



Master's thesis
Master's Programme in Data Science

Multi-Perspective Evaluation of Relational and Graph Databases

Cheng Chen

March 11, 2022

Supervisor(s): Prof. Jiaheng Lu

Examiner(s): Assoc. Prof. Michael Mathioudakis

UNIVERSITY OF HELSINKI
FACULTY OF SCIENCE

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki

Tiedekunta — Fakultet — Faculty		Koulutusohjelma — Utbildningsprogram — Degree programme	
Faculty of Science		Master's Programme in Data Science	
Tekijä — Författare — Author			
Cheng Chen			
Työn nimi — Arbetets titel — Title			
Multi-Perspective Evaluation of Relational and Graph Databases			
Työn laji — Arbetets art — Level		Aika — Datum — Month and year	
Master's thesis		March 11, 2022	
		Sivumäärä — Sidantal — Number of pages	
		51	
Tiivistelmä — Referat — Abstract			
<p>How to store data is an enduring topic in the computer science field, and traditional relational databases have done this well and are still widely used today. However, with the growth of non-relational data and the challenges in the big data era, a series of NoSQL databases have come into view. Thus, comparing, evaluating, and choosing a better database has become a worthy topic of research.</p> <p>In this thesis, an experiment that can store the same data set and execute the same tasks or workload on the relational, graph and multi-model databases is designed. The investigation proposes how to adapt relational data, tables on a graph database and, conversely, store graph data on a relational database. Similarly, the tasks performed are unified across query languages. We conducted exhaustive experiments to compare and report the performance of the three databases. In addition, we propose a workload classification method to analyze the performance of the databases and compare multiple aspects of the database from an end-user perspective.</p> <p>We have selected PostgreSQL, ArangoDB, Neo4j as representatives. The comparison in terms of task execution time does not have any database that completely wins. The results show that relational databases have performance advantages for tasks such as data import, but the execution of multi-table join tasks is slow and graph algorithm support is lacking. The multi-model databases have impressive support for simultaneous storage of multiple data formats and unified language queries, but the performance is not outstanding. The graph database has strong graph algorithm support and intuitive support for graph query language, but it is also important to consider whether the format and interrelationships of the original data, etc. can be well converted into graph format.</p> <p>ACM Computing Classification System (CCS): Information systems → Data management systems → Database administration → Database performance evaluation Information systems → Data management systems → Query languages Information systems → Data management systems → Database design and models</p>			
Avainsanat — Nyckelord — Keywords			
Relational Database, Graph Database, Database Benchmark, Database Comparison			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

Contents

1	Introduction	2
2	Background and Related Work	5
2.1	Relational Database	5
2.2	Graph Database	6
2.3	Multi-model Database	6
2.4	Database Benchmarks	7
2.5	Comparison of Databases	8
2.5.1	Data Science Support	8
2.5.2	GUI and Graph Visualization	9
2.5.3	Toward Cloud Database	10
3	Methodology	12
3.1	Data Sets	12
3.1.1	TPC-DS Set (Table-like)	12
3.1.2	Twitch Set (Graph-like)	15
3.2	Quantitative Analysis	15
3.2.1	Table Data	16
3.2.2	Graph Data	21
4	Experiments and Results	25
4.1	Environment and Configuration	25
4.2	Data Production and Import	26
4.2.1	TPC-DS Data	26
4.2.2	Twitch Data	26
4.3	Tasks Execution Time	27
4.4	Workload Classification	32
4.4.1	Original Data Description	32
4.4.2	Dimensionality Reduction	32
4.4.3	Clustering	34

5 Discussion	38
5.1 Performance	38
5.1.1 Execution Times	38
5.1.2 Cache	39
5.2 Which One Should I Choose?	39
6 Conclusions	41
Acknowledgement	42
Bibliography	43
Appendix A TPC-DS Data ERD Supplementary and Edge Schema	47
A.1 ER-Diagram for TPC-DS Web/Store Parts	47
A.2 TPC-DS Data Graph Relationship Mapping Table	48
Appendix B Experiments Raw Results	49

1. Introduction

This century is an era of dramatic data growth. The Big Data industry has proposed the 6-Vs: Volume, Variety, Velocity, Value, Veracity, and Variability, and how to store and use these data has become a hot topic. A good example is that within the recent decade, In addition to the common table data, the rapid growth of emerging applications such as social networks, cryptocurrencies, bioinformatics network analysis, and traffic navigation has brought about extensive graphical data. With large scale, complex internal structure, and diverse query requirements in different domains.

As one of the most common and popular data storage solutions in recent decades, relational databases represent data with relationships, use constraints to control data consistency and completion, and have SQL to perform a range of operations. Although storing graph data in a relational database has proven to be feasible [9], it was evident that it can't meet everyone's requirements anymore in the face of the increasing complexity of large-scale graphs and diverse demands on graph algorithms.

Some NoSQL databases are starting to make their mark, Neo4j, a native graph database, supports graph data storage built from the ground up. It can not only store data but also exploit data and even relationships between data. It even proposes a new declarative language to work with the database for operations such as querying. According to Neo4j itself [26], the speed and efficiency advantage of the graph database has driven dozens of game-changing use cases in crime identification such as fraud detection, life sciences, data science, knowledge graphs, etc.

More than that, a number of multi-model databases are being introduced and used, which, as the name suggests, are designed to solve the problem of how to store many different types of data within a single database. ArangoDB as a representative, its CEO called it a native multi-model database. It is both a document storage database and a key-value storage database and a graph storage database. Multiple formats in a single database engine, only one query language which can be used in all data models, even allowing them to be integrated and simultaneous queries in the same database [4].

For a long time, it has been discussed whether NoSQL databases can replace SQL databases as the mainstream because of better query performance and support for more data storage types. A certain number of researchers have the view that SQL may fade

into history because it only supports table data storage. This does not fit with the variety of data formats today [5]. On the other hand, some researchers say that SQL databases can also be used to store a variety of data types, such as graph data, which requires only some conversion work, and that SQL databases also did not lose in performance aspect to graph databases [29]. A question at hand is that there is no convincing test to prove which is faster and better - the lack of uniform benchmarking.

This thesis will conduct an exploration of such a unified benchmarking perspective, try to solve a simple but comprehensive coverage problem; which database should I choose to store my data? The contributions of this thesis are as follows:

- We designed a unified comparison experiment on PostgreSQL (relational), ArangoDB (multi-model), and Neo4j (graph) from the perspective of both tabular data and graph data.
- We applied graph algorithms (graph traversal and shortest path) on a relational database.
- We proposed a method to transform relational data into graph data, present a way to store graph data into a relational database, and store the same data in three databases for testing with the same workload.
- We explore three databases by analyzing the workloads regarding unsupervised clustering using both the experiment results and features we defined related to multiple executions.

In detail, We select TPC-DS, a relational database benchmark test for big data, for the relational data generated by TPC-DS; we also select a graph data set from the real world. We can numerically compare the advantages and disadvantages between them. Meanwhile, we will also compare the three databases: relational database, graph database, and multi-model database in new trendy perspectives such as visualization, database science, and Cloud service.

The results show that databases that natively support graph data storage have significant performance advantages when dealing with graph algorithms, but converting relational data into graph type and importing it takes a lot of time. A relational database is faster than graph and multi-model databases when importing data. Still, it does not perform as well as graph databases when faced with multi-table join workloads. The multi-model database does not have significant performance advantages over the other two, but the support for a unified query language and native multi-data type storage is impressive.

Chapter 2 of this paper introduces the research background and related work, provides a detailed description of the two data sets involved in the experiments. Chapter 3

shows our proposed methods and algorithms on how to conduct the experiments. Chapter 4 shows the specific implementation of the experiments and the results, also an unsupervised learning-based clustering analysis of the results. Chapter 5 presents our analysis of the experimental results, as well as a comparison of the three databases.

2. Background and Related Work

2.1 Relational Database

As early as the 1960s, a kind of special computer program was proposed to perform the task of storing and managing data in a computer and help users to organize and structure their data, namely a database management system (DBMS) [15]. Twenty years later, in the 1980s, relational database management systems (RDBMS) became popular for a long time. Since then, RDBMS, which organized data in a simple tabular format, has become a database standard. Accordingly, a relational database language, SQL (Structured Query Language), was proposed to handle relational databases [15].

RDBMS stores data in a database object called a table which consists of many columns and rows, where a row is a piece of data, and a column is a corresponding name or label. A database can consist of multiple tables, and some constraints are followed, which are imposed on data columns so that the database can ensure the legality and accuracy reliability of the data in the table. Some common constraints:

UNIQUE - All values in this column are unique and cannot be repeated;

PRIMARY Key - Uniquely identifies each row in the table. A table should only have one primary key, but a primary key does not necessarily contain only one column;

FOREIGN Key - A foreign key is the primary key of another table and is used to establish relationships between different tables;

NOT NULL - The value of the column cannot be null.

In addition, the RDBMS will ensure the integrity of the data. There are no exact duplicate rows in the table, protect the rows used by other records, or some rules defined by the users.

There are many kinds of RDBMS commonly used. MySQL is one of the most popular open-source SQL databases, widely used in web development. PostgreSQL is an open-source SQL database not controlled by any institution or company, typically used for various open-source application development has a large and active developer community, also used in our experiment. There are also some commercial versions of RDBMS, such as Oracle Database, SQL Server, etc., that charge for using the service.

2.2 Graph Database

Graph database management systems (GDBMS) are designed to store and process graph data that could not be well represented by tables natively. The components of a graph are edges and vertices, and it is a data structure that focuses on node-to-node relationships.

Most network data is inherently graphical in structure, such as social networks, traffic road networks, collaborative networks, etc. We can seamlessly model it through graph databases. And some data stored in tabular form can also be represented as graphs after transformation. We can consider a row in a table as a node and then create edges based on the relationship between tables, so graph databases are not only applicable to web data. The characteristics of graphs determine that they are more suitable for modeling data where the connections between data points or the topology are more important [23]. In practical applications, graphs are divided into attribute and non-attribute graphs. The nodes or edges of an attribute graph will store certain information, and similarly, if there are directions or weights for the edges, they will be called directed or weighted graphs [1].

Neo4j is a java-based GDBMS that provides a graphical interface and introduces a declarative query language for attribute graphs, Cypher [14]. Neo4j uses native graph storage and is utilized in many industries, not only with Cypher to model, query, and modify complex data, but also to implement other definitions and even provide plug-ins for data science and graph algorithms, such as graph machine learning [13].

2.3 Multi-model Database

A common point of RDBMS and GDBMS is that they can only store one type of data, while multi-model database management systems (MMDBMS) is proposed to achieve storage of multiple data models, such as documents, graphs, key-value stores, within one integrated backend, and to perform operations such as query and modification. Although the number of multi-model databases has been growing, many of them are not currently mature enough to become a commonly used storage solution. There are still many fields to improve for MMDBMS, such as in data representation: coexistence of single-model and multi-model models; query language: how to design a query language with reasonable syntactic semantics across storage models; switching storage between different models for the same data set, or support for data science features [17, 20].

ArangoDB is a native multi-model database management system with three storage models, document, graph, and key-value. ArangoDB proposes a declarative query language AQL (ArangoDB Query Language), common across the three storage schemas and similar to SQL. AQL allows different types of storage to be involved in a single query.

2.4 Database Benchmarks

RDBMS Benchmark

RDBMS has a long history and is widely used, so there are many benchmark tests for it, the most common of which is the series proposed by TPC. This series of benchmarking involves transaction processing, big data, IoT, virtualization, decision support, artificial intelligence aspects. In this thesis, we use the TPC-DS test, which is a benchmark for decision support [31]. It involves modeling, maintaining, and querying the data of a decision support system and provides a holistic assessment of the performance of this type of system. Specific tests include query execution time (single-user), query throughput (multi-user), and measurement of hardware usage under several tasks and given constraints. TPC-DS is designed to provide a database benchmark reference for emerging areas such as Big Data in the era of data explosion [8, 35].

GDBMS Benchmark

Similar to RDBMS, for graphs, there is also a popular test series, the Linked Data Benchmark Committee (LDBC). It is an organization that wants to set the standards for graph benchmarking and hopes to use this to advance the communication and development of the graph community. There are currently three standards for graph-related evaluation, a benchmark for graph algorithms; a semantic data testing benchmark based on RDF, and a comprehensive evaluation for GDBMS based on social network data in a hypothetical system for interaction and business intelligence; this system is still under development [12].

MMDBMS Benchmark

There is no absolutely prevalent benchmarking framework for multi-model databases, but some mainstream ones, such as Unibench. UniBench is a complete MMDBMS benchmark that contains an internal data generator that can generate a multi-model data set (JSON, XML, key-value, table, and graph) and a set of social commerce-based workloads, including multi-model queries and transactions [42, 41].

In addition, some multi-model databases officially do some benchmark tests to highlight their database performance excellence; for example, ArangoDB made a performance comparison involving four databases in 2018 [4], and the test results are shown in the Figure 2.1.

NoSQL Performance Benchmark 2018
 Absolute & normalized results for ArangoDB, MongoDB, Neo4j and OrientDB

	single read (s)	single write (s)	single write sync (s)	aggregation (s)	shortest (s)	neighbors 2nd (s)	neighbors 2nd data (s)	memory (GB)
ArangoDB 3.3.3 (rocksdb)	100%	100%	100%	100%	100%	100%	100%	100%
	23.25	28.07	28.27	01.08	0.42	1.43	5.15	15.36
ArangoDB 3.3.3 (mmfiles)	102.16%	102.55%	103.89%	102.40%	816.06%	122.07%	99.32%	92.87%
	23.76	28.79	29.37	1.10	3.40	1.75	5.12	14.27
MongoDB 3.6.1 (Wired Tiger)	422.38%	1123.36%	1652.09%	136.65%		518.83%	192.88%	50.64%
	98.24	315.33	466.99	1.47		7.42	9.94	7.70
Neo4j 3.3.1	153.65%		149.37%	203.45%	199.94%	208.96%	214.22%	240.68%
	35.73		43.22	2.18	0.83	2.99	11.04	37.00
Postgres 10.1 (tabular)	231.17%	129.03%	127.70%	29.62%		307.96%	76.87%	26.68%
	53.77	36.22	36.10	0.32		4.41	3.96	4.10
Postgres 10.1 (jsonb)	135.96%	104.34%	101.55%	204.55%		292.57%	126.14%	35.36%
	31.62	29.29	28.70	2.20		4.19	6.50	5.43
OrientDB 2.2.29	198.84%	110.37%		2526.29%	12323.67%	636.45%	400.97%	107.04%
	46.25	30.98		27.19	51.34	9.11	20.67	16.45

Figure 2.1: NoSQL Performance Benchmark by ArangoDB 2018.

2.5 Comparison of Databases

Comparisons for several DBMSs are a common area of research. One study for an actual healthcare data application scenario [34] stated that compared to PostgreSQL, Neo4j provides better data visibility and is cleaner in query language and faster in query speed, but data modeling requires a lot of extra work. On the other hand, a study [29] has proposed a benchmarking architecture based on LDBC-SNB and evaluated a range of RDBMS and GDBMS, with the result that no performance advantage was found for GDBMS. Some researchers have compared Neo4j and MySQL through the data provenance perspective and concluded that there are mutual advantages and disadvantages to their performance [38]. There are also studies aiming to design a unified benchmark test standard for both RDBMS, GDBMS and MMDDBMS [8].

2.5.1 Data Science Support

Unfortunately, PostgreSQL, an old and classic relational database, does not support or include machine learning internally. Some researchers embed data mining algorithms such as rule induction and decision trees into SQL extensions [39]; some scholars integrated PostgreSQL with interactive data science platforms in designing a platform [43], but none of these is a well-developed data science system. But on the other hand, PostgreSQL and Python can be seamlessly integrated so that we can seamlessly explore data science features on the data set through powerful, versatile, and reputable libraries such as Scikit-Learn [30] with the support of Python. In this case, the database is only playing a traditional database role.

ArangoDB proposes a tool called ArangoML, which is a pipeline focusing on meta-

data engineering for machine learning tasks, and it can store the metadata involved in the middle of the whole process of machine learning tasks, provide monitoring and auditing, and ensure repeatability [2]. Its role is shown in Figure 2.2. And ArangoDB contains a series of operational functions that can also pre-process the data in the machine learning feature engineering and implement some graph algorithms to complete the response to the corresponding needs.

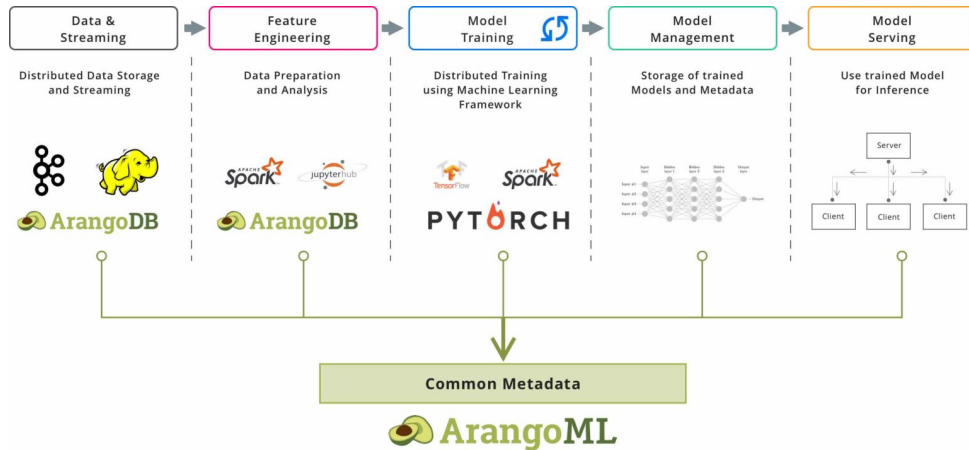
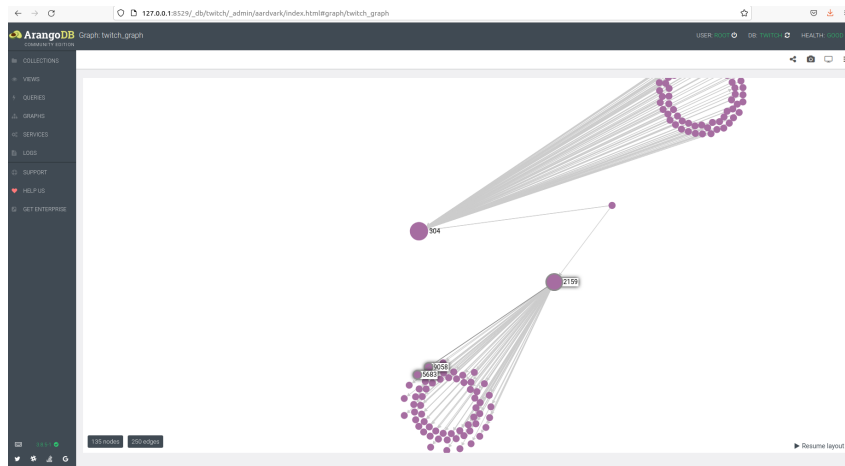


Figure 2.2: The Role of ArangoML in Machine Learning Pipeline

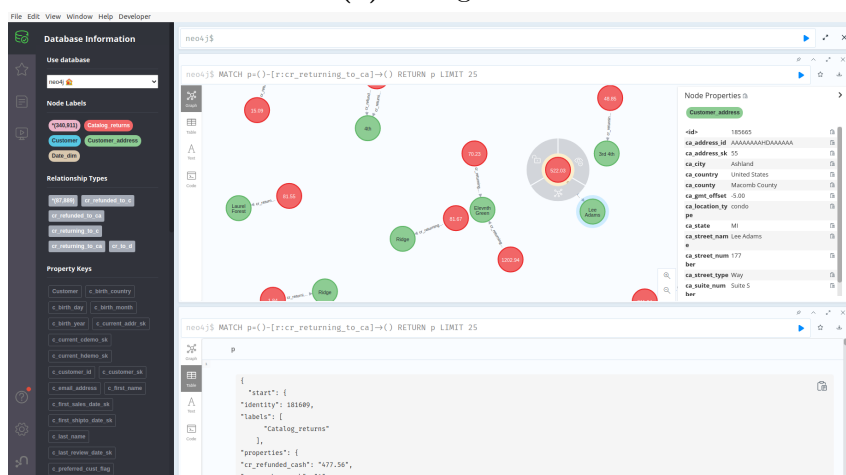
Neo4j, as a pure graph database, provides a complete cycle data science framework that allows model training and deployment in its single environment; it offers up to 65 scalable algorithms and supervised machine learning methods[26]. Neo4j’s data science features have been used in many areas of industry. Examples include recommender systems, fraud detection, data privacy compliance, biopharmaceuticals, etc [26].

2.5.2 GUI and Graph Visualization

All three databases support visual interfaces, with PostgreSQL through pgAdmin, ArangoDB through a browser and port mapping, and Neo4j with powerful desktop software. As shown in Figure 2.3, ArangoDB and Neo4j also support graph visualization, with Neo4j’s graph-based interface being richer in reality and ArangoDB being slightly more rudimentary.



(a) ArangoDB



(b) Neo4j

Figure 2.3: Visual interface for ArangoDB and Neo4j

2.5.3 Toward Cloud Database

PostgreSQL is an old-school RDBMS, and almost all Cloud providers choose to support it on the cloud, including AWS, Google Cloud, and Microsoft Azure, all support deploying PostgreSQL on their clouds. Users can use any standard SQL client application to remotely access and send commands from client computers for any of these Cloud providers. This includes pgAdmin, which we talked about in the previous section, or using the `psql` command-line tool inside PostgreSQL.

ArangoDB launched a cloud database service called Oasis, and users can choose one of AWS, google cloud, or Microsoft Azure as the running vehicle. Oasis comes with its API, which, like a standard API, enables users to control all resources within Oasis by writing scripts. For example, you can start the deployment of ArangoDB Oasis and perform a series of operations during the DevOps process [28].

Similar to ArangoDB, Neo4j has its own fully managed Cloud service called AuraDB,

allowing users to choose a Cloud service provider from AWS and Google Cloud. It is also possible to migrate across Clouds among service providers. It's worth discussing that AuraDB is permanently free for small networks, while ArangoDB only offers a 14-day free trial, and PostgreSQL has a 12-month free trial (AWS) or 300 USD credit (Google Cloud) or 12-month free trial plus 200 USD credit (Azure) depending on the Cloud provider.

3. Methodology

We chose three databases, PostgreSQL, Neo4j, and ArangoDB, as representatives of the relational, graph, and multi-model databases, respectively, for the comparison. In this chapter, we present the details of the comparison methods.

3.1 Data Sets

This thesis involves two datasets, one is a tabular data TPC-DS data set while the other is a graph data based on the real Twitch social network. In this chapter, we will introduce them separately in detail.

3.1.1 TPC-DS Set (Table-like)

The TPC-DS toolkit [24] provides a data schema that models a merchant’s sales and returns process for three sales channels: catalog, stores, and the Internet. For each table, there are different columns, and each column is named uniquely and starts with a lowercase acronym of the table name. Some columns are individual primary keys for a table, and some tables have a primary key that is a composite of several columns. Some columns are foreign keys, that is, keys that establish joins between different tables. In contrast, some keys are called business keys in the data warehouse schema, which are neither primary nor foreign keys. They are used only for data insertion and update during data maintenance.

For the values in the column, there are four different data types, the first is **Identifier**, which means that this column can represent any value generated by this column; the second is **Integer**, which means an integer value from -2^{63} to $2^{64} - 1$; the third is **Decimal (d,f)**, which as the name implies is a decimal value, where **d** represents digits and **f** is the number of decimal places, the decimal value may be precise as defined or maybe other decimals in the range; the last is **Char (N)**, which is a string of length **N** that fill the column.

Figure 3.1 is the definition description of the table `Catalog Sales` [35]. We can see that this table has a composite primary key, consisting of `cs_item_sk` and `cs_order_sk`, while the second column is a different data type. We can also see that many of the keys

are foreign keys that establish the relationship between the table and other tables. `cs` is the initial lowercase abbreviation for `Catalog Sales`.

Column	Datatype	NULLs	Primary Key	Foreign Key
<code>cs_sold_date_sk</code>	identifier			<code>d_date_sk</code>
<code>cs_sold_time_sk</code>	identifier			<code>t_time_sk</code>
<code>cs_ship_date_sk</code>	identifier			<code>d_date_sk</code>
<code>cs_bill_customer_sk</code>	identifier			<code>c_customer_sk</code>
<code>cs_bill_cdemo_sk</code>	identifier			<code>cd_demo_sk</code>
<code>cs_bill_hdemo_sk</code>	identifier			<code>hd_demo_sk</code>
<code>cs_bill_addr_sk</code>	identifier			<code>ca_address_sk</code>
<code>cs_ship_customer_sk</code>	identifier			<code>c_customer_sk</code>
<code>cs_ship_cdemo_sk</code>	identifier			<code>cd_demo_sk</code>
<code>cs_ship_hdemo_sk</code>	identifier			<code>hd_demo_sk</code>
<code>cs_ship_addr_sk</code>	identifier			<code>ca_address_sk</code>
<code>cs_call_center_sk</code>	identifier			<code>cc_call_center_sk</code>
<code>cs_catalog_page_sk</code>	identifier			<code>cp_catalog_page_sk</code>
<code>cs_ship_mode_sk</code>	identifier			<code>sm_ship_mode_sk</code>
<code>cs_warehouse_sk</code>	identifier			<code>w_warehouse_sk</code>
<code>cs_item_sk (1)</code>	identifier	N	Y	<code>i_item_sk</code>
<code>cs_promo_sk</code>	identifier			<code>p_promo_sk</code>
<code>cs_order_number (2)</code>	identifier	N	Y	
<code>cs_quantity</code>	integer			
<code>cs_warehouse_cost</code>	decimal(7,2)			
<code>cs_list_price</code>	decimal(7,2)			
<code>cs_sales_price</code>	decimal(7,2)			

Figure 3.1: Definition of Table `Catalog Sales`.

The TPC-DS data schema contains a total of 24 tables, seven of which are fact tables:

- For each of the three sales channels, two fact tables correspond to sales and returns information:
 - Store Sales (ss):** each row represents a sales record through the store sales channel. This table has a composite primary key consisting of `ss_item_sk` and `ss_ticket_number`, and 9 foreign keys pointing to other tables;
 - Store Returns (sr):** each row is a record of items sold through the store sales channel but were returned. This table also has a composite primary key, consisting of `sr_item_sk` and `sr_ticket_number` and 10 foreign keys;
 - Catalog Sales (cs):** similar to **Store Sales (ss)**, except that this table represent records sold through catalog channels and have 17 foreign keys;
 - Catalog Returns (cr):** records of catalog-sold and returning items with 17 foreign keys;
 - Web Sales (ws):** similar to **Catalog Sales (cs)**, this table represent records sold through online channels and have 17 foreign keys;
 - Web Returns (wr):** records of sold online and returning items with 17 foreign keys.
- The one table which stores inventory information for the two sales channels catalog and web called:
 - Inventory (inv):** each row indicates the number of specific items available in a

particular warehouse for a given week. This table has only 4 columns, `inv_date_sk`, `inv_item_sk`, `inv_warehouse_sk` three columns form a composite primary key, and they are also three foreign keys.

- For each of the three sales channels, there are two fact tables that correspond to sales and returns information:

The remaining 17 tables are dimension tables, which are related to the tables of the three sales channels, namely: Store (s), Call Center (cc), Catalog Page (cp), Web Site (web), Web Page (wp), Warehouse (w), Customer (c), Customer Address (ca), Customer Demographics (cd), Date Dim (d), Household Demographics (hd), Inventory (inv), Inventory (inv), Item (i), Income Band (ib), Promotion (p), Reason (r), Ship Mode (sm), Time Dim (t).

So far we have introduced all the tables, which have complex relationships with each other, and for the seven fact tables we show their Entity Relationship Diagram (ER diagram). The six tables for sales and returns are very similar, as shown in Figure 3.2, the ER diagram for Catalog Sales and Catalog Returns respectively, the Appendix A.1 on page 47 contains the ER diagram for remaining 4 tables, and Figure 3.3 is the ER diagram for Inventory [35].

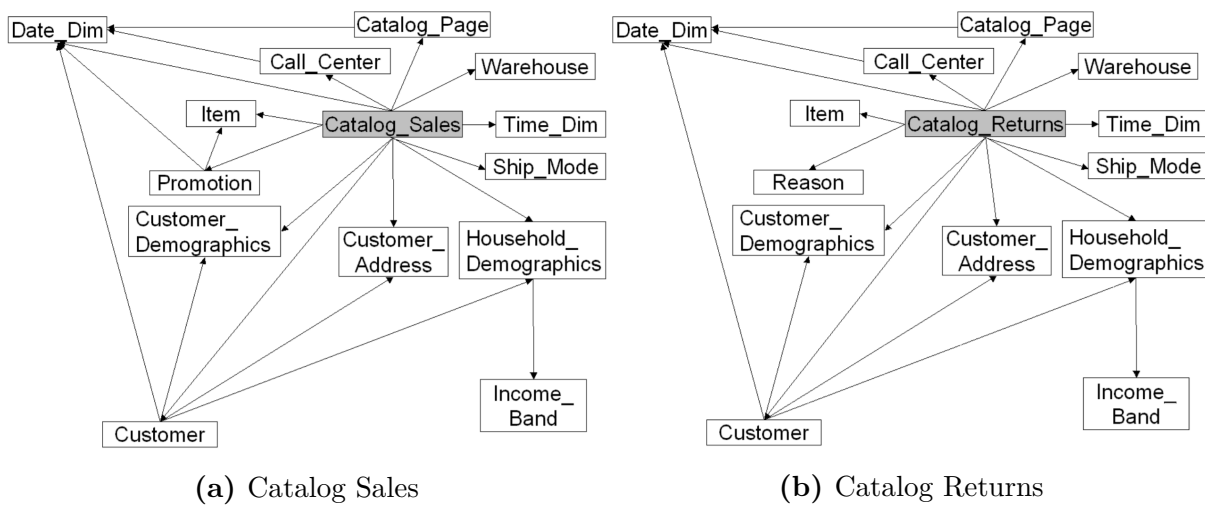


Figure 3.2: ER-Diagrams of Catalog Sales/Returns

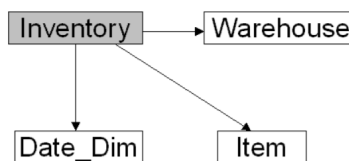


Figure 3.3: ER Diagram of Inventory.

3.1.2 Twitch Set (Graph-like)

We have selected a set of undirected and unweighted graphs from Stanford Large Network Dataset Collection: SNAP which is a network of Twitch (a video live streaming platform) users to users who stream in the same language [19, 33]. Nodes represent different players, and if two players have a friend relationship with each other, then an edge will hold accordingly. Although edges have no characteristics, each node has attributes that are extracted from the player’s characteristics, the time spent online, and whether they are mature. These social networks were collected in May 2018.

The authors [33] propose that these networks can be used for transfer learning applications, as well as for supervised tasks related to binary node classification, such as predicting whether a user uses a selected language or not. For this data set, the attribute information can be used to predict new relationships between nodes and the location of nodes [32]. Some researchers [10] have also used Twitch data to conduct interesting studies, such as whether each game attracts an audience at similar points and how closely/far from each other different mainstream gamer subcultures are connected.

Table 3.1: Dataset Statistics

	DE	EN	ES	FR	PT	RU
Nodes	9498	7126	4648	6549	1912	4385
Edges	153138	35324	59382	112666	31299	37304
Density	0.003	0.002	0.006	0.005	0.017	0.004
Transitivity	0.047	0.042	0.084	0.054	0.131	0.049

Table 3.1 is the statistics of the graphs corresponding to the six different languages [19, 33]. Each node has six attributes: `id`, `days`, `mature`, `views`, `partner`, `new_id`. `mature` and `partner` are Boolean variables, the rest are integer numbers. For each language, the data set consists of two CSV files, where the *edge* CSV file has only two columns, the first column is `from` and the second column `to`, and each line is the `new_id` of the two corresponding nodes, while the other *target* CSV file stores the six attributes of the node, where the `new_id` is corresponding to the edge file.

3.2 Quantitative Analysis

Performance is a core metric for comparing different databases, mainly in terms of their execution time when faced with different tasks. We used two different types of data sets, a table-like data set and a graph-like data set, to adapt to three databases and compare the performance.

We selected 14 evaluation tasks for the table-like data, while for the graph-like data, we selected three. For each evaluation task, the execution time of ten consecutive runs will be recorded, starting from the first, and the median, mean and minimum values were taken for one task.

3.2.1 Table Data

Storage Schema

- For Postgresql, TPC-DS toolkit [24] provides a SQL file that defines the storage schema for the relational database, defining constraints and primary keys, etc. We just need to import the 25 tables as needed.
- For ArangoDB, similarly, as a multi-model database, it also natively supports document type file storage, and we can directly store 25 tables in collections on demand.
- For Neo4j, as a graphical database, it does not support the direct import of relational or table data. Refer to the paper by Yijian and other researchers [8] and the guidance for migrating relational databases to graph databases provided in Neo4j’s developer documentation [27]. We propose a method to store this tabular data in Neo4j: import the desired table as a node first, where each row in each table is a node, and the name of the table is the label of that node, and the attributes of the node have column attributes defined for each row. After importing all the nodes, we then create the edges without any attribute based on foreign key dependencies, as shown in the algorithm in Algorithm 1. The Appendix A.2 on page 48 contains the edge mapping and naming samples.

Data Selection

Six of the seven fact tables are comprised of sold and returned records from three different sales channels, so they have strong similarities. Due to the limitation of my experimental equipment performance, we selected the data of two of the sales channels and a part of the definition tables to form the final table data of this experiment, a total of 14 tables, the ER relationship is shown in Figure 3.4. The four nodes in the blue dashed box represent the four fact tables selected for this time, while the red dashed boxes are the three similar definition tables that share relationships with all other nodes (tables).

However, since the Neo4j import took too long in the experiment, we performed a secondary extraction, involving only five tables, with the ER diagram shown in Figure 3.5, and the same subset of data was used in PostgreSQL, ArangoDB, and Neo4j if they have the evaluation tasks (Task 1 to Task 10 in the Evaluation Tasks Section 3.2.1)

Algorithm 1: Relational Data to Graph Storage

```

Input: Table_set
 $N \leftarrow \text{length}(\text{Table\_set});$ 
Create Nodes:
for  $k \leftarrow 1$  to  $N$  do
   $N\_row \leftarrow \text{length}(\text{table}_k);$            /* Get row number of table_k */
   $N\_col \leftarrow \text{width}(\text{table}_k);$          /* Get column number of table_k */
  for  $j \leftarrow 1$  to  $N\_row$  do
    create a node with the label:  $\text{name}(\text{table}_k);$ 
    for  $m \leftarrow 1$  to  $N\_col$  do
      add property  $\text{Column}(\text{table}_k)[m]$  to node
    end
  end
end
Add Edges:
for  $k \leftarrow 1$  to  $N$  do
   $N\_col \leftarrow \text{width}(\text{table}_k)$ 
  for  $m \leftarrow 1$  to  $N\_col$  do
    if  $\text{Column}(\text{table}_k)[m]$  is a foreign key then
      for each node  $V$  in  $\text{table}_k$  do
        create a edge from  $V$  to possible node from the foreign table
        based on property matching
      end
    end
  end
end
end

```

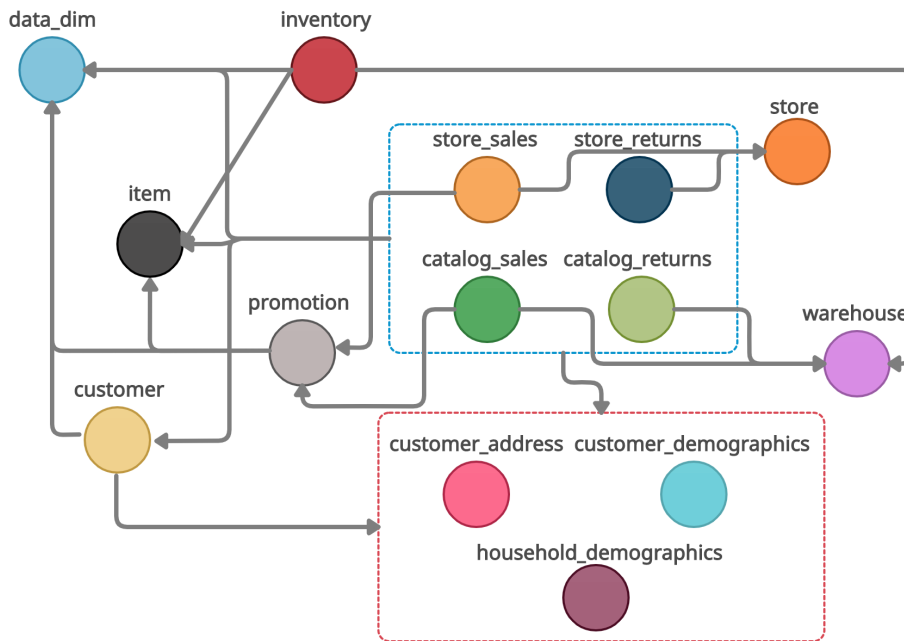


Figure 3.4: Selected TPC-DS data ER Diagram.

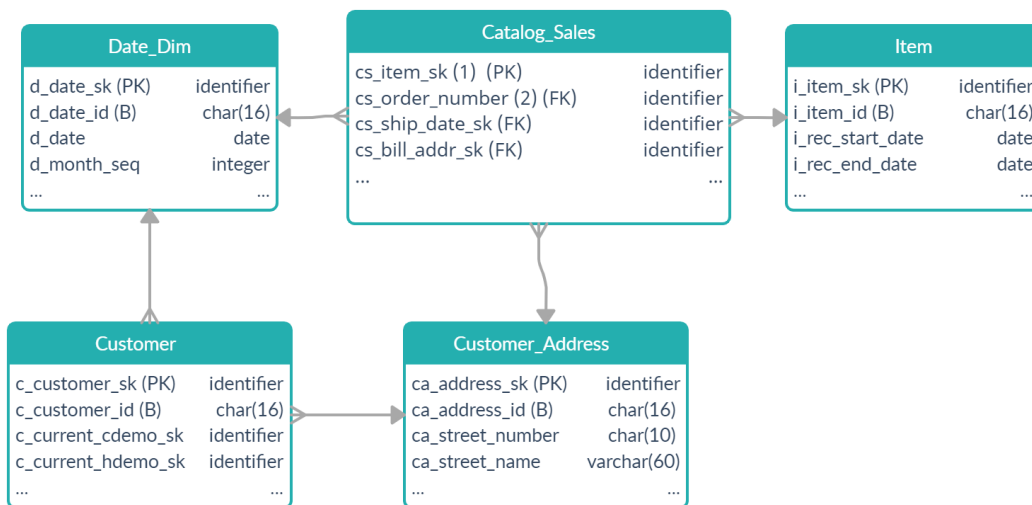


Figure 3.5: 5-table TPC-DS data ER Diagram.

Evaluation Tasks

SQL, as the originator of the database query language, supports very rich operations, while AQL, on the other hand, as a derivative of SQL [3], is as close as possible to SQL’s functionality but does not entirely cover SQL’s functionality. Cypher, as a graph query language, has a very different usage from SQL and AQL, and all its logic is based on graphs. The most basic use of the database can be summarized by the word *CRUD*, that is, Create (C), Read (R), Update (U), Delete (D). Some researchers have proposed

some *CRUD* testing tasks for NoSQL databases [36]. Considering the above, we propose ten basic query tasks that can compose complex statements and cover most of the tasks in production work. In addition, TPC-DS provides 100 query statements by itself, and we selected 4 of them and translated them into AQL and Cypher languages for cross-database performance comparison. We also measure a total of fourteen different tasks multiple times to see if the query is affected by repeated execution. We take the average (Ave), median (Med), fastest, slowest, range, and the standard deviation for each query's execution time.

$$Ave(TQ) = \frac{TQ_1 + TQ_2 + \dots + TQ_N}{N}$$

$$Med(TQ) = \begin{cases} TQ_{\frac{N+1}{2}} & \text{if } N \text{ is odd} \\ \frac{TQ_{\frac{N}{2}} + TQ_{\frac{N}{2}+1}}{2} & \text{if } N \text{ is even} \end{cases}$$

$$Range(TQ) = Max(TQ) - Min(TQ)$$

where TQ_n is the execution time of query Q for the n th time and N is the number of execution times for query Q .

The specific tasks are defined as follows:

1. Create:

- | | |
|------------|--|
| PostgreSQL | Insert 1 row on the <code>customer</code> table with only <code>c_customer_sk</code> and <code>c_customer_id</code> at one time, 100000 rows in total; |
| ArangoDB | Insert a document to the <code>customer</code> collection at one time with only <code>c_customer_sk</code> and <code>c_customer_id</code> attributes at a time, 100000 documents in total; |
| Neo4j | Add a <code>customer</code> node with <code>c_customer_sk</code> , <code>c_customer_id</code> as node properties at a time, 100000 nodes in total. |

2. Read:

- | | |
|------------|---|
| PostgreSQL | Select all rows in <code>customer</code> table; |
| ArangoDB | Return all documents in <code>customer</code> collection; |
| Neo4j | Return all nodes with label <code>customer</code> . |

3. Update:

- | | |
|------------|---|
| PostgreSQL | Update the <code>c_birth_day</code> column of the 100000 rows created in task Create from null to 1, one row at a time; |
| ArangoDB | Update the <code>c_birth_day</code> attribute of the 100000 documents created in task Create from null to 1, one document at a time; |
| Neo4j | Update the <code>c_birth_day</code> node property of the 100000 nodes created in task Create from null to 1, one node at a time. |

4. Delete:

- PostgreSQL Delete all 100000 rows created previously, one row at a time;
- ArangoDB Delete the the 100000 documents created previously, one document at a time;
- Neo4j Delete the 100000 nodes created previously, one node at a time.

5. Projection:

- PostgreSQL Select first three columns from `customer` table;
- ArangoDB Return the same three attributes from `customer` collection;
- Neo4j Return the same three node properties of nodes with label `customer`.

6. Sorting:

- PostgreSQL Select all rows from `customer` table, order by age, i.e. birthday, with month and year;
- ArangoDB Return all documents from `customer` collection, order by birthday, with month and year;
- Neo4j Return all nodes with label `customer`, order by order by birthday, with month and year.

7. Counting:

PostgreSQL/ArangoDB/Neo4j:

Count how many customer were born in January and February.

8. Grouping:

PostgreSQL/ArangoDB/Neo4j:

Count how many customer were born in the same year, month.

N.B. For Neo4j, since there is no `Group By` in Cypher, we return the year and month as combination and calculate the frequency instead.

9. Min/Max:

For three databases:

Report both the maximum and minimum born year of customer.

10. Join:

PostgreSQL/ArangoDB/Neo4j:

Return inner join of all columns/attributes/node properties from `catalog_returns` and `data_dim` where `cr_returned_date_sk` equals to `d_date_sk`

11. TPC-DS Q15:

Report total sales via catalog for customers in chosen geographic areas or for customers who made significant acquisitions in a given quarter (2) of a given year (2001).

12. TPC-DS Q20:

Calculate the total income and the ratio of total income to income from the specified item category (Sports, Books, Home) for a time period (from February 20, 1999 and inside 1999)

13. TPC-DS Q29:

Report all items sold via store channel in a specific month (September) in a specific year (1999) and returned within the following 6 months from the purchase in the same year (1999) and these items have been purchased again by the same customer within the next three years but via the catalog sales channel.

After selecting the items, calculate the total number of items sold via the store sales channel, the number returned, and the number purchased via the catalog sales channel and group all the results by store and item.

14. TPC-DS Q72:

Calculate and report the amount of sales with promotions or not for each warehouse, item and weekly mix in 1999.

The Task 11 to Task 14 are not applicable for Neo4j due to the data selection.

3.2.2 Graph Data

Storage and querying of graph data in relational databases have been an interesting and heavily researched topic, researchers focus on graph matching queries over SQL, but these are often accompanied by a relatively complex schema for storing graph data [21, 18]. In this study we stored graph data in a simple and straightforward schema and propose three evaluation tasks to compare the performance. Similar to the table data, we also test the different tasks several times and calculate some indices.

Storage Schema

- For Postgresql, We store the edge csv and node property csv as two separate tables, table `link` and table `account`, then set `new_id` column of table `account` as the primary key and foreign key to establish a relation with the table `link`, while the

two columns in table `link`, `id_1` and `id_2`, consist of a composite primary key. Figure 3.6 shows the corresponding entity relationship diagram.

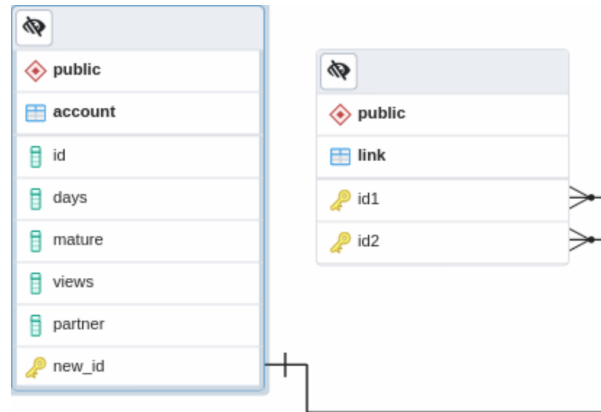


Figure 3.6: Entity Relationship Diagram of twitch data set.

- For ArangoDB, the graphical functionality is similar to that of a graph database, for each document in a collection of stored nodes, a unique `_id` attribute is automatically stored. To establish a relationship (i.e., an edge) between two documents (i.e., nodes), both `_id` attributes are stored in special edge documents called `_from` and `_to` attributes, thus forming a directed edge between two arbitrary nodes. The edges are then stored in another special collection.

ArangoDB achieves efficient and scalable graph query performance by using special hash indexes for the `_from` and `_to` attributes (i.e., edge indexes). This allows constant lookup times. Using edge indexes, ArangoDB can handle graph queries very efficiently. Figure 3.7 shows two collections of twitch, `account` storing node attributes and `links` is for edges.

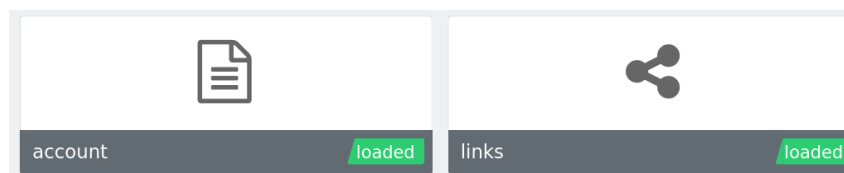


Figure 3.7: Twitch collections in ArangoDB

- For Neo4j, as a native graph database, we can import the twitch data directly as a graph, where the nodes have the corresponding properties.

Evaluation tasks

1. N-hop Friends:

An N-hop friends problem, understood theoretically, is to find all nodes that are N or less distant from the initial node. It is a ubiquitous algorithm, some algorithms, such as machine learning k-nearest neighbors, also use a similar principle, but there are some differences in the distance calculation. This algorithm is very useful for search and basically, it is a traversal of the graph so we can use it to evaluate the performance of different databases.

Neo4j has a native Breadth-First Search (BFS) API called `gds.alpha.bfs.stream` to achieve this function, as results, it will return all the nodes within a certain distance and the route how to get there from starting node. This function also supports multiple termination conditions, for example, based on reaching one of a given set of target nodes, reaching the maximum depth (hops), or having no more cost budget for a given traversal relationship.

Similarly, ArangoDB, a database that natively supports graph structure storage, also supports graph traversal functionality, i.e., how to accomplish N-hop friends search. After searching all the compliant vertices visited in the running, the function will return a set containing three items: first, all the vertices visited; second. the edge pointing to it; third, the complete path from the initial node to the visited node as an object, for this object there are two different attributes, edges and vertices, each of them is a list of respective components. These lists will be sorted from the initial node to the last node.

As a traditional relational database, PostgreSQL does not natively support graph traversal operations. Thus, reference related research [11], we propose an algorithm to complete the n-depth traversal of the graph to solve the related problem, as shown in Algorithm2. This function has two inputs, the starting node `node_1` and the depth (hops) `n`, while the return will contain all the nodes that meet the

requirements and the route from the starting node to that node.

Algorithm 2: N-hop friend search in PostgreSQL

Input: *node1*: the starting node, *n*: n hop

```

1 WITH RECURSIVE n_hop_search(
2   node_1,    -- a vertex from graph, the starting node
3   node_2,    -- another vertex from graph
4   hop, -- hop, by default is 1
5   route -- the route from node_1, an array
6 ) AS (
7     SELECT    -- root query
8       graph.node_1,    -- node 1
9       graph.node_2,    -- ndoe 2
10      1 as hop,        -- by default, set hop as 1
11      ARRAY[graph.node_1] as route -- the first route
12 FROM link AS graph -- read edge table link
13 WHERE
14   node_1 = ?          -- starting_node, INPUT
15 UNION ALL
16 SELECT    -- recursive query
17   graph.node_1,    -- node 1
18   graph.node_2,    -- node 2
19   rec_graph.hop + 1 as hop,    -- hop +=1
20   route || graph.node_1 as route -- update the route
21 FROM link AS graph, n_hop_search AS rec_graph -- recursive
22 WHERE
23   graph.node_1 = graph.node_2 -- new node 1 = old node 2
24   AND (graph.node_1 <> ALL(rec_graph.hop)) -- avoid loop
25   AND rec_graph.hop <= n -- n hop, INPUT
26 )

```

2. Shortest Path:

The shortest path problem is easy to understand, i.e., finding the shortest path between two nodes. While Neo4j and ArangoDB still have native support for this task, PostgreSQL needs to be modified from our Algorithm2 by setting *node_2* as the destination node, removing the maximum depth termination condition, and finally using the `LIMIT 1` to get the first row of the returned table, i.e. the shortest path.

4. Experiments and Results

4.1 Environment and Configuration

The experiments were all conducted on one single laptop, and in order to remain impartial, there was no operation been performed after installing the operating system but only installed the components and software necessary for the experiments. The details of the laptop are described as follows:

Processor (CPU)	AMD Ryzen 5 4600U (6C 12T) @2.1GHz
Memory	16 GB, 3200 MHz DDR4
Operating System	Ubuntu 20.04.3 LTS 64-bit
Storage	Pioneer APS-SE10N-Pcie3*2 M.2 NVMe SSD 256G
Video	AMD Radeon RX Vega 6
Network	100 Mbps Wired Connection

Three database versions and other extensions were used for this experiment, the relevant software versions are as follows:

PostgreSQL	12.9 (pgAdmin4 6.4)
ArangoDB	3.8.5-1 (Vpack 0.1.35, RocksDB 6.8.0)
Neo4j	Community Edition 4.4.1 (Desktop App Version 1.4.12)

As previously stated, we test all evaluation tasks multiple times, recording the execution time for each one. For PostgreSQL, this is done using `psql` within the terminal; for ArangoDB, it is achieved via the web API, and for Neo4j, it is done via its officially provided desktop software. For timing, both ArangoDB and Neo4j automatically report the execution time at the end of each query; in PostgreSQL, we set `\timing` as **ON** to obtain the execution time for each query.

Between tasks or when switching databases we clear the cache through command: `sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"` and `require("@arangodb/aql/cache").clear();`. Run only one database at a time.

4.2 Data Production and Import

4.2.1 TPC-DS Data

We used the data generator `dsdgen` from TPC-DS tool-kit (Version 2.10.0) to generate the data sets for our tests, there is a `Scale Factor` we can set to 1 as the minimum valid input, then a data set of 1G size will be generated, detailed statistics can be found in Table 4.1.

TPC-DS Data 1GB Row Counts [35]		
Table	Average Row Size (b)	Rows
catalog_sales	226	1441548
catalog_returns	166	144067
store_sales	164	2880404
store_returns	134	287514
customer	132	100000
customer_address	110	50000
customer_demographics	42	1920800
date_dim	141	73049
household_demographics	21	7200
warehouse	117	5
inventory	16	11745000
item	281	18000
promotions	124	300
store	263	12

Table 4.1: TPC-DS Data 1GB Statistics

When importing the corresponding nodes into the database, PostgreSQL provides the `psql COPY` command, ArangoDB has the `arangoimport` command, and Neo4j requires script writing to create the nodes. Subsequently, it is also necessary to establish corresponding relationships/edges between nodes with different labels. Using the `date dim` table as an example, PostgreSQL and ArangoDB both took less than 5 seconds to import, while Neo4j took a staggering 1990 seconds, shown in Figure 4.1.

4.2.2 Twitch Data

The Twitch database consists of 6 graphs, and since our task is to evaluate database performance rather than exploring transfer learning, we chose the largest relational graph

```

1 LOAD CSV WITH HEADERS FROM 'file:///date_dim.csv' AS row
2 MERGE (date_dim:Date_dim {d_date_sk: row.d_date_sk})
3 ON CREATE SET
4 date_dim.d_date_id = row.d_date_id,
5 date_dim.d_date = row.d_date,
6 date_dim.d_month_seq = row.d_month_seq,
7 date_dim.d_week_seq = row.d_week_seq,
8 date_dim.d_quarter_seq = row.d_quarter_seq,
9 date_dim.d_year = row.d_year,
10 date_dim.d_dow = row.d_dow,
11 date_dim.d_moy = row.d_moy,
12 date_dim.d_dom = row.d_dom,

```

Added 73049 labels, created 73049 nodes, set 2045372 properties, completed after 1990029 ms.

Table

Code

Figure 4.1: Import date dim Table to Neo4j

of German-speaking players. In total, there are 9498 nodes and 153138 edges.

Unlike the TPC-DS data, there is no defined schema for PostgreSQL out of the box this time, we create two tables and import the node data and edge data separately as described in the previous chapter.

```

1 CREATE TABLE account
2 (
3   new_id      INTEGER,
4   days        INTEGER,
5   mature      BOOLEAN,
6   views       INTEGER,
7   partner     INTEGER,
8   PRIMARY KEY (new_id));

```

```

1 CREATE TABLE link
2 (
3   id1         INTEGER,
4   id2         INTEGER,
5   PRIMARY KEY (id1, id2)
6 );

```

One thing to note about ArangoDB is that its collection for storing edge information is a specialized form, and we need to add the additional option `-create-collection-type edge` when importing edge information. For Neo4j we just need to import the nodes and edges with command `LOAD CSV` and `CREATE` [25].

4.3 Tasks Execution Time

Besides the metrics we defined in Section 3.2.1, We define the following metrics here:

1. **Percentage of range (Perc):**

$$Perc(TQ) = Range(TQ)/Max(TQ)$$

where TQ is the execution time series of query Q and $Range(TQ)$ is the fluctuation range of all the execution times of query Q while $Max(TQ)$ is the slowest of all

execution times. This metric can be used to measure the volatility of continuous queries.

2. Coefficient of Variation (CoV):

$$CoV(TQ) = Std(TQ)/Ave(TQ)$$

where TQ is the execution time series of query Q and $Std(TQ)$ is the Standard deviation of all the execution times of query Q while $Ave(TQ)$ is the average value of all execution times. This is another metric that can also be used to measure the volatility of continuous queries, and it shows the degree of variation associated with the overall mean. Unlike *Perc*, it eliminates the effect of absolute values to some extent and focuses more on the relative intensity of fluctuations, a higher the *CoV*, means the dispersion is greater.

3. Time Reduction value of the 10th verse the 1st (TR(1-10)):

$$TR(1 - 10) = \frac{TQ_{10} - TQ_1}{TQ_1}$$

where TQ_{10} is the execution time of query Q for the 10th time, similarly, TQ_1 is for the first execution. This metric is used to measure the execution time improvement of the database from the first to the tenth query. To eliminate the effect of the absolute value of the different query times, similar to *CoV*, we divide by the first query time spent.

We performed automated tests using scripts to time 16 tasks and three databases, and the execution time results for all tests are in Appendix B. For the calculation of the measurement metrics, we did it with Python and the corresponding libraries, where Table 4.3 shows the first execution time and the fastest execution time for the three databases on 16 tasks, and the best performance among the three is marked with a color. Figure 4.4 shows the comparison of *CoV*.

It is also a good way to perform cross-database comparisons for the same query to perform database performance evaluation; for example, for Table Task 1: Create, Table 4.2 is its performance metrics statistics for on three databases. Figure 4.2 shows a line graph of the change in time spent on ten queries for each of the three databases, noting that the y-axis range is not uniform. Again with this example, we can see from the performance metrics that the average time difference between the three databases for this task is very large, with PostgreSQL taking significantly less time than ArangoDB, which in turn is an order of magnitude faster than Neo4j. On the other hand, by analyzing the ten execution times, we can see that PostgreSQL has a significant degradation between the first and the third time, which means that PostgreSQL is probably doing some optimization or

	Ave	Med	Std	Max	Min	Perc	CoV	TR(1-10)
PostgreSQL	361.3	349.5	40.1	457	324	0.291	0.111	0.232
ArangoDB	8573.4	8570.5	143.5	8829	8367	0.052	0.017	-0.002
Neo4j	18700.7	18959.5	786.5	19890	17014	0.145	0.042	-0.060

Table 4.2: Table Task 1 Metrics.

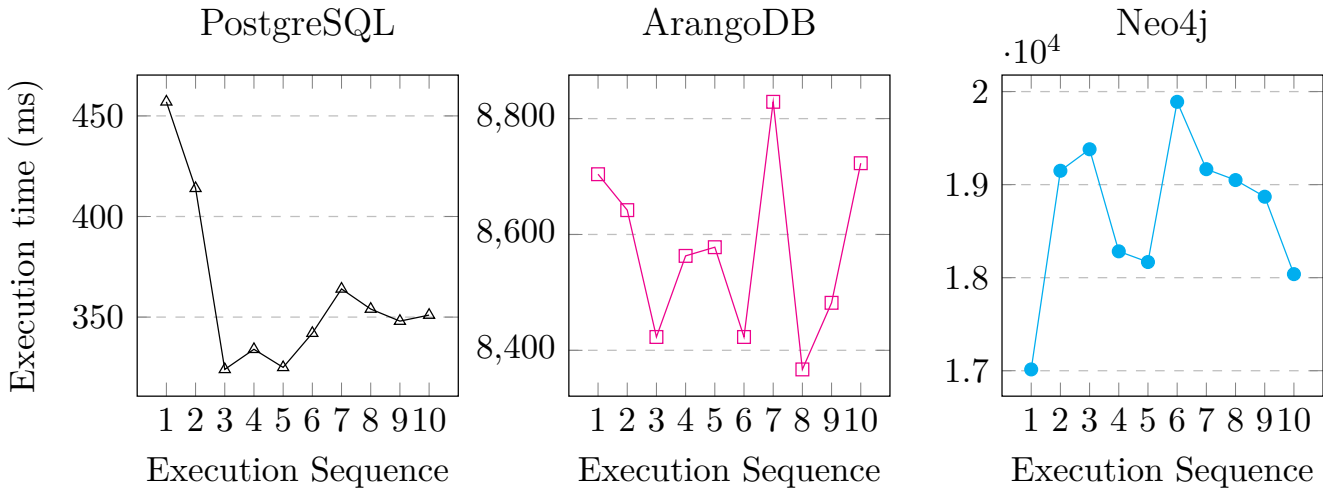


Figure 4.2: Table Task 1 Execution Times

"learning." Figure 4.3 then shows a simultaneous comparison of $3 \times 10 = 30$ execution times. With this figure, we can see that PostgreSQL is in an absolute advantageous position for this task, as we discussed before, but on the other hand, due to the y-axis scaling, we cannot clearly see the comparison within the database itself.

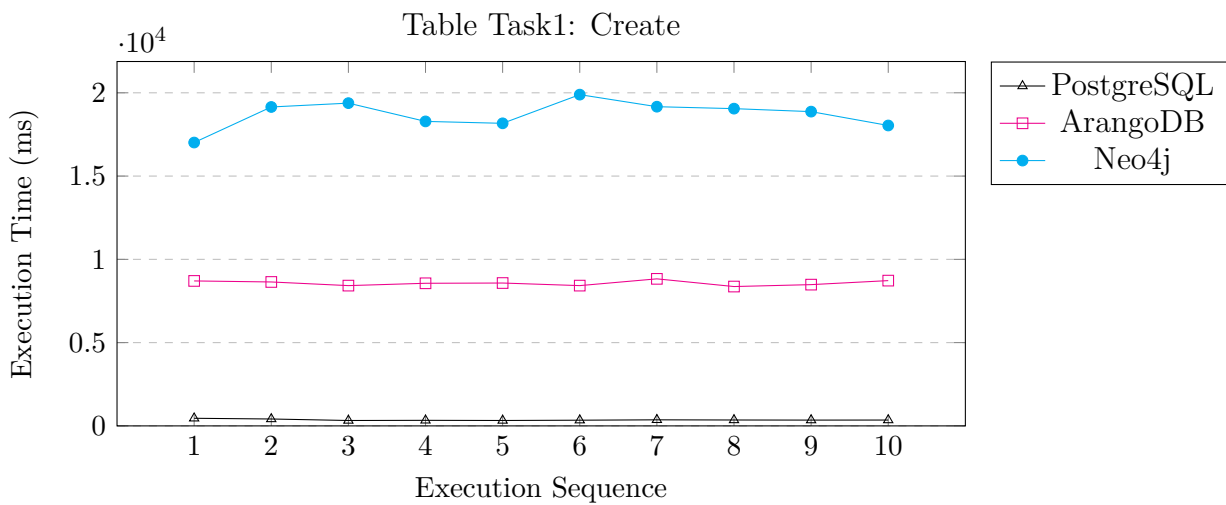
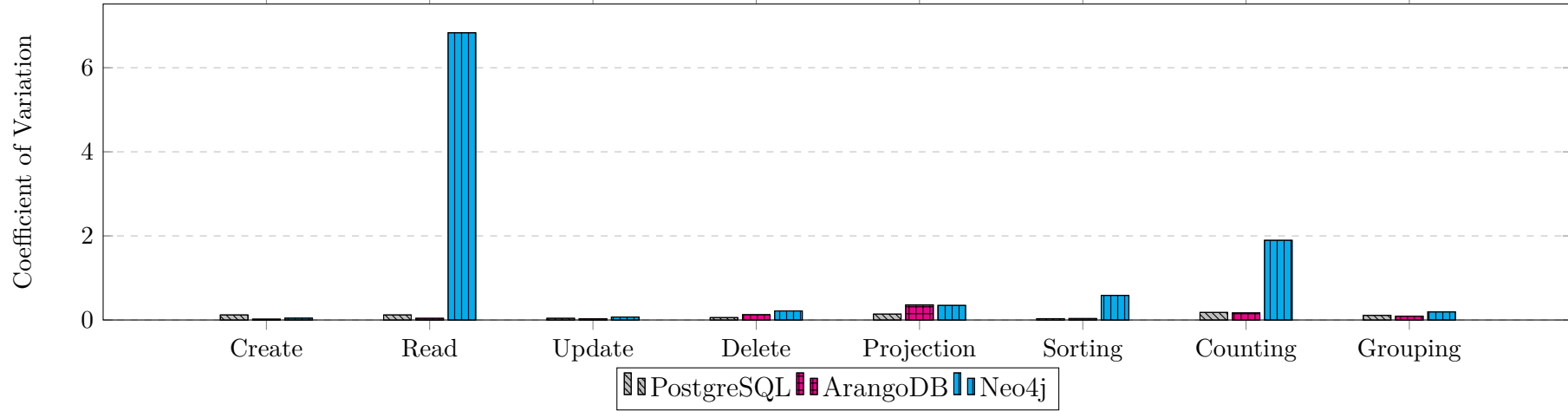


Figure 4.3: Table Task 1 Execution Time Comparison

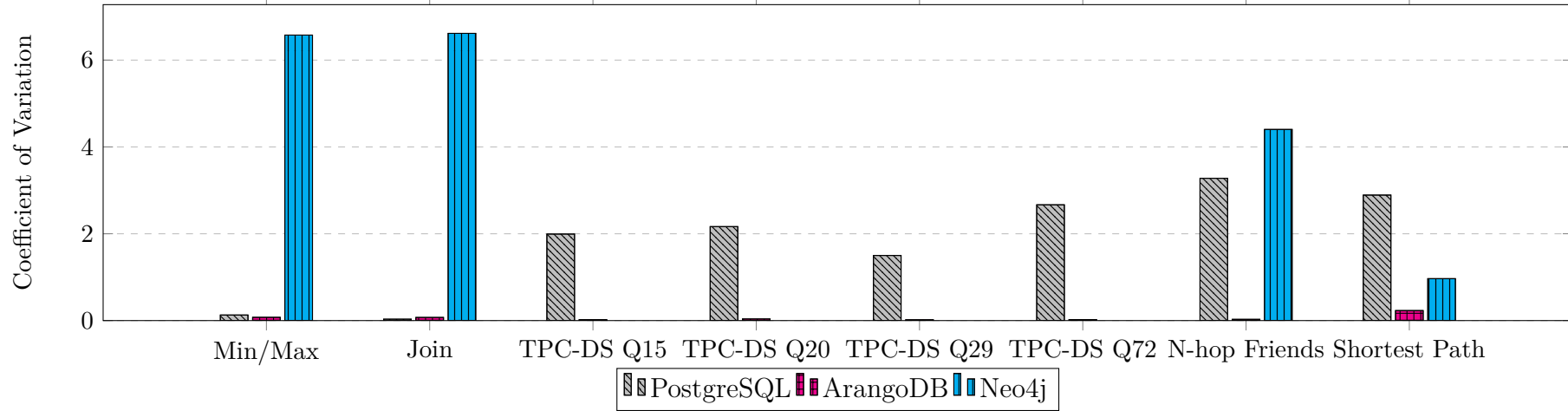
First Execution Time & Minimum Execution Time Comparison

	Pos.(1st)	Ara.(1st)	Neo.(1st)	Pos.(Min)	Ara.(Min)	Neo.(Min)
Table Task 1: Create	457.00	8704.00	17014.00	324.00	8367.00	17014.00
Table Task 2: Read	360.00	663.00	542.00	249.00	583.00	20.00
Table Task 3: Update	765.00	3133.00	524.00	765.00	3015.00	524.00
Table Task 4: Delete	379.00	4728.00	278.00	330.00	4273.00	255.00
Table Task 5: Projection	145.00	864.00	514.00	96.00	354.00	204.00
Table Task 6: Sorting	407.00	632.00	40.00	367.00	632.00	10.00
Table Task 7: Counting	116.00	209.00	42.00	70.00	209.00	5.00
Table Task 8: Grouping	122.00	156.00	42.00	89.00	136.00	42.00
Table Task 9: Min/Max	92.00	275.00	152.00	59.00	217.00	5.00
Table Task 10: Join	846.00	2732.00	142.00	758.00	2132.00	4.00
Table Task 11: TPC-DS Q15	1409.00	9428.00	N/A	185.00	8932.00	N/A
Table Task 12: TPC-DS Q20	1363.00	15478.00	N/A	151.00	14237.00	N/A
Table Task 13: TPC-DS Q29	5364.00	27382.00	N/A	925.00	27382.00	N/A
Table Task 14: TPC-DS Q72	3952.00	20619.00	N/A	387.00	20574.00	N/A
Graph Task 1: N-hop Friends	2832.00	2414.00	15.00	242.00	2287.00	1.00
Graph Task 2: Shortest Path	2932.00	0.80	4.00	253.00	0.64	1.00

Table 4.3: The first execution time and the minimum (fastest) time of each task for three databases, the better performance records have been marked



(a)



(b)

Figure 4.4: The coefficient of variation of each task for three databases. (a) Table Task 1 - Table Task 8. (b) Table Task 9 - Graph Task 2.

4.4 Workload Classification

The experimental results and measurement metrics show databases have somewhat different response patterns when faced with those same tasks. For example, the coefficient of variation indicates that Neo4j has more significant fluctuations in executions time for many tasks than the other two. Meanwhile, considering only one database may have different processing patterns when faced with different workloads; for instance, when PostgreSQL was in the execution of the TPC-DS Q15, the time dropped sharply, but this didn't happen when it dealt with the Update task. In this chapter, we perform unsupervised learning on the results to determine whether the workload can be classified or clustered and explore the processing patterns of databases from a workload aspect.

4.4.1 Original Data Description

First, for each database, we put the 10 execution times and standard deviation (Std), average (Ave), maximum (Max) and minimum (Min) values, the difference between the first and tenth query times (TR(1-10)), as well as the percentage of execution time fluctuation range (Perc) and coefficient of variation (CoV) into one data frame, thus, we have two of 16*17 and one 12*17 data frames. Figure 4.5 shows the Pearson correlation coefficient analysis for these 3 data sets.

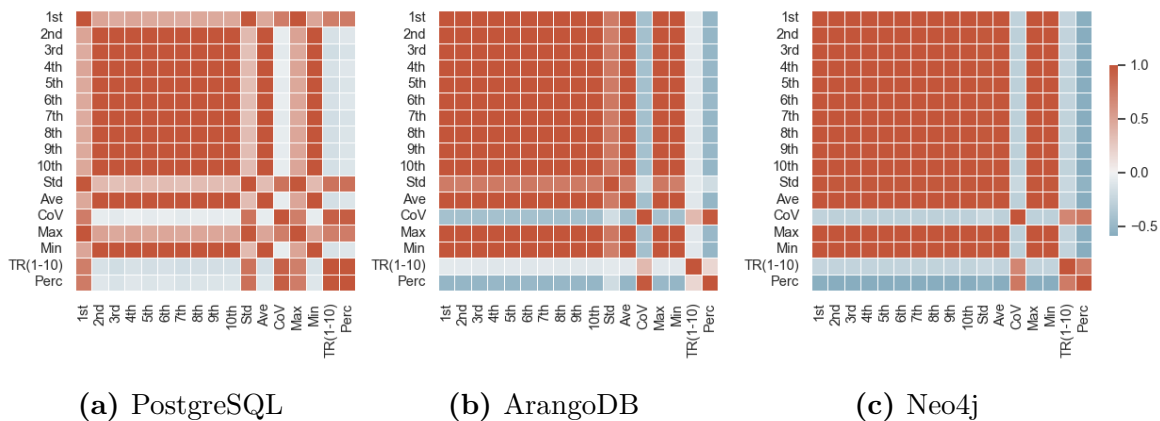


Figure 4.5: Correlation Coefficient of Three original data sets (Pearson)

4.4.2 Dimensionality Reduction

Feature Selection

The 17-dimensional raw data is high-dimensional and has many redundant variables. Based on the related research and literature, we first implement dimensionality reduction in the 17-dimensional data by correlation analysis and feature selection [40, 16].

As shown in Figure 4.5, the data for the ten execution times, are highly correlated with each other (correlation coefficient > 0.90), but PostgreSQL differs slightly from the rest of the two databases in that the correlation between the first execution time and the second to tenth time is less than the intercorrelation within the latter set. On the other hand, the execution time of PostgreSQL from the second to the ninth time is highly correlated with the average execution time. Therefore, we extract the execution time for the three datasets by keeping only the first execution time (**1st**). The original data is now reduced from 17 to 8 dimensions.

For all the three original data, **Min** and **Ave** features are highly correlated, while **Max** is highly correlated with the first execution time **1st**, so we can remove **Min** and **Max** from the data and won't lose much information. Both **CoV** and **Std** are used to measure the dispersion of the data set, and **CoV** is referenced to **Std** and removes some of the effect of the absolute size of the original data, so we can choose to keep **CoV** and remove **Std**. At this point, we now have a 16×5 or 12×5 feature-selected data set for three databases where the five features are **{1st, Ave, CoV, TR(1-10), Perc}**.

Principal Component Analysis (PCA)

Although five-feature looks not like a very high dimension, we can still perform further dimensionality reduction by Principal Component Analysis (PCA) [7]. PCA is a dimensionality reduction method that projects the data onto the directions drawn according to the features by looking at the eigenvectors of the covariance matrix as indicators and starting from the lines that cross the data along the direction of maximum variance [22].

We perform PCA analysis by using the `PCA()` function in the scikit-learn Python library [30], the visualization provided in the book [37] was used to analyze the number of components chosen for PCA concerning the explained variance it covers. We normalize all features by the minimum-maximum at the beginning.

The results of the PCA about the number of components verse cumulative explained variance analysis are shown in Figure 4.6. With only one PCA component, PostgreSQL's cumulative explained variance ranges from 0.75 (75%) to less than 0.8 (80%), while ArangoDB and Neo4j are below 0.7 (70%). When two PCA components are selected, the cumulative explained variance of the three covers more or almost 0.9 (90%), so we set the number of components for PCA as 2, and then we will use this 2-component PCA data for clustering.

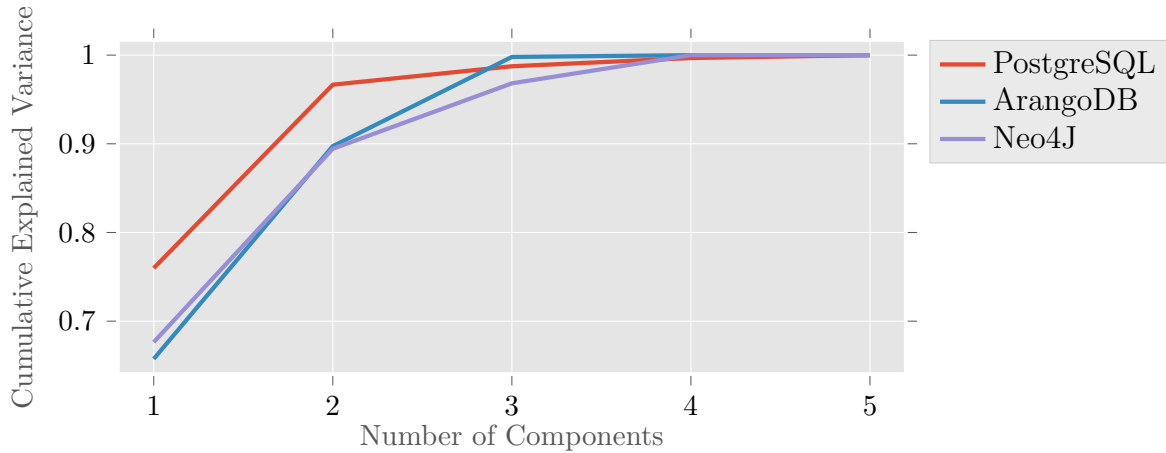


Figure 4.6: PCA Cumulative Explained Variance Analysis

4.4.3 Clustering

The workload or task classification now becomes a straightforward clustering analysis of two-feature data. In this section, we make use of the K-means clustering method in the scikit-learn library [30], which is a very classical unsupervised learning method. The results are represented graphically using the visualization methods mentioned in the book [37]. In addition, We also plot PCA biplots with the method provided in bioinfokit software [6], which will be used to analyze the clustering results.

PostgreSQL

From Figure 4.7 we can see that the PCA analysis of PostgreSQL, Ave and CoV TR(1-10) Perc almost perfectly account for the direction of the two components, while 1st is showing almost half and half positive correlation with both components. We then performed 4-class K-means clustering on the 16 or 12 task points, and the results are shown in Figure 4.8. The results of clustering show that the four clusters are respectively then 4 corners.

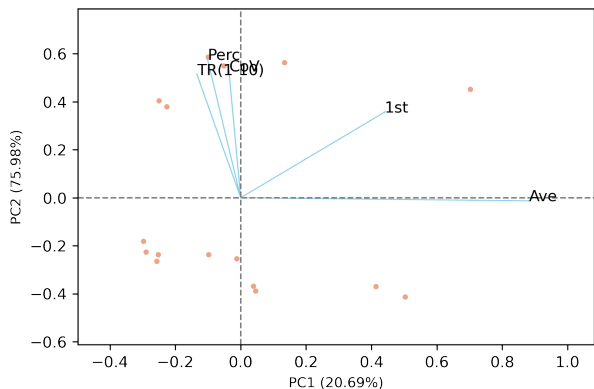


Figure 4.7: PCA Biplot of PostgreSQL

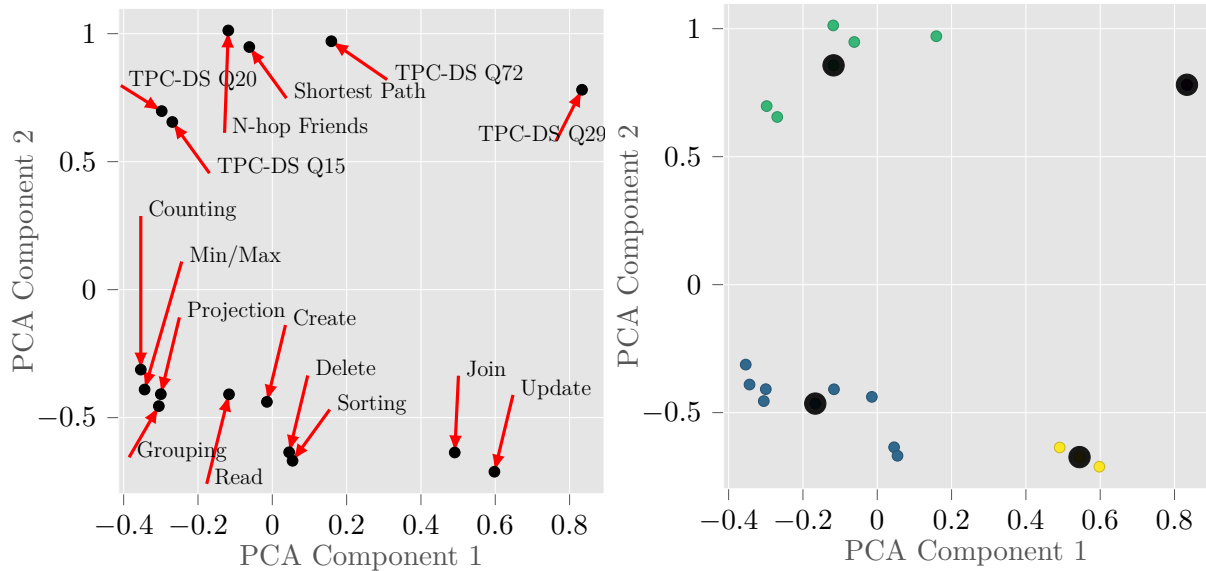


Figure 4.8: PostgreSQL Workload K-means Clustering Results. On the left is the projection of each data point on a 2-dimensional PCA, and on the right is the clustering result for 4-class K-means, where the center of each cluster represented by a larger point.

- Group 1: {TPC-DS Q29}
- Group 2: {TPC-DS Q20, TPC-DS Q15, Shortest., N-hop., TPC-DS Q72}
- Group 3: {Join, Update}
- Group 4: {Count., Min., Proj., Crea., Del., Sort., Group., Read}

It is distinct that Group 1 is the one with a long average time and a large time variation in 10 executions; Group 2 is with a small average time, but there is some variation with multiple executions; Group 3 is on the small side of both; Group 4 is the one with a long time but no big fluctuations in execution time;

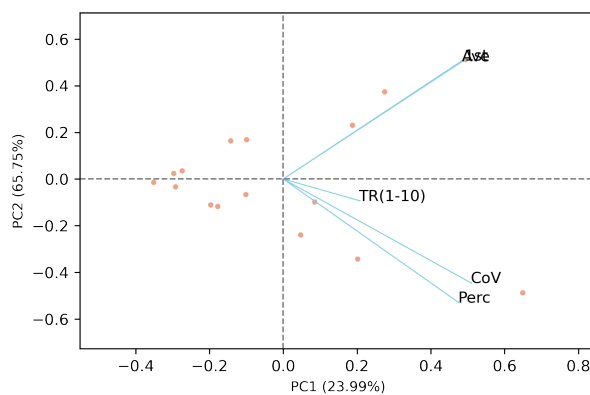


Figure 4.9: PCA Biplot of ArangoDB

ArangoDB

The PCA biplot of ArangoDB is shown in Figure 4.9, where `Ave` and `1st` are in one direction, while `Perc` and `CoV` are in the other direction, and `TR (1-10)` is also in the middle of that tow direction but not very well projected.

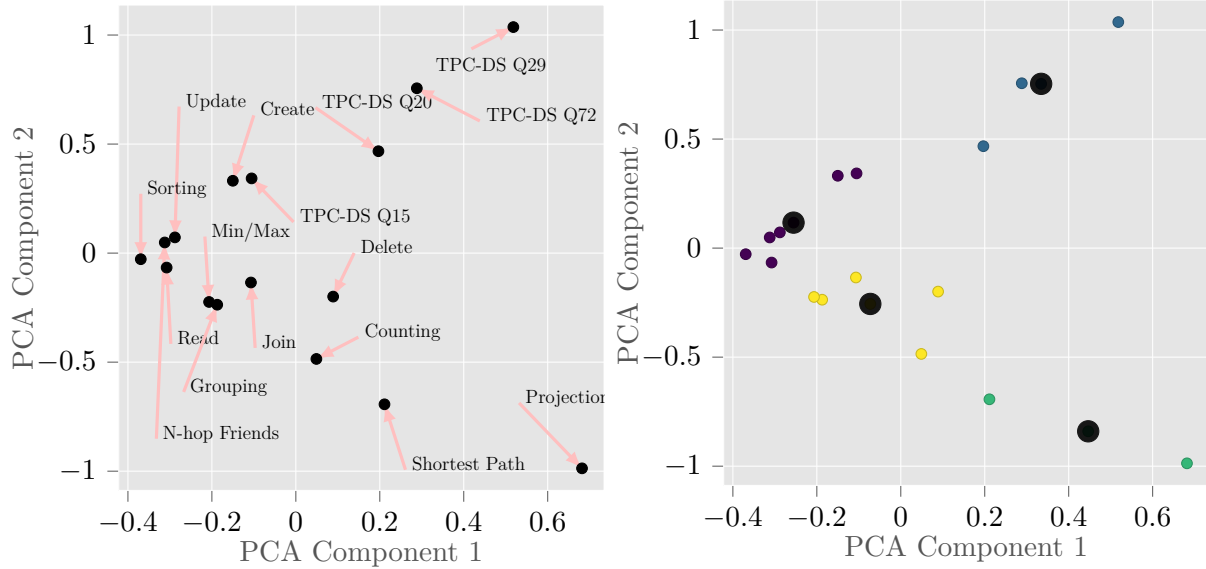


Figure 4.10: ArangoDB Workload 4-class K-means Clustering Results.

- Group 1: {TPC-DS Q29, TPC-DS Q20, TPC-DS Q72}
- Group 2: {Create, TPC-DS Q15, Update, Sorting, Read, N-hop Friends.}
- Group 3: {Join, Grouping, Min/Max, Counting, Delete}
- Group 4: {Projection, Shortest Path}

Figure 4.10 shows the result of 4-class K-means clustering, the data points are distributed in a 'V' shape, where the ends of two edges, Group1 and Group4, have a high average time/first execution time and a large time variation in multiple executions respectively, respectively. At the bottom of the V, Group2 and Group3 have relatively small those two dimensional metrics.

Neo4j

Figure 4.11 shows the PCA biplot for Neo4j, and we can see that the angle between the two components is acute. At the same time, the average time spent and the first execution time lead one direction and the time variation indicate the other. Figure 4.12 is the results of 4-class K-means clustering.

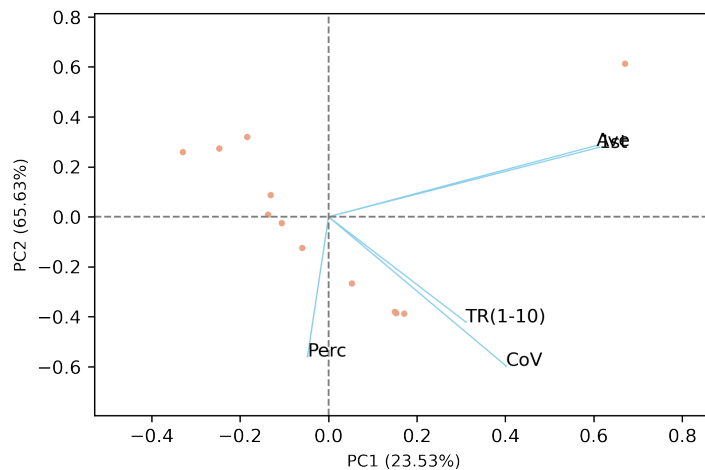


Figure 4.11: PCA biplot of Neo4j

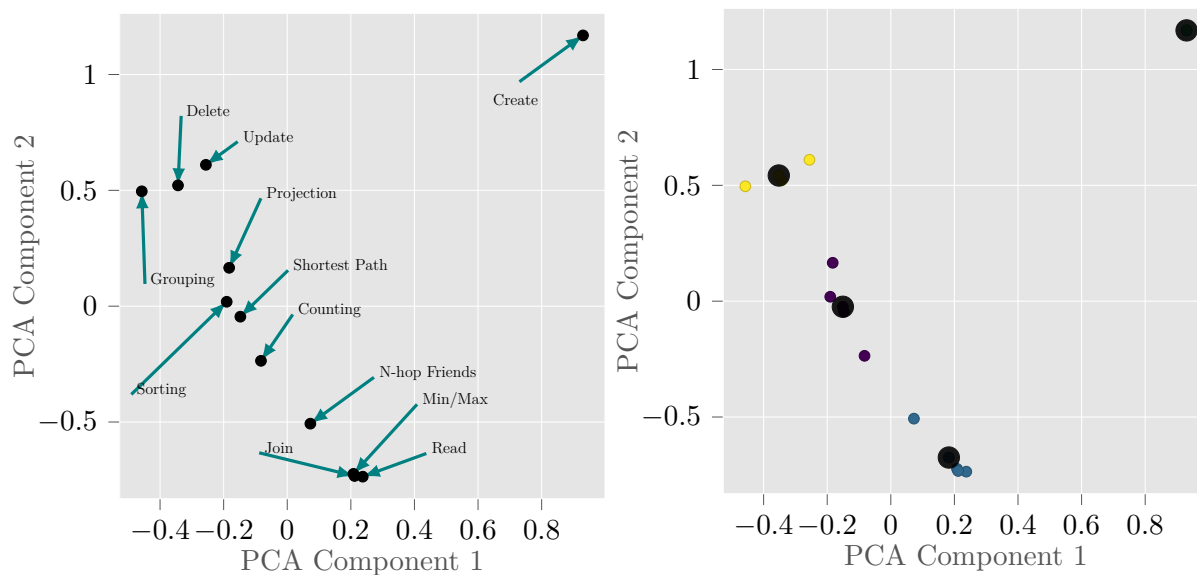


Figure 4.12: Neo4j Workload K-means Clustering Results.

- Group 1: {Create}
- Group 2: {Delete, Update, Grouping}
- Group 3: {Projection, Shortest Path, Counting, Sorting}
- Group 4: {Min/Max, Read, N-hop Friends, Join}

We can see that only one task, Create, is exclusive to a cluster because of high execution time, while the rest of the tasks can be said to be evenly distributed in the dimension of execution time variation.

5. Discussion

5.1 Performance

5.1.1 Execution Times

First of all, for the Create task, PostgreSQL has the best performance among the three databases in terms of both first query time and fastest time, ArangoDB has the second-best performance, and Neo4j has the longest time. In fact, this is also positively related to the time we spent importing data into the three databases, the reason for this difference is each time a new record (node) is created, Neo4j needs to generate many edges based on node dependencies, while in the relational or multi-model databases it is only necessary to update the table by constraints. The storage logic of different databases determines this, and the graph database maintains all relationships for each piece of data separately. Also, in terms of first execution time and fastest execution time, PostgreSQL is optimized for the Create task with multiple executions, while ArangoDB and Neo4j do not change in multiple significantly.

On the other hand, the execution time of the Join task on Neo4j is notably lower than on PostgreSQL and ArangoDB, where PostgreSQL is faster than ArangoDB. This is because we do not need to do too much computation in the graph database when we do the Join operation. After all, the relationship between the nodes is already apparent. However, relational databases still need to perform Join based on a series of inter-table links such as primary key foreign key, and the computation is greater than that of graph databases.

For Read and Min/Max tasks, PostgreSQL is the fastest in first execution time, but Neo4j wins in the fastest execution time. For Update, Delete, Sorting, Counting, and Grouping tasks, Neo4j achieves the fastest of all three in both first execution time and fastest execution time. In the tasks based on TPC-DS workload queries, we tested only PostgreSQL and ArangoDB because the time to import the entire data set into Neo4j was too long, with PostgreSQL having an advantage in both first execution time and fastest execution time.

In two graph task experiments, Neo4j is faster than both PostgreSQL and ArangoDB

in N-hop Friends. At the same time, ArangoDB is faster than the remaining two in the shortest path, both in first execution time and fastest execution time. The speed advantage of a graph database in applying graph algorithms is obvious, and it also supports a variety of graph algorithms. We do not need to write complex statements like the ones designed for PostgreSQL in this experiment.

5.1.2 Cache

For PostgreSQL, we can see that it gets a significantly faster execution time on the second, third execution than on the first for many tasks. In particular, it shows impressive optimization capabilities for Table Tasks 11 to 15. Access to system tables and standard table schema is persistent for a typical PostgreSQL system. To improve the efficiency of these accesses, PostgreSQL has set up a cache to improve access efficiency. The results prove that this cache mechanism can optimize some tasks that have already been queried.

ArangoDB's cache mechanism is to cache query strings as keys. When running two queries with identical values, the database will query from the cache to see if there is an exact cache key match, and if so, return the result corresponding to that cache key. But ArangoDB's cache has many restrictions. The query string must be identical, case, space-sensitive, and must be a read-only query, streaming cursor is not applied, has the same bind parameter values, etc.; we can see some tasks in ArangoDB. The first query is cached to provide a faster response for the second execution of the same job.

Similarly, Neo4j has impressive caching features. When we use its query language Cypher, Neo4j will compute a hash of the string of the Cypher statement as a cache key. Similarly, two Cypher statements perform the same operation but have different hash cache keys due to case differences or spaces. The last query will not get results from the cache as quickly. It is also necessary to note that although Neo4j's execution time looks fast if it is in the cache, there is a considerable delay between the end of execution and the return of results. It isn't easy to appreciate the performance advantage in practice.

5.2 Which One Should I Choose?

As we got in all comparisons, no database is better for every item, so there is no best choice. We also did not get results entirely consistent with any other literature. The answer to this question is that it needs to be case-specific, or the user needs to set up their tests.

As a traditional relational database, PostgreSQL has both wins and losses in performance with the other two databases. And relational databases have a strong guarantee of maintaining data consistency (transaction processing). Because of the premise of stan-

standardization, all use table structure, the overhead of data update is tiny. Easy to maintain. Ease of use is also a significant advantage; SQL query language is prevalent and can be used for complex queries; for example, it can be used for very complex queries between a table and multiple tables. But at the same time, the performance disadvantage of handling multi-table join loads compared to graph databases is also evident. As a classical language, SQL is powerful but still seems insufficient when facing non-relational data. For example, for the two graph tasks tested in this experiment, we need to write complex SQL functions that are only oriented to simple undirected unweighted graphs and were not general methods, which is very inconvenient for users to try graph algorithms.

ArangoDB's native support for data storage and querying in multiple formats is impressive, and it also provides a unified query language AQL that allows users even to involve data from different models in a single query and as a query language similar to SQL, AQL is not too difficult for users to learn. Its storage and query for graph data are also very powerful, with rich graph algorithm support. However, ArangoDB has some performance gaps compared to single format databases, its queries on relational data are not as fast as PostgreSQL, and some are not as fast as Neo4j when it comes to graph queries.

The three databases have different query languages, among which, Cypher, as a declarative graph query language, has the most intuitive representation when querying graphs. And for Neo4j, If the origin data is natively graph-structured, such as road networks, social networks, etc., it is not difficult to convert from the original table data to graph format. Then this graph database has a series of advantages over the traditional relational database, such as graph data storage schema, powerful graph query language support, graph algorithm library, and excellent performance in the face of the "multi-table join" problem. However, when the user needs to model table data to graph and there are a large number of types for nodes and complex relationships, then the workload and import time will increase exponentially, for example, in Appendix A.2, this is only the dependency of the first three tables and other tables, and it takes several minutes or even hours to build each relationship, which is unacceptable to the regular users.

In summary, the choice of which database to store the data is still a slightly complex issue, and it is closely related to the format of the original data. If the original data is of network or graph type, then using a graph database for storage is primarily supported by richer graph algorithms and can use a more intuitive graph query language. When the original data is tabular data, you need to consider the cost of converting the format to graph and what kind of tasks you usually need to perform. Sometimes, blindly choosing a graph database will not bring the desired faster, better and easier user experience.

6. Conclusions

The purpose of this thesis is to propose a reasonable and credible experimental comparison of three different types of databases. We evaluate databases on the same data set under the same metrics. The interconversion of Cypher between SQL and graph query languages and the multi-model query language AQL is implemented in PostgreSQL and Neo4j and ArangoDB. We performed various tasks based on a big data benchmark standard TPC-DS, which is commonly used in RDBMS for decision support, and another graph data set from the real world; in the end, we report our results in detail. We also compare the three databases by end-user analysis and give our summary. Among the performance advantages, PostgreSQL has a performance advantage in import, read, etc. operations, while Neo4j has a clear advantage in multi-table join tasks, graph algorithms, and ArangoDB natively supports many different formats of data storage and a unified query language, as well as having impressive support for graph algorithms as well. We also propose an unsupervised workload classification model to explore the tasks involved in the benchmarking, which helps design benchmark tests and evaluate database behavior and property.

For future work, I think it is interesting to evaluate performance and price/cost towards Cloud databases. All three databases support Cloud deployment and migration, then the price or cost would be a very interesting topic of discussion in the face of the same data set and workload. We also did not cover SQL-based graph databases for this experiment, such as AgensGraph, and since ArangoDB is not the only MMDBMS, our future direction is to cover more databases for benchmarking and to provide multiple data sets and corresponding tasks support.

Acknowledgement

I would like first to thank Professor Jiaheng Lu and doctoral student Zhengtong Yan for providing me with such an interesting and practical thesis topic. They have been accommodating and caring during my thesis works. I also would like to sincerely thank Associate Professor Michael Mathioudakis for taking precious time to help me review this thesis.

My master's study started, continued, and now is going to finish with the COVID-19, I haven't taken even one lecture in contact teaching. However, that doesn't affect me as much as I expected since I got a bunch of help and support from the teachers, staff, and classmates here in Helsinki, thanks to you all (not including you, COVID).

The last and most important people I would like to thank are my family, thank you for everything you have given me, I couldn't have made those tough days through without your encouragement and support, I love you all!

Bibliography

- [1] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.
- [2] ArangoDB. Arangodb for machine learning. <https://www.arangodb.com/machine-learning/>. Online; accessed 19-February-2022.
- [3] ArangoDB. SQL/AQL Comparison. <https://www.arangodb.com/community-server/sql-aql-comparison/>. Online; accessed 12-February-2022.
- [4] ArangoDB. NoSQL Performance Benchmark 2018 - MongoDB, PostgreSQL, OrientDB, Neo4j and ArangoDB. <https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/>, 2018. Online; accessed 12-February-2022.
- [5] P. Atzeni, C. S. Jensen, G. Orsi, S. Ram, L. Tanca, and R. Torlone. The relational model is dead, sql is dead, and i don't feel so good myself. *ACM Sigmod Record*, 42(2):64–68, 2013.
- [6] R. Bedre. reneshbedre/bioinfokit: Bioinformatics data analysis and visualization toolkit, May 2020.
- [7] G. Chandrashekar and F. Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.
- [8] Y. Cheng, P. Ding, T. Wang, W. Lu, and X. Du. Which category is better: benchmarking relational and graph database management systems. *Data Science and Engineering*, 4(4):309–322, 2019.
- [9] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227, 2005.
- [10] B. C. Churchill and W. Xu. The modem nation: A first study on twitch.tv social structure and player/game relationships. In *2016 IEEE International Conferences*

- on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, pages 223–228, 2016.
- [11] Digoal. PostgreSQL Graph Search Practices - 10 Billion-Scale Graph with Millisecond Response. https://www.alibabacloud.com/blog/postgresql-graph-search-practices---10-billion-scale-graph-with-millisecond-response_595039, 2019. Online; accessed 12-February-2022.
- [12] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630, 2015.
- [13] D. Fernandes and J. Bernardino. Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb. In *Data*, pages 373–380, 2018.
- [14] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, 2018.
- [15] J. R. Groff, P. N. Weinberg, and A. J. Opper. *SQL: the complete reference*, volume 2. McGraw-Hill/Osborne, 2002.
- [16] M. A. Hall et al. Correlation-based feature selection for machine learning. 1999.
- [17] I. Holubova, P. Contos, and M. Svoboda. Multi-model data modeling and representation: State of the art and research challenges. In *25th International Database Engineering & Applications Symposium*, pages 242–251, 2021.
- [18] C. Krause, D. Johannsen, R. Deeb, K.-U. Sattler, D. Knacker, and A. Niadzelka. An sql-based query language and engine for graph pattern matching. In *International Conference on Graph Transformation*, pages 153–169. Springer, 2016.
- [19] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [20] J. Lu and I. Holubová. Multi-model databases: a new journey to handle the variety of data. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.
- [21] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang. G-sql: Fast query processing via graph exploration. *Proceedings of the VLDB Endowment*, 9(12):900–911, 2016.

- [22] A. Malhi and R. X. Gao. Pca-based feature selection scheme for machine defect classification. *IEEE transactions on instrumentation and measurement*, 53(6):1517–1525, 2004.
- [23] J. J. Miller. Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, volume 2324, 2013.
- [24] R. O. Nambiar and M. Poess. The making of tpc-ds. In *VLDB*, volume 6, pages 1049–1058, 2006.
- [25] Neo4j. Getting started / introduction to cypher / import data. <https://neo4j.com/docs/getting-started/current/cypher-intro/load-csv/>. Online; accessed 12-February-2022.
- [26] Neo4j. Neo4j graph data science. <https://neo4j.com/product/graph-data-science/>. Online; accessed 19-February-2022.
- [27] Neo4j. Tutorial: Import Relational Data Into Neo4j. <https://neo4j.com/developer/guide-importing-data-and-etl/>. Online; accessed 13-February-2022.
- [28] A. Oasis. Neo4j graph data science. <https://cloud.arangodb.com/home>. Online; accessed 19-February-2022.
- [29] A. Pacaci, A. Zhou, J. Lin, and M. T. Özsu. Do we need specialized graph databases? benchmarking real-time social networking applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, pages 1–7, 2017.
- [30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [31] M. Poess, R. O. Nambiar, and D. Walrath. Why you should run tpc-ds: A workload analysis. In *VLDB*, volume 7, pages 1138–1149, 2007.
- [32] B. Ramya, N. S. Varma, and R. Indra. Recommendations in social network using link prediction technique. In *2020 International Conference on Smart Electronics and Communication (ICOSEC)*, pages 782–786, 2020.

- [33] B. Rozemberczki, C. Allen, and R. Sarkar. Multi-scale attributed node embedding. *Journal of Complex Networks*, 9(2):cnab014, 2021.
- [34] J. A. Stothers and A. Nguyen. Can neo4j replace postgresql in healthcare? *AMIA Summits on Translational Science Proceedings*, 2020:646, 2020.
- [35] TPC. TPC Benchmark DS - Standard Specification, Version 2.10.0 . http://tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.10.0.pdf, 2018. Online Standard Specification; accessed 12-February-2022.
- [36] C.-O. Truica, F. Radulescu, A. Boicea, and I. Bucur. Performance evaluation for crud operations in asynchronously replicated document oriented database. In *2015 20th International Conference on Control Systems and Computer Science*, pages 191–196, 2015.
- [37] J. VanderPlas. *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc.", 2016.
- [38] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*, pages 1–6, 2010.
- [39] A. Viloría, G. C. Acuña, D. J. A. Franco, H. Hernández-Palma, J. P. Fuentes, and E. P. Rambal. Integration of data mining techniques to postgresql database manager system. *Procedia Computer Science*, 155:575–580, 2019.
- [40] L. Yu and H. Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Proceedings of the 20th international conference on machine learning (ICML-03)*, pages 856–863, 2003.
- [41] C. Zhang and J. Lu. Holistic evaluation in multi-model databases benchmarking. *Distributed and Parallel Databases*, 39(1):1–33, 2021.
- [42] C. Zhang, J. Lu, P. Xu, and Y. Chen. Unibench: A benchmark for multi-model database management systems. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 7–23. Springer, 2018.
- [43] Y. Zhang and Z. G. Ives. Finding related tables in data lakes for interactive data science. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1951–1966, 2020.

Appendix A. TPC-DS Data ERD Supplementary and Edge Schema

A.1 ER-Diagram for TPC-DS Web/Store Parts

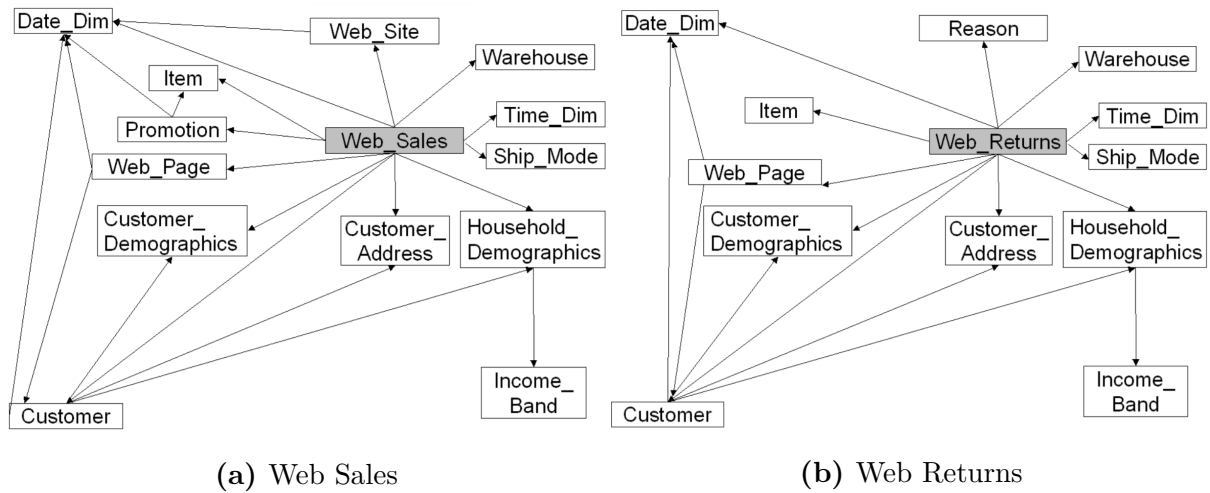


Figure A.1: ER-Diagrams of Web Sales/Returns

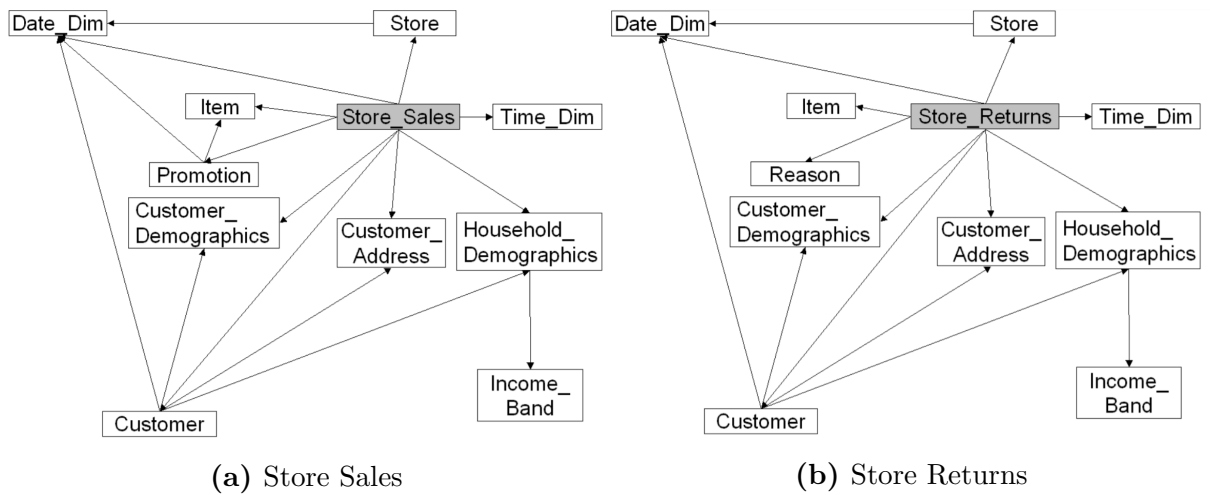



Figure A.2: ER-Diagrams of Store Sales/Returns

A.2 TPC-DS Data Graph Relationship Mapping Table

TPC-DS data graph relationship mapping table (sample)		
From key	To key	Relationship/Edge
ss_sold_date_sk	d_date_sk	ss_to_d
ss_sold_time_sk	t_time_sk	ss_to_t
ss_item_sk	i_item_sk	ss_to_i
ss_customer_sk	c_customer_sk	ss_to_c
ss_cdemo_sk	cd_demo_sk	ss_to_cd
ss_hdemo_sk	hd_demo_sk	ss_to_hd
ss_addr_sk	ca_address_sk	ss_to_ca
ss_store_sk	s_store_sk	ss_to_s
ss_promo_sk	p_promo_sk	ss_to_p
sr_returned_date_sk	d_date_sk	sr_to_d
sr_return_time_sk	t_time_sk	sr_to_t
sr_item_sk	i_item_sk	sr_to_i
sr_item_sk	ss_item_sk	sr_to_ss_item
sr_customer_sk	c_customer_sk	sr_to_c
sr_cdemo_sk	cd_demo_sk	sr_to_cd
sr_hdemo_sk	hd_demo_sk	sr_to_hd
sr_addr_sk	ca_address_sk	sr_to_ca
sr_store_sk	s_store_sk	sr_to_s
sr_reason_sk	r_reason_sk	sr_to_r
sr_ticket_number	ss_ticket_number	sr_to_ss_ticket
cs_sold_date_sk	d_date_sk	cs_sold_to_d
cs_sold_time_sk	t_time_sk	cs_to_t
cs_ship_date_sk	d_date_sk	cs_ship_to_d
cs_bill_customer_sk	c_customer_sk	cs_bill_to_c
cs_bill_cdemo_sk	cd_demo_sk	cs_bill_to_cd
cs_bill_hdemo_sk	hd_demo_sk	cs_to_hd
cs_bill_addr_sk	ca_address_sk	cs_bill_to_ca
cs_ship_customer_sk	c_customer_sk	cs_ship_to_c
cs_ship_cdemo_sk	cd_demo_sk	cs_ship_to_cd


Appendix B. Experiments Raw Results

PostgreSQL Results (ms):




	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
Table Task 1	457.00	414.00	324.00	334.00	325.00	342.00	364.00	354.00	348.00	351.00
Table Task 2	360.00	254.00	249.00	259.00	287.00	262.00	287.00	269.00	273.00	260.00
Table Task 3	765.00	816.00	881.00	853.00	855.00	904.00	873.00	867.00	861.00	861.00
Table Task 4	379.00	371.00	413.00	330.00	371.00	354.00	375.00	368.00	382.00	381.00
Table Task 5	145.00	106.00	121.00	105.00	104.00	96.00	101.00	97.00	104.00	108.00
Table Task 6	407.00	375.00	385.00	386.00	375.00	373.00	380.00	367.00	396.00	375.00
Table Task 7	116.00	83.00	74.00	72.00	85.00	70.00	73.00	74.00	73.00	75.00
Table Task 8	122.00	94.00	107.00	89.00	99.00	94.00	104.00	95.00	93.00	89.00
Table Task 9	92.00	68.00	71.00	70.00	71.00	74.00	62.00	59.00	77.00	67.00
Table Task 10	846.00	773.00	786.00	767.00	778.00	769.00	768.00	758.00	763.00	772.00
Table Task 11	1409.00	192.00	193.00	203.00	189.00	199.00	193.00	185.00	194.00	185.00
Table Task 12	1363.00	169.00	183.00	172.00	176.00	192.00	182.00	155.00	151.00	171.00
Table Task 13	5364.00	984.00	930.00	930.00	930.00	936.00	925.00	928.00	943.00	937.00
Table Task 14	3952.00	456.00	439.00	428.00	395.00	413.00	423.00	415.00	387.00	409.00
Graph Task 1	2832.00	242.00	243.00	245.00	261.00	248.00	246.00	250.00	276.00	251.00
Graph Task 2	2932.00	256.00	253.00	293.00	282.00	306.00	257.00	287.00	295.00	304.00

ArangoDB Results (ms):



	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
Table Task 1	8704.00	8642.00	8423.00	8563.00	8578.00	8423.00	8829.00	8367.00	8482.00	8723.00
Table Task 2	663.00	614.00	604.00	603.00	605.00	603.00	629.00	613.00	605.00	583.00
Table Task 3	3133.00	3015.00	3130.00	3076.00	3117.00	3119.00	3164.00	3225.00	3347.00	3157.00
Table Task 4	4728.00	4273.00	4923.00	5832.00	4378.00	6389.00	4982.00	5362.00	5398.00	5264.00
Table Task 5	864.00	468.00	354.00	362.00	391.00	423.00	432.00	371.00	409.00	482.00
Table Task 6	632.00	645.00	673.00	652.00	692.00	662.00	653.00	683.00	705.00	652.00
Table Task 7	209.00	235.00	301.00	274.00	234.00	332.00	275.00	262.00	341.00	224.00
Table Task 8	156.00	162.00	142.00	152.00	136.00	173.00	146.00	136.00	174.00	152.00
Table Task 9	275.00	253.00	285.00	253.00	262.00	270.00	217.00	274.00	263.00	285.00
Table Task 10	2732.00	2638.00	2582.00	2348.00	2642.00	2427.00	2349.00	2427.00	2132.00	2489.00
Table Task 11	9428.00	8932.00	9324.00	9283.00	9137.00	9473.00	9327.00	9512.00	8993.00	9238.00
Table Task 12	15478.00	14237.00	15923.00	15237.00	15657.00	15392.00	15238.00	15207.00	16524.00	15294.00
Table Task 13	27382.00	28329.00	28234.00	28691.00	27382.00	27691.00	28270.00	28622.00	28471.00	27397.00
Table Task 14	20619.00	20787.00	20783.00	20790.00	22068.00	21558.00	21264.00	20574.00	20987.00	21115.00
Graph Task 1	2414.00	2482.00	2518.00	2287.00	2432.00	2350.00	2372.00	2363.00	2466.00	2379.00
Graph Task 2	0.80	0.87	1.35	0.99	0.83	0.79	0.64	0.84	0.78	0.98

Neo4j Results (ms):



	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
Table Task 1	17014.00	19149.00	19380.00	18283.00	18168.00	19890.00	19166.00	19049.00	18870.00	18038.00
Table Task 2	542.00	20.00	23.00	24.00	25.00	24.00	23.00	24.00	25.00	24.00
Table Task 3	524.00	594.00	601.00	543.00	555.00	525.00	549.00	632.00	625.00	574.00
Table Task 4	278.00	289.00	255.00	266.00	386.00	312.00	413.00	317.00	265.00	428.00
Table Task 5	514.00	212.00	265.00	264.00	275.00	213.00	271.00	250.00	204.00	245.00
Table Task 6	40.00	10.00	19.00	13.00	13.00	15.00	14.00	15.00	13.00	16.00
Table Task 7	42.00	8.00	6.00	5.00	5.00	6.00	5.00	7.00	8.00	6.00
Table Task 8	42.00	94.00	107.00	89.00	99.00	94.00	104.00	95.00	93.00	89.00
Table Task 9	152.00	6.00	7.00	7.00	7.00	7.00	6.00	5.00	7.00	6.00
Table Task 10	142.00	8.00	6.00	5.00	5.00	4.00	7.00	7.00	9.00	4.00
Graph Task 1	15.00	1.00	2.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Graph Task 2	4.00	1.00	1.00	1.00	1.00	1.00	2.00	1.00	1.00	1.00