FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Ondřej Zálešák

Experimental Analysis of Querying 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 would like to thank my supervisor, Ing. Pavel Koupil, Ph.D., for his supervision of this thesis, his helpful explanation of the technicalities of database systems, and his helpful advice during the process of writing this work.

Title: Experimental Analysis of Querying in Modern Database Systems

Author: Ondřej Zálešák

Department: Department of Software Engineering

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

Abstract: In today's landscape, with a multitude of available database systems, it is often hard to choose which one would fit our needs best. In this thesis we focus on choosing a performant database system from a choice between PostgreSQL, Virtuoso, OrientDB, ScyllaDB, Couchbase, and RavenDB. We compare the static properties and features of said database systems, and we include a brief discussion on data extraction and transformation, for which we developed a helper library for Python. We then assess dataset import times into each database system. To determine the most efficient database, we measure query performance across multiple dataset sizes, and finally we offer a recommendation based on the results, and discuss further possible considerations.

Keywords: database systems comparison, data integration, query expressiveness, query performance

Název práce: Experimentální analýza dotazování v moderních databázových systémech

Autor: Ondřej Zálešák

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Ing. Pavel Koupil, Ph.D., Katedra softwarového inženýrství

Abstrakt: V současné době s vysokým množstvím dostupných databázových systému je často obtížné vybrat mezi nimi ten nejlepší pro naše potřeby. V této práci se zaměříme na výběr výkonného databázového systému mezi PostgreSQL, Virtuoso, OrientDB, ScyllaDB, Couchbase a RavenDB, u kterých porovnáváme statické vlastnosti a dostupné funkce a přidáváme diskuzi o extrakci a transformaci dat, pro kterou vytváříme pomocnou knihovnu v Pythonu. Dále porovnáváme časy importů dat do databázových systému a pro stanovení nejefektivnější databáze měříme výkonnost dotazů pro několik různých velikostí datových sad a nakonec nabízíme doporučení na základě výsledků a diskutujeme další možné úvahy.

Klíčová slova: porovnání databázových dystémů, integrace dat, vyjadřovací síla dotazů, výkon dotazů

Contents

Introduction **7**

1 Static analysis of databases **9**
1.1 PostgreSQL . . . 11
1.2 Virtuoso . . . 12
1.3 OrientDB . . . 13
1.4 ScyllaDB . . . 15
1.5 Couchbase . . . 16
1.6 RavenDB . . . 17
1.7 Summary comparisons . . . 18

2 Data integration **23**
2.1 Extract, Transform, Load . . . 23
2.2 ETL using Python . . . 23
2.2.1 Functional requirements . . . 23
2.2.2 Non-functional requirements . . . 24
2.2.3 Programmer's documentation . . . 24
2.2.4 User's documentation . . . 24
2.2.5 Database drivers . . . 32

3 Experiments **35**
3.1 Queries . . . 35
3.1.1 Query categories . . . 35
3.2 Experiments environment . . . 37
3.2.1 Server details . . . 37
3.2.2 Docker . . . 37
3.2.3 PostgreSQL . . . 37
3.2.4 Virtuoso . . . 38
3.2.5 OrientDB . . . 39
3.2.6 ScyllaDB . . . 40
3.2.7 Couchbase . . . 40
3.2.8 RavenDB . . . 41
3.3 Importing data into the databases from local files . . . 41
3.3.1 PostgreSQL . . . 41
3.3.2 Virtuoso . . . 42
3.3.3 OrientDB . . . 42
3.3.4 ScyllaDB . . . 43
3.3.5 Couchbase . . . 43
3.3.6 RavenDB . . . 44
3.4 Dataset . . . 44
3.5 Results . . . 50

4 Related work **62**

Conclusion **63**

Bibliography **65**

List of Figures **68**

List of Tables **69**

List of Abbreviations **70**

A Attachments **71**

A.1 etl 71
A.2 imports 71
A.3 results_data 71
A.4 PostgreSQL queries 71
A.5 Virtuoso queries 72
A.6 OrientDB queries 73
A.7 ScylllaDB queries 73
A.8 Couchbase queries 74
A.9 RavenDB queries and indexes 74

Introduction

In the current data-driven landscape, there is a multitude of available database systems, each offering their own unique features and benefits, as well as their cons. Therefore we need to be able to choose the correct database system for our requirements. Often the default choice was to use relational database systems such as PostgreSQL [1] due to its strong reputation and robust performance, however new database systems based on *"Not only SQL"* (NoSQL) approaches have emerged and were used successfully in variety of tasks with various requirements, such as the need for highly scalable system.

These new approaches required both the users and database system vendors to rethink their database systems, as there were pros and cons to them. The users would need to consider which features they want and which features they can lack, however the main task that users require from their database systems is to perform queries on their data and get the desired results within acceptable time limits, thus the query performance is critical.

The choice of which database system to use is dependant on many factors and in general, it is not an easy one. Therefore we will try to make a comparison between various factors:

- Static features of database systems and their analysis.

  We need to know what data model can we work with in a database system, how do we query our data, is the system scalable, is the system consistent, does it offer replication or sharding, are our desired queries even supported.

- Data integration.

  Before we can use our database system, we need to prepare our data into correct formats that can be used to load into the database system.

- Database drivers.

  What database drivers exist for our database system, can we use it in our client application that is written in some programming language, or do we need to make adapters, or switch languages, do these drivers even work and do they support all the features we want?

- Query expressiveness.

  Is it possible to create a query that gives us a desired result directly without the need for in-application processing, or are the operations we require not supported.

- Query performance.

  A query that takes too long may be useless, therefore we require certain performance characteristics from each query type, which will differ in the time each query takes.

[1] https://www.postgresql.org/

Outline This thesis is then structured as follows: In this chapter we give an introduction to various features we will compare between the database systems. In Chapter 1 we discuss the static features of database systems, and then compare them among a chosen selection of database systems. In Chapter 2 we discuss data integration, that is, extraction of data, transformation, and loading into database systems, we create a helper library for the extraction and transformation part and we discuss available database drivers for our choice of systems. In Chapter 3 we detail query types with examples, describe the environment used to set up our database systems, introduce the data loading mechanisms and measure the time taken to import our data, and finally perform our queries and measure their performance. In Chapter 4 we review existing related work. Finally, we conclude.

1 Static analysis of databases

Before choosing a database, we must first confirm whether it is able to fit our needs, as different databases are often created with different requirements in mind. Some databases are easier to use with certain types of data, others are able to scale up to much higher amounts of data, therefore we must compare various databases and their properties and compose a static analysis to find our desired database for our needs.

On a static analysis level we compare database systems in several criteria: (1) what is the *data model* used in the database, e.g., does it model data into tables or into documents, (2) how do we talk to the database, that is what is the *query language* used to manipulate and retrieve data, (3) what is the *consistency* level of the queries, (4) how do the databases *scale*, is data *partitioned*, *sharded*, or *replicated*, and (5) what *entity types* are used in the database, what is the support of data aggregates, is the data structure rigid or loose, and how do we reference other related data. All of these questions require a deep dive into the marketing material, documentation, and community forums as each database vendor has their own language and methods that they use to communicate the answers.

Data Model. Database systems can support one or multiple data models [1], on a high level they are usually sorted into *relational databases* [2] and *NoSQL databases* [3, 4], the former stores data into structured tables with rows and columns, often with a rigid schema that defines and limits the structure, while the latter usually has no rigid schema and structure of data can vary between different data points, i.e., one document or row of data.

- Relational database consists of tables with columns and rows.
- Object-oriented database consists of data represented as objects as used in object-oriented programming.
- Object-relational database is a hybrid of relational and object databases, where objects are supported inside tables.
- Document store database model stores data directly as *semi-structured* documents, e.g., JavaScript Object Notation (JSON) or eXtensible Markup Language (XML).
- *Resource Description Framework* (RDF) store consists of subject-predicate-object triples which can be composed into more complex expressions.
- Graph database are similar to relational database stores with the added key concept of relationships (edges), which allows related data to be linked directly together.
- Wide-column store (also known as columnar stores) is similar to document database system, where the columns are not required to be present for each of the row.
- Key-value store models data as key and value, where the key is some identifier and value not further specified and can be of any type.

Query Language. Query languages are created to interact with the database such as inserting, updating, deleting, and retrieving data, therefore they play the main role of being the interface to work with data in a database system. Relational databases usually use the widespread language *Structured Query Language* (SQL) [5], whilst NoSQL databases each have their own query languages, often similar or based on SQL [6].

Consistency. Databases offer various levels of data and index consistency, that is how reliable transactions are, whether they support ACID [7] (*Atomicity, Consistency, Isolation, Durability*), or are BASE [8] (*Basically Available, Soft state, Eventually consistent*). Users need to consider their requirements and choose a database that best fits their needs, although many database systems nowadays are able vary their consistency levels between transactions (e.g., ScyllaDB, Couchbase, RavenDB).

Scalability. Depending on the usage and the amount of data, users will need to consider how databases scale as the user applications grow. Databases can scale *vertically* with more system resources and with faster components, usually CPU cores, memory, faster disk, or they can scale *horizontally* with multiple nodes acting as one, where data may replicated or sharded across multiple smaller and cheaper servers.

Partitioning. Partitioning of data allows related data to be divided into multiple physical structures for faster and parallel query executions and retrievals of data, whilst still acting as one logical unit. This is sometimes called *vertical partitioning*.

Sharding. Whilst vertical partitioning of data is segmenting data on one system, *sharding* is the horizontal scaling equivalent. Data is segmented into multiple *shards* and each shard is then stored on different physical system.

Replication. To backup data, make system resistant to server outages, and allow *load balancing*, data is often replicated among multiple servers, where each server can either standby, or participate in queries.

Schema. Whether a database is *schema-first* or *schema-later* determines the requirement to first define the schema, and then allow importing of data (e.g., in PostgreSQL we first need to create a table which defines our schema, and then we can insert data), that is schema-first, sometimes also called *schema-full*, or in the case of schema-later database, we can insert data immediately, and if the schema is needed, then it is inferred from the data, which is sometimes also called *schema-less* or *schema-hybrid*.

As a consequence in schema-first database systems the structure of data is rigid and required to be the same across all of data (e.g., all rows of data in PostgreSQL table contain all columns), whilst schema-later database systems allow for unstructured data, which can vary in their defined properties.

Aggregates. As opposed to normalized data in relational database systems, databases might support aggregating commonly accessed data into one unit. Some database models inherently support aggregates, such as documents, others offer some limited support, which may not be as efficient as a native support.

Entity types. Database entities are the building blocks used to represent data, they are often tables, documents, vertices and edges in a graph, and others.

References. Some databases allow users to refer to different data inside the database, such as a value inside a different table via a foreign key, or referencing a different document by its ID, or linking vertices together with edges, where each such reference enables more complex manipulation of related data.

Absence of value. Sometimes certain values are optional or make no sense for a given document, and as such these values may be either omitted or replaced with a metavalue that signifies its absence, thus often requiring different way of handling of data.

1.1 PostgreSQL

PostgreSQL[1] is open source object-relational database system that uses SQL language to store and retrieve data. It features transactions with ACID properties, contains multiple data types and can be extended with *user defined types* (UDT), stored functions and procedures and through foreign data wrappers PostgreSQL can connect to other data sources such as other databases or streams with SQL/MED specification.

Data management is done with SQL – Structured Query Language, which was initially developed at IBM and now is in use in many relational databases, and is often adapted to be used in other non relational databases. Whilst SQL language has ISO and ANSI standards, often different database vendors have minor differences, either in the implementation or in its conformance to said standards [9] [10] [11].

Data itself is stored in tables, which are made up of columns that define the type of value, and rows that contain the values. The number and order of columns in a table is fixed and rows order is often implementation defined, however it can be sorted by column values if requested. Relational databases often require that each table must be first defined before it can be used, and PostgreSQL as a schema-full database is no exception to this.

Model: Multi-model, i.e., object-relational and document (namely JSON, XML) store.

Query Language: SQL.

Consistency: ACID.

[1]`https://www.postgresql.org`

Scalability: Supports both *vertical* and *horizontal* scaling. In the former case, one or more CPU cores can serve more clients concurrently and can be used with automatic parallel queries, OS memory caching and tweaking settings to use more memory, disk sequential or random access costs and concurrency can be hinted at. In the latter case, hot standby replicas for read only queries or sharding supported by extensions.

Partitioning: *Table partitioning*, i.e., what is logically one table is split into multiple physical ones based on partition key by range, by listing values, or by hashing.

Replication: Supports multiple replication strategies, namely:

- *Log shipping*, i.e., replicas asynchronously receive *Write-Ahead Logging* (WAL) file segments in blocks to update themselves.
- *Streaming replication*, i.e., WAL records are streamed to standbys.
- *Cascading standby*, i.e., a standby can act both as receiver and sender to other replicas.
- *Asynchronous replication*, i.e., replicas can fall behind master.
- *Synchronous replication*, i.e., primary can be forced to wait until all replicas commit.

Sharding: Supported manually, e.g., by foreign-data wrappers or extensions such as *Citus[12]*.

Schema: Schema-first.

Aggregates: Support of JSON and XML documents as value types.

Entity types: Relation (table), i.e., a set of tuples (rows) with the same attributes (columns).

References: Foreign key.

Absence of value: Metavalue `Null`.

1.2 Virtuoso

Virtuoso Universal Server[2] is a cross-platform server that implements multiple server-side protocols as part of a single product.

Virtuoso offers web services and applications, content management, email services, but most importantly it is also a virtual database engine that provides the ability to search across different databases with a single query and natively

[2]https://virtuoso.openlinksw.com/

implements a traditional object-relational database engine with SQL functionality. Virtuoso also manages to support user defined types and through its native support of XML documents with use of standard XML languages [13], such as XQuery [14], XPath [15], XSLT [16] and SQLX [17], XML documents can be stored and queried. Despite being an object-relational database, Virtuoso also offers support of RDF data which can be accessed and managed with *SPARQL Protocol and RDF Query Language* (SPARQL) queries [18].

All of these features are already offered in the OpenSource edition of Virtuoso and more features such as data replication and clustering can be found in the Enterprise edition.

Model: Multi-model, i.e., object-relational, column-wise table (columns are stored contiguously physically), XML document storage, RDF store.

Query Language: SQL.

Consistency: ACID.

Scalability: Supports both *vertical* and *horizontal* scaling. In the former case, more ram for more caching and less disk IO operations, more CPU cores for query parallelism and more concurrent queries. In the latter case, cluster operation with replicas and sharding (partitioning).

Partitioning: No.

Replication: One-way and bi-directional replication between external databases. Replication on nodes in a cluster.

Sharding: Partitioning on nodes in a cluster.

Schema: Schema-first.

Aggregates: Supports XML documents and RDF data.

Entity types: Tables, XML documents, RDF data.

References: Foreign keys, RDF references.

Absence of value: Null metavalue, RDF blank node.

1.3 OrientDB

OrientDB[3] is a multi-model document-graph database that aims to provide flexible and high-performance operations in one database.

[3]`https://orientdb.org/`

It offers full native graph and document capabilities, such as data stored in so called *Classes* that can be configured as schema-full, schema-free, or with mixed schema, and can be further organized into *clusters*, which optimizes how the data is stored. These clusters are the base units that enable powerful scaling and distributed architecture of OrientDB. In contrast with rows, data in OrientDB is stored in so called *Records*, which can be in the form of `BLOB`(s) for binary data, Vertices and Edges for Graphs, and Documents, often with links other documents. Graphs are created using the Vertex and Edge classes from which user-defined types can inherit. To make querying easier on users who are often familiar with relational databases, OrientDB makes use of SQL like language which is extended to support graph data.

Model: Multi-model, i.e., document and graph database.

Query Language: SQL with graph extensions.

Consistency: ACID.

Scalability: Both *vertical* and *horizontal*. In the former case, classes are clustered into physical partitions, each core gets a cluster. Faster disk means faster reading and writing and data gets cached in available memory. In the latter case, multi-master distributed architecture, each replica can read and write and clusters can be assigned to different replicas.

Partitioning: Classes can be partitioned using class clusters.

Replication: Multi-master replication on multiple nodes.

Sharding: Class clusters can be assigned to servers and each server has its own local cluster.

Schema: Schema-first, i.e., class fields are mandatory, schema-less, i.e., records can have arbitrary fields, and schema-hybrid (or schema-mixed), where some fields are mandatory, others can be arbitrary added.

Aggregates: Documents, that is, fields are handled in a flexible manner. Vertex as a unit of data in a Graph and Edge connecting vertices. Vertices and edges are also documents and so can store arbitrary properties.

Entity types: Documents, vertices and edges in a graph.

References: Links to documents, edges are represented as links on the connected vertices. Documents can also be embedded.

Absence of value: Null metavalue and absence of value.

1.4 ScyllaDB

Scylla[4] is a fast and scalable NoSQL database supporting key-value store and wide-column data, it is also a *drop-in* replacement for Apache Cassandra [5], with higher performance that comes from highly scalable approach of clustering servers per CPU cores via so called *shards*. This performance can be leveraged through Scylla's first-party shard-aware drivers [6] that connect clients directly to the correct shard server, which can also be replicated to offer high availability and fault tolerance.

As a Cassandra replacement, ScyllaDB supports the *Cassandra Query Language* (CQL) [19], which is similar to SQL language, with extra *add-ins* and improved features, such as select and insert statements that interpret JSON documents. Data is stored in *tables*, which define the layout of data, and are grouped into *keyspaces*, with configuration options that apply to all tables inside it, such as the *replication strategy*. Despite its sharded approach to performance and availability, Scylla also offers the clients the possibility to choose consistency levels on a per operation basis.

ScyllaDB is also able to replace Amazon DynamoDB [7], another NoSQL database, although managed and closed source, through its Scylla Alternator open source DynamoDB compatible API, however notable differences exist.

Model: Multi-model, i.e., wide-column store and key-value store.

Query Language: CQL, i.e., SQL-like query language

Consistency: BASE, consistency is tunable for a given query.

Scalability: Both vertical and horizontal scaling. In the former case, ScyllaDB is designed to scale linearly with CPU core count with its shard-per-core approach, uses available memory for caching and recommends at least 2 GB per logical core, the more memory available, the better the performance. Recommended storage/memory ratio is 30:1 per node, the faster the drive, the better performance. In the latter case, entire dataset is sharded across cluster into individual nodes.

Partitioning: No.

Replication: Data is replicated on multiple nodes based on the chosen Replication Factor.

Sharding: Data is sharded using hashes.

Schema: Schema-first.

[4]https://www.scylladb.com/
[5]https://cassandra.apache.org/
[6]https://www.scylladb.com/product/scylla-drivers/
[7]https://aws.amazon.com/dynamodb/

Aggregates: ScyllaDB supports conversions to JSON for insertions and retrieval, but does not natively support them.

Entity types: Column family tables, columns can be added to a family, each row does not need to have all columns.

References No explicit references between tables.

Absence of value Null metavalue or unset column.

1.5 Couchbase

Couchbase[8] is a distributed database that combines strengths of relational databases such as SQL and ACID transactions into a document store of flexible JSON documents. For data manipulation it uses its own SQL++ language [20], which is an SQL-compatible query language extended for use with JSON documents.

Couchbase's offerings include distributed multi-document ACID transactions, auto-sharding, inter-cluster and cross-data center replication and multi-dimensional scaling, in which the basic services such as querying or indexing can be scaled independently.

To speed up write operations, Couchbase allows them to happen in memory cache while asynchronously processing replication, persistence and index management, although it is possible for users to configure consistency level for the each independent operation.

Data in Couchbase is stored as individual *documents* comprising *key* and *value*, where the the value can be JSON formatted to enable rich access capabilities. Such JSON documents are highly flexible and can be of varied schemas and contain nested structures, therefore Couchbase does not enforce any uniformity requirements and the structuring into Bucket-Scope-Collection-Document hierarchy is dependant upon the application, though internally each Bucket is implemented by vBuckets, which are akin to shards in other database systems, and are distributed evenly across the cluster.

Model: Multi-model, i.e., document and key-value store.

Query Language: SQL++ (SQL extended for use with JSON documents).

Consistency: Data transactions have ACID properties, indexes have BASE properties, i.e., queries can tune their consistency level requirements.

Scalability: Each Couchbase service (i.e., Data, Query, Index, Search, Analytics, Eventing, Backup) can be configured per node basis to optimize utilization of hardware resources. In particular, *vertical* and *horizontal* scaling can be combined. In the former case, some services (e.g., Index, Query) benefit from faster or more

[8]https://www.couchbase.com/

CPU cores, more memory for cache, faster disk. In the latter case, data and services are shared across a cluster, data is stored in vBuckets and can have up to three replicas, i.e., in Intra-Cluster replication. Cross Data Center Replication allows replicating across multiple data clusters, that is, can be in both directions.

Partitioning: No.

Replication: Intra-Cluster across nodes and Cross Data Center across clusters.

Sharding: Automatic hash based sharding according to vBuckets.

Schema: Schema-later, i.e., based on JSON documents.

Aggregates: Aggregate-oriented, i.e., data is stored in JSON documents.

Entity types: Documents, indexes.

References: No explicit references between tables.

Absence of value: Null metavalue, absence of value.

1.6 RavenDB

RavenDB[9] is a high performance, distributed, NoSQL document database that stores data as JSON documents, which are the primary use case, and has extensions for binary data, time series and counters. RavenDB can run on a cluster of nodes which gives the user high availability, load balancing, and geo distribution of data, while still using ACID compliant transactions, meaning that operations on a document using its ID to put, modify or delete are always consistent, the indexes are eventually consistent (that is, BASE), and can lag behind document updates, however clients can also choose to wait for replication and index changes confirmations. A single cluster can store multiple databases each of which can span some or all of the nodes in the cluster.

RavenDB can be used as a key-value store for information caching with automatic document expiration. It is encouraged to use independent, isolated and coherent documents, where embedding is usually the default approach, however data belonging to other documents can also be referenced with IDs, making many domain driven design techniques highly useful and recommended.

Model: Multi-model, i.e., document and key-value store.

Query Language: *Raven Query Language* (RQL), i.e., SQL-like query language.

Consistency: Documents are stored and accessed in an ACID manner. Queries are handled with BASE. User can tune the behaviour on a case-by-case basis.

[9]`https://ravendb.net/`

Scalability: Allows both *vertical* and *horizontal* scaling. In the former case, prefer faster cores rather than more cores, RavenDB will use all available memory to cache, and prefers high performance and exclusive storage. In the latter case, replication into external clusters, or internally among nodes in cluster based on Replication Factor, and sharding of data among nodes.

Partitioning: No.

Replication: External, that is, among different database instances, or internal, i.e., among nodes in a cluster (either each shard, or whole database).

Sharding: Data is sharded among nodes in a cluster.

Schema: Schema-later, i.e., RavenDB operates on JSON documents.

Aggregates: Aggregate-oriented, i.e., documents are aggregates.

Entity types: Document, indexes.

References: No explicit references to other documents, but referenced documents can be preloaded.

Absence of value: Null metavalue and absence of value.

1.7 Summary comparisons

Comparisons of the features can be summarized into the following tables that compare static database system features, the *Data Definition Language (DDL)*, and the Data Modeling Language (DML) between the various database systems.

Static database system features The static database features, such as the used model, language, or scaling aspects, can be compared between various database systems, which can help with choosing the best database system for our needs. If we for example need all or nothing transactions that are consistent across all replicas, we might want to choose a database system with ACID properties, which could be useful for financial systems [21], on the other hand, if we were designing a chat application, we would want fast scalable queries with BASE properties [22]. This comparison can be found in the table 1.1.

Comparison of Data Definition Language features Data Definition Language can usually be summed up into few various concepts, for schema-full database systems we create, alter, or drop the schema (e.g., a table, or class), but for schema-less database systems there is no such thing. However what all kinds of database systems needs is the creation of data, its update, and deletion. And since the query languages of compared databases are based on SQL, the commands are therefore similar in wording, and the comparison can be viewed in table 1.2.

Comparison of Data Modeling Language features It is not often that we need to view all of our data, if we did, we could just use plain data files and not bother with any database systems, thus we use the Data Modeling Language features to transform our data into our desired representation. Since SQL is one of the most common query languages, and the compared databases were made based on this, the data modeling is therefore done in a similar way, albeit some database vendors' choices of database features limit what can be done. For example ScyllaDB does not support joins between different tables, which is also mirrored in the fact that its CQL language is unable to select from multiple tables to form joins or sub-queries. Though sometimes different ways, to do the same things are available, however they may not be optimal or wanted, e. g. data denormalization. The full comparison can be found in table 1.3.

Table 1.1 Comparison of individual database features

	PostgreSQL	**Virtuoso**	**OrientDB**	**ScyllaDB**	**Couchbase**	**RavenDB**
Model	object-relational, document	object-relational, column-wise, document, RDF	document-graph	wide-column, key-value	document store, key-value	document store, key-value
Language	SQL	SQL	SQL with extensions for graph concepts	CQL	SQL++	RQL
Consistency	ACID	ACID	ACID	BASE – tunable queries	Data-ACID, BASE queries – tunable	Data-ACID, BASE queries – tunable
Scalability	vertical and horizontal	vertical and horizontal	vertical and horizontal	vertical and horizontal	vertical and horizontal	vertical and horizontal
Partitioning	table partitioning	no	class clusters	no	no	no
Replication	external	external, cluster	cluster	cluster	external, cluster	external, cluster
Sharding	manual or extensions	yes	yes	yes	yes	yes
Schema	full	full	full, free, hybrid	full	free	free
Aggregates	supported	supported	oriented	some support	oriented	oriented
Entity types	tables	tables, RDF data	documents, graph vertices and edges	column family tables	documents	documents
References	foreign keys	foreign keys, RDF references	links to documents, edges, embedded documents	denormalized data modelling	denormalized data modelling	denormalized data modelling
Absence of value	null metavalue	null metavalue, RDF blank node	null metavalue, absence of value	null metavalue, absence of value	null metavalue, absence of value	null metavalue, absence of value

Table 1.2 Comparison of Data Definition Language (DDL) features

	PostgreSQL	Virtuoso	OrientDB	ScyllaDB	Couchbase	RavenDB
CREATE	CREATE TABLE	CREATE TABLE	CREATE CLASS [1]	CREATE TABLE	[2]	[3]
ALTER	ALTER TABLE	ALTER TABLE	CREATE PROPERTY [1]	ALTER TABLE	[2]	[3]
DROP	DROP TABLE	DROP TABLE	TRUNCATE CLASS <class> UNSAFE [1]	DROP TABLE	[2]	[3]
INSERT	INSERT INTO	INSERT INTO	INSERT INTO	INSERT INTO	INSERT INTO	Store [4]
UPDATE	UPDATE	UPDATE	UPDATE	UPDATE	UPDATE	Load [4]
DELETE	DELETE FROM	DELETE FROM	DELETE VERTEX/EDGE	DELETE FROM	DELETE FROM	Delete [4]

[1] Schema-full Enables strict-mode at a class-level and sets all fields as mandatory. Schema-less Enables classes with no properties. Default is non-strict-mode Schema-hybrid Enables classes with some fields
[2] Flexible, Dynamic Schema In the document model a schema is the result of an application's structuring of its documents: schemas are entirely defined and managed by applications. A document's structure consists of its inner arrangement of attribute-value pairs. Documents can be grouped in collections which can be grouped in scopes.
[3] Schema-free documents – web UI or client application SDKs (C#, Java, ...)
[4] Store, Load and update, or Delete the document using client SDKs, and then SaveChanges to send it to the database.

Table 1.3 Comparison of Data Modeling Language (DML) Features

	PostgreSQL	**Virtuoso**	**OrientDB**	**ScyllaDB**	**Couchbase**	**RavenDB**
Projection	SELECT	SELECT	SELECT #1	SELECT	SELECT	select
Source	FROM	FROM	FROM	FROM	FROM	from
Selection (search condition)	WHERE	WHERE	WHERE	WHERE	WHERE	where
Aggregation	GROUP BY	GROUP BY	GROUP BY	GROUP BY	GROUP BY	group by
Aggregation selection	HAVING	HAVING	-	-	HAVING	Multi-Map-Reduce indexes [2]
Join	JOIN	JOIN	Represented by LINKs	-	JOIN	Multi-Map indexes [2]
Graph Traversal	JOIN (multiple tables)	JOIN (multiple tables)	MATCH, TRAVERSE, SELECT	-	JOIN (multiple tables)	Multi-Map-Reduce indexes [2]
Unlimited Traversal	WITH RECURSIVE	-	-	-	-	Multi-Map-Reduce indexes [2]
Optional	OUTER JOIN	OUTER JOIN	-	-	OUTER JOIN	Multi-Map-Reduce indexes [2]
Union	UNION	UNION	unionall()	-	UNION	-
Intersection	INTERSECT	INTERSECTION	intersect()	-	INTERSECT	-
Difference	EXCEPT	EXCEPT	difference()	-	EXCEPT	-
Sorting	ORDER BY	ORDER BY	ORDER BY	ORDER BY [1]	ORDER BY	order by
Skipping	OFFSET	TOP SKIPINTNUM, INTNUM	SKIP	-	OFFSET	limit {skip},{limit}
Limitation	LIMIT / FETCH	TOP SKIPINTNUM, INTNUM	LIMIT	LIMIT	LIMIT	limit {skip},{limit}
Distinct	DISTINCT	DISTINCT	DISTINCT	DISTINCT	DISTINCT	distinct
Aliasing	AS	AS	AS	AS	AS	as
Nesting	(select)	(select)	(select)	-	(select)	-
MapReduce	GROUP BY	GROUP BY	GROUP BY	GROUP BY	GROUP BY	Map-Reduce indexes [2]

[1] ScyllaDB possible orderings are limited by clustering order
[2] Multi-Map-Reduce indexes need to be created manually [23]

2 Data integration

Data usually comes in various formats and contain various parts, some of which we might need to extract, modify or transform before we can then load them into a desired database with its own formats and loading mechanisms. To make this data integration step easier, users can use different libraries to help. We make our own library to help us with the extraction and transformation part.

2.1 Extract, Transform, Load

Extract, Transform, Load (ETL) process is a data integration process that extracts data from one data source, transforms it as specified by the user, and then loads it into a target database.

ETL processing is executed using tools or libraries (e.g. Oracle Data Integrator [1], IBM DataStage [2]). Many ETL vendors offer additional data profiling and analysis capabilities, such as investigating metadata, row sizes, value uniqueness, exploring relationships between entities, or data compliance checks or rules [24]. Common use case for ETL tools include extracting raw data from source locations (such as SQL or NoSQL servers, JSON, CSV, or XML files), transforming them (e.g. filtering, aggregating, de-duplicating, formatting, etc.) and formatting them to match target database and loading them inside [25].

2.2 ETL using Python

Many programming languages offer easily accessible building blocks to create one's own specialized ETL tools. Python[3] is a common environment for such tools, with its ease of use and many importable libraries.

2.2.1 Functional requirements

Such a tool should offer a variety of features such as:

- Reading and writing data in CSV, XML, and JSON format
- Projections of columns by index or name
- Selections of rows by value or regex
- Join multiple data tables into one based on a key
- Add surrogate key
- Substitution – map values onto other values
- Number formatting

[1]`https://www.oracle.com/middleware/technologies/data-integrator.html`
[2]`https://www.ibm.com/products/datastage`
[3]`https://www.python.org/`

- Data normalization
- Split a column into multiple
- Extract from a column into multiple by pattern matching
- Combine multiple columns into one
- Apply a transformation to values in a column
- Apply a transformation to all values
- Concatenation of rows and columns
- Return first N rows (head)
- Return last N rows (tail)
- Get a row count
- Get a column count
- Duplicate rows until a specified row count is reached

2.2.2 Non-functional requirements

This tool should also desire the following properties:

- Performance, i.e., ability to handle larger amounts of data
- Usability, i.e., simple and intuitive usage, ease of use
- Maintainability and Extensibility, i.e., functions should be easy to maintain and extend

2.2.3 Programmer's documentation

etlpy library is a single-file python library that eases the manipulation and transformation of data files, it reads data from various file formats, such as CSV, JSON, and XML. In addition, data can be extracted, transformed and saved into these file formats.

Supplied library functions can be easily modified and the library can be easily extended with more desired functions. The library is dependant on the *pandas* framework, which is a Python package that provides fast, flexible, and expressive data structures to simplify usage with relational data. Although it aims to simplify, the users can often find themselves in pitfalls, either through highly technical naming of functions and parameters, or due to its technical and sometimes difficult to understand documentation. Therefore *etlpy* library aims to further simplify the use into simple functions providing common features for use with relational data.

2.2.4 User's documentation

This part of documentation defines the software requirements, and the installation steps, following with the API documentation.

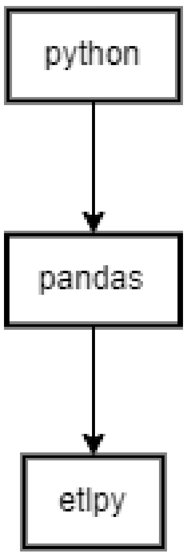

Figure 2.1 Framework dependency graph

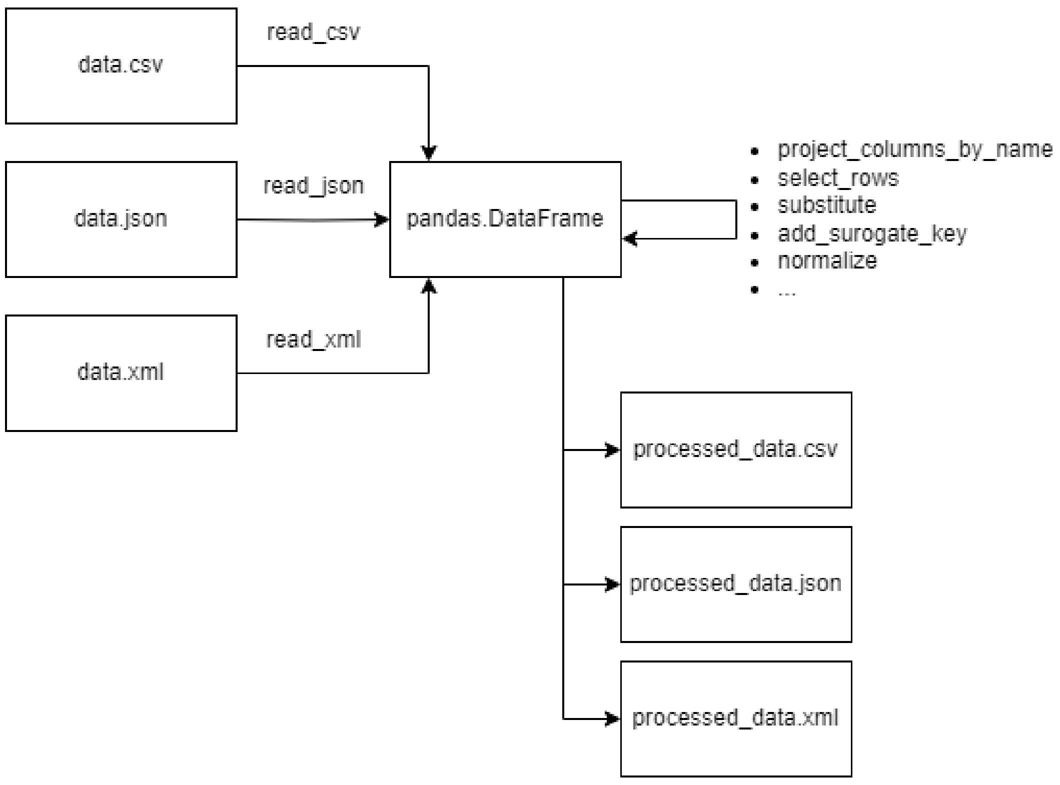

Figure 2.2 Usage flowchart

Software Requirements: Python[4], Pandas[5]

etlpy library depends on the Python programming language and thus requires that Python is downloaded and installed, futher *etlpy* depends on the pandas library, which can be installed by following the instructions in the pandas documentation from the Python Package Index[6] via pip[7] which may come preinstalled with Python or may need to be installed separately.

Installation: As the *etlpy* is a single-file library, installation is as easy as copying the *etlpy.py* file into the desired project folder, and importing the desired (or all) functions:

```
from etlpy import *
```

API Documentation. *etlpy* library provides the following functions to read, manipulate or transform, and save data.

First the user needs to load data from the CSV, XML, or JSON files:

```
def read_csv(file_path, sep=",", nrows=None) -> pd.Dataframe
def read_xml(file_path) -> pd.Dataframe
def read_json(file_path) -> pd.Dataframe
```

```
file_path: str = path to the csv, xml, or json file
sep: str = separator used in csv file, by default "," (comma)
nrows: int = number of rows to read from a csv file, by default
    None (all rows)
```

Resulting data type is of `pd.Dataframe` type, which is supplied into further data manipulation functions or it can also be manipulated with directly using pandas library.

On the other end, there are functions to save the data into files, into CSV, XML, or JSON files:

```
def save_csv(df, file_path, index=False, sep=",") -> None
```

```
df: pd.DataFrame = pandas DataFrame with the data to save
file_path: str = file path of the resulting csv file
index: bool = whether to write row names (indexes), by default
    False
sep: str = separator character, by default "," (comma)
```

```
def save_xml(df, file_path, index=False) -> None
```

[4]https://www.python.org/downloads/
[5]https://pandas.pydata.org/pandas-docs/stable/getting_started/install.html
[6]https://pypi.org/
[7]https://pip.pypa.io/en/stable/

```
df: pd.DataFrame = pandas DataFrame with the data to save
file_path: str = file path of the resulting xml file
index: bool = whether to write row names (indexes), by default
    False
```

```
def save_json(df, file_path, orient="records") -> None
```

```
df: pd.DataFrame = pandas DataFrame with the data to save
file_path: str = file path of the resulting xml file
orient: str = indication of expected JSON format, by default
    "records" which is list-like, other values are split, index,
    columns, values, table, which can be found in pandas
    documentation
```

Dataframe is then saved into the given `file_path`, `index` variable controls whether to include row indices inside the file, and `orient` variable in JSON describes how the data is translated into JSON types, that is, value of records mean that data is saved as a list of rows, where each row is an object of key-value pairs. Pandas also allows other possible JSON formats, which are described in Pandas documentation[26].

After loading the data, user can use following transformation functions:

```
def project_columns_by_name(df, cols) -> pd.DataFrame
```

```
    df: pd.DataFrame = pandas DataFrame with the data
    cols: list[str]: list of columns to project by name
```

```
def project_columns_by_index(df, cols) -> pd.DataFrame
```

```
    df: pd.DataFrame = pandas DataFrame with the data
    cols: list[int]: list of columns to project by index
```

Supply a list of column names or column integer indexes `cols` to keep inside data.

```
def select_rows(df, column_name, value) -> pd.DataFrame
def select_rows_regex_contains(df, column_name, pattern) ->
↪   pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
column_name: str = name of the column to select in
value = value to select rows by
pattern: str = select rows by this regex pattern
```

Execute a selection over data by equality with `value` or by regex matching with `pattern`.

```
def join(left_df, right_df, on, how) -> pd.DataFrame
```

```
left_df: pd.DataFrame = left data table to join
right_df: pd.DataFrame = right data table to join
on: list[str] = list of columns to join on
how: str = type of merge to be performed: left, right, outer,
    inner, cross
```

Perform a join over two DataFrames, supplying a list of column names to perform a merge over with a description of how, e.g., left, right, inner, outer, cross.

```
def add_surogate_key(df, new_column_name, loc = None, inplace =
↪   True) -> pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
new_column_name: str = name of the newly added column with the
    key
loc: int = column index, or at end if equal to None
inplace: bool = add column in place or return a copy instead
```

Add a surrogate sequential key to each row, in column specified by `loc` value.

```
def substitute(df, column_name, mapping, inplace = True) ->
↪   pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
column_name: str = column name in which to perform the
    substitution
mapping: dict = mapping of values, e.g. {"M": "Male", "F":
    "Female"}
inplace: bool = perform inplace or return a copy
```

Perform a substitution with a given `mapping` in the form of dictionary.

```
def format_numbers(df, column_name, inplace = True) ->
↪   pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
column_name: str = column in which to perform the number
    reformatting
inplace: bool = perform inplace or return a copy
```

Perform number formatting, replacing multiple whitespaces with just one, and replacing decimal comma with a decimal point.

```
def normalize(df, columns, new_column_name) -> tuple[pd.DataFrame,
↪   pd.DataFrame]
```

```
df: pd.DataFrame = pandas DataFrame with the data
columns: list[str] = list of column names which are to be
    normalized
new_column:_name: the name of replacement column
```

Performs a normalization of data in supplied `columns`. Distinct tuples of values are mapped onto integers, which are then used in the new column named as `new_column`. The merged and newly created normalized tables are returned as a tuple.

```
def split(df: pd.DataFrame, column, pattern, n=-1, regex=True,
↪   expand=True) -> pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
column: str = which column to split
pattern: str = how to split the column, either by delimiter or
    regex
n: int = limit the number of columns in the result, use -1 for no
    limit
regex: bool = split either by regex or by string delimiter
expand: bool = true to return multiple columns, false to return
    one column of list of values
```

Splits values in a column into multiple columns based on supplied `pattern` and returns a table of the split columns or column with a list of values.

```
def extract(df, column, pattern) -> pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
column: str = column name to extract from
pattern: str = regex pattern to extract values out
```

Extracts values from one column into multiples based on supplied regex `pattern`, returns a DataFrame with a column per capture group.

```
def combine(df: columns, sep=" ") -> pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
columns: list[str] = list of columns to combine together
sep: str = separator inbetween values from columns
```

Combines values from multiple columns into one using given separator.

```
def apply(df, column, func) -> pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
column: str = name of column to apply given function on
func: function that gets applied on the values
```

Apply a function transformation `func` on values inside given column named by the variable `column`.

```
def applyall(df, func) -> pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
func: function that gets applied on the whole dataframe
```

Apply a function transformation `func` on values inside each column.

```
def concat_columns(dfs: list[pd.DataFrame]) -> pd.DataFrame
```

```
df: list[pd.DataFrame] = pandas DataFrames with the data
```

Concatenates columns, that is adds more columns to one DataFrame.

```
def concat_rows(dfs: list[pd.DataFrame]) -> pd.DataFrame
```

```
df: list[pd.DataFrame] = pandas DataFrames with the data
```

Concatenates rows, that is adds more rows to one DataFrame, values in columns available only in one DataFrame are filled with `NaN` values.

```
def head(df, n) -> pd.DataFrame
def tail(df, n) -> pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
n: int = number of values to keep
```

Returns the top `n` rows – `head`, or bottom `n` rows – `tail`.

```
def row_count(df: pd.DataFrame) -> int
def column_count(df: pd.DataFrame) -> int
```

```
df: pd.DataFrame = pandas DataFrame with the data
```

Returns the row count or column count.

```
def duplicate_until_rows(df, n) -> pd.DataFrame
```

```
df: pd.DataFrame = pandas DataFrame with the data
n: int = desired number of rows
```

Duplicates rows inside a DataFrame, until a desired number n of rows is reached.

Usage example The library can be easily used, and its usage is demonstrated on the following example. Let's use an example of hypothetical data about churner clients in a bank, the data includes various information about the client and their account.

Include the library:

```
from etlpy import *
```

First we load our data:

```
data = read_csv("BankChurners.csv")
```

This file contains a lot of data that we do not need so we project the desired columns – we keep the ones we want, and get rid of the others:

```
desired_data = project_columns_by_name(data, ["CLIENTNUM",
 ↪  "Customer_Age", "Gender", "Income_Category", "Card_Category"])
```

Next we select only certain data from the dataset, for example, customers whose age is 50 and their card category is Blue:

```
desired_data = select_rows(desired_data, "Customer_Age", 50)
desired_data = select_rows(desired_data, "Card_Category", "Blue")
```

Lastly to demonstrate how we can join tables, we project different columns and then join the tables:

```
join_data = project_columns_by_name(data, ["CLIENTNUM",
 ↪  "Marital_Status", "Education_Level"])
out = join(desired_data, join_data, on="CLIENTNUM", how="inner")
```

The original data contained gender in the form of M or F, we can substitute this data with Male and Female:

```
out = substitute(out, "Gender", {"M": "Male", "F": "Female"})
out1, out2 = normalize(out, ["Education_Level", "Income_Category"],
 ↪  "Education_and_Income_Category")
out1 = add_surogate_key(out1, "Surrogate_Key")
```

We expect the data or perhaps we inspected the data and saw the limited domain of education level and income category, and we can save some space by normalizing this data:

```
out1, out2 = normalize(out, ["Education_Level", "Income_Category"],
 ↪  "Education_and_Income_Category")
out1 = add_surogate_key(out1, "Row Index")
```

We also added a surrogate key Row Index that indexes the data sequentially from one.

At last we can save our data into desired files:

```
    save_csv(out1, "BankChurners1.csv")
    save_csv(out2, "BankChurners2.csv")

    save_xml(out1, "BankChurners1.xml")
    save_xml(out2, "BankChurners2.xml")

    save_json(out1, "BankChurners1.json", indent=2)
    save_json(out2, "BankChurners2.json", indent=2)
```

2.2.5 Database drivers

Database drivers are used by client applications to communicate with the database, send query requests, upload data, and otherwise interact with the database.

PostgreSQL

PostgreSQL offers a C library called *libpq*[8] that clients can use to connect and interact with PostgreSQL, it is also the underlying engine for several other libraries used in other programming languages, such as the open source python driver *psycopg*[9] used in the experiments. Whilst libpq or psycopg is the driver used to connect and interact with PostgreSQL, there are other libraries simplifying the usage of such drivers, one of them is *SQLAlchemy*[10] that provides a comprehensive set of tools for working with databases in Python.

Importing bulk data into PostgreSQL can be done using the SQL `COPY` instruction, or `COPY` from client-side using the
`copy` command inside *psql*, the PostgreSQL interactive terminal. Bulk importing with the `COPY` command can be done from various formats, usually from csv files, into an already existing table.

Virtuoso

Virtuoso as a universal server provides plenty database drivers, among them are drivers adhering to the *Open Database Connectivity* (ODBC)[11], *Java Database Connectivity* (JDBC)[12], and Microsoft's ADO.Net[13] and OLE-DB[14] data providers. Connecting to Virtuoso database from Python can be achieved using SQLAlchemy and the open source Python ODBC bridge library *pyodbc*[15].

Since virtuoso supports mapping csv files as tables and querying in files without loading them into the database or they can be imported into a table, however the bulk loader SQL scripts may need to be first run manually to create the procedures needed.

[8] https://www.postgresql.org/docs/current/libpq.html
[9] https://www.psycopg.org/
[10] https://www.sqlalchemy.org/
[11] https://docs.openlinksw.com/virtuoso/odbcimplementation/
[12] https://docs.openlinksw.com/virtuoso/virtuosodriverjdbc/
[13] https://docs.openlinksw.com/virtuoso/virtclientrefintro/
[14] https://docs.openlinksw.com/virtuoso/virtoledb/
[15] https://github.com/mkleehammer/pyodbc

OrientDB

As OrientDB is a database management system written in Java[16], the most up to date drivers with the most features are the native Java APIs, the Java Multi-Model API[17] providing unified API for both document and graph related queries, Apache TinkerPop API[18], i.e., an interface for handling graphs, and OrientDB also provides a JDBC driver[19] that can be used to interact with the database using the standard way in Java.

There are also other drivers written in multiple programming languages, such as PHP[20], Python[21], Ruby[22], C[23], and others, but they're community made, and often have not been updated in years, therefore they either lack new features, or are completely incompatible with newer versions of OrientDB.

Importing into OrientDB is a bit more complex than in other databases, since it is also a graph database, there may be a need to further specify relations between data. Therefore OrientDB contains an ETL module that allows to extract, transform and load data as described in a JSON configuration file from CSV and JSON files or other databases using JDBC drivers.

ScyllaDB

ScyllaDB offers first party-drivers in multiple languages such as Python[24], Java[25], and others, all of them are shard-aware and provide additional benefits over third-party drivers used with Apache Cassandra/CQL driver.

ScyllaDB is also DynamoDB compatible with the so called Scylla Alternator, however notable differences exist and new DynamoDB features may take time to implement, therefore Alternator is not configured to be used by default and usually not recommended unless already using applications or libraries written for DynamoDB.

Importing into Scylla from a csv file into a predefined table is straightforward using the `COPY` command using the command-line interface *cqlsh*, the CQL shell.

Couchbase

Couchbase provides several SDKs[26] to access the Couchbase cluster, all of them are available with documentation, API reference and examples, however some features are shipped as separate SDK extension libraries that may be less available in a given programming language. Couchbase SDKs offer both traditional synchronous API as well as scalable asynchronous APIs in programming languages

[16] https://www.java.com/
[17] https://orientdb.com/docs/3.2.x/java/Java-MultiModel-API.html
[18] https://orientdb.com/docs/3.2.x/tinkerpop3/OrientDB-TinkerPop3.html
[19] https://orientdb.com/docs/3.2.x/jdbc-driver/index.html
[20] https://github.com/orientechnologies/PhpOrient
[21] https://github.com/orientechnologies/pyorient
[22] https://github.com/topofocus/active-orient
[23] https://github.com/tglman/orientdb-c
[24] https://python-driver.docs.scylladb.com/stable/
[25] https://java-driver.docs.scylladb.com/stable/
[26] https://www.couchbase.com/developers/sdks/

such as C[27], .NET[28], Python[29], Java[30], and a lot more others, and is also compatible with ODBC and JDBC standards.

Bulk loading data into Couchbase is done using the *cbimport* utility tool, which can import data from csv or json files into specified bucket and scope with specified key expression.

RavenDB

RavenDB offers multiple client drivers, however the main ones are written in C#[31], Java[32], Python[33], and Node.JS[34], and in addition to these there are drivers in few other programming languages and even a REST API is provided.

RavenDB comes with a management studio inside a browser, that is used to define various parameters, or perform maintenance actions. Importing into RavenDB can be done through this studio and can be done from either other RavenDB servers, or RavenDB file dumps, from SQL using available drivers for databases such as PostgreSQL, MySQL Server[35] and others, or from NoSQL databases such as MongoDB[36] or CosmosDB[37].

RavenDB Studio also allows importing from CSV files, albeit with limited configuration, therefore it is often better to use available drivers with *BulkInsert* operations.

[27]https://docs.couchbase.com/c-sdk/current/hello-world/overview.html
[28]https://docs.couchbase.com/dotnet-sdk/current/hello-world/overview.html
[29]https://docs.couchbase.com/python-sdk/current/hello-world/overview.html
[30]https://docs.couchbase.com/java-sdk/current/hello-world/overview.html
[31]https://www.nuget.org/packages/RavenDB.Client/
[32]https://central.sonatype.com/artifact/net.ravendb/ravendb
[33]https://github.com/ravendb/RavenDB-Python-Client
[34]https://www.npmjs.com/package/ravendb
[35]https://www.mysql.com/
[36]https://www.mongodb.com/
[37]https://azure.microsoft.com/en-us/products/cosmos-db

3 Experiments

To compare the performance of the various database systems we need to set up some experiments, that can give us some information, first we look at what kind of queries there are, that is, what kind of results do we want from the systems, then we look into setting up our database systems, preparing the environment and installing them. Afterwards we need to know how to import our data, what file formats do the systems accept, and then we compare the database disk sizes between systems and the data in text format, we also measure how long each database takes when importing the datasets. Finally we take a look at query execution times and summarize the results.

3.1 Queries

How databases are designed and created often has an effect on which queries are supported and how fast querying for data is. Therefore we often want to find the best database for our needs.

3.1.1 Query categories

Our queries come in a few categories.

Projection We want to define which parts of dataset we want returned in the query, i.e. which column in a table, or key-value pair from an object.

Therefore for a person we might want only his last name.

```
SELECT lastName FROM people
```

This query will return last names of all people in our table. We might want to use the symbol * to project all columns.

Selection We want to select only certain people depending on our selection criteria.

So we might want to select only people born on a certain year.

```
SELECT lastName FROM people WHERE birthYear = 1963
```

This query will return the last names of all people born in the year of 1963.

Aggregation Aggregation is used to return a single value calculated from the collection of values, such as the average birth year of our customers, or the total count of customers.

```
SELECT COUNT(*) FROM people
```

This query will return the total count of all people in our database.

```
SELECT AVG(birthYear) FROM people
```

This query will return the average birth year of all people in our database.

Join

```
    SELECT people.id, people.name, homes.ownerid, homes.address
    FROM people INNER JOIN homes
    ON people.id = homes.owner.id
```

This query will fetch records from people and buildings and return a combined results of people and their own homes.

Union

```
    SELECT address FROM shopping_centres
    UNION
    SELECT address FROM department_stores
```

This query will fetch addresses from two different tables and join them into one result set.

Intersection

```
    SELECT address FROM shopping_centres
    INTERSECT
    SELECT address FROM department_stores
```

This query will fetch addresses from two different tables and only keep those that are in both.

Difference

```
    SELECT address FROM shopping_centres
    EXCEPT
    SELECT address FROM department_stores
```

This query will keep only those addresses from the first table, that are not in the second.

Sorting

```
    SELECT * FROM people ORDER BY last_name
```

Return all people ordered in the result set by their last name.

Limiting the total values returned and offsetting

```
    SELECT * FROM people ORDER BY last_name LIMIT 10;
    SELECT * FROM people ORDER BY last_name OFFSET 10;
    SELECT * FROM people ORDER BY last_name LIMIT 10 OFFSET 5;
```

Select those people when ordered by last name are either the first ten (limit), all after the first ten (offset), or a combination of both (limit and offset)

Map-reduce

```
SELECT last_name, COUNT(*) AS last_name_holder_count FROM
↪ people GROUP BY last_name
```

This query will group all people by their last name and count how many people share a last name.

3.2 Experiments environment

This section covers how to replicate the environment used in the experiments, and how can anyone use the databases for their own needs.

3.2.1 Server details

Databases were set up on a virtual server, running the operating system Ubuntu 22.04.4 LTS with Linux version 5.15.0-105-generic. The hardware configuration was 8 cores of Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz and 32 GB of ram, and an SSD with 80 GB of disk space.

3.2.2 Docker

The easiest way to reliably use many software packages and products is through virtualization. One of the common available options is through Docker[1] products, which offer light weight containerization. Software is then packaged into docker containers, which act like a standardized unit, that is portable and ready to use on multiple systems.

Docker itself is supported on all platforms, Windows, macOS, and Linux, where it is available on multiple distributions such as CentOS [2], Debian [3], Fedora [4], Ubuntu [5] and others.

As it is a widely used and supported virtualization platform, it is often installed through the standard procedures, such as *apt* [6] on Debian or through `.deb` package, or *yum* [7] on CentOS or through `.rpm` package, or *dnf* [8] on Fedora, or through installation files on Windows or Mac. We can also mount additional data drives or folders, which will be used to help importing our data sets.

3.2.3 PostgreSQL

To download PostgreSQL version 16.3 docker image based on debian-bookworm simply use:

[1]https://www.docker.com/
[2]https://www.centos.org/
[3]https://www.debian.org/
[4]https://fedoraproject.org/
[5]https://ubuntu.com/
[6]https://wiki.debian.org/AptCLI
[7]http://yum.baseurl.org/
[8]https://rpm-software-management.github.io/

```
docker pull postgres:16.3
```

And to set-up an image:

```
docker run --name some-postgres -e
    POSTGRES_PASSWORD=mysecretpassword -d --mount
    type=bind,source=/data,target=/appdata postgres:16.3
```

Where the name `some-postgres` is the container name, `POSTGRES_PASSWORD` is the required password parameter, and the option `-d` launches the container in detached state, then we mount the host folder `/data` that will be available in the container as `/appdata`, and finally we specify that we want to use the `postgres` image.

Starting and stopping the container can be done with simple commands:

```
docker start some-postgres
docker stop some-postgres
```

And using the the terminal-based front-end to PostgreSQL, *psql*, is easily done with this command:

```
docker exec -it some-postgres psql -U postgres
```

where `-it` specifies interactive execution of the `psql` command by the linux user `postgres`. The database is then ready to use.

To install the PostgreSQL Python3 client *psycopg* and the Python SQL toolkit *SQLAlchemy* we can simply use pip

```
pip install "psycopg[binary]"
pip install sqlalchemy
```

After that we can simply use python to connect to our database and run our queries.

3.2.4 Virtuoso

Using docker we download the opensource edition version 7:

```
docker pull openlink/virtuoso-opensource-7:7
```

Setting up the image is a bit more complicated, because we also need to create an additional mount point where our data will be stored, which simplifies migration.

```
mkdir /my_virtdb

docker run --name my_virtdb --env DBA_PASSWORD=mysecret
   --publish 1111:1111 --publish 8890:8890 --volume
    /my_virtdb:/database --mount
    type=bind,source=/data,target=/appdata
    openlink/virtuoso-opensource-7:7
```

Our data and config files will be stored in our folder `my_virtdb`, we also publish the port `8890` to the host system, where we can connect to the Virtuoso web server and the port `1111` to provide external driver access, although if we connect from inside the container, we might not need this. We also set up the user `dba` password which we can use to manage the database.

If we want to use the Virtuoso interactive SQL interface, *iSQL*, we can connect with this command:

```
docker exec -it my_virtdb isql 1111
```

To connect with our Python clients we need to install the *SQLAlchemy* toolkit and the *pyodbc* module that is used to connect to ODBC databases:

```
pip install pyodbc
pip install sqlalchemy
```

To connect via ODBC we also need to set up ODBC Data Source, we can use `/etc/odbc.ini` file and name our data source as `VOS`, which we will use to connect:

```
[VOS]
Description = Open Virtuoso
Driver      = /opt/virtuoso-opensource/lib/virtodbcu_r.so
Database    = CSV
Address     = localhost:1111
WideAsUTF16 = Yes
```

We also specified database `CSV`, where we will later import our data.

3.2.5 OrientDB

We use docker to install OrientDB version 3.2.30:

```
docker pull orientdb:3.2.30
```

And then we set up our docker container:

```
docker run -d --name orientdb -p 2424:2424 -p 2480:2480 -e
    ORIENTDB_ROOT_PASSWORD=rootpwd --mount
    type=bind,source=/data,target=/appdata orientdb:3.2.30
```

where we configure our root password and we publish ports, where the database listens on, port 2424 for binary connections, and port 2480 for http connections, where we can also reach the OrientDB Studio web server.

We can use the console inside the docker container to connect to the database:

```
docker exect -it orientdb /orientdb/bin/console.sh
```

There are plenty of client drivers in multiple languages, but some of them are community made and may not be updated to latest versions, however the database itself is made in Java and offers both a standard JDBC driver and a specialized interface, we can grab both from either OrientDB directly, or through various Java repositories.

3.2.6 ScyllaDB

We download the Scylla image onto our docker system:

```
docker pull scylladb/scylla:5.4.6
```

And we create our ScyllaDB container:

```
docker run --name some-scylla -d --mount
    type=bind,source=/data,target=/appdata
    scylladb/scylla:5.4.6
```

We can connect using cqlsh inside our container:

```
docker exec -it some-scylla cqlsh
```

Or we can use the Python client to connect, once we install the driver:

```
pip install scylla-driver
```

After that we can use the Scylla driver inside our client application by importing from the `cassandra` namespace.

3.2.7 Couchbase

We pull our docker image from the docker hub:

```
docker pull couchbase:7.6.1
```

And set up our container:

```
docker run -d --name couchbasedb -p 8091-8097:8091-8097 -p
    9123:9123 -p 11207:11207 -p 11210:11210 -p 11280:11280 -p
    18091-18097:18091-18097 --mount
    type=bind,source=/data,target=/appdata couchbase:7.6.1
```

Couchbase uses a variety of ports to communicate between nodes and its services, and the docker image description tells us to publish many of these ports, however the most crucial port is 8091, where we can connect to the Web Console and set up and administrate our cluster.

To install our Python client we again use pip:

```
pip install couchbase
```

And we can now use our Couchbase database.

3.2.8 RavenDB

We pull our RavenDB image using:

```
docker pull ravendb/ravendb:6.0.103-ubuntu.22.04-x64
```

And set up our container with the port 8080 published:

```
docker run --name ravendb -d --mount
    type=bind,source=/data,target=/appdata -p 8080:8080
    ravendb/ravendb:6.0.103-ubuntu.22.04-x64
```

Then we go to the web interface on port 8080 and set up our cluster using the installation wizard.

Finally for the Python client we again install it through pip:

```
pip install ravendb
```

And now our client applications can make use of RavenDB.

3.3 Importing data into the databases from local files

3.3.1 PostgreSQL

Importing data into PostgreSQL table is done using the COPY command from the *psql* client:

```
\copy my_table_name FROM '/appdata/my_data.csv' DELIMITER '|'
    HEADER CSV NULL as '\N'
```

Where we can choose the delimiter and the way we represent null values, our table must also be already created and the data must fit into this table.

3.3.2 Virtuoso

To import data into Virtuoso we can use the CSV File Bulk Loader, which can be pre-loaded in some versions of Virtuoso, or we import the script found in documentation [27] ourselves.

We first register our CSV files using the `csv_register` function which creates an entry into the table `DB.DBA.csv_load_list`, if we want to reload a CSV file, we will have to delete this entry and the data table where it was imported and start again.

By default the CSV file is loaded into the `csv.DBA.` schema of the database, but we can create a `.tb` file to use the schema location and table name inside.

We can also create a `.cfg` file if we need to describe our CSV file:

```
[csv]
csv-delimiter=|
csv-quote="
header=0
offset=1
```

After that we use `csv_register(path, mask)` to register files inside single directory, or `csv_register_all(path, mask)` to recursively register files inside a directory, and then we use `csv_loader_run()` to load our data.

3.3.3 OrientDB

Importing into OrientDB is done with the `oetl.sh` script provided with the database, where we specify a single JSON ETL configuration file which controls the ETL process. Using specified extractor options which handles data extraction from source, e.g. JSON, CSV or others, transformer options to transform data before insertion, and loader options where we specify our database and classes, we can do import whatever data we want.

How to specify the configuration file we can find in the ETL section of OrientDB documentation [28] and the basic structure of the configuration file looks like the following:

```
{
  "config": {
    <name>: <value>
  },
  "begin": [
    { <block-name>: { <configuration> } }
  ],
  "source" : {
    { <source-name>: { <configuration> } }
  },
  "extractor" : {
    { <extractor-name>: { <configuration> } }
  },
  "transformers" : [
    { <transformer-name>: { <configuration> } }
  ],
  "loader" : { <loader-name>: { <configuration> } },
  "end": [
   { <block-name>: { <configuration> } }
  ]
}
```

To start the ETL process we then simply call the oetl script:

```
    oetl.sh config-mydata.json
```

3.3.4 ScyllaDB

Loading CSV data into ScyllaDB is very easy and quite the same as loading data into PostgreSQL, we just need to define our table structure and then load data using the `COPY` command, where we need to specify the columns:

```
COPY my_data (id, first_name, last_name, address) FROM
    '/appdata/my_data.csv' WITH DELIMITER='|' AND HEADER=TRUE AND
    NULL='NULL';
```

Again we can specify additional options such as the delimiter, whether our CSV file contains a header, and how we specify null values.

3.3.5 Couchbase

To import data into Couchbase we use the *cbimport* tool [29] which can extract data from CSV or JSON files, however the handling of null values works only in JSON files, which are then preferred.

```
cbimport json -c couchbase://127.0.0.1 -u Administrator -p
    Password -b my_bucket -d file:////appdata/my_data.json -f
    list -g key::name::%id% --scope-collection-exp
    my_scope.my_data -t 4
```

We can specify plenty of options, mainly our database cluster URL, our user name and password, the bucket and scope collection we import into, the key generator expression, which describes how will our key look like, the data set format, whether our JSON file is a list of values, or just plain lines of values, and lastly the dataset file.

3.3.6 RavenDB

Importing data into RavenDB can be done using the Management Studio, where we can import data from our previous RavenDB dump files or other RavenDB servers, from NoSQL databases (MongoDB and CosmosDB), from SQL databases (Microsoft SQL Server[9], MySQL Server, PostgreSQL, and Oracle Database), and lastly from a CSV file, where we can specify some options, such as the delimiter, although limited to predefined values.

However loading large collections using the CSV File import option inside the Management Studio seems to get stuck and fail, therefore the best way to load data is to use Bulk Insert functions directly from client applications.

3.4 Dataset

The dataset we will use to test our queries will come from the *IMDb*[10] which is a searchable database including millions of movies, TV and entertainment programs[30]. It offers a subset of this data for non-commercial use which can be downloaded directly from their developer portal [11]. The dataset is then transformed using *etlpy* from Chapter 2 into datasets of lenghts 1000, 64000, 256000, and 1024000. Depending on the database, it is then saved as `.csv` file for use with PostgreSQL, Virtuoso, and ScyllaDB, and as `.json` file for use with OrientDB, Couchbase, and RavenDB.

The process is in Figure 3.1. To prepare the names datasets, we read the data from `tsv` - tab separated value format, which can be read with `read_csv` with specified separator, then we get the top `n` values using `head`, where `n` is the desired dataset size, we add in a surrogate key, and then using `apply_all` we change the representation used for null values from `\N` to `NULL` in each table cell, finally, we duplicate rows if we do not have the desired row count already, we save the dataset into `csv` file format, and for `json` we change `NULL` to `None` type, to ensure actual null values inside json file.

Similarly, the Figure 3.2 depicts the process of titles transformation. To get the titles datasets, first get the top `n` values, add a surrogate key, change the null

[9]https://www.microsoft.com/en-us/sql-server/
[10]https://www.imdb.com/
[11]https://developer.imdb.com/non-commercial-datasets/

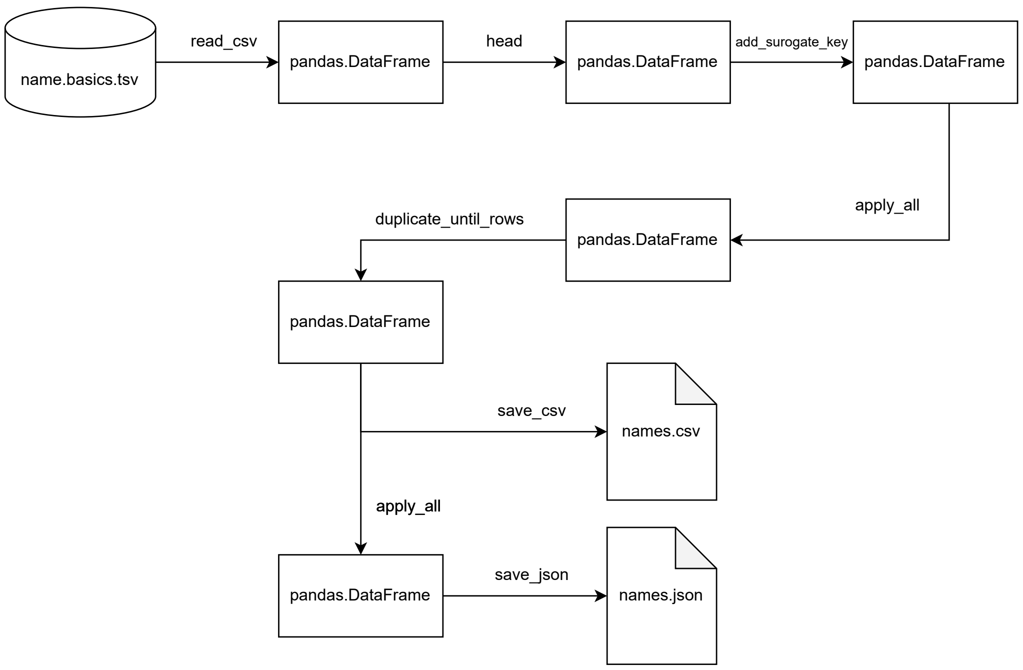

Figure 3.1 Names transformation diagram

representation, and duplicate rows, afterwards we save the datasets in `csv` files and `json` files with actual null values.

In the following table 3.1 the file sizes are shown for both `.csv` and `.json` format so that further comparison about used disk space can be made.

Table 3.1 Raw file size comparison

Dataset size	**names.csv**	**names.json**	**titles.csv**	**titles.json**
1000	104 kB	203 kB	50 kB	122 kB
64000	6 MB	12 MB	4 MB	8 MB
256000	22 MB	47 MB	15 MB	33 MB
1024000	86 MB	185 MB	58 MB	130 MB

PostgreSQL It is very easy to get the table size in PostgreSQL, as it offers the `pg_total_relation_size` function which computes the total disk space used by the specified table, including all indexes and TOAST data, The Oversized-Attribute Storage technique, responsible for certain data types, that can contain very large values directly.

Virtuoso Unfortunately Virtuoso does not offer any functions that would make it easy to calculate the table sizes, however it offers a view that we can select from and find out the total size for each table (namely `DB.DBA.SYS_INDEX_SPACE_STATS`), since the index also contains the row data, therefore the total size is the sum of the size of the primary index and the size of all secondary indexes [31].

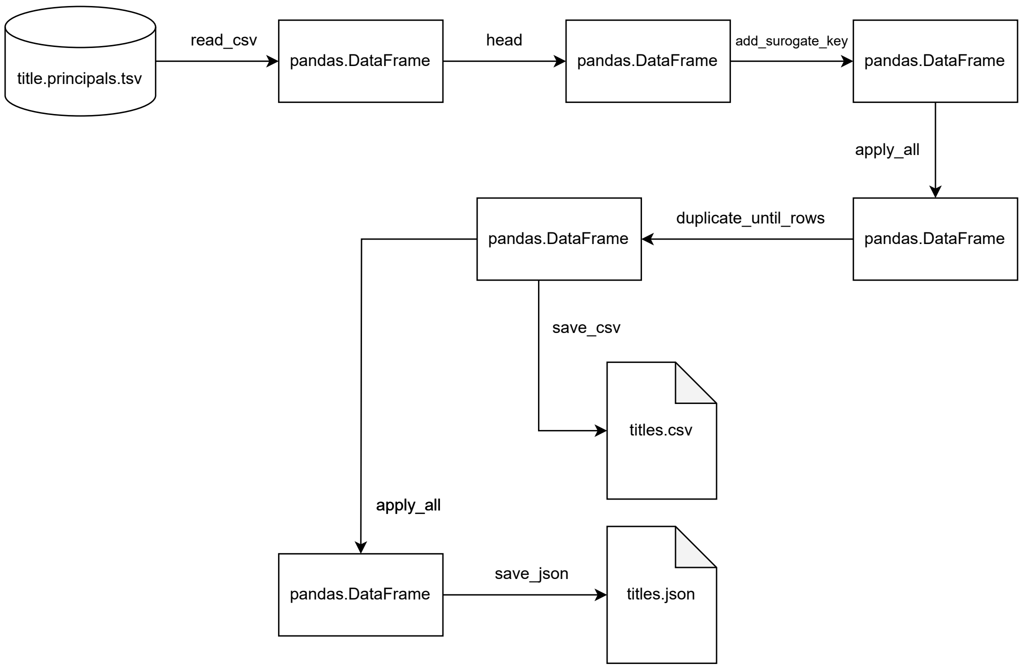

Figure 3.2 Titles transformation diagram

Table 3.2 PostgreSQL table sizes

Dataset size	names	titles
1000	240 kB	160 kB
64000	9 MB	7 MB
256000	34 MB	27 MB
1024000	132 MB	108 MB

OrientDB Even more lacking is OrientDB which offers no way to get the size of our classes or the whole database, although several outdated drivers do hint at a binary protocol command for whole database size, such a command is not present in either of the up to date drivers, and unfortunately the old drivers are not compatible with the current database system version.

However at least the database system file structure was easy enough to navigate and all the files corresponding to a database were in the folder of database name, therefore at first I measured this folder size, i.e., database size, with all the files present, such as the Write-Ahead Log file. Afterwards, I measured the database

Table 3.3 Virtuoso table sizes

Dataset size	names	titles
1000	91 kB	35 kB
64000	5 MB	3 MB
256000	21 MB	11 MB
1024000	76 MB	43 MB

folder without inserted data, and any data that needed to be flushed to disk after creating the database, then I inserted the data, and measured again, and calculated the size corresponding to the inserted data.

Table 3.4 OrientDB database sizes

Dataset size	names[1]	titles[1]	names[2]	titles[2]
1000	3 MB	5 MB	5 MB	8 MB
64000	144 MB	152 MB	22 MB	27 MB
256000	198 MB	358 MB	83 MB	83 MB
1024000	580 MB	627 MB	319 MB	368 MB

[1] Database size with Write-Ahead Log files
[2] Database size without Write-Ahead Log files and without common files

ScyllaDB The database size was easily obtained using `nodetool`[12], the provided admin utility tool with command line interface, using the `tablestats` command, which immediately gave us the total disk size of each of our tables.

Table 3.5 ScyllaDB table sizes

Dataset size	names	titles
1000	821 kB	774 kB
64000	6 MB	4 MB
256000	23 MB	16 MB
1024000	91 MB	60 MB

Couchbase Using the Couchbase Web Console interface we can easily find the disk size used for each of our collections and their scopes separately, alternatively, we can also use the Couchbase `cbstats`[13] with its `collections` and `scopes` commands to get information about our collections or scopes.

Table 3.6 Couchbase scope sizes

Dataset size	names	titles
1000	339 kB	260 kB
64000	15 MB	11 MB
256000	58 MB	43 MB
1024000	229 MB	171 MB

RavenDB Using the RavenDB Management Studio, the web interface, we can find the disk usage of our database, and find out how it is divided into our datasets and indexes, however RavenDB is quite lax in deleting temporary files, which

[12] https://opensource.docs.scylladb.com/stable/operating-scylla/nodetool.html
[13] https://docs.couchbase.com/server/current/cli/cbstats-intro.html

can be quite large, at 256 MB, furthermore RavenDB is also reserving quite a lot of extra free space, which gets included into the total database size, fortunately we can still see how big our actual datasets or indexes are, be they manual, or automatic, which were generated due to our queries. Moreover, RavenDB also uses lot of metadata, such as *Etags*[14], which are used to track changes of documents, and *Tombstones*[15], which are used as deletion markers to speed up apparent deletion and get periodically cleaned up, and other miscellaneous data.

Table 3.7 Ravendb database sizes

Dataset size	indexes	metadata[1]	names	titles
1000	2778 kB	352 kB	2600 kB	2360 kB
64000	8 MB	12 MB	27 MB	29 MB
256000	25 MB	49 MB	110 MB	98 MB
1024000	97 MB	196 MB	420 MB	387 MB

[1] This column contains the most direct metadata relating to documents, their Etags and Tombstones, although there were also other miscellaneous metadata, their disk size was comparably negligible.

Summary To compare between different database systems, the sizes of import files, be it in `csv` or `json` format, and the database systems that used these files were summarized into tables for each dataset.

In the following table 3.8, the dataset *names* was used in its `csv` format, which was used to import it into PostgreSQL, Virtuoso, and ScyllaDB.

Table 3.8 Names dataset and database systems with import from csv file

Dataset size	names.csv	PostgreSQL	Virtuoso	ScyllaDB
1000	104 kB	240 kB	91 kB	821 kB
64000	5662 kB	9 MB	5 MB	6 MB
256000	22 MB	34 MB	21 MB	23 MB
1024000	86 MB	132 MB	76 MB	91 MB

The table 3.9 was the *titles* dataset imported from the `csv` format into the same database systems, PostgreSQL, Virtuoso, and ScyllaDB.

Table 3.9 Titles dataset and database systems with import from csv file

Dataset size	titles.csv	PostgreSQL	Virtuoso	ScyllaDB
1000	50 kB	160 kB	35 kB	774 kB
64000	4 MB	7 MB	3 MB	4 MB
256000	15 MB	27 MB	11 MB	16 MB
1024000	58 MB	108 MB	43 MB	60 MB

On the other hand, database systems OrientDB, Couchbase, and RavenDB worked better with `json` file format, which is shown in the table 3.10 for the dataset *names*, and in the table 3.11 for the dataset *titles*.

Table 3.10 Names dataset and database systems with import from json file

Dataset size	names.json	OrientDB[1]	Couchbase	RavenDB[2]
1000	203 kB	5 MB	339 kB	2600 kB
64000	12009 kB	22 MB	15 MB	27 MB
256000	47 MB	83 MB	58 MB	110 MB
1024000	185 MB	319 MB	229 MB	420 MB

[1] This is the database size when ignoring the WAL and common files
[2] This is just the data size, without indexes and extra metadata

Table 3.11 Titles dataset and database systems with import from json file

Dataset size	titles.json	OrientDB[1]	Couchbase	RavenDB[2]
1000	122 kB	8 MB	260 kB	2360 kB
64000	8220 kB	27 MB	11 MB	29 MB
256000	33 MB	83 MB	43 MB	98 MB
1024000	130 MB	368 MB	171 MB	387 MB

[1] This is the database size when ignoring the WAL and common files
[2] This is just the data size, without indexes and extra metadata

From the summary tables 3.8, 3.9, 3.10, 3.11 we can see that databases that were imported from `csv` files, that is, PostgreSQL, Virtuoso, and ScyllaDB, could store the data in an optimized way, with Virtuoso being the smallest, closely followed by ScyllaDB and then by PostgreSQL, whilst the databases OrientDB, Couchbase, and RavenDB that worked with `json` documents were storing the data in a less optimized way, with Couchbase being the smallest, followed by OrientDB and RavenDB.

Import times Measuring import times in PostgreSQL were easy, we just need to enable it in *psql* with `\timing on`. Measuring time in virtuoso was also easy, *isql* was already doing it for us. In OrientDB the ETL tool *oetl* was once again measuring the time for us. As for ScyllaDB, the `COPY` command also gave us the elapsed times. Importing into Couchbase was using the command-line *cbimport* tool, therefore we could easily use the linux `time` command to measure our import times. And finally for RavenDB using bulk insert from the Python driver and timing how long it took using Python `time`[16] library's `text_counter` functions.

Each import was performed into empty tables or databases, and was performed 5 times and the average results are for names dataset in table 3.12 and for titles dataset in table 3.13.

In summary we can see that PostgreSQL, Virtuoso, ScyllaDB, and Couchbase were comparably fast in the range of seconds, whilst RavenDB and parallelized OrientDB imports were an order of magnitude slower, and ultimately non-parallelized OrientDB import was yet another order of magnitude slower.

[14] https://ravendb.net/docs/article-page/6.0/csharp/glossary/etag
[15] https://ravendb.net/docs/article-page/6.0/csharp/glossary/tombstone
[16] https://docs.python.org/3/library/time.html

Table 3.12 Names dataset import times in seconds

Dataset size	**PosgreSQL**[1]	**Virtuoso**[1]	**OrientDB**[2]	**ScyllaDB**[1]	**Couchbase**[3]	**RavenDB**
1000	0,011	0,020	1,008	0,422	0,326	0,215
64000	0,174	0,667	4,561	0,739	0,784	4,286
256000	0,635	2,530	15,484	2,792	1,635	16,946
1024000	2,268	8,532	58,033	10,460	4,979	66,960

[1] PostgreSQL, Virtuoso and ScyllaDB used internal commands, that were not possible to tune further.
[2] It is possible to set parallel mode for oetl import, and it was used in this names dataset.
[3] Couchbase's cbimport tool allows setting thread count, and recommends it to be no higher than cpu count, so it was set to cpu count.

Table 3.13 Titles dataset import times in seconds

Dataset size	**PosgreSQL**[1]	**Virtuoso**[1]	**OrientDB**[2]	**ScyllaDB**[1]	**Couchbase**[3]	**RavenDB**
1000	0,009	0,015	1,377	0,392	0,334	0,192
64000	0,146	0,609	18,410	0,752	0,806	4,395
256000	0,556	2,270	78,855	2,715	1,873	17,433
1024000	2,259	8,002	441,541	10,449	5,028	68,914

[1] PostgreSQL, Virtuoso and ScyllaDB used internal commands, that were not possible to tune further.
[2] It is possible to set parallel mode for oetl import, but during importing titles dataset with edge links to names dataset, the tool was having internal errors, therefore this import was not parallel.
[3] Couchbase's cbimport tool allows setting thread count, and recommends it to be no higher than cpu count, so it was set to cpu count.

3.5 Results

We use the same categories as defined in 3.1.1 to sort the following results, we usually performed each query 20 times and discarded the outliers, then we averaged over the rest. Usually the first query was among the outliers, due to various software and hardware reasons, such as not yet allocated memory, or some data not yet read from disk, or cold execution of machine code with untrained CPU branch predictors. Other outliers could have been due to general system jitter, such as the operating system performing its own task, or other users on the system performing their own tasks.

Projection In the first query we project two text attributes `primaryName` and `primaryProfession` with results in Figure 3.3.

We can see that object-relational databases such as PostgreSQL and Virtuoso can handle projection queries quickly, PostgreSQL more so, as it focuses heavily on performance. OrientDB, ScyllaDB and Couchbase come next, though it can be seen that the queries can take a lot of time with larger datasets.

Selection First we use selection on an exact value equality on `birthYear` and get results in Figure 3.4.

PostgreSQL and Virtuoso were on par with each other, with RavenDB being close, even for large datasets, the former take around 50 milliseconds, whilst the latter takes under 200 milliseconds. OrientDB then takes up to 3 seconds, whilst Couchbase and Scylla take around 10-12 seconds.

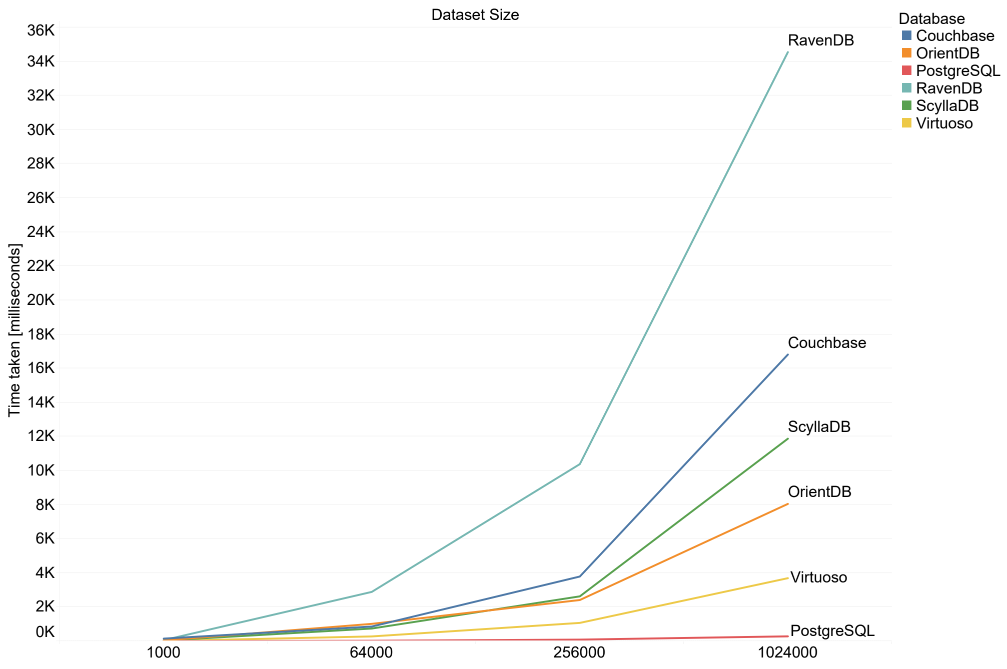

Figure 3.3 Projection

And in Figure 3.5 we select values of `birthYear` in between two years.

True to its performance claims, PostgreSQL still takes around 50 milliseconds for the large dataset, Virtuoso slows down to just under 300 milliseconds, but noteworthy RavenDB does not like this kind of query for values in between, it slow down to the same level of OrientDB, which along with Scylla and Couchbase do not distinguish between the two kinds of queries.

Aggregation Here in Figure 3.6 we aggregate the rows and count the number of rows in dataset.

PostgreSQL, Virtuoso, Couchbase, and RavenDB both comfortably return the result count under 50 milliseconds, yet ScyllaDB takes up to half a second, and OrientDB above 2 seconds.

Afterwards we have a query, where we get the maximum of `birthYear` in all rows resulting in Figure 3.7.

Selecting the max value yielded similar results, except with Couchbase, which instead of previous instant results, took almost 11 seconds. Unfortunately selecting a Max value in RavenDB is not supported out of the box, but may be possible with user defined map-reduce indexes.

Join A join query is used to gather for all names identified by an `nconst` all the titles (their `tconst`) where there is the same `nconst`, that is all titles each name was mentioned in, and the time taken is graphed into Figure 3.8.

Join queries are supported only in PostgreSQL, Virtuoso, and Couchbase, being the slowest on Couchbase, and up to dataset size of 256 000 rows, Virtuoso was almost up to par with PostgreSQL, which can handle even larger datasets

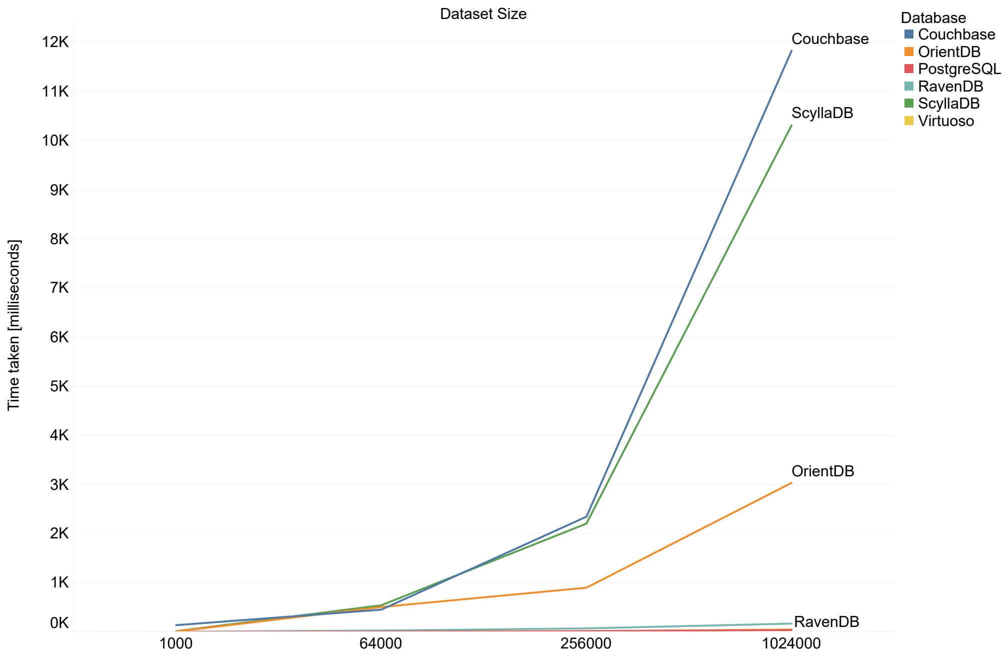

Figure 3.4 Selection 1

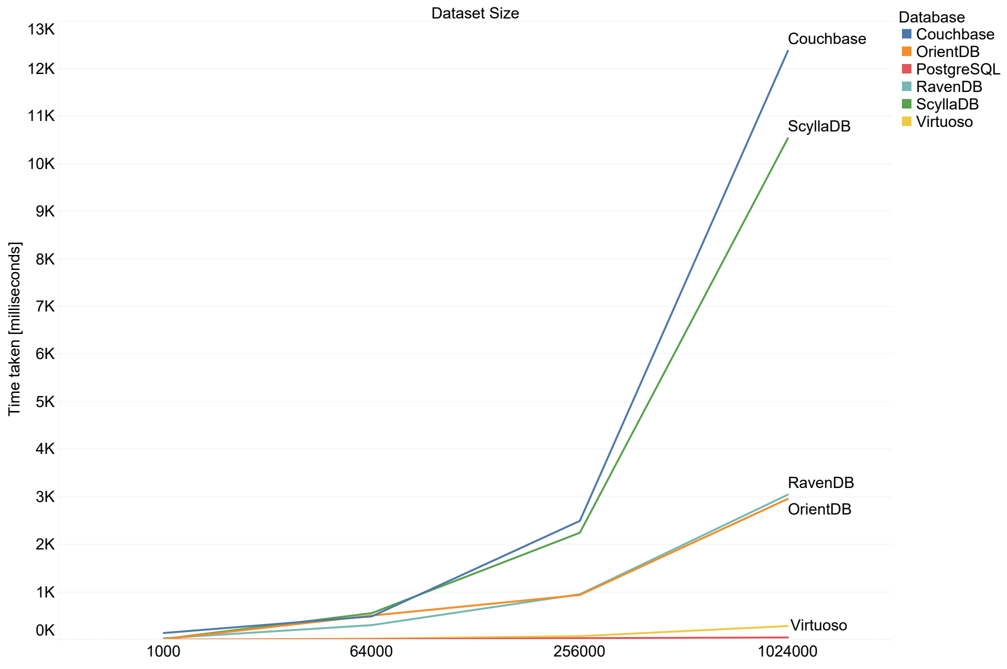

Figure 3.5 Selection 2

without significant slowdown.

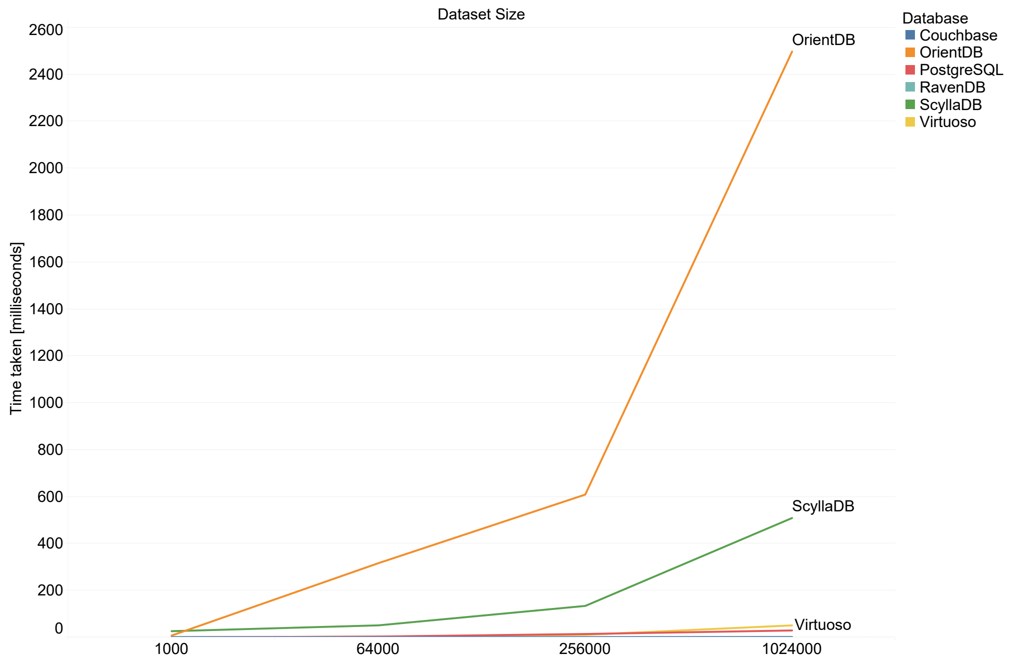

Figure 3.6 Aggregation 1 – Count

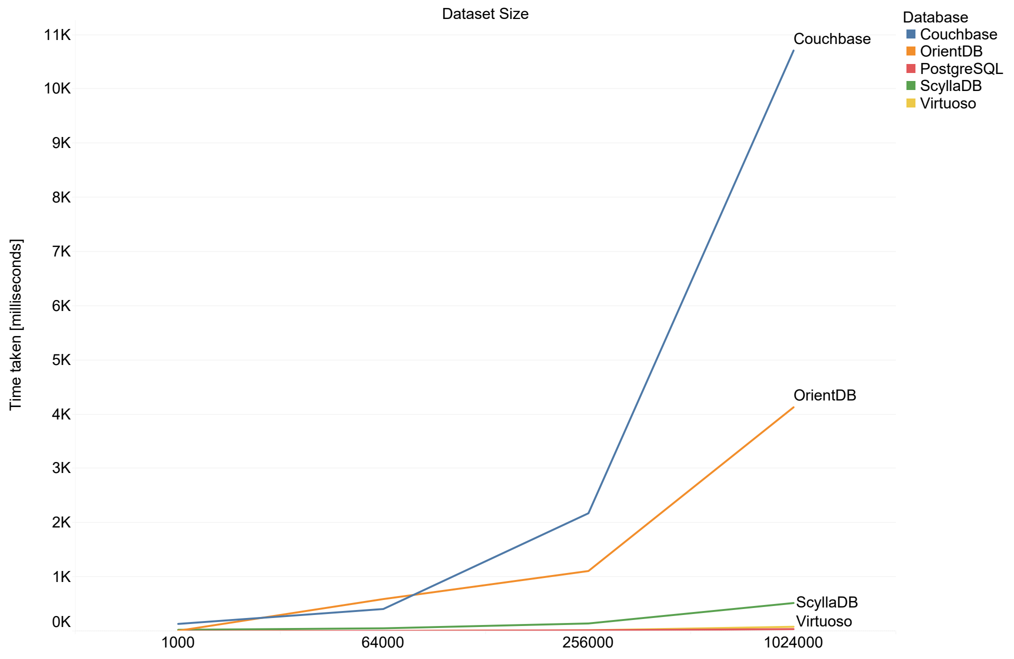

Figure 3.7 Aggregation 2 – Max

Union A query for union of all names identified by `nconst` found in names and titles, first with deduplication in Figure 3.9, then with duplicates (`UNION ALL`) inside Figure 3.10.

Unions best scale on Virtuoso and PostgreSQL, unexpectedly, on every system

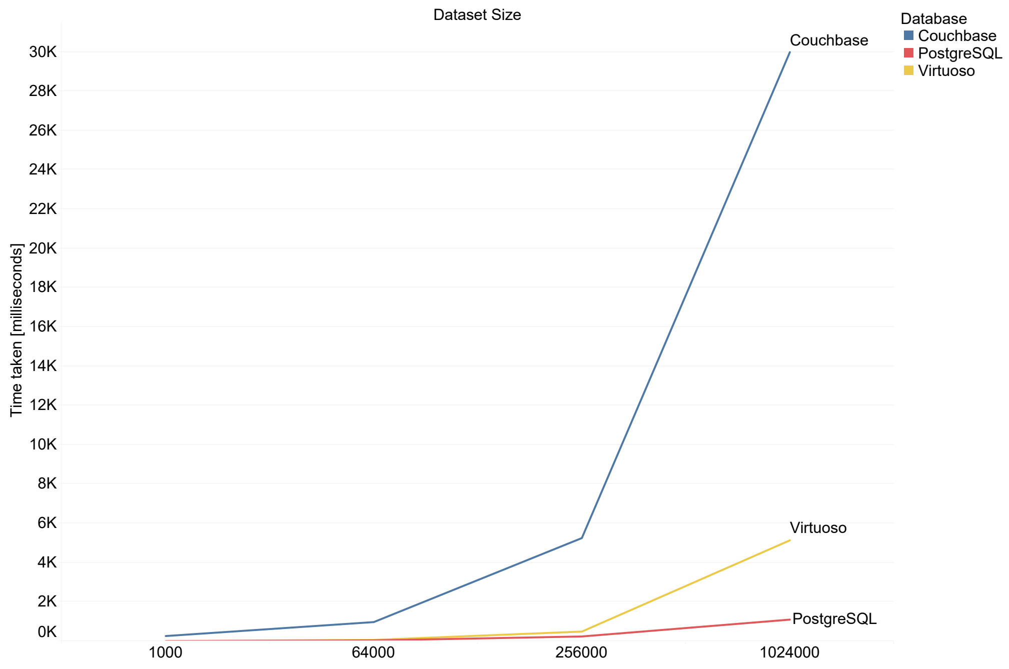

Figure 3.8 Join

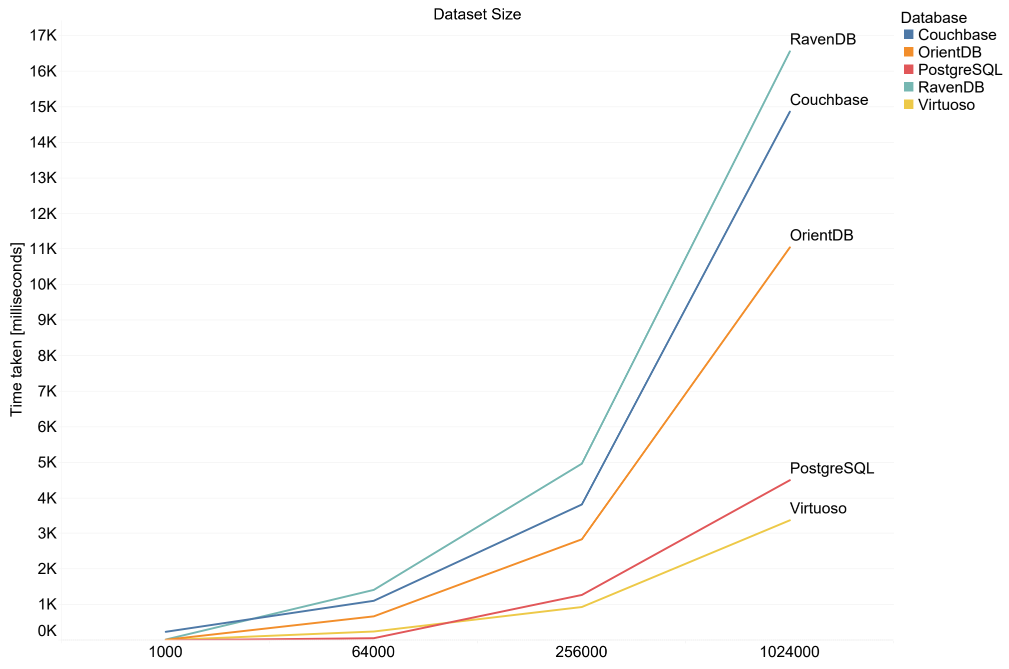

Figure 3.9 Union 1 – de-duplicated

beside PostgreSQL did union with duplicates fare worse, and RavenDB slowed significantly, and ScyllaDB does not support any unions.

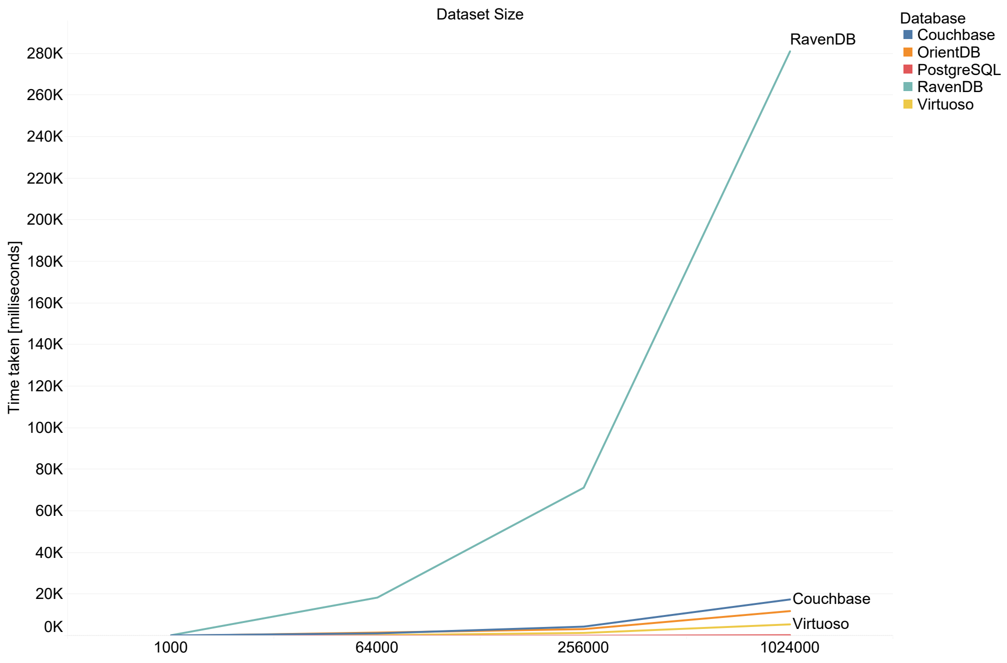

Figure 3.10 Union 2 – all

Intersection Here we get an intersection, that is names identified by `nconst` that are in both tables, and again we try both variants, with (Figure 3.11) and without (Figure 3.12) de-duplication.

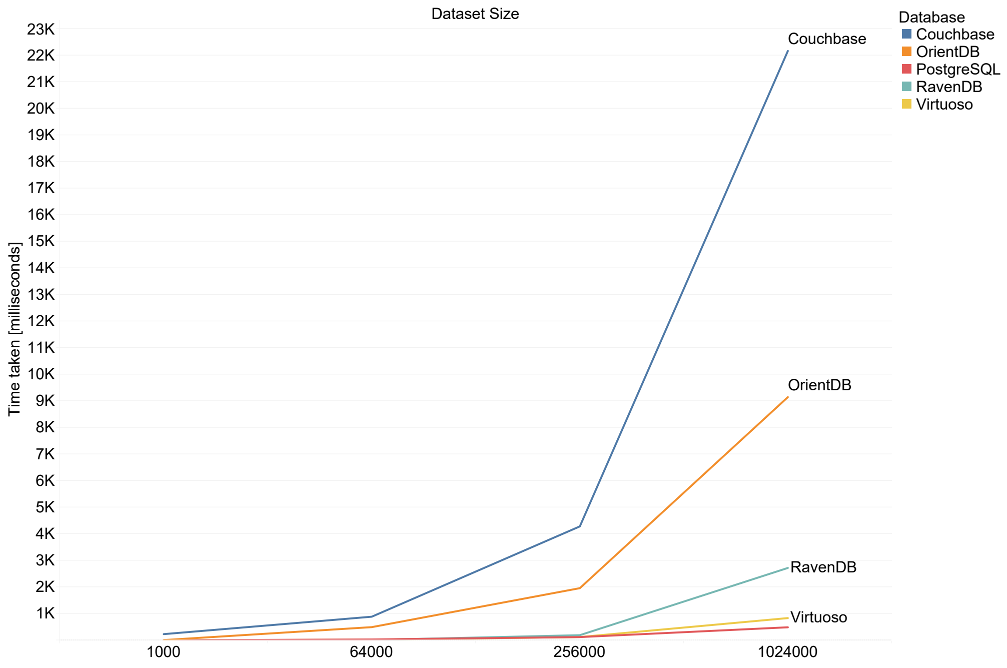

Figure 3.11 Intersect 1 – de-duplicated

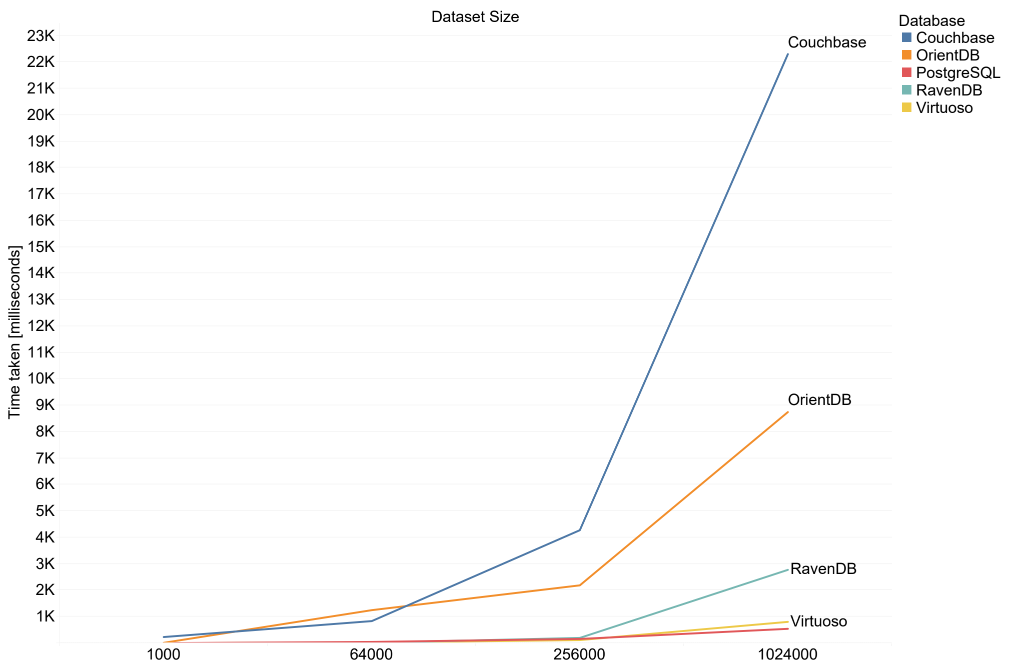

Figure 3.12 Intersect 2 – all

Both variants perform similarly, with PostgreSQL and Virtuoso performing under a second, RavenDB and OrientDB in matter of seconds, but Couchbase taking tens of seconds, again ScyllaDB has no support for intersection queries.

Difference Here we only get names identified by `nconst` found in names but not in titles in Figures 3.13 and 3.14.

Both variants with and without de-duplication perform similarly, however Couchbase is an order of magnitude slower than Virtuoso and PostgreSQL, and OrientDB could not give results before a 5 minute timeout, for anything other than dataset of size 1000.

Sorting First we try sorting by a text attribute `nconst` with results in 3.15.

And then we try sorting by non-unique integer attribute, `birthYear`, in Figure 3.16.

We see a dramatic increase in sort times as we go from dataset of size 25600 to dataset of size 1024000 in database systems RavenDB, OrientDB, and similarly in Couchbase, PostgreSQL is the most performant and Virtuoso second.

Limiting the total values returned and offsetting Here in Figure 3.17 we first try getting only a small number of 10 of results in specified order by `nconst`.

Then in Figure 3.18 we try the same query but only offsetting by a small value of 10.

And finally in Figure 3.19 we try both limiting by 10 and offsetting by 5.

When limiting values we see that each database system, except Couchbase, could give us our top 10 values almost immediately, and as expected, when we

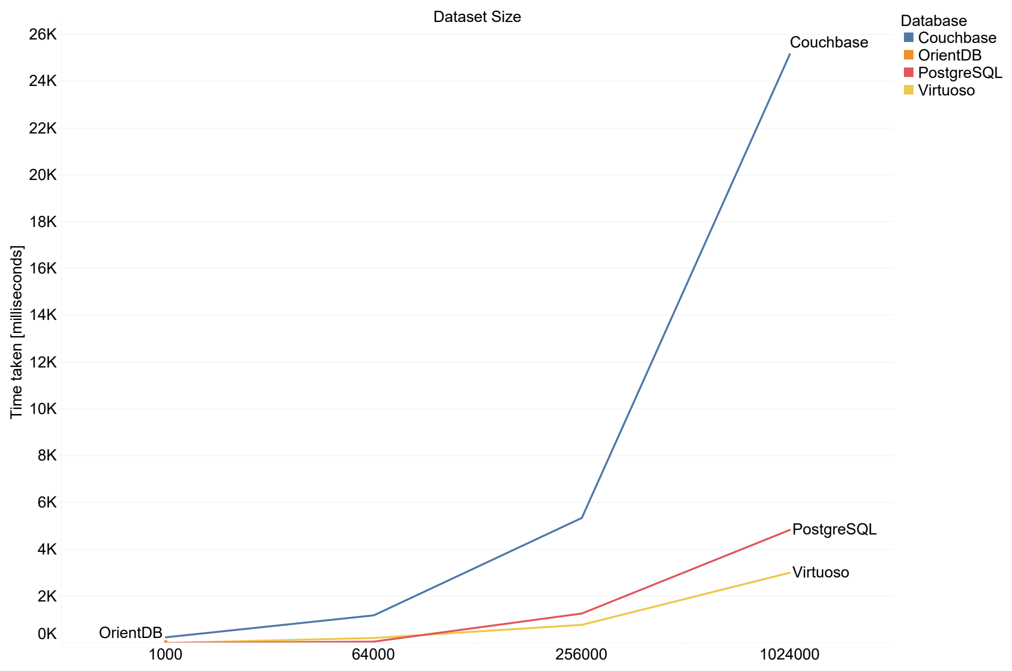

Figure 3.13 Difference 1 – de-duplicated

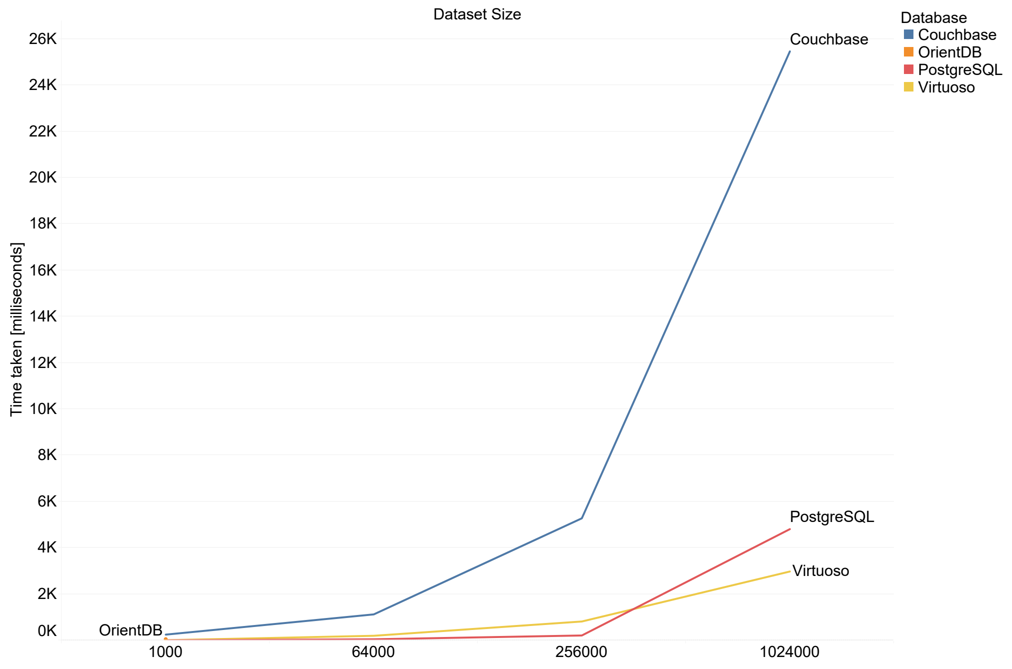

Figure 3.14 Difference 2 – all

only skip a few values, we get similar results as when we sorted all the values.

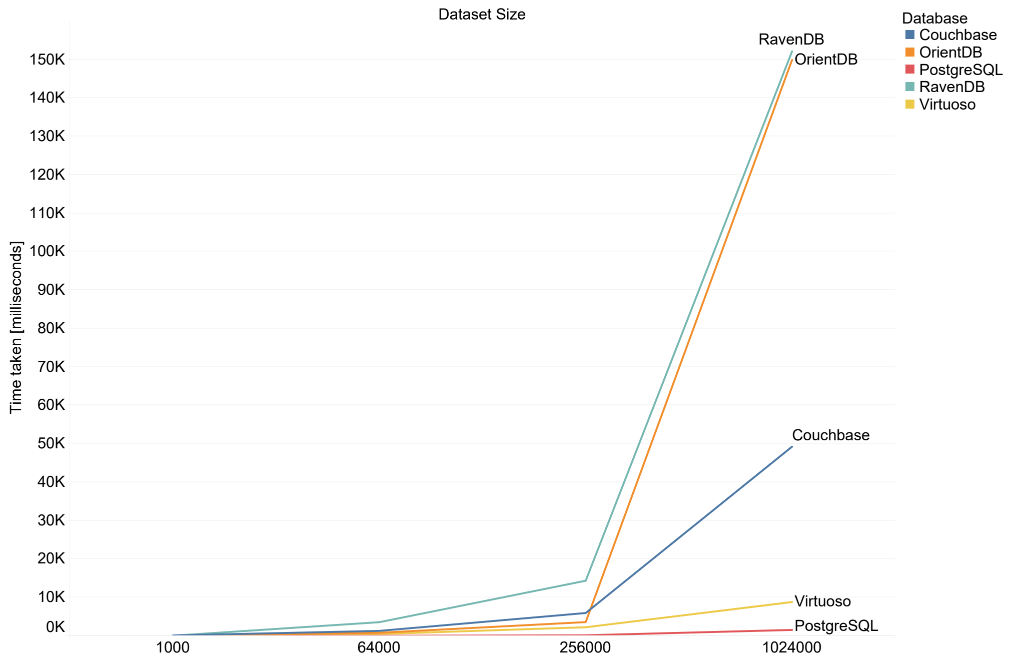

Figure 3.15 Sorting 1 – by text attribute

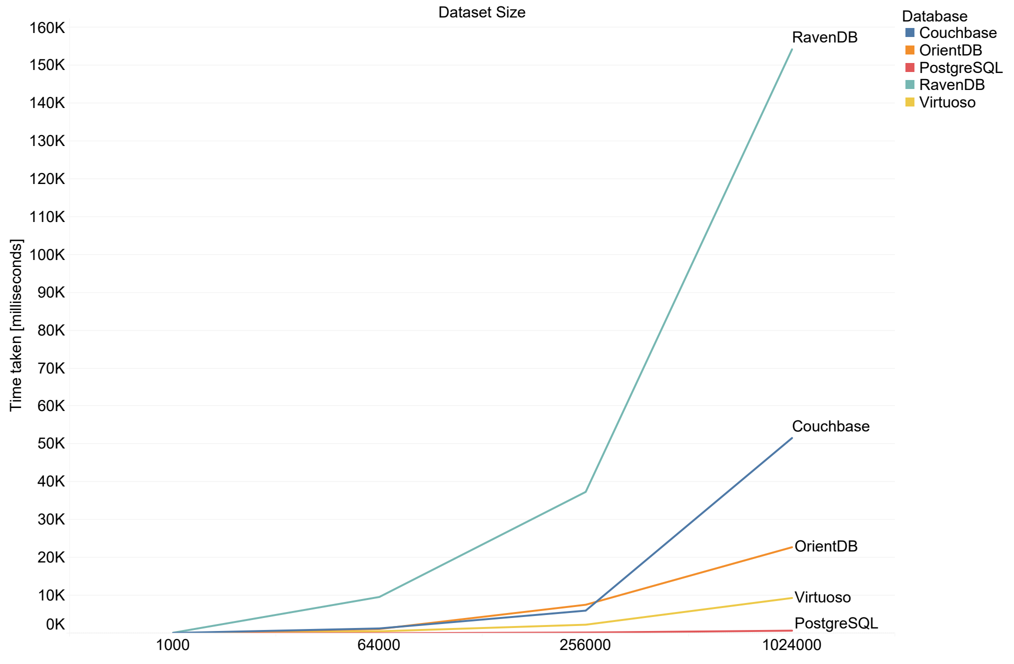

Figure 3.16 Sorting 2 – by integer attribute

Map-reduce And lastly we calculate the number of appearances of each name in titles, with the results in Figure 3.20.

We only compared PostgreSQL and Virtuoso, as other systems would need careful crafting of correct aggregates, however that would mean a lot of redundant

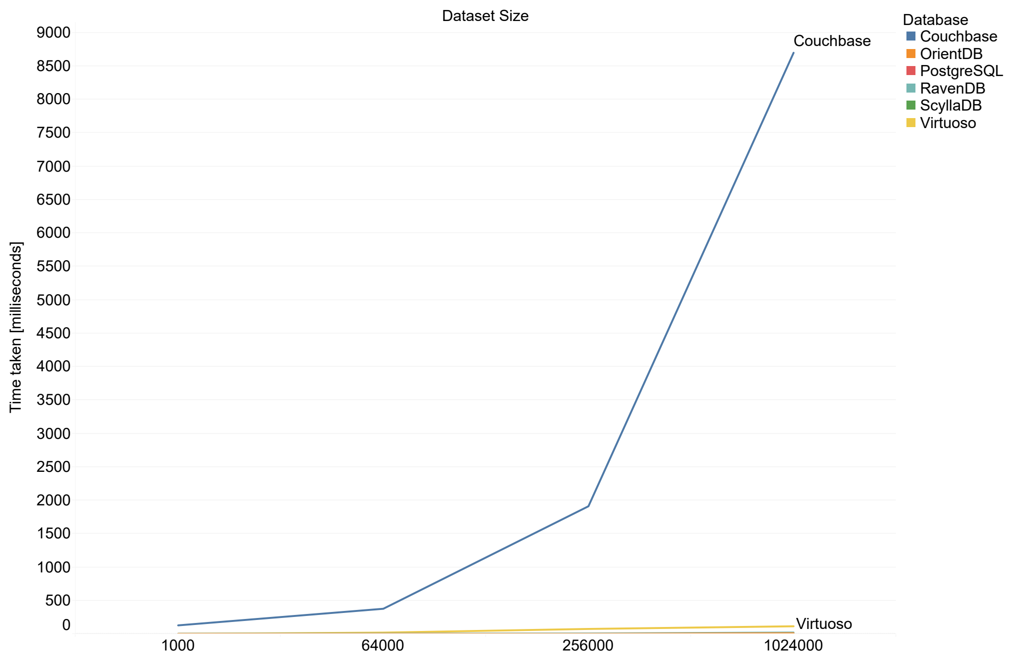

Figure 3.17 Limit 1 – top 10

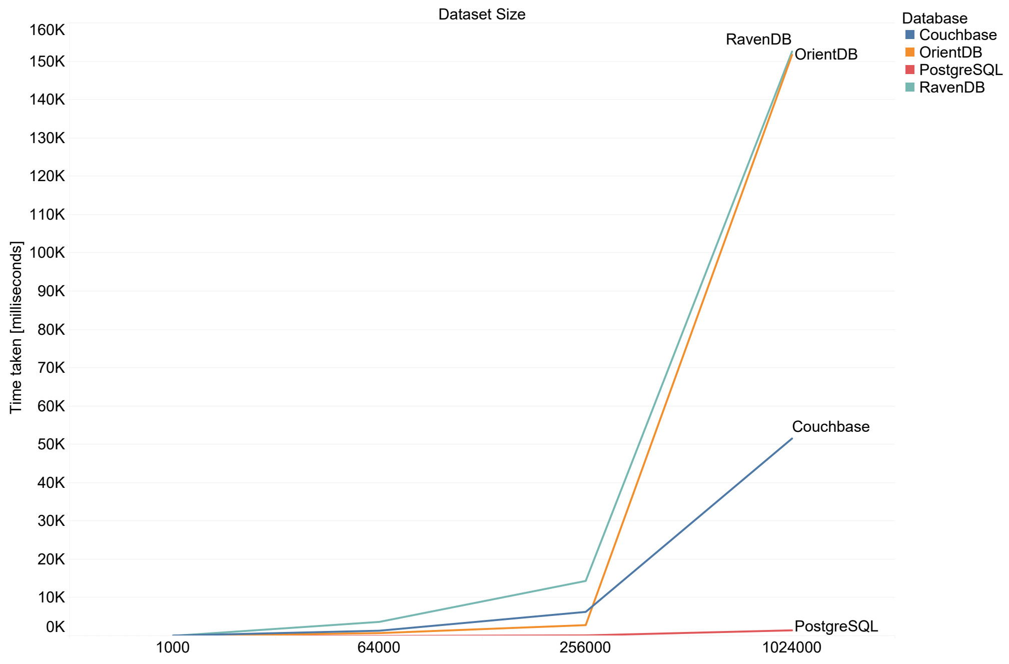

Figure 3.18 Limit 2 – skip 10

data, and no ad-hoc queries, as we would need to know the queries beforehand.

Summary If we take a look at the different datasets, we can see that usually we only need to look at the largest dataset to get a measure of how performant

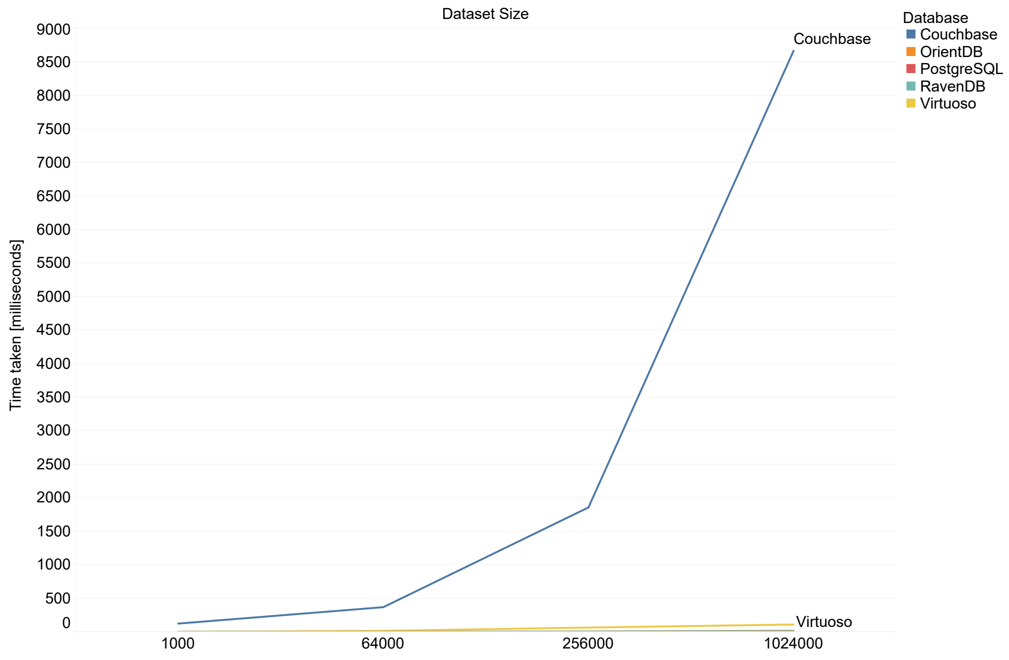

Figure 3.19 Limit 3 – skip 5 top 10

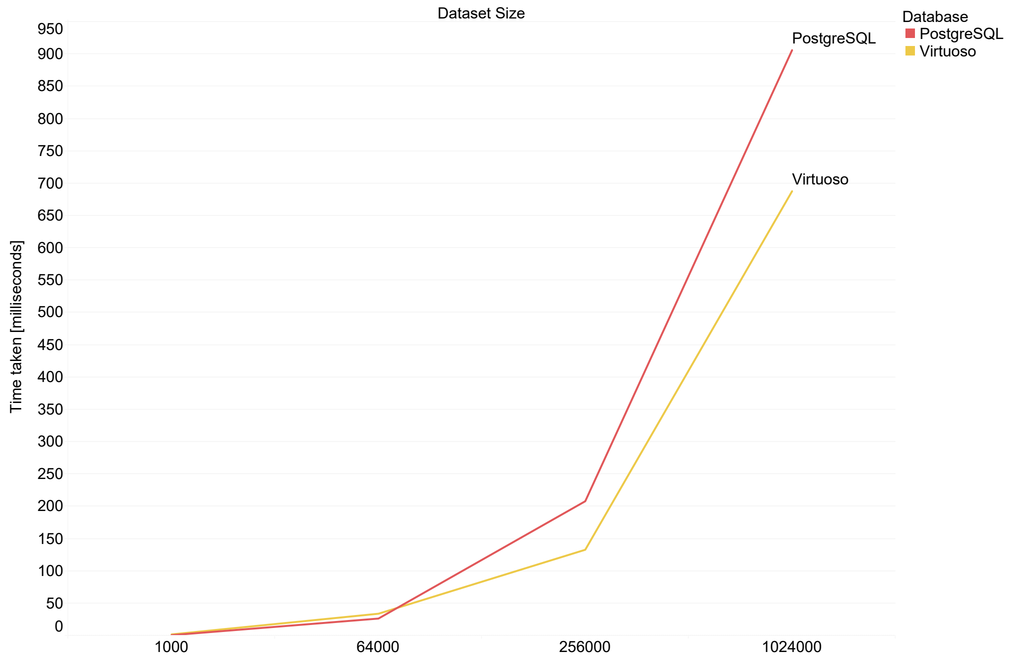

Figure 3.20 Map-reduce

a system was, therefore if we order the results, lowest time taken first, and sum them up over all the queries we get a number that reflects overall how well each database system handled itself, in relation to the others.

With the results in Table 3.14, we can see that PostgreSQL and Virtuoso did well in almost all queries, with OrientDB and RavenDB following next, and lastly Couchbase and ScyllaDB, where in the case of the former, the performance was lacking, while in the case of the latter, its supported features for queries were lacking, and together they were suffering from not well optimized data structures and aggregates.

Table 3.14 Ranking comparison between database systems based on query performance

Query Type	**PostgreSQL**	**Virtuoso**	**OrientDB**	**RavenDB**	**Couchbase**	**ScyllaDB**
aggregation1	1	1	2	1	1	2
aggregation2	1	1	3	1	4	2
difference1	2	1	4	4	3	4
difference2	2	1	4	4	3	4
distinct1	2	1	4	3	5	5
distinct2	1	2	3	4	5	5
intersect1	1	2	4	3	5	6
intersect2	1	2	4	3	5	6
join1	1	2	4	4	3	4
limit1	1	1	1	1	2	1
limit2	1	4	3	3	2	4
limit3	1	1	1	1	2	1
mapreduce1	2	1	3	3	3	3
project1	1	2	3	6	5	4
select1	1	1	3	2	5	4
select2	1	2	3	3	5	4
sorting1	1	2	4	5	3	6
sorting2	1	2	3	5	4	6
union1	2	1	3	5	4	6
union2	1	2	3	5	4	6
Total Points[1]	**25**	**32**	**62**	**66**	**73**	**83**

[1] Lower is better

4 Related work

As many database systems are nowadays designed to be multi-model, it is often the case that they aim to provide a single unified query language, in particular, SQL with extensions, a comprehensive survey have been made in [32].

For RDF data the query language SPARQL has been in use, and there are both native storage solutions, as for example Virtuoso offers, or SPARQL endpoints over relational databases, a benchmark and an evaluation of these virtual endpoints have been made in [33].

As data is represented in various ways in multi-model database systems, so is the querying of the data different, a universal approach to simplify this has been introduced in [6].

To help with evaluation of multi-model database systems, a performance and usability benchmark has been presented in [34].

A similar work of experimental analysis of query languages has been also made for SQLite[1], MySQL, Neo4j[2], ArrangoDB[3], Cassandara, and MongoDB in [35].

[1]https://sqlite.org/
[2]https://neo4j.com/
[3]https://arangodb.com/

Conclusion

We tasked ourselves to compare a select choice of database systems and choose the best one among them, to do so we made comparisons of their static features, query languages, database drivers for use with client applications, and finally what kind of queries can we make on the systems and how fast do they perform.

To help with our analysis, we also created a client library to extract and transform data from the IMDb datasets, so that we could load them into our systems and start performing the queries.

We first performed a static analysis of various database system features, such as the database model, various kinds of horizontal and vertical scaling, and what level of consistency levels can we achieve within the databases.

Then using our library we prepared our dataset to be imported into the database systems, where commonly used formats such as CSV, JSON, and XML were often used for data imports and some database systems even offered imports with direct connections to other database systems.

As for the database drivers, that we will want to use in our client applications, the compared database systems usually had multiple, and in commonly used programming languages. Some databases had a just one first-party driver, for example, *libpq* in PostgreSQL, however plenty of community made and open source adaptations or alternatives were available, although they may stop being maintained as was the case with many third-party drivers for OrientDB. Other database systems had multiple first-party drivers, as was the case in Scylla, Couchbase, OrientDB, and RavenDB, or also supported ODBC and JDBC connectors.

To create our desired quires in relational databases was quite easy, however other databases were either missing some features, as was mainly the case with Scylla, and or would require an experienced user to design the data model to work with the desired queries. Such an approach would usually incur some other costs, for example, redundancy of data, or complex update queries.

Finally measuring our queries yielded results such as PostgreSQL and Virtuoso, the only relational database systems on the list, outperforming other database systems, sometimes by orders of magnitude, therefore we expect these databases to be quite well suited to any chosen kind of queries. However as we already mentioned, careful planning with other database systems could find tasks that work quite well or better in those systems.

As for extending this work, interesting topics could be finding the tasks in which each database system does the best, or try searching for hidden correlations between different queries, that could when performed together yield faster results than when performed apart. Further work could go in performance tuning of the database systems, as they quite often have tens or hundreds of various performance knobs. As it is nowadays quite common for database systems to be multi-model, comparison between different logical representation of data in one database system could be of interest, as well as comparison of same logical representation of data between multiple systems.

From our work we conclude that as single servers PostgreSQL and Virtuoso offered great performance out of the box, however the NoSQL databases lagged behind, as they were designed from the ground up to take greater benefits from horizontal scaling with multiple server nodes [36][37][38][39], therefore a comparison in a multi-server environment could also be of further interest.

Bibliography

1. Lu, Jiaheng; Holubová, Irena. Multi-model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.* 2019, vol. 52, no. 3. issn 0360-0300.
2. Codd, E. F. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM.* 1970, vol. 13, no. 6, pp. 377–387.
3. Holubová, Irena; Kosek, Jiří; Minařík, Karel; Novák, David. *Big Data a NoSQL databáze*. Grada Publishing, as, 2015.
4. Sadalage, Pramod J; Fowler, Martin. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2013.
5. ISO. *ISO/IEC 9075-1:2008 Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. ISO, 2008.
6. Koupil, Pavel; Crha, Daniel; Holubová, Irena. A Universal Approach for Simplified Redundancy-Aware Cross-Model Querying. *Available at SSRN 4596127*. 2023.
7. Little, Mark. Transactions and web services. *Communications of the ACM*. 2003, vol. 46, no. 10, pp. 49–54.
8. Pritchett, Dan. BASE: An Acid Alternative. *ACM Queue*. 2008, vol. 6, no. 3, pp. 48–55.
9. *MySQL :: MySQL 8.4 Reference Manual :: 1.7 MySQL Standards Compliance — dev.mysql.com* [https://dev.mysql.com/doc/refman/8.4/en/compatibility.html]. [N.d.]. [Accessed 15-07-2024].
10. *Appendix D. SQL Conformance — PostgreSQL Documentation* [https://www.postgresql.org/docs/current/features.html]. [N.d.]. [Accessed 15-07-2024].
11. *SQL Features That SQLite Does Not Implement — sqlite.org* [https://www.sqlite.org/omitted.html]. [N.d.]. [Accessed 15-07-2024].
12. *Citus Data | Distributed Postgres. At any scale. — citusdata.com* [https://www.citusdata.com/]. [N.d.]. [Accessed 11-06-2024].
13. *Virtuoso XML Functionality* [https://vos.openlinksw.com/owiki/wiki/VOS/VOSXML]. [N.d.]. [Accessed 16-07-2024].
14. W3C. *XQuery 1.0: An XML Query Language (Second Edition)*. 2015.
15. W3C. *XML Path Language (XPath) Version 1.0*. 2015.
16. *15.9. XSLT Transformation* [https://docs.openlinksw.com/virtuoso/xslttrans]. [N.d.]. [Accessed 16-07-2024].
17. *XML Composing Functions in SQL Statements (SQLX)* [https://docs.openlinksw.com/virtuoso/composingxmlinsql]. [N.d.]. [Accessed 16-07-2024].
18. *Chapter 16. RDF Data Access and Data Management* [https://docs.openlinksw.com/virtuoso/ch-rdfandsparql/]. [N.d.]. [Accessed 16-07-2024].

19. CMS, Tenrec. *What is Cassandra Query Language (CQL)? Definition & FAQs | ScyllaDB — scylladb.com* [https://www.scylladb.com/glossary/cassandra-query-language-cql/]. [N.d.]. [Accessed 15-07-2024].

20. *SQL++ | Couchbase — couchbase.com* [https://www.couchbase.com/sqlplusplus/]. [N.d.]. [Accessed 15-07-2024].

21. *The backbone of financial transactions is atomic. — vspry.com* [https://www.vspry.com/acid-financial-transaction-processing/]. [N.d.]. [Accessed 15-07-2024].

22. *How Discord Stores Trillions of Messages — discord.com* [https://discord.com/blog/how-discord-stores-trillions-of-messages]. [N.d.]. [Accessed 15-07-2024].

23. *Multi-Map-Reduce Indexes — ravendb.net* [https://ravendb.net/docs/article-page/6.0/csharp/indexes/map-reduce-indexes#creating-multi-map-reduce-indexes]. [N.d.]. [Accessed 15-07-2024].

24. *Oracle® Data Profiling and Oracle Data Quality for Data Integrator Sample Tutorial* [https://www.oracle.com/technetwork/middleware/data-integrator/learnmore/oracledq-tutorial-132022.pdf]. [N.d.]. [Accessed 15-07-2024].

25. *What is ETL (Extract, Transform, Load)? | IBM — ibm.com* [https://www.ibm.com/topics/etl]. [N.d.]. [Accessed 15-07-2024].

26. *pandas.DataFrame.to_json — pandas 2.2.2 documentation — pandas.pydata.org* [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_json.html]. [N.d.]. [Accessed 17-06-2024].

27. *Virtuoso CSV File Bulk Loader* [https://vos.openlinksw.com/owiki/wiki/VOS/VirtCsvFileBulkLoader]. [N.d.]. [Accessed 11-07-2024].

28. *ETL - OrientDB — orientdb.com* [https://orientdb.com/docs/latest/etl/ETL-Introduction.html]. [N.d.]. [Accessed 11-07-2024].

29. *cbimport | Couchbase Docs* [https://docs.couchbase.com/server/current/tools/cbimport.html]. [N.d.]. [Accessed 11-07-2024].

30. *IMDb | Help — help.imdb.com* [https://help.imdb.com/article/imdb/general-information/what-is-imdb/G836CY29Z4SGNMK5]. [N.d.]. [Accessed 15-07-2024].

31. *Performance Tuning* [https://docs.openlinksw.com/virtuoso/ptune/#statusfunc_01]. [N.d.]. [Accessed 16-07-2024].

32. GUO, Qingsong; ZHANG, Chao; ZHANG, Shuxun; LU, Jiaheng. Multi-model query languages: taming the variety of big data. *Distributed and Parallel Databases*. 2023, vol. 42, no. 1, pp. 31–71. ISSN 1573-7578. Available from DOI: 10.1007/s10619-023-07433-1.

33. CHALOUPKA, Milos; NECASKY, Martin. Using Berlin SPARQL benchmark to evaluate virtual SPARQL endpoints over relational databases. *Data & Knowledge Engineering*. 2024, vol. 152, p. 102309. ISSN 0169-023X. Available from DOI: 10.1016/j.datak.2024.102309.

34. Zhang, Chao; Lu, Jiaheng; Xu, Pengfei; Chen, Yuxing. UniBench: A Benchmark for Multi-model Database Management Systems. In: Nambiar, Raghunath; Poess, Meikel (eds.). *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence*. Cham: Springer International Publishing, 2019, pp. 7–23. isbn 978-3-030-11404-6.

35. Čorovčák, Martin. *Experimental Analysis of Query Languages in Modern Database Systems*. Czech Republic, 2024. Master Thesis. Charles University.

36. Angell, Julia. *ScyllaDB | Modern NoSQL Database Architecture — scylladb.com* [https://www.scylladb.com/product/technology/]. [N.d.]. [Accessed 18-07-2024].

37. *High availability — ravendb.net* [https://ravendb.net/why-ravendb/high-availability]. [N.d.]. [Accessed 18-07-2024].

38. *Couchbase Architecture Innovations — couchbase.com* [https://couchbase.com/developers/architecture/]. [N.d.]. [Accessed 18-07-2024].

39. *Distributed Architecture - OrientDB — orientdb.com* [https://orientdb.com/docs/last/distributed/Distributed-Architecture.html]. [N.d.]. [Accessed 18-07-2024].

List of Figures

2.1 Framework dependency graph 25
2.2 Usage flowchart 25

3.1 Names transformation diagram 45
3.2 Titles transformation diagram 46
3.3 Projection 51
3.4 Selection 1 52
3.5 Selection 2 52
3.6 Aggregation 1 – Count 53
3.7 Aggregation 2 – Max 53
3.8 Join 54
3.9 Union 1 – de-duplicated 54
3.10 Union 2 – all 55
3.11 Intersect 1 – de-duplicated 55
3.12 Intersect 2 – all 56
3.13 Difference 1 – de-duplicated 57
3.14 Difference 2 – all 57
3.15 Sorting 1 – by text attribute 58
3.16 Sorting 2 – by integer attribute 58
3.17 Limit 1 – top 10 59
3.18 Limit 2 – skip 10 59
3.19 Limit 3 – skip 5 top 10 60
3.20 Map-reduce 60

List of Tables

1.1 Comparison of individual database features 20
1.2 Comparison of Data Definition Language (DDL) features 21
1.3 Comparison of Data Modeling Language (DML) Features 22

3.1 Raw file size comparison 45
3.2 PostgreSQL table sizes 46
3.3 Virtuoso table sizes 46
3.4 OrientDB database sizes 47
3.5 ScyllaDB table sizes 47
3.6 Couchbase scope sizes 47
3.7 Ravendb database sizes 48
3.8 Names dataset and database systems with import from csv file . . 48
3.9 Titles dataset and database systems with import from csv file . . 48
3.10 Names dataset and database systems with import from json file . 49
3.11 Titles dataset and database systems with import from json file . . 49
3.12 Names dataset import times in seconds 50
3.13 Titles dataset import times in seconds 50
3.14 Ranking comparison between database systems based on query performance 61

List of Abbreviations

ACID Atomicity, Consistency, Isolation, Durability. 10, 11, 13, 14, 16–18, 20

API Application Programming Interface. 15, 24, 26, 33, 34

BASE Basically Available, Soft state, Eventual consistency. 10, 15–18, 20

CPU Central Processing Unit. 10, 12, 13, 15, 17, 37, 50

CQL Cassandra Query Language. 15, 19, 20

CSV Comma-Separated Values. 23, 24, 26, 33, 34, 42–44, 63, 71

ETL Extract, Transform, Load. 23, 33, 42, 43, 49, 71

JDBC Java Database Connectivity. 32–34, 40, 63

JSON JavaScript Object Notation. 9, 11, 12, 15–18, 23, 24, 26, 27, 33, 42–44, 63, 71

MED Management of External Data. 11

NoSQL Not only SQL. 7, 9, 10, 15, 17, 23, 34, 63

ODBC Open Database Connectivity. 32, 34, 39, 63

RDF Resource Description Framework. 9, 13, 20, 62

RQL Raven Query Language. 17, 20

SDK Software Development Kit. 21, 33

SPARQL SPARQL Protocol and RDF Query Language. 13, 62

SQL Structured Query Language. 10, 11, 13–20, 23, 32, 34, 38, 39, 44

SQL++ The next-generation query language for managing JSON data. 16, 20

SSD Solid State Drive. 37

UDT User Defined Type. 11

WAL Write-Ahead Logging. 12, 49

XML eXtensible Markup Language. 9, 11–13, 23, 24, 26, 63

A Attachments

A.1 etl

Folder etl contains the *etlpy* library file, and the dataset transformation script.

A.2 imports

Folder imports contains folders 1000, 64000, 256000, 1024000, which contain the datasets in CSV, and JSON file formats, plus config file for Virtuoso, and ETL config files for OrientDB. Next imports folder contains the folder scripts with Java Maven project for OrientDB, and python scripts for other databases for measuring the query performance.

A.3 results_data

Folder results_data contains measured results of query performances.

A.4 PostgreSQL queries

```
SELECT primaryName, primaryProfession FROM name_basics;
SELECT primaryName, primaryProfession FROM name_basics WHERE birthYear
↪   = 1963;
SELECT primaryName, primaryProfession FROM name_basics WHERE birthYear
↪   BETWEEN 1950 AND 1970;
SELECT COUNT(*) FROM name_basics;
SELECT MAX(birthYear) FROM name_basics;
SELECT name_basics.nconst, name_basics.primaryName,
↪   title_principals.tconst, title_principals.nconst FROM name_basics
↪   INNER JOIN title_principals ON name_basics.nconst =
↪   title_principals.nconst;
SELECT nconst FROM name_basics UNION SELECT nconst FROM
↪   title_principals;
SELECT nconst FROM name_basics INTERSECT SELECT nconst FROM
↪   title_principals;
SELECT nconst FROM name_basics EXCEPT SELECT nconst FROM
↪   title_principals;
SELECT * FROM name_basics ORDER BY nconst;
SELECT * FROM name_basics ORDER BY birthYear;
SELECT DISTINCT primaryProfession, pk_id FROM name_basics ORDER BY
↪   pk_id;
SELECT primaryProfession, pk_id FROM name_basics ORDER BY pk_id;
SELECT * FROM name_basics ORDER BY pk_id LIMIT 10;
SELECT * FROM name_basics ORDER BY pk_id OFFSET 10;
SELECT * FROM name_basics ORDER BY pk_id LIMIT 10 OFFSET 5;
SELECT name_basics.pk_id, COUNT(*) AS name_count FROM name_basics JOIN
↪  title_principals ON title_principals.nconst = name_basics.nconst
↪  GROUP BY name_basics.pk_id;
SELECT nconst FROM name_basics UNION ALL SELECT nconst FROM
↪  title_principals;
SELECT nconst FROM name_basics INTERSECT ALL SELECT nconst FROM
↪  title_principals;
SELECT nconst FROM name_basics EXCEPT ALL SELECT nconst FROM
↪  title_principals;
```

A.5 Virtuoso queries

```
SELECT primaryName, primaryProfession FROM CSV.DBA.name_basics_csv
SELECT primaryName, primaryProfession FROM CSV.DBA.name_basics_csv
↪  WHERE birthYear = 1963
SELECT primaryName, primaryProfession FROM CSV.DBA.name_basics_csv
↪  WHERE birthYear BETWEEN 1950 AND 1970
SELECT COUNT(*) FROM CSV.DBA.name_basics_csv
SELECT MAX(birthYear) FROM CSV.DBA.name_basics_csv
SELECT CSV.DBA.name_basics_csv.nconst,
↪  CSV.DBA.name_basics_csv.primaryName,
↪  CSV.DBA.title_principals_csv.tconst,
↪  CSV.DBA.title_principals_csv.nconst FROM CSV.DBA.name_basics_csv
↪  INNER JOIN CSV.DBA.title_principals_csv ON
↪  CSV.DBA.name_basics_csv.nconst =
↪  CSV.DBA.title_principals_csv.nconst
SELECT nconst FROM CSV.DBA.name_basics_csv UNION SELECT nconst FROM
↪  CSV.DBA.title_principals_csv
SELECT nconst FROM CSV.DBA.name_basics_csv INTERSECT SELECT nconst FROM
↪  CSV.DBA.title_principals_csv
SELECT nconst FROM CSV.DBA.name_basics_csv EXCEPT SELECT nconst FROM
↪  CSV.DBA.title_principals_csv
SELECT nconst FROM CSV.DBA.name_basics_csv EXCEPT ALL SELECT nconst
↪  FROM CSV.DBA.title_principals_csv
SELECT * FROM CSV.DBA.name_basics_csv ORDER BY nconst
SELECT * FROM CSV.DBA.name_basics_csv ORDER BY birthYear
SELECT DISTINCT primaryProfession FROM CSV.DBA.name_basics_csv ORDER BY
↪  nconst
SELECT primaryProfession FROM CSV.DBA.name_basics_csv ORDER BY nconst
SELECT TOP 10 * FROM CSV.DBA.name_basics_csv ORDER BY nconst
SELECT TOP 5, 10 * FROM CSV.DBA.name_basics_csv ORDER BY nconst
SELECT nconst FROM CSV.DBA.name_basics_csv UNION ALL SELECT nconst FROM
↪  CSV.DBA.title_principals_csv
SELECT nconst FROM CSV.DBA.name_basics_csv INTERSECT ALL SELECT nconst
↪  FROM CSV.DBA.title_principals_csv
SELECT CSV.DBA.name_basics_csv.pk_id, COUNT(*) AS name_count FROM
↪  CSV.DBA.name_basics_csv JOIN CSV.DBA.title_principals_csv ON
↪  CSV.DBA.title_principals_csv.nconst =
↪  CSV.DBA.name_basics_csv.nconst GROUP BY
↪  CSV.DBA.name_basics_csv.pk_id
```

A.6 OrientDB queries

```
SELECT primaryName, primaryProfession FROM NameBasic TIMEOUT 300000
SELECT primaryName, primaryProfession FROM NameBasic WHERE birthYear =
 ↪  1963 TIMEOUT 300000
SELECT primaryName, primaryProfession FROM NameBasic WHERE birthYear
 ↪  BETWEEN 1950 AND 1970 TIMEOUT 300000
SELECT COUNT(*) FROM NameBasic TIMEOUT 300000
SELECT MAX(birthYear) FROM NameBasic TIMEOUT 300000
SELECT EXPAND( $c ) LET $a = ( SELECT nconst FROM NameBasic ), $b = (
 ↪  SELECT nconst FROM TitlePrincipal ), $c = UNIONALL( $a, $b )
 ↪  TIMEOUT 300000
SELECT EXPAND( $c ) LET $a = ( SELECT nconst FROM NameBasic ), $b = (
 ↪  SELECT nconst FROM TitlePrincipal ), $c = INTERSECT( $a, $b )
 ↪  TIMEOUT 300000
SELECT EXPAND( $c ) LET $a = ( SELECT nconst FROM NameBasic ), $b = (
 ↪  SELECT nconst FROM TitlePrincipal ), $c = DIFFERENCE( $a, $b )
 ↪  TIMEOUT 300000
SELECT * FROM NameBasic ORDER BY nconst TIMEOUT 300000
SELECT * FROM NameBasic ORDER BY birthYear TIMEOUT 300000
SELECT DISTINCT primaryProfession FROM NameBasic ORDER BY nconst
 ↪  TIMEOUT 300000
SELECT primaryProfession FROM NameBasic ORDER BY nconst TIMEOUT 300000
SELECT * FROM NameBasic ORDER BY nconst LIMIT 10 TIMEOUT 300000
SELECT * FROM NameBasic ORDER BY nconst SKIP 5 TIMEOUT 300000
SELECT * FROM NameBasic ORDER BY nconst LIMIT 10 SKIP 5 TIMEOUT 300000
SELECT EXPAND( $c.asSet() ) LET $a = ( SELECT nconst FROM NameBasic ),
 ↪  $b = ( SELECT nconst FROM TitlePrincipal ), $c = UNIONALL( $a, $b )
 ↪  TIMEOUT 300000
SELECT EXPAND( $c.asSet() ) LET $a = ( SELECT nconst FROM NameBasic ),
 ↪  $b = ( SELECT nconst FROM TitlePrincipal ), $c = INTERSECT( $a, $b
 ↪  ) TIMEOUT 300000
SELECT EXPAND( $c.asSet() ) LET $a = ( SELECT nconst FROM NameBasic ),
 ↪  $b = ( SELECT nconst FROM TitlePrincipal ), $c = DIFFERENCE( $a, $b
 ↪  ) TIMEOUT 300000
```

A.7 ScylllaDB queries

```
SELECT primaryName, primaryProfession FROM name_basics
SELECT primaryName, primaryProfession FROM name_basics WHERE birthYear
 ↪  = 1963 ALLOW FILTERING
SELECT primaryName, primaryProfession FROM name_basics WHERE birthYear
 ↪  >= 1950 AND birthYear <= 1970 ALLOW FILTERING;
SELECT COUNT(*) FROM name_basics
SELECT MAX(birthYear) FROM name_basics
SELECT * FROM name_basics LIMIT 10
```

A.8 Couchbase queries

```
SELECT primaryName, primaryProfession FROM name_basics
SELECT primaryName, primaryProfession FROM name_basics WHERE birthYear = 1963
SELECT primaryName, primaryProfession FROM name_basics WHERE birthYear BETWEEN 1950 AND 1970
SELECT COUNT(*) FROM name_basics
SELECT MAX(birthYear) FROM name_basics
SELECT name_basics.nconst as nbnconst, name_basics.primaryName, title_principals.tconst, title_principals.nconst as tpnconst FROM name_basics INNER JOIN title_principals ON name_basics.nconst = title_principals.nconst
SELECT nconst FROM name_basics UNION SELECT nconst FROM title_principals
SELECT nconst FROM name_basics UNION ALL SELECT nconst FROM title_principals
SELECT nconst FROM name_basics INTERSECT SELECT nconst FROM title_principals
SELECT nconst FROM name_basics INTERSECT ALL SELECT nconst FROM title_principals
SELECT nconst FROM name_basics EXCEPT SELECT nconst FROM title_principals
SELECT nconst FROM name_basics EXCEPT ALL SELECT nconst FROM title_principals
SELECT * FROM name_basics ORDER BY nconst
SELECT * FROM name_basics ORDER BY birthYear
SELECT * FROM name_basics ORDER BY nconst LIMIT 10
SELECT * FROM name_basics ORDER BY nconst OFFSET 10
SELECT * FROM name_basics ORDER BY nconst LIMIT 10 OFFSET 5
```

A.9 RavenDB queries and indexes

```
from Name.basics select primaryName, primaryProfession
from Name.basics where birthYear = 1963 select primaryName, primaryProfession
from Name.basics where birthYear between 1950 and 1970 select primaryName, primaryProfession
from Name.basics order by nconst
from Name.basics order by nconst select distinct primaryProfession
from Name.basics order by nconst select primaryProfession
from Name.basics order by nconst limit 0, 10
from Name.basics order by nconst offset 5
from Name.basics order by nconst limit 5, 10
from Name.basics limit 0, 0
from index union1
from index union2
from index intersect1
from index intersect2
from Name.basics order by birthYear
```

union1

```
from n in docs["Name.basics"]
select new
{
    nconst = n.nconst
}

from t in docs["Title.principals"]
select new
{
    nconst = t.nconst
}

from result in results
group result by result.nconst into g
select new
{
    nconst = g.Key
}
```

intersect1

```
from n in docs["Name.basics"]
select new
{
    nconst = n.nconst,
    source = 1
}

from t in docs["Title.principals"]
select new
{
    nconst = t.nconst,
    source = 2
}

from result in results
group result by result.nconst into g
let namecount = g.Count(x => x.source == 1)
let titlecount = g.Count(x => x.source == 2)
where namecount > 0 && titlecount > 0
select new
{
    nconst = g.Key,
    source = 3
}
```

union2

```
from n in docs["Name.basics"]
select new
{
    nconst = n.nconst
}

from t in docs["Title.principals"]
select new
{
    nconst = t.nconst
}
```

intersect2

```
from n in docs["Name.basics"]
select new
{
    nconst = n.nconst,
    source = 1
}

from t in docs["Title.principals"]
select new
{
    nconst = t.nconst,
    source = 2
}

from result in results
group result by result.nconst into g
let namecount = g.Count(x => x.source == 1)
let titlecount = g.Count(x => x.source == 2)
where namecount > 0 && titlecount > 0
from repeated in Enumerable.Repeat(new { nconst = g.Key, source =
    2 }, namecount > titlecount ? titlecount : namecount)
select repeated
```