DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

A Comparative Study of Schemaless Storage on SQL Databases

A Comparison on Performance

DANIEL LINDFORS

MARTIN LIND OROS

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

TRITA TRITA-EECS-EX-2018:58

Sammanfattning

Det finns många anledningar till varför det är användbart att lagra data på ett schemalöst sätt. Anpassningsbara fält för användaren och oenhetliga datatyper är bra orsaker till att använda ett schemalöst tillvägagångssätt. De vanligaste databaserna som tillhandahåller schemalöshet är de så kallade NoSQL-databaserna. Trots de positiva egenskaperna som NoSQL-databaserna för med sig så finns det fortfarande incitament till att behålla relationsdatabaserna, mycket på grund av att de kan uppnå både prestanda och konsistens. I den här studien undersöker vi möjligheterna att använda en schemalös modell i en relationsdatabas (MySQL) för att uppnå flexibilitet och samtidigt behålla fördelarna med transaktioner (ACID) som relationsdatabaser erbjuder. De här teknikerna används sedan för att jämföras med varandra med avseende på prestanda. Yahoo! Cloud Serving Benchmark, som blivit något av en industristandard, används för att utföra mätningarna. Metoderna, å andra sidan, som använts för att uppnå schemalöshet har samlats genom litteraturstudier och resonemang. Prestandamätningarna kunde enbart fastställa en signifikant skillnad mellan metoderna för READ operationen.

Nyckelord: schemaless, database, SQL, comparison, performance, YCSB, benchmark

Abstract

There are many reasons to store data in a schemaless manner. Custom database fields and non-uniform data types are both good reasons to use a schemaless approach. The most common databases which attain schemaless properties are the non-relational, i.e. NoSQL databases. However, there still exists motives to rely on a relational database, rather than using a non-relational database, due to its ability to achieve both performance and consistency. In this study we investigate the possibilities to deploy a schemaless model within the relational database MySQL with purpose to achieve flexibility and still keep the benefits of ACID transactions. These techniques are then compared to each other with regard to performance. The industry standard Yahoo Cloud Serving Benchmark has been used to gather the results from the performance measurements. The methods, on the other hand, used to achieve schemaless abilities has been gathered through literature studies and reasoning. The performance measurements could only show that there was a significant difference between methods for the READ operation.

Keywords: schemaless, database, SQL, comparison, performance, YCSB, benchmark

Contents

Sammanfattning **3**

Abstract **4**

Contents **5**

Glossary **7**

Acronyms **8**

List of Figures **8**

List of Tables **9**

Listings **9**

1 Introduction **11**
1.1 Background . . . 11
1.2 Problem . . . 12
1.3 Purpose . . . 12
1.4 Goal . . . 12
1.5 Methodology . . . 13
1.6 Delimitations . . . 13
1.7 Overview . . . 13

2 Theoretical Background **14**
2.1 Schemaless Data . . . 14
2.2 Schemaless Data Storage Methods . . . 15
2.3 Performance . . . 17
2.4 Containerization . . . 19
2.5 Related Work . . . 20

3 Methodology **22**

4 Design **24**
4.1 Platform and method selection . . . 24
4.2 Result requirements and MSA . . . 26

5 Implementation **28**
5.1 Yahoo! Cloud Serving Benchmark (YCSB) . . . 28

5.2 Docker 30
5.3 Benchmark 31

6 Testing **33**

7 Analysis and comparison **36**

8 Results **37**

9 Discussion and Conclusion **41**

Bibliography **43**

A Datamodels and SQL code **45**
A.1 JSON 45
A.2 EAV-table 46
A.3 OSC 47

B Data **49**

Glossary

Indexing An index is a copy of selected columns of data from a table that can be searched very efficiently. 25

Java A programming language and a virtual machine. 19, 28, 29

MySQL An open source relational database management system. 13, 16, 17, 20, 21, 25, 29–31, 34

prepared statement Also known as parametrized statements, involves storing queries for efficient execution multiple times with parameters. 28

schemaless A concept related to data which can be viewed as the opposite of restricting data to a schema or a data model. 5, 9, 11–17, 21–23, 28–30, 38, 41, 42

Sharding A method for storing data across multiple machines. 13, 16, 25

Acronyms

CSV Comma-Separated Values. 31, 32

DBMS Database Management System. 16, 23, 25

EAV Entity-Attribute-Value. 16, 25, 28, 29, 39, 41

JDBC Java Database Connectivity. 28, 29

JSON JavaScript Object Notation. 14–16, 21, 25, 28–31, 39, 41

KV Key-Value. 15, 16, 21, 25

NoSQL Not Only SQL. 11, 12, 15, 16, 20, 21, 42

OSC Online Schema Change. 17, 25, 28, 29, 39, 41

POM Project Object Model. 29

RDBMS Relational Database Management System. 21

SQL Structured Query Language. 12, 13, 16, 21, 28, 29, 42

XML eXtensible Markup Language. 15, 16

YAML Yet Another Markup Language. 16

YCSB Yahoo! Cloud Serving Benchmark. 5, 19, 20, 24, 25, 28–31, 34, 36, 37

List of Figures

2.1 Database viewed as a queueing system 17

8.1 Average latencies and standard deviation for all operations used with different schemaless methods and workloads, executed on both the main and validation test machine 38

List of Tables

6.1 Technical details for the main test machine 34
6.2 Technical details for the validation test machine 34
6.3 The different workloads and their properties 34

8.1 Main T-test 39
8.2 Validation T-test 39
8.3 average RAM and CPU usage for the main test 40
8.4 average RAM and CPU usage for the validation test 40

A.1 Data model for the JSON method 45
A.2 Data model for the EAV-table method 46
A.3 Data model for the OSC method 47

B.1 Latencies from main test 49
B.2 Latencies from validation test 50
B.3 Throughput of overall performance from main test 51
B.4 Throughput of overall performance from validation test 51

Listings

5.1 The docker commands used to start the containers 30
5.2 An example of the docker commands used to run a YCSB test 31
A.1 JSON Table creation 45
A.2 JSON Insert operation 45
A.3 JSON Update operation 46
A.4 JSON Read operation 46
A.5 JSON Delete operation 46
A.6 EAV Table creation 46
A.7 EAV Insert operation 46

A.8 EAV Update operation 47
A.9 EAV Read operation 47
A.10 EAV Delete operation 47
A.11 OSC Table creation 47
A.12 OSC Insert operation 47
A.13 OSC Update operation 47
A.14 OSC Read operation 47
A.15 OSC Delete operation 48

Chapter 1

Introduction

1.1 Background

Databases have been an important technology within IT for a long time. When relational databases were introduced in the 70s it soon became a standard that is still the most widespread technology for storage and handling of data. By storing data in tables and linking them to each other by relations, both performance and data consistency is achieved. One of the most important steps when it comes to using relational databases is the data modeling. This is a process where all data requirements are taken in to produce a data model, also called schema, which will be the template used for storing data. One drawback of relational databases is that they are not built to handle changes to the data model that easily. The principle is to decide on the data requirement, implement it and not change it while it is used.

In the modern age, requirements of more flexibility has lead to development of different kinds of techniques and technologies that are based on a non-relational way of storing data. Some of these are a whole new kind of databases that go under the common banner Not Only SQL (NoSQL). Among other things, these databases trade data consistency for flexible data models. The kinds of requirements that favour this tradeoff may be the requirements of an e-commerce site where new kinds of products constantly requires changes to the data model, projects where the data requirements are not fully known at the start, or applications in need of a way to let the users define their own data fields.

Trying to meet these kinds of requirements with a traditional relational database is neither easy, nor straightforward. It requires a schemaless approach in contrast to a relational approach that is based around a fixed schema. One solution is to adopt one of the many NoSQL databases that exists instead.

Even though a schemaless approach is not straightforward when it comes to relational databases it is not impossible. These mentioned kinds of requirements are not new and the techniques to meet them has been applied long before NoSQL became popular. In fact, most NoSQL databases are directly based on some of these techniques. Because relational databases still is the most widespread and used technology for storing data it is easy to see an incentive of trying to stick with it in projects. Exactly what schemaless solutions to choose and how these compare to the NoSQL alternative is a problem on its own.

1.2 Problem

The need for a schemaless way of storing data can be problematic, especially if this need was not foreseen and an Structured Query Language (SQL) database has already been settled on in a project. Such projects have lots of different choices. One of the parameters that often dictates when choosing between different data storage solutions is performance.

The questions posed by this study are the following:

- What are some of the methods available for achieving schemaless data storage on traditional relational databases?
- How do these methods compare to each other and to a typical SQL solution when it comes to performance?

1.3 Purpose

The purpose behind the study is to present some insights into the problems faced when schemaless data is a requirement and the different solutions to this problem. Moreover, it is to directly present facts that can be of use when choosing between different schemaless methods and provide a basis or complement to deeper or broader studies into the same subject.

1.3.1 Benefits, Ethics and Sustainability

This study can both promote and discourage the adoption of schemaless SQL-methods or the NoSQL counterpart. A discussion about the potential implications of choosing either one of them is therefore appropriate.

The sustainability aspect of this revolves around energy consumption and hardware utilization. One key feature of NoSQL databases is horizontal scalability (scaling by adding more host machines). Most relational databases are hard to scale horizontally and are therefore often stuck with vertical scaling (scaling by upgrading the host machines hardware). How these two scaling methods can influence energy consumption and utilization of hardware is out of the scope of this study, but is nonetheless an important aspect of it.

Another aspect centers around the fact that SQL still is the most frequently used and adopted technology for storing data. Having to adopt a NoSQL technology to meet new data storage needs can result in large costs related to educating, hiring and analyzing the market to find the best solution. If it is possible to meet the needs with an already used SQL database much of this cost can be eliminated.

1.4 Goal

The first goal of the project is to shed some light on the different ways one can store data in a schemaless manner in a relational database. The second goal is to deliver performance measurements that will be used to analyse the differences between a subset of these schemaless methods. Moreover the hope is that this comparison will lead to a conclusion with insightful recommendations as to when and how the different methods studied should be used.

1.5 Methodology

The study is centered around evaluating different objects by comparing their performance. Since performance is easy to quantify this will be a quantitative study [9].

The problem statement is not of the kind that can be formulated as a yes or a no question, nor does it demand a specific solution to a problem. It merely seeks to make inquiries about the state of reality. The reality must thus be presumed to be real and objectively approachable, if the result is to have any meaning. The study is thus a descriptive study and leans against a positivistic viewpoint [9].

The description of performances and the comparison between them will be made in a deductive manner based on observation. Observations will be made by setting up performance measurements of a set of chosen schemaless methods. It is arguable if this approach is to be considered an experiment instead of just observation. However, the outline of this study makes it hard to define a clear null hypothesis to verify or falsify. This is due to the variable that the study centers around multiple different schemaless methods. If the study was centered around only two classes of methods this would have been different.

Exactly what methods that will be included and how the performance measurements will be conducted will be based on a qualitative analysis of literature and related work on the subject.

1.6 Delimitations

The delimitations for this study mainly concerns certain aspects that could make the study too broad if they were examined and included. Specifically, we delimitate the study from the kind of methods that are specific for individual development tools, in order to free ourselves from such restrictions. Next, the study is based on the definition of schemaless data storage, that we have constructed from reasoning and deduction which is part of the literature study. Next, we will delimitate the scope by using MySQL for its popularity and wide usage [17] since we consider that to be enough to give legitimate answers. We do not go into horizontal scalability of databases (Sharding, clustering, replication) and we do not evaluate methods with transactions that demands operations other than SQL queries.

1.7 Overview

In **chapter 2** we go through some theoretical background related to the two key components of this study: performance and schemaless. Some related work into these subjects, both individually and combined, is presented and discussed in relation to this study.

In **chapter 3** we present the strategy and foundation on which this study and its performance measurements will be conducted upon. First, by overview of the different steps, then by presenting the research methodology and lastly by going through the different practical methods that will be used and the choices that have to be made.

In **chapter 4-7** the actual designing, implementation and execution of the testing is described. This is followed up by presenting and analysing of the results in **chapter 8**. Lastly the conclusions could be drawn and a general discussion about the validity of the study is done in **chapter 9**.

Chapter 2

Theoretical Background

2.1 Schemaless Data

Before defining what schemaless data is, it is good to know what a schema actually is. A schema formally defines the way in which data is to be stored and places integrity constraint on the data. It specifies how the system should enforce data integrity while at the same time specifying for the user how the data is to be used [20]. Schemaless data is not that easily defined. If we assume schemaless data to be the opposite of what we just describe then we would have to conclude that schemaless data is useless data. No schema at all means the user have no way of knowing how to access the data, how to interpret it or even what the data is. Little literature exists on the subject and most places where it is used it is used in a very informal way. The best description found is done by Martin Fowler [8], and many other sources refer to his standpoint. This standpoint is based on the notion that what we mean by schemaless data is just the allocation of schema enforcement responsibility from the database to the user/application that uses the database. The argument is, that for any data that is useful, there at least exists an implicit schema enforced by the user itself. If we elaborate on this thought we can see that there are two elements we need to know to decide if something is schemaless:

- What data we refer to
- Who the user is

Schemaless is not a global absolute property. It is relative to what user we are referring to. Even if data in a database has a schema we can still claim that the data is schemaless from the administrator's point of view because the admin has the power to change the schema. An interesting thing to note is that even though the database enforces a schema, it may still be possible for the user to store schemaless data. This could be done by storing JavaScript Object Notation (JSON) in a text field, effectively nestling one schema inside another [19]. The top level schema may then be enforced while the nested schema is implicit. This leads to two possible ways to implement schemaless data on top of a relational database:

- By giving up the responsibility and privileges of the enforcing entity to the user. Examples of this in the context of a relational database would be to let the user use ALTER TABLE and CREATE TABLE statements.
- By creating a schemaless structure which the user controls on top of the already existing schema. Examples of this is the notorious Entity-Attribute-Value (EAV) model or storing JSON formated text in the database.

The main property one seeks in schemaless data is the flexibility that comes from being able to alter the schema. Designing a schema can be a rather tedious and complex task which requires time and skilled resources. Schemaless databases also reduces this activity. In fact, there are many occasions when a schemaless approach might be useful. Especially when there is a need to:

- Store unstructured data, e.g. social media posts and multimedia.
- Work with data without having to define a schema upfront.
- Eliminate much of the tedious work of designing schemas.
- Enable easy evolution of data formats.
- Facilitate quick integration of data from different sources.

With this being stated it is not recommended to discard the relational model. It depends heavily on the use-case and the type of data that needs to be stored.

2.2 Schemaless Data Storage Methods

2.2.1 NoSQL

The techniques for storing data using NoSQL differs from the traditional relational techniques. Relationships between entities in tables are essential in relational databases. NoSQL is the common name for databases not using this technique, or perhaps only partially using it. NoSQL databases are designed for scaling by using distributed clusters of low-cost hardware to increase throughput without increasing latency. There are four main NoSQL storage methods.

2.2.1.1 Document store

A document store utilizes a document data model where each record and its associated data is thought of as a "document"[6]. It is designed to store semi-structured data, typically in JSON or eXtensible Markup Language (XML) format. One advantage of this type of storing method is that the schema for each document can vary and easily be changed, which makes the architecture more flexible in organizing and storing data [21]. Another advantage of the document store is that it offer great performance and horizontal scaling options.

However, with the flexibility of document stores which lacks the relational restrictions, vulnerabilities of accidents arise since a programmer easily can put any type of data into any collection of documents. Nayak et al [11] claims that document stores should be avoided if there is a need of a lot of relations and normalization. The typical use cases are content management systems, blog software etc.

2.2.1.2 Key-value Store

This sort of data storage stores pairs of keys and values in a schemaless manner. It can be seen as an associative array or more commonly as a hash. From the hash a value can be retrieved when the corresponding key is known. The simplicity of resource-efficient Key-Value (KV) stores can be applied in embedded systems or as high performance in-process databases [6].

2.2.1.3 Column store

Instead of storing data in rows like a relational database does, these databases store data in columns. One effect of this is that data compression is more efficient because data belonging to the same type are stored next to each other in memory. Some operations also becomes much faster while others become slower. It also offers high scalability in data storage [6].

2.2.1.4 Graph Database

These kind of databases has an explicit graph structure and a powerful data model based on nodes and relationships where each node knows its adjacent nodes. Using this technique, millions of records can be traversed. As the number of nodes increases, the cost of a local step (or hop) remains the same. It also has indices for lookups.

A graph database such as Neo4J becomes useful for highly connected data (e.g. social networks) recommendations (e.g. e-commerce) and path finding. Generally, graph databases are easy to query [6]. On the other hand, graph databases are not suitable for Sharding and they are difficult to cluster. Apart from graph databases that are not as common as the rest, most NoSQL databases facilitates some kind of schemaless storage.

2.2.2 SQL Based Methods

The different schemaless data storage methods that can be applied on top of a relational database can be classified into four categories:

2.2.2.1 Semi-structured Data

By storing data in a format such as JSON, XML or Yet Another Markup Language (YAML) schemaless data is achieved. These structures can then be stored in the relational database as text strings, BLOB or any specific data type that is meant for the format. Support for JSON was added to MySQL since version 5.7.8 [19]. This adds syntax to the SQL that handles JSON data. Varying levels of support for manipulating and handling these formats exists in different Database Management Systems (DBMSs).

2.2.2.2 Schemaless Data Model

By implementing a data model such as the KV model or Entity-Attribute-Value (EAV) model, data can be stored in a schemaless manner in the relational database directly. A Key-value table is essentially just a hash table that associate a key with a specific value. By using different kinds of naming conventions for the key it is possible to achieve different kinds of data models. Storing a table could for instance use keys in the format "entity-attribute-value" to store tabular data. EAV is another concept that does exactly this, but stores the three values in either separate tables that are related to each other, or in one single table with one column for each. This setup also makes it possible to let different records have different fields associated with them. This is done without using more storage than is needed, compared to a traditional relational table where all records must have exactly the same fields.

2.2.2.3 Online Schema Change

By using Alter-table and Create table statements as a part of the runtime operation of the database, fields can be dynamically added and removed, essentially resulting in what

could be considered schemaless use of the database. These kinds of operations has been, and still is in some relational databases, a slow process. Adding a new column to a table will in most cases result in a new table being created with the new column, and all data from the previous table copied over to the new one. This process can block any concurrent queries to the database. For use cases with large tables this can be acceptable. Since version 5.6, MySQL introduced DDL (Data definition language) which makes many Online Schema Change (OSC) operations non blocking[1]. Adding a new column can thus be done concurrently as long as the column is not an auto-increment column, and thus making its use a more viable option for schemaless operation of the data. This is however a risky option, and the operations should be implemented with care and tested to avoid potential data loss.

2.2.2.4 Extra Fields

By creating a set of unused extra fields in the table new fields can, from the users point of view, be added by essentially just to start use these unused fields. This method needs an additional table and coding to be able to name the fields, But is essentially a viable ad-hoc solution to the problem. The downside is that the number of fields that will be necessary in the lifespan of the database must be estimated. These extra fields will also take up extra space and can be considered wasteful. Apart from this, the data type has to be pre-defined.

2.3 Performance

Depending on the purpose of a system, different kinds of metrics becomes important when talking about performance. The two main purposes of databases are to store and distribute data. Storage performance can be measured by how much space a certain amount of data takes up in the system. The type of performance this study is concerned with however, is the time aspects of serving clients. This chapter first introduce some theoretical concepts behind performance when it comes to systems like databases. Thereafter some common types of performance measuring standards are presented.

2.3.1 Measurements and Testing

A database system serving clients can be viewed as a queueing system: Queries comes in and a response goes out. This means queueing theory can be applied when studying the performance of databases.

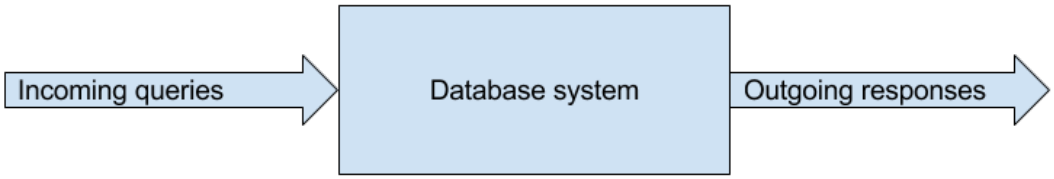

Figure 2.1: Database viewed as a queueing system

[1]See the reference manual for more details: `https://dev.mysql.com/doc/refman/5.6/en/innodb-create-index-overview.html`

In such systems the latency, throughput and utilization are some amongst other common metrics used to measure performance. A set of fundamental laws exist that describes the various performance aspects of queueing systems.

Performance refers to to the way that a computer system behaves in response to a particular workload. It can be measured in terms of system response time, throughput, and resource utilization amongst other metrics. These are all well known metrics belonging to the study of queueing network models. A set of fundamental law's exists that describes the various performance aspects of these kind of systems. We will outline and describe the fundamentals of this here.

When it comes to queueing theory we can express the occupancy in a system with the terms of latency and throughput [22]:

$$\text{Occupancy} = \text{Latency} \times \text{Throughput} \tag{2.1}$$

In equation 2.1 the occupancy is the number of requestors in the system (number of queries). Latency is the time it takes for one query to go from one end to the other end of the system. Throughput is defined as how many queries entering or leaving the system per time unit on average. This relationship can be used to understand peak throughput of a database. Max throughput is limited by the latency in the system and the maximum number of requests that can be serviced concurrently.

Performance tests that utilizes these metrics can be put into three categories based on the function and structure of the database:

Structural testing involves the testing of tables and columns, schema, stored procedures and views. Basically, it involves testing of the components that are not exposed to the end users, meaning the repository for storing data

Functional testing is considered from the users point of view and if the requirements for the transactions and operations that users perform are met. This is usually done by black box testing or white box testing [2]. White box testing methods are based on an analysis of the internal structure of the system while black box testing is a method used when the internal structure is not known to the tester

Nonfunctional testing is centered around load-testing, risk testing, stress testing and is primarily concerned with performance of a system. The transactions in a database system usually have some impact regarding the performance. The primary target of load-testing is to study how vast this impact is and the metrics usually studied are:

- The response time for executing the transactions for multiple remote users
- Time taken by the database to fetch specific records.

Benchmark testing can be used when the database is free from data problems and bugs, in order to study the system performance. This is also the primary focus of this study. The parameters usually studied are:

- System level performance
- Identify most-likely-used functions/features
- Timing - maximum time, minimum time and average time to perform functions
- Access volume

Based on pre-defined synthetic workloads, often generated by a benchmark tool, it is possible to evaluate the performance of a system, given the benchmark results. Benchmarks are often used in comparative analyses of products and systems and they are also used as monitoring and diagnostic tools [1].

According to [3] benchmarking is the best method when multiple database systems needs to be evaluated and compared to each other. However, as previously mentioned, the systems must be fully operational and bugfree in order for the benchmark to be suitable.

The non-profit Transactional Processing Performance Council (TPC) provides benchmarks that measure transaction processing and database performance. Typically, the intent of the benchmarks for relational databases is to measure how many transactions a given system and database can perform per unit of time, e.g. transactions per second. Today, the active TPC benchmarks [16] include several standards for measuring the performance in different circumstances, e.g TPC-C which is one of the most frequently used benchmarks and involves on-line transaction processing.

Since it is getting more common to use non-relational databases there is also a need of benchmark standards for these kind of databases. Yahoo! Cloud Serving Benchmark (YCSB) is an open source micro-benchmark tool developed by Yahoo! Research [4]. An important aspect of YCSB is the fact that it is generic and extensible in terms of implementing new workloads and interfaces for every kind of database there is. This makes it easy to benchmark and compare new systems.

YCSB comes with a core set of workloads, which each represents a particular mix of read/write operations and request distribution. Drivers for a diverse range of databases also exists in the YCSB repository on Github. Each driver implements its own interface that YCSB will use to interact with the database. To make YCSB compatible with a new database one simply inherits an abstract class and implement its methods. The framework also makes it easy to define new workloads, either by adjusting the parameters of an existing workload, or by defining a new workload by writing Java code.

The default core workloads consist of:

- Workload A is Update heavy: 50% Read, 50% Update
- Workload B is Read heavy: 95% Read, 5% Update
- Workload C is Read only: 100% Read
- Workload D is Read latest: 95% Read, 5% Insert
- Workload E is Short ranges: 95% Scan, 5% Insert
- Workload F is Read-Modify-Write: 50% Read, 50% Read-Modify-Write

Apart from these, YCSB also measures the performance of the loading of the database. This process loads the database with initial data that will be used when running a workload.

2.4 Containerization

Reproducibility is an important part of research in general, and running experiments on software is no different in this regard. Challenges faced when dealing with reproducibility in computational science involves:

- Problems with dependency. Building and installing the nessesary software fails due to missing dependencies.

- Imprecise documentation. Often it is impossible to reproduce an experiment due to insufficient documentation.
- Code rot. Parts of the software changes over time, which can alter the result of the system.
- Barriers to adoption and reuse in existing solutions.

Docker is a containerization technology that help dealing with these problems[13].

The OS feature known as Containerization or Operating-system-level virtualization makes the kernel allowing for the existence of several isolated user-space[2] instances. When running, these instances are called containers. The programs running inside such containers can only see the container's contents and devices assigned to it. This makes it possible to run programs within containers to which only parts of the system's resources are allocated. Docker is an open-source project which rely on the previous mentioned technology and is a possible solution for the challenges of reproducibility. A Docker container is based of what is called a `Dockerfile`. It is basically a plain text file (without extension, such as .txt) which describes how to build up a runtime environment, and provides a description of necessary software dependencies, environmental variables and so forth. This file is later compiled into an image file, which can be run as a container. Some examples of the most essential commands in a `Dockerfile` are:

FROM <image> at the start of the `Dockerfile`. Specifies what image to use as a base for the image to be built

RUN <command> Executes a command in the default shell of the image

ADD/COPY <sourcDir> <targetDir> adds files and/or directories from the local file system to the image file system

CMD <command> at the end of the `Dockerfile`. Specifies what command should be run on the container when it starts

The `docker build` command is used to build an image from a specified `Dockerfile`. When run, it executes every line in the `Dockerfile` in sequence, making a preliminary image of the work so far at every step. This is a caching feature which makes later builds on edited `Dockerfile`s skip to the top most edited line, making compilation faster. For more information on Docker and technical details, see `docs.docker.com`.

2.5 Related Work

A couple of studies centered around comparing the performance of different database storage strategies were found, mostly published by IEEE.

Bucur and Tudorica [5] set out to compare different NoSQL databases belonging to the Wide-column store/Column Families. From these categories they chose two of the most frequently used at the present time (i.e HBase and Cassandra). They also used MySQL as a reference element to see "what is lost and what is gained by using a NoSQL solution instead of a classic one". While the study was mostly qualitative they did compare performance between the databases by using data from another study centered around testing YCSB. The data collected showed latency for different

[2]The part of the system memory where user processes run

throughputs up to maximum throughput. These studies were focused on large scalable database systems, which is why data collected from a cloud service benchmark was fitting. Also studying how latency varies for different throughputs makes sense in such a setup.

Li and Manoharan [10] compared different KV stores with a Microsoft SQL (MS SQL) database that were also used as a KV store. If this was done on a single machine or multiple nodes is not mentioned. The only measured metric was the average time for completing a varying number of operations. This would suggest that the test was conducted on a single node at maximum throughput. The operations that were tested were the fundamental CRUD (Create, Read, Update, Delete) operations. The dataset this was tested on was randomly generated KV pairs.

Aboutarabi et al [12] compared the performance of mongoDB, a document database, and MS SQL used for e-commerce data. The dataset and schema were chosen to mimic typical data found in an e-commerce database. The data model was normalized to N3. The concept of relations and join operations were translated to the mongoDB also by implementing the joins on the client side. They too tested the fundamental CRUD operations and measured the time taken to perform varying numbers of operations. Three considerations were noted in the study:

- The structural differences between the databases must be taken into account such that the databases are tested in the way they were meant to be used.
- To be consistent with reality the databases should be tested in a as close to real use as possible.
- A common interface should be used for all databases to eliminate faulty performance differences related to different overheads.

ARGO is an automated mapping layer for storing and querying JSON data in a relational system. Chasseur et al [7] investigated whether the advantages of the JSON data model could be added to Relational Database Management Systems (RDBMSs) and also seeked to gain some of the traditional benefits of relational systems in the bargain. Additionaly, they evaluated and compared the performance between MySQL, PostgreeSQL with ARGO on top, and the NoSQL database MongoDB. As benchmark tool they used NoBench. The result showed that it is possible to combine the flexibility of JSON with the query processing and transactional properties that are offered in relational databases. They also found that ARGOs performance on MySQL is high enough to be a very compelling alternative to MongoDB.

In all of these studies at least one NoSQL database was tested which often falls into the category of schemaless. Furthermore, the studies centered more or less around the performance aspects of the databases and how they compared to each other. Chausseur et al [7] made a study that are similar to this one in the sense of extending the relational database with schemaless properties and compare the performance to a NoSQL database. Our studies are, however, oriented in testing and comparing the performance on schemaless-extended relational databases.

Chapter 3

Methodology

The act of benchmarking falls into the category of experimental research strategy. If the benchmark is set up in a similar way to how some other study has done theirs, there is also a possibility to include their data ex post facto into the total dataset. The purpose of this benchmark is to make a comparison of different alternatives, which makes this a comparative study. A scientific methodology when it comes to comparative studies is not well defined. Depending on what the comparison is about and what kind of data is compared different methods can be used. In this study the comparison will mostly be centered around when and how the different methods perform better than each other. This makes the study very descriptive. If anything interesting can be deduced from the data is hard to say beforehand.

The experimental research strategy provides guidelines for the implementation of the research. These guidelines cover organization, planning, design and implementation of the research. The control of all the factors that possibly can affect the result of an experiment is included in the experimental research method. This is a method often used on large datasets. The data collection methods in quantitative research are commonly experiments, questionnaire case studies and observations, according to [9].

The purpose of experiments is to collect a large data set for variables. In questionnaires the purpose is to collect data through questions. In quantitative research it is common that these are closed questions (yes/no) compared to qualitative questions that are open and reviewing. Due to the experimental nature of the study a quantitative method for data collection is used, which is an experiment. Accordingly, the choice of using qualitative methods and using questionnaires or interviews as data collection methods is not considered.

The methods used for data analysis concerns inspection, sorting, cleaning, transforming and modeling of the data. The analysis supports the conclusions. As an example conclusions can be drawn with the help of statistics by evaluating the significance of the results. Regarding this study the analysis method for comparing performance is statistics since it is useful in quantitative, experimental research. Lastly, the quality assurance of the research material can be achieved with validity, reliability, replicability and ethics. These methods to validate and verify material is used in quantitative research with a deductive approach. By making sure that the test instruments are measuring what they are supposed to measure, validity is achieved. Reliability can be reached by stabilizing the measurements and make sure every test result is consistent. Replicability or repeatability is the capability for another researcher to reach the same results. Therefore, it is necessary to describe the procedures in detail.

In the following sections the research procedures are described. To compare the performance of the schemaless methods benchmark testing will be used to gather data that

can be compared. The outline that will be used for the study is designing, implementing, testing and analyzing.

Designing

The result from the designing should be a testing plan. Designing the benchmark testing involves making a series of various decisions based on questions like:

- Which schemaless methods should be tested and compared?
- Which metrics should be used to quantify performance?
- What operations on the schemaless storage should be tested?
- What optimizations should be used when implementing the methods and configuring the database?
- What kind of use case should the dataset and workload represent?
- What DBMS, benchmarking platform, OS, etc. should be used?
- How is the result validated and reproducibility guaranteed?
- How should the test be conducted to give accurate enough data?

Implementing

The implementation consists of setting up the platform and implement the different components needed to perform the test. Some components might be implemented by configuring the testing tool or coded by hand if a tool is not used.

Testing

The testing involves using the implementation to execute a testing plan. The data from the test should be captured, processed and stored in a way, easily handled when analysing starts.

Analysing

The analyzing phase is about making statistical calculations on the data, visually presenting it and interpreting it to try to describe what it means. Statistical significance testing is a common way to estimate how certainly a result can be stated as a fact.

Chapter 4

Design

In this chapter we pressent the overall design choices made and what they were based on. Mainly, this involves answering the series of design questions posed in chapter 3.

4.1 Platform and method selection

4.1.1 Benchmarking platform

The choice of benchmarking tool was YCSB mainly because it is the most widely used tool for benchmarking non-relational databases [15] and was simple and flexible for the authors. By using the YCSB benchmarking tool, some of the questions regarding design automatically gets answered. Specifically, the design requirements regarding the metrics, operations and use cases are predefined by YCSB.

The implication of using YCSB is that only INSERT, UPDATE and READ operations will be tested. YCSB does not support measurements of the delete operation as of writing this. The core workloads mentioned in 2.3.1 will be used except, Workload E. What this workload measures is an operation called SCAN. This operation returns all records in a key range. The reason for omitting it is because of ambiguities and complexity in the ways it could be implemented in for one of the chosen schemaless methods. This is discussed in greater detail at the end of 5.1.

Choosing YCSB also decides what kind of data to and use-case to used. YCSB measures performance of a set of workloads, run against a large table. For the core workloads provided by YCSB, this table contains 10 fields, each holding a 100 byte string. At the beginning of a test, the table contains 1000 records. One of the fields is used as a key to reference all records. All CRUD-operations are done record wise. Each workload will run 1000 operations against the database. We choose to use these default parameters because of simplicity and higher degree of comparability with results from other studies using YCSB.

YCSB does not care how each database driver implements the CRUD-operations or how it stores this table. This large table becomes the implicit data model for our schemaless methods to store.

YCSB measures the average latency for each kind of operation in the workload. A total run time and average throughput is also given for the whole run of the workload. Average latency for each method-workload-operation combination will be the main metric to gather.

4.1.2 Method selection

The selection of methods to test is supposed to at least reflect one method from each category mentioned in section 2.2.2. But because no testing of adding and removing fields is done, the OSC and extra fields approach is almost identical. Also, the way in which the JSON, KV and EAV-table is adapted to work with YCSB makes the line between them blurry. In other words, storing JSON under a key could also be considered a KV storage. And using a KV storage with two fields associated to each key is essentially an EAV-table. Hence three methods were chosen:

JSON: Data in a JSON format stored inside a field

OSC: A normal relational table that could be modified through alter-table statements

EAV-table: A table with 3 fields, representing entity, attribute and value

A side effect of OSC being just a normal table is that, from the perspective of the benchmark, it is identical to just storing data the regular way in a relational database. YCSB does not measure performance of altering the schema. Thus, no alter-table statements will be part of the test.

4.1.3 DBMS selection

MySQL was chosen as the DBMS to implement the methods on, because of its widespread use[17], but also because it supports JSON data types, as of version 5.7.8[1]. Linux as the operating system was chosen because of its low overhead and flexibility.

4.1.4 Optimizations

There always exists ways of tweaking a database system to optimize it for different purposes. This adds extra choices that must be made. Some optimizations might be in favour of different storing methods. For the comparison to be valid, it is important that optimizations are done in a way that does not give either of the methods an unfair advantage over the others. Optimizations could either be done separately for each methods so that we get the maximum performance possible from each of them, or we could apply the exact same kinds of optimizations to all of them. Three kinds of optimizations were considered:

- Caching
- Indexing
- Sharding

No caching other than what occurs in the system by default should be used. Indexing is done for the key field in all the methods. Sharding would be interesting for testing scalability, but is dropped in favour of just focusing on the performance on a single noded system.

[1]See the reference manual for more details: `https://dev.mysql.com/doc/refman/5.7/en/json.html`

4.2 Result requirements and MSA

4.2.1 Result requirements

In order to collect accurate results we want the measurements to be as unbiased and precise as possible. In addition, end result of the test should be repeatable and, to some degree, reproducible. All of these requirements are typical in Measurement System Analysis (MSA)[18]. The requirements of the result of the measurement system can be summed up in a couple of points:

- The system must be able to detect differences in performance between tested methods.
- These differences should fall inside some confidence intervall with an appropriate significance level.
- The result should at least be reproducible in the sense that the list of methods, sorted by performance, should remain constant over different experiments. At best the difference in performance should be proportionally the same.

With these requirements stated some delimitations and assumptions follows naturally. The absolute measurements of performance are of no interest, only the relative measurement and that the relative performance stays consistent. This fact leads to the assumption that the test will be more or less independent of the underlying hardware and OS.

4.2.2 Precision

Precision is often separated into repeatability and reproducibility in MSA. Repeatability is here the ability to do the same exact measurement twice and get the same result. Difference in result means there exist some random element in the measuring system. The basic property measured in this case is the time taken to perform a query. The only part of this process this study is interested in is the part that will differ between methods. A large portion of the total time will, however, be due to overhead in the system that does not differ substantially across different methods. Examples of this is reading from hard drive, tcp communication, context switching. Most of the error in the result is expected to come from randomness in latency from these other parts of the process. Two strategies can be used to minimize the error due to this:

- Minimize overhead
- Use large enough sample size

Example of minimizing overhead is to put the workload generator and database on the same machine to minimize time taken to send a query. Using larger sample size will not decrease variance of latency for individual operations, but will decrease variance of the average throughput, which is what is taken as a result.

4.2.3 Reproducibility

By reproducibility we refer to the ability to setup multiple identical experiments that yields the same result. Depending on the result requirement the necessary steps to achieve this can range from demanding every single component in the system to be identical in every experiment, to just using the same kind of database and benchmarking tool. The assumption

is that hardware and OS won't have a large impact on the ability to reproduce relative measurements. But setting up a measuring system involves installing and configuring software. Subtle differences in versions and configuration can have large unforeseen consequences to the result of the measurement. To ensure reproducibility in this regard some kind of encapsulation technique should be used. The more deterministically the measurement system can be set up and configured the better. Containers is one such technique. It creates a separate environment that uses the resources of the host OS but is totally self contained in regards to software. Docker[13] is one such container system that runs on Linux and allows the user to specify what software should be installed and how it should be configured in a Dockerfile. An image is then created from this specification which can be run as a Docker container. This container acts like a separate host. Monitoring and allocation of hardware resources can be controlled.

Two separate tests are performed on two different systems to validate that the result follows requirements, and that reproducibility has been achieved. The choice of using docker makes redeployment very simple and is another reason as to why containers were chosen.

Chapter 5

Implementation

In this chapter, the implementation of the methods, test bed and container is explained. The actual source code can be found at `https://github.com/dlindf/schemalessMySQLBenchmarking`

5.1 YCSB

The test performed by the YCSB benchmarking tool is centered around storing and performing basic CRUD-operations on a large table. How many records, or fields, this table has is configurable. YCSB provides an abstract Java class, containing methods for the basic CRUD-operations and an additional scan operation, to be implemented by the user. Many such implementations already exists for various database drivers, of which none is directly applicable to the test of this study as of writing this. However, implementation for the Java Database Connectivity (JDBC) driver exists that implement operations for using YCSB on a standard SQL database. With some modifications this implementation can be used to implement the schemaless methods of this study. The original implementation could also be kept to represent the OSC method.

The only parts that had to be edited for the JDBC module to test our JSON and EAV implementation were some methods in two files:

DefaultDBFlavor.java which implements the actual SQL query for each operation as prepared statements

JdbcDBClient.java which implements the abstract YCSB methods representing the operations, and uses the prepared statements from DefaultDBFlavor.java

Each schemaless method had its own version of these two files. `DefaultDBFlavor.java` holds the SQL code for the method, and `JdbcDBClient.java` holds the execution and response handling of the query. Because the data, returned from the database, had some structural differences between methods, the handling of this data had to be modified for the JSON and EAV methods. One such difference was being able to handle JSON strings. The READ query returns the result as a JSON string, which has to be parsed. For YCSB to be able to time the query, the result has to be returned by the READ method in a specific format. These additional procedures are thus also counted as part of the operation itself.

The need to parse JSON also lead to the inclusion of a JSON library. To include this library, it had to be added to the JDBC modules pom.xml file. YCSB is written in Java

with Maven as the project manager, which defines all the project dependencies in Project Object Model (POM) files[1].

YCSB holds the tools for testing an already existing database, but does not come with a way of initialize the database. Because each schemaless method is based on different data models, the database had to be initialized differently for each of them. This was done by implementing a short Java program called InitDB, using the JDBC package. The data models used for each schemaless method all used a single table, containing the following columns:

OSC VARCHAR(255) ycsb_key (PK), VARCHAR(255) field0, VARCHAR(255) field1, ..., VARCHAR(255) field9

JSON VARCHAR(255) ycsb_key (PK), JSON fields

EAV VARCHAR(255) ycsb_key (CK), VARCHAR(255) field (CK), VARCHAR(255) value

PK and CK stands for Primary Key and Composite Key respectively. When using these models, both the READ and DELETE operation can be written identical in SQL:

READ `SELECT * FROM usertable WHERE ycsb_key = ?`

DELETE `DELETE FROM usertable WHERE ycsb_key = ?`

For the other operations, the query varied. For all but one query an intuitive straight forward way of formulating the query could be found. The exception was for the UPDATE operation of the EAV method. Here, a couple of alternatives were found. In the end, the choice was made based on ease of writing and reading:

```
UPDATE usertable SET VALUE =
        CASE field
        WHEN 'field0' THEN ?
        WHEN 'field1' THEN ?
        ...
        WHEN 'field9' THEN ?
        END WHERE field IN('field0', 'field1', ..., 'field9')
        AND ycsb_key = ?)
```

In appendix chapter A the MySQL tables and SQL queries implemented are shown in greater details for all methods.

Apart from the normal CRUD-operations, YCSB also comes with an operation called SCAN. This operation takes a range of keys and should return all records in that range. This operation was not implemented as mentioned in 4.1.1. The SCAN operation was already implemented for the OSC method and would have been easy to implement for the JSON method. However, for the EAV method there were no obvious ways of implementing it. The choices ranged from either using an overly complex SQL query and less result handling in Java, to a very simple query and a more complex, iterating result handling algorithm in Java. In the end the choice not to use the SCAN operation was made.

[1]The raw form of the JDBC module explained here can be viewed at `https://github.com/brianfrankcooper/YCSB/tree/master/jdbc`

5.2 Docker

To make the benchmark as reproducible and flexible as possible, Docker containers were used [14]. This automates the whole process of setting up an enclosed environment and makes sure the exact same dependencies are installed independent of the host environment.

Two containers, one for MySQL and one for the YCSB benchmark, were to be used. The MySQL container was pulled directly from an official MySQL repository at Docker Hub[2]. No changes were made apart from specifying the MySQL version 8.0.1. This was necessary to be able to use the latest functionality built into the JSON datatype.

As for the YCSB container, no already existing image of any use was found on Docker Hub, so a new one had to be created. This is done by writing a `Dockerfile`, specifying all the steps to install and configure software and files on the container. The `Dockerfile` is later used to build an image that can be run as a container. The outline of the `Dockerfile` as follows:

import base image In this case Debian Jessie was used as a base for the image

install packages Needed packages like JDK, Python and Maven are installed with APT

fetch YCSB downloads YCSB from Github and place it in the image file system

add files adds files and folders discussed in section 5.1 to the container

compile ycsb use Maven to compile YCSB with the additional files

compile InitDB compiles the InitDB program mentioned in section 5.1

clean up removes unneeded files

Most of the files used to build the YCSB images did not differ between methods. For those that did, these were put into separate directories and referenced by a parameter in the `Dockerfile`. The build command was `docker build -t name --build-arg type=name`. Option -t sets the name of the container and –build-arg sets the parameter that is used in the `Dockerfile` to reference the directory with specific files. For simplicity, the name of the container and directory with specific files were named the same for each method. These were ycsb_osc, ycsb_eav and ycsb_json.

The command used to start up both the MySQL and YCSB container is shown in 5.1:

Listing 5.1: The docker commands used to start the containers

```
#MySQL
docker run -dit --cpuset-cpus=0 --name=mysql
    --env="MYSQL_ROOT_PASSWORD=root" -p 3306:3306 mysql:8.0.1

#YCSB
docker run -dit --cpuset-cpus=1 --name=ycsb --link mysql:mysql $1
```

The last argument in a `docker run` command is always the name of the image to run as a container. For convenience, the `docker run` commands were put in scripts. Because three different versions (one for each schemaless method) of the YCSB container were implemented, the name was parametrized. The options used does the following:

-dit runs the container detached, interactive and with a pseudo-tty

[2]The official MySQL repository at Docker Hub can be accessed at `https://hub.docker.com/_/mysql/`

–**cpuset-cpus=x** sets the cpu core which is to be used by the container

–**name=x** sets the name of the container

–**env=x** sets an environment parameter. In this case the root password for MySQL

-p x:y maps the containers port y to port x on the host system

–**link x:y** adds the ip of container x to the hosts file in the starting container under the hostname y

Normally a command that is to be executed when the container starts is specified at the end of the `Dockerfile`. However, in this case the container is started interactively to allow for execution of bash commands when the container is already running. This allows for controlling the container from the outside.

To make YCSB connect to the database running on the MySQL container, YCSB has to know the IP-address of the MySQL container. Hard coding the IP won't work because a container is assigned a new IP each time it is started. A feature Docker comes with is the ability to link containers by specifying the name of another container and a host name to associate its IP with. This is done when a container is started, and essentially results in that the IP and chosen host name of the container to link with is written into the hosts file of the started container. By making use of this feature, YCSB can use the chosen host name (in this case MySQL) instead of the actual IP of the MySQL container. This also makes it necessary that the MySQL container is already running when the YCSB container is started.

5.3 Benchmark

When both a MySQL container and a YCSB container is up and running, running tests can begin. An example of the actual commands executed to do this is shown in 5.2.

Listing 5.2: An example of the docker commands used to run a YCSB test

```
docker exec ycsb_json java InitDB

docker exec ycsb_json ./bin/ycsb load jdbc -P workload/workloada
    -P db.properties

docker exec ycsb_json ./bin/ycsb run jdbc -P workload/workloada
    -P db.properties
```

`docker exec` executes a command in the specified container, in the example above, ycsb_json. The first command initialize the database with the data model needed for the JSON method. The second command loads the database with initial data to be used in the test. The third command runs the actual test. The result is printed to `stdout`. As parameters to both load and run, the YCSB driver, workload file and property file to be used are specified.

Running this sequence multiple times for different combinations of method and workload by hand and gather all data would be a tedious job. Therefore, a couple of bash scripts were written:

benchmark input: (iterations, workload, method), output: Comma-Separated Values (CSV) file with results of all iterations

job a simple script where the series of different runs of the benchmark script are lined up, and resulting CSV file copied and renamed to a results directory

monitor used by the benchmark script. It logs the cpu, memory and network use of the containers each second

Apart from these scripts, a Python script named **aggregation.py** was created, that takes the result of every CSV file outputted by the benchmark scripts, calculates average values and standard deviation of all measurements, and adds these to a new CSV file. To make sure the test environment did not exhibit strange behaviours related to the use of hardware, a monitoring script was implemented that uses Dockers own monitoring tool `docker stats`. It was important to make sure that the tests did not exhaust available RAM. This could result in some measurements being considerably slower than others because of page misses.

Chapter 6

Testing

This chapter describes the overall testing procedure and details not mentioned in earlier chapters.

The testing procedure can algorithmically be written as follows:

```
input N, methods, workloads

results <- []

For each method in methods:
    For each workload in workloads:
        _ <- runBenchmark(method, workload)
        measurements <- []
        For i=1...N:
            measurement <- runBenchmark(method, workload)
            measurements <- append(measurements, measurement)
        results <- append(results, measurements)
return results
```

Where the runBenchmark procedure can be defined as

```
runBenchmark(method, workload) ->
    initializeDatabase(method)
    loadMeasurement <- loadWorkload(method, workload)
    runMeasurement <- runWorkload(method, workload)
    return (loadMeasurement, runMeasurement)
```

Noteworthy in the algorithm is the first use of `runBenchmark`, where the returned result is thrown away. The reason for doing this is because the first run, in a series of runs with the same method and workload, can perform slightly different from the rest, due to caching behaviours in the system. The solution to the caching problem is either to purge all caching systems between each iteration, or make sure that each iteration benefits the same way from the caching. By throwing away the first result in a series of identical tests the latter is achieved. This was a much simpler strategy than the former.

The resulting output was processed into tables with N rows and each column representing a measurement parameter.

Tests has been performed in two instances. The first on a computer with the specs shown in table 6.1. We will call this run the "Main test". For validation purposes, tests were also done on another computer. The specs for this computer is shown in table 6.2 In both instances, 40 iterations was used.

CPU	Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz
cores	8
architecture	x86_64 bit
ram memory	8 GB
OS kernel	4.9.45
OS distribution	NixOS
Docker version	17.06.2-ce

Table 6.1: Technical details for the main test machine

CPU	Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz
cores	8
architecture	x86_64
ram memory	12 GB
OS kernel	4.4.0-78-generic
OS distribution	Linux Mint 18.1 Serena
Docker version	17.05.0-ce

Table 6.2: Technical details for the validation test machine

Before the test was executed, both the MySQL and all three YCSB containers were started. The MySQL container is assigned to one CPU core and all YCSB containers to another.

The different workloads aims to simulate different kinds of uses of the database, as shown in table 6.3

Workload A	50% read, 50% write
Workload B	95% read, 5% write
Workload C	100% read
Workload D	Read most recently inserted record mostly. Around 50 inserts and 950 reads are used in this case
Workload F	Reads, modifies and writes back to the database
Load	100% inserts

Table 6.3: The different workloads and their properties

The workload `Load` in table 6.3 is not an actual workload per definition, but something that must be done before the database can be used for a test. However, YCSB is kind enough

to make performance measurements during this phase too. Analysed as the result from a workload, the `Load` phase represents a database where data is only inserted. Because the `Load` procedure differs only between methods, but not workloads, we get five times as many measurements as we need. The solution to this was to pick the median `Load` measurements for each method to represent the `Load` measurement for that specific method.

Chapter 7

Analysis and comparison

To get a grasp on what kind of information could be extracted from a measurement with the current set up a couple of test runs were made. The result from these test runs were aggregated in different ways to figure out how the data is best represented to more easily actually answer questions about differences. The aggregation step used, calculated both the mean and standard deviation of each column. The aggregation from every table was merged into an aggregation table to be studied and further processed.

The problem statement asks if there are any significant differences between the methods. Since both mean and standard deviation is returned for each target with YCSB, this makes it easy to calculate a confidence interval for the relevant metrics and then compare this between methods to see if they overlap or not. This is great for visually analysing the data in a plot and getting an intuitive understanding of the data. A more strict way of deducing how significant the differences between measurements are is through a two-sample T-test. Here we test the null-hypothesis that A-B = 0, where A and B are two measurements. The T-test gives us the significance level at which the null-hypothesis will be rejected. In other words, the probability that there is no difference between A and B if we assume there is.

Two key types of key metrics is available in the data:

1. Average Latency for a specific method-workload-operation combination
2. Average Throughput for a specific method-workload combination

The throughput deals with how effectively the workload is processed overall, while the latencies measures how a specific operation performs under a certain workload. Because the different workloads only performs some of all possible operations the total data set is kept in a manageable size.

Because two benchmark runs were made on two separate machines, all the results can be double checked to ensure reliability. While doing this it is important to look for some important signs of unreliable results. The most obvious being if they produce the same kind of significance level in the T-test. But even though the T-test yields the same result, this does not reveal the direction of the difference. for example, the T-test might claim a less than 5% significance level in both tests, But in one test we have that $A < B$ while in the other that $B > A$. This is an error that must be checked for in every comparison.

Chapter 8

Results

This is the chapter were the results of the benchmark are presented.

In figure 8.1, the average latencies for all operations under the different methods and workloads are shown for both the main and validation test runs.

Each of these charts shows the mean latency measured for a specific operation. The sample set used for each of them consists of 40 measurement runs done with the YCSB benchmarking tool. Each of these runs used 1000 initial records and performed 1000 operations. The standard deviation of the measurements is represented in the plots by an error bar.

The data from both benchmarks was used to do a two-sample t-significance test to test the null hypothesis $A - B = 0$ for the latency of each relevant method-workload-operation combination, and additionally throughput for each method-workload combination. By relevant, we mean all choices of A and B that is comparable. For instance, comparing `JSON-A-READ` with `OSC-B-INSERT` has no real meaning. To be more specific, the comparison is done on all combinations of method-workload-operation where only the method differs between A and B, because we are only interested in how the different methods compare to each other.

Table 8.1 and 8.2 shows the significance level under which the null-hypothesis would be rejected. In other words, the probability of the null-hypothesis being true if we reject it.

Apart from just showing the significance level, some features of the data when comparing the main test and validation test is marked by different cell colors. Marked in red are the results where the main test and validation test strongly contradicts each other. We define a strong contradiction as when the significance levels differ with more than 50%.

Marked in green are the results where the direction of the difference measured differs between the two tests. In other words, where the measurements of latencies A and B for both tests have the properties of $A < B$ for the main test while $B < A$ for the validation test. In these cases both tests might yield the same kind of significance level, when in fact the actual measurements are very different.

Marked with brown are cells with both the red and green property.

No conclusions can be drawn on the basis of any significance level with either of these properties.

Looking at the results so far we can see that the only operation for which a clear difference between methods can be observed is for the READ operation. Not only are the standard deviation for the other operations to high, but where we might seem to notice some difference, the main and validation run largely contradict each other.

When it comes to the overall throughput of the workloads we can in the significance

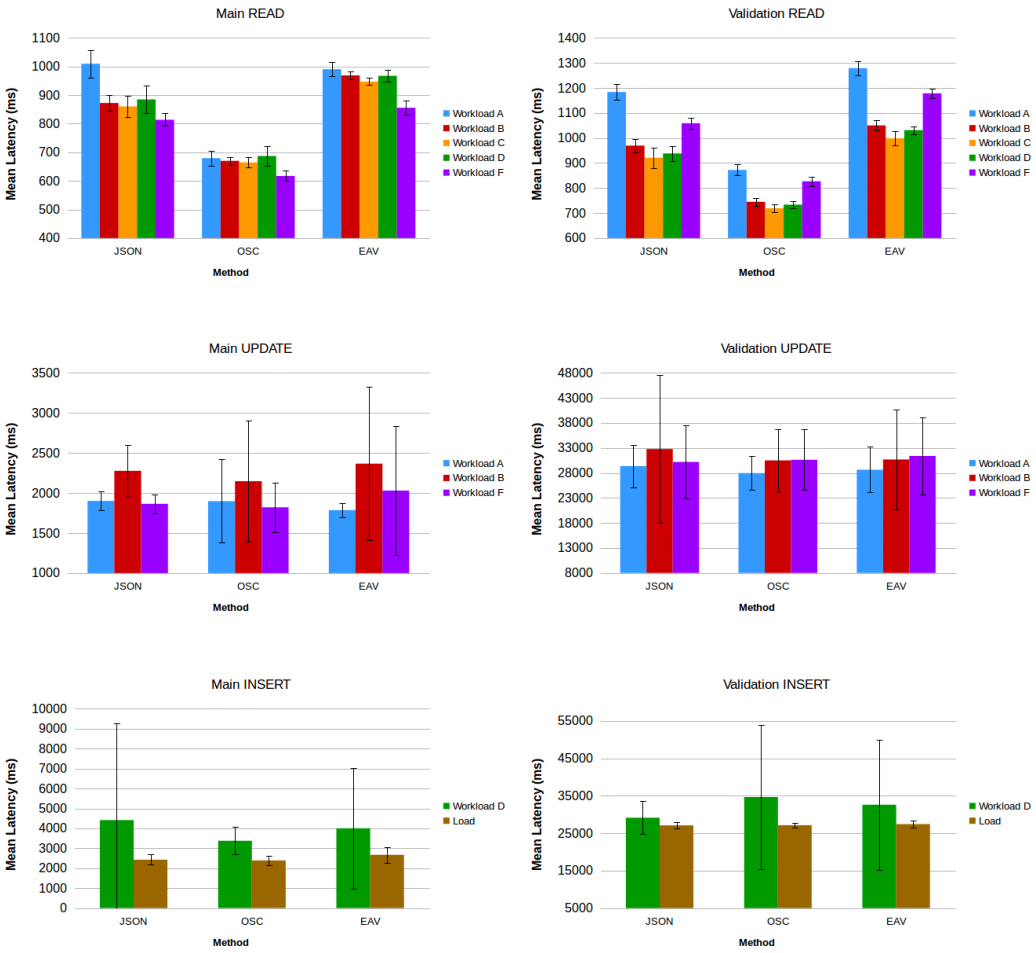

Figure 8.1: Average latencies and standard deviation for all operations used with different schemaless methods and workloads, executed on both the main and validation test machine

	JSON-EAV	EAV-OSC	JSON-OSC
A READ	2.84%	0.00%	0.00%
A UPDATE	0.00%	18.73%	97.01%
B READ	0.00%	0.00%	0.00%
B UPDATE	57.82%	25.93%	32.34%
C READ	0.00%	0.00%	0.00%
D READ	0.00%	0.00%	0.00%
D INSERT	65.15%	21.14%	18.98%
F READ	0.00%	0.00%	0.00%
F UPDATE	20.89%	13.44%	41.48%
Load INSERT	0.16%	0.02%	50.71%
A	0.00%	0.02%	0.00%
B	0.00%	0.00%	0.00%
C	0.00%	0.00%	0.00%
D	2.19%	0.00%	0.00%
F	6.12%	0.00%	0.00%
Load	0.02%	0.00%	50.75%

Table 8.1: Main T-test

	JSON-EAV	EAV-OSC	JSON-OSC
A READ	0.00%	0.00%	0.00%
A UPDATE	46.38%	45.69%	10.88%
B READ	0.00%	0.00%	0.00%
B UPDATE	46.12%	92.32%	37.42%
C READ	0.00%	0.00%	0.00%
D READ	0.00%	0.00%	0.00%
D INSERT	22.81%	61.84%	8.34%
F READ	0.00%	0.00%	0.00%
F UPDATE	47.51%	62.32%	77.23%
Load INSERT	10.53%	12.18%	81.38%
A	42.34%	31.52%	5.60%
B	54.40%	0.07%	2.17%
C	0.00%	0.00%	0.00%
D	0.24%	0.75%	67.65%
F	38.25%	46.88%	84.08%
Load	8.87%	11.59%	77.64%

Table 8.2: Validation T-test

test observe significant differences in many of the comparisons. However, this is almost exclusively attributed to the large use of the READ operation in these workloads.

When looking deeper into the results for the READ operation we see that the OSC method has the highest performance, followed by the JSON method that mostly performs just under the EAV method.

The data that table 8.1, 8.2 and plots in figure 8.1 are based on, is presented in appendix B.

Tables 8.3 and 8.4 shows some metrics of the CPU and memory usage for each con-

	YCSB		MySQL	
	CPU (%)	MEM (%)	CPU (%)	MEM (%)
Average Utilization	62.93%	1.09%	13.20%	4.83%
Standard Deviation	21.20%	0.38%	7.21%	0.005%
Min	21.54%	0.03%	1.14%	0.38%
Max	94.38%	1.49%	27.69%	4.84%

Table 8.3: average RAM and CPU usage for the main test

	YCSB		MySQL	
	CPU (%)	MEM (%)	CPU (%)	MEM (%)
Average Utilization	9.26%	0.93%	2.85%	5.26%
Standard Deviation	13.43%	0.12%	1.01%	0.03%
Min	0.08%	0.05%	0.06%	5.18%
Max	81.24%	1.15%	5.95%	5.28%

Table 8.4: average RAM and CPU usage for the validation test

tainer during the two test runs. These measurements were taken using the command `docker stats` at regular intervals of one second.

In both runs the average usage of RAM is moderate and in a range that suggests no effect on the reproducibility of the tests. The CPU usage differs substantially however, and is much larger for the main test than the validation test.

Another peculiar difference between the main and validation test lies in the average latencies. Looking at the READ plots in 8.1 we see that the main test is just slightly faster than the validation test. However, when looking at UPDATE and INSERT we see that the main test performs the operations with one order of magnitude faster than the validation test.

Chapter 9

Discussion and Conclusion

What was tried to be accomplished with this study was to investigate and bring light upon different ways to achieve schemaless data in a relational database and how these compared to each other with regard to performance.

The first part of the problem statement was to find some methods for achieving schemaless data storage on top of a relational database. This part of the problem has mainly been answered in the theoretical background. Only some of the methods brought up in the theoretical background were tested in the end. Much due to ambiguities between the methods, as mentioned in 4.1.2.

The other part of the problem statement was to compare the performance of the chosen methods with the goal of generating some general decision basis for choosing between the methods in a real world scenario. The problem thereafter was to limit the number of possible ways to test the performance. The result of this delimitation has the drawback of being too simplistic to be representative of real world use cases. It mainly gives an idea of the baseline in performance difference one can expect. Also, to really be able to state that one method performs better than another in general, the possible ways of optimizing the methods and how these differ must also be taken into account.

The benchmarking platform chosen for the tests did not feature measurements of the DELETE operation, which was one of the four basic CRUD-operations to be tested. This further limits the potential use cases the result can be used for.

A large focus when constructing the test bed was on reproducibility. Two different tests were run on two separate machines to verify both the results and if a reproducible test had been achieved. The points where both results agree are assumed to be verified. Based on this assumption, a clear difference in performance when it comes to READ operations was observed. Other than that, no significant difference with a significance level of under 5% could be verified.

The performance measurement of the READ operation sugests that the OSC method is the fastest, followed by the JSON and EAV method in that order. That the OSC method performed better comes as little surprise considering it essentially uses the MySQL database the way it is supposed to be used. The magnitude of the differences can also be considered substantial enough to have an impact on the choice of method if the use case is based on mostly READ.

Two questions regarding the data remains unanswered when it comes to the reproducibility:

- Why does the performance for INSERT and UPDATE differ substantially between the two tests, but not of READ?

- Why is the standard deviation for INSERT and UPDATE so high and inconsistent in both tests compared to READ?

We can only speculate. Looking at the specification of both systems used, there is no obvious answer. Further studies trying to reproduce our results could help find the cause of this behaviour and maybe eliminate the variance to a degree where it is possible to measure significant differences even for INSERT and UPDATE.

Apart from confirming the result of this study there are several other broadening areas of interest, not covered in this study. These include testing the performance when sharding the database and doing more complex operations found in real world scenarios. To broaden the perspective even more, a comparison between some popular NoSQL databases and schemaless SQL based methods would also be of interest. Apart from comparing performance, other features also need comparison when deciding upon what method best suits a certain task. These include scalability, compatibility and ease of use.

As a basis for deciding upon methods the result of this study is not sufficient. Partly because it needs validation, and partly because it measures performance on a setup that is too far from a real world scenario. It could however provide some useful information and guidelines for further studies into the subject.

Bibliography

[1] D.A. Menascé and V.A.F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, 2002. Chap. Chapter 7 - Benchmarks and Performance Tests. ISBN: 9780130659033. URL: https://books.google.se/books?id=s99QAAAAMAAJ.

[2] R. Patton. *Software Testing*. Sams Pub., 2006. Chap. Chapter 4 - Examining the Specification. ISBN: 9780672327988. URL: https://books.google.se/books?id=MTEiAQAAIAAJ.

[3] Subharthi Paul. "Database Systems Performance Evaluation Techniques". In: *St. Louis, MI, USA* (2008).

[4] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152. URL: http://doi.acm.org/10.1145/1807128.1807152.

[5] B. G. Tudorica and C. Bucur. "A comparison between several NoSQL databases with comments and notes". In: *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*. June 2011, pp. 1–5. DOI: 10.1109/RoEduNet.2011.5993686.

[6] P.J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Always learning. Addison-Wesley, 2012. ISBN: 9780321826626. URL: https://books.google.se/books?id=tYhsAQAAQBAJ.

[7] Craig Chasseur, Yinan Li, and Jignesh M. Patel. "Enabling JSON Document Stores in Relational Systems." In: *WebDB*. Ed. by Angela Bonifati and Cong Yu. 2013, pp. 1–6. URL: http://dblp.uni-trier.de/db/conf/webdb/webdb2013.html#ChasseurLP13.

[8] Martin Fowler. *Schemaless Data Structures*. Jan. 2013. URL: https://martinfowler.com/articles/schemaless/ (visited on).

[9] Anne Håkansson. "Portal of research methods and methodologies for research projects and degree projects". In: *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). 2013, p. 1.

[10] Y. Li and S. Manoharan. "A performance comparison of SQL and NoSQL databases". In: *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. Aug. 2013, pp. 15–19. DOI: 10.1109/PACRIM.2013.6625441.

[11] Ameya Nayak, Anil Poriya, and Dikshay Poojary. "Type of NOSQL databases and its comparison with relational databases". In: *International Journal of Applied Information Systems* 5.4 (2013), pp. 16–19.

[12] S. H. Aboutorabi et al. "Performance evaluation of SQL and MongoDB databases for big e-commerce data". In: *2015 International Symposium on Computer Science and Software Engineering (CSSE)*. Aug. 2015, pp. 1–7. DOI: 10.1109/CSICSSE.2015.7369245.

[13] Carl Boettiger. "An Introduction to Docker for Reproducible Research". In: *SIGOPS Oper. Syst. Rev.* 49.1 (Jan. 2015), pp. 74–77. ISSN: 0163-5980. DOI: 10.1145/2723872.2723882. URL: http://doi.acm.org/10.1145/2723872.2723882.

[14] Jürgen Cito, Vincenzo Ferme, and Harald C. Gall. "Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research". In: *Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings*. Ed. by Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso. Cham: Springer International Publishing, 2016, pp. 609–612. ISBN: 978-3-319-38791-8. DOI: 10.1007/978-3-319-38791-8_58. URL: http://dx.doi.org/10.1007/978-3-319-38791-8_58.

[15] Vincent Reniers et al. "On the State of NoSQL Benchmarks". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17 Companion. L'Aquila, Italy: ACM, 2017, pp. 107–112. ISBN: 978-1-4503-4899-7. DOI: 10.1145/3053600.3053622. URL: http://doi.acm.org/10.1145/3053600.3053622.

[16] *Active TPC Benchmarks*. URL: http://www.tpc.org/information/benchmarks.asp (visited on).

[17] *DB-Engines Ranking*. URL: https://db-engines.com/en/ranking (visited on).

[18] *Measurement System Analysis (MSA)*. URL: https://www.moresteam.com/toolbox/measurement-system-analysis.cfm (visited on).

[19] *MySQL 5.7 Reference Manual :: 11.6 The JSON Data Type*. URL: https://dev.mysql.com/doc/refman/5.7/en/json.html.

[20] *Purpose of database schemas*. URL: https://www.ibm.com/support/knowledgecenter/en/SSBJG3_2.5.0/com.ibm.gen_busug.doc/c_fgl_DatabaseSchema_002.htm (visited on).

[21] Amazon Web Services. *What is a Document Database? – AWS*. URL: https://aws.amazon.com/nosql/document/ (visited on).

[22] Karl Sigman. *Lecture Notes on Stochastic Modeling I*. URL: http://www.columbia.edu/~ks20/stochastic-I/stochastic-I.html (visited on).

Appendix A

Datamodels and SQL code

This chapter contains the data models and queries implemented for the three methods tested in this study. The queries are written in the same way they are implemented in JAVA with JDBC. JDBC uses stored procedures that can be executed with parameters like a function. In the following SQL code, such parameters are written as question marks (?). When using a JDBC stored procedure, the parameters are supplied in the same order as they are read in the query.

A.1 JSON

ycsb_key	fields
0	{"field0": A, "field1": B, "field2": C, ... }
1	{"field0": D, "field1": E, "field2": F, ... }
...	...

Table A.1: Data model for the JSON method

Listing A.1: JSON Table creation

```
CREATE TABLE usertable (
        ycsb_key VARCHAR(255) PRIMARY KEY,
        fields JSON
)
```

Listing A.2: JSON Insert operation

```
INSERT INTO usertable (ycsb_key, fields)
VALUES(?, JSON_OBJECT(
        "FIELD0", ?, "FIELD1", ?, "FIELD2", ?, ...)
)
```

Listing A.3: JSON Update operation

```
UPDATE usertable SET fields = JSON_SET(
        fields, '$.field0', ?, '$.field1', ?, '$.field2', ?, ...
)
WHERE ycsb_key = ?
```

Listing A.4: JSON Read operation

```
SELECT * FROM usertable WHERE ycsb_key = ?
```

Listing A.5: JSON Delete operation

```
DELETE FROM usertable WHERE ycsb_key = ?
```

A.2 EAV-table

ycsb_key	field	value
0	"field0"	A
0	"field1"	B
0	"field2"	C
1	"field0"	D
1	"field1"	E
1	"field2"	F
...	...	...

Table A.2: Data model for the EAV-table method

Listing A.6: EAV Table creation

```
CREATE TABLE usertable (
        ycsb_key VARCHAR(255) NOT NULL,
        field VARCHAR(255) NOT NULL,
        value TEXT,
        CONSTRAINT pk_usertable PRIMARYKEY(ycsb_key, field)
)
```

Listing A.7: EAV Insert operation

```
INSERT INTO usertable (ycsb_key, field, value)
VALUES (?, 'field0', ?), (?, 'field1', ?), (?, 'field2', ?), ...
```

Listing A.8: EAV Update operation

```
UPDATE usertable SET VALUE =
        CASE field
        WHEN 'field0' THEN ?
        WHEN 'field1' THEN ?
        WHEN 'field2' THEN ?
        END WHERE field IN('field0', 'field1', 'field2', ...)
        AND ycsb_key = ?)
```

Listing A.9: EAV Read operation

```
SELECT * FROM usertable WHERE ycsb_key = ?
```

Listing A.10: EAV Delete operation

```
DELETE FROM usertable WHERE ycsb_key = ?
```

A.3 OSC

ycsb_key	field0	field1	field2	...
0	A	B	C	...
1	D	E	F	...
...	...	...	...	...

Table A.3: Data model for the OSC method

A.3.1 SQL code

Listing A.11: OSC Table creation

```
CEATE TABLE usertable(
        ycsb_key VARCHAR(255) PRIMARY KEY
        field0 TEXT, field1 TEXT, field2 TEXT, ....
)
```

Listing A.12: OSC Insert operation

```
INSERT INTO usertable(ycsb_key, field0, field1, field2, ...)
VALUES (?, ?, ?, ?, ...)
```

Listing A.13: OSC Update operation

```
UPDATE usertable SET field0 = ?, field1 = ?, field2 = ?, ...
WHERE ycsb_key = ?
```

Listing A.14: OSC Read operation

```
SELECT * FROM usertable WHERE ycsb_key = ?
```

Listing A.15: OSC Delete operation

```
DELETE FROM usertable WHERE ycsb_key = ?
```

Appendix B

Data

Method Workload Operation	Mean Latency (ms)	Standard Deviation	Standard Deviation %
JSON A READ	1010	48	5%
JSON B READ	872	27	3%
JSON C READ	861	36	4%
JSON D READ	885	48	5%
JSON F READ	814	22	3%
EAV A READ	991	26	3%
EAV B READ	969	15	2%
EAV C READ	947	12	1%
EAV D READ	967	20	2%
EAV F READ	856	24	3%
OSC A READ	680	26	4%
OSC B READ	670	13	2%
OSC C READ	665	19	3%
OSC D READ	687	33	5%
OSC F READ	617	18	3%
JSON Load INSERT	2426	259	11%
EAV Load INSERT	2671	393	15%
OSC Load INSERT	2389	235	10%
JSON D INSERT	4413	4872	110%
EAV D INSERT	4001	3041	76%
OSC D INSERT	3376	680	20%
JSON A UPDATE	1901	117	6%
JSON B UPDATE	2278	326	14%
JSON F UPDATE	1865	121	6%
EAV A UPDATE	1786	88	5%
EAV B UPDATE	2367	958	40%
EAV F UPDATE	2030	810	40%
OSC A UPDATE	1898	521	27%
OSC B UPDATE	2148	756	35%
OSC F UPDATE	1821	311	17%

Table B.1: Latencies from main test

Method Workload Operation	Mean Latency (ms)	Standard Deviation	Standard Deviation %
JSON A READ	1184	31	3%
JSON B READ	970	27	3%
JSON C READ	921	41	4%
JSON D READ	938	29	3%
JSON F READ	1059	23	2%
EAV A READ	1279	29	2%
EAV B READ	1050	20	2%
EAV C READ	999	30	3%
EAV D READ	1032	16	2%
EAV F READ	1179	18	2%
OSC A READ	872	22	3%
OSC B READ	745	15	2%
OSC C READ	719	15	2%
OSC D READ	734	15	2%
OSC F READ	827	18	2%
JSON Load INSERT	27072	823	3%
EAV Load INSERT	27416	1037	4%
OSC Load INSERT	27112	655	2%
JSON D INSERT	29113	4396	15%
EAV D INSERT	32588	17429	53%
OSC D INSERT	34640	19225	56%
JSON A UPDATE	29369	4290	15%
JSON B UPDATE	32775	14759	45%
JSON F UPDATE	30204	7247	24%
EAV A UPDATE	28640	4563	16%
EAV B UPDATE	30687	9982	33%
EAV F UPDATE	31404	7711	25%
OSC A UPDATE	27969	3366	12%
OSC B UPDATE	30507	6180	20%
OSC F UPDATE	30638	6085	20%

Table B.2: Latencies from validation test

Method Workload	Mean Throughput (ops/sec)	Standard Deviation	Standard Deviation %
JSON A	527	19	4%
JSON B	736	16	2%
JSON C	796	26	3%
JSON D	690	86	12%
JSON F	454	16	4%
EAV A	545	14	3%
EAV B	684	24	4%
EAV C	738	8	1%
EAV D	655	43	7%
EAV F	438	50	11%
OSC A	586	62	11%
OSC B	863	26	3%
OSC C	934	17	2%
OSC D	801	38	5%
OSC F	505	35	7%
JSON Load	348	27	8%
EAV Load	322	32	10%
OSC Load	352	25	7%

Table B.3: Throughput of overall performance from main test

Method Workload	Mean Throughput (ops/sec)	Standard Deviation	Standard Deviation %
JSON A	64	9	14%
JSON B	349	65	19%
JSON C	736	26	4%
JSON D	362	38	11%
JSON F	62	13	20%
EAV A	66	9	14%
EAV B	341	46	14%
EAV C	691	15	2%
EAV D	331	47	14%
EAV F	60	12	21%
OSC A	68	7	11%
OSC B	379	49	13%
OSC C	858	12	1%
OSC D	367	66	18%
OSC F	62	11	18%
JSON Load	36	1	3%
EAV Load	36	1	4%
OSC Load	36	1	2%

Table B.4: Throughput of overall performance from validation test