
**Gonçalo Alexandre Pinto Tomás**

Bachelor of Science

# FMKe: a real-world benchmark for key-value stores

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
**Computer Science and Engineering**

Adviser: João Leitão, Assistant Professor,
NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
**UNIVERSIDADE NOVA** DE LISBOA

**June, 2017**

# Abstract

Standard benchmarks are essential tools to evaluate and compare database management systems in terms of both relevant properties and performance. They provide the means to evaluate a system with workloads that mimic real applications. Although a number of realistic benchmarks already exist for relational database systems, the same cannot be said for No-SQL databases. This latter class of data storage systems is increasingly relevant for geo-distributed systems, and this has led developers and researchers to either rely on benchmarks that don't have realistic workloads or to adapt the aforementioned benchmarks for relational databases to work for No-SQL databases. Since these benchmarks assume an isolation and transactional model in the database, they are inherently inadequate to evaluate No-SQL databases.

We propose a new benchmark that addresses the lack of realistic evaluation tools for distributed key value stores. We adapt the workload from information we have acquired about a real world deployment of a large-scale application that operates over a distributed key value store that is responsible for managing patient prescriptions at a nation-wide level in Denmark. We design our benchmark to be extensible to a wide range of distributed key value storage systems with minimal effort for programmers, which only need to design and implement specific data storage drivers to benchmark their solutions.

# Resumo

Os *benchmarks* são ferramentas essenciais para avaliar e comparar sistemas de gestão de bases de dados relativamente às suas propriedades e desempenho. Estes fornecem os meios para avaliar um sistema através da injeção sistemática de cargas de trabalho que imitam aplicações reais. Embora já existam vários benchmarks realistas para sistemas de gestão de bases de dados relacionais, o mesmo não se verifica para bases de dados No-SQL. Esta última classe de sistemas de armazenamento de dados é cada vez mais relevante para os sistemas geo-distribuídos, o que levou a que programadores e investigadores recorressem a benchmarks que não possuem cargas de trabalho realistas ou adaptassem os benchmarks para bases de dados relacionais para operarem sobre bases de dados No-SQL. No entanto, como esses benchmarks assumem um modelo de isolamento e transacional na base de dados, são inerentemente inadequados para avaliar uma base de dados No-SQL.

Propomos um novo benchmark que aborda a falta de ferramentas de avaliação realistas para sistemas de armazenamento chave-valor distribuídos. Para esse fim recorremos a um gerador de operações que injeta carga sobre a base de dados a partir de informação que obtivemos sobre uma instalação de uma aplicação real em larga escala que opera sobre um sistema de armazenamento chave-valor distribuídos responsável pela administração de receitas de pacientes a nível nacional na Dinamarca. Desenhámos o nosso benchmark para ser extensível a uma ampla gama de sistemas de armazenamento chave-valor distribuídos com um esforço mínimo para programadores, que só precisam desenhar e implementar *drivers* específicos para avaliar as suas soluções.

# Contents

# List of Figures

# List of Tables

# Listings

Chapter 1

# Introduction

## 1.1 Introduction

Database Management Systems (DBMS) are becoming ubiquitous in today's society. As companies and governments transition to digital storage of documents and information, almost every new project requires some form of persistent data storage. DBMS also provide easy access to data while abstracting away complex issues related with efficiently retriving data, processing transactions, recovering from crashes, etc.

In the last decades developers have been given plenty of choices for databases: we've seen the rise and widespread adoption of projects like MySQL, PostgreSQL among others; over time the developer teams behind these database systems tend to implement useful missing features that are present in competing databases, and nowadays traditional databases that follow the SQL standard provide very similar feature sets. This causes a problem for project managers, since it is hard to compare and decide between similar database management systems.

Despite offering similar features, their implementation is not identical across different storage systems and this leads to significant performance variations. Without a clear specification of evaluation circumstances and scenarios, it is hard to compare performance between distinct systems. This problem resulted in the establishment of the Transaction Processing Council (TPC), which has taken for many years the responsibility of developing and specifying standard benchmarks for different types of DBMS, that can be used to sistematically exercise the operations of databases and compare their performance through the use of workloads inspired by real world use cases.

The release of the Yahoo's PNUTS [Coo+08] and the Amazon Dynamo [DeC+07] systems marked the beggining of a new important type of data storage system: distributed

key value stores. These new types of databases offered a simpler interface than conventional DBMS while explicitely departing from the relational model and not supporting SQL-like languages, leading to their famous "No-SQL"classification. Many projects based on PNUTS and Dynamo were founded (e.g. Riak [Nosb], CouchDB [Nosa]. This new class of databases started to see wide adoption particularly in the context of web applications [DeC+07]. In fact, the emergence of these databases was motivated by the need to provide high performance for use cases where the semantics of relational databases were not required.

There have been many efforts to adapt existing TPC benchmarks to work with these distributed key value stores as a way to effectively compare their performance is a systematic way. However, this adaptations are rarely effective, as the use cases captured by them do not reflect appropriately to the use cases targetted by distributed key value stores. This lead to the need of new benchmarks specifically targetted for distributed key value stores. In 2010 Yahoo also released a specification for a benchmark called Yahoo Cloud Serving Benchmark (YCSB) [Coo+10] that was tailored for distributed key-value stores, which quickly became the evaluation standard for this subset of databases.

Aside from being adapted for key value stores, YCSB still had a fundamental difference from the TPC benchmarks: its workload consisted of synthetic read/write operations, while in the TPC benchmarks the workload mimicked real world application patterns. While YCSB is considered a very good baseline performance benchmark for key value stores, this benchmark still lacks somewhat in providing workloads that successfully emulate more sophisticated and realistic application patterns.

## 1.2 Problem definition

In the thesis we propose to tackle the existing limitations of benchmarks for distributed key value stores, that on one hand can provide a sistematic way to evaluate and compare the performance benefits of different distributed key value stores, and on the other hand is based on a realistic use case with operations that illustrate more realistic and interesting data access patterns. To this end we will base our work on information that we have acquired regarding a realistic use case of distributed key value stores, in particular the FMK system which is responsible for managing patient prescriptions at a nation-wide level in Denmark.

Our benchmark, that we name FMKe, will feature multiple representative workloads of the original system, that can be parameterised by the user in order to control the size of the dataset, and the (average) amount of conflicting operations, among others. Moreover, the benchmark will be suitable to experimentally access the performance of distributed key value stores in single site deployments and in geo-replicated deployment, which are both extremely relevant nowadays.

Furthermore, and to ease the wide adoption of our benchmark, we will design it to be extensible to a wide range of distributed key value storage systems, this could

be achieved through the use of (application-level) drivers that transparently translate a generic specification of the operation in the FMK representative workloads into the concrete operation offered by the distributed key value store interface.

## 1.3 Structure of the Document

In the following chapter of this document an extended set of concepts are introduced to provide some context for this thesis, which involves a brief overview and definition of databases and some types of databases that are relevant for this work, followed by some properties like replication and consistency guarantees that are inherently different in the two mentioned database types. The related work chapter also includes a primer on database benchmarks and their importance for comparing available storage systems. A division is then made between benchmarks appropriate for relational databases and benchmarks for No-SQL databases, and some limitations of existing solutions are provided to justify this work.

In chapter 3 we present our work FMKe, a proposal of a novel benchmark for distributed key-value stores. Its specification and workload are detailed, and some focus is given into the architecture and how to deploy the benchmark. A planning section is included that enumerates work that has been done as well as the next milestones to complete the thesis, with a clear distribution of work over time until the final delivery phase in March 2018. Finally, a discussion compiles some conclusions and summarises our expectations for the future.

Chapter 2

# Related Work

## 2.1 Introduction

This chapter contains the main related work for the proposed work described here. First, a brief introduction about databases is provided to give essential context, followed by a comparison of the most relevant aspects between relational and No-SQL databases. For each one of these two types of databases, we describe the data model and programmer interface in order to clearly outline the differences in goals and architectures. We then discuss the deployment scenario, specifically involving properties like data replication and consistency between replicas and further analyze the behaviour of the two types of databases under different alternatives for these aspects.

Once a good contrast is made between relational and No-SQL databases, we present some of the existing industry benchmarks, the metrics they obtain when they evaluate a database, and their suitability for providing systematic and valid experimental comparisons between different alternatives. Finally, we argue how some of these benchmarks are poorly adequate to evaluate different types of databases and provide our arguments supporting that the current state of the art is insufficient to fairly, realistically and systematically evaluate distributed key-value stores.

## 2.2 Databases

A database is a collection of data that is usually organized to support fast search and data retrieval [Mer17]. Despite some applications being able to operate in a stateless way, most computer systems will need to handle state data that needs to be safely stored and thus require a some form of database system. A diverse range of applications use databases: from websites that need to keep a record of purchased products and orders,

to governments needing to digitally store information about citizens, virtually any new systems will need to allocate resources for a data storage solution.

Currently available data storage systems can be broadly divided into two different categories: SQL-based databases (also referred to as relational databases) and No-SQL databases (or distributed key value stores). When designing a computer system, system requirements usually dictate what type of database is chosen since they inherently are better tailored for different use cases. Making the wrong decision for the data storage component of a system may impact its scalability and performance, as will become clear in the following sections.

In the next sections we go into detail about the characteristics of SQL and No-SQL databases, comparing their data model, programmer interface, and discussing some of the relevant examples that are in use today. This information will serve as a base to understand in which circumstances each type of database should be used, and secondly to discuss how each one of the databases types handles data replication, as well as keeping each one of the replicas consistent, which are key aspects to fully grasp the different scenarios where each of these class of storage solutions are more appropriate, and hence relevant to design adequate benchmarks for their evaluation.

## 2.3 Relational Databases

Relational databases are an iteration over the first-generation navigational databases [Bac73]. With the appearance of the SEQUEL language [CB74] and the following Structured Query Language (SQL) standard that first described a declarative data access language, this type of database quickly became the most used, and they still hold that title nowadays.

Databases that follow the SQL standard (and typically implement the relational data model) usually also provide transactional support with varying levels of isolation, as will be detailed further ahead. Having support for transactions is particularly useful for application programmers, since it becomes easier for them to reason about the evolution of the application state (at least when operating with more restrictive isolation levels). It is not hard to conceptualize examples of application operations where atomicity is needed: a classical example is a bank transfer between two accounts, which consists in subtracting an amount in one account and crediting another account with the same amount. This feature is much harder to implement in other types of database systems due to the inherent constraints that have to be ensured in its execution, particularly in what regards atomicity, either both operations succeed or both fail, otherwise money could either disapear or be created; and operations should ensure that the balance of each account does not go to a negative value.

This type of databases is pervasive, with relevant uses in banking, websites, online games, etc.

### 2.3.1 Data Model

Relational databases use the relational model firstly proposed by Edgar Codd in 1970 [Cod70]. In the relational model, all data entries are represented in *tuples* and tuples are grouped into *relations*. More precisely, data is organised into tables called *relations*, and each record inside a table is called a *tuple*. Table columns are called *attributes* and form the basis for the types of queries that can be performed over tables. SQL includes other aggregation constructs built into the language that also allow to effectively compute maximums, minimums, averages, counts, etc. among stored values.

### 2.3.2 Programmer Interface

Programmers can execute operations on a relational DBMS using transactions. Transactions are conceptually arbitrarily complex operations over data that feature what is commonly denominated ACID properties [Sil+97], which is an acronym for the following properties:

- **Atomicity**: the database system must ensure that every transaction is not partially executed, in any circumstance (may include system crashes or power failures). If a transaction contains an instruction that fails then the whole transaction must fail and the database must be left unchanged. This provides an easy to reason with "all or nothing" semantics.

- **Consistency**: before and after transactions execute, the database must be in a correct state. This means that a successful transaction makes the database transition between two correct states, which means that any data written must abide by all applicable rules including referential integrity, triggers, etc.

- **Isolation**: this property is only relevant when considering the concurrent execution of multiple transactions. Usually several isolation levels are provided by the DBMS, the weakest being that two concurrent transactions will be able to see each others effects in the database even when both are incomplete (i.e. before they commit). The highest isolation level is an equivalence to a serialized execution of the transactions, which means that any two concurrent transactions cannot modify the same data.

- **Durability**: the system must ensure that once transactions enter the committed state they are recorded in non-volatile storage in a fault tolerant way (again, this may include system crashes or power failures), and that their effects will not be forgotten.

Regarding the provided isolation levels, despite some vendors not supporting the higher Serializable level, many provide the following levels:

- **Read Uncommitted**: This is the lowest possible isolation level, which does not provide any guarantees. Transactions may read values that have not yet been committed. In practice this value is not used very often.

- **Read Committed**: Transactions can only read values that have been committed to the database, so any two concurrent transactions will not be able to read each other's values.

- **Snapshot Isolation**: Like the name indicates, each transaction reads from a snapshot of the database that is taken when the transaction begins. This level, despite one of the highest and being used often, is not equivalent to the Serializable level since at this isolation level some anomalies may occur (write skews)

- **Serializable**: This is the highest possible isolation level, and also the one with the biggest performance impact. A concurrent scheduling of multiple transactions executed at this isolation level should be equivalent to one sequential execution of the same set of transactions.

### 2.3.3 Relevant Examples

Some examples of relational DBMS include Microsoft SQL Server [Sqlb], Oracle 12 [Sqld], MySQL [Sqlc], PostgreSQL [Sqle] and MariaDB [Sqla].

## 2.4 No-SQL Databases

The need for scaling data storage beyond what the state of the art allowed was the catalyst behind the development of modern databases that fall into the category of No-SQL databases. The No-SQL label is however too broad: any storage system that does not follow the SQL standard (even in cases where a relational model is provided) can be considered to some extent a No-SQL database. In this context, there are document stores, graph-oriented databases, and others, but for the remainder of this document we restrict the meaning of the term "No-SQL database"to refer only to key value storage systems, typically based on distributed hash tables like the Amazon Dynamo[DeC+07] or Yahoo PNUTS[Coo+08].

### 2.4.1 Data Model

In No-SQL databases, data is stored in key-value pairs. A *key* is a unique identifier of the data (typically a string of characters), and the *value* can be of any type, depending on the types of values supported by each data storage system. Some distributed key-value stores associate types with some values. For instance, in Riak, values can be maps (and support nested key-value pairs), sets, registers, etc.

### 2.4.2 Programmer Interface

While relational databases have support for rich queries using SQL, most No-SQL databases only provide a $(get, put)$ interface. At first sight this appears to only allow for very basic queries, but there have been several No-SQL databases that provide support for subsets of SQL like Apache Cassandra [Apa] [Coc]. These include queries using predicate selection using exact matches, and even basic transactional support.

### 2.4.3 Relevant Examples

The more relevant examples of No-SQL databases are Cassandra, CouchDB, Dynamo, Redis and Riak.

## 2.5 Replication and Data Consistency

Data replication is one of the most important properties in distributed databases. Being able to replicate information means that in case of a system failure data will not be lost, and the system may provide higher availability than keeping all data in a centralized component which becomes a single point of failure. With replication comes additional performance overhead of maintaining consistency between all copies, which also poses problems when network partitions are considered.

Nowadays there are many services that operate at a global level like Google Mail or Dropbox. To provide a good service to all users a centralized data storage component is a bad approach, since clients further away from the datacenter where the database is stored will experience drastically increased latency. Hence, a common approach for these types of systems is to provide geo-replication: having the database be replicated in multiple datacenters spread across the world, and each client connects to the closest replica. Of course, in this scenario the aforementioned overhead of keeping the replicas consistent is orders of magnitude higher, and these types of systems usually relax their consistency requirements to avoid spending a significant portion of time synchronizing the information between all the datacenters. Weaker consistency constraints are captured by consistency models such as eventual consistency, which allows outdated values to persist on a database with guarantees that after write operations stop being issues, all replicas will converge to the same value at some undefined point in time. This apparently small change has a big impact in performance, since only periodic synchronization is required to be performed across datacenters, contrary to the per-operation synchronization of the more classical consistency model (featured in relational databases) that is commonly referred as strong consistency.

### 2.5.1 Availability versus Consistency

In 2002 Brewer et all formally proved that a distributed system cannot simultaneosly have the three following properties: Consistency, Availability, and Partition tolerance. This proof was later referred as the CAP Theorem [GL02]. Figure **??** gives a visual representation of the three properties labelled by their first letter.

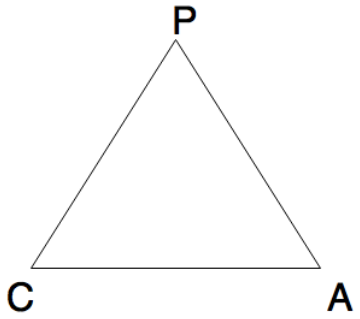


Figure 2.1: Visual representation of the CAP theorem

Let us first assume that we can ensure no network partitions will occur. Under these circumstances, we are able to build a system that is available (theoretically) 100% of the time and that all replicas of the data it stores remain consistent. This is some of the reasoning behind Google's latest database as a service product, Spanner [Bre17].

As soon as network partitions are considered (a reality for any distributed systems, and more probable for geo-distributed systems [BK14]), any database system that experiences a partition will either lose Availability or Consistency. This is a trade-off that really exposes the difference between strong consistency (typically featured in SQL based systems) and weak consistency (a design choice available in many No-SQL databases). Due to this, under a network partition, relational databases will lose Availability. This is the only way to preserve all of the invariants that are required for correct operation, which may include referential integrity.

On the other hand, since No-SQL databases have simpler data models and usually operate with relaxed consistency requirements, they are usually able to maintain availability. Any operation that is performed under a network partition will lead replicas to diverge, which makes it harder to reason about the evolution of the system state.

The choice of a DBMS to prioritize either consistency or availability is so important that these systems are labelled according to their behaviour under partition (CP for "Consistent under Partition" and AP for "Available under Partition").

One noteworthy aspect of the CAP properties is that Availability is measured as percentage of time that the system is running correctly and that it is able to perform read and write operations. This means that a CP database will not become completely unavailable, it will simply be unable to guarantee availability 100% of the time (namely under network partitions).

### 2.5.2 Common Replication Strategies for Relational Databases

Due to the constraints that are commonly associated with the data model in a relational database, some properties like referential integrity invalidate any other strategy that does not involve synchronizing between replicas as soon as an operation is executed (and before notifying the client of the operation outcome). For each operation submitted by a client, the servers need to immediately replicate the operation which might have a negative impact on performance on its own. Typically the client only receives a reply to its request after all replicas have successfully executed the operation (as a single replica being unable to execute the operation should make the operation fail). Understandably, a bigger number of replicas will translate into more synchronization and a bigger latency, and this effect is magnified in geo-replicated scenarios, since synchronization happens across datacenters that are geographically distant and therefore add to the latency overhead even further.
Any replication strategy in this context, in order to maintain the guarantees provided by SQL constraints such as referential integrity needs to use replication strategies based on state machine replication or group communication primitives, which all suffer from the previously described problem, while exhibiting limited scalability.

### 2.5.3 Common Replication Strategies for No-SQL Databases

Unlike relational databases which are constrained by the data model, No-SQL databases are more flexible in the sense that they may provide strong or weak consistency models. Relying on replication strategies similar to the ones employed for relational databases would only minimize the divergence between replicas, but since these types of systems already assume that the data may diverge, in practice it is better to take the advantage of being better performant and supporting operations under network partitions by providing weaker forms of consistency. This allows replicas to synchronize only periodically, at the expense of requiring mechanisms to handle conflicts due to the execution (on different replicas) of conflicting operations.

## 2.6 Benchmarks

A database benchmark is an application that is used to evaluate the performance and or performance of a database management system. Benchmarks use workload generation components that have data access patterns similar to those that would be seen in a real world deployment. This enables the detection of possible performance issues and even the violation of properties that the programmers expect the database system to guarantee. There are multiple metrics that benchmarks are able to obtain, but for the context of this work we will focus on the main metrics that are related to performance. Considering performance, two key metrics should be obtained: *throughput* (the number of operations

that the system is able to perform in a given time unit) and *latency* (the response time from the moment the client requests an operation until the moment it receives a reply).

### 2.6.1 Benchmarks for Relational Databases

Relational databases use the benchmarks from the Transaction Processing Council (TPC). At least three benchmarks are relevant for this work: TPC-C [Tpcc], TPC-E [Tpcd] and TPC-W [Tpce]. All of these benchmarks are realistic, and each one of them models a different application (stock management in a warehouse, brokerage firm, ecommerce website).
TPC-W has been deprecated since its workload contained transactions to be executed in deferred mode, but the remaining two benchmarks are still credited as valid and up to date.

### 2.6.2 Benchmarks for No-SQL Databases

The first benchmark to be tailored for No-SQL databases was the Yahoo Cloud Serving Benchmark (YCSB) [Coo+10] that was subject to multiple iterations like YCSB++ [Pat+11]. Despite containing a simple read/write workload that is unrealistic, this set of benchmarks have been widely accepted as one of the few options to evaluate distributed key value stores.

### 2.6.3 Adaptations of Benchmarks

Programmers trust the realistic workload of the TPC benchmarks, and have attempted to adapt some of the specifications to work with No-SQL databases. For instance, there is an active project for running the TPC-W benchmark on the Cassandra distributed key value store [Tpca], and another project that implements TPC-C with compatibility for multiple back ends [Tpcb].

Chapter 3

# The FMKe benchmark

## 3.1 Motivation

As was hinted in the previous chapter, it appears that there is an open space relative to distributed key-value store benchmarks. The available realistic benchmarks are not suitable for benchmarking these types of databases, and others that have been designed specifically for key-value stores have only synthetic workloads, which may distort performance metrics. For instance, if a distributed key-value store is optimized for read operations it will surely rank high when compared to other storage systems. However, in a workload with a higher write operation ratio, that same system might not perform as well as before. Hence and similar to what TPC benchmarks have achieved in the context of relational databases, a good benchmark for No-SQL databases is needed containing a realistic workload that is able to more accurately point out performance differences under realistic circumstances.

In the context of the SyncFree European Research project and in order to create a realistic benchmark for the AntidoteDB distributed key value store (reference platform of the project), we had several meetings with the CTO of Trifork, a company with a system in production responsible for managing some patient health data for the Denmark national medical system. This system mostly keeps records of patient prescriptions, and pharmacies have access to some patient information as well as the prescriptions in order to dispense medication. The real world system (Fælles Medicinkort, FMK) is part of the Danish National Joint Medicine Card and is also deployed using distributed key-value stores.

Our first implementation of the benchmark was tightly coupled with AntidoteDB logic, but shortly after evaluating AntidoteDB we realized that this would be a valid contribution if the benchmark could be adapted to other key-value stores. We reviewed

and iterated the architecture of the benchmark in order to use the database component in a modular way, and a significant rewrite of the benchmark code was required. In the end we succeeded in creating a modular architecture, and adding support for any database is only dependant on writing a driver for that particular storage system.

## 3.2 Benchmark Overview

FMKe is a system that manages medical information about patients. In this domain there is a need to keep records for pharmacies, treatment facilities, patients, medical staff, prescriptions, patient treatments, and medical events (such as taking medicine or medical prognosis updates). The benchmark includes a set of application-level operations, each one of them leading to the execution of a sequence of read and update operations on these entities. The set of hospitals, pharmacies, patients and medical staff act as static entities in the benchmark, so records of these entities can be populated in data stores prior to the benchmark execution (and these can be scaled in number to fit the needs of the system under evaluation). Figure 3.1 presents a simplifed view of the main entities.

FMKe can be used to evaluate distributed key value stores in multiple deployment types.

### 3.2.1 System Entities

FMKe's system entities are: pharmacies, treatment facilities, patients, medical staff, prescriptions, patient treatments, and medical events. Figure 3.1 shows a simplified Entity-Relationship diagram of the benchmark.

### 3.2.2 Architecture

## 3.3 Main Operations

Table 1 shows the operations performed by the benchmark together with their relative frequency. We have developed two variants of the benchmark with different data layouts. The non-normalized variant follows the strategy of storing data in a denormalized form, which allows to serve most reads without joining data from different records. The other variant stores data in a normalized form, which leads to smaller object sizes, but requires to join data from multiple records when reading. Table 1 includes the respective number of reads and writes for the operations in the two implementations. We now describe the individual operations.

- **Create prescription**: registers a new prescription record that is associated with a patient, medical staff and pharmacy. After creation the prescription is considered to be open (i.e. it was not yet handled by a pharmacy in order to deliver medicine to the patient).

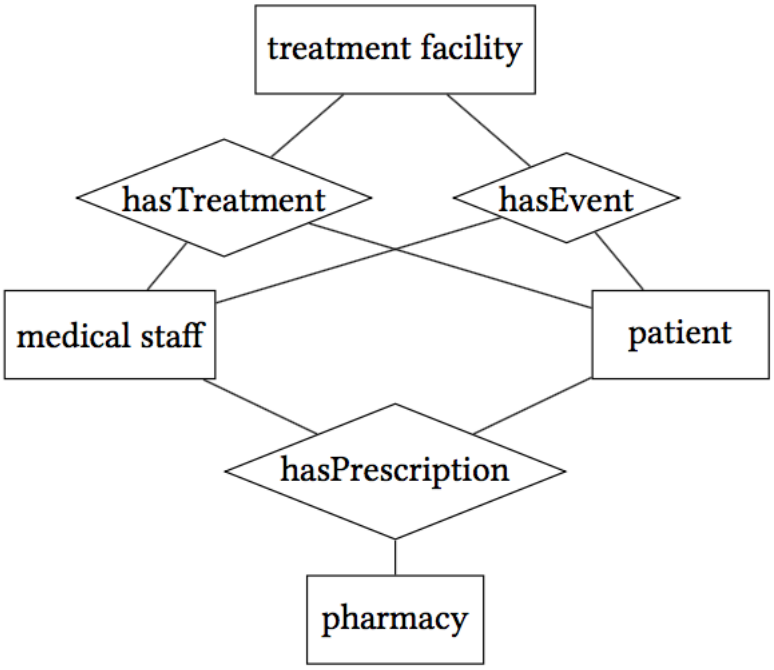


Figure 3.1: Simplified ER diagram that models FMKe.

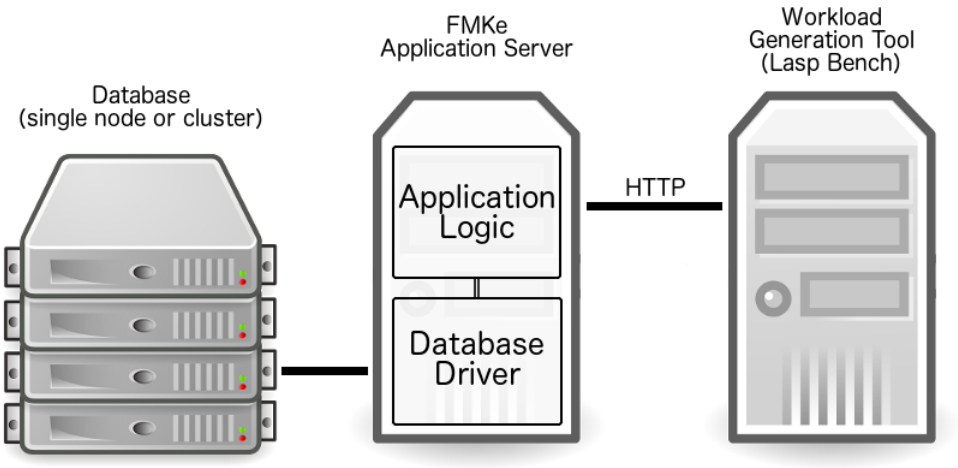


Figure 3.2: Benchmark architecture.

- **Create prescription**: registers a new prescription record that is associated with a patient, medical staff and pharmacy.
- **Process prescription**: changes the state of a prescription record to signal that it has been handled, so it transitions its state to the closed value.
- **Get staff prescriptions**: returns all prescription records that are associated with a specific medical staff member.
- **Get pharmacy prescriptions**: returns all prescription records associated with a given pharmacy.
- **Get processed prescriptions**: returns only prescriptions that have been handled (closed).
- **Get prescription medication**: returns the medication for a specific prescription record.
- **Update prescription medication**: changes the medication for a prescription that has not been processed.

## 3.4 Workload Generation

The workload generation of FMKe is handled by Basho Bench, an open source workload generation component. This component reads a configuration file that includes a workload specification and additional data required to perform the benchmark. One key aspect is the number of client threads - each must be able to emulate an interaction of a user with the system through a browser, since operations are performed via HTTP, and the duration of the benchmark is equally configurable, with a granularity of minutes.

The FMKe application server contains several REST endpoints that are used by clients to send operations, and all of the workload operations can be encoded in this way. For instance, to perform a *get pharmacy prescriptions* operation on a pharmacy with id *5*, a client will submit an HTTP GET request to the relative URL "/pharmacies/5/prescriptions".

## 3.5 Compatibility with Multiple Systems

As detailed in the previous sections and illustrated in figure 3.2, FMKe's application server interacts with the storage component through a database driver. We have designed a simple interface that is generic for key value stores, so any written driver must follow the specification of the interface:

- *init*(): a setup function, where the driver module should read information from configuration files and attempt to establish a connection to all available database servers.

- *stop*(): a generic teardown function to gracefully stop the driver module.
- *start_transaction*(*Context*): starts a transaction within a certain transactional context. Key-value stores that do not have transactional support may return an empty value.
- *commit_transaction*(*Context*): commits a transaction within a certain transactional context. Key-value stores that do not have transactional support may return an empty value.
- *get*(*Context*, *Key*, *KeyType*): Given a certain transactional context, fetches a key from the database. *KeyType* contains information about what type of entity that is being retrieved, but that information can also be inferred through *Key*. It is necessary to know what entity type is being fetched in order to later convert it to an application level record (akin to the behaviour of an Object-Relational Model). Non-transactional key-value stores can safely ignore the *Context*.
- *put*(*Context*, *Key*, *KeyType*, *Value*): Given a certain transactional context, adds a key-value pair to the database. *KeyType* contains information about what type of entity is being retrieved, but that information can also be inferred through *Key*. Non-transactional key-value stores can safely ignore the *Context*.

If any developer is able to write a driver module for a particular database that follows the previous interface, then FMKe will be able to benchmark that database. So far we have support for AntidoteDB and Riak, but there is ongoing work to support Redis, ETS (a built in key value store for Erlang) and the Lasp programming system (the latter as a part of a Google Summer of Code project [Gso]).

## 3.6 Experimental Evaluation

With ongoing work to support multiple distributed key value stores, we can begin experimenting the benchmark on all supported storage systems and see if the results we obtain correspond to our assumptions. This would also require running YCSB to see if there is a discrepancy (and its significance) in the benchmark results compared to FMKe. Over time, if we add support for a significant number of distributed key value stores, a performance ranking could also be extracted from the benchmark results.

FMKe was already used as a prototype in early 2017 to benchmark AntidoteDB as part of a final evaluation milestone of the SyncFree European Research Project. The evaluation took place in Amazon Web Services using *m3.xlarge* instances which have 4 vCPUs, 15GB RAM and 2x40GB SSD storage. The biggest test case used 36 AntidoteDB instances spread across 3 data centers (Germany, Ireland and United States), 9 instances of FMKe and 18 instances of (former Basho Bench) Lasp Bench that simulated 1024 concurrent clients performing operations as quickly as possible. Before the benchmark, AntidoteDB

was populated with over 1 million patient keys, 50 hospitals, 10.000 doctors and 300 pharmacies.
These preliminary results were published in the PaPoC workshop in April 2017 [Tom+17].

## 3.7 Planning and Scheduling

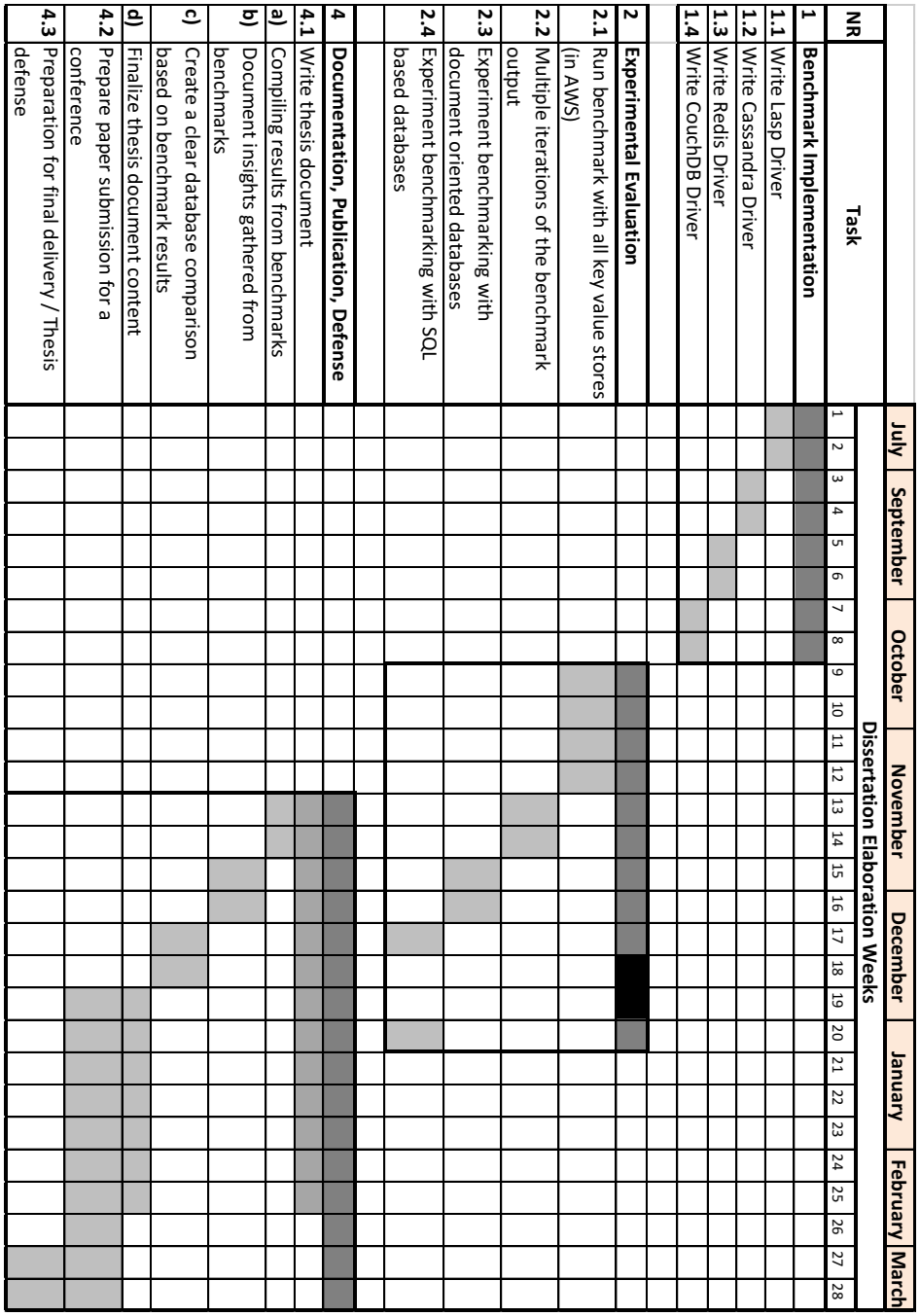


Figure 3.3: Planned scheduling of work tasks until final delivery date

As seen in the diagram, the first planned task is to proceed adding more database drivers to the benchmark prototype. Once this is complete we will be able to use the benchmark to finally compare a representative set of different databases. This will potentially yield very interesting results, and it could be interesting to contrast the results of our benchmarks compared to the YCSB and YCSB++ benchmarks. Different results, if the

drivers are implemented correctly will dictate that indeed the synthetic state of the art benchmarks was affecting results, and thus validate our work. Some adjustments to the output format and plots generated by the benchmarks are predicted, so two additional weeks are allocated towards polishing the output.

We also plan to investigate if any useful data can be extracted by executing the benchmark with other types of databases. This will require the creation of new interfaces for document oriented and SQL based databases, and would be an additional contribution if we were able to approximate the TPC benchmark values for existing SQL databases, for example.

Since we plan to make it as simple as possible to adopt this new benchmark, a set of tools will also be required that are able to simplify the configuration, deployment and execution of benchmarks.

The final step of the process is to document every finding and write the final thesis document, and this will include preparing a paper submission for a computer systems conference (to be determined later).

## 3.8 Summary

This works aims at improving the state of the art for benchmarking distributed key-value stores. A clear outline was drawn between relational databases and No-SQL databases, which served as a basis for discussing some of the properties of these databases under replicated scenarios. We've seen that relational databases, to enforce all of the constraints they allow over the data model require synchronization, and in replicated scenarios this situation is not ideal. This problem is only amplified in *geo-replicated* environments.

Available industry benchmarks were analysed and some intuition was given about why the most used benchmarks model real applications. This is because a realistic application in a testing environment is a better indicator of performance than a simple, synthetic workload. Real world applications will have data access and data writing patterns closer to what is emulated by realistic benchmarks, thus proving that realistic benchmarks are the most appropriate evaluation tools. Next, we discussed why the benchmarks from the Transaction Processing Council, despite emulating real applications, are not appropriate to benchmark key value stores.

We presented FMKe, a novel benchmark with a realistic workload component that was adapted from a system that manages the patient prescriptions at a national level in Denmark. We argument that our architecture is modular and is easily extensible by other developers wishing to evaluate other databases that are currently not supported. Our plans for the future include widening our support for the most used key value stores that support persistent data storage (because this fits the use case better), and then perform a comparative evaluation from all storage systems. Comparing to existing benchmarks such as YCSB is also essential to verify if in fact there is a discrepancy between realistic and synthetic benchmarks in the particular case of distributed key value stores.

# Bibliography


[Tpca] *A NoSQL TPC-W benchmark with a Cassandra interface - GitHub*. https://github.com/PedroGomes/TPCw-benchmark. Accessed: 2017-07-10.

[Apa] *Apache Cassandra - The Cassandra Query Language (CQL)*. http://cassandra.apache.org/doc/latest/cql. Accessed: 2017-07-09.

[Nosa] *Apache CouchDB*. https://couchdb.apache.org. Accessed: 2017-07-10.

[Bac73] C. W. Bachman. "The programmer as navigator." In: *Communications of the ACM* 16.11 (1973), pp. 653–658.

[BK14] P. Bailis and K. Kingsbury. "The Network is Reliable." In: *Commun. ACM* 57.9 (Sept. 2014), pp. 48–55. ISSN: 0001-0782. DOI: 10.1145/2643130. URL: http://doi.acm.org/10.1145/2643130.

[Bre17] E. Brewer. *Spanner, TrueTime and the CAP Theorem*. Tech. rep. 2017.

[CB74] D. D. Chamberlin and R. F. Boyce. "SEQUEL: A structured English query language." In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. ACM. 1974, pp. 249–264.

[Coc] *CockroachDB - Mapping Table Data to Key-Value Storage*. https://www.cockroachlabs.com/blog/sql-in-cockroachdb-mapping-table-data-to-key-value-storage. Accessed: 2017-07-09.

[Cod70] E. F. Codd. "A relational model of data for large shared data banks." In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[Coo+08] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s hosted data serving platform." In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1277–1288.

[Coo+10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking cloud serving systems with YCSB." In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[DeC+07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store." In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[GL02] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[Gso] *Implementing a Real World Application in the Lasp Programming Language - A project for the Beam Community*. https://summerofcode.withgoogle.com/projects/#6064361486417920. Accessed: 2017-07-10.

[Sqla] *MariaDB*. https://mariadb.com. Accessed: 2017-07-10.

[Mer17] Merriam-Webster Online. *Merriam-Webster Online Dictionary*. Database. 2017. URL: http://www.merriam-webster.com.

[Sqlb] *Microsoft SQL Server*. https://www.microsoft.com/en-us/sql-server/default.aspx. Accessed: 2017-07-10.

[Sqlc] *MySQL Enterprise*. https://www.mysql.com. Accessed: 2017-07-10.

[Nosb] *NoSQL Key Value Database - Riak KV - Basho*. http://basho.com/products/riak-kv/. Accessed: 2017-07-10.

[Sqld] *Oracle 12c*. http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html. Accessed: 2017-07-10.

[Pat+11] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. "YCSB++: benchmarking and performance debugging advanced features in scalable table stores." In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM. 2011, p. 9.

[Sqle] *PostgreSQL*. https://www.postgresql.org. Accessed: 2017-07-10.

[Tpcb] *Python-based framework of the TPC-C OLTP benchmark for NoSQL systems - GitHub*. https://github.com/apavlo/py-tpcc. Accessed: 2017-07-10.

[Sil+97] A. Silberschatz, H. F. Korth, S. Sudarshan, et al. *Database system concepts*. Vol. 4. McGraw-Hill New York, 1997.

[Tom+17] G. Tomás, P. Zeller, V. Balegas, D. Akkoorath, A. Bieniusa, J. a. Leitão, and N. Preguiça. "FMKe: A Real-World Benchmark for Key-Value Data Stores." In: *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC'17. Belgrade, Serbia: ACM, 2017, 7:1–7:4. ISBN: 978-1-4503-4933-8. DOI: 10.1145/3064889.3064897. URL: http://doi.acm.org/10.1145/3064889.3064897.

[Tpcc] *Transaction Processing Council - TPC C Benchmark Specification Document*. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf. Accessed: 2017-07-08.

[Tpcd] *Transaction Processing Council - TPC E Benchmark Specification Document*. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-e_v1.14.0.pdf. Accessed: 2017-07-08.

[Tpce] *Transaction Processing Council - TPC W Benchmark Specification Document*. http://www.tpc.org/tpc_documents_current_versions/pdf/tpcw_v2.0.0.pdf. Accessed: 2017-07-08.