Overview of the System:

1. Data Sources:

   - The system utilizes two external APIs, namely RAWG API and IGDB API, to fetch information about games.
   - The RAWG API is used to obtain general information about games, such as name, platforms, release date, and genre.
   - The IGDB API is used to retrieve additional data, including ratings and summaries.

2. Data Processing:

   - The system consists of three main Python scripts: ***Rawg.py***, ***IGDB.py***, and ***main.py***.
   - ***Rawg.py*** fetches game data from the RAWG API and stores it in a JSON file (***games_data.json***).
   - ***IGDB.py*** uses the obtained RAWG data to make requests to the IGDB API, fetches additional data, and merges it with the RAWG data. The merged data is then stored in another JSON file (***merged_data.json***).
   - ***main.py*** reads the merged data and inserts it into a MySQL database.

3. Database Design:

   - The MySQL database is named **GAMES** and consists of several tables: **Games**, **ActionGames**, **Platforms**, **Genres**, **GamePlatforms**, and **GameGenres**.
   - Views (**ActionGamesView** and **GameDetailsView**) provide different perspectives on the game data.
   - The database also includes triggers to enforce constraints, such as not allowing the insertion of games with a release date more than two years in the future.

4. Queries and Views:

   - The system includes various types of SQL queries, including basic selects, group by, joins, set operations, correlated subqueries, and views.
   - Views, such as ActionGamesView and GameDetailsView, simplify the retrieval of specific information from the database.

5. Constraints:

- The database includes constraints to ensure data integrity, such as primary key constraints, foreign key relationships, and custom triggers for specific conditions.

Data Model: Using MySQL

The data model involves several entities, including:

Games: Basic information about each game.
Genres: Different genres that games can belong to.
Platforms: Various gaming platforms.
GameGenres: Associative table linking games to genres.
GamePlatforms: Associative table linking games to platforms.
ActionGames: A specialized table indicating games classified as "Action."
Views: Provide simplified and specific perspectives on the data.

Approach and Challenges:

1. **API Integration:**

- Fetching data from two different APIs (RAWG and IGDB) required coordinating requests and merging responses.
- Handling rate limits and potential API changes was a consideration.

2. **API Authentication and Authorization:**

- Managing authentication tokens, API keys, and OAuth tokens posed challenges during the interaction with external APIs.
- Understanding and correctly implementing the authentication and authorization processes for both RAWG and IGDB APIs was crucial for secure API interactions.
- As first-time users, overcoming challenges related to API authentication, authorization, and token management required additional effort and learning.

3. **Data Merging:**

    - Merging data from two different sources required careful mapping and handling of potential missing or conflicting information.

4. **Database Population:**

    - The Python script main.py reads the merged data and inserts it into the MySQL database.
    - Handling relationships and ensuring data consistency posed challenges, especially with the insertion of genres, platforms, and associated tables.

5. **Database Design and Constraints:**

    - Designing an effective database structure to represent the relationships between games, genres, and platforms.
    - Implementing constraints, such as the trigger to prevent inserting games with release dates more than two years in the future.

6. **Data Integrity:**

    - Ensuring data integrity through proper use of foreign keys, unique constraints, and avoiding duplicate entries.

The process of populating the game database faced challenges related to data migration, validation, handling duplicate entries, and managing API authentication. These challenges provided valuable learning opportunities, emphasizing the importance of robust data management practices, clear documentation, and continuous training for the development team. By addressing these issues, the system can maintain data accuracy, integrity, and security throughout its lifecycle.

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
});
```

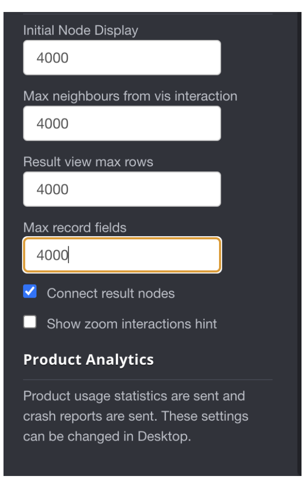
// Create Platform nodes
LOAD CSV WITH HEADERS FROM 'file:///Platforms.csv' AS row
CREATE (:Platform {
    platform_id: toInteger(row.platform_id),
    nameOfPlatform: row.nameOfPlatform
});

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

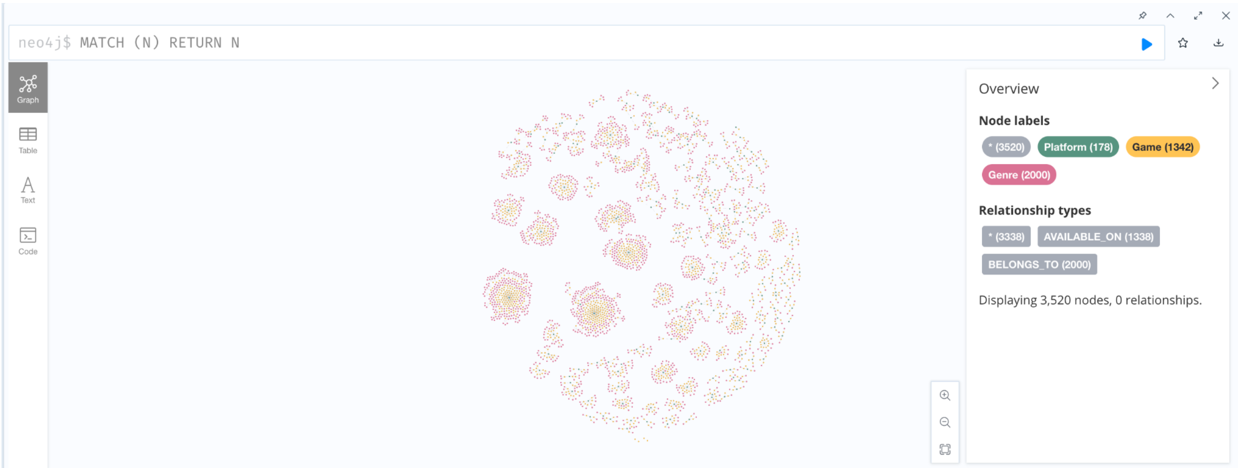
// Create GamePlatforms relationships
LOAD CSV WITH HEADERS FROM 'file:///GamePlatforms.csv' AS row
MATCH (g:Game {game_id: toInteger(row.game_id)}), (p:Platform {platform_id: toInteger(row.platform_id)})
CREATE (g)-[:AVAILABLE_ON]->(p);

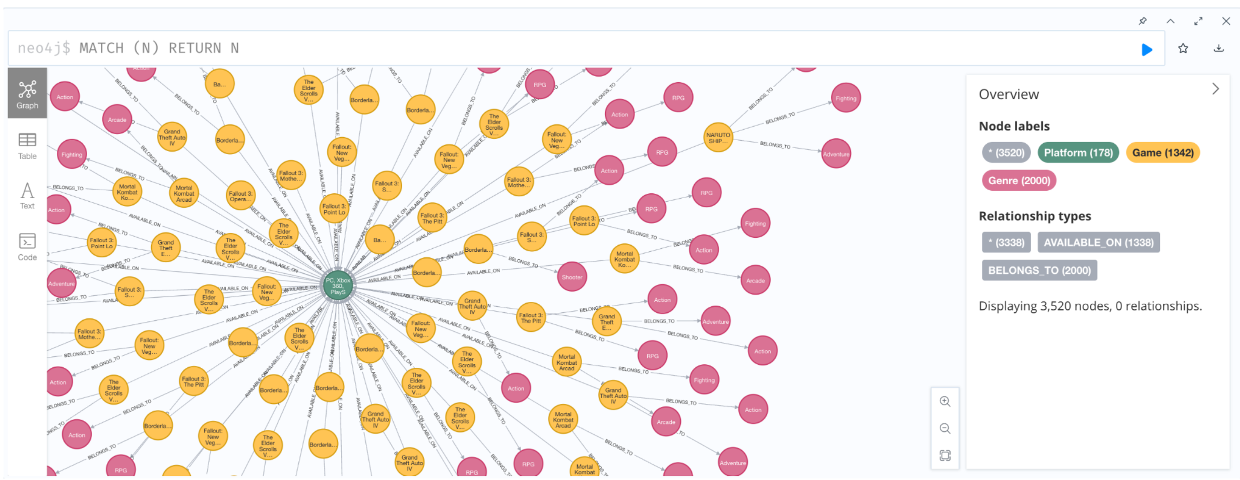
// Create GameGenres relationships
LOAD CSV WITH HEADERS FROM 'file:///GameGenres.csv' AS row
MATCH (g:Game {game_id: toInteger(row.game_id)}), (gn:Genre {genre_id: toInteger(row.genre_id)})
CREATE (g)-[:BELONGS_TO]->(gn);

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



Result:


neo4j$ MATCH (N) RETURN N
Graph
Table
Text
Code
Overview
Node labels
* (3520)
Platform (178)
Game (1342)
Genre (2000)
Relationship types
* (3338)
AVAILABLE_ON (1338)
BELONGS_TO (2000)
Displaying 3,520 nodes, 0 relationships.

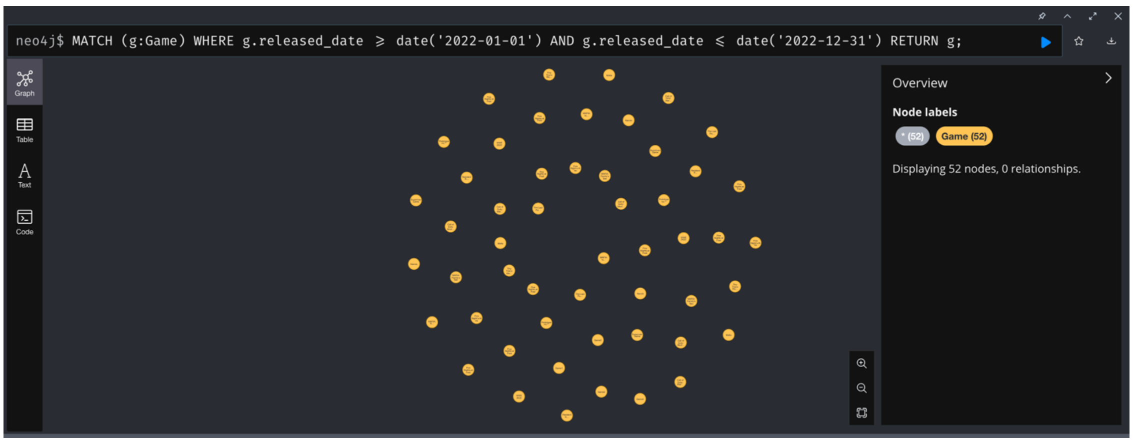

neo4j$ MATCH (N) RETURN N
Graph
Table
Text
Code
Overview
Node labels
* (3520)
Platform (178)
Game (1342)
Genre (2000)
Relationship types
* (3338)
AVAILABLE_ON (1338)
BELONGS_TO (2000)
Displaying 3,520 nodes, 0 relationships.

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:

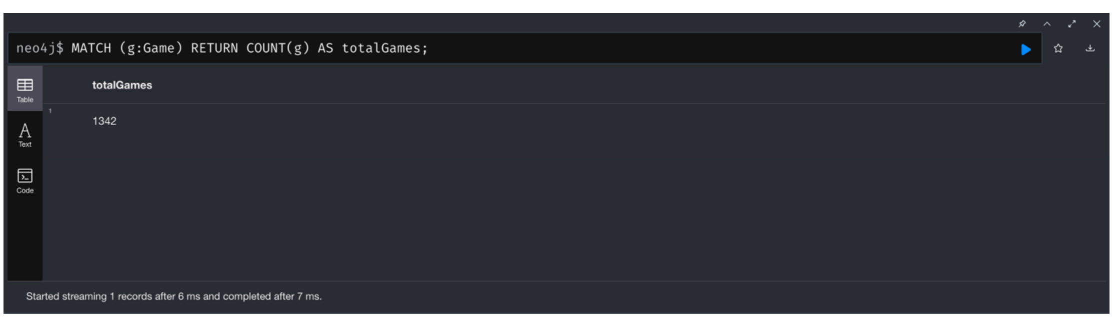


Q2. Aggregate Data Query

Query 1: (Without Criteria)

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

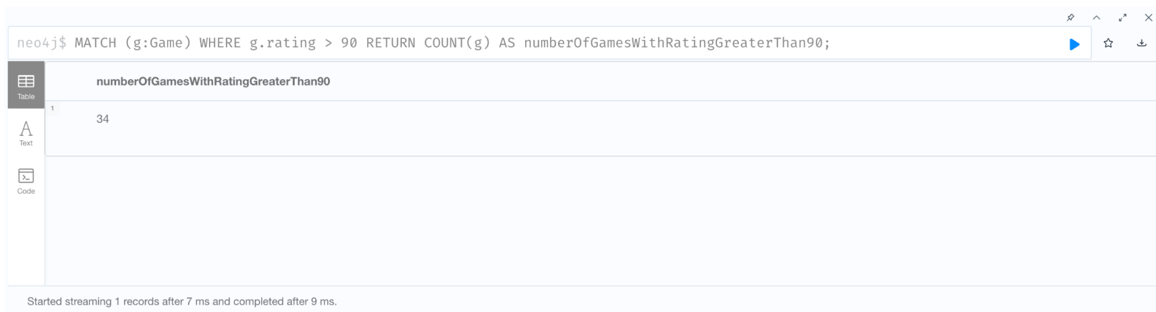
  Result Output:



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:



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

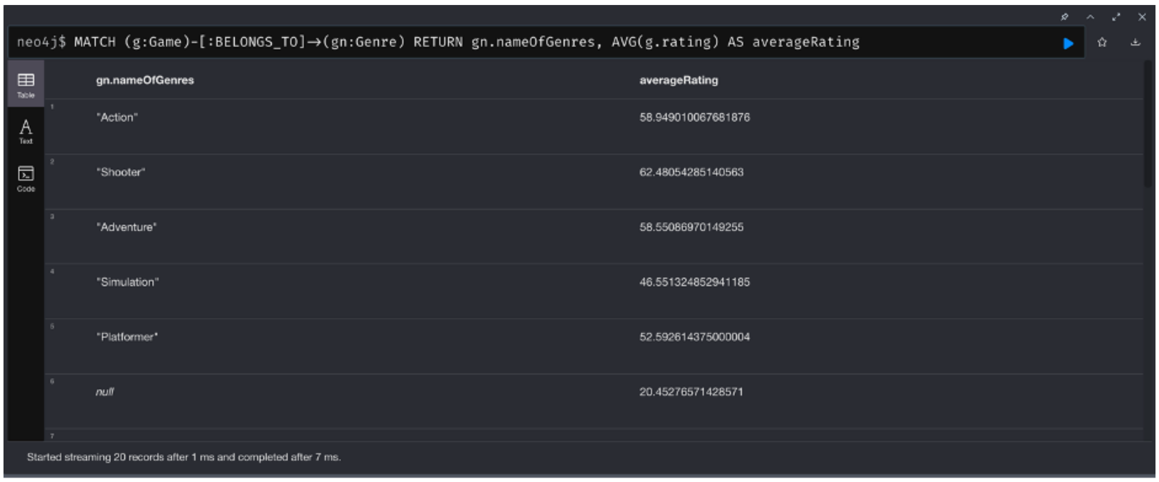
- **Retrieves top 5 games with a rating greater than 50, sorting them in descending order based on their ratings:**
  MATCH (g:Game)
  WHERE g.rating > 50
  RETURN g
  ORDER BY g.rating DESC
  LIMIT 5;

  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"	58.949010067681876
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:

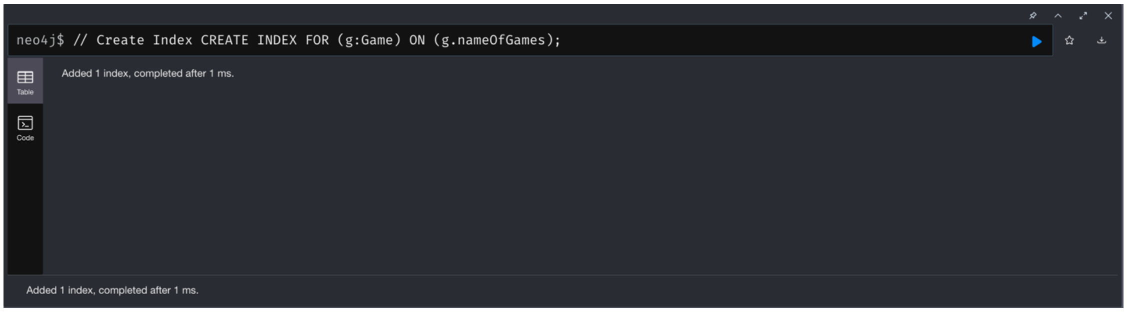
Query 1:

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

  // Query without index
  MATCH (g:Game)
  WHERE g.nameOfGames = 'Spiderman'
  RETURN g;

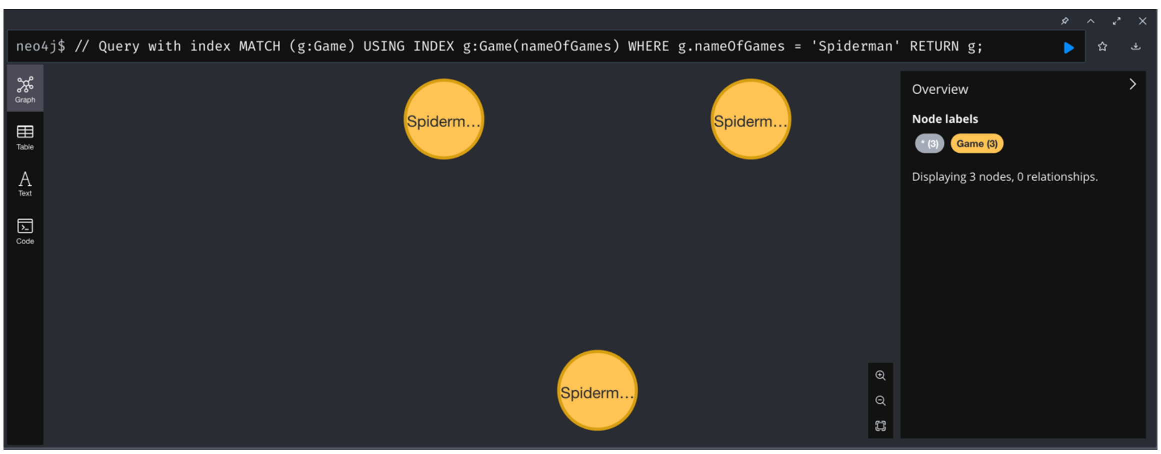
  Result Output:

// Create Index
CREATE INDEX FOR (g:Game) ON (g.nameOfGames);



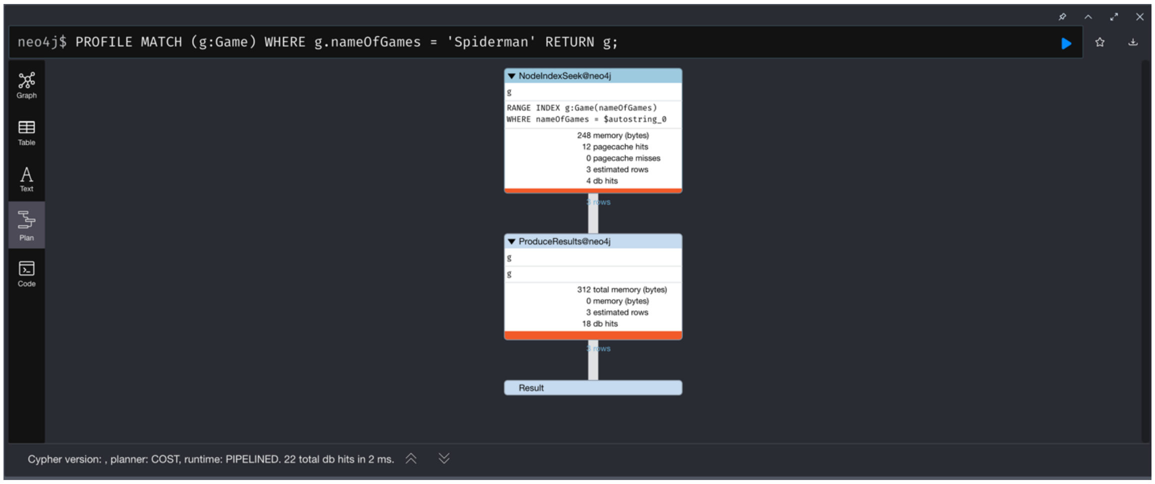
// Query with index
MATCH (g:Game)
USING INDEX g:Game(nameOfGames)
WHERE g.nameOfGames = 'Spiderman'
RETURN g;

Result Output:

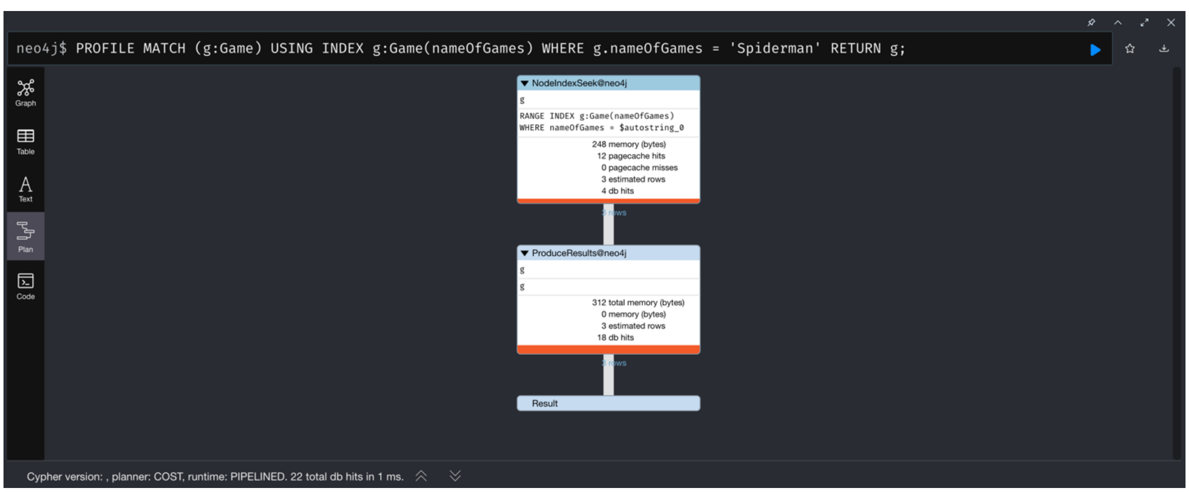


Using *Profile Match* to see performance difference between with and without index:

//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;

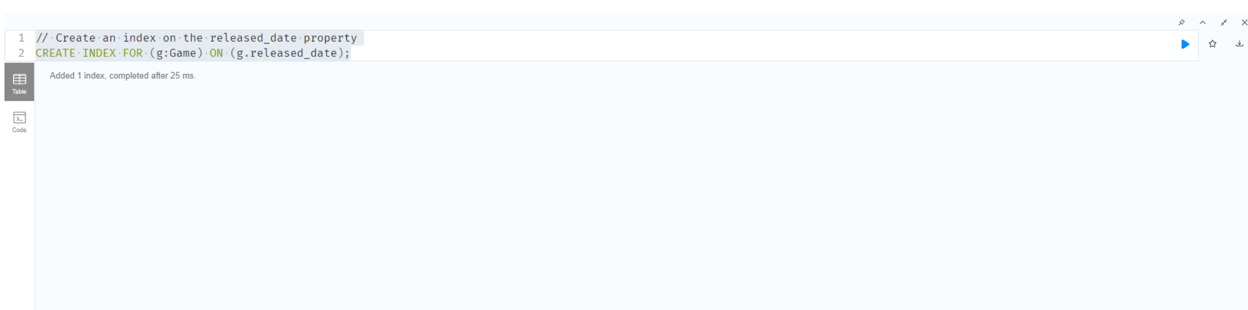


You can see that with the index the speed is slightly faster 1ms compared to 2ms without index, the speed would be drastically more significant depending on the complexity of query and number of instances.

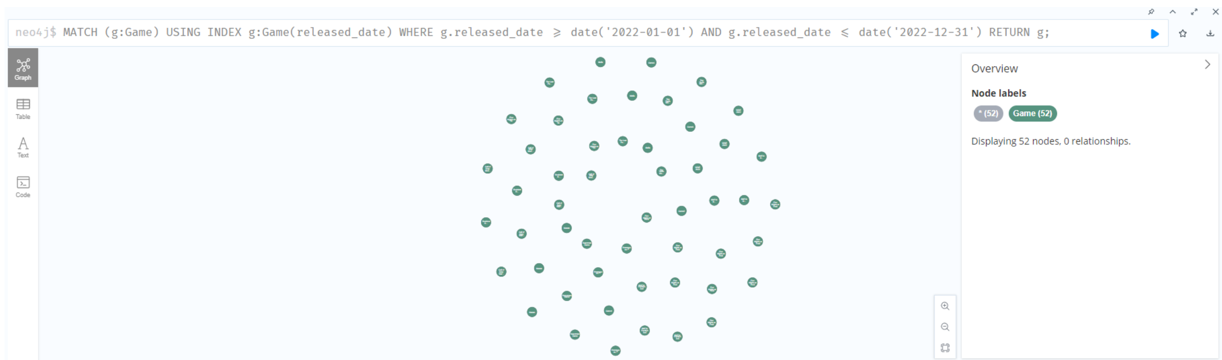
Query 2 Based on Q1:

// Query without index
MATCH (g:Game)
WHERE g.released_date >= date('2022-01-01') AND g.released_date <= date('2022-12-31')
RETURN g;

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

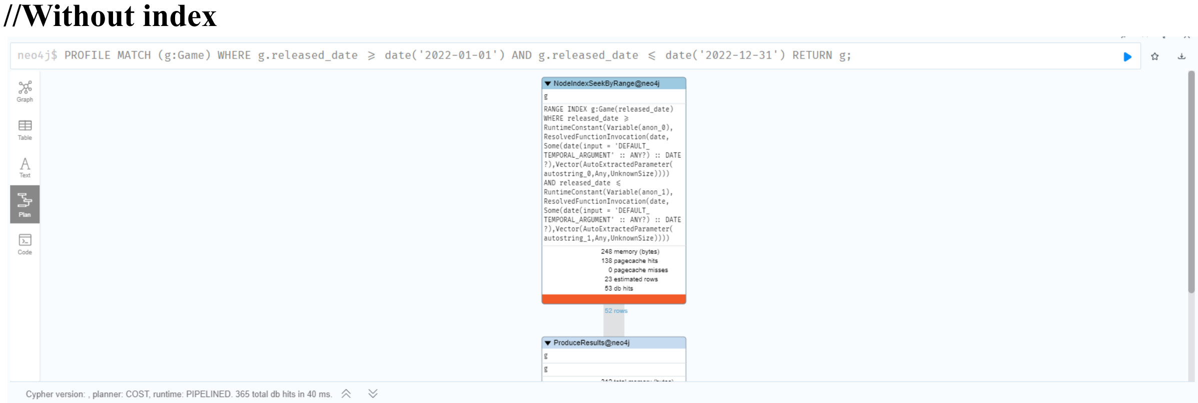


// Query with index
MATCH (g:Game)
USING INDEX g:Game(released_date)
WHERE g.released_date >= date('2022-01-01') AND g.released_date <= date('2022-12-31')
RETURN g;



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

//Without index



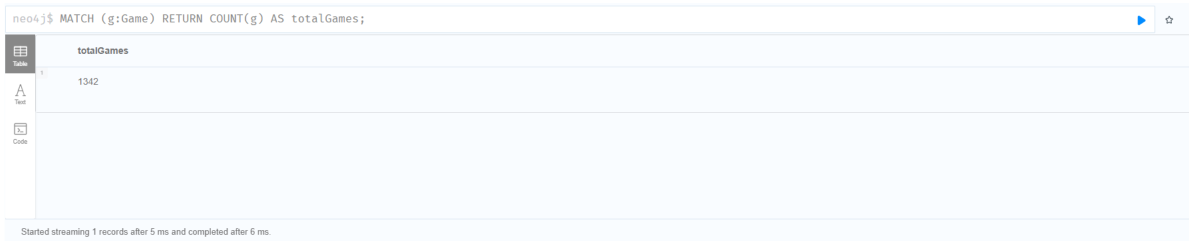
// With index

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;
Graph
Table
Text
Plan
Code
NodeIndexSeekByRange@neo4j
g
RANGE INDEX g:Game(released_date) WHERE released_date ⩾ RuntimeConstant(Variable(anon_0), ResolvedFunctionInvocation(date, Some(date(input = 'DEFAULT_TEMPORAL_ARGUMENT' :: ANY?) :: DATE ?),Vector(AutoExtractedParameter(autostring_0,Any,UnknownSize)))) AND released_date ⩽ RuntimeConstant(Variable(anon_1), ResolvedFunctionInvocation(date, Some(date(input = 'DEFAULT_TEMPORAL_ARGUMENT' :: ANY?) :: DATE ?),Vector(AutoExtractedParameter(autostring_1,Any,UnknownSize))))
248 memory (bytes)
138 pagecache hits
0 pagecache misses
23 estimated rows
53 db hits
52 rows
ProduceResults@neo4j
g
g
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



// Create index

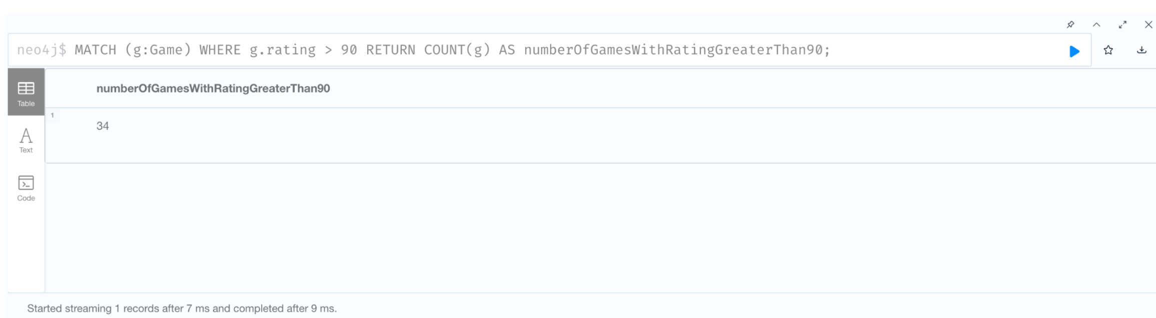
NOTE: *In this case, creating an index won't have a significant impact on the performance of this specific query because it doesn't involve filtering based on a specific property. Indexes are generally useful for improving the performance of queries that involve filtering or sorting based on specific 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):**

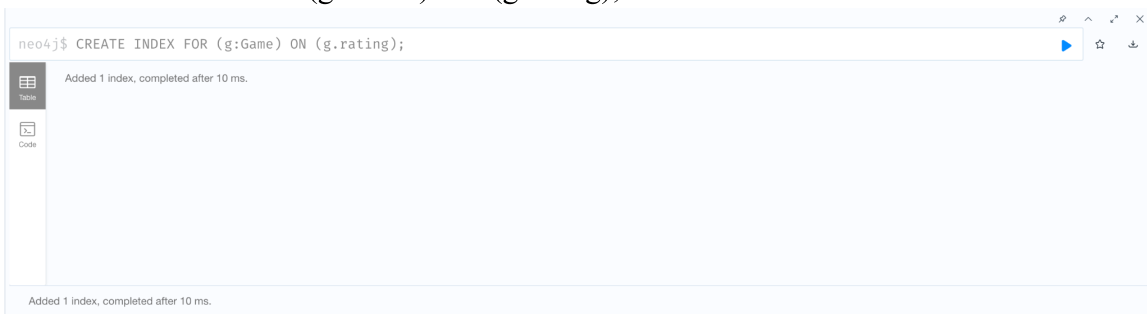
Count the total number of games in the database that have a rating greater than 90:

// Query without index

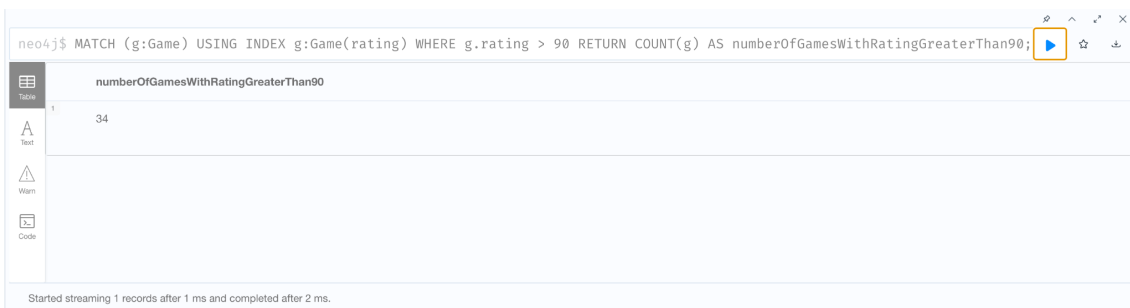
MATCH (g:Game)
WHERE g.rating > 90
RETURN COUNT(g) AS numberOfGamesWithRatingGreaterThan90;



// 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;

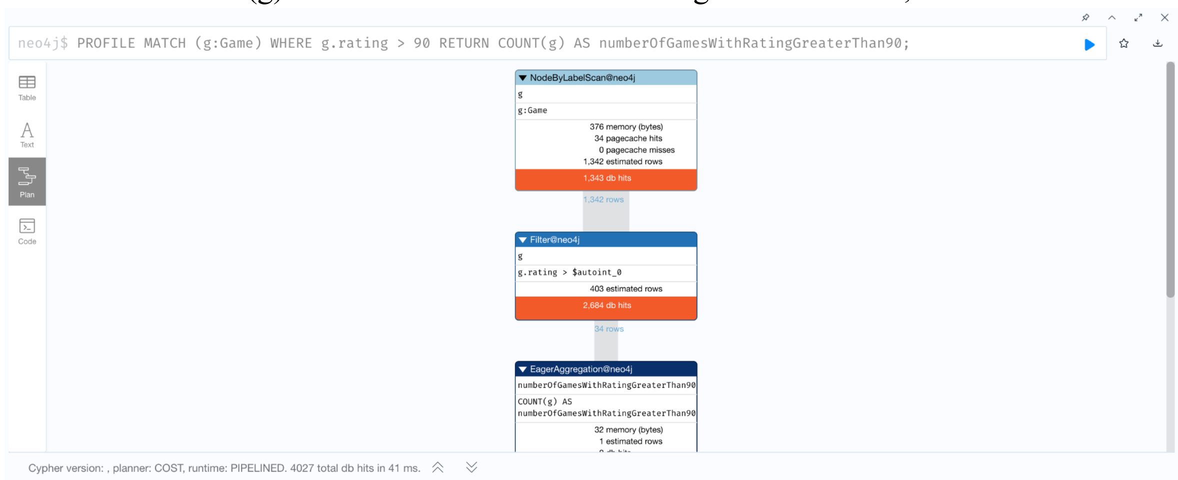


As you can see there is a difference of 2ms vs 9ms.

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

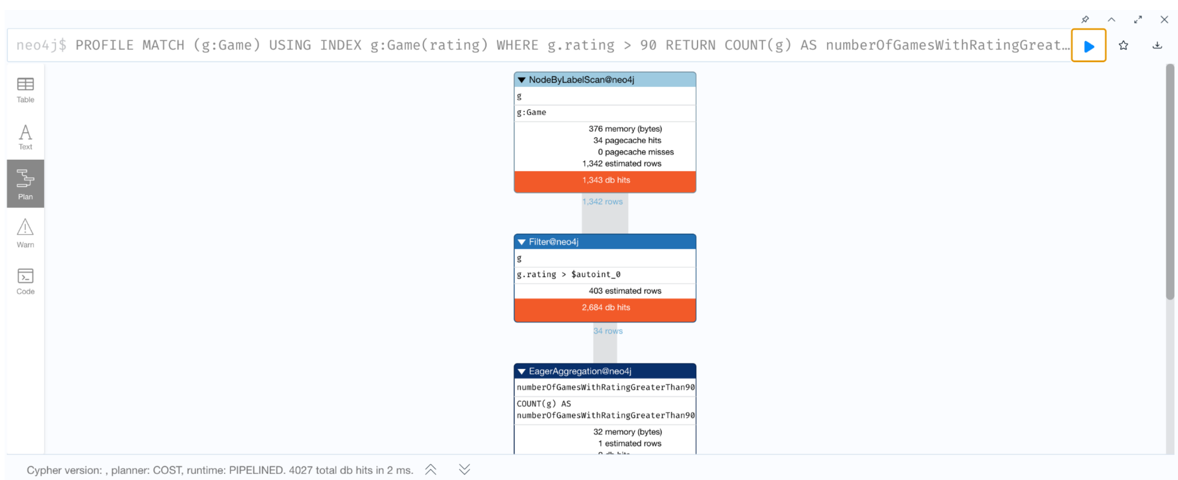
//Without Index:

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:

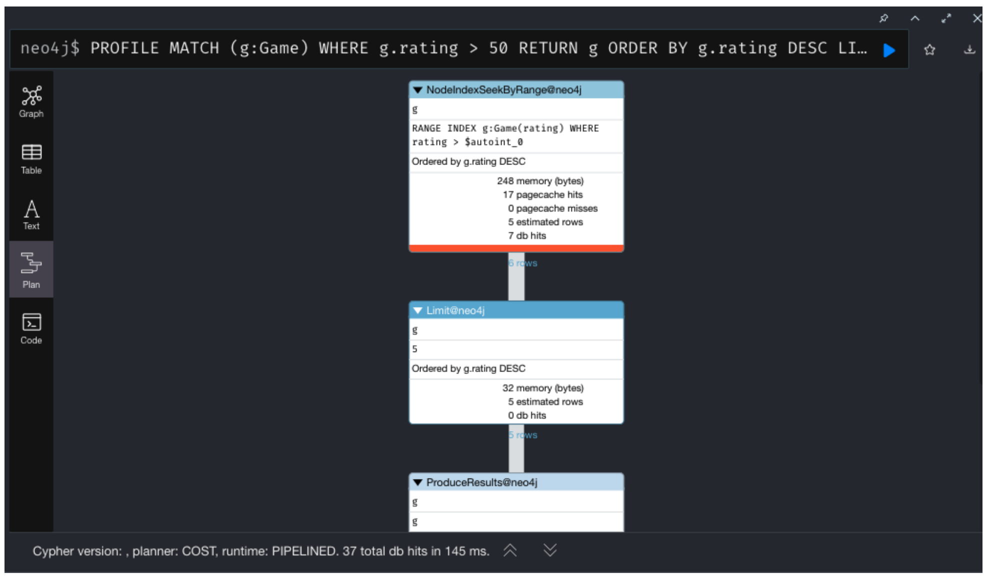
// Query without index
Retrieves top 5 games with a rating greater than 50, sorting them in descending order based on their ratings:
MATCH (g:Game)
WHERE g.rating > 50
RETURN g
ORDER BY g.rating DESC
LIMIT 5;

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

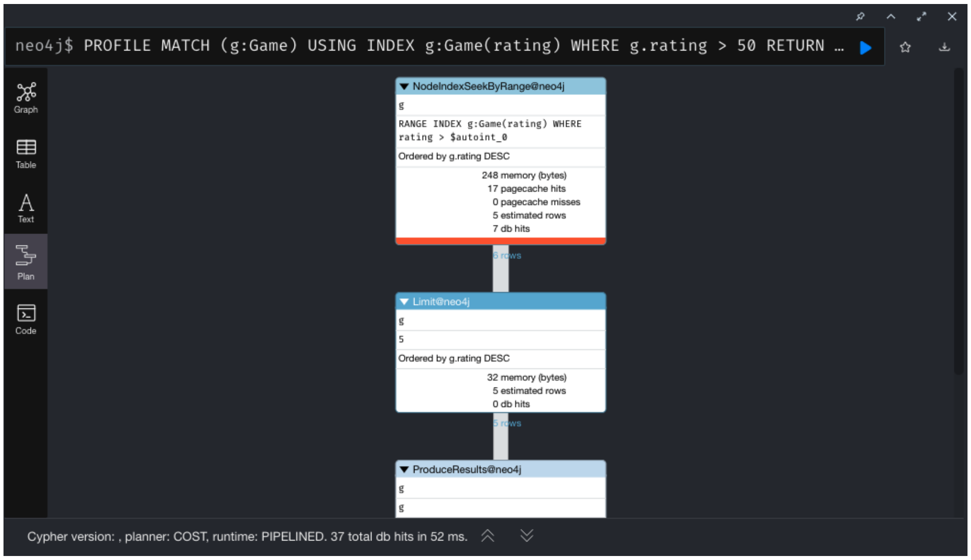
// Query with index
MATCH (g:Game)
USING INDEX g:Game(rating)
WHERE g.rating > 50
RETURN g
ORDER BY g.rating DESC
LIMIT 5;

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

//Querry without index
PROFILE
MATCH (g:Game)
WHERE g.rating > 50
RETURN g
ORDER BY g.rating DESC
LIMIT 5;



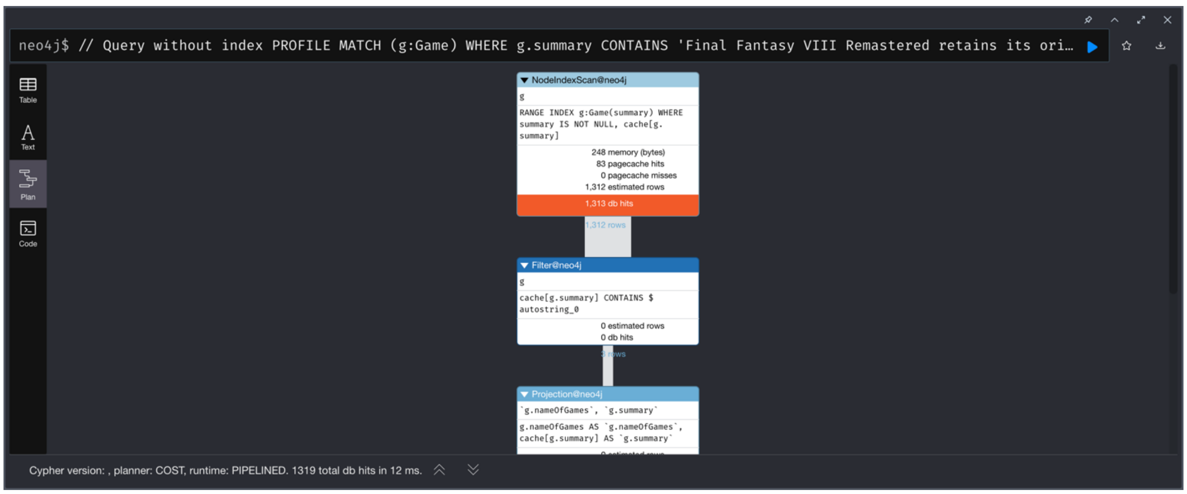
//Querry with index
PROFILE
MATCH (g:Game)
USING INDEX g:Game(rating)
WHERE g.rating > 50
RETURN g
ORDER BY g.rating DESC
LIMIT 5;



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

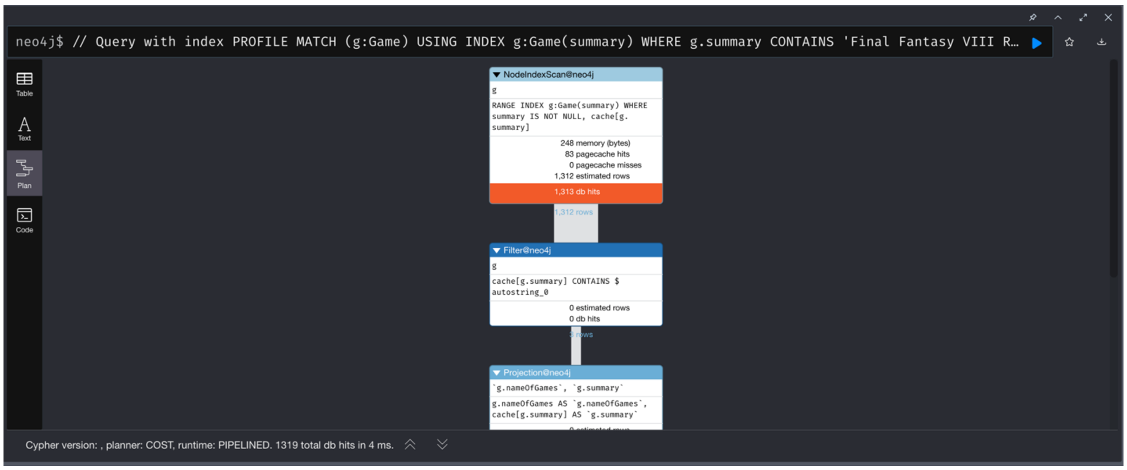
Q6. Demonstrate a full text search. Show the performance improvement by using indexes:

//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.