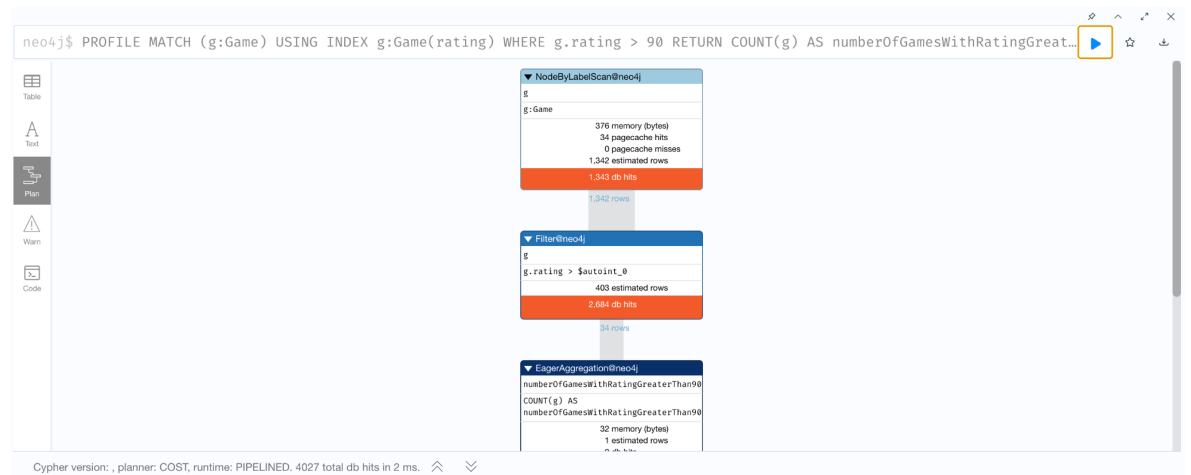
//With Index:

PROFILE MATCH (g:Game)

USING INDEX g:Game(rating)

WHERE g.rating > 90

RETURN COUNT(g) AS numberOfGamesWithRatingGreaterThan90;



Query 4 Based on Q3:

// Query without index

Retrieves top 5 games with a rating greater than 50, sorting them in descending order based on their ratings:

```
MATCH (g:Game)
WHERE g.rating > 50
RETURN g
ORDER BY g.rating DESC
LIMIT 5;
```



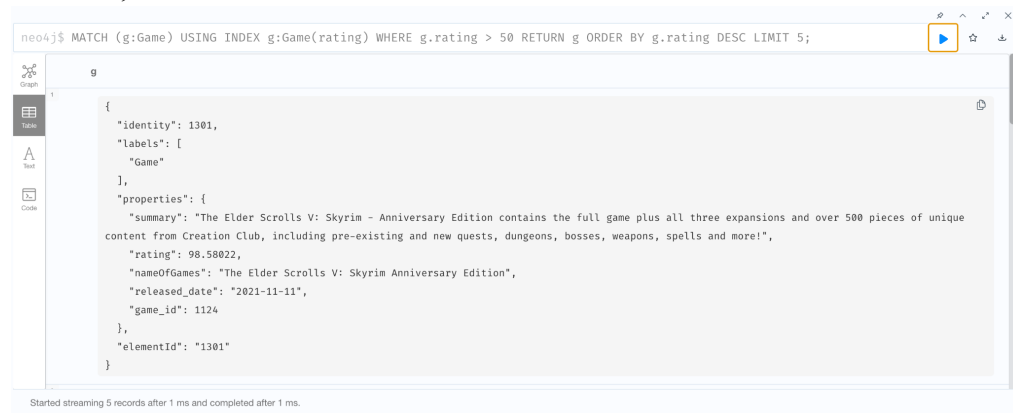
The screenshot shows the Neo4j query interface. The query entered is: `neo4j$ MATCH (g:Game) WHERE g.rating > 50 RETURN g ORDER BY g.rating DESC LIMIT 5;`. The results are displayed in a table view, showing a single record for a game. The record is a JSON object with the following fields: `"identity": 1301`, `"labels": ["Game"]`, and `"properties": {"summary": "The Elder Scrolls V: Skyrim - Anniversary Edition contains the full game plus all three expansions and over 500 pieces of unique content from Creation Club, including pre-existing and new quests, dungeons, bosses, weapons, spells and more!", "rating": 98.58022, "nameOfGames": "The Elder Scrolls V: Skyrim Anniversary Edition", "released_date": "2021-11-11", "game_id": 1124}`. The status bar at the bottom indicates: "Started streaming 5 records after 5 ms and completed after 13 ms."

// Create Index

CREATE INDEX FOR (g:Game) ON (g.rating); (*Already Previously Created*)

// Query with index

```
MATCH (g:Game)
USING INDEX g:Game(rating)
WHERE g.rating > 50
RETURN g
ORDER BY g.rating DESC
LIMIT 5;
```

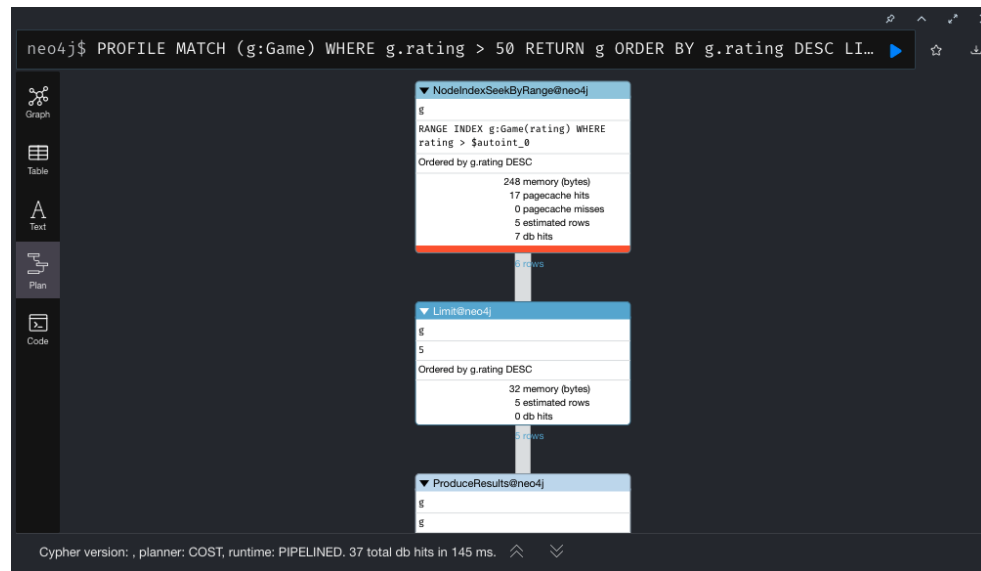


The screenshot shows the Neo4j query interface. The query entered is: `neo4j$ MATCH (g:Game) USING INDEX g:Game(rating) WHERE g.rating > 50 RETURN g ORDER BY g.rating DESC LIMIT 5;`. The results are displayed in a table view, showing a single record for a game. The record is a JSON object with the following fields: `"identity": 1301`, `"labels": ["Game"]`, and `"properties": {"summary": "The Elder Scrolls V: Skyrim - Anniversary Edition contains the full game plus all three expansions and over 500 pieces of unique content from Creation Club, including pre-existing and new quests, dungeons, bosses, weapons, spells and more!", "rating": 98.58022, "nameOfGames": "The Elder Scrolls V: Skyrim Anniversary Edition", "released_date": "2021-11-11", "game_id": 1124}`. The status bar at the bottom indicates: "Started streaming 5 records after 1 ms and completed after 1 ms."

Using Profile Match to see performance difference between with and without index:

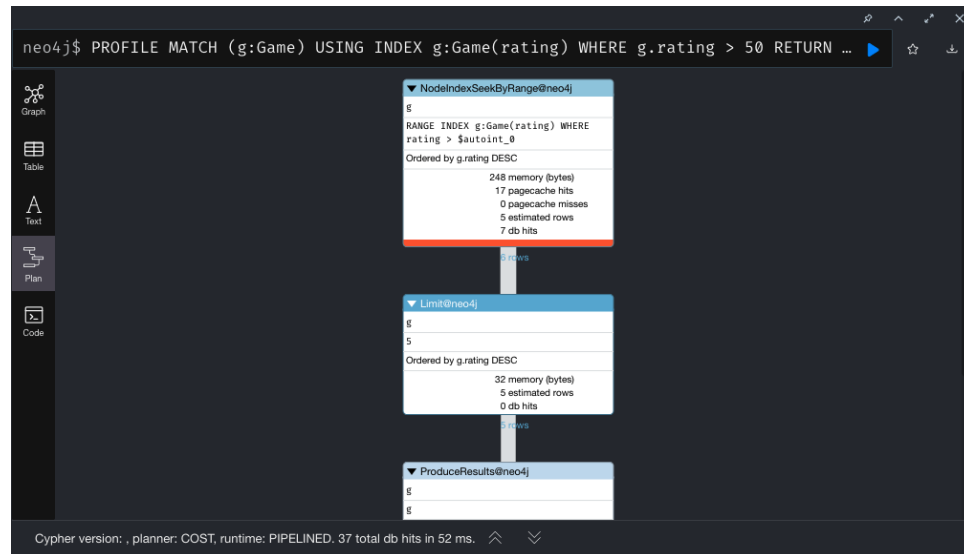
//Query without index

```
PROFILE
MATCH (g:Game)
WHERE g.rating > 50
RETURN g
ORDER BY g.rating DESC
LIMIT 5;
```



//Query with index

```
PROFILE
MATCH (g:Game)
USING INDEX g:Game(rating)
WHERE g.rating > 50
RETURN g
ORDER BY g.rating DESC
LIMIT 5;
```



As displayed the results with index vs without index vary greatly, having time of 52ms vs 145ms.

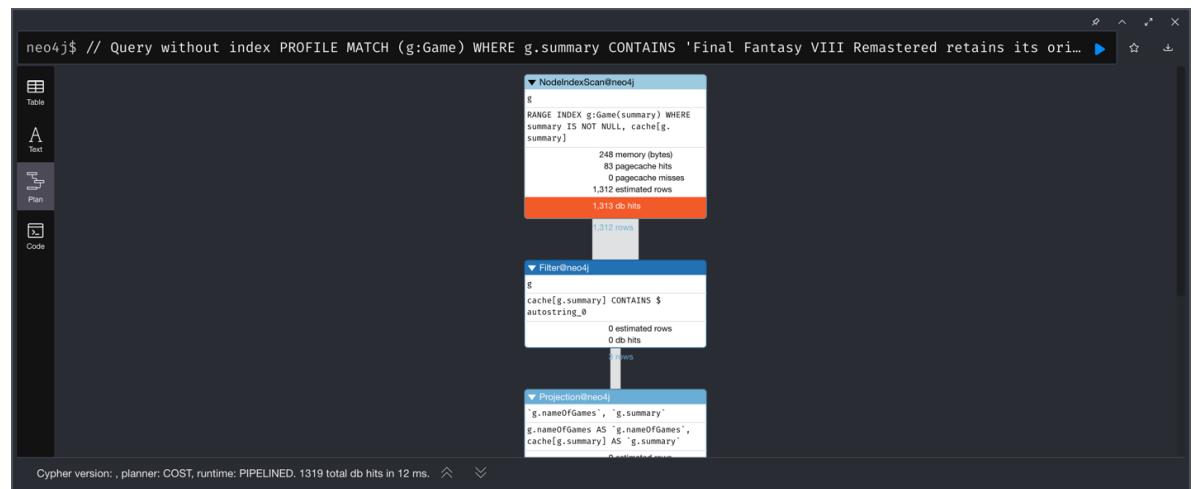
Q6. Demonstrate a full text search. Show the performance improvement by using indexes:

//Query without Index

PROFILE MATCH (g:Game)

WHERE g.summary CONTAINS 'Final Fantasy VIII Remastered retains its original 4:3 aspect ratio in both FMVs and real-time graphic rendering, but enhance visuals with several characters, enemies, GF, and objects refined to look better. The music is unchanged from the original PlayStation version.'

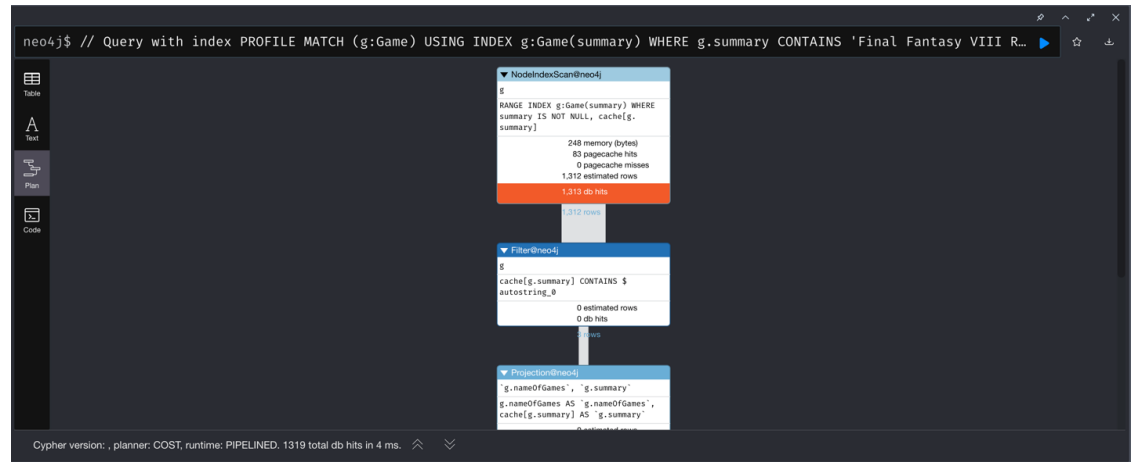
RETURN g.nameOfGames, g.summary;



// Create Index

CREATE INDEX FOR (g:Game) ON (g.summary);

```
// Query with index
PROFILE MATCH (g:Game)
USING INDEX g:Game(summary)
WHERE g.summary CONTAINS 'Final Fantasy VIII Remastered retains its
original 4:3 aspect ratio in both FMVs and real-time graphic rendering, but
enhance visuals with several characters, enemies, GF, and objects refined to look
better. The music is unchanged from the original PlayStation version.'
RETURN g.nameOfGames, g.summary;
```



5. Challenges

In addressing the challenges of Phase II, our initial hurdle was selecting a suitable platform, and we opted for Neo4j based on our team's familiarity with its features based on A3. Leveraging Neo4j's visualization tools became essential for comprehending the graph data structure efficiently. Migrating data from a relational model posed its own interesting task, and we chose a methodical approach by transferring data through CSV files to ensure a smooth integration process while maintaining data integrity.

Next was creating queries that was a considerable task for our team, given our limited prior experience. This required adapting to the unique syntax and structure of NoSQL queries, expanding our expertise in database query languages. Creating indexes in Neo4j emerged as a pivotal challenge in optimizing query efficiency. The process involved not only establishing the necessary indexes but also discerning the most efficient ways to analyze query performance, both with and without indexes. Balancing these considerations showcased our team's ability to fine-tune the Neo4j database for optimal speed and responsiveness, ultimately overcoming the challenges posed by the transition and further strengthening our proficiency in diverse database technologies.

6. Conclusion

In conclusion, Phase I of our video game database project marked a significant achievement as we successfully implemented a robust relational database by seamlessly integrating data from RAWG and IGDB APIs. Overcoming challenges such as API integration, rate limits, and data merging, our Python script efficiently populated a MySQL database, establishing consistent relationships and enforcing essential constraints. As we transitioned to Phase II and embraced the Neo4j NoSQL platform, we navigated new challenges, exploring innovative ways to integrate data from a relational database into a graph database. This tested our technical skills but also provided a unique opportunity to delve deep into the intricate world of diverse databases, showcasing our adaptability and problem-solving skills in the dynamic field of database development.