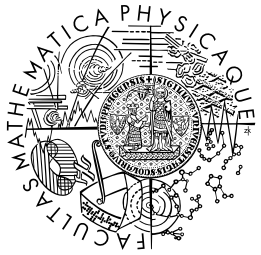


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Martin Čorovčák

Experimental Analysis of Query Languages in Modern Database Systems

Department of Software Engineering

Supervisor of the bachelor thesis: Ing. Pavel Koupil, Ph.D.

Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, Ing. Pavel Koupil, Ph.D., for his guidance and support. I would also like to thank my family and friends for their encouragement and understanding.

Title: Experimental Analysis of Query Languages in Modern Database Systems

Author: Martin Čorovčák

Department: Department of Software Engineering

Supervisor: Ing. Pavel Koupil, Ph.D., Department of Software Engineering

Abstract: The rise of Big Data has highlighted the limitations of relational databases while handling large datasets, leading to the growth of NoSQL databases. This has made DBMS benchmarking crucial for performance evaluation and decision-making. This thesis compares relational (MySQL, SQLite), graph (Neo4j, ArangoDB), document (MongoDB), and column-family (Cassandra) databases. We analyze the expressive power of their query languages and their runtime efficiency across varying data sizes. We conclude, that there's no "number one" solution for all use cases. The choice depends on factors like data volume, query complexity, and the need for joins. For complex queries and frequent joins, MySQL and SQLite are the most expressive but may struggle with very large datasets. Cassandra and MongoDB excel in performance and scalability but require efficient schema design and targeted data redundancy. ArangoDB presents a versatile option capable of handling multiple data models but might require further investigation into its performance compared to Neo4j.

Keywords: database management systems, performance, benchmark, static analysis, experimental analysis

Název práce: Experimentální analýza dotazovacích jazyků v moderních databázových systémech

Autor: Martin Čorovčák

Katedra: Katedra Softwarového Inženýrství

Vedoucí bakalářské práce: Ing. Pavel Koupil, Ph.D., Katedra Softwarového Inženýrství

Abstrakt: Príchod Veľkých Dát poukázal na obmedzenia relačných databáz pri spracovaní veľkých datasetov, čo viedlo k nárastu NoSQL databáz. Z tohto dôvodu sa DBMS benchmarking stal kľúčovým pre hodnotenie výkonnosti a celkový rozhodovací proces. Táto práca porovnáva relačné (MySQL, SQLite), grafové (Neo4j, ArangoDB), dokumentové (MongoDB) a stĺpcovo-orientované (Cassandra) databázy. Analyzujeme vyjadrovaciu silu ich dopytovacích jazykov a efektivitu počas behu pri rôznych veľkostiach dát. Dospeli sme k záveru, že neexistuje žiadne riešenie "číslo jeden" pre všetky prípady použitia. Výber závisí od faktorov, ako je objem dát, zložitosť dopytov a potreba spájania. V prípade zložitých dotazov a častého spájania majú MySQL a SQLite najväčšiu vyjadrovaciu silu, avšak môžu mať problémy s veľmi veľkými datasetmi. Cassandra a MongoDB vynikajú výkonom a škálovateľnosťou, ale vyžadujú efektívny návrh schématu a cielenú redundanciu dát. ArangoDB predstavuje univerzálnu možnosť, ktorá dokáže pracovať s viacerými dátovými modelmi, ale pre hlbšie porovnanie s Neo4j sa môže vyžadovať ďalší výskum ich výkonu.

Klíčová slova: databázové systémy, výkon, benchmark, statická analýza, experimentální analýza

Contents

Introduction **7**

1 Related work **9**

2 Static Analysis **10**
2.1 Selected Database Management Systems 10
2.2 Objectives 11
2.3 Individual Analysis 11
2.3.1 MySQL 12
2.3.2 SQLite 14
2.3.3 Neo4j 16
2.3.4 ArangoDB 19
2.3.5 Cassandra 23
2.3.6 MongoDB 27
2.4 Summary 30

3 ETL **33**
3.1 Data Domain and Data Models 33
3.2 Data Generation (Export) 41
3.3 Data Transformation 42
3.4 Data Loading (Import) 46
3.5 ETL Statistics 47

4 Dynamic Analysis **52**
4.1 Testing Environment 52
4.2 Methodology 53
4.3 Results 54
4.4 Summary and Recommendations 60
4.4.1 Recommendations 66

Conclusion **69**

Bibliography **70**

List of Figures **76**

List of Tables **77**

List of Listings **78**

List of Abbreviations **79**

Glossary **81**

A Attachments **82**
A.1 Project Directory 82
A.1.1 Repository Structure 82
A.1.2 Scripts 85
A.1.3 Prerequisites 86
A.1.4 Installation, Configuration, and Initialization 86
A.1.5 Legacy Generator 90
A.1.6 Query Testing 90
A.2 Source Codes 91

Introduction

At the beginning of the 1970s, the relational data model, introduced by E. F. Codd [1] became the universal standard for database systems. However, advancements in science and technology and the need to store and process massive amounts of data, have led to the development of new data models and Database Management Systems (DBMSs).

NoSQL databases (i.e. non-relational databases) [2], which were introduced to address the limitations of relational databases, have increased in popularity in recent years. NoSQL databases were designed based on the new data management requirements, that is to handle the three **V**s of Big Data, i.e. **v**olume (huge amount), **v**elocity (generated fast), and **v**ariety (un/semi-/structured data). Such databases are created with scalability, availability, and performance in mind, and they are often used in social networks, IoT, e-commerce, and other applications that exceed the capabilities of traditional RDBMSs.

Nowadays, the **v**ariety requirement has split the NoSQL world into several categories based on the underlying data model, i.e. key-value, document, column-family, graph and array data models. Furthermore, each of these models requires a query language to define data structures (DDL) and to retrieve and manipulate data (DML) [3].

Performance is a pivotal aspect of any software system, influencing the user experience and its competitiveness among other systems. The DBMS performance benchmark is often evaluated based on the query execution time [4], which is influenced by many factors, such as hardware, data model, distributed environment, DBMS and OS parameters, physical database design, and more [5].

Usually, the query performance can be significantly improved, but at the cost of a less expressive query language. For that reason, the concept of an aggregate was formed. By having a single data unit or a collection of related objects that are treated as a single unit, it allows NoSQL databases to store data in a duplicated form and speed up data retrieval [2].

With that in mind, we can classify the current state of DBMSs further. Aggregate-ignorant systems, such as relational, graph, and array databases, use strict schema (schema-first, data-later) approach and prioritize data consistency and transaction ACID properties. On the contrary, aggregate-oriented systems, such as key-value, document, and column-family databases, use aggregates and a schema-less (data-first, schema-later) approach allowing them to be more performant, available and scalable, but at the cost of an increasing data redundancy [6].

Goals The goal of this thesis is to focus on various data models (i.e. relational, graph, column-oriented, and document) and to compare typical query languages

for these data models in terms of expressive power and run-time efficiency, taking into account scalability with varying numbers of stored data.

We will analyze the current state of knowledge. Then, we will select six databases, i.e. relational (MySQL, SQLite), graph (Neo4j, ArangoDB), column-family (Cassandra) and document (MongoDB), which represent the most popular open-source representatives of their underlying data models. Next, we will statically compare their individual features and supported query languages. Next, based on the static comparison, we will propose query scenarios and perform dynamic query comparison over the selected database systems using the proposed queries. Finally, we will suggest appropriate recommendations in querying over various data representations.

Outline The rest of the thesis is structured as follows: In chapter 1, we present the related work on the topic of database querying and performance benchmarking. In chapter 2, we select DBMSs for comparison, and for each, we perform a static analysis of their capabilities and limitations that we use in the subsequent experiments. In chapter 3, we choose the data domain, propose queries used for testing, and for each chosen DBMS, we design a data model and perform ETL process, i.e. generate, process, and import data that will be used in the subsequent experiments. In chapter 4, we perform a dynamic analysis (experiments) of the DBMSs based on the proposed queries and present the results in the form of tables and charts. Afterward, we provide recommendations for choosing the best DBMS in proposed scenarios. Finally, we summarize the findings and suggest future work in Conclusion.

1. Related work

Firstly, we would like to mention the work of Taipalus [5] who created a clear and concise systematic literature review of various DBMS benchmarks and helped us understand common mistakes in DBMS benchmarking. Taipalus was inspired by the work of Raasveldt et al. [7], who proposed a checklist for fair DBMS performance testing, which we also tried to follow in this thesis.

The current state of knowledge in the field of database performance testing is diverse and covers many databases implementing different data models, utilizing various query languages, and providing different levels of scalability. TPC-H [8] and other benchmarks by the Transaction Processing Performance Council (TPC) are well-known in the field and one of the most respected benchmarks in the relational domain. For NoSQL databases, there is the Yahoo! Cloud Serving Benchmark (YCSB) [9], a framework with different workloads for R/W operations, that can be used to evaluate the performance of NoSQL databases. Namely, YCSB is used in the work of Abramova and Bernardino [10] to compare the performance of MongoDB and Cassandra, showing that Cassandra outperforms MongoDB in almost all tested scenarios. UniBench (Zhang and Lu [11]), a benchmarking tool for Multi-Model DBMSs, can be used to evaluate the performance of, e.g. ArangoDB. Gunawan et al. [12] compares document-oriented databases, showing that MongoDB is faster in query performance, but slower in create operations than ArangoDB.

Furthermore, the work of Győrödi et al. [13] compares SQL to a NoSQL system, recommending MongoDB over MySQL "if the application is data intensive and stores many data and queries lots of data". In-memory databases, such as SQLite compared to MongoDB and MySQL in the work of Wang et al. [14], showing that the in-memory processing is faster than disk-based processing of MySQL or MongoDB.

In the context of highly connected data as in social networks, the paper by Almabdy [15] shows that Neo4j is better suited for such cases than MySQL. On the other hand, Sholichah et al. [16] claims that Neo4j is slower overall and has higher memory usage than MySQL, but conversely also has bigger flexibility than MySQL.

2. Static Analysis

In this chapter, we will select DBMSs for comparison and perform a static analysis of their capabilities and limitations based on the official documentation and other research studies.

First, we will present the selected DBMSs in section 2.1 and explain their primary selection criteria. Then, we will define our objectives in section 2.2 and describe what will be the scope of our research in the individual analysis of each DBMS in section 2.3. Finally, we will summarize the analysis in respective tables in section 2.4.

2.1 Selected Database Management Systems

For this thesis, we have selected the following *DBMSs*: *MySQL*[1], *SQLite*[2], *Neo4j*[3], *ArangoDB*[4], *Cassandra*[5], and *MongoDB*[6]. The main reason for this selection was to compare different types of DBMSs in terms of their underlying *data models* used for logical data representation and various *query languages* used for data querying. We wanted to compare traditional *SQL* with more novel *NoSQL* databases and, for that reason, we chose data models that are widely used in practice and research, such as *Relational*, *Document*, *Graph*, and *Wide-column* data models. Based on the Goals of this thesis, we selected two representatives from the Relational domain (MySQL 2.3.1, SQLite 2.3.2), two from the Graph domain (Neo4j 2.3.3, ArangoDB 2.3.4), and one from Wide-Column (Cassandra 2.3.5) and Document (MongoDB 2.3.6) domain.

The subsequent selection is mainly based on the popularity of the systems presented by a well-known DBMS ranking website *DB-Engines*[7]. The selected systems are widely used in various domains and have different features and capabilities and as such are scalable to different extents. The selection criteria have also emphasized their open-source nature and having a large community of users, developers, and researchers. The selected systems are also used in various research studies and have been compared in different research papers. The primary selection criteria are summarized in table 2.1.

[1]Version 8.1.0: https://dev.mysql.com/doc/refman/8.0/en/
[2]Version 3.42.0: https://www.sqlite.org/docs.html
[3]Version 5.12.0: https://neo4j.com/docs/
[4]Version 3.11.3: https://docs.arangodb.com/3.11/
[5]Version 4.1.3: https://cassandra.apache.org/doc/4.1/
[6]Version 7.0.2: https://www.mongodb.com/docs/v7.0/
[7]https://db-engines.com/en/ranking

The popularity ranking presented by DB-Engines is based on the number of mentions on websites, general interest in the system, frequency of discussions about the system, and more.[8]

Table 2.1: DBMS primary selection criteria

#	Inclusion Criterion
1	Data model
2	Popularity ranking
3	Supported features
4	Research citations

2.2 Objectives

The main objectives of the static analysis are to:

- Analyze the selected *DBMS*s in terms of *Consistency*, *Scalability*, *Sharding*, *Replication*, *Schema*, *Aggregates*, *Entity types*, *Relations*, and the used representation of the *Absence of value*.
- Evaluate the *Data Definition Language (DDL)* features of the selected systems.
- Assess the *Data Manipulation Language (DML)* features of the selected systems.
- Compare the individual features, DDL, and DML between the selected systems.
- Summarize the analysis in respective tables (2.3, 2.4, 2.5).

The analysis will be based on the official documentation of the systems and information from other research studies. The analysis will be presented in the following sections.

2.3 Individual Analysis

In this section, we will analyze the selected DBMSs based on the objectives defined in section 2.2. Each subsection will focus on an individual DBMS, i.e. MySQL in section 2.3.1, SQLite in section 2.3.2, Neo4j in section 2.3.3, ArangoDB in

[8] https://db-engines.com/en/ranking_definition

section 2.3.4, Cassandra in section 2.3.5, and MongoDB in section 2.3.6. Further, each paragraph will analyze one aspect of the system that influences its performance, scalability, and expressive power of the query language.

2.3.1 MySQL

MySQL [17] is an open-source *Relational Database Management System (RDBMS)* that is widely used in various domains. It is developed and maintained by the *Oracle Corporation*. Ranked 2nd on DB-Engines ranking (March 2024), MySQL is a popular choice for web applications and is used by many high-profile websites. Furthermore, it is also used in various research studies and has been compared with other DBMSs [5]. The system is known for its high performance, reliability, and ease of use. It is also scalable and can be used in various environments, from small applications to large-scale enterprise systems.

MySQL requires a server to run since it uses a *client-server* architecture and needs to interact over a network. The server allows multiple user access and provides access control, user management, and strong security out of the box. Furthermore, since the DBMS offers an extensive set of features, the DBMS size can exceed 600 MB and the initial setup might require a more detailed configuration, than for example, SQLite (2.3.2).

Supported data types include, e.g.: `Int`, `Bigint`, `Numeric`, `Timestamp`, `Datetime`, `Char`, `Varchar`, `Blob`, `Text`, `Enum`, or `Set`. Additionally, the JSON [18, 19] data type is supported in a format that allows storing, querying and performing operations on JSON documents.[9] Interestingly, since version 8.0.17 MySQL supports Multi-Valued indexes, which allow indexing values of a JSON array stored in a column.[10]

Consistency MySQL is an *ACID*-compliant system, which means that it supports *Atomicity*, *Consistency*, *Isolation*, and *Durability* properties. [20]

Scalability MySQL is a *Vertically* scalable system. It can be scaled up by adding more resources to the existing server. It supports various clustering and replication features that enable it to be used in large-scale environments.

Sharding MySQL does not support sharding out of the box, but it can be implemented using various third-party tools and plugins. Oracle also provides customers with *MySQL NDB Cluster* [21], their "distributed database combining

[9] https://dev.mysql.com/doc/refman/8.0/en/json.html

[10] https://dev.mysql.com/doc/refman/8.0/en/create-index.html#create-index-multi-valued

linear scalability and high availability", that supports sharding and is designed for mission-critical applications.

Replication MySQL supports *Master-Slave* (that is, "Source-Replica") replication. [22]

Schema MySQL is a fully *Schema* based system. It requires a schema to be defined before data can be inserted into the database. It supports various data types and constraints that can be used to define the schema of the database.

Aggregates MySQL is an *aggregate-ignorant* system.

Entity types MySQL is a Relational DBMS, so it supports *Relations* as entity types.

Relations MySQL supports *Foreign Key* constraints that can be used to define relations between tables.

Absence of value MySQL uses *NULL* as a "metavalue" to represent the absence of a value.

Query language MySQL uses *Structured Query Language (SQL)* [23] as its query language. SQL is the primary query language used in all RDBMSs. It is extremely robust when it comes to expressive power and very powerful when it comes to querying and manipulating data. The relational data model [1], the data model behind MySQL, is based on relational algebra and relational calculus [24], and SQL is the language that is used to interact with the data stored in the database. The schema or structure of the database is fixed. The data is stored in tables, and the tables are related to each other using foreign keys. SQL supports all traditional *Create, Read, Update, Delete (CRUD)* operations.

MySQL follows the ANSI/ISO standards (with the newest one being *SQL-2023* [25, 26]) and has been extended to support various other features such as *Stored Procedures*, *Triggers*, *Views*, and *User-Defined Functions*. It differs from the standard in some aspects [27], but it is mostly compliant with the standard.

The DDL features include `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE` statements to manipulate tables and their schema, and `INSERT INTO`, `UPDATE`, and `DELETE FROM` statements to manipulate data (rows).

The DML features include the use of `SELECT`, `FROM`, `WHERE` as the main clauses for projection, source (tables/s), and selection of data. Integral to joining tables (entities) based on foreign keys (relationships) is the use of `JOIN` clause and

`OUTER JOIN` for optional relationships. Aggregation of data (maximum, minimum, average, count) is done using the `GROUP BY (properties) HAVING (condition)` statement. It also supports various other set operations (operations on sets of rows) such as `UNION`, `INTERSECTION`, `EXCEPT`, and `DISTINCT`[11]. Furthermore, the `ORDER BY`, `LIMIT`, `OFFSET`, and `AS` clauses are used for sorting, limiting, and skipping the results and aliasing the tables/columns.

The `WITH RECURSIVE` clause is used for unlimited traversal. It can perform the joining of a table with itself if a recursive condition is met. It allows for the execution of recursive queries within a Common Table Expression (CTE). A CTE is a named temporary result set that can be referred to within the context of other expressions. It can be used to traverse hierarchical data, graph-like data, or to perform recursive calculations on a table.

Also, the nesting of queries is supported using the `SELECT` statement within another `SELECT` statement. The *MapReduce* [28] operation can be expressed using the `GROUP BY ... HAVING` clause, but it is not a native feature of MySQL. It is a feature that is supported by the *Hadoop*[12] ecosystem and is used for distributed data processing.

2.3.2 SQLite

SQLite [29] is an open-source RDBMS available in the public domain (not owned by anyone). Developed by Dwayne Richard Hipp and ranked 10th on DB-Engines (March 2024), SQLite is a popular choice for many web browsers, operating systems, mobile phones, and embedded systems.

SQLite operates as a server-less, self-contained, zero-configuration database, commonly known as an embedded database. It is lightweight (about 250 kB in size) and uses a single file to store data and schema. It can also work as an *in-memory database*[13], where the entire database persists in primary memory to speed up access to the data (i.e. to avoid the I/O operation in transaction processing [14]). However, as the data requirements grow, performance optimization can get more complex, than in MySQL for example.

SQLite does not offer user management, access control, or authentication (features common in client-server DBMSs like MySQL). Instead, it relies on the host operating system's file access permissions to control access. Interestingly, "SQLite is the only known server-less DB that allows multiple applications to access the same database at the same time."[14]

[11] Not technically a set operation, but a feature of a set itself
[12] https://hadoop.apache.org/
[13] https://sqlite.org/inmemorydb.html
[14] https://www.sqlite.org/serverless.html

Supported data types ("Storage Classes")[15] include: `NULL`, `INTEGER`, `REAL`, `TEXT`, `BLOB`. Common data types from other SQL DBs, e.g. `VARCHAR`, `NUMERIC`, `DATETIME`, `BOOLEAN`, fall into so-called "Affinity Types", that are used to describe column types and are stored as one of the five storage classes. As SQLite is a "flexibly typed" database, it does not enforce data types. It allows any type of data to be stored in any column, regardless of the declared type. "This is a feature of SQLite, not a bug."[16]

Consistency SQLite is an ACID-compliant system, and thus all changes made in a transaction either succeed or fail together, even in the event of a program crash, an OS crash, or a computer power failure. To maintain exclusive access for a single write at any given time, the database utilizes a locking mechanism that encompasses the entire DB when a lock is required. While this approach may pose a limitation for high volumes of write operations, it proves to be efficient for applications of smaller to moderate sizes.[17] To enhance performance, SQLite takes advantage of shared locks implemented at the level of the OS's disk cache, rather than directly on the hard drive. What's important, is that SQLite emphasizes the significance of minimizing disk I/O to optimize transaction commit time.[18]

Scalability SQLite is *not designed* to be scalable. It can be horizontally scalable, but only to a certain extent, and only via third-party solutions.[19] Vertical scalability can be achieved by adding more resources (CPU, RAM, etc.) or by using a more powerful machine.

Schema SQLite is a fully *Schema* based system. Apart from "flexible typing" mentioned before, some other features are worth mentioning. The use of `AUTOINCREMENT` keyword can get quite computationally expensive as it requires extra CPU, memory, disk space, and I/O to process, so the documentation advises against using it.[20] "In SQLite, table rows normally have a 64-bit signed integer `ROWID` which is unique among all rows in the same table." Special clause `WITHOUT ROWID` can be used to create tables without the `ROWID`[21] column, that use a *clustered index* as the primary key and requires less disk space and I/O to process.

Some notable information: SQLite allows the use of double-quoted string literals, SQLite strings can contain NUL characters, and non-aggregate result columns in

[15] https://www.sqlite.org/datatype3.html
[16] https://www.sqlite.org/quirks.html
[17] https://www.sqlite.org/atomiccommit.html
[18] https://www.sqlite.org/atomiccommit.html - Chapter 7. Optimizations
[19] https://rqlite.io/
[20] https://sqlite.org/autoinc.html
[21] https://sqlite.org/withoutrowid.html

aggregate queries that are not in the `GROUP BY` clause are allowed.

Aggregates SQLite is an *aggregate-ignorant* system.

Entity types SQLite supports *Relations* as entity types.

Relations SQLite supports *Foreign Key* constraints, though they are not enforced by default. It is possible to enable them using the `PRAGMA foreign_keys=1`[22] command.

Absence of value SQLite uses *NULL* as a "metavalue" to represent the absence of a value.

Query language SQLite uses SQL as its query language. Supports all traditional CRUD operations and follows the ANSI/ISO standards. Some features from the standard are omitted[23], and some are changed or added (see 2.3.2).

The DDL features are the same as in MySQL (2.3.1). The only caveat may be that `ALTER TABLE` does not support some clauses like `ALTER COLUMN`, or `ADD CONSTRAINT`, etc.

The DML features include the same features as in MySQL (2.3.1). The notable difference is the limit imposed on the number of tables that can be joined in a single query, which is not more than 64.[24] This limitation also affects the `WITH RECURSIVE` recursive queries.

2.3.3 Neo4j

Neo4j [30] is an open-source *Graph* DBMS (or GDBMS) developed by *Neo4j, Inc.* and is ranked 23rd on DB-Engines (March 2024). Neo4j is widely used in domains that require interconnectedness and complex relationships between entities, such as social networks, recommender systems, fraud detection, network analysis, or quite novel applications in Machine Learning (ML), Artificial Intelligence (AI), and Internet of Things (IoT).

Neo4j utilizes a *Labeled Property Graph (LPG)* model [31] to represent data. The model consists of *nodes* (vertices), *relationships* (edges), and *properties* (key-value pairs). *Path*, retrieved from a *traversal* result is a sequence of one or more nodes connected by relationships. In DBMS, paths and relationships are first-class citizens. Nodes and relationships can have properties, and labels are used to group

[22] https://www.sqlite.org/pragma.html#pragma_foreign_keys
[23] https://sqlite.org/omitted.html
[24] https://sqlite.org/limits.html - 4. Maximum Number Of Tables In A Join

them into logical sets. Such a simple, yet efficient, data model allows invoking straightforward graph traversals (e.g. depth-first search, breadth-first search) to complex graph algorithms (e.g. shortest path, minimum spanning tree, etc.).

Neo4j is a client-server-based DBMS. Communication can be done via HTTP endpoint, or via the *Bolt* protocol, which is a binary protocol designed for high-performance communication. There exist various drivers for many programming languages, though originally Neo4j was developed in Java.

The supported (property) data types are either simple, i.e. `STRING`, `INTEGER`, `FLOAT`, `BOOLEAN`, structural, i.e. `LIST`, `MAP`, temporal, i.e. `DATE`, `LOCAL DATETIME`, `DURATION`, `LOCAL TIME` or spatial `POINT`.

Consistency Neo4j is an ACID-compliant system.

Scalability Neo4j Community (open-source) Edition is a vertically scalable system. It can be scaled up by adding more resources to the existing server.

It also supports clustering and sharding, but these features are available only in the Enterprise (commercial) version of the DBMS.[25]

Neo4j Clustering provides Fault Tolerance (FT) and High Availability (HA) via a master-slave architecture. *Causal consistency*, achieved by Neo4j's clustering, guarantees that causally related operations are observed in the same order by every instance in the cluster. As a result, clients are assured of reading their own writes, and it does not matter which instance they interact with. The server can be run in two modes (or both):

- *Primary* mode - Allows read and write operations. A database can be hosted by one or more primary hosts. HA in Primary nodes is accomplished through the replication of all transactions using the *Raft* protocol. This protocol guarantees the secure durability of data by requiring a transaction to be acknowledged by a majority of primaries in a database (N/2+1) before confirming its commit.
- *Secondary* mode - Database secondaries are asynchronously replicated from primaries via transaction log shipping. Periodically, they poll an upstream server for any new transactions and then proceed to have them transferred. They can be used for read-only operations.

Sharding Neo4j Enterprise Edition supports sharding via so-called *Composite databases*.[26] This technique divides a single logical database into several smaller

[25] https://neo4j.com/product/neo4j-graph-database/scalability/

[26] https://neo4j.com/docs/operations-manual/5/database-administration/composite-databases/concepts/

databases (shards).

Replication Neo4j Enterprise Edition supports automatic replication in a cluster.[27]

Schema Neo4j is a *schema-free* system. It does not enforce a schema, but it allows for the definition of a schema using *Constraints*, which can be used to enforce the uniqueness or existence of a property.

Aggregates Neo4j is an *aggregate-ignorant* system.

Entity types Neo4j supports *Nodes* as entity types.

Relations Neo4j supports *Relationships* as relations.

Absence of value Neo4j does not have a special representation for the absence of value. `null` in Neo4j means a missing or an unknown value. Interestingly, *null* = *null* yields `null` and not `true`, because two unknown values do not necessarily mean that they are the same.

Query language Neo4j uses *Cypher* as its query language. Cypher is a declarative (that is, we describe what we want) query language that is designed to be human-readable. It is based on pattern matching and is used to query and manipulate graph data. The language features all CRUD operations, but by not requiring a schema to be defined, it allows for more flexibility in data representation and querying. Additionally, *APOC (Awesome Procedures on Cypher)* serves as an extension library for Neo4j, offering an extensive collection of additional procedures and functions.

The DDL features include clauses like `CREATE` to create nodes and relationships, and `DELETE`/`DETACH DELETE` to delete them. The `MERGE` clause can be used to create nodes and relationships if they do not exist yet. Properties or labels can be modified using the `SET` clause, and the `REMOVE` clause can be used to remove them.

The DML features revolve around the `MATCH` clause to find nodes and relationships followed by a *graph specification* of a path/nodes/relationships to be matched in a traversal. Next, `WHERE` is used to filter/select the resulting properties, and finally `RETURN` to return/project the results (paths, nodes, relationships, properties, etc.). Also, the `OPTIONAL MATCH` statement can be used to optionally match entities in a traversal.

[27] https://neo4j.com/docs/operations-manual/current/clustering/introduction/

Optionally, we can use `ORDER BY`, `SKIP`, `LIMIT` to sort, skip, and limit the results. The `WITH` clause can be used to pass the results of one query to another query. The `UNION` and `WHERE NOT` (SQL `EXCEPT` alternative) clauses can be used to perform set operations on the results, including the `apoc.coll.intersection()` intersection function from the APOC library. The `CALL` clause can be used to call user-defined/external functions. The `FOREACH` clause can be used to perform operations on a list of items. Additionally, subqueries are supported as well using the `CALL {MATCH ...}` combination.

By default, Neo4j matches all nodes and relationships matching the graph specification, so there is no special clause for "MATCH *". This also implies that aggregation is oriented towards the graph specification, and not the properties themselves. An aggregating expression is a statement that contains one or more aggregating functions. The `collect()` aggregating function can be used to collect the results of a query into a list, and the `count()`, `min()`, `max()`, `avg()`, `sum()` functions can be used to perform aggregation on the results.

2.3.4 ArangoDB

ArangoDB [32] is an open-source *Multi-Model* DBMS (MMDBMS) made by *ArangoDB GmbH* and is ranked 84th on DB-Engines (March 2024). ArangoDB's flexibility supports a wide array of applications, from social networking to real-time analytics, recommendation systems, fraud detection, and more. It is also available commercially, deployable on-premises, or as a managed cloud service through ArangoGraph Insights Platform[28].

ArangoDB is classified as a MMDBMS [33, 34, 35], that supports three data models – *Document*, *Graph*, and *Key-Value* in a single core. The Document model is based on the JSON [19, 18] model, and the Graph model is based on the Labeled Property Graph model. The Key-Value model is based on simple key-value pairs.

It accommodates various data types through its schema-free nature, primarily working with JSON-formatted data stored in a binary format called *VelocyPack*[29] internally. Values in documents can be of a primitive type (`null`, `boolean`, `number`, `string`) that represents one value or of a compound type (`array`, `object`) that can contain multiple values.[30]

Internally, ArangoDB uses *RocksDB* storage engine – embeddable, persistent, key-value, log-structured store [36] optimized for fast storage. It is designed to be scalable to large datasets and is optimized for fast storage and retrieval of data (by using caches and indexes). It employs document-level locks allowing for concurrent

[28] https://arangodb.com/arangograph-insights-platform/
[29] https://github.com/arangodb/velocypack
[30] https://docs.arangodb.com/3.11/aql/fundamentals/data-types/

writes without blocking reads and uses a write-ahead log to ensure durability.[31]

Consistency ArangoDB ensures *ACID* properties using a single instance or, in a cluster, using *OneShard*[32] deployments, offering snapshot isolation and key-level pessimistic locking for write-write conflict detection. "This is more than many other NoSQL database systems support. In cluster mode, single-document operations are also fully ACID."[33] On the other hand, transactions are mostly optimized for short-running and small-volume operations, while long-running transactions can lead to performance degradation. Furthermore, the recommended document size is 50-75 kB.

Scalability ArangoDB is *Vertically* and *Horizontally* scalable.[34] The DBMS server will automatically use more threads if more CPUs are present, so scaling up is just adding more resources.

Sharding Sharding in ArangoDB is achieved via *Master-Master* replication.[35] Each server instance can host multiple shards as local collections, of which one is always the *leading shard* and the others are *replicas* of their respective leading shards stored in different servers. The *replication factor* number controls the number of replicas and the *Shard key* is used to distribute the collections across the shards via the use of hashing functions. "Not all use cases require horizontal scalability. In such cases, consider the OneShard feature as alternative to flexible sharding."

Replication Replication in ArangoDB can be done *synchronously* and *asynchronously*.[36] Synchronous replication, used for clusters, stores copies of a shard's data on another server and keeps them in sync (see Sharding 2.3.4). Asynchronous replication, used for *Active Failover*[37] setups, achieves eventual consistency by applying changes from the *Leader*'s (read/write instance) log to the *Followers* (read-only) at a later time.

[31] https://docs.arangodb.com/3.11/components/arangodb-server/storage-engine
[32] https://docs.arangodb.com/stable/deploy/oneshard/
[33] https://docs.arangodb.com/3.11/develop/transactions/limitations/#in-clusters
[34] https://docs.arangodb.com/stable/deploy/architecture/scalability/
[35] https://docs.arangodb.com/stable/deploy/architecture/data-sharding/
[36] https://docs.arangodb.com/stable/deploy/architecture/replication/
[37] https://docs.arangodb.com/stable/deploy/active-failover/

Schema ArangoDB is a *schema-free* system. It supports optional schema validation via *JSON Schema* [37] for documents.[38]

Aggregates ArangoDB is an *aggregate-oriented* system.

Entity types ArangoDB supports *Documents* as entity types.

Relations ArangoDB supports intra-document relations, inter-document relations, and relationships via *Edges* collection in the LPG model. Intra-document relations are achieved by embedding documents within documents, and inter-document relations are achieved by referencing documents from other documents.

Absence of value ArangoDB uses *null* to represent the absence of value.

Query language ArangoDB uses *ArangoDB Query Language (AQL)*, a declarative query language able to query multiple data models at once, regardless of whether one is dealing with documents, graphs, or key-value pairs. AQL is fully equipped to handle all CRUD operations.

The DDL features start with the creation of collections using the `db._create(collection,...)` function from the `db` object of the JavaScript [38] API[39], using the WEB Interface or HTTP API. The `db._createEdgeCollection()` function is used to create an *Edges* collection used in the Graph model. `db._update()` and `db._remove(collection)` functions can be used to update the collection's options/properties and drop collections, respectively. Afterward, the documents can be modified using the `INSERT`, `UPDATE`, and `REMOVE` statements, including `REPLACE` and `UPSERT` operations.

The central part of DML involves the use of `FOR doc IN docs RETURN doc` statement. `RETURN {attr1:val1,attr2:...}` allows to specify document projection, including the `RETURN DISTINCT` for unique projection of documents. `FILTER` allows restricting/selecting documents with given properties and supports a huge variety of (logical) operators and functions.

AQL shines in aggregating data, as showcased by its `COLLECT ... INTO` (or `WINDOW`) operation, which enables grouping by arbitrary criteria with optional aggregation, facilitating sophisticated analytical queries and data summaries.[40]

[38] https://docs.arangodb.com/stable/concepts/data-structure/documents/schema-validation/

[39] https://docs.arangodb.com/stable/develop/javascript-api/@arangodb/db-object/

[40] https://docs.arangodb.com/stable/aql/examples-and-query-patterns/grouping/#aggregation

`SORT`, `LIMIT offset, count` clauses are used for sorting, limiting, and skipping the results. The `LET` clause is used to define variables (or aliasing documents). Also, there is no special AQL clause for set operations, but rather the clever use of `FILTER` statements in different ways.[41]

AQL supports quite a straightforward declaration of a document join, whether it be between One-To-Many or Many-To-Many types of entity relationships. The use of `FOR doc2 IN coll2` within another `FOR doc1 IN coll1` (and so on) is a typical example of a One-To-Many join.

```
FOR doc1 IN coll1
  FOR doc2 IN coll2
    FILTER doc1._id == doc2._id
    RETURN {doc1, doc2}
```

Many-To-Many join can be modeled by using Embedded Lists of (other) document IDs inside our document and then subquerying (similarly to the One-To-Many situation) and filtering on the document ID.

```
FOR doc1 IN coll1
  LET joined = (FOR id2 IN doc1.embeddedList
                  FOR doc2 IN coll2 FILTER id2 == doc2._id)
  RETURN {doc1, joined}
```

Another example of a join, tested extensively in the Chapter 4 of this thesis, is the use of Edge Collections, which is a more efficient approach to model Many-To-Many relationships in the graph model. What's more, outer joins can be achieved by previous methods and then filtering on zero-length arrays.[42]

AQL supports Graph traversals by using

```
FOR vertex[, edge[, path]] IN [min[..max]]
INBOUND/OUTBOUND/ANY startVertex
GRAPH graphName
```

statement (more in Chapter 4).[43] Unlimited graph traversals are also supported but limited to the setting of the `max` parameter of the initial query.

A comparison of graph model terminology between AQL and Neo4j's Cypher (section 2.3.3) graph query language can be seen in table 2.2.[44]

[41] https://docs.arangodb.com/stable/aql/examples-and-query-patterns/diffing-two-documents/

[42] https://docs.arangodb.com/stable/aql/examples-and-query-patterns/joins/#outer-joins

[43] https://docs.arangodb.com/3.11/aql/graphs/traversals

[44] https://arangodb.com/learn/graphs/comparing-arangodb-aql-neo4j-cypher/

Table 2.2: Comparison of AQL and Cypher terminology.

AQL	Cypher
vertex	node
edge	relationship
collection	(group of nodes)
document	(node with properties)
document collection	node label
edge collection	relationship type
attribute	property
depth	hops
array	list
object	map

2.3.5 Cassandra

Cassandra [39] is a DBMS of the *Wide-Column* (columnar, column-oriented) family developed originally by Avinash Lakshman and Prashant Malik at Facebook [40], later open-sourced and now maintained by *Apache Software Foundation*. The current DB-Engines ranking gives Cassandra 12th place (March 2024). Cassandra is widely used in domains that require High Availability, Fault Tolerance, such as event logging, messaging, e-commerce, content management systems, and other applications that require fast writes and reads.

The system uses the Wide-Column model for data representation. The model is based on the concept of *Column Families* (tables) that contain *Rows* (records) and *Columns* (fields). Columns are the basic units of data storage, each consisting of a name-value (key-value) pair stored with a timestamp (for TTL, conflict resolution, etc.). Rows are key-linked collections of columns, and Column Families are collections of "similar" rows. All encompassed in a *Keyspace* (similar to a database in RDBMS) of a *Cluster*. The data is distributed across the cluster using a *partitioner* and a *replication strategy*.[45]

Cassandra inherits the majority of design introduced in *Amazon Dynamo*[46] and *Google Bigtable*[47] systems. Its storage engine is based on the *Log-Structured Merge-Tree* [36] data structure, which is optimized for write-heavy workloads.[48]

As Cassandra is a typed language, it supports everything from native types

[45] https://cassandra.apache.org/doc/4.1/cassandra/architecture/overview.html
[46] https://aws.amazon.com/dynamodb/
[47] https://cloud.google.com/bigtable
[48] https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html

(strings, numbers, booleans, date/times, etc.), to collections (lists, sets, maps), user-defined types (UDTs), and tuples. Notably, the use of collection types can lead to performance issues – as they are not indexed internally, access can lead to full scans of the data. It is also advised to use Sets over Lists, as writes to Sets are idempotent and never incur read-before-write penalties.[49]

Consistency Cassandra is a *BASE* storage system. It supports eventual consistency with tunable consistency levels which allow Cassandra to trade-off between availability and consistency.[50] Cassandra ensures atomicity and isolation at the row level. Writes are durable, recorded in memory as well as in a commit log on disk, and the system uses client-side timestamps to resolve conflicts, with the most recent update prevailing. The database can be customized to offer full consistency for particular operations, datacenters, or clusters.[51] It can also support lightweight transactions with linearizable consistency using the Paxos protocol.[52]

Scalability It is a *Vertically*, but mostly *Horizontally* scalable system. The system is designed to be fault-tolerant and highly available, with no single point of failure.

Sharding Since Cassandra is highly distributed, the data is stored across a cluster of nodes, distributed internally by the *partitioner*. It can be set to Murmur3Partitioner (default), RandomPartitioner, and OrderPreservingPartitioner. It uses each table's primary key, which consists of a partition key and clustering columns/keys, to determine the distribution. The partition key is hashed to determine the actual node in the cluster, and the clustering keys are used to sort the data within the partition (to speed up range queries).[53]

Replication Cassandra replicates all partitions to numerous nodes throughout the cluster via *Multi-Master* replication according to the *replication strategy* defined in the keyspace. The replication strategy can be set to SimpleStrategy (for single datacenter deployments) or NetworkTopologyStrategy (for multi-datacenter

[49] https://cassandra.apache.org/doc/4.1/cassandra/cql/types.html

[50] https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html#tunable-consistency

[51] https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlTransactionsDiffer.html

[52] https://cassandra.apache.org/doc/4.1/cassandra/architecture/guarantees.html#lightweight-transactions-with-linearizable-consistency

[53] https://cassandra.apache.org/doc/4.1/cassandra/data_modeling/intro.html

deployments).[54] Additionally, the *replication factor* can be set to control the number of replicas per range.

Schema Cassandra is a *schema-optional* system. Schema must be optimized for querying, so unlike in RDBMS, data must be denormalized and duplicated to support different queries.

Aggregates Cassandra is an *aggregate-oriented* system.

Entity types Cassandra supports *Tables (Column Families)* as entity types.

Relations Since Cassandra does not support the concept of foreign keys, relations between entities (tables) are modeled with *data denormalization* and the use of User-Defined Types (the concept of super column families have been deprecated and is no longer recommended).

Absence of value The `NULL` constant denotes the absence of value.

Query language Cassandra uses *Cassandra Query Language (CQL)* as its query language. CQL's resemblance to SQL syntax makes it easy for developers to learn and use. It provides a rich set of features for CRUD operations, querying, batch operations, and more. The language is designed to be simple and intuitive, with a focus on performance and scalability. It is also designed to be extensible, allowing developers to add custom functions and data types.

DDL operations in Cassandra are very similar to that of SQL. First, one must create keyspace using the `CREATE KEYSPACE` statement and define its mandatory option of *replication strategy* with its `replication_factor` discussed above. After its creation, `ALTER KEYSPACE` and `DROP KEYSPACE` can be used to further modify and delete the keyspace, respectively. Next, tables can be created using the `CREATE TABLE` statement, which must include the `PRIMARY KEY` with its partition and clustering parts. Additionally, a column can be declared `STATIC`, which makes it shared among all rows in the partition. Furthermore, table options can be set with the most notable being `CLUSTERING ORDER BY` to sort the data within the partition ascending or descending (since the use of `ORDER BY` is limited to the selected clustering columns ordering – and its reverse).[55] The use of right `Compaction`

[54] https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html#multi-master-replication-versioned-data-and-tunable-consistency

[55] https://cassandra.apache.org/doc/4.1/cassandra/cql/ddl.html

strategy can also be beneficial for further performance tuning.[56] Finally, the table can be altered/dropped using the `ALTER TABLE/DROP TABLE` statements.

Rows can be inserted into the table using the `INSERT` statement, updated using the `UPDATE` statement, and deleted using the `DELETE` statement. Unlike in SQL, each primary key uniquely identifies a row, so inserting a new one with the same primary key will `UPSERT` the existing one. `JSON` [18, 19] values are also supported, but limited.[57] `USING TTL` can be used to expire the row after a certain time. The `BATCH` statement can be used to group multiple DML operations into a single request operation, where all operations belonging to a given partition key are performed in isolation.[58] Batches are not meant to be the same as SQL transactions.

DML operations in Cassandra support the use of SQL's `SELECT`, `FROM`, `WHERE` clauses with important differences.[59] Selecting results using `WHERE` is limited to the partition key, clustering columns, and (secondary) indexed columns. First, the partition key has to be specified (by equality), and only then clustering columns can be restricted in contiguous order. The use of `ALLOW FILTERING` can be used to allow filtering on non-indexed columns, but it is discouraged due to unpredictable performance costs. It performs a full scan of all partitions, where the query performance is proportional to the amount of data returned. Results can be further grouped with `GROUP BY`, but only in groups sharing the primary key values in a defined order (partition key/s > clustering key/s), and then aggregated using the standard aggregate functions or user-defined aggregates.[60] Ordering can be done using `ORDER BY` on clustering columns, but only in specific order. The use of `LIMIT` and `PER PARTITION LIMIT` can be used to further limit the number of returned rows.

The most notable characteristic of CQL is that it does not support *joins* between tables. That is why the Query-Driven architecture is employed and data must be denormalized and duplicated to enable complex queries (spanning multiple relationships between entities/tables). This is a trade-off for the High Availability and Fault Tolerance that Cassandra provides.[61]

MapReduce is also supported using the *Hadoop* integration, which allows for the distributed processing of large data sets across clusters of nodes.

[56] https://cassandra.apache.org/doc/4.1/cassandra/operating/compaction/index.html

[57] https://cassandra.apache.org/doc/4.1/cassandra/cql/json.html

[58] https://cassandra.apache.org/doc/4.1/cassandra/cql/dml.html#batch_statement

[59] https://cassandra.apache.org/doc/4.1/cassandra/cql/dml.html#select-statement

[60] https://cassandra.apache.org/doc/4.1/cassandra/cql/functions.html

[61] https://cassandra.apache.org/doc/4.1/cassandra/data_modeling/data_modeling_rdbms.html

2.3.6 MongoDB

MongoDB is a *document*-based DBMS developed by *MongoDB Inc.* and is ranked 5th on DB-Engines (March 2024). The primary use cases are content management, real-time analytics, high-speed logging, etc. It is available as a self-hosted open-source Community Edition and a commercial Enterprise Advanced server managed locally, or using Software as a Service (SaaS) solution through *MongoDB Atlas*.[62]

MongoDB is a *document-oriented* DBMS that stores JSON-based [19, 18] documents with flexible schema. Internally, *BSON* (Binary JSON) is a binary representation of JSON documents (with a maximum size of 16 MB), which allows for more efficient storage and retrieval of data.[63] Each MongoDB instance can contain a database (like RDBMS's schema) which in turn consists of *collections* (like tables in RDBMS) that group related documents together. *Document* is a set of key-value pairs, where the keys are represented as strings and the values are either of BSON data type or another document/array. Documents can be nested, allowing for complex aggregates to be stored in a single document.

BSON supports all JSON data types, i.e. strings, numbers, null, arrays, and objects, with additional types like datetimes, or geospatial types, all of them represented in a binary format.

MongoDB uses a *WiredTiger* storage engine by default, which is optimized for most workloads, offering a document-level concurrency model, checkpointing, compression, and more. The Enterprise version also supports the *In-Memory* storage engine.[64] Certain use cases might require the use of an appropriate storage engine.

Consistency MongoDB's consistency model allows choosing between data redundancy and normalization, leveraging both *BASE* and *ACID* properties for different use cases (thus making it ACID-complaint[65]). For the majority of cases, where the application always reads and updates related data (aggregates) as one, data that is accessed together should be stored together. A write operation is atomic on the level of a single document, thus denormalization using (related) document embeddings (documents within a document) is encouraged. However, for cases where the related document is accessed often independently, or it changes frequently, or when the document sizes become unmanageable, normalization using inter-document references can be used.[66]

[62] https://www.mongodb.com/products
[63] https://bsonspec.org/
[64] https://www.mongodb.com/docs/manual/core/storage-engines/
[65] https://www.mongodb.com/databases/acid-compliance
[66] https://www.mongodb.com/docs/manual/data-modeling/data-consistency/

Furthermore, when the application must always return up-to-date data, MongoDB also supports *multi-document transactions* since version 4.0, allowing for multiple operations to be grouped and executed atomically.[67] On the other hand, the performance cost of a distributed transaction is typically higher than that of a single document (aggregate) write, so efficient schema design should not be overlooked (see Chapter 4 for further experimental comparison).[68]

Scalability MongoDB is a *Vertically* and *Horizontally* scalable system. While it is possible to scale up by increasing the processing power of a single node or cluster, sharing load across multiple nodes is more common in NoSQL systems.[69]

Sharding Partitioning distributes data across multiple shards (replica sets - see Replication) based on a shard key, either using "hashed" or "ranged" sharding strategy, which determines the shard it belongs to. The `mongos` (or query routers), interfaces for client applications, are then used to route queries to the appropriate shards. The *Config Servers* store cluster metadata, which also contain ranges for shard keys and the location of individual data.[70]

Replication *Master-slave* replication is achieved using replica sets, which are groups of `mongod` (server) instances that maintain the same data set. The primary node (master) accepts write operations, while the secondary nodes (slaves) replicate the primary's operations. Read requests can then be distributed across all nodes (controlled by *Read Preference* setting). If the primary fails (writes become unavailable), an election takes place to determine a new one out of all secondaries. The *Write Concern* can be set to control the number of nodes that must acknowledge a write operation before it becomes committed. Also, all primary's operations are recorded in the *oplog* (operations log) to ensure durability.[71]

Schema MongoDB is a *schema-free* system. Optional document schema validation can be enforced using JSON Schema [37] or using MQL's query operators.[72]

Aggregates MongoDB is an *aggregate-oriented* system.

Entity types MongoDB supports *Documents* as entity types.

[67] https://www.mongodb.com/docs/manual/core/transactions/
[68] https://www.mongodb.com/docs/manual/core/write-operations-atomicity/
[69] https://www.mongodb.com/basics/scaling
[70] https://www.mongodb.com/docs/manual/sharding/
[71] https://www.mongodb.com/docs/manual/replication/
[72] https://www.mongodb.com/docs/manual/core/schema-validation/

Relations As discussed in Consistency, relationships are modeled using intra-document relations which are achieved by embedding documents within documents, and inter-document relations that are achieved by referencing documents from other documents.[73]

Absence of value MongoDB uses *null* to represent the absence of a value.

Query language MongoDB provides its Query API (further referenced as *MongoDB Query Language (MQL)*), which serves as a foundation for all CRUD and aggregation operations. The language is optimized to work with JSON [19, 18] and JavaScript [38] in mind, though it offers many embeddings in a variety of other programming languages (Java, Python, C#, etc.), including the MongoDB Compass GUI tool and the *MongoDB Shell* (`mongosh`) (used in this thesis).[74]

The DDL operations start with the creation of a database using the `mongosh`'s `use` command, which switches to the specified database or creates a new one if it does not exist yet. Afterward, collections are created by any document "insert" operation or can be explicitly created using the `db.createCollection()` method, which also allows setting options for capped (insertion order preserved), clustered collections, or custom validation. After, the `.drop()` method is used to drop the collection. Collection update is not present, since the system does not enforce schema like in RDBMS's tables. Documents can be inserted using the `.insertOne()` or `.insertMany()` methods, updated using the `.updateOne()` or `.updateMany()` methods, and deleted using the `.deleteOne()` or `.deleteMany()` methods. All documents must contain the `_id` (indexed primary key) field of type ObjectId, which is unique within the collection and is automatically generated if not provided.[75]

The DML operations revolve around the `db.collection.find({attr1:val1, attr2:val2})` method, used to query the collection for documents based on provided query operators (criteria). Comparison, logical, element, evaluation, geospatial, bitwise and array operators can be used.[76] Extending find method with `.sort()`, `.limit()`, `.skip()` methods can be used to sort, limit, and skip the results. Furthermore, `.count()` return collection size and the `.distinct()` returns an array of unique values for a field across the collection (these are simple single-collection aggregation methods per se).

The `db.collection.aggregate()` method can be used to run complex aggregation operations on the collection using the *Aggregation Pipeline* array of *stages*,

[73] https://www.mongodb.com/docs/manual/data-modeling/concepts/embedding-vs-references/

[74] https://www.mongodb.com/docs/mongodb-shell/

[75] https://www.mongodb.com/docs/manual/tutorial/insert-documents/

[76] https://www.mongodb.com/docs/manual/reference/operator/query/

each stage transforming the documents in some way.[77] Main stages are `$match` and `$project` similar to find operation. Minimum, maximum, average, sum, and count operations can be performed using the `$group` stage. Set operations like `$unionWith`/`$setUnion`, `$setIntersection` and `$setDifference` stages are also supported.

The most interesting of aggregation stages are the `$lookup` stage, used to perform a left outer join between two collections (running slow multi-document transactions where required), and the `$graphLookup` stage used to perform recursive graph queries (or unlimited traversals with caveats) on multiple collections. Nesting queries can also be expressed using lookups, since they allow running an additional pipeline on the joined document.[78] These lookups have been studied extensively in Chapter 4 of this thesis, comparing them to the traditional denormalized joins.

MapReduce is also supported using the `.mapReduce()` method (deprecated since 5.0), though it is advised to use the aggregation pipelines now since they provide better performance and usability.[79]

2.4 Summary

This section summarizes the analysis presented in the previous sections by comparing the DBMS's individual features, DDL, and DML of the selected systems.

Table 2.3 compares individual features of DBMSs in terms of Consistency, Scalability, Sharding, Replication, Schema, Aggregates, Entity types, Relations, and the used representation of an Absence of value. Table 2.4 shows the Data Definition Language (DDL) features and Table 2.5 [41] shows the Data Manipulation Language (DML) features of individual DBMSs.

[77] https://www.mongodb.com/docs/manual/aggregation/
[78] https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup/
[79] https://www.mongodb.com/docs/manual/core/map-reduce/

Table 2.3: Comparison of individual features of Database Management Systems (DBMS)

	SQLite	**MySQL**	**Neo4j**	**ArangoDB**	**Cassandra**	**MongoDB**
Consistency	ACID	ACID	ACID	BASE / ACID (OneShard)	BASE	BASE / ACID
Scalability	"V"	V / H	V / H	V / H	V / H	V / H
Sharding	N	Y[1]	Y[2]	Y	Y	Y
Replication	N	Y	Y	Y	Y	Y
Schema	full	full	free	free	free	free
Aggregates	ignorant	ignorant	ignorant	oriented	oriented	oriented
Entity types	Relation	Relation	Vertex	Document	Table	Document
Relations	Foreign Key	Foreign Key	Edge	Reference / Embedded Doc; Edges collection	Denormalization + UDTs	Reference / Embedded Doc
Absence of value	metavalue	metavalue	-[3]	metavalue	metavalue	metavalue

[1] MySQL NDB Cluster [2] Neo4j Enterprise [3] Absence of a property

Table 2.4: Data Definition Language (DDL) features of individual Database Management Systems

	SQLite (SQL)	**MySQL (SQL)**	**Neo4j (Cypher)**	**ArangoDB (AQL)**	**Cassandra (CQL)**	**MongoDB (MQL)**
CREATE	CREATE TABLE	CREATE TABLE	-[1]	db._create[2]	CREATE TABLE / TYPE	createCollection
ALTER	ALTER TABLE	ALTER TABLE	-[1]	-[1]	ALTER TABLE	-[1]
DROP	DROP TABLE	DROP TABLE	-[1]	-[1]	DROP TABLE	drop
INSERT	INSERT INTO	INSERT INTO	CREATE / MERGE	INSERT .. INTO	INSERT	insertOne, insertMany, save
UPDATE	UPDATE	UPDATE	SET / REMOVE	UPDATE .. IN, REPLACE .. IN	UPDATE .. SET	updateOne, updateMany
DELETE	DELETE FROM	DELETE FROM	DELETE	REMOVE .. IN	DELETE	deleteOne, deleteMany

[1] Schema-free [2] Not part of AQL

Table 2.5: Data Modeling Language (DML) features of individual Database Management Systems

	SQLite (SQL)	**MySQL (SQL)**	**Neo4j (Cypher)**	**ArangoDB (AQL)**	**Cassandra (CQL)**	**MongoDB (MQL)**
Projection	SELECT	SELECT	RETURN	RETURN {attr1:val1,attr2:val2}	SELECT	$project, find(attr1:val1,attr2:val2)
Source	FROM	FROM	graph specification	FOR doc IN docs	FROM	db.[collection_name]
Selection	WHERE	WHERE	WHERE	FILTER	WHERE	$match, find()
Aggregation	GROUP BY ... HAVING	GROUP BY ... HAVING	count, min, max, avg	COLLECT ... INTO; WINDOW	GROUP BY	aggregation pipeline
Join	JOIN	JOIN	-	FOR a IN b FOR c IN d FILTER a.cId == c._id RETURN	-	$lookup
Graph Traversal	JOIN [4]	JOIN	MATCH	FOR v IN INBOUND/OUTBOUND ...	-	$graphLookup
Unlimited Traversal	WITH RECURSIVE	WITH RECURSIVE	[1]	FOR v IN 0..MAX	-	- ($graphLookup$ with limitations)
Optional	OUTER JOIN	OUTER JOIN	OPTIONAL MATCH	"outer joins"	-	-
Union	UNION	UNION	UNION	[42]	-	$unionWith, $setUnion (aggregation)
Intersection	INTERSECTION	INTERSECTION	apoc.coll.intersection	[42]	-	$setIntersection (aggregation)
Difference	EXCEPT	EXCEPT	WHERE NOT	[42]	-	$setDifference (aggregation)
Sorting	ORDER BY	ORDER BY	ORDER BY	SORT	ORDER BY	sort
Skipping	OFFSET	OFFSET	SKIP	LIMIT offset, count	-	skip
Limitation	LIMIT	LIMIT	LIMIT	LIMIT	LIMIT	limit
Distinct	DISTINCT	DISTINCT	DISTINCT	RETURN DISTINCT	DISTINCT	db.docs.distinct(...)
Aliasing	AS	AS	AS	LET doc = (...)	AS	"alias" : "$field"
Nesting	(SELECT ...)	(SELECT ...)	CALL {MATCH . . . }	FOR d IN docs FOR u IN d ...	-	-
MapReduce	- (GROUP BY ... HAVING)	- (GROUP BY ... HAVING)	- [2]	- (COLLECT ... INTO)	GROUP BY	.mapReduce[3]; aggregation pipeline

[1] everything is matched by default with no limitation [2] everything is aggregated by default [3] deprecated [4] maximum of 64 tables

3. ETL

ETL stands for Extract, Transform, Load. It is a process in which data is extracted from various sources, transformed into a format that is suitable for analysis, and then loaded into data storage, e.g. DBMS. In this chapter, we will discuss the ETL process used to generate, process, and import data used in the subsequent dynamic analysis in more detail (see Chapter 4). We will also discuss the tools and technologies used in the process.

We will start by choosing the data domain and specific data models used for testing in section 3.1. Afterward, we create an ER diagram for all tested data models, either normalized (i.e. relational and graph data model) or denormalized (i.e. columnar and document data model) based on chosen queries. For each of the tested DBMSs we present an individual UML diagram, which will include all entities, relationships, keys, attributes, and their data types.

After we know what we are doing, we will start by generating relational data using a pseudo-random generator script in section 3.2. Next, in section 3.3, we will use a series of processes, pipelines, and tools to transform data into their required data models and database-specific forms. This will be the most complex step.

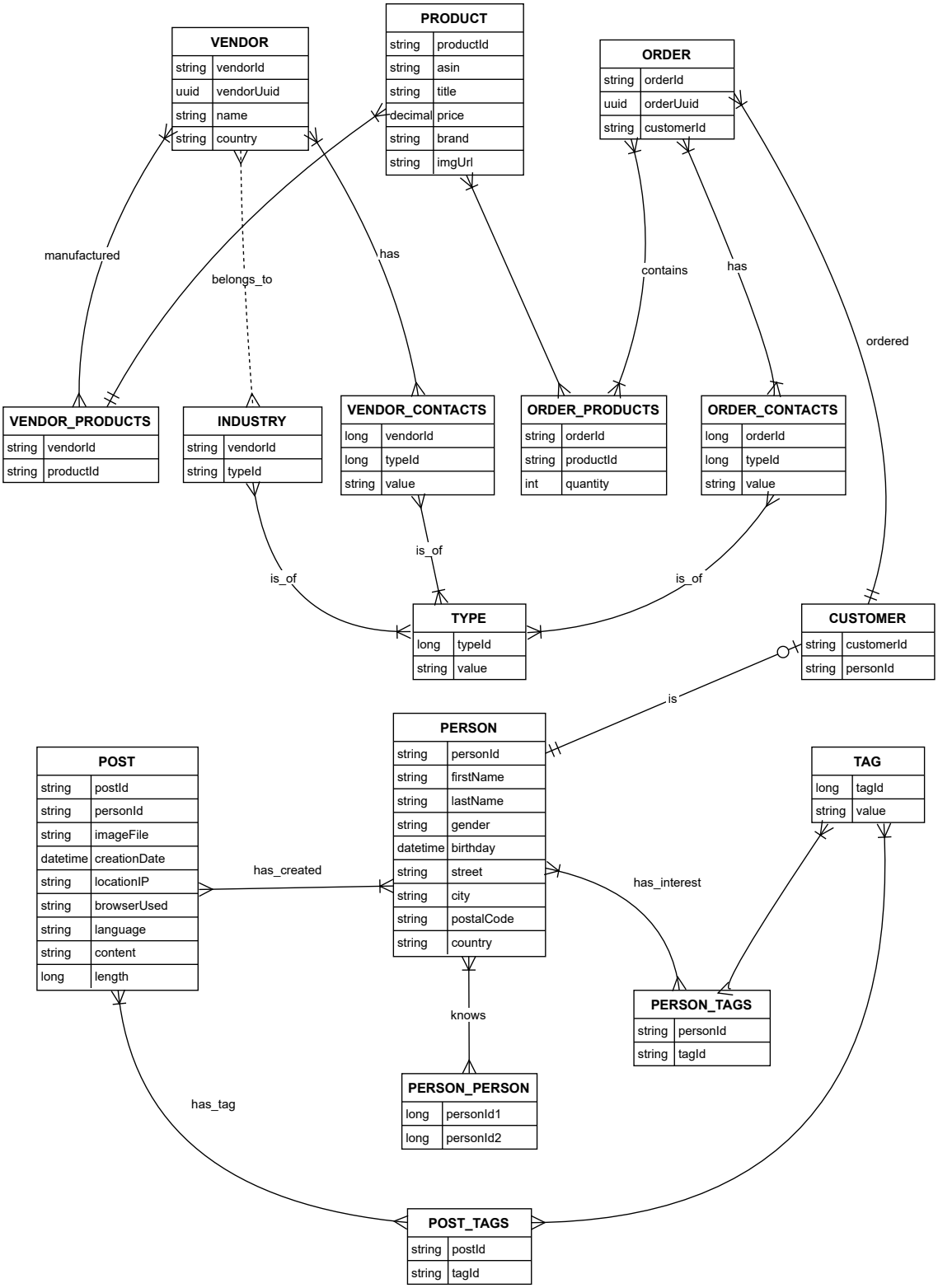
Lastly, in section 3.4, we will import the data using vendor-recommended import tools or other open-source utilities. Afterward, in section 3.5, we will include dataset sizes (entity/relationship counts) in tables 3.1, 3.2, 3.3, and 3.4, as well as approximate database sizes on disk in table 3.5. We will also try to record transformation and import times, and then present them in table 3.6.

3.1 Data Domain and Data Models

The data domain used in the ETL process is an e-commerce platform inspired by the multimodel benchmark *UniBench* [11]. The platform is an e-shop consisting of products, their vendors in certain industries, orders, customers, and a social network part that includes people, that can write posts ("reviews"), can have interest in some topics (tags), and can make relationships (friends) with other people. We tried to achieve as many real-life e-commerce aspects and data as possible, and that is why the model is so complex, encompassing most of the nowadays high variety, velocity, and data volume requirements.

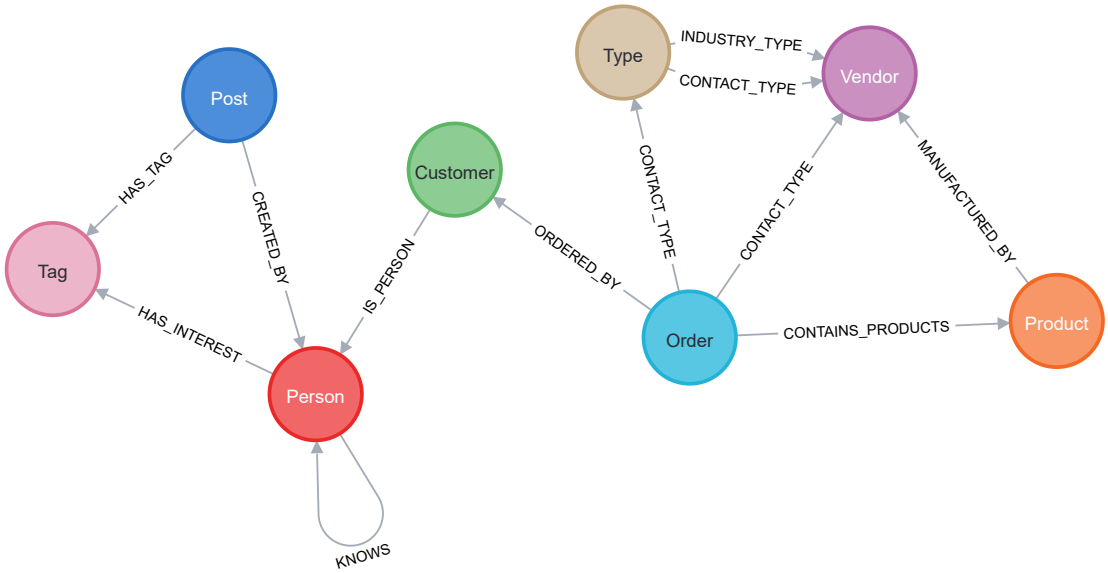
Figure 3.1 can be used to express most of the reality we try to model in the databases. It is an extended relational ER diagram, that is a high-level logical data model defining the relationships between entities and their attributes, but with additional entities to represent 1:N or M:N relationships.

Figure 3.1: Relational ER diagram of an e-commerce platform.



Next, we convert the relational schema into a graph schema in figure 3.2. Since the graph data model is a superset of the relational data model, the conversion can be quite straightforward. Entities are converted into nodes, and relationships are converted into edges.

Figure 3.2: Graph ER diagram of an e-commerce platform.



Subsequently, we will model the data in a Cassandra-specific columnar data model and a (MongoDB-specific) document data model. Both, the columnar and document models should be based on domain-specific and/or application-specific queries, hence the name — Query-driven database architecture. So before we dig into these models, we need to choose the queries that will be used to model the data.

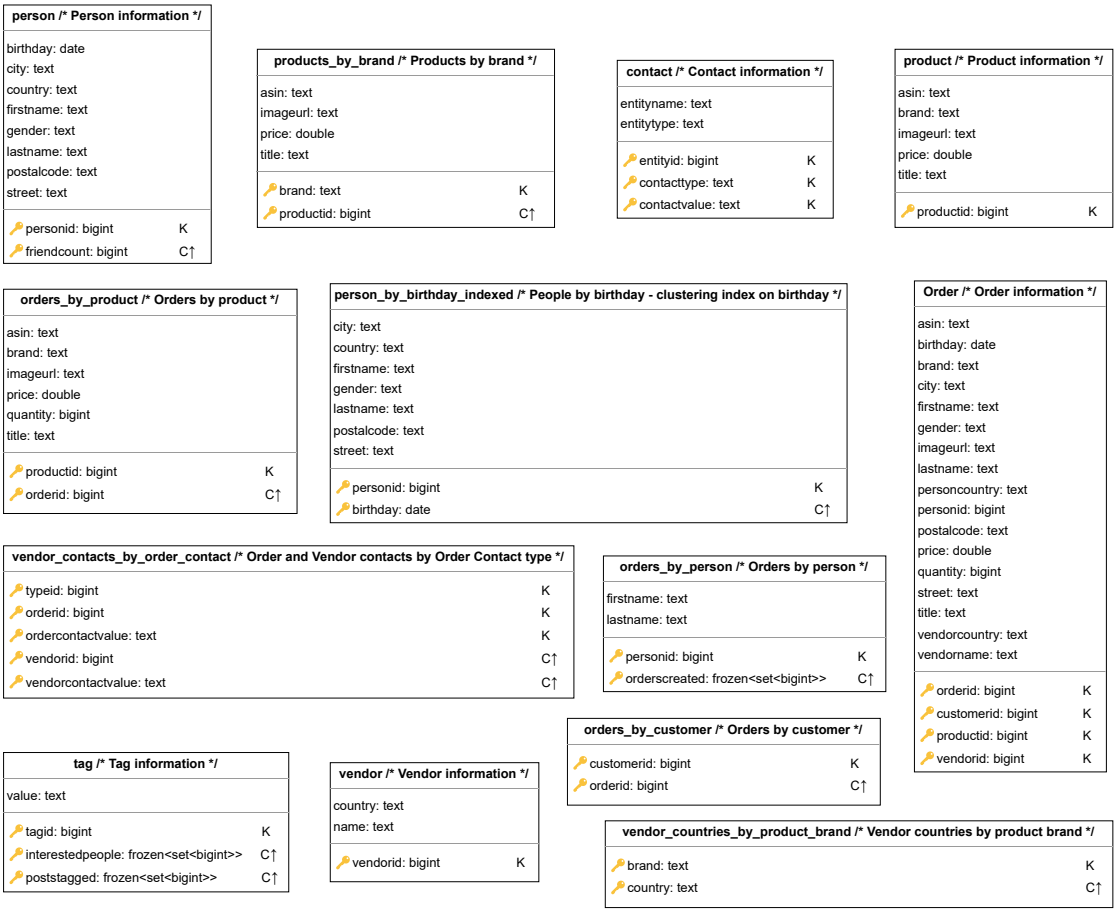
When choosing the queries, we need to consider all query "types" presented in table 2.5 to cover all possible data access patterns. We want to be able to compare the expressive power of individual query languages, so we want to be as query language agnostic as possible. We will use the queries presented in the following list.

List of Queries

1. **Selection, Projection, Source (of data)**
   - 1.1 **Non-Indexed Columns**: Select vendor with the name 'Bauch - Denesik'.
   - 1.2 **Non-Indexed Columns — Range Query**: Select people born between 1980-01-01 and 1990-12-31.
   - 1.3 **Indexed Columns**: Select vendor with the ID 24.
   - 1.4 **Indexed Columns — Range Query**: Select people born between 1980-01-01 and 1990-12-31.
2. **Aggregation**
   - 2.1 **COUNT**: Count the number of products per brand.
   - 2.2 **MAX**: Find the most expensive product per brand.
3. **Joins**
   - 3.1 **Non-Indexed Columns**: Join vendor and order contacts on the type of contact.
   - 3.2 **Indexed Columns**: Join all products with their orders.
   - 3.3 **Complex Join 1**: Retrieve all order details.
   - 3.4 **Complex Join 2**: Retrieve all people having more than 1 friend.
4. **Unlimited Traversal**
   - 4.1 **Neighborhood search**: Find all direct and indirect relationships between people up to a depth of 3.
   - 4.2 **Shortest path**: Find the shortest path between two people.
5. **Optional Traversal**: Get a list of all people and their friend count (0 if they have no friends).
6. **Union**: Get a list of contacts (email and phone) for both vendors and customers.
7. **Intersection**: Find common tags between posts and people.
8. **Difference**: Find people who have not made any orders.
9. **Sorting**
   - 9.1 **Non-Indexed Columns**: Sort products by brand.
   - 9.2 **Indexed Columns**: Sort products by brand.
10. **Distinct**: Find unique combinations of product brands and the countries of the vendors selling those products.
11. **MapReduce**: Find the number of orders per customer (only those who have made at least 1 order).

Now, that we have all the queries, we can start designing Cassandra columnar data model based on their own documentation tutorial[1]. The resulting schema, modeled using Chebotko notation[2] [43], is shown in figure 3.3. How these column families (tables) relate to the queries above can be seen in `cassandra/query.cql` file in the attached project folder (appendix A.1). Notably, queries 4.1, 4.2 and 9.1 have been skipped in Cassandra testing, since they are either impossible or extremely impractical to model with the columnar model (see the listing A.1 and source code comments for more information).

Figure 3.3: (Cassandra-specific) columnar UML Diagram of an e-commerce platform.

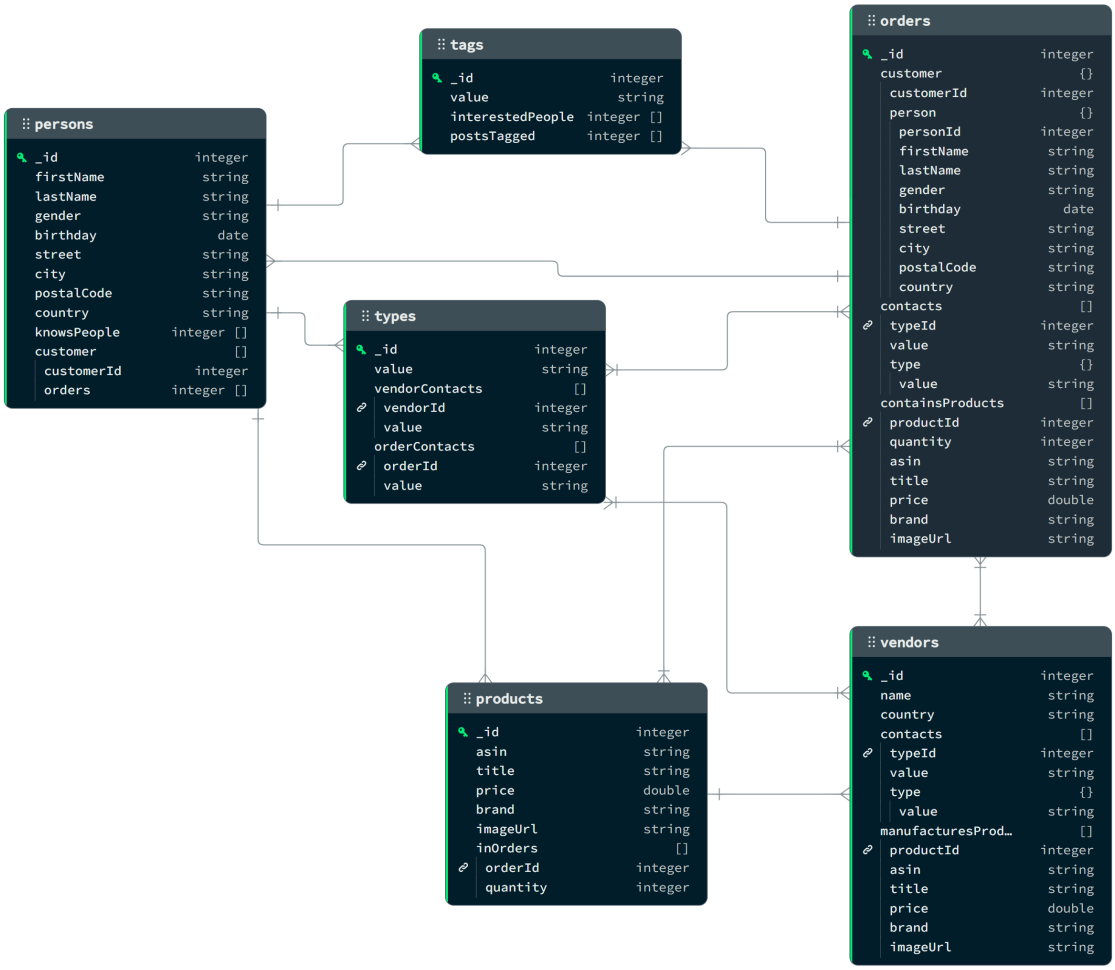


[1] https://cassandra.apache.org/doc/stable/cassandra/data_modeling/intro.html

[2] **K** represents *Partition Key* part, whereas **C↑/C↓** represents *Clustering Column* sorted by ASCending or DESCending order respectively

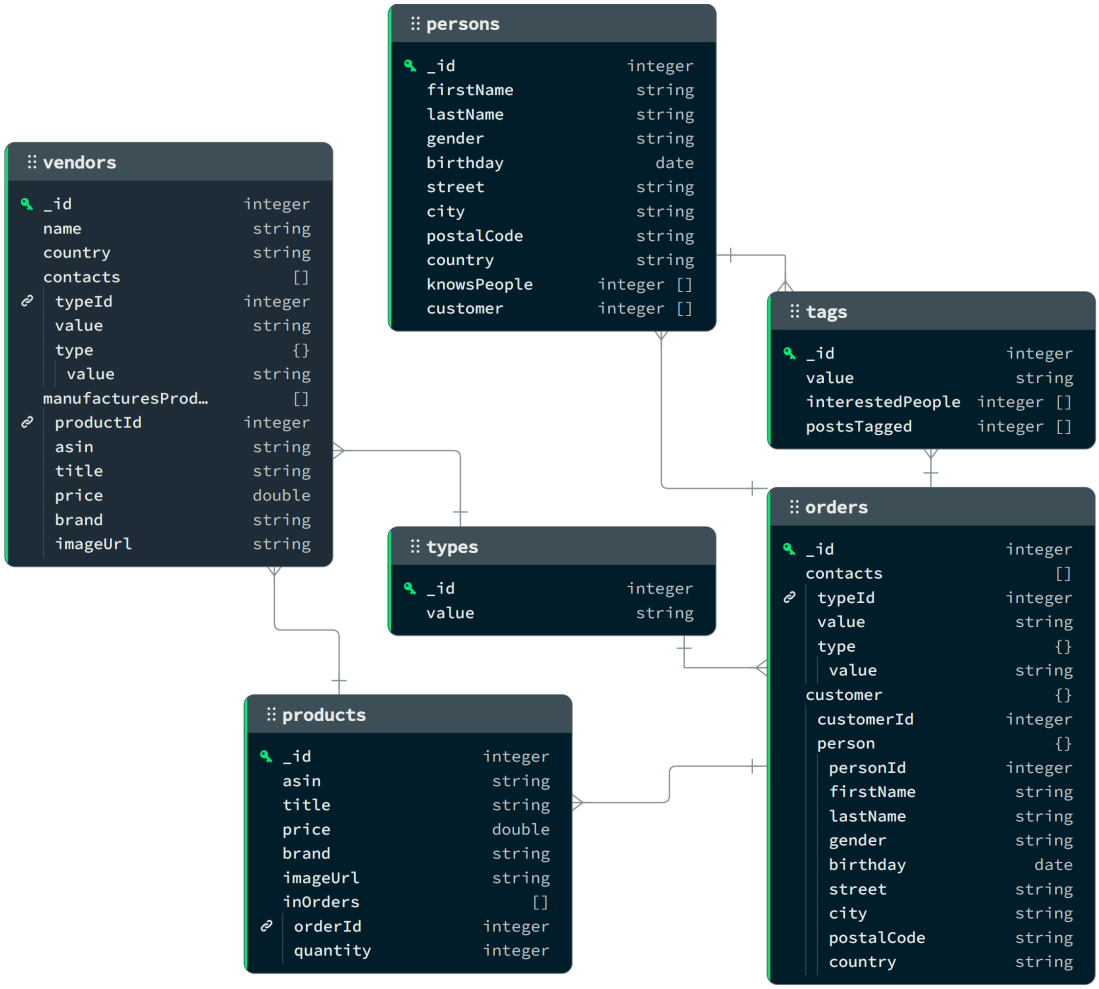
Finally, we will create DB schema for MongoDB's document data model, the results of which are seen in the next figure 3.4.

Figure 3.4: (MongoDB-specific) document ER Diagram of an e-commerce platform.



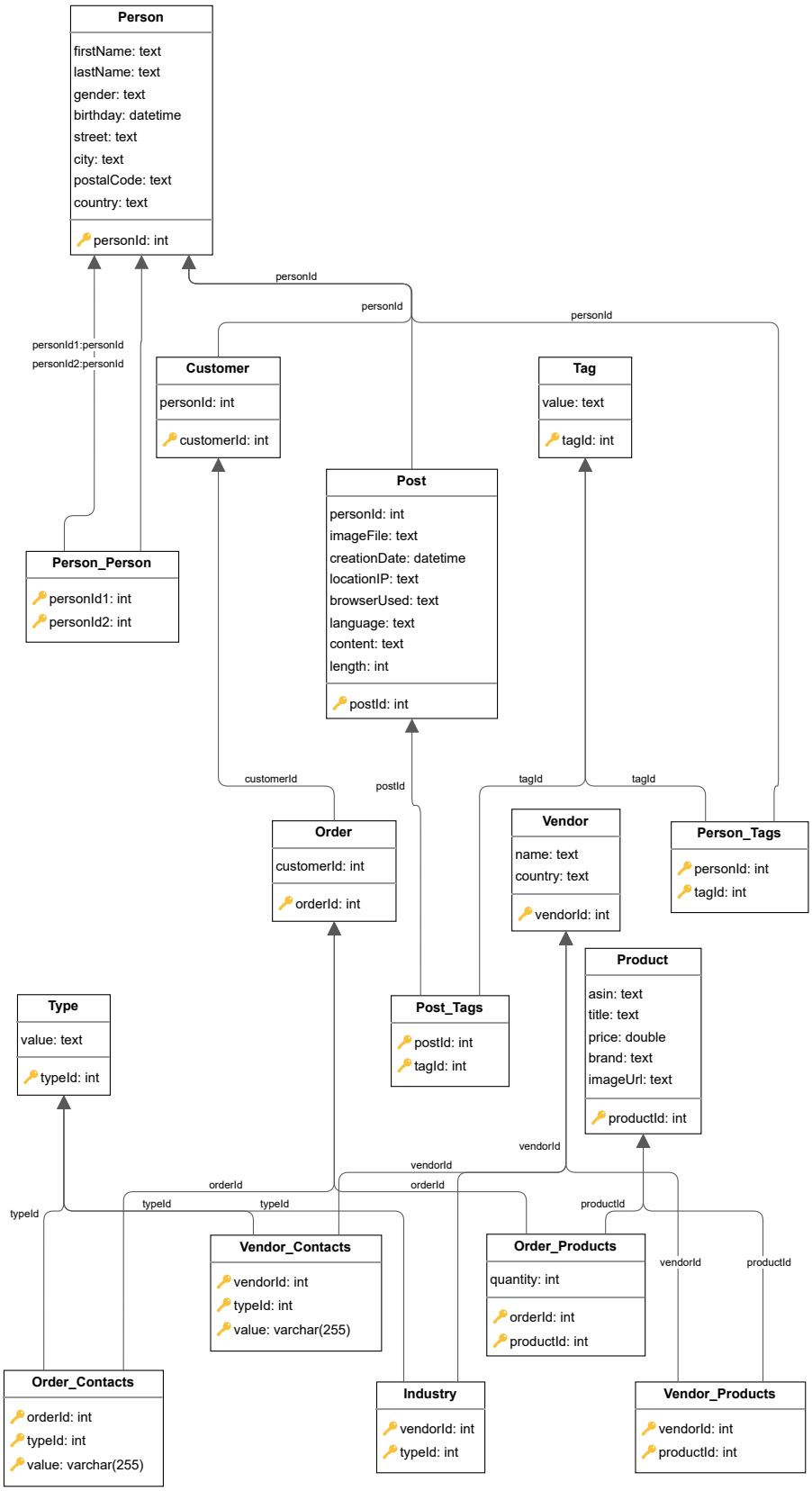
Notably, during MongoDB data import and query testing we ran into certain issues that needed to be addressed by slightly modifying the schema and respective queries. The modifications were necessary to avoid migration errors in the MongoDB Relational Migrator tool (section 3.3), which was used to import the data into MongoDB. The modified queries – 3.1 and 3.3 and some special mentions can be found in the listing A.2. The modified schema is shown in the figure 3.5.

Figure 3.5: Modified MongoDB schema due to reasons mentioned in listing A.2.



For completeness, we also present the UML diagram of the MySQL database with all entities, foreign keys and attributes in figure 3.6. SQLite differs from MySQL only slightly in the data types ("affinity types" — see section 2.3.2), so we will omit its UML diagram for brevity.

Figure 3.6: Relational UML diagram of the MySQL database used in the ETL process.



3.2 Data Generation (Export)

The data generation process is the first step in the ETL process. Using TypeScript [44] (i.e. JavaScript [38]) language, and Node.js (v20.8) [45] as its runtime, we created a script that generates pseudo-random relational data. The script source files are located in the `common/data-generator` NPM [46] package directory. The documentation and instructions can also be found there. For detailed information about the entire project structure, see the appendix A.1.

The `run.sh <entity_count>` BASH script takes the base number of entities to generate as a reference point (since some entities like Customers, Orders, and Types can be of different count). The output folder then contains CSV file for each table (or a single SQL dump from the old generator – see below).

An example of the Vendor entity generator function can be found in the listing A.3. The script uses the Faker.js NPM module [47], which is a JavaScript library that generates fake data for testing purposes. We also utilize the Scramjet framework [48] for parallel data streaming and processing using pipelines and Node.js Stream API[3]. The script is designed to be easily extendable and modifiable for other data model transformations and use cases. To be able to reproduce the datasets used in our experiments, we set a constant seed for the Faker's random number generator and subsequently, we call each generator function in a specific sequential order. For that reason, parallelization is only used for the data streaming and writing to the file system, not for the data generation itself.

The generator project was initially implemented in pure JavaScript, without any parallel streaming or pipelines, handling and storing raw strings of SQL clauses into main memory and thus quickly outgrowing server RAM memory limits, Node.js maximum heap size and JavaScript internal string and array length limits. The ScramJet framework was then used to solve these issues by streaming data from the generator to the file system using a series of parallel pipes, thus reducing the memory footprint and increasing the performance of the generator. Additionally, we increased the Node.js default heap size to 16 GB with `node --max-old-spac⌋ e-size=16384` flag. Furthermore, CSV files replaced the previously used singular SQL dump (for all tables).

We also decided to limit the maximum number of products to be equal to the `<entity_count>` parameter, as the increasing number of products caused MongoDB to crash during the ETL process. For reasons of reproducibility, we include the old version of the generator in the `common/data-generator/data-generator-old.js` file, ran with the `common/data-generator/run-old.sh` script. To showcase what was changed, we include a diff in the listing 3.1. This difference is also reflected in the section 3.5 and subsequently in the execution times for

[3] https://nodejs.org/docs/latest-v20.x/api/stream.html

queries 3.3 and 10 in the Chapter 4.

Also, we would like to mention that due to the little changes in the Vendor-Products generator function, the generated data attributes are not consistent with lower-record datasets (e.g. first 1k vendors in 256k generation are not the same as the first 1k vendors in 512k generation in terms of attribute values). This is due to the random nature of the Faker.js library and the seed setting. This only reflects in query 1.1, where the `'Bauch - Denesik'` vendor name is not present in the 512k dataset (and further), but is present in the 1k, 4k, 128k and 256k datasets. Since the non-indexed property (vendor name) does not exist, the query will return an empty result set.

Listing 3.1: Omitted part of code from the Vendor-Products generator.

```
// With probability of 0.7 assign some previous Products to Vendor
productsAssigned !== 0 && faker.helpers.maybe(() => {
  let productIds = Array.from({ length: productsAssigned - 1 }, (value,
  ↪  index) => index + 1);
  let productCountPerVendor = faker.number.int(
    { min: 0, max: 5 < productsAssigned ? 5 : productsAssigned - 1 }
  );
  for (let j = 0; j < productCountPerVendor; j++) {
    const randomIndex = faker.number.int({ min: 0, max: productIds.length - 1
    ↪  });
    const chosenProductId = productIds[randomIndex];

    vendorProducts.push(`(${vendorId}, ${chosenProductId})`);
    const brandCountry = brandVendorsByProductId[chosenProductId];
    countriesByBrand[brandCountry.brand].add(brandCountry.vendorCountry);

    productIds.splice(randomIndex, 1);
  }
}, { probability: 0.7 });
```

3.3 Data Transformation

The data transformation process is the most complex step of the ETL process. It involves converting the generated relational data into the required data models for each DBMS. The whole process is visualized in figure 3.7, with some detailed steps described in appendix A.1.

For the sake of reproducibility, isolation, and simplicity, Docker (v25.0.3)[4] and

[4] https://www.docker.com/

Docker Compose (v2.24.5) have been utilized to create a containerized environment for the whole ETL process and testing. The Docker Compose YAML specification is declared in the `docker-compose.yml` file in the project root directory. The file defines services for MySQL[5], SQLite[6], Neo4j[7], ArangoDB[8], Cassandra[9], and MongoDB[10]. Each service is based on an official Docker image distribution created by their respective vendors and published on Docker Hub[11] (except for SQLite, which is published by the community). The database folders are mounted as volumes to the containers, including the internal database files and configuration files. The `docker compose up` command is used to jump-start the whole environment. The file content is shown in the listing A.4.

[5] https://hub.docker.com/layers/library/mysql/8.1.0/images/sha256-c458b26f2f9f9fe086ed75d1f8db8e2dde371801403f2c7da9be4fb228a2944a [Accessed: 29 April 2024]

[6] https://hub.docker.com/layers/keinos/sqlite3/3.42.0/images/sha256-cc7aa975b234eb06ddccdc6d2debd1e401f1d1447c6e352aa5fbab519a36acb8 [Accessed: 29 April 2024]

[7] https://hub.docker.com/layers/library/neo4j/5.12.0/images/sha256-56182061dba6477e38b38fab9228a8d8b3fd379bb54e9000924285aaa15f1ae2 [Accessed: 29 April 2024]

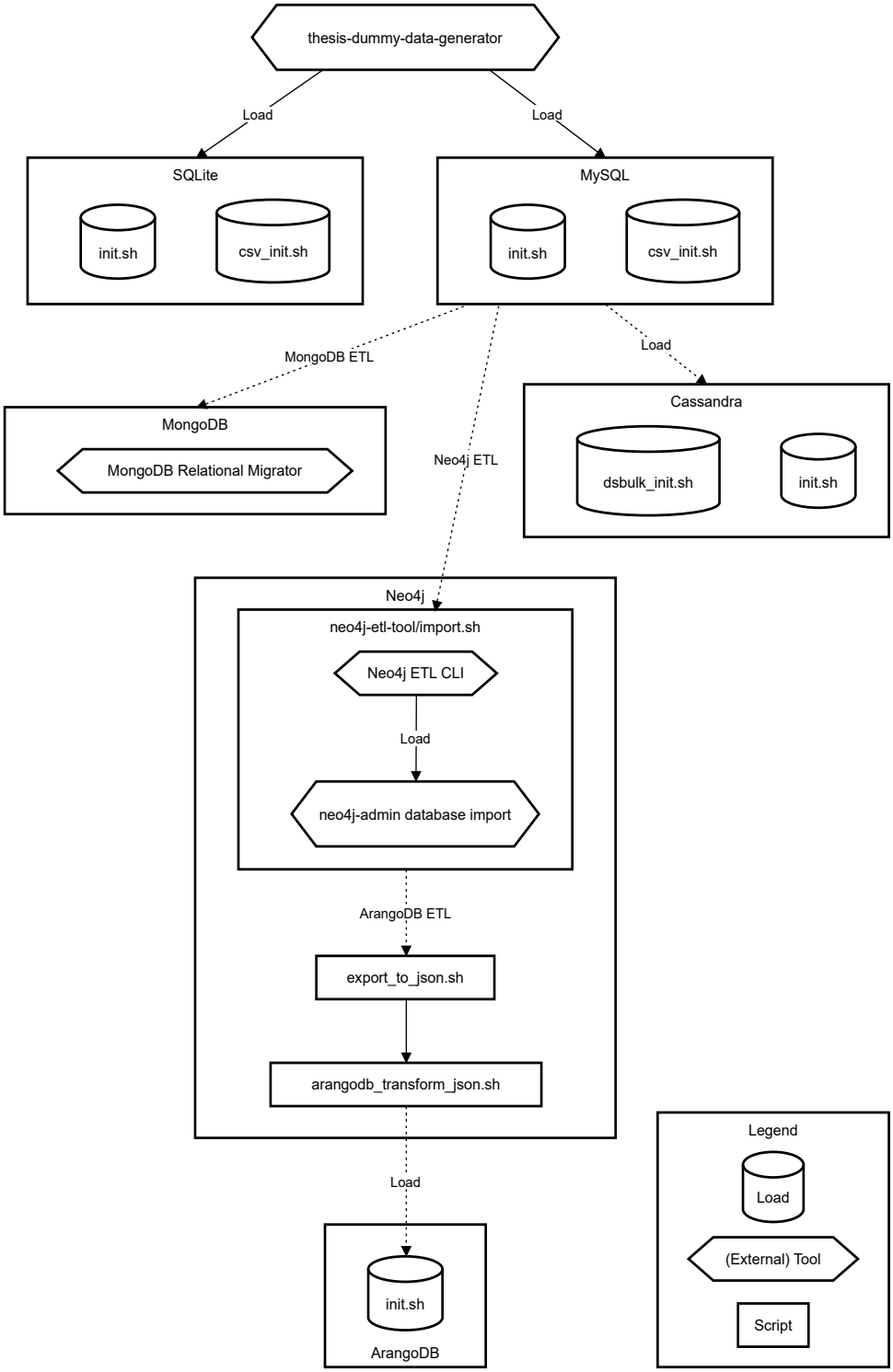
[8] https://hub.docker.com/layers/library/arangodb/3.11.3/images/sha256-085b45e8c56d5d4114e409482694d40fc8d1678c6b5d98d774bab31193034d6a [Accessed: 29 April 2024]

[9] https://hub.docker.com/layers/library/cassandra/4.1.3/images/sha256-02e877c29ebb8e11fbdc5b30c372769f7efa1cc3db4f9837218ccc881318ff86 [Accessed: 29 April 2024]

[10] https://hub.docker.com/layers/library/mongo/7.0.2/images/sha256-075e0577e2989efee25f8e6cd615ae1ce84da1f8c79adc3557aaf253dbf7a5e0 [Accessed: 29 April 2024]

[11] https://hub.docker.com/

Figure 3.7: ETL process flowchart.



Before we start the individual transformations, we need to install and set up the necessary tools and libraries. Some of them have to be installed and set up manually, while others are already downloaded and set up in the project folder for the reader's convenience. CLI tools like `mysql`, `sqlite3`, `cypher-shell`, `arangosh` `cqlsh`, and `mongosh` are already included in the used Docker images. That also includes the `mongoimport`, `mongodump` and `mongorestore` utilities for MongoDB, `arangoimport` for ArangoDB, and `neo4j-admin` for Neo4j (which can also be installed separately, see below). The external tools and libraries that need to be installed and set up are:

- **Neo4j ETL Tool CLI** (v1.6.0): A tool for extracting, transforming and loading data from relational databases (i.e. MySQL) into Neo4j [49]. Configuration files (including mapping) are located in the `neo4j/neo4j-etl-tool` directory.
- **neo4j-admin** (v5.12.0): A CLI utility for managing Neo4j databases, especially bulk loading Big Data. [50]. Provided in the Neo4j Docker image.
- **MongoDB Relational Migrator** (v1.5.0): An enterprise-grade tool for migrating/syncing schema and relational data from e.g. MySQL to MongoDB [51]. Mapping files are located in the `mongodb/relational-migrator` directory.
- **DataStax Bulk Loader (dsbulk)** (v1.11.0): A CLI utility for loading, counting (and unloading) large amounts of data into Cassandra [52].

These tools must be configured and initialized according to the official documentation. For convenience, most of the tools are already pre-configured by running their respective scripts (see figure 3.7). Neo4j ETL Tool CLI is already set up via `neo4j/mysql_ecommerce_mapping.json` by using the `neo4j/neo4j-etl-tool/import.sh` script. On the other side, MongoDB Relational Migrator has to be configured manually, by importing project file `mongodb/relational-migrator/ecommerce-mapping.relmig` (for $<$ 128k experiments) or `mongodb/ecommerce-modified-orders-types-persons.relmig` (for $\geq$ 128k experiments) into the Relational Migrator GUI when creating a new project.

The arrows and horizontal layers in figure 3.7 represent the data flow and the (possibly parallel) transformation steps. Note, that some `*.sh` scripts have to be manually modified before execution (e.g. setting up the data dump directory or entity count). The comments in the scripts should guide the user further through the process.

Notably, Neo4j ETL Tool and `neo4j/neo4j-etl-tool/import.sh` was initially configured not to use `neo4j-admin database import`, but rather use the

`cypher:fromSQL` method, which exports relational data to CSV in batches and transactions (listing 3.2). This method turned out to be extremely slow, where 4k dataset entity volumes took more than 2 weeks to process. The `neo4j-admin` method was established in 256k volume experiments.

Listing 3.2: Old and ineffective version of `neo4j/neo4j-etl-tool/import.sh` setup.

```
./neo4j-etl-cli-1.6.0/bin/neo4j-etl export \
    --mapping-file mysql_ecommerce_mapping.json \
    --rdbms:password test \
    --rdbms:user test \
    --rdbms:url "jdbc:mysql://localhost:3306/ecommerce?autoReconnect=true&useSSL=false \
        &useCursorFetch=true&allowPublicKeyRetrieval=true" \
    --options-file import-tool-options.json \
    --using cypher:fromSQL \
    --unwind-batch-size 1000 \
    --tx-batch-size 10000 \
    --neo4j:url neo4j://localhost:7687 \
    --neo4j:user neo4j \
    --neo4j:password neo4j
```

3.4 Data Loading (Import)

The data loading process is the final step in the ETL process. In the case of MongoDB and Neo4j, the MongoDB Relational Migrator and Neo4j ETL Tool + neo4j-admin take care of the loading phase as well. For MySQL and SQLite we use either the `init.sh` or `csv_init.sh` script, depending on the output from the data generation process (section 3.2). The data must be in the required directories set in the script variables.

Notably, Cassandra's `cassandra/init.sh` singular CQL dump import using `cqlsh -f ./queries/"$data_file".cql` was initially used up to 256k entity volumes. The CQL file contained single row `INSERT INTO` statements without any bulk loading setup. Even the 4k dataset took more than 1 day to import, so we switched to the `dsbulk` utility instead.

During the loading process (including the transformation and possibly extract phase), we also measure the time it takes to import the data into the database. The time is measured in seconds, and results are stored in the `logs/` directory.

3.5 ETL Statistics

In this section, we present the results of the ETL process. We tried to scale the datasets to different sizes to see how the DBMSs behave with increasing data volumes. Initially, we started with smaller datasets (1k, 4k, and 128k entities) to build the ETL pipelines and tweak the tools, configurations and scripts to work with the data. Afterward, we scaled the datasets to 256k, 512k and 1M24k entities to see how the DBMSs behave with larger data volumes.

First, we present the generated datasets and their sizes. The dataset sizes for each entity and relationship are presented in subsequent tables 3.1, 3.2, 3.3, and 3.4.

Next, we determine rough estimates of database file sizes on disk using various tools and methods. The results are presented in table 3.5. The sizes are measured in megabytes and are rounded to the nearest integer. Notably, Neo4j does not provide any direct way to measure the database size on disk, so we used the `du -hc /var/lib/neo4j/data/databases/neo4j/*store.db` command to measure the size of the whole database directory. Furthermore, Cassandra provides the `nodetool tablestats` command to get a rough estimate of the entire keyspace from the `Space used (total)` property.[12] For MongoDB we use the `db.stats().totalSize` database command (which should be more accurate).[13] For SQLite, we use the `sqlite3_analyzer` utility (included in the project directory) with `Bytes of user payload stored` property.[14] For MySQL we use just a simple query found in `mysql/queries/table_sizes_in_mb.sql` utilizing the `information_schema` database.[15] And lastly, for ArangoDB we use `coll.figures().documentsSize` property summed for each collection invoked by the `arangodb/get_collection_sizes.sh` script.[16]

In the end, we show approximate measurements of the import and/or processing times for each DBMS in table 3.6. The times are measured in seconds and are rounded to the nearest integer. Note, that the times may be variably inaccurate due to mostly using BASH's `date` command, which measures the time of the whole process start and end. For MySQL and SQLite, we included integrity constraints (foreign keys) in the schema definition so the SQL dump import using `init.sh` can be influenced by integrity constraint checking. On the other hand, the `csv_init.sh`

[12] https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/tools/toolsTablestats.html

[13] https://www.mongodb.com/docs/manual/reference/command/dbStats/

[14] https://www.sqlite.org/sqlanalyze.html

[15] https://dev.mysql.com/doc/mysql-infoschema-excerpt/8.0/en/information-schema-introduction.html

[16] https://docs.arangodb.com/3.11/develop/javascript-api/@arangodb/collection-object/#collectionfiguresdetails

explicitly turns it off. Other systems do not include any schema constraints and they are not checked during import or are automatically disabled by the ETL tools used. The testing environment is discussed in the section 4.1.

Table 3.1: Dataset sizes for SQLite and MySQL.

Dataset / Table	Customer[1]	Industry[1]	Order[1]	Person	Post	Product	Tag	Type	Vendor
1k	774	1499	1568	1000	1000	1000	1000	13	1000
4k	2071	5978	4097	4000	4000	4000	4000	13	4000
128k	90031	191869	180319	128000	128000	128000	128000	13	128000
256k	170627	383829	341365	256000	256000	256000	256000	13	256000
512k	371187	1023699	742467	512000	512000	512000	512000	13	512000
1M24k	953371	2047593	1905923	1024000	1024000	1024000	1024000	13	1024000

Dataset / Table	Order_Contacts[2]	Order_Products[2]	Person_Person[2]	Person_Tags[2]	Post_Tags[2]	Vendor_Contacts[2]	Vendor_Products[2]
1k	4704	4777	5140	4790	5026	2037	2719
4k	12291	12212	20114	20192	19699	8005	11212
128k	540957	541120	641403	639129	640300	256090	352397
256k	1024095	1025494	1279600	1281585	1279277	511961	702504
512k	2227401	2225484	2561390	2557056	2562686	1023509	512000
1M24k	5717769	5717344	5125904	5113446	5118511	2048689	1024000

[1] The entity count is higher/lower than the base entity count to reflect a real-world ecommerce platform (e.g. Orders are the most important in an e-shop, so we have more, etc.) (section 3.1)
[2] The relationship count is higher/lower than the base entity count to reflect real-world M:N relationships between entities in an e-commerce platform (section 3.1)

Table 3.2: Dataset sizes for Cassandra.

Dataset / Table	Products_by_brand	Vendor_countries_by_product_brand	Orders_by_customer	Vendor_contacts_by_order_contact	Product	Vendor	Person
1k	1000	1949	1568	3194016	1000	1000	1000
4k	4000	8070	4097	0	4000	4000	4000
128k	128000	241167	180319	0	128000	128000	128000
256k	256000	70295	341365	0	256000	256000	256000
512k	512000	131735	742467	0	512000	512000	512000
1M24k	1024000	234906	1905923	0	1024000	1024000	1024000

Dataset / Table	Tag	Orders_by_person	Person_by_birthday_indexed	Orders_by_product	Contact	Order
1k	1000	1000	1000	4777	6741	12950
4k	4000	4000	4000	12212	20296	34140
128k	128000	128000	128000	541120	797047	1487271
256k	249404	256000	256000	1025494	1535754	1025494
512k	512000	512000	512000	2225484	3250910	2225484
1M24k	1024000	1024000	1024000	5717344	7766458	5717344

Table 3.3: Dataset sizes for Neo4j and ArangoDB.

Dataset / Nodes	**Customer / customers**	**Product / products**	**Post / posts**	**Vendor / vendors**	
1k	774	1000	1000	1000	
4k	2071	4000	4000	4000	
128k	90031	128000	128000	128000	
256k	170627	256000	256000	256000	
512k	371187	512000	512000	512000	
1M24k	953371	1024000	1024000	1024000	
Dataset / Nodes	**Type / types**	**Tag / tags**	**Order / orders**	**Person / persons**	
1k	13	1000	1568	1000	
4k	13	4000	4097	4000	
128k	13	128000	180319	128000	
256k	13	256000	341365	256000	
512k	13	512000	742467	512000	
1M24k	13	1024000	1905923	1024000	
Dataset / Edges	**IS_PERSON / isPerson**	**CREATED_BY / createdBy**	**ORDERED_BY / orderedBy**	**HAS_TAG / hasTag**	**HAS_INTEREST / hasInterest**
1k	774	1000	1568	5026	4790
4k	2071	4000	4097	19699	20192
128k	90031	128000	180319	640300	639129
256k	170627	256000	341365	1279277	1281585
512k	371187	512000	742467	2562686	2557056
1M24k	953371	1024000	1905923	5118511	5113446
Dataset / Edges	**KNOWS / knows**	**CONTAINS_PRODUCTS / containsProducts**	**MANUFACTURED_BY / manufacturedBy**	**CONTACT_TYPE / contactType**	**INDUSTRY_TYPE / industryType**
1k	5140	4777	2719	6741	1499
4k	20114	12212	11212	20296	5978
128k	641403	541120	352397	797047	191869
256k	1279600	1025494	702504	1536056	383829
512k	2561390	2225484	512000	3250910	1023699
1M24k	5125904	5717344	1024000	7766458	2047593

Table 3.4: Dataset sizes for MongoDB.

Dataset / Collection	orders	persons	products	tags	types
1k	1568	1000	1000	1000	13
4k	4097	4000	4000	4000	13
128k	180319	128000	128000	128000	13
256k	341365	256000	256000	256000	13
512k	742467	512000	512000	512000	13
1M24k	1905923	1024000	1024000	1024000	13

Table 3.5: Approximate database sizes on disk measured in MBs.

DB Size [MB]	SQLite[1]	MySQL[2]	Neo4j[3]	ArangoDB[4]	Cassandra[5]	MongoDB[6]
1000	1	2	4	2	5	2
4000	5	10	14	19	13	7
128000	167	441	494	229	502	216
256000	333	850	923	631	620	285
512000	451	1398	1638	831	1262	861
1024000	968	3005	3584	1774	3107	1867

[1] `sqlite3_analyzer` – "Bytes of user payload stored" [2] `mysql/queries/table_sizes_in_mb.sql`
[3] `df -hs`, includes indices, extremely inaccurate [4] `coll.figures().documentsSize`
[5] `cassandra/table_stats.sh` – "Space used (total)" [6] `db.stats().totalSize`, compressed, docs + indices

Table 3.6: Approximate import times for individual load or processing stages measured in seconds.

Import time [s]	SQLite	MySQL	Neo4j ETL Tool	neo4j-admin database import	ArangoDB	Cassandra	MongoDB Relational Migrator
1k	1	1	1	5	3	106 [1]	11
4k	1	2	2	5	5	62	60
128k	14	60	19	13	65	144	300
256k	35	126	42	17	98	222	600
512k	50	290	86	23	151	234	1500
1M24k	115	610	166	43	300	454	4860

[1] The time includes the import of massive `Vendor_contacts_by_order_contact` table, which is not present in larger datasets.

4. Dynamic Analysis

In this chapter, we will dynamically analyze the performance of the selected DBMSs described in section 2.1. We will run a series of experiments by testing queries chosen in section 3.1, modeled using underlying data models and individual query languages.

Firstly, we will describe the testing environment in section 4.1. Then, we will discuss the methodology and statistics we will use to demonstrate findings in section 4.2. Subsequently, we will measure the execution times of these queries, and present the results in the form of tables 4.1 and 4.2, and in the form of charts 4.3, 4.4, 4.5, and 4.6. We will also analyze the results, draw conclusions and present recommendations based on the performance of the DBMSs in section 4.3.

4.1 Testing Environment

The experiments will be performed on a virtual machine running on the VMWare[1] infrastructure with the following hardware specifications:

- **OS:** Ubuntu 20.04 LTS
- **CPU:** Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz (8 core)
- **RAM:** 32 GB
- **Storage:** 80 GB SSD

As already mentioned in section 3.3, we are using Docker and Docker Compose to set up the testing environment. The `docker-compose.yml` attached in listing A.4 shows the definition of the used services. We are using the vendor-recommended official Docker images found on the Docker Hub (SQLite being an exception). The database configuration was kept as close to the default as possible, with some exceptions where we had to adjust the configuration to fit the requirements of the ETL process or queries. The configuration was either set as environment variables or as mounted configuration files (kept in `<dbms>/data` directories). For all systems, we are using default single-node deployments.

[1] https://www.vmware.com/

4.2 Methodology

For each DBMS we modeled our chosen queries in the respective query language using the knowledge we gathered in the chapter 2. The queries were created in a way to show the capabilities, expressive power and efficiency of the query languages. We tried to cover most of the DML features, query types, access patterns and clauses/expressions presented in table 2.5.

Across DBMSs, the queries were designed to be as similar as possible to ensure a fair comparison, while still taking into account their unique features. Logically, the data retrieved should be the same, though the format of the output may differ (e.g. MongoDB's embedded document vs SQL's joined tables). The queries were designed to be as simple as possible, utilizing as few features as possible (or only the necessary) and trying to use vendor-recommended query modeling practices found in the documentation.

If a query uses additional index or other optimization hints, it is explicitly stated in the query description (comment). Most of the systems enforce the use of primary keys, i.e. the data is indexed by default. MySQL and SQLite use classic primary keys on columns. Notably, Neo4j uses auto-generated `LOOKUP INDEX`[2] for each node/relationship label, whereas ArangoDB automatically indexes each `_id`, `_key` fields for all documents and also `_from`, `_to` fields for edges.[3] To conclude, MongoDB uses the `_id`[4] field as a collection's primary key and Cassandra has its primary key concept defined in the schema (see section 2.3.5). For complete schema information, check section 3.1.

For query testing, we disabled query caching and/or cleared the cache before each query execution. MySQL has it off by default.[5] SQLite doesn't support the concept of query caching, only page caching for the whole database.[6] Neo4j has it off by default, but to ensure its absence, we added certain environment variables[7] into the `docker-compose.yml` file. ArangoDB Docker image has it off by default, but the cache can also be explicitly disabled.[8] Cassandra allows caching partition keys and rows differently[9], so we disabled both directly in the schema definition.

[2] https://neo4j.com/docs/cypher-manual/current/indexes/search-performance-indexes/managing-indexes/#create-lookup-index

[3] https://docs.arangodb.com/3.11/index-and-search/indexing/basics

[4] https://www.mongodb.com/docs/manual/indexes/#default-index

[5] https://dev.mysql.com/doc/refman/5.7/en/query-cache-configuration.html

[6] https://www.sqlite.org/pragma.html#pragma_cache_size

[7] https://neo4j.com/docs/cypher-manual/current/query-caches/

[8] https://docs.arangodb.com/3.11/aql/execution-and-performance/caching-query-results/

[9] https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/operations/opsSetCaching.html

Lastly, MongoDB supports only query plan caching[10], so to be sure nothing is cached, we used `.clear()` on the query plan cache before each execution.

Furthermore, we set a specific *timeout threshold* to 300 seconds (5 minutes) for each query execution. If a query hit the threshold or Did Not Finish (DNF) in general (an exception occurred), the query was terminated, and the execution time was set to 300 seconds. Some DBMSs have a specific setting on how to handle timeouts but for those that do not, custom methods had to be implemented. For instance, `sqlite3` had to be called with external `timeout 300` shell command. For MongoDB, simple queries utilize the `.maxTimeMS()` method, but for aggregation pipelines we had to use `setTimeout()` function in JavaScript.

To eliminate bias and ensure accuracy, we ran each query 20 times. Some extremes (i.e. minimum and maximum) for each testing cycle were discarded, and the average was calculated. The processing and filtering script can be found in `common/filter_results.ipynb` as a Jupyter notebook[11] using Pandas[12] and NumPy[13] packages. The results are presented in the subsequent section.

4.3 Results

In this section, we will present the results of the experiments. We will show the measured statistics of the queries executed on the selected DBMSs. The visualization is done in the form of tables 4.1 and 4.2, and in the form of charts 4.3, 4.4, 4.5, and 4.6 to provide a clear comparison of the average execution times in seconds and to show which DBMS performed better for each query type (lower execution time is better). Note that, the query names are hyphened instead of dot-separated.

[10] https://www.mongodb.com/docs/manual/reference/method/js-plan-cache/
[11] https://jupyter.org/
[12] https://pandas.pydata.org/
[13] https://numpy.org/

Figure 4.1: Table visualization of the query execution times for record volumes 1k, 4k and 128k.

Average DB query execution times per record volume

volume	db	Query (8+8.2 merged) 1-1	1-2	1-3	1-4	2-1	2-2	3-1	3-2	3-3	3-4	4-1	4-2	5	6	7	8	8-1	9-1	9-2	10	11
1k	sqlite	0.00	0.00	0.00	0.00	0.00	0.00	4.34	0.01	0.70	0.00	5.76	300.00 DNF	0.01	0.02	0.01	0.00		0.01	0.00	0.01	0.00
	mysql	0.00	0.00	0.00	0.00	0.00	0.00	1.67	0.02	0.06	0.01	16.60	300.00 DNF	0.01	0.03	0.02	0.01		0.00	0.00	0.02	0.00
	neo4j	0.05	0.05	0.02	0.04	0.01	0.02	28.54	0.05	0.19	0.03	4.73	0.03	0.04	0.08	0.32	0.04		0.01	0.01	0.04	0.02
	arangodb	0.00	0.00	0.00	0.00	0.00	0.00	45.96	0.07	0.19	0.05	15.33	0.00	0.02	0.09	0.08	0.01		0.00	0.00	0.03	0.00
	cassandra	0.01	0.01	0.00	0.01	0.01	0.01	0.00	0.00	0.01	0.00			0.00	0.00	0.00	0.01			0.00	0.01	0.01
	mongodb	0.00	0.00	0.00	0.00	0.00	0.00	4.69	0.00	0.37	0.06	2.41	0.02	0.00	0.03	0.00	0.00	0.01	0.00	0.00	0.01	0.00
4k	sqlite	0.00	0.00	0.00	0.00	0.00	0.01	43.99	0.04	6.77	0.02	39.95	300.00 DNF	0.02	0.05	0.04	0.01		0.01	0.01	0.03	0.00
	mysql	0.01	0.01	0.00	0.01	0.01	0.01	15.45	0.04	0.17	0.03	87.23	300.00 DNF	0.04	0.08	0.05	0.01		0.02	0.01	0.05	0.01
	neo4j	0.03	0.04	0.02	0.03	0.02	0.02	300.00 DNF	0.11	1.09	0.06	58.04	0.03	0.11	0.15	5.73	0.06		0.02	0.01	0.05	0.03
	arangodb	0.00	0.01	0.00	0.01	0.01	0.01	300.00 DNF	0.20	1.12	0.28	144.54	0.00	0.04	0.28	0.42	0.04		0.01	0.00	0.14	0.01
	cassandra	0.02	0.01	0.00	0.00	0.01	0.01		0.00	0.00	0.00			0.00	0.00	0.00	0.00			0.00	0.00	0.00
	mongodb	0.00	0.00	0.00	0.00	0.00	0.00	48.24	0.00	0.79	0.22	13.94	0.13	0.01	0.09	0.01	0.00	0.04	0.01	0.01	0.05	0.00
128k	sqlite	0.02	0.05	0.00	0.07	0.12	0.13	300.00 DNF	1.85	300.00 DNF	0.43	300.00 DNF	300.00 DNF	1.00	2.13	2.14	0.27		0.40	0.24	0.87	0.12
	mysql	0.07	0.14	0.00	0.14	0.20	0.21	300.00 DNF	2.16	9.97	0.93	300.00 DNF	300.00 DNF	1.36	8.33	3.97	0.49		0.37	0.24	2.38	0.25
	neo4j	0.07	0.12	0.05	0.04	0.09	0.15	300.00 DNF	2.87	18.02	1.33	300.00 DNF	0.09	2.68	3.09	300.00 DNF	0.80		0.24	0.10	1.15	0.20
	arangodb	0.03	0.15	0.00	0.15	0.15	0.19	300.00 DNF	6.31	20.67	8.07	300.00 DNF	0.07	0.89	8.56	10.38	1.21		0.29	0.10	3.63	0.35
	cassandra	0.32	0.01	0.00	0.00	0.00	0.01		0.00	0.01	0.00			0.00	0.00	0.00	0.00			0.00	0.00	0.00
	mongodb	0.06	0.06	0.00	0.06	0.10	0.14	300.00 DNF	0.06	300.00 DNF	5.23	300.00 DNF	122.63	0.24	3.09	0.13	0.05	300.00 DNF	0.19	0.14	1.49	0.23

Average query execution time in seconds broken down by Query (8+8.2 merged) vs. (record) volume and (database) db. Stronger color tint shows higher average execution time. The cells are labeled by the average value (rounded to 2 decimals) and the data is filtered to keep non-Null values only. White cells represent queries that were not tested. Value 300 represents Timeout threshold (DNF=Did Not Finish). Some cells include special notes in footnotes.

Averag.. 0.00 10.00

Figure 4.2: Table visualization of the query execution times for record volumes 256k, 512k and 1M24k.

Average DB query execution times per record volume

volume	db	Query (8+8.2 merged) 1-1	1-2	1-3	1-4	2-1	2-2	3-1	3-2	3-3	3-4	4-1	4-2	5	6	7	8	8-1	9-1	9-2	10	11
256k	sqlite	0.04	0.11	0.00	0.15	0.22	0.24	300.00 DNF	4.00	300.00 DNF	0.83	300.00 DNF	300.00 DNF	2.48	4.52	5.50	0.54		0.79	0.45	2.11	0.24
	mysql	0.15	0.31	0.00	0.31	0.46	0.47	300.00 DNF	5.62	25.02	2.13	300.00 DNF	300.00 DNF	3.35	18.77	9.34	0.97		0.82	0.53	5.70	0.48
	neo4j	0.15	0.30	0.02	0.19	0.18	0.30	300.00 DNF	5.78	38.68	2.83	300.00 DNF	0.16	5.73	7.07	300.00 DNF	1.60		0.55	0.22	2.48	0.42
	arangodb	0.07	0.29	0.00	0.26	0.22	0.33	300.00 DNF	13.61	40.86	16.41	300.00 DNF	0.19	1.70	15.78	22.00	2.22		0.46	0.15	8.33	0.75
	cassandra	0.78	0.01	0.00	0.01	0.01	0.01		0.00	0.01	0.00			0.00	0.00	0.00	0.01			0.00	0.00	0.00
	mongodb	0.10	0.10	0.00	0.13	0.15	0.18	300.00 DNF,#1	0.07	300.00 DNF	10.80	300.00 DNF	300.00 DNF,#2	0.40	5.65	0.26	0.10	300.00 DNF	0.32	0.23	3.10	0.28
512k	sqlite	0.06	0.21	0.00	0.32	0.47	0.51	300.00 DNF	7.99	300.00 DNF	1.54	300.00 DNF	300.00 DNF	4.81	8.44	10.39	1.07		1.65	0.96	0.75	0.54
	mysql	0.27	0.52	0.00	0.50	0.80	0.83	300.00 DNF	6.35	23.69	3.14	300.00 DNF	300.00 DNF	6.26	38.28	21.02	3.21		1.24	0.66	2.80	1.03
	neo4j	0.26	0.63	0.02	0.37	0.35	0.59	300.00 DNF	12.98	37.74	5.99	300.00 DNF	0.31	11.40	16.60	300.00 DNF	3.23		1.18	0.48	1.69	0.96
	arangodb	0.14	0.58	0.00	0.51	0.52	0.60	300.00 DNF	32.92	53.16	33.48	300.00 DNF	0.27	3.13	36.08	45.78	4.85		1.02	0.30	6.33	1.67
	cassandra	1.85	0.01	0.00	0.00	0.01	0.01		0.00	0.01	0.00			0.00	0.00	0.00	0.00			0.00	0.00	0.00
	mongodb	0.21	0.20	0.00	0.29	0.31	0.38	300.00 DNF,#1	0.16	300.00 DNF	21.54	300.00 DNF	300.00 DNF,#2	0.89	12.31	0.58	0.23	300.00 DNF	1.20	0.48	1.09	0.66
1M24k	sqlite	0.13	0.43	0.00	0.81	1.18	1.28	300.00 DNF	20.89	300.00 DNF	3.30	300.00 DNF	300.00 DNF	10.13	20.26	22.33	2.32		3.26	1.96	1.80	1.69
	mysql	0.65	1.27	0.00	1.27	1.88	1.92	300.00 DNF	47.01	133.16	7.89	300.00 DNF	300.00 DNF	51.89	111.85	44.02	10.28		3.23	2.02	7.82	6.91
	neo4j	0.41	0.69	0.30	0.07	0.69	1.17	300.00 DNF	300.00 DNF	300.00 DNF	14.12	300.00 DNF	0.61	24.93	300.00 DNF	300.00 DNF	6.77		2.23	0.82	3.98	2.36
	arangodb	0.27	0.91	0.00	1.17	0.96	1.16	300.00 DNF	78.06	146.85	69.19	300.00 DNF	0.50	6.35	87.44	95.39	11.50		2.40	0.59	12.38	4.52
	cassandra	3.60	0.01	0.00	0.00	0.01	0.01		0.00	0.01	0.00			0.00	0.00	0.00	0.01			0.00	0.00	0.00
	mongodb	0.40	0.47	0.00	0.68	0.50	0.67	300.00 DNF,#1	0.30	300.00 DNF	44.29	300.00 DNF	300.00 DNF,#2	1.77	30.38	1.06	0.44	300.00 DNF	2.70	0.94	2.26	2.16

#1: MongoServerError: Used too much memory for a single array. Memory limit: 104857600 bytes. The array contains 186021 elements and is of size 104857151 bytes. The element being added has size 574 bytes.
#2: MongoServerError: $graphLookup reached maximum memory consumption.

Average query execution time in seconds broken down by Query (8+8.2 merged) vs. (record) volume and (database) db. Stronger color tint shows higher average execution time. The cells are labeled by the average value (rounded to 2 decimals) and the data is filtered to keep non-Null values only. White cells represent queries that were not tested. Value 300 represents Timeout threshold (DNF=Did Not Finish). Some cells include special notes in footnotes.

Averag.. 0.00 10.00

Figure 4.3: Logarithmically scaled visualization of the average execution time per record volume for queries 1.1 — 2.2.

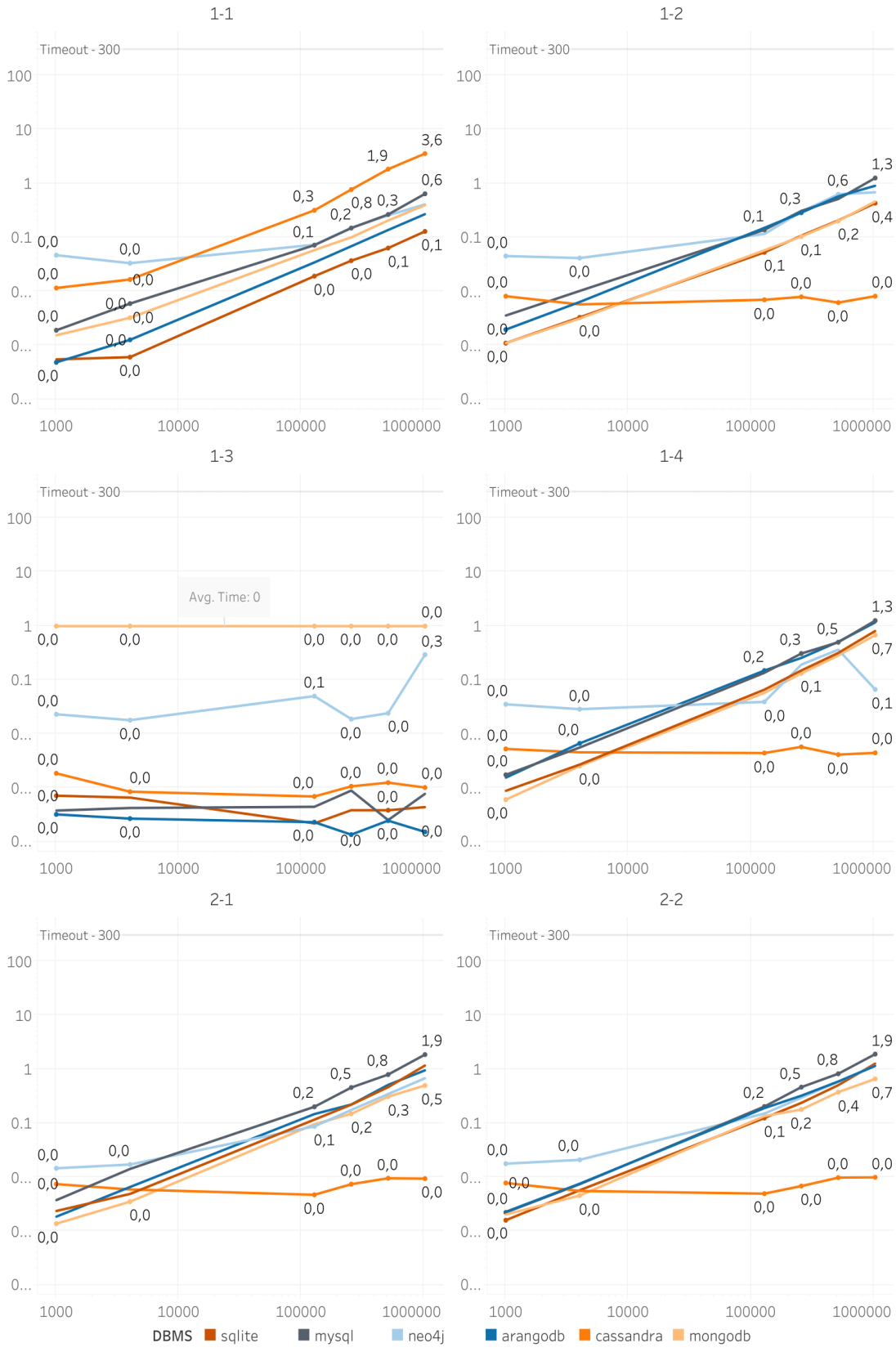


Figure 4.4: Logarithmically scaled visualization of the average execution time per record volume for queries 3-1 — 4-2

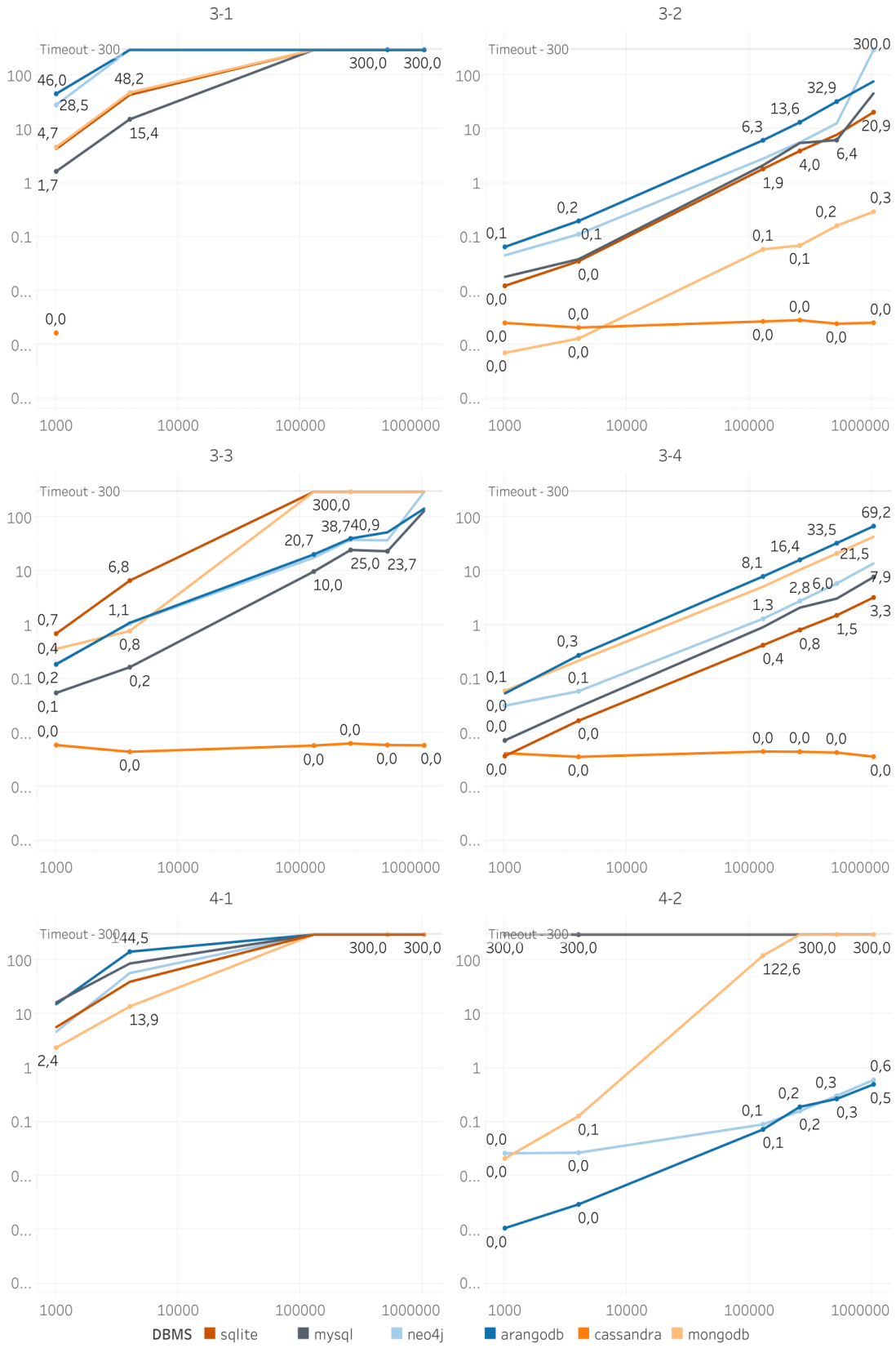


Figure 4.5: Logarithmically scaled visualization of the average execution time per record volume for queries 5 — 9-2

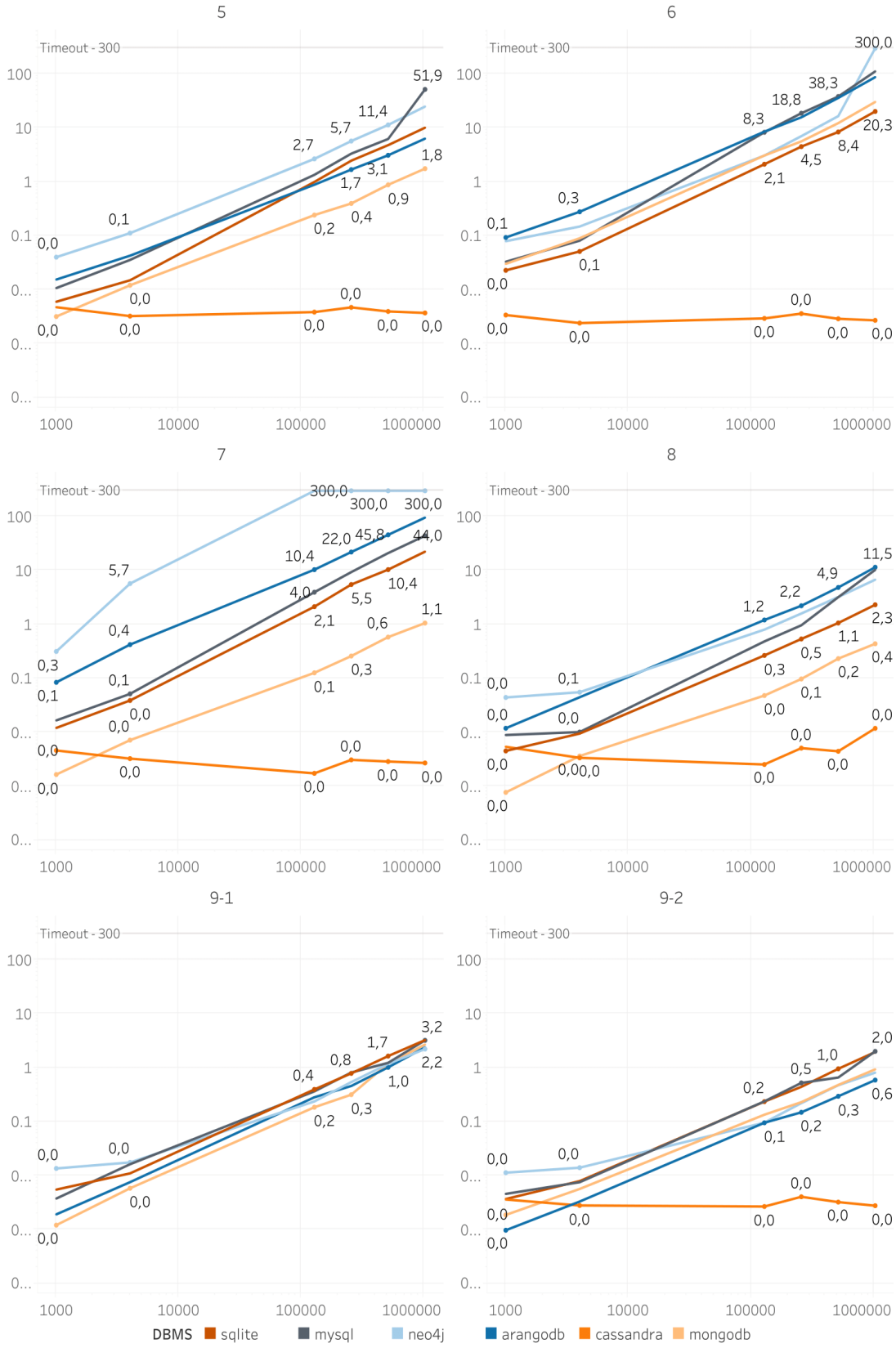
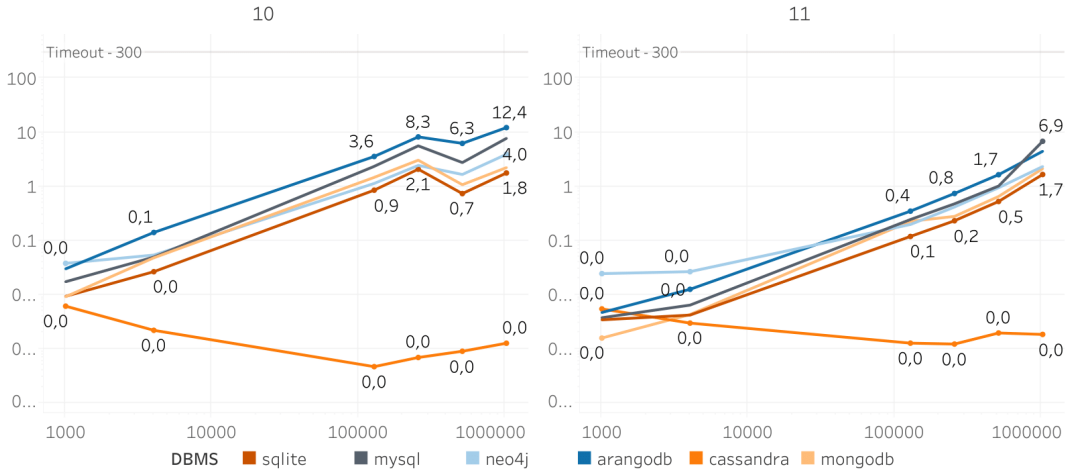


Figure 4.6: Logarithmically scaled visualization of the average execution time per record volume for queries 10 and 11



4.4 Summary and Recommendations

In this section, we will give some insights into the experimental results in section 4.3 and conclude what performs better in which context (data access pattern). For each query as in section 3.1, we will provide a brief summary of the results, the best and worst performing DBMS for that query and the possible reasons behind such performance. At the end in section 4.4.1, we will provide recommendations on how to choose the best DBMS for a given scenario.

1. **Selection, Projection, Source (of data)** (figure 4.3)

   1.1 **Non-Indexed Columns**: Select vendor with the name 'Bauch - Denesik'.

   Best: SQLite, (ArangoDB, MongoDB, Neo4j, MySQL) — This one does not have a clear winner.

   Worst: Cassandra — Since we must use `ALLOW FILTERING` for non-indexed columns, Cassandra performed a full table scan. Note that, after 256k experiments we modified the generator slightly and the 'Bauch - Denesik' vendor name was not present in the dataset. Even so, Cassandra had to scan the entire cluster for the non-indexed column and thus performed the worst.

   1.2 **Non-Indexed Columns — Range Query**: Select people born between 1980-01-01 and 1990-12-31.

Best: Cassandra — Cassandra is optimized for range queries, but the use of `ALLOW FILTERING` could still be a bottleneck if the range is too large.

Worst: Inconclusive

1.3 **Indexed Columns**: Select vendor with the ID 24.

Best: All

Worst: Inconclusive — Neo4j performed the worst, but the difference from other systems was not significant.

Comments: Since the ID is indexed in all systems, the performance was similar across all systems. MongoDB profiling seems to always output 0 seconds for this query, but the real execution time could be comparable to other systems.

1.4 **Indexed Columns — Range Query**: Select people born between 1980-01-01 and 1990-12-31.

Best: Cassandra — As already said, Cassandra is heavily optimized for range queries. However, if the dataset is small, the performance is similar to other systems.

Worst: Inconclusive

2. **Aggregation** (figure 4.3)

2.1 **COUNT**: Count the number of products per brand.

Best: All, (Cassandra)

Worst: None

2.2 **MAX**: Find the most expensive product per brand.

Best: All, (Cassandra)

Worst: None

3. **Joins** (figure 4.4)

3.1 **Non-Indexed Columns**: Join vendor and order contacts on the type of contact.

Best: MySQL — The performance difference was considerable, as MySQL seems to be optimized for large cross-product joins.

Worst: ArangoDB, Neo4j — Finding cross-products in graph databases seems not to be efficient even with small datasets.

Comments: Note that this query is the most data volume intensive in the whole tested set. To put it into perspective, the 4k dataset included more than 6 GB of table data to be imported into Cassandra, and since

it became unmanageable for us to store such data, we dropped it from the Cassandra testing even though it was the best performer in 1k testing by a large margin.
Interestingly, the performance of SQLite was not as expected. Seemingly, SQLite is not optimized for heavy joins between non-indexed columns.
To conclude, more experiments are needed with higher timeout thresholds to get a clearer picture of the performance.

3.2 **Indexed Columns**: Join all products with their orders.

Best: Cassandra, MongoDB — Both systems performed well here using purely denormalized schemas. Cassandra seems to handle bigger data better and, comparatively, MongoDB may be better at handling lower data volumes.

Worst: Neo4j, ArangoDB — Both compared, Neo4j is worse in higher data volumes than ArangoDB since it timeouts in 1M24k testing. Both use similar indexes on Node/Edge labels or document fields, the queries are similar and in their simplest form, with notable differences being Neo4j's pure graph data model and ArangoDB's document-centered graph data model.

Comments: It's important to mention that having a denormalized schema for this type of query consumes a lot of disk space, but it's a trade-off for performance. On the other hand, MySQL and SQLite performed reasonably well with much smaller memory footprints.

3.3 **Complex Join 1**: Retrieve all order details.

Best: Cassandra

Worst: SQLite — As previously said, SQLite performs poorly when it comes to having to join multiple tables.

Comments: This query requires the most direct joins out of all queries tested. Special mention goes to MySQL, which performed well in this query, even though it was not the best performer. MySQL's performance was consistent across all join queries (3.1 – 3.4). Compared to Cassandra's columnar model, storing wide tables of this kind may not be ideal in most situations, so choosing between the two would depend on the use case and domain. Also, ArangoDB seems to be consistent in joins as well, but the performance is worse than MySQL's overall.
Note that, since we modified MongoDB's schema to include less information in the `orders` collection (figure 3.5), we needed to perform more `$lookup` queries, which influenced performance considerably. Also, since even in the prior schema (figure 3.4) we used one `$lookup` query, the

performance degraded a bit compared to Cassandra, so this has to be taken into account as well when comparing the two.

3.4 **Complex Join 2**: Retrieve all people having more than 1 friend.

Best: Cassandra, SQLite

Worst: ArangoDB — The comparison between ArangoDB and Neo4j is interesting, as both graph queries are logically the same. The multi-model approach of ArangoDB may be the reason for the performance difference in this case.

Comments: Note that Cassandra doesn't include the full scope of information in the schema (the exact friend IDs), only the count of friends. MongoDB, on the contrary, includes an array – `knowsPeople` with the exact friend IDs and uses summation to get the count and self `$lookup` to get the person details. This difference in expressive power has to be taken into consideration. That's also why we include SQLite as another best performer alongside Cassandra.
Also, note that SQLite seems to be better at handling one-join queries better than MySQL even in higher data volumes.

4. **Unlimited Traversal** (figure 4.4)

4.1 **Neighborhood search**: Find all direct and indirect relationships between people up to a depth of 3.

Best: SQLite, Neo4j — Inconclusive

Worst: ArangoDB, (MySQL) — Inconclusive

Comments: Note that MongoDB should not be considered for comparison as it doesn't include the full result set and should be seen only as a demonstration of MongoDB's neighborhood (graph) search capabilities. Interestingly, SQLite performed quite well for a graph traversal query via the use of recursive common table expressions, but it may become a bottleneck if the joined table count increases to more than 64 (section 2.3.2). Comparing Neo4j and ArangoDB as the only GDBMSs tested, Neo4j performed unexpectedly better in this graph traversal query.

4.2 **Shortest path**: Find the shortest path between two people.

Best: ArangoDB, Neo4j — Both have direct methods/functions for finding the shortest path and are highly optimized for it.

Worst: SQLite, MySQL — The recursive queries are terrible at handling graph traversals, especially at higher depths. Even the 1k dataset was too much for both to handle.

Comments: Note that MongoDB should not be considered here, as it doesn't output one shortest path but only performs breadth-first search till it finds the target person (listing A.2). Also, Cassandra doesn't support graph traversals, so it's not included in this comparison, but it has to be said that it would require a lot of work and data duplication to achieve the same result (listing A.1).

5. **Optional Traversal**: Get a list of all people and their friend count (0 if they have no friends). (figure 4.5)

   Best: Cassandra, MongoDB — Both systems performed well, even with higher data volumes. Again, in a lower data count, Cassandra is comparable to all other systems.

   Worst: MySQL, Neo4j

   Comments: Interestingly, ArangoDB outperforms Neo4j here by some non-negligible margin. ArangoDB may be better suited for querying on possibly non-existent relationships.

   SQLite seems to perform relatively consistently across all volumes.

6. **Union**: Get a list of contacts (email and phone) for both vendors and customers. (figure 4.5)

   Best: Cassandra, SQLite — SQLite performed quite well even when having a normalized schema.

   Worst: Neo4j — Unexpected timeout in 1M24k testing even with multiple reruns.

   Comments: Interestingly, MySQL performance was unexpectedly poor relative to SQLite and overall. The SQLite query optimizer might be better at handling union queries, or lower amount of joins as seen previously. ArangoDB seems to have a very similar performance to MySQL.

   Special mention goes to MongoDB which handled this type of query surprisingly well, even by using an inter-collection join/union operation between `vendors` and `orders`, which could have deteriorated performance.

7. **Intersection**: Find common tags between posts and people. (figure 4.5)

   Best: Cassandra, MongoDB — Again, lower data volume seems to be better for MongoDB (and other systems).

   Worst: Neo4j — Neo4j perform very poorly in $\geq$4k datasets. The use of `apoc.coll.intersection` could be a bottleneck.

Comments: SQLite seems to again outperform MySQL in another set operation utilizing only 2 independent joins for this query. ArangoDB's performance in intersection queries is worse than that of relational databases but better than Neo4j's.

8. **Difference**: Find people who have not made any orders. (figure 4.5)

   Best: Cassandra, MongoDB, SQLite

   Worst: ArangoDB, MySQL

   Comments: There seems to be a decrease in MySQL's performance in the 1M24k dataset, but still the difference is not as significant. More data is needed to make a clear distinction.

   8.1 **MongoDB's `$lookup` Special Mention**: Comparing to MongoDB's Query 8.2 (shown in column 8 of figure 4.1 or 4.2) which doesn't use `$lookup`, we can notice the sheer difference in performance where even the 128k testing timed out. Interestingly, a smaller 4k dataset is still close to 0 seconds.

9. **Sorting** (figure 4.5)

   9.1 **Non-Indexed Columns**: Sort products by brand.

   Best: All (except Cassandra) — All systems perform remarkably well in sorting by non-indexed columns.

   Worst: None

   Comments: It is worth mentioning that since Cassandra doesn't support sorting by non-clustering columns, and thus such a query pattern is impossible to execute in Cassandra.

   9.2 **Indexed Columns**: Sort products by brand.

   Best: Cassandra, All — All perform well, with Cassandra having an almost unnoticeable performance advantage.

   Worst: None

10. **Distinct**: Find unique combinations of product brands and the countries of the vendors selling those products. (figure 4.6)

    Best: Cassandra, (all except ArangoDB)

    Worst: ArangoDB

    Comments: It is hard to say which one performed the worst in higher volumes, as we had to limit the amount of Vendor-Products relationships (section 3.2) resulting in an execution time drop across all systems as seen in the figure 4.6.

11. **MapReduce**: Find the number of orders per customer (only those who have made at least 1 order). (figure 4.6)

    Best: Cassandra

    Worst: Inconclusive

    Comments: This query could be considered as another variant of aggregation and so the measured execution times are almost the same as in 2.1 and 2.2. The difference is only in the utilized join operation, and so we cannot deem Cassandra as the best performer here because of the lack of join operation and higher volume of denormalized data required. Interestingly, MySQL's performance has degraded again in the 1M24k testing just like in Query 8.

    Here it is worth mentioning that we wanted to test the possible MapReduce applications in all systems, so the best performers should be the ones that can be easily scaled horizontally to support such operations, i.e. Cassandra and MongoDB.

4.4.1 Recommendations

Based on the results of the experiments, we can provide some recommendations on how to choose the best DBMS for a given scenario. The recommendations are based on the performance of the DBMSs in the experiments and the static analysis presented in chapter 2.

Selection, Projection, Source (of data) Query category 1 includes the most fundamental query patterns in any database system. All databases perform well in this category. Out of all, Cassandra is the best performer in range queries, whether they are using clustering or non-clustering columns, using the same number of records in an aggregate as in all tested DBMSs and thus winning both in query performance and disk space requirements.

For querying on exact values, it depends on whether we have a prior index set up or not and so the recommended system depends on the schema. Overall, all systems are recommended for this query pattern, but when we don't know the exact schema, or we are not sure how the data will be queried, aggregate-ignorant systems like SQLite, MySQL, Neo4j, or multi-model DB like ArangoDB should be preferred (and that applies to all query patterns as well).

Aggregations Simple one-entity aggregations in Query category 2 are handled well by all systems. All tested aggregations seem to be consistent in execution times across all tested systems so distinguishing the best and worst performers here would not be fair.

Notably, Cassandra supports `GROUP BY` operation the same as other systems with the same amount of data in the used aggregate and still holds near zero execution times even in higher data volumes, for which reason Cassandra could be deemed as the best recommendation for this query pattern. However, all systems are recommended for aggregations, and when the aggregations are combined with other operations, such as joins, other systems may be more suitable.

Possibly special case of aggregation in query 11 could be considered as a MapReduce operation, and so the best performers should be the ones that scale for the use case as already mentioned, i.e. Cassandra and MongoDB.

Joins When the business strategy requires joining and connecting many entities as in Query category 3, the decision of the best DBMS falls between the amount of data stored and the complexity of the queries. For example, query 3.1 places an exponential load on the system, as the record count increases the cross-product count increases exponentially. This amount of data has to be accounted for when importing into e.g. Cassandra and that is why we don't recommend denormalized systems for querying cross-products but rather normalized DBMS system, i.e. MySQL.

That accounts for queries 3.2 and 3.3 as well, where the amount of data stored and joined is the most important decision factor. When disk space is not an issue, Cassandra and MongoDB are the best candidates for query pattern 3.2. On the contrary, when the disk space is limited, SQLite is the best candidate for small aggregate joins, and MySQL for more complex joins (query 3.3). Note that the practice of querying all Order details (joining more than 5 other entities/relationships) at once in Cassandra might be only common in situations requiring a lot of data to be stored and queried at once (dashboard, reporting, backups, administration, etc.) and since it will not be utilized often in the application, it's not the best suitable system for pattern 3.3 in production use cases.

For pattern 3.4, the best candidate could be SQLite as it scales well, outperforms mostly all systems, supports joins and with SQL has bigger expressive power than Cassandra and its CQL (section 2.3.5).

Optional traversal, or "outer join" query 5 don't change the previous recommendations much as they are quite similar to how they process data. Notably, this query provides an interesting opportunity to showcase how one query can be understood and logically modeled differently across systems to speed up performance and utilize less disk space. To put into perspective, in Cassandra, `friendCount` property in `Person` table removes the need for another `Person_Person` table as in e.g. MySQL, but conversely, puts higher pressure on the application layer to count friends and/or handle more complex operations. For that reason, aggregate-ignorant system like SQLite, or multi-model system like ArangoDB may be more suitable for this exact

query pattern.

Unlimited Traversal (graph traversals) The queries in the category 4 are typical patterns for querying vastly interconnected data. Both tested GDBMSs – Neo4j and ArangoDB (graph DB) – are best suited for the use cases. Notably, 4.1's neighborhood search was an interesting example where Neo4j and SQLite might be better utilized compared to ArangoDB, but more research has to be done to confirm SQLite's performance in this type of access pattern.

Set (Union, Intersection, Difference) and Distinct Operations The set operations in queries 6, 7, and 8 are typical cases for querying data in a set-like manner. As in the previous categories, data aggregation volume plays a significant role in the decision process between Cassandra/MongoDB and others.

In union query 6 we should take into account how often we merge two sources (query results) and if it's really necessary to have them in one table. SQLite could be the best if we don't do it often, Cassandra if we do, and MongoDB if we want to do both (though more research has to be done to see how `$unionWith` scales).

Next, in intersection query 7, by comparing Cassandra/MongoDB to other systems, we have a similar situation as in query 5, where we need to process possibly a lot of data in real-time in the application layer to create arrays/sets used for intersection. The query could be modeled differently (maybe without arrays), but with this approach, if we don't want to strain our server with real-time synchronization, one can also resort to SQLite, which scales better than MySQL.

To proceed, finding a difference as in query 8, one can choose any system, but with Cassandra, MongoDB, and SQLite being possible favorites. Again, joins utilized in relational databases, are replaced by an array in Cassandra or one attribute in MongoDB, which also puts another requirement on the server to handle synchronization and denormalization.

Lastly, to find distinct values in the result set as in query 10, we can choose any system, and it would be a good choice. Especially, if inserting a row into Cassandra we don't need to think about any logic in the application layer, since by using `PRIMARY KEY ((brand), country)` any duplicates are automatically upserted (discarded in this case). Even then, SQLite could be the best choice for this query pattern based on the performance in the experiments.

Sorting To decide which system is the best in terms of sorting as in query 9, a decision has to be made during schema definition. If we are okay to set up a clustering column in Cassandra, it is the best system for sorting. If we are not sure how the data will be queried, Cassandra cannot be used if not set up properly, and all other systems are recommended.

Conclusion

With the onset of the 21st century, the **v**ariability, **v**elocity, and **v**olume of data have increased exponentially. This has led to the development of new Big Data organization and management solutions, such as the NoSQL databases. Since that time, the NoSQL database market has grown to include many types of databases and vendors offering various solutions for different use cases. For this reason, DBMS benchmarking is a crucial factor in the decision-making process.

In this thesis, we have concentrated on the comparison of MySQL, SQLite, Neo4j, ArangoDB, Cassandra, and MongoDB databases. We have analyzed the individual capabilities of these databases and later performed experiments to compare their performance. We conclude, that MySQL, SQLite, and ArangoDB have the most expressive query languages, but they are not as performant as Cassandra and MongoDB when it comes to querying large datasets. Neo4j and ArangoDB are the best choices for traversing interconnected data, though ArangoDB is more expressive and capable of handling more data models than Neo4j. ArangoDB seems to outperform Neo4j in many cases, but more research has to be done to find out why. Cassandra and MongoDB are the clear winners in performance and horizontal scalability, but Cassandra outperforms MongoDB in terms of speed, while MongoDB is more expressive and offers more features.

When the data count increases, one has to decide between the possibility of joins and flexibility, or targetted data redundancy and speed. SQLite scales the best for a few joins, while MySQL is the only choice for a high number of joins and offers more complex features. SQLite outperforms MySQL in terms of speed, but more experiments have to be done to confirm this. On the other hand, Cassandra and MongoDB require more disk space to scale and possibly more data processing in the application layer, but the speed is unmatched. Cassandra wins in range queries and simple aggregations but loses when we need to scan the entire cluster for a single record or to sort without index as opposed to, e.g. MongoDB.

When unknown how the data will be structured or queried, one should use an aggregate-ignorant database like MySQL, SQLite, Neo4j or opt for a more versatile multi-model system like ArangoDB. Conversely, aggregate-oriented databases like Cassandra and MongoDB require more planning and aggregate design and then monitoring of how the data is queried to adapt the schema to the queries and future requirements, but the speed benefits may outweigh the additional complexity.

Future work might follow up on the surprising inconsistencies in the results, especially in the case of Neo4j and ArangoDB, SQLite and MySQL, or MongoDB overall. Additionally, experiments could be repeated with more data, a higher timeout threshold, and less complex queries or could be horizontally scaled to see how the databases perform in a distributed environment.

Bibliography

[1] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL https://doi.org/10.1145/362384.362685.

[2] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012. ISBN 0321826620.

[3] Tariq N. Khasawneh, Mahmoud H. AL-Sahlee, and Ali A. Safia. Sql, newsql, and nosql databases: A comparative survey. In *2020 11th International Conference on Information and Communication Systems (ICICS)*, pages 013–021, 2020. doi: 10.1109/ICICS49469.2020.239513. URL https://doi.org/10.1109/ICICS49469.2020.239513.

[4] S. W. Dietrich, M. Brown, E. Cortes-Rello, and S. Wunderlin. A practitioner's introduction to database performance benchmarks and measurements. *Comput. J.*, 35(4):322–331, aug 1992. ISSN 0010-4620. doi: 10.1093/comjnl/35.4.322. URL https://doi.org/10.1093/comjnl/35.4.322.

[5] Toni Taipalus. Database management system performance comparisons: A systematic literature review. *Journal of Systems and Software*, 208:111872, February 2024. ISSN 0164-1212. doi: 10.1016/j.jss.2023.111872. URL http://dx.doi.org/10.1016/j.jss.2023.111872.

[6] ThoughtWorks. NoSQL Databases: An Overview, 2024. URL https://www.thoughtworks.com/insights/blog/nosql-databases-overview. [Online; accessed 26-April-2024].

[7] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. Fair benchmarking considered difficult: Common pitfalls in database performance testing. In *Proceedings of the Workshop on Testing Database Systems*. Association for Computing Machinery, 2018. ISBN 9781450358262. doi: 10.1145/3209950.3209955. URL https://doi.org/10.1145/3209950.3209955.

[8] Transaction Processing Performance Council (TPC). TPC-H Benchmark (Decision Support) Standard Specification, 2022. URL https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf. [Online; accessed 10-April-2024].

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings*

of the 1st ACM Symposium on Cloud Computing, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300360. doi: 10.1145/1807128.1807152. URL https://doi.org/10.1145/1807128.1807152.

[10] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, page 14–22, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319768. doi: 10.1145/2494444.2494447. URL https://doi.org/10.1145/2494444.2494447.

[11] Chao Zhang and Jiaheng Lu. Holistic evaluation in multi-model databases benchmarking. *Distributed and Parallel Databases*, 39(1):1–33, March 2021. ISSN 1573-7578. doi: 10.1007/s10619-019-07279-6. URL https://doi.org/10.1007/s10619-019-07279-6.

[12] Rohmat Gunawan, Alam Rahmatulloh, and Irfan Darmawan. Performance evaluation of query response time in the document stored nosql database. In *2019 16th International Conference on Quality in Research (QIR): International Symposium on Electrical and Computer Engineering*, pages 1–6, 2019. doi: 10.1109/QIR.2019.8898035. URL https://doi.org/10.1109/QIR.2019.8898035.

[13] Cornelia Győrödi, Robert Győrödi, George Pecherle, and Andrada Olah. A comparative study: Mongodb vs. mysql. In *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 1–6, 2015. doi: 10.1109/EMES.2015.7158433. URL https://doi.org/10.1109/EMES.2015.7158433.

[14] Yinfeng Wang, Guiquan Zhong, Lin Kun, Longxiang Wang, Huang Kai, Fuliang Guo, Chengzhe Liu, and Xiaoshe Dong. The Performance Survey of in Memory Database. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 815–820, December 2015. doi: 10.1109/ICPADS.2015.109. URL https://doi.org/10.1109/ICPADS.2015.109.

[15] Soad Almabdy. Comparative analysis of relational and graph databases for social networks. In *2018 1st International Conference on Computer Applications & Information Security (ICCAIS)*, pages 1–4, 2018. doi: 10.1109/CAIS.2018.8441982. URL https://doi.org/10.1109/CAIS.2018.8441982.

[16] Rahmatian Jayanty Sholichah, Mahmud Imrona, and Andry Alamsyah. Performance analysis of neo4j and mysql databases using public policies de-

cision making data. In *2020 7th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, pages 152–157, 2020. doi: 10.1109/ICITACEE50144.2020.9239206. URL https://doi.org/10.1109/ICITACEE50144.2020.9239206.

[17] MySQL Corporation. MySQL 8.0 Reference Manual, 2024. URL https://dev.mysql.com/doc/refman/8.0/en/. [Online; accessed 2-May-2024].

[18] ISO. ISO/IEC 21778:2017, 2017. URL https://www.iso.org/standard/71616.html. [Online; accessed 20-March-2024].

[19] ECMA International. ECMA-404, 2021. URL https://ecma-international.org/publications-and-standards/standards/ecma-404/.

[20] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, dec 1983. ISSN 0360-0300. doi: 10.1145/289.291. URL https://doi.org/10.1145/289.291.

[21] MySQL Corporation. MySQL NDB Cluster CGE, 2024. URL https://www.mysql.com/products/cluster/. [Online; accessed 11-March-2024].

[22] MySQL Corporation. Chapter 19 Replication, 2024. URL https://dev.mysql.com/doc/refman/8.0/en/replication.html. [Online; accessed 11-March-2024].

[23] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. SIGFIDET '74, page 249–264, New York, NY, USA, 1974. Association for Computing Machinery. ISBN 9781450374156. doi: 10.1145/800296.811515. URL https://doi.org/10.1145/800296.811515.

[24] E. F. Codd. A data base sublanguage founded on the relational calculus. In *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '71, page 35–68, New York, NY, USA, 1971. Association for Computing Machinery. ISBN 9781450373005. doi: 10.1145/1734714.1734718. URL https://doi.org/10.1145/1734714.1734718.

[25] ISO. ISO/IEC 9075-1:2023, 2023. URL https://www.iso.org/standard/76583.html. [Online; accessed 11-March-2024].

[26] Brad Kelechava. The SQL Standard - ISO/IEC 9075:2023 (ANSI X3.135) - ANSI Blog. *The ANSI Blog*, October 2018. URL https://blog.ansi.org/sql-standard-iso-iec-9075-2023-ansi-x3-135/. [Online; accessed 11-March-2024].

[27] MySQL Corporation. MySQL :: MySQL 8.0 Reference Manual :: 1.6 MySQL Standards Compliance, 2024. URL https://dev.mysql.com/doc/refman/8.0/en/compatibility.html. [Online; accessed 11-March-2024].

[28] Apache Software Foundation. MapReduce Tutorial, 2024. URL https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html. [Online; accessed 11-March-2024].

[29] SQLite Contributors. SQLite Documentation, 2024. URL https://www.sqlite.org/docs.html. [Online; accessed 2-May-2024].

[30] Neo4j Inc. Neo4j Documentation, 2024. URL https://neo4j.com/docs/. [Online; accessed 2-May-2024].

[31] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), feb 2008. ISSN 0360-0300. doi: 10.1145/1322432.1322433. URL https://doi.org/10.1145/1322432.1322433.

[32] ArangoDB GmbH. ArangoDB 3.11 Docs, 2024. URL https://docs.arangodb.com/3.11/. [Online; accessed 2-May-2024].

[33] Jiaheng Lu and Irena Holubová. Multi-model databases: A new journey to handle the variety of data. *ACM Comput. Surv.*, 52(3), jun 2019. ISSN 0360-0300. doi: 10.1145/3323214. URL https://doi.org/10.1145/3323214.

[34] Jiaheng Lu and Irena Holubová. Multi-model data management: What's new and what's next? In *International Conference on Extending Database Technology*, 2017. URL https://doi.org/10.5441/002%2Fedbt.2017.80.

[35] Qingsong Guo, Chao Zhang, Shuxun Zhang, and Jiaheng Lu. Multi-model query languages: taming the variety of big data. *Distrib. Parallel Databases*, 42(1):31–71, may 2023. ISSN 0926-8782. doi: 10.1007/s10619-023-07433-1. URL https://doi.org/10.1007/s10619-023-07433-1.

[36] Weitao Zhang, Yinlong Xu, Yongkun Li, and Dinglong Li. Improving Write Performance of LSMT-Based Key-Value Store. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 553–560, 2016. doi: 10.1109/ICPADS.2016.0079. URL https://doi.org/10.1109/ICPADS.2016.0079.

[37] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. JSON Schema: A Media Type for Describing JSON Documents. Internet-Draft draft-bhutton-json-schema-01, Internet Engineering Task Force, June 2022. URL https://datatracker.ietf.org/doc/draft-bhutton-json-schema/01/. [Online; accessed 30-April-2024].

[38] Mozilla Corporation. JavaScript – MDN, 2024. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript. [Online; accessed 3-April-2024].

[39] ArangoDB GmbH. Cassandra 4.1 Documentation, 2024. URL https://cassandra.apache.org/doc/4.1/. [Online; accessed 2-May-2024].

[40] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010. ISSN 0163-5980. doi: 10.1145/1773912.1773922. URL https://doi.org/10.1145/1773912.1773922.

[41] Pavel Koupil, Daniel Crha, and Irena Holubová. A Universal Approach for Simplified Redundancy-Aware Cross-Model Querying. October 2023. doi: 10.2139/ssrn.4596127. URL https://papers.ssrn.com/abstract=4596127.

[42] ArangoDB GmbH. Arangodb docs - diffing two documents in AQL, 2024. URL https://docs.arangodb.com/stable/aql/examples-and-query-patterns/diffing-two-documents/. [Online; accessed 3-March-2024].

[43] Artem Chebotko, Andrey Kashlev, and Shiyong Lu. A big data modeling methodology for apache cassandra. pages 238–245, 06 2015. doi: 10.1109/BigDataCongress.2015.41. URL https://doi.org/10.1109/BigDataCongress.2015.41.

[44] Microsoft Corporation. Typescript, 2024. URL https://www.typescriptlang.org/. [Online; accessed 3-April-2024].

[45] Node.js, 2024. URL https://nodejs.org/en. [Online; accessed 4-April-2024].

[46] NPM Inc. NPM — Node Package Manager, 2024. URL https://www.npmjs.com/. [Online; accessed 13-April-2024].

[47] Faker.js Contributors. Faker.js, April 2024. URL https://github.com/faker-js/faker. [Online; accessed 4-April-2024].

[48] Scramjet sp. z o.o. Scramjet framework v4, January 2024. URL https://github.com/scramjetorg/framework-v4. [Online; accessed 4-April-2024].

[49] Neo4j Inc. Neo4j ETL Tool, April 2024. URL https://github.com/neo4j-contrib/neo4j-etl. [Online; accessed 10-April-2024].

[50] Neo4j Inc. Neo4j-admin database import tool, 2024. URL https://neo4j.com/docs/operations-manual/5/tools/neo4j-admin/neo4j-admin-import/. [Online; accessed 10-April-2024].

[51] MongoDB Inc. MongoDB Relational Migrator, 2024. URL https://www.mongodb.com/docs/relational-migrator/. [Online; accessed 10-April-2024].

[52] Datastax Inc. Datastax bulk loader (dsbulk), March 2024. URL https://github.com/datastax/dsbulk. [Online; accessed 10-April-2024].

[53] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, jan 2009. ISSN 0001-0782. doi: 10.1145/1435417.1435432. URL https://doi.org/10.1145/1435417.1435432.

List of Figures

3.1 Relational ER diagram of an e-commerce platform. 34
3.2 Graph ER diagram of an e-commerce platform. 35
3.3 (Cassandra-specific) columnar UML Diagram of an e-commerce platform. 37
3.4 (MongoDB-specific) document ER Diagram of an e-commerce platform. 38
3.5 Modified MongoDB schema due to reasons mentioned in listing A.2. 39
3.6 Relational UML diagram of the MySQL database used in the ETL process. 40
3.7 ETL process flowchart. 44

4.1 Table visualization of the query execution times for record volumes 1k, 4k and 128k. 55
4.2 Table visualization of the query execution times for record volumes 256k, 512k and 1M24k. 56
4.3 Logarithmically scaled visualization of the average execution time per record volume for queries 1.1 — 2.2. 57
4.4 Logarithmically scaled visualization of the average execution time per record volume for queries 3-1 — 4-2 58
4.5 Logarithmically scaled visualization of the average execution time per record volume for queries 5 — 9-2 59
4.6 Logarithmically scaled visualization of the average execution time per record volume for queries 10 and 11 60

List of Tables

2.1 DBMS primary selection criteria 11
2.2 Comparison of AQL and Cypher terminology. 23
2.3 Comparison of individual features of Database Management Systems (DBMS) 31
2.4 Data Definition Language (DDL) features of individual Database Management Systems 31
2.5 Data Modeling Language (DML) features of individual Database Management Systems 32

3.1 Dataset sizes for SQLite and MySQL. 49
3.2 Dataset sizes for Cassandra. 49
3.3 Dataset sizes for Neo4j and ArangoDB. 50
3.4 Dataset sizes for MongoDB. 51
3.5 Approximate database sizes on disk measured in MBs. 51
3.6 Approximate import times for individual load or processing stages measured in seconds. 51

List of Listings

3.1 Omitted part of code from the Vendor-Products generator. 42
3.2 Old and ineffective version of `neo4j/neo4j-etl-tool/import.sh` setup. 46

A.1 Special mentions for Cassandra's omitted queries followed by their respective schema definitions. 91
A.2 Special mentions for MongoDB query design decisions. 92
A.3 An example of a TypeScript generator function for generating pseudo-random Vendor and Product entities using the Faker.js library. . . . 94
A.4 A Docker Compose file that defines the DBMS services for the project. 95

List of Abbreviations

ACID Atomicity, Consistency, Isolation, Durability. 7, 12, 15, 17, 20, 27, 81

AI Artificial Intelligence. 16

API Application Programming Interface. 21, 29

AQL ArangoDB Query Language. 21, 22, 31

BASE Basically Available, Soft state, Eventually consistent. 24, 27

CLI Command Line Interface. 45

CPU Central Processing Unit. 15, 20, 52

CQL Cassandra Query Language. 25, 26, 46, 67

CRUD Create, Read, Update, Delete. 13, 16, 18, 21, 25, 29

CSV Comma Separated Values. 41

DB Database. 14, 15, 38, 66, 68, 81

DBMS Database Management System. 7–14, 16, 17, 19, 20, 23, 27, 30, 33, 42, 47, 52–54, 60, 66, 67, 69

DDL Data Definition Language. 7, 11, 13, 16, 18, 21, 25, 29, 30

DML Data Manipulation Language. 7, 11, 13, 16, 18, 21, 26, 29, 30, 53

DNF Did Not Finish. 54

ER Entity-Relationship. 33

ETL Extract, Transform, Load. 8, 33, 41, 42, 46–48, 52

FT Fault Tolerance. 17, 23, 26

GDBMS Graph Database Management System. 16, 63, 68

GUI Graphical User Interface. 29, 45

HA High Availability. 17, 23, 26

I/O Input/Output. 14, 15

IoT Internet of Things. 7, 16

JSON Javascript Object Notation. 12, 19, 21, 26–29

LPG Labeled Property Graph. 16, 19, 21

ML Machine Learning. 16

MMDBMS Multi-Model Database Management System. 19

MQL MongoDB Query Language. 28, 29

NPM Node Package Manager. 41

OS Operating System. 7, 15, 52

R/W Read/Write. 9

RAM Random Access Memory. 15, 41, 52

RDBMS Relational Database Management System. 7, 12–14, 23, 25, 27, 29

SaaS Software as a Service. 27

SQL Structured Query Language. 10, 13, 15, 16, 25, 26, 41, 47, 53, 67

TTL Time To Live. 23

UML Unified Modeling Language. 33, 39

Glossary

aggregate A data unit or a collection of related objects that are treated as a single unit. Provides information on which bits of data will be manipulated or accessed together [2, 6]. 7, 11, 27, 28, 30, 66, 67, 69, 81

aggregate-ignorant Aggregate-ignorant systems, such as relational, graph, and array DBs, do not use aggregates, but rather store data in a normalized form, using strict schema, prioritizing data consistency and ACID properties. 7, 13, 16, 18, 66, 67, 69

aggregate-oriented Aggregate-oriented systems, such as key-value, document and column-family DBs, use aggregates to store data in a denormalized form, using a schema-less approach, prioritizing performance, data availability and scalability. 7, 21, 25, 28, 69

Big Data Large volumes of data that cannot be processed by traditional relational DBs. Characterized by the 3Vs: volume (amount of data), velocity (time at which data is generated), and variety (different data models). 7, 45, 69

eventual consistency In distributed computing, eventual consistency is a model that helps achieve high availability by implicitly guaranteeing that, if no new updates are recorded for a certain data item, eventually, all accesses to the particular data item will return the most recent updated value [53]. 20, 24

query caching Query caching is a feature that saves the results of a query in memory, so the next time the query is invoked, the results are retrieved from the memory instead of executing the query again. 53

A. Attachments

A.1 Project Directory

The current state of the entire project source code and additional data is attached to this thesis in the form of a ZIP archive. For future updates of the project, please consult the GitHub repository at https://github.com/corovcam/Query-Languages-Analysis-Thesis (attached ZIP project is at commit with hash 1fda966).

In this section, we also include a slightly modified project's README text that describes the structure of the project, scripts, configuration, prerequisites, instructions, and other relevant information.

A.1.1 Repository Structure

The top-level repository structure contains folders for all databases used in the thesis, structured as follows:

- `docker-compose.yml` - Docker Compose file for setting up all services, volumes, environment variables, and networks
- `common/` - common files and scripts for all databases
  - `docs/` - extra documentation files (e.g. schema diagrams)
  - `thesis-dummy-data-generator/` - TypeScript NPM project for generating pseudo-random relational data using Faker.js[1] and Scramjet[2] framework
  - `count_table_rows.sql` - for MySQL and SQLite
  - `filter_results.ipynb` - Jupyter Notebook for filtering and processing experiment results
  - `results.csv` - a combined CSV file containing all results from experiments with a header: `db,record_volume,query,iteration,time_in_seconds`
  - `db_sizes.csv` - a CSV file containing database sizes in MB for each database system and record volume
  - `filtered_results.csv` - post-processed CSV file
- `mysql/` - MySQL database files and scripts
  - `data/` - configuration files mounted to the MySQL container
  - `dumps/` - MySQL SQL and/or CSV dumps copied from generated `common/thesis-dummy-data-generator/data_<entity_count>*` folder
  - `exports/` - contains (denormalized) exported data from MySQL database (later imported into Cassandra)
  - `queries/` - SQL queries for MySQL

[1] https://fakerjs.dev/

[2] https://docs.scramjet.org/category/framework

    - `testing/` - individual files with individual queries used for testing
    - **`query.sql`** - complete list of queries and their descriptions (check comments for more details)
    - **`schema.sql`** - complete schema of the MySQL database (check comments for more details)
- `sqlite/` - SQLite database files and scripts (is similar to MySQL structure, so only differences are mentioned)
  - `data/` - database file (`ecommerce.db`) mounted to the SQLite container
  - ***`sqlite3-analyzer`***[3] - binary for analyzing SQLite database and generating statistics about tables called with `sqlite3-analyzer data/ecommerce.db` outside the container, inside `sqlite/` folder
- `neo4j/` - Neo4j database files and scripts
  - `arangodb-json-transform/` - NPM package for transforming exported JSON data from Neo4j to ArangoDB JSON format
  - `dumps/` - Neo4j database dumps (entire database files in compressed binary format)
  - `exports/` - exported JSON documents from Neo4j (later transformed to ArangoDB JSON format)
    - `edges/` - each file inside this folder contains JSON Lines of edges (relationships) generated by `export_to_json.sh`
    - `nodes/` - each file inside this folder contains JSON Lines of nodes (vertices) generated by `export_to_json.sh`
  - `neo4j-etl-tool/` - *Neo4j ETL tool*[4] files for importing data from CSV files (tested with version 1.6.0)
    - `neo4-etl-cli-1.6.0/` - release files downloaded from above GitHub repository
      - `bin/neo4j-etl` - binary for running ETL tool
  - `queries/` - Cypher queries for Neo4j
    - **`query.cypher`** - complete list of queries and their descriptions (check comments for more details)
- `arangodb/` - ArangoDB database files and scripts
  - `dumps/` - ArangoDB database dumps (copied from exported `neo4j/exports` folder)
  - `queries/` - AQL queries for ArangoDB
    - **`query.js`** - definition of AQL queries and their descriptions (check comments for more details)
    - **`query_testing.js`** - entire testing script with error handling for running AQL queries and generating logs

[3] https://www.sqlite.org/sqlanalyze.html
[4] https://neo4j.com/labs/etl-tool/

- `cassandra/` - Cassandra database files and scripts
  - `data/` - configuration file `cassandra.yaml` mounted to the Cassandra container
  - `dsbulk/` -*DataStax Bulk Loader (dsbulk)*[5] JAR file for loading and counting data in Cassandra (tested with version 1.11.0)
  - `dumps/` - Cassandra CQL dumps or CSV files copied from exported `mysql/exports` folder
  - `queries/` - CQL queries for Cassandra
    - **`query.cql`** - complete list of queries and their descriptions (check comments for more details)
    - **`schema.cql`** - complete schema of the Cassandra database (check comments for more details)
- `mongodb/` - MongoDB database files and scripts
  - `dumps/` - MongoDB database dumps (generated with `dump.sh`)
  - `queries/` - MongoDB queries for MongoDB
    - **`query.js`** - complete list of queries and their descriptions (check comments for more details)
    - **`query_testing.js`** - entire testing script with error handling similar to ArangoDB
  - `relational-migrator/` - installation and configuration files for *MongoDB Relational Migrator*
    - `mongodb-relational-migrator_1.5.0_amd64.deb` - installation file for Debian based systems (NOTE: **GIT LFS** is required to download this file!)
    - `ecommerce-mapping*.relmig` - these are configuration files used to setup *MongoDB Relational Migrator*[6] tool (tested with version 1.5.0)

Folders and files common to most of the database directories:

- `logs/` - all logs generated from scripts
  - `queries/` - logs from query testing invoked by `run_queries.sh`
- `queries/` - queries for each database system
  - `testing/` - individual files with individual queries used for testing
- `stats/` - final statistics generated from queries, imports, and exports (later used for analysis)
- `dumps` - database dumps or exported data from other databases (used for importing data)

[5] https://github.com/datastax/dsbulk
[6] https://www.mongodb.com/docs/relational-migrator/

A.1.2 Scripts

Some scripts are common for all databases, some are specific to a particular DBMS. Some like `init.sh` and `run_queries.sh` require manual variable editing to select required data directories (see NOTES inside scripts before execution!). If not explicitly specified otherwise (see script comments), scripts must be run inside respective Docker containers, i.e. with `docker exec -it query-languages-analysis-thesis-<service_name>-1 bash` or `docker exec -it query-languages-analysis-thesis-<service_name>-1 sh`.

Common scripts:

- `/run_queries.sh` - convenience script for running all queries for selected databases (in parallel)
- `/init.sh` - convenience script for inializing all databases (creating schemas, importing data, etc.) for selected databases (in parallel)
- `/<dbms>`
  - `run_queries.sh`, `/init.sh` - individual scripts for running queries or initializing (see comments for more details)

Notable scripts and flows:

- `common/thesis-dummy-data-generator/run.sh` - run with `./run.sh <entity_count>` to generate CSV files with `<entity_count>` number as reference count for top-level entities
- `mysql`
  1. Initialization scripts:
     - `csv_init.sh` - initialize with CSV files as source of data
     - `init.sh` - initialize with SQL dump as source of data (older method used with small datasets - see Legacy generator in A.1.5)
  2. `cassandra_export_to_csv.sh` - transform MySQL tables to denormalized CSV files for importing into Cassandra
- `neo4j`
  1. `neo4j-etl-tool/`
     - `import.sh` - automatically run *Neo4j ETL tool CLI* and subsequently import data using `neo4j-admin database import full` command
  2. `export_to_json.sh` - export Neo4j vertices and relationships to JSON Lines format
  3. `arangodb_transform_json.sh` - transform exported JSON Lines to ArangoDB JSON format
- `cassandra`
  1. Initialization scripts:
     - `dsbulk_init.sh` - initialize Cassandra with CSV files as source of data

- `init.sh` - initialize with CQL dump as source of data (older method used with small datasets - see Legacy generator in A.1.5) - **NOTE:** Very slow for large datasets.

2. `count_table_rows.sh` - count rows in each table in Cassandra using `dsbulk count`
3. `table_stats.sh` - generate statistics for each table in Cassandra using `nodetool tablestats`

A.1.3 Prerequisites

Tools and software required for running the experiments:

- Docker (tested with v25.0.3)
- Docker Compose (tested with v2.24.5)
- Node.js (v20.8+)
- NPM
- Java (tested with OpenJDK 11.0.22)

Tools that need to be (downloaded and) installed manually: (for reproducibility, binaries and installation files are already included in the respective folders)

- Neo4j ETL Tool (v1.6.0) - download link: https://github.com/neo4j-contrib/neo4j-etl/releases/tag/1.6.0
- DataStax Bulk Loader (dsbulk) (v1.11.0) - download link: https://github.com/datastax/dsbulk/releases/tag/1.11.0
- MongoDB Relational Migrator (v1.5.0) - download link: https://migrator-installer-repository.s3.ap-southeast-2.amazonaws.com/index.html#1.5.0/

Tools and software required for filtering the results:

- Python 3
- Pandas, NumPy
- Jupyter Notebook

A.1.4 Installation, Configuration, and Initialization

1. Clone the repository:

```
git clone
↪  https://github.com/corovcam/Query-Languages-Analysis-Thesis.git
cd Query-Languages-Analysis-Thesis
```

2. If not already done, download and install the required tools and software and place them in the respective folders.

- MongoDB Relational Migrator's installation file[7] at `mongodb/relational-migrator/mongodb-relational-migrator_1.5.0_amd64.deb` should be installed on the host machine using any Debian package manager (e.g. `sudo apt install ./mongodb-relational-migrator_1.5.0_amd64.deb`).

3. Run the generator script to generate dummy data for MySQL and SQLite databases:

```
cd common/thesis-dummy-data-generator
./run.sh <entity_count>
```

   where `<entity_count>` is the number of entities to generate (e.g. 1000, 4000, 128000, etc.)

4. Copy the generated data to MySQL and SQLite directories:

```
cp -r data_<entity_count>_<timestamp> ../mysql/dumps/data_<entity_count>
cp -r data_<entity_count>_<timestamp>
↪  ../sqlite/dumps/data_<entity_count>
```

5. Run the following command in the root project directory to build and start the Docker container/s:

```
docker-compose up -d --build <service_name>
```

6. For most databases (starting with SQLite and MySQL), check `init.sh` script for initializing the database, change required variable parameters, and then run the script inside the respective Docker container:

```
docker exec -it query-languages-analysis-thesis-<service_name>-1 bash
./init.sh
```

   or if bash is not available:

```
docker exec -it query-languages-analysis-thesis-<service_name>-1 sh
./init.sh
```

   - **NOTE:** For MySQL and SQLite, you can use `csv_init.sh` script to initialize the database with CSV files as source of data, which comes default from the generator (see Legacy Generator in A.1.5).
   - **NOTE:** For Cassandra, you can use `dsbulk_init.sh` script to initialize the database with CSV files as source of data.

[7] https://www.mongodb.com/docs/relational-migrator/installation/install-on-a-local-machine/install-ubuntu/

7. For Cassandra, you must generate the CSV files from MySQL data using the `cassandra_export_to_csv.sh` script:
    1. Export the MySQL query outputs to CSV files: (as denormalized tables for Cassandra import)

    ```
    docker exec -it query-languages-analysis-thesis-mysql-1 bash
    ./cassandra_export_to_csv.sh
    ```

    - **NOTE:** The script will generate CSV files in the `exports/` folder, where each CSV represents one table in the Cassandra database.

    2. Copy the exported CSV files to the Cassandra container:

    ```
    cp -r exports/ ../cassandra/dumps/data_<entity_count>
    ```

    3. Initialize the Cassandra database using the `dsbulk_init.sh` script (inside the Cassandra container):

    ```
    docker exec -it query-languages-analysis-thesis-cassandra-1 bash
    ./dsbulk_init.sh
    ```

    - **NOTE:** Change the required variable parameters in the script before running it!

- **NOTE:** You can also use the `init.sh` script to initialize the database with CQL dump as source of data (which is generated by the legacy generator script - see Legacy generator in A.1.5).

8. For MongoDB, you must proceed according to the official documentation[8].
    1. After installation, run the binary:

    ```
    cd /opt/mongodb-relational-migrator/bin
    ./mongodb-relational-migrator
    ```

    - **NOTE:** The tool will start a GUI interface accessible at http://localhost:8278/

    2. You must configure the tool using the provided configuration files, i.e. `mongodb/relational-migrator/ecommerce-mapping*.relmig`.
        - **NOTE:** `ecommerce-mapping.relmig` was used for **less than 256k** record volume experiments and `ecommerce-mapping-modified-orders-types-persons.relmig` for **256k+** record volume experiments.
    3. Import the config file by following offical guide[9].
    4. Run the *Sync Job* by following official guide[10].

[8] https://www.mongodb.com/docs/relational-migrator/installation/install-on-a-local-machine/install-ubuntu/

[9] https://www.mongodb.com/docs/relational-migrator/projects/import-project/

[10] https://www.mongodb.com/docs/relational-migrator/jobs/creating-jobs/

- **NOTE:** Both, the MySQL and MongoDB containers must be running before running the sync job.
- **NOTE:** Use the default settings for the sync job. Use `root` as username and password for MySQL and leave defaults for MongoDB.
- **NOTE:** `Drop destination collections before migration` must be checked to avoid conflicts with existing data!

5. Check the import stats and logs for any errors or warnings.

9. For Neo4j, you can use `neo4j-etl-tool/import.sh` script to automatically run the ETL tool and import data into Neo4j.
   1. Run the script inside the Neo4j container:

```
docker exec -it query-languages-analysis-thesis-neo4j-1 bash
cd neo4j-etl-tool
./import.sh
```

   - **NOTE:** MySQL and Neo4j containers must be running before running the script. (conversely, neo4j database instance must be stopped before running neo4j-admin commands - use `neo4j stop` inside neo4j container).
   - **NOTE:** After running the script, you must manually start the Neo4j database instance using `neo4j start` inside the Neo4j container or by restarting the container to proceed.

10. For ArangoDB, you must do the following:
    1. Export the Neo4j data to JSON format using `export_to_json.sh` script (inside the Neo4j container).

```
docker exec -it query-languages-analysis-thesis-neo4j-1 bash
./export_to_json.sh
```

    - **NOTE:** The script will generate two directories: `nodes/` and `edges/` in the `exports` folder containing JSON Lines files for each node and edge type.

    2. Transform the exported JSON data to ArangoDB JSON format using `arangodb_transform_json.sh` (outside the Neo4j container).

```
cd neo4j
./arangodb_transform_json.sh
```

    - **NOTE:** The script will generate the `exports/transformed_<timestamp>/` directory containing JSON Lines files for each node and edge type in ArangoDB format.

    3. Copy the transformed data to the ArangoDB container:

```
cp -r exports/transformed_<timestamp>
↪  ../arangodb/dumps/data_<entity_count>
```

4. Initialize the ArangoDB database using the `init.sh` script (inside the ArangoDB container).

```
docker exec -it query-languages-analysis-thesis-arangodb-1 sh
./init.sh
```

   - **NOTE:** Change the required variable parameters in the script before running it!

A.1.5 Legacy Generator

The legacy generator script is used to generate SQL dumps for MySQL and SQLite databases and CQL dumps for Cassandra database. It was used for generating data for experiments with record volumes up to 256k (included). The script is located in the `common/thesis-dummy-data-generator` folder and is named `data-generator-old.js`. To run the script, use the following command:

```
cd common/thesis-dummy-data-generator
./run-old.sh <entity_count> <sql_dump_dir> <cql_dump_dir>
```

where `<entity_count>` is the number of entities to generate (e.g. 1000, 4000, 128000, 256000), `<sql_dump_dir>` and `<cql_dump_dir>` are the directories where the generated SQL and CQL dumps will be saved.

A.1.6 Query Testing

1. Run the following command in the root project directory to start the Docker container/s:

```
docker-compose up -d <service_name>
```

2. Either run the `run_queries.sh <service_name>` in the root project directory to run all queries for the selected database systems or run the individual scripts for each database system inside the respective Docker container:

```
docker exec -it query-languages-analysis-thesis-<service_name>-1 bash
./run_queries.sh
```

   or if bash is not available:

```
docker exec -it query-languages-analysis-thesis-<service_name>-1 sh
./run_queries.sh
```

   - **NOTE:** Change the required variable parameters in the script before running it!

A.2 Source Codes

Listing A.1: Special mentions for Cassandra's omitted queries followed by their respective schema definitions.

```
-- Query 3.1: Return all Orders and Vendors sharing the same Contact typeId
-- This one was intentionally left out for 4k+ experiments because it takes a lot of disk space
-- (more than 6 GB of disk space during 4k experiments) and it is impractical for ETL purposes
-- (also it is not very useful in real-world scenarios)

SELECT * FROM Vendor_Contacts_By_Order_Contact;

CREATE TABLE Vendor_Contacts_By_Order_Contact (
    typeId             BIGINT,
    orderId            BIGINT,
    orderContactValue  TEXT,
    vendorId           BIGINT,
    vendorContactValue TEXT,
    PRIMARY KEY ((typeId, orderId, orderContactValue), vendorId, vendorContactValue)
);

-- Query 4.1.: Find all direct and indirect relationships between people (up to depth of 3)
-- Using only CQL constructs, its highly inneficient (exponential complexity) to model a
↪  neighbourhood
-- search in Cassandra. The application layer would have to handle all possible paths recursively
-- (up to depth of 3) for each person

SELECT * FROM Person_Relationships;

CREATE TABLE Person_Relationships (
    sourcePersonId  BIGINT,
    relatedPersonId BIGINT,
    depth           INT,
    PRIMARY KEY ((sourcePersonId), depth, relatedPersonId)
);

-- Query 4.2. (Shortest path): Find the shortest path between two people
-- Only by explicitly inserting each and every shortest path between all tuples of (person1id,
↪  person2id)
-- in a specific table, e.g. Shortest_Paths_By_Person
-- We find it highly ineffective to use Cassandra for calculating shortest paths this way
-- (due to reasons mentioned above)

-- 9.1 Non-Indexed Columns: Sort products by brand.
-- In Cassandra, ORDER BY (ordering) is only allowed on the clustering columns of the PRIMARY KEY
-- therefore, the following query is not allowed.

SELECT * FROM Product ORDER BY brand;

CREATE TABLE Product (
    productId BIGINT PRIMARY KEY,
    asin      TEXT,
    title     TEXT,
    price     DOUBLE,
    brand     TEXT,
    imageUrl  TEXT
);
```

Listing A.2: Special mentions for MongoDB query design decisions.

```
// Query 3.1 Non-Indexed Attributes - Join vendorContacts and orderContacts on the type of contact (the same document)

// 3.1 hit the !BSONObj size not supported! error with `BSONObj size: 16806194 (0x1007132) is invalid. Size must be between 0 and 16793600(16MB)` with 256k volume import
// Solution could be not to embed both arrays in the same document, but to split them into separate documents and use $lookup to join them in the query and that is why we have two queries for 3.1 below

// < 256k volume experiments
db.types.aggregate([
  {$unwind:"$orderContacts"},
  {$unwind:"$vendorContacts"},
  {$project:
    {orderContact:{orderId:"$orderContacts.orderId",value:"$orderContacts.value"},
     vendorContact:{vendorId:"$vendorContacts.vendorId",value:"$vendorContacts.value"}}}
]);

// This one also hit the `MongoServerError: Too much memory for single array` which is 16MB as well, more in Dynamic Analysis section
// >= 256k volume experiments
db.types.aggregate([
  {$lookup:{from:"orders",localField:"_id",foreignField:"contacts.typeId",as:"orderContacts"}},
  {$lookup:{from:"vendors",localField:"_id",foreignField:"contacts.typeId",as:"vendorContacts"}},
  {$unwind:"$orderContacts"},
  {$unwind:"$vendorContacts"},
  {$project:
    {orderContact:{orderId:"$orderContacts.orderId",value:"$orderContacts.value"},
     vendorContact:{vendorId:"$vendorContacts.vendorId",value:"$vendorContacts.value"}}}
]);

// Query 3.3 - Complex query with "lookup" to retrieve order details

// Cannot embed "vendor" in "orders.containsProducts" because MongoDB Relational Migrator fails to migrate it (probably due to circular reference) - so the following query is not possible:
// db.orders.find();

// Need to join "vendors" and "orders" on "containsProducts.productId" and "manufacturesProducts.productId" respectively
// < 256k volume experiments
db.orders.aggregate([
  {$unwind:"$containsProducts"},
  {$lookup:
    {from:"vendors",localField:"containsProducts.productId",foreignField:"manufacturesProducts.productId",as:"containsProducts.vendors"}},
  {$unset:["containsProducts.vendors.manufacturesProducts","containsProducts.vendors.contacts"]}
]);

/*
Enormous waiting time for MongoDB Relational Migrator to migrate orders.containsProducts array
  - Waiting time: almost 74 hours with 4 retries and various errors:
    - `java.io.EOFException: Can not read response from server. Expected to read 4 bytes, read 0 bytes before connection was unexpectedly lost.`
    - `java.net.SocketTimeoutException: Read timed out`
  - First run took 2 weeks, and the entire testing server froze (due to memory leak and unrestricted page swapping)
  - Problem turned out to be in MySQL halting the connection unexpectedly
  - Solution: remove orders.containsProducts array
*/
// >= 256k volume experiments
db.orders.aggregate([
  {$lookup:{from:"products",localField:"_id",foreignField:"products.inOrders.orderId",
            as:"containsProducts"}},
  {$unwind:"$containsProducts"},
  {$lookup:{from:"vendors",localField:"containsProducts.productId",
            foreignField:"manufacturesProducts.productId",as:"containsProducts.vendors"}},
```

```
  {$unset:["containsProducts.vendors.manufacturesProducts","containsProducts.vendors.contacts"]}
]);

// 4.1 Find all direct and indirect relationships between people up to 4 (0, 1, 2, 3) levels deep

// This query is the best possible approximation of neighborhood search in MongoDB
// It does not reflect the same results in higher volume datasets as in all other DBMSs and should not be
// taken into consideration when comparing with other systems. Take it only as a demonstration of MongoDB's
// graph traversal capabilities.
db.persons.aggregate([{
  $graphLookup:
    {from:"persons",startWith:"$knowsPeople",connectFromField:"knowsPeople",connectToField:"_id",
     as:"relationships",maxDepth:3,depthField:"depth"}},
  {$unwind:"$relationships"},
  {$project:{_id:0,sourcePersonId:"$_id",relatedPersonId:"$relationships._id"}}
]);

// Query 4.2 Find the shortest path between two persons using $graphLookup

// This one stops traversing in BFS until it finds the target person (not necessarily the shortest path)
// The target person in not included in the result
// The result set is not the same as in other tested DBMSs, but the path is included in the result
// It basically performs BFS until it finds the target person, stops and prints each visited node
// The backtracking is not possible in mongoDB, but can be done in application layer
db.persons.aggregate([
  {$match:{_id:1}},
  {$graphLookup:
    {from:"persons",startWith:"$_id",connectFromField:"knowsPeople",connectToField:"_id",
     as:"relationships",depthField:"depth",restrictSearchWithMatch:{_id:{$ne:10}}}},
  {$unwind:"$relationships"},
  {$project:{_id:0,relationships:1}},
  {$replaceRoot:{newRoot:{$mergeObjects:["$relationships","$$ROOT"]}}},
  {$addFields:{knowsPeople:"$relationships.knowsPeople"}},
  {$unset:["relationships"]},
  {$sort:{depth:1}}
]);

// Query 8 Difference - Find people who have not made any orders

// 8.1 Using Lookup (not recommended for large datasets)
// All people
db.persons.aggregate([
  {$lookup:{from:"orders",localField:"_id",foreignField:"customer.person.personId",as:"orders"}},
  {$match:{orders:{$eq:[]}}},
  {$project:{firstName:1,lastName:1}}
]);

// 8.2 Without Lookup
// Match only people with no customer attribute (and thus no orders)
db.persons.find({customer:{$exists:!1}},{firstName:1,lastName:1});

// Query 11 MapReduce - Find the number of orders per customer (only those who have made at least 1 order)

// 11.1. - Using the deprecated mapReduce() method:
db.orders.mapReduce("function() { emit(this.customer.customerId, 1); }", "function(key, values) { return Array.sum(values); }",
                    {out:{inline:1}});

// 11.2. - Using the simpler (this case at least), more efficient and recommended aggregation pipeline:
db.orders.aggregate([{$group:{_id:"$customer.customerId",orderCount:{$sum:1}}}]);
```

Listing A.3: An example of a TypeScript generator function for generating pseudo-random Vendor and Product entities using the Faker.js library.

```typescript
export function* generateVendorsProducts(
  vendorCount = 100,
  productCount = 1000,
  industryTypes: IndustryType[],
  contactTypes: ContactType[]
): Generator<Vendor> {
  logger.info(`Generating data for ${vendorCount} vendors and ${productCount} products`);
  let productsAssigned = 0;
  for (let i = 0; i < vendorCount; i++) {
      const vendorId = i + 1;
      const vendorName = faker.company.name().replace(/'/g, "''");
      const vendorCountry = faker.location.country().replace(/'/g, "''");

      const vendor: Vendor = { vendorId, name: vendorName, country: vendorCountry,
                               products: [], industries: [], contacts: [] };

      // Assign Products to Vendor
      let productsPerVendor = faker.number.int({ max: MAX_VENDOR_PRODUCTS });
      const productsAssignable = productCount - productsAssigned;
      if (productsPerVendor > productsAssignable) {
          productsPerVendor = productsAssignable;
      }

      for (const product of generateProductsForVendor(
        vendor, productsPerVendor, productsAssigned
      )) {
          vendor.products.push(product);
      }

      // Update productsAssigned count
      productsAssigned += productsPerVendor;

      // Assign Industries to Vendor
      faker.helpers.arrayElements(industryTypes, { min: 1, max: 3 })
        .forEach(type => vendor.industries.push(type));

      // Assign Contacts to Vendor
      const chosenContactTypes = faker.helpers.arrayElements(contactTypes);
      chosenContactTypes.forEach(type => {
          const chosenContactValue = chooseContactValue(type.value);
          const contact = { typeId: type.typeId, value: chosenContactValue,
                            type: { value: type.value } };
          vendor.contacts.push(contact);
      });

      if (i % logger.batchSizeToLog === 0) {
          logger.info(`Generated ${i + 1} vendors and ${productsAssigned} products`);
      }

      yield vendor;
  }
  logger.info(`Generated data for ${vendorCount} vendors and ${productsAssigned} products`);
}
```

Listing A.4: A Docker Compose file that defines the DBMS services for the project.

```yaml
version: "3.8"
services:
  sqlite:
    image: keinos/sqlite3:3.42.0
    volumes:
      - ./sqlite:/sqlite
    restart: always
    working_dir: /sqlite
    stdin_open: true
    tty: true
  mysql:
    image: mysql:8.1.0
    ports:
      - 3306:3306
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: ecommerce
      MYSQL_USER: test
      MYSQL_PASSWORD: test
    volumes:
      - mysql_data:/var/lib/mysql
      - ./mysql:/mysql
      - ./mysql/data/conf.d:/etc/mysql/conf.d
    restart: always
    working_dir: /mysql
  neo4j:
    image: neo4j:5.12.0
    ports:
      - 7474:7474
      - 7687:7687
    environment:
      NEO4J_AUTH: none
      NEO4J_PLUGINS: '["apoc"]'
      NEO4J_apoc_export_file_enabled: true
      NEO4J_apoc_import_file_enabled: true
      NEO4J_apoc_import_file_use__neo4j__config: false
      NEO4J_dbms_security_procedures_unrestricted: apoc.*
      NEO4J_server_db_query__cache__size: 0 # Disable query cache to not cache queries -
      ↪ DEPRECATED
      NEO4j_server_memory_query__cache_per__db__cache__num__entries: 0 # Disable query cache to
      ↪ not cache queries
      # https://neo4j.com/developer/kb/understanding-transaction-and-lock-timeouts/
      NEO4J_db_transaction_timeout: 5m # Set query timeout to 5 minutes
      NEO4J_db_lock_acquisition_timeout: 5m # Set query timeout to 5 minutes
    volumes:
      - neo4j_data:/data
      - ./neo4j:/neo4j
    # Enable for Neo4j ETL Tool MySQL link
    # links:
    #   - mysql
    restart: always
    working_dir: /neo4j
```

```yaml
  arangodb:
    image: arangodb:3.11.3
    ports:
      - 8529:8529
    environment:
      ARANGO_NO_AUTH: 1
      ARANGODB_DOCKER_TTY: true
    volumes:
      - arangodb_data:/var/lib/arangodb3
      - arangodb_apps_data:/var/lib/arangodb3-apps
      - ./arangodb:/arangodb
      - ./arangodb/data/arangod.conf:/etc/arangodb3/arangod.conf
    restart: always
    working_dir: /arangodb
  cassandra:
    image: cassandra:4.1.3
    ports:
      - 9042:9042
    environment:
      CASSANDRA_CLUSTER_NAME: ecommerce
    volumes:
      - cassandra_data:/var/lib/cassandra
      - ./cassandra:/cassandra
      - ./cassandra/data/cassandra.yaml:/etc/cassandra/cassandra.yaml
    restart: always
    working_dir: /cassandra
  mongodb:
    image: mongo:7.0.2
    ports:
      - 27017:27017
    environment:
      MONGO_INITDB_DATABASE: ecommerce
    volumes:
      - mongodb_data:/data/db
      - ./mongodb:/mongodb
    restart: always
    working_dir: /mongodb

volumes:
  mysql_data:
  neo4j_data:
  arangodb_data:
  arangodb_apps_data:
  cassandra_data:
  mongodb_data:
```