A Survey of Neo4j and its use in Retail

Sharath Kashyap
Master's Candidate of Information Systems Management
Carnegie Mellon University
Pittsburgh, PA, USA

***Abstract*—This paper outlines how many online retailers today have benefited from using Neo4j over common relational database management systems to maintain their connected data resources and build intelligent applications for their customers. These applications are important in terms of being able to provide the customers with cutting-edge conveniences and maintain a solid competitive advantage in an industry that is in constant flux. Going further beyond, I have also benchmarked the query performance of a popular RDBMS, MySQL against that of Neo4j in the context of a product recommendation engine.**

I. Introduction

The retail industry is continuously evolving. Popular retailers are constantly looking for ways to provide their users with a better shopping experience. The proliferation of connected data from various sources stemming from the digital transformation of the retail industry had created many challenges for the companies in terms of being able to store and process its data.

Traditionally, relational databases were used to manage all the data assets of retailers. The exponential increase in the amount of data being produced and recorded coupled with a very large user base that is constantly trying to query this data and a vast coverage space has rendered the performance of relational databases inadequate. Companies began to explore and implement alternate database technologies for optimizing the performances of individual use-cases that would give them key competitive advantages.

This paper aims to discuss how some popular retail brands such as eBay, Walmart, etc. addressed some complex challenges posed by their modern information systems by moving from relational databases to neo4j, an open-source graph database.

II. Common problems of RDBMS in retail Applications

Relational databases require adherence to a predefined schema and format. Today's dynamic retail environment requires business agility to gain and maintain a competitive edge. Developers need to relax some of the constraints that relational databases place on schema structure to be able to match the performance requirements of current information systems.

Despite the name, relational databases are not robust with their performance while handling the multitude of relationships between connected data blocks. As the size and number of the database tables increase, so does the computational complexity while trying to connect and query information by joining these tables.

Retail information systems are complex and require the maintenance of hundreds of tables and a single operation may require the joining all of these tables. With hundreds of thousands of requests being made every second, an incredible amount of stress is being placed on the servers.

A. Non-Intuitive Data Model

When visualizing connected data and building a data model, developers usually think in terms of graphs as they sketch the design on a whiteboard. They would then force it into a tabular format which is an RDBMS schema with tables for similar entities and joining tables to handle M:M relationships [3].

In the retail industry, agility is very important. The object relational impedance mismatch stemming from the disconnect between the intuitive visualization of the data model as a graph and the actual RDBMS data model interferes with productivity. These databases would take some time to conceptualize and an even longer time to implement.

Furthermore, the complexity and rigidity of the RDBMS data model, while useful from the data integrity standpoint, slows down the development and upgradation of the tool being developed. The release cycles could also be slowed down, which is costly from a revenue standpoint because the first mover advantage could be lost.

B. Extensive Join Operations

Consider the example of an application that does delivery plan mapping. Most online retailers today provide these delivery services and must plan out delivery routes carefully; both to ensure quick delivery to satisfy customers as well as minimum expenditure by choosing the optimum courier and route.

Figure 1 shows a simple abstraction of the number of joins that would be required to plan a delivery route for an individual customer. In reality, there would be several tens of more tables including payment processing functions and possible connections to external courier databases, traffic databases and weather databases through APIs. The retailer's server would also be handling hundreds of thousands of requests per second.

These types of operations consume an incredible amount of compute resources and require the execution of very lengthy and

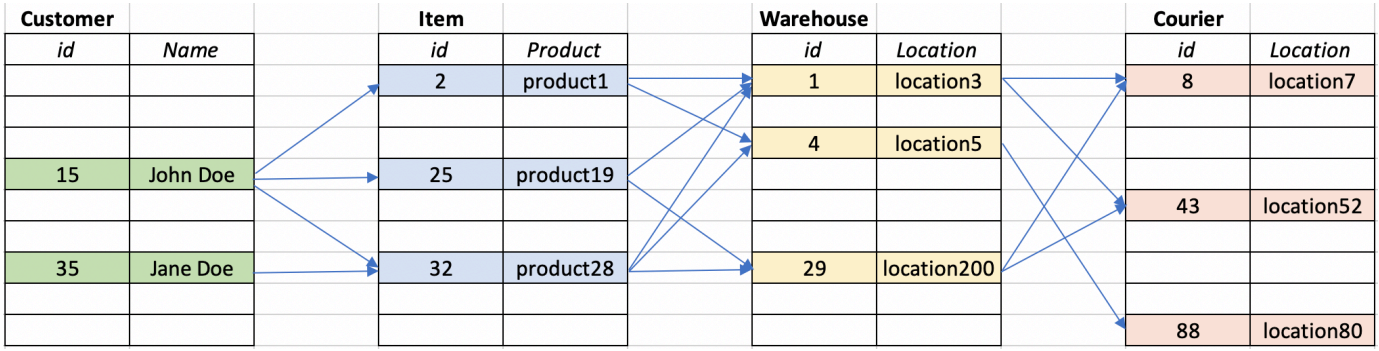

Figure 1: A sample subset of tables and associated join operations required for routing using a RDBMS

complex queries. Due to the complexity of the queries and the sheer number of join and lookup operations, the query would take a very long time to return the results to the consumers. This would disrupt the shopping experience of customers and the delays due to these complex and convoluted operations might hinder the server's ability to provide the consumers with real-time information [1].

Furthermore, in the context of a recommendation service, several recursive joins would need to be executed which would cause delays to many customers due to availability restrictions. These types of operations are required to understand the consumers' purchasing and browsing trends by studying past transactions so that the recommendations made would be precise and exclude items that the customer has purchased already or has no interest in [2].

The technologies that are commonly used to process Big Data such as Hadoop and Spark work well with recommending products via email, perhaps once a day or during standard intervals. However, even they are not suitable and performant enough for real-time query processing.

C. *Constant Schema Updates*

Retail software applications evolve rapidly in terms of functional requirements and the constant addition of new data sources to differentiate them from competitors and maintain a strong advantage in terms of the unique services they offer. Today, social engines power commerce by connecting people and this is being incorporated into many product advertising and marketing schemes.

Application developers need to be able to create and incorporate new features and services to the existing portfolio of features every few weeks. These can range from a small addition of data tables to a complete overhaul of the database layer. These changes must be incorporated in a non-intrusive manner so that the service remains live at all times and the new features don't disrupt the working of the existing services.

In the case of RDBMS and its static schema design, the entire system needs to be taken down to enable such an update. This kind of disruption cannot be tolerated in today's retail environment which hedges its popularity on availability. If the service is down or someone's user experience is interrupted, the damage to the brand can be severe [1].

More flexible models are required to counter this weakness and allow for smooth transitions and data model upgrades.

D. *Data Inconsistencies*

Providing real-time and precise service to the customers is paramount in the online retail industry. Customers require that high-quality results become available to them instantly and are not willing to wait more than a few seconds to be serviced.

RDBMS implementations struggle to provide fast and efficient query processing for these kinds of applications that sift through millions of rows and require several joins to fetch a single result. And, this operation has to be executed for thousands of users simultaneously.

In the past, to avoid this kind of resource utilization, results for recommendations, routing, etc. were pre-aggregated and stored [2]; much like how the values of an OLAP cube of a data-warehouse are re-computed during the off-hours after the ETL process adds the latest data to the fact tables. However, this would mean that users are not getting real-time results. Outdated results in the context of product recommendations or critical logistics planning could in most cases be meaningless and possibly harmful. Inefficiencies in these areas could accelerate churn.

Furthermore, using pre-aggregated data as a substitute for real-time data is not efficient in situations where all the data needs to be pre-computed even though only a fraction of it is being accessed in real-time.

III. NEO4J AND HOW IT ADDRESSES THESE ISSUES

Unlike RDBMS, Neo4j is flexible in terms of its schema and developers can freely make changes to the database without the need to take the database down. The Neo4j data schema is built completely around data relationships. Relational databases treat relationships as a structure, but graph databases treat it as data that holds information like directionality and weights.

These relationships or 'joins' are established during the insertion in graph databases unlike in relational databases where they have to be re-computed each time while executing a fresh query. Hence, with graph databases, it would feel like the

relationships are available at all times and this drastically speeds up queries that would traditionally require a long time to run due to join operations. [4] Applications that make use graph databases enjoy extremely high real-time performance even while executing complex and lengthy queries.

The query structure of Neo4j is vastly different from SQL and is optimized for the function of traversing relationships. The Cypher query language of Neo4j requires far fewer lines of code than SQL for the same operation that has to join many tables. The computational complexity of SQL queries in relational databases increases drastically as the number and size of tables increases because the number of look-ups and indexes would also go up. However, graph databases don't share this weakness as the relationships are pre-established.

Despite being flexible, Neo4j offers ACID compliant transaction support and ensures data integrity. All in all, from using Neo4j, developers are able to build and deploy highly performant and competitive products in a shorter development window.

IV. Case Studies

I am going to use the examples of some popular online retailers and explain how Neo4j has been used to transform some of the important services delivered by these companies and their importance to the customers.

A. Product Recommendations

Walmart deals with around 250 million customers every week through all of its stores and retail websites around the globe. Hence, massive amounts of transactional data get generated and need to be processed to be able to deliver specialized and targeted services to all of its customers to improve their shopping experience.

The company was using an RDBMS system at the backend of its product recommendations engine. To be able to deliver personalized recommendations rather than general ones to all of its customers, the amount of data that would need to be processed in each instance was massive and too much for the RDBMS to handle [5]. The SQL queries to accomplish this were also complex and tough to maintain.

The task of connecting huge masses of the complex buyer and product data to understand purchase trends in real-time was required in order to cross-sell product lines to different customers. Hence, an alternative, Neo4j was considered and used first by its eCommerce team in Brazil. The performance gains were immediately apparent.

Since migrating to Neo4j, Walmart has been able to understand customer and product trends by querying past purchases quickly and capturing any new interests by studying the latest transaction patterns. This has enabled them to make accurate real-time recommendations with ease by matching past purchases and the latest session data. Short and simple Cypher queries were used over the complex SQL queries and batch processes of the past. Cypher queries were performant, provided low latency and easy to maintain and upgrade.

eBay has been also been developing Neo4j to deploy a new solution that revolutionizes conversational commerce. It is using Neo4j to manage a natural language understanding knowledge graph and take the process of making product recommendations to the next level [6]. The company had identified the lack of assimilating context in its recommendation engine algorithm as details such as size, color, season, etc. are what inform buying decisions.

The upgraded functionality provides general users with precise product recommendations that match product characteristics as well when they type in a sentence to search. This is done by traversing the knowledge graph whose nodes have been loaded with advanced natural language understanding algorithms that match historical purchase trends and grammatical intention to the current product catalog. The Neo4j backend has remained performant and responsive to user requests in real-time even with over a million nodes in the knowledge graph. The front-end for this Neo4j powered tool is available to the users in the form of chatbots and also a google assistant plugin in the effort to provide the user with the experience of conversational commerce.

In Neo4j, relationships are stored as data elements. Understanding this in the conventional RDBMS sense, tables such as customers, products, etc. are stored as nodes in the graph and joins such as 'bought', 'likes', 'purchased', etc. are stored as data points that indicate named and directed relationships in Neo4j. [5] Relationships connect nodes and are structurally pre-materialized during the graph creation phase.

This makes Neo4j a clear-cut winner in terms of defining real-time recommendations for customers who have a long purchase history and could be recommended the products that have been purchased by other customers who share similar interests and buying patterns. Rather than recommending generally popular products, systems built on the backbone of Neo4j are able to personalize recommendations by querying deeper into the customer information pool with little impact on performance; even to the point of recommending items that are in the long tail.

B. Optimization of Delivery Routing Services

Amazon had set the standard for guaranteed 2-day purchase deliveries. Customers now don't tolerate longer delivery times. Since eBay's acquisition of Shutl, a platform that coordinates between stores and warehouses, courier services and customers on a 24/7 schedule to provide the most optimum delivery plans. The platform fuels their 'eBay Now' service which aims to provide same-day delivery. Also, it allows users to be able to specify a time and date for the delivery for additional convenience [7].

However, as the service began to grow in coverage and functionality, the MySQL backed could not handle the increased traffic; again, due to the number of joins required to service a single customer. The request would have to take into consideration several different factors such as the physical

network conditions, weather, traffic and also visibility of the stores, distribution centers, etc.

The routing service would have to consider the best option out of all the available permutations to return the final result. The query would have to sift through tens of lengthy tables in the schema and accomplishing this in real-time was a challenge considering the number of requests. The SQL queries and procedures to retrieve the best courier were computationally very expensive to be viable for a real-time and on-demand service.

Since the inherent properties of Neo4j resonated with the problem statement perfectly, the service was re-designed with a Neo4j backed. This implementation overcame the past scalability, complexity and rigidity issues and gave the developers the freedom to add more features to the platform that wouldn't be supported by the prior implementation due to its load restrictions.

The technology of Neo4j coupled with the crisp and short Cypher queries achieved constant time performance for the service. This was achieved because of the localized graph traversals to calculate the best route plan for the current request. Furthermore, the ACID properties of Neo4j added reliability to the mix.

C. *Managing Network and IT infrastructure*

Most online retail applications have invested a significant amount of resources in their IT stack. This is to ensure that the information systems stay online and active; both for internal management as well as to provide an uninterrupted and smooth experience to customers.

However, as the reach and number of services provided by the retailer increases due the increase in physical data centers or virtual machines in the cloud, so does the complexity of the system. This limits the ability of the administrators to constantly monitor the infrastructure and attend to any problems quickly because RDBMS implementations have very slow response times for systems that have a vast scale.

Neo4j can be used to catalog all physical as well as cloud assets, their deployment as well as the relationships between each of these elements as shown in Figure 2. Having the infrastructure set up with a graph backend would help the administrators to visualize the complete system more intuitively and identify performance bottlenecks with ease. It could also help with assessing and dealing with latency issues for operations that require lengthy traversals. Security risks are also highlighted when the entire network is visualized as a graph [8].

Graph database implementations could also help with the development of analytics for internal process optimizations and IT resource profiling. They could inform applications, services, hardware, etc. that are consuming more resources during specific time periods and also the resources that are servicing customers for specific queries and requests. By the nature of their architecture, queries and traversals in Neo4j remain localized which results in highly performant and available systems.

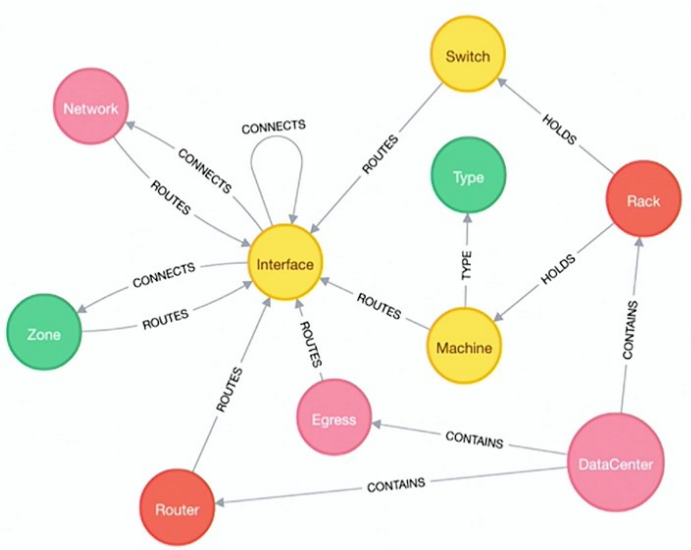

Figure 2: A sample IT network visualized in a graph-DB context [9]

Furthermore, an important requirement of IT infrastructure is its flexibility to change and promote physical growth. Unlike RDBMS systems, new services as applications, data elements as nodes and dependencies as relationships can be added or modified with ease agility, while maintaining ACID compliance and the associated data integrity. This makes periodic architectural overhauls a less daunting task when using Neo4j.

D. *Supply Chain Management*

Owing to the vast supply chain that most retailers use today, there is a high chance that anomalies get unnoticed without a secure information system. Items go through several entities from their origin point and the destination entities are rarely aware of the origin points of the items that get delivered to them. The lack of the ability to track the movement of these items can cause several problems ranging from contaminated inventory to illicit suppliers getting added to the chain.

Transparency One is a company that built a platform atop Neo4j to provide their customers with the ability to track activities in their supply chain end-to-end. The main goal was to make the platform highly available and performant at any instant to allow users to get a completely transparent view of the full supply chain and then be able to query the map for specific results as well as build analytics around all the data in the map. This would help the customers to stay informed about the best sources from which they are purchasing products and available alternatives in the event of an emergency [10].

Building such a platform was going to be impossible with an RDBMS solution due to the scale and structure of the supply chain for most online retail stores. The SQL search queries would not be performant enough and the complexity of the queries would hinder productivity. Hence Neo4j was adopted and since the initial release, several new modules have been added to the platform that are providing the customers with an unrestricted view of their supply chain.

E. *Differential Pricing*

It has never been easier for customers to be able to compare the price of a particular item on different retail websites. Hence, it is important for pricing optimization to be precise in real-time to stay competitive.

This, of course, is based on several different fluid factors such as the company's current inventory, location, etc. Consider for instance, the merchandise for a particular NBA team that is currently playing in the finals. The winning team's merchandise needs to be priced higher than the losing team's merchandise as it is definitely going to sell quicker and in large numbers after the game.

Retailers need to be able to match this supply and demand in real-time and optimize their prices ahead of competitors so that they gain an edge. Furthermore, this optimization must be done in a micro-market scale as the value of the merchandise varies by region. Relational databases cannot keep up with these complex economic rules and cannot deliver the required performance in real-time.

Marriot International wanted to optimize its pricing engine and make it faster. Their initial RDBMS implementation ran on top of powerful mainframe hardware that was constantly tuned to maximize performance. Mainframes run processes in batches, and this was not conducive to the provisioning of real-time results. The pricing updates would take hours, and this caused users to churn. This problem was magnified with the addition of more properties and locations. Pricing is a very sensitive area and required the database to be highly normalized which increased the number of join operations required to process even trivial operations. Some 30,000 lines of SQL queries were required to run a few of their complex optimization queries. This was hardly optimal for a business requirement that required the results to be returned in less than a minute [11].

Since moving the data resources of all its properties to Neo4j, setting up individual graphs for each of their properties and then interlinking them globally, the pricing queries have seen a 10-fold increase in performance and massive reductions in query execution times and in turn server and infrastructure expenses.

V. Benchmarking Neo4j and MySQL

I designed and executed an experiment to empirically measure and confirm the benefits of Neo4j over a RDBMS like MySQL. I created instances on my local machine within a sandboxed environment having 8 gigabytes of RAM.

Once the databases were created and populated, I executed queries that accomplish the same purpose in Neo4j using its REST client [15] and MySQL using the sqlalchemy [16] engine. The query calls were made from within the Jupyter environment of Python. All of the code and associated data have been uploaded to my Github repository that is linked in the references section [12].

A. *The Dataset and Pre-Processing*

The dataset that I am using comes from a coding challenge about market basket analysis. The goal was to use association rule mining algorithms or clustering to gain an understanding of buying patterns. So, it was the perfect dataset for me to experiment with. The data was very unclean and came from two different comma separated value files. I wrote a script in python within the Jupyter environment to decompose the data into a format that was amenable to load into the two databases with which I was experimenting. The original dataset that mapped purchased items to the customers contained 39,474 rows. However, these rows were aggregated as a row of lists to indicate all of the items that each customer had bought, within a single row. So, I used a flattening logic to disaggregate these rows and the final result contained 319,995 data points. Furthermore, I grouped this dataset by the item name and user id to compute how many times users had repurchased the same item. Figure 3 shows a sample of the resulting dataset.

Item_name	user_id	count
pasta	562712	1
spaghetti sauce	672572	2
soda	980851	1
baby items	354671	1
cauliflower	1246489	1

Figure 3: A sample of cleaned data used in the experiment

B. *Product Recommendations using an RDBMS*

I set up an instance of MySQL in my local sandbox and created a database with 2 tables. Table 'ITEM' stores all unique items, 'AGG' stores the mapping of users and items along with the quantities of each of the items. The data points were loaded through python code; specifically using the 'sqalchemy' [16] package that abstracts most of the complex database functionality in a light and simple package. I found a SQL query

```
SELECT item.item AS Recommendation, count(1) AS Frequency
FROM item, agg, (SELECT agg3.item_name, agg3.user_id
    FROM agg agg, agg agg2, agg agg3
    WHERE agg.user_id = "18710"
    AND agg.item_name = agg2.item_name
    AND agg2.user_id != "18710"
    AND agg3.user_id = agg2.user_id
    AND agg3.user_id NOT IN (SELECT DISTINCT item_name
    FROM  agg agg
    WHERE agg.user_id = "18710")
    ) recommended_products
WHERE agg.item_name = item.item
AND agg.item_name IN (recommended_products.item_name)
AND agg.user_id = recommended_products.user_id
GROUP BY item.item
ORDER BY Frequency DESC;
```

Figure 4: An SQL query for product recommendations

for product recommendations in one of the case-studies from the Neo4j website [4] and adapted it to suit my specific requirement.

The query in Figure 4 uses multiple aliases on table AGG to query the best recommendations for user '18710'. It also works to eliminate the items that have already been purchased by the user. The logic is based loosely on the concept of collaborative filtering that is used in popular recommender systems [13].

C. *Product Recommendations using a Neo4j Graph*

I set up a local graph database instance using the Neo4j desktop environment. The inbuilt loading utilities [14] of Neo4j allowed me to quickly push the data and relationships into the graph. The graph is built around two types of nodes; 'User' nodes to indicate customers and 'Item' nodes to indicate purchased products.

The 'PURCHASES' relationship connects these entities and the quantity of the items purchased by users is stored as an attribute of this relationship. To facilitate quicker querying, I built indexes on the 'item' field of Item nodes and 'user' field of User nodes. Figure 5 shows a small section of the resulting graph database.

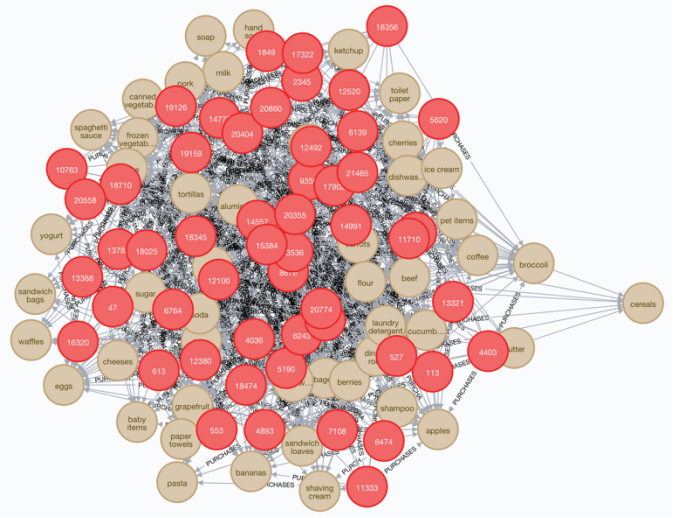

Figure 5: Graph view of the database

Once again, I adapted an existing product recommendations query [4] that was written in Cypher to my database and executed the query using a python script to measure the time. Figure 6 shows the cypher query used.

```
MATCH (u:User {user:'18710'})-[:PURCHASES]->(i:Item)<-
[:PURCHASES]-(peer:User)-[:PURCHASES]->(reco:Item)
WHERE NOT (u)-[:PURCHASES]->(reco)
RETURN reco as Recommendation, count(*) as Frequency
ORDER BY Frequency DESC LIMIT 50;
```

Figure 6: A Cypher query for product recommendations

The Cypher query was much simpler and shorter to understand and adapt when compared with the related SQL query.

D. *Comparing Execution Times*

I ran both the Cypher and SQL query in a loop over ten iterations. As expected, the Cypher query performed significantly better in each iteration.

As is evident from Figure 7, Cypher had an average query performance that was over four times that of SQL in the same environment and while performing the same task. This is all because of the number of self joins and subqueries that are involved in the SQL query.

Trial	Cypher Results (seconds)	SQL Results (seconds)
1	6.27	26.3
2	5.37	26.4
3	5.4	27
4	5.34	26.9
5	5.37	28.2
6	6.37	26
7	6.42	26.5
8	5.82	26.7
9	6.05	26.3
10	6.02	26.9
Average	5.843	26.72

Figure 7: Benchmarking Results (obtained using '%time' in Jupyter)

A stark difference in performance can be observed in this simple implementation of a product recommendation engine. However, when this product scales to the degree that is required to power e-commerce websites, the performance difference is much more substantial. This is due to the numerous relationships that can exist between each of the nodes in the graph and also the sheer size of the data. All of these aspects coupled with the number of ad-hoc requests that need to be serviced at any instant puts an enormous strain on an RDBMS backed system.

Neo4j has been proven to provide staggering gains in performance along with an easy way to maintain and upgrade some elements of the overall data ecosystem of a retail company.

REFERENCES

[1] Staff, N. (2019). *The Database Model Showdown: An RDBMS vs. Graph Comparison - Neo4j Graph Database Platform*. [online] Neo4j Graph Database Platform. Available at: https://neo4j.com/blog/database-model-comparison/ [Accessed 24 Feb. 2019].

[2] Staff, N. (2019). 5 Sure Signs It's Time to Give Up Your Relational Database - Neo4j Graph Database Platform. [online] Neo4j Graph Database Platform. Available at: https://neo4j.com/blog/five-signs-to-give-up-relational-database/ [Accessed 24 Feb. 2019].

[3] Neo4j Graph Database Platform. (2019). *Data Modeling Concepts and Techniques | Neo4j*. [online] Available at: https://neo4j.com/developer/guide-data-modeling/ [Accessed 24 Feb. 2019].

[4] Neo4j Graph Database Platform. (2019). White Paper: Overcoming SQL Strain and SQL Pain - Neo4j Graph Database Platform. [online] Available at: https://neo4j.com/resources-old/overcoming-sql-strain-white-paper/ [Accessed 24 Feb. 2019].

[5] Neo4j Graph Database Platform. (2019). Walmart - Neo4j Graph Database Platform. [online] Available at: https://neo4j.com/case-studies/walmart/ [Accessed 24 Feb. 2019].

[6] Neo4j Graph Database Platform. (2019). Neo4j Powers Intelligent Commerce for eBay App on Google Assistant. [online] Available at: https://neo4j.com/case-studies/ebay/ [Accessed 24 Feb. 2019].

[7] Neo4j Graph Database Platform. (2019). eBay's Competitive Advantage in Same-Day Delivery Using Neo4j - Neo4j Graph Database Platform. [online] Available at: https://neo4j.com/blog/ebay-competitive-advantage-neo4j/ [Accessed 24 Feb. 2019].

[8] Neo4j Graph Database Platform. (2019). Managing Network Operations with Graphs - Neo4j Graph Database Platform. [online] Available at: https://neo4j.com/business-edge/managing-network-operations-with-graphs/ [Accessed 24 Feb. 2019].

[9] YouTube. (2019). Building a Real-time Recommendation Engine With Neo4j - Part 1/4 - William Lyon - OSCON 2017. [online] Available at: https://www.youtube.com/watch?v=wbI5JwIFYEM [Accessed 24 Feb. 2019].

[10] Neo4j Graph Database Platform. (2019). Transparency-One - Neo4j Graph Database Platform. [online] Available at: https://neo4j.com/case-studies/transparency-one/ [Accessed 24 Feb. 2019].

[11] Neo4j Graph Database Platform. (2019). Retail & Neo4j: Pricing & Revenue Management. [online] Available at: https://neo4j.com/blog/retail-neo4j-pricing-revenue-management/ [Accessed 24 Feb. 2019].

[12] Kashyap, S. (2019). sharathkashyap92/Neo4j-for-Retail. [online] GitHub. Available at: https://github.com/sharathkashyap92/Neo4j-for-Retail [Accessed 24 Feb. 2019].

[13] En.wikipedia.org. (2019). Collaborative filtering. [online] Available at: https://en.wikipedia.org/wiki/Collaborative_filtering [Accessed 24 Feb. 2019].

[14] Neo4j.com. (2019). 3.20. LOAD CSV - Chapter 3. Clauses. [online] Available at: https://neo4j.com/docs/cypher-manual/current/clauses/load-csv/ [Accessed 24 Feb. 2019].

[15] Neo4j-rest-client.readthedocs.io. (2019). neo4j-rest-client's documentation — neo4j-rest-client 2.0.0 documentation. [online] Available at: https://neo4j-rest-client.readthedocs.io/en/latest/info.html [Accessed 24 Feb. 2019].

[16] Towards Data Science. (2019). SQLAlchemy — Python Tutorial – Towards Data Science. [online] Available at: https://towardsdatascience.com/sqlalchemy-python-tutorial-79a577141a91 [Accessed 24 Feb. 2019].