Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Edvard Viggaklev Bakken

Evaluating FoundationDB as a replacement for MongoDB

An emerging NOSQL system with ACID transactions at its core

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
May 2019

NTNU
Norwegian University of Science and Technology

Edvard Viggaklev Bakken

Evaluating FoundationDB as a replacement for MongoDB

An emerging NOSQL system with ACID transactions at its core

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
May 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Is it possible to replace several different database types with a single database? The newly open-sourced FoundationDB promises just that, without compromising on integrity or performance. It claims to provide a minimalist storage engine, with strong guarantees, that can be molded to fit any data model. It could possibly reduce the complexity and costs of any large project, by acting as a single replacement for several systems at once. In the fall of 2018, FoundationDB released a plugin, the Document Layer, that claims to act as a plug-and-play replacement for MongoDB in existing applications.

This thesis examines the architecture and features of FoundationDB and the Document Layer, uncovering their strengths and weaknesses. Additionally, the system is compared to MongoDB, a well-established and popular NOSQL database. Preliminary benchmarks are performed, and the results are used to guide the creation of a custom benchmarking tool, specialized for the two systems. The intention is to evaluate FoundationDB's potential as a replacement for existing systems.

The examination shows that FoundationDB provides a highly scalable and robust key-value store, stripped of supplemental features. It has fully ACID-compliant transactions, providing much stronger guarantees than most competitors, including MongoDB. FoundationDB exposes an extensive API, allowing users to tailor the system to their needs. However, the newly released plugin is deemed not mature enough to replace MongoDB. Performance results were somewhat disappointing, and a wide range of benchmarking workloads revealed flaws that must be resolved before the Document Layer can be considered as a true competitor.

Sammendrag

Er det mulig å erstatte flere databasetyper med én enkelt database? FoundationDB, som nylig ble sluppet som åpen kildekode, lover å tilby akkurat dette, uten å måtte ofre integritet eller ytelse. Den påstår å kunne levere en minimalistisk databasemotor, med sterke garantier for pålitelige transaksjoner, som kan tilpasses enhver datamodell. Den kan potensielt redusere kompleksiteten og kostnadene i ethvert stort prosjekt, ved å fungere som en samlet erstatning for flere systemer på en gang. Høsten 2018 lanserte FoundationDB en utvidelse til databasen, *FoundationDB Document Layer*, som skal kunne være en "plug-and-play"-erstatning for MongoDB i eksisterende systemer.

Denne masteroppgaven undersøker akitekturen og nøkkeltrekkene ved FoundationDB og Document Layer, for å avdekke deres sterke og svake sider. Systemet sammenlignes med MongoDB, en veletablert og populær NOSQL-database. Det gjennomføres innledende tester, og resultatene benyttes i utviklingen av et spesialisert test-verktøy for de to systemene. Intensjonen er å evaluere FoundationDB sitt potensiale som en erstatning for eksisterende systemer.

Undersøkelsen viser at FoundationDB tilbyr en robust og skalerbar *key-value*-database, uten unødvendig ekstra funksjonalitet. Den har transaksjoner som oppfyller ACID-kravene, noe som gir sterkere garantier enn mange andre utfordrere, inkludert MongoDB. FoundationDB tilbyr et ekstensivt API som tillater brukere å tilpasse systemet til deres behov. Den nylig lanserte utvidelsen virker dog ikke å være moden nok til å erstatte MongoDB. Ytelsestester avslørte noe skuffende resultater, og det ble oppdaget flere feil og mangler som må utbedres før Document Layer kan anses som en reell utfordrer.

Preface

This Masters Thesis was written for the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU), as the final deliverable for the Master of Science in Computer Science program. The research was conducted in the spring of 2019 by Edvard Viggaklev Bakken, under the supervision of Svein Erik Bratsberg. The thesis builds upon a preliminary, exploratory project, which was completed in December 2018.

Acknowledgements

First of all, I'd like to thank my supervisor Svein Erik Bratsberg for his kind and valuable assistance throughout this project. He kept me on the right track when the volatility of young, emerging technologies attempted to derail this project more than once.

I'd also like to thank Abakus, the student union for Computer Science and Communication Technology; they've created a wonderful social environment for their students, and helped me persevere throughout this degree. I'd like to give extra thanks to the Web Committee, which taught me everything I know about programming.

Finally, I'd like to thank my family, and the friends I've made along the way.

Contents

Abstract i

Sammendrag ii

Preface iii

Acknowledgements iii

Table of Contents v

List of Tables viii

List of Figures viii

Glossary and abbreviations ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research goals . . . 2
- 1.3 Caveats . . . 2
- 1.4 Structure of thesis . . . 2

2 Background 3
- 2.1 Background theory and definitions . . . 3
  - 2.1.1 Transactions . . . 3
  - 2.1.2 The CAP theorem . . . 4
- 2.2 NOSQL systems . . . 4
  - 2.2.1 Categories of NOSQL databases . . . 5
- 2.3 Document stores . . . 5

3 Foundation DB 7
- 3.1 History . . . 7
  - 3.1.1 Early beginnings and mission . . . 7
  - 3.1.2 Acquisition by Apple . . . 7
  - 3.1.3 Re-release of the software as an open source project . . . 8
  - 3.1.4 Technical developments . . . 8
- 3.2 Architecture . . . 8
  - 3.2.1 Client . . . 10
  - 3.2.2 Distributed key-value store . . . 10
- 3.3 Features . . . 12
  - 3.3.1 The foundation . . . 12
  - 3.3.2 Operations, concurrency and performance . . . 12
  - 3.3.3 Fault tolerance . . . 12
- 3.4 Anti-features and drawbacks . . . 13
  - 3.4.1 Excluded features . . . 13
  - 3.4.2 Technical issues . . . 13
- 3.5 Current limitations . . . 13
- 3.6 Layers in FoundationDB . . . 13
  - 3.6.1 The idea behind layers . . . 14
  - 3.6.2 Capabilities of layers . . . 14
  - 3.6.3 Design and implementation . . . 14
    - 3.6.3.1 Example of data model mapping . . . 15
    - 3.6.3.2 Example of a simple index implementation . . . 16
- 3.7 FoundationDB Document Layer . . . 16

3.7.1 Overview . . . 16
3.7.2 Key differences to MongoDB . . . 17
3.7.3 Deployment and usage . . . 17

4 MongoDB **19**
4.1 What is MongoDB? . . . 19
4.2 Key features . . . 19
4.2.1 Structures and query operations . . . 19
4.2.2 Scalability and availability . . . 20
4.2.3 Storage engines . . . 20
4.2.4 Transactions . . . 20
4.3 Comparing MongoDB and FoundationDB . . . 21

5 Preliminary benchmarks **23**
5.1 FoundationDB's built-in test framework . . . 23
5.1.1 Drawbacks . . . 23
5.1.2 Benefits . . . 24
5.1.3 Method . . . 24
5.2 YCSB . . . 24
5.2.1 Standard workloads . . . 25
5.2.2 Configuration, installation and application versions . . . 25
5.2.3 Benchmarking FoundationDB . . . 26
5.2.4 Benchmarking MongoDB . . . 26
5.2.5 Benchmarking MongoDB with stricter write concern . . . 26
5.2.6 Benchmarking FoundationDB Document Layer via MongoDB . . . 27

6 Results of preliminary tests **29**
6.1 Integrated test suite . . . 29
6.2 YCSB benchmarks . . . 29
6.2.1 Workload A . . . 30
6.2.2 Workload B . . . 31
6.2.3 Workload E . . . 32
6.2.4 TimeSeries Workload A . . . 33

7 Discussion of preliminary results **35**
7.1 Integrated test suite . . . 35
7.2 YCSB benchmarks . . . 35
7.2.1 Workload A . . . 35
7.2.2 Workload B . . . 36
7.2.3 Workload E . . . 36
7.2.4 TimeSeries Workload A . . . 36
7.2.5 Configurations . . . 37
7.3 Limitations and caveats . . . 37
7.4 Findings . . . 37

8 Research method **39**
8.1 Document store benchmarking tool . . . 39
8.1.1 Technical considerations from preliminary findings . . . 39
8.1.2 Implementation . . . 40
8.1.3 Installation and application versions . . . 40
8.1.4 Limitations and caveats . . . 41
8.2 Workloads . . . 41
8.3 Method and execution . . . 42

9 Results **43**
9.1 Workloads . . . 43
9.1.1 Workload A . . . 43
9.1.2 Workload B . . . 44

9.1.3 Workload C . . . 45
9.1.4 Workload D . . . 45
9.1.5 Workload E . . . 46
9.1.6 Workload F . . . 47
9.2 Database configurations . . . 47
9.2.1 FoundationDB Document Layer . . . 47
9.2.2 Standard MongoDB . . . 47
9.2.3 Transactional MongoDB . . . 48

10 Discussion and evaluation **49**
10.1 Evaluation of workloads . . . 49
10.1.1 Workloads A, B, and C . . . 49
10.1.2 Workload D . . . 49
10.1.3 Workload E . . . 50
10.1.4 Workload F . . . 50
10.2 Evaluation of configurations . . . 50
10.2.1 FoundationDB Document Layer . . . 50
10.2.2 Standard MongoDB . . . 51
10.2.3 Transactional MongoDB . . . 51
10.3 Limitations and caveats . . . 51
10.3.1 Performance and environments . . . 51
10.3.2 Explicit transactions and future updates to the Document Layer . . . 51
10.3.3 Insight and scope . . . 52

11 Conclusion and further work **53**
11.1 Conclusion . . . 53
11.2 Related work . . . 54
11.2.1 MongoDB and DynamoDB transactions . . . 54
11.2.2 ArangoDB . . . 54
11.3 Further work . . . 54

Bibliography **55**

Appendices 61

A Native test suite output **61**

B YCSB benchmark outputs **62**
B.1 FoundationDB benchmark . . . 62
B.2 MongoDB benchmark . . . 70
B.3 MongoDB benchmark with stricter write concern . . . 78
B.4 FoundationDB Document Layer benchmark via MongoDB . . . 86

C Source code for custom benchmarking tool **94**
C.1 README.md - fdb-benchmarks . . . 94
C.1.1 First time setup . . . 94
C.1.2 Regular setup . . . 95
C.1.3 Usage . . . 95

D Custom tool benchmark outputs **96**
D.1 FoundationDB Document Layer benchmark . . . 96
D.2 Standard MongoDB benchmark . . . 98
D.3 Transactional MongoDB benchmark . . . 100

List of Tables

6.1 Key results from the native test framework of FoundationDB 29

List of Figures

3.1 A simple overview of FoundationDB's architecture, as described by FDB documentation[1] 9
3.2 A simple JSON object, containing to persons and their attributes. 15
3.3 Tuples representing the deserialized JSON object. 16
3.4 Keys and values, representing the tuples seen in figure 3.3 16

6.1 Throughput when loading workload A 30
6.2 Throughput when running workload A 30
6.3 Average latencies when running workload A 31
6.4 Throughput when running workload B 31
6.5 Average latencies when running workload B 32
6.6 Throughput when running workload E 32
6.7 Throughput when running TimeSeries workload A 33

9.1 Average throughput for each database configuration, across all workloads 43
9.2 Average throughput when running workload A 44
9.3 Average throughput when running workload B 44
9.4 Average throughput when running workload C 45
9.5 Average throughput when running workload D 45
9.6 Average throughput when running workload E 46
9.7 Average throughput when running workload F 46
9.8 Average throughput of all workloads for runner `fdbdl`, i.e. the FoundationDB Document Layer . 47
9.9 Average throughput of all workloads for runner `mongo3`, i.e. standard MongoDB 48
9.10 Average throughput of all workloads for runner `mongo4`, i.e. transactional MongoDB 48

Glossary and abbreviations

Term	Definition
NOSQL	Not only SQL
ACID	Atomicity, consistency, isolation, durability
CAP	Constistency, availability, partition tolerance
FDB	FoundationDB
MDB	MongoDB
KV	Key-value
API	Application Programming Interface
CRUD	Create, read, update, delete
FDB-DL	FoundationDB Document Layer
YCSB	Yahoo! Cloud Serving Benchmark
Throughput	Operations per second
tps	Transactions per second
GRV	Get Read Version
The tool	Custom benchmarking tool, written for the purpose of this research. Used to evaluate and compare FoundationDB Document Layer and MongoDB.
RMW	Read-Modify-Write

Chapter 1

Introduction

1.1 Motivation

Internet traffic has exploded in the last decade. According to Cisco it has increased by a factor of ten from 2008[2] to 2016[3]. Applications handle millions of users, and the amount of data is enormous[4]. This data must be accessible at all times, and wait times are expected to be nonexistent; traditional relational databases do not fulfill these requirements. As explained in [4], they offer many services that may be unnecessary, causing more overhead and reduced database availability. Also, their relational model may be too restrictive for the kind of data global applications are storing today. These drawbacks led to the development and growth of NOSQL databases.

NOSQL databases are distributed storage systems that differ from traditional SQL systems in several ways[4]. They provide high availability and performance, horizontal scaling, and schema-free data storage. NOSQL systems rarely include complex query languages and relational structures, and often forego serializable consistency for *eventual consistency*[5]. All these features and anti-features are great for modern applications with millions of users, where unstructured or semi-structured data is stored and fetched hundreds of thousands of times per second.

However, NOSQL systems do not come without drawbacks. Some applications may require traditional query languages, and giving up ACID-compliant transactions means that NOSQL systems must employ other methods to ensure key database properties, such as consistency and durability. Also, most NOSQL databases can be categorized based on their features and data model[4]. While not a drawback in and of itself, this means that choosing a database system also means selecting a certain data model, not just the optimal storage engine for a certain application.

FoundationDB is a newly open-sourced NOSQL system[6, 7, 8], aiming to provide the best of both worlds by taking a different approach to the problem. It provides ACID-compliant transactions while remaining highly scalable, and claims to deliver ”industry-leading performance”[7, 9, 10]. FDB is an ordered key-value store, with a focus on providing a minimalist core. Through custom *layers* and an API exposing the KV-store, the user is able to add features, structures, data models, and query languages as needed.

One such layer is the FoundationDB Document Layer[11, 12], which uses the MongoDB protocol. It allows users to use the MongoDB API and existing client bindings, while persisting data in the FoundationDB key-value store. It retains all the strong guarantees found in FoundationDB, such as ACID-complaint transactions and sequential consistency, while providing a document store that is compatible with all official MongoDB drivers.

This project seeks to investigate the viability of the FoundationDB Document Layer as a replacement for MongoDB, in terms of performance. The Document Layer brings many perks that may change the concept of what NOSQL systems can and can’t be, making it an interesting test subject. Additionally, should the Document Layer prove to be viable, developers may exploit FoundationDB’s adaptable data structure by using FoundationDB as a single, unified replacement for several systems at once.

1.2 Research goals

The following research goals were defined for this thesis, based on the motivation, and the background presented in chapter 2:

- Explore the architecture and features of FoundationDB, and how it differs from other NOSQL databases.
- Evaluate the FoundationDB Document Layer, and compare FoundationDB and the Document Layer to MongoDB, a popular competitor.
- Benchmark and compare the performance of MongoDB and the Document Layer, using a a custom benchmarking tool.
- Review the viability of the Document Layer as a replacement for MongoDB in existing systems.

1.3 Caveats

While both databases evaluated in this project are intended for highly distributed use, this thesis will focus on single core database environments. This is mainly due to limited time and resources, but also an effort to reduce the complexity of the study.

Additionally, the insight, theory, and observations presented on MongoDB and FoundationDB rely heavily on documentation and forum posts published by the developers, as well as GitHub issues and pull requests. From time to time, the contents of such publications may be slightly lacking. As such, the presented material may be somewhat temporal, and affected by the author's understanding of the documentation.

1.4 Structure of thesis

The remainder of this thesis has been divided into the following chapters:

Chapter 2: Background presents the premise of transactions, the ACID properties, and the CAP theorem. Also, a brief introduction to NOSQL systems is given, with further details on a type of NOSQL systems known as Document Stores.

Chapter 3: FoundationDB presents the background of FDB, and how it has developed over the past years. The architecture of the database is explained in detail, as well as the database client. The main feautures of FDB are discussed, as well as current limitations of the system. *Layers*, a customization feature of FoundationDB, is presented, and the Document Layer is discussed.

Chapter 4: MongoDB briefly introduces MongoDB, and some of its key features. It also includes a comparison of MongoDB and FoundationDB.

Chapter 5: Preliminary benchmarks presents the methods used to perform preliminary benchmarks of the two database systems.

Chapter 6: Results of preliminary tests presents the most interesting results found in the initial benchmarks

Chapter 7: Discussion of preliminary results analyzes the results found in these initial benchmarks, and discusses which considerations must be made when thoroughly testing the two database systems.

Chapter 8: Research method introduces a custom benchmarking tool, which was written from the ground up to compare the two systems, and the considerations made when designing it.

Chapter 9: Results presents some key results that were found when using the custom tests

Chapter 10: Discussion and evaluation analyzes the results, relating them to the insight found in previous chapters.

Chapter 11: Conclusion presents a final conclusion, and outlines related and further work.

Chapter 2

Background

In the first section of this chapter, theory and explanations around the concepts of transactions and ACID properties is presented. The CAP theorem is also discussed. The second section provides an explanation of the concept of NOSQL databases, while the final section dives deeper into the category of NOSQL systems known as *document stores*. These sections are based on the author's own findings, from a project report written in the fall of 2018.

2.1 Background theory and definitions

This section briefly presents the concept of ACID-compliant transactions, as well as the CAP theorem. Both concepts are quite relevant when discussing the features and drawbacks of NOSQL systems, and provide helpful context when evaluating the premise of FoundationDB.

2.1.1 Transactions

The term *transaction* stems from contract law, where two parties negotiate a deal that they both agree upon[13]. Sometimes the parties may be suspicious of each other, so they employ a middleman to oversee the commitment of the transaction. In terms of database systems, a transaction is "a unit of consistent and reliable computation"[13], but the concept shares many similarities with that of a contract negotiation: a client and a database system negotiate or agree upon changes to the database, and how they should be performed.

In general, a transaction can be seen as a sequence of read and write operations that should be performed together, combined with related computational steps.

ACID properties

Transactions have four key properties, justifying their claim as consistent and reliable computations[14]. These four properties, known as ACID, are responsible for guaranteeing the reliability of transactions, and are seen as essential features of a dependable database system[15].

Atomicity An atomic transaction is seen as a single unit of work[14]. Either the entire transaction succeeds, or none of it. If a failure occurs halfway through the computations done by a transaction, the system must be rolled back to its previous state. This ensures that the database is not left in an inconsistent state, due to incomplete operations.

Consistency The property of consistency refers to the correctness of a transaction; it should map one legitimate database state to another[14]. Transactions should not read or write dirty data from other transactions, and should not commit any writes before the entire transaction is completed. Any data written must be valid, according to all defined constraints, and operations must be performed accurately, correctly, and with validity.

Isolation Isolating transactions ensures that their end result is unaffected by concurrent execution of other transactions[14]. Transaction A should not be able to see the effects of transaction B until B is committed, and the end result of executing concurrent transactions should be the same as if they were executed sequentially.

Durability Durability ensures that once a transaction commits, the results are permanently stored to the database[14]. The database system must guarantee that results survive potential system failures. The system must have recovery procedures in place, or replicate data for greater fault tolerance. For transactions, this usually means that data must be persisted or durably logged *before* clients are notified of the transaction's success.

2.1.2 The CAP theorem

Originally introduced as the CAP principle, this theorem explains some competing conditions of a distributed system[5]. It refers to three desirable properties: consistency, availability and partition tolerance. The term consistency is not the same concept as the 'C' in ACID, it refers to the fact that replicated copies of data should be consistent with each other. Availability means that the system should either successfully handle all received read and write requests, or notify the client of any failures. Partition tolerance refers to the systems ability to continue operations, even if network failures lead to the system being *partitioned.*

The theorem states that in the event of a failure, it's impossible to guarantee all three properties at once[16]. In fact, one could argue that the choice is between consistency and availability alone, as abandoning partition tolerance is practically the same as abandoning availability. A system cannot be both consistent and available, but not partition tolerant, during a partition.

In general, traditional relational systems will opt for consistency over availability, due to their ACID properties. However, if a weaker consistency level is acceptable, systems can achieve greater performance by abandoning (or relaxing) the property of consistency.

2.2 NOSQL systems

As the amount of data stored by truly global applications grew immensely in the late '00s, it became clear that relational systems were unfit when working with this new kind of information[4]. Restrictive data models and unnecessary services made companies investigate and develop alternative database systems. These new distributed systems were schema-less, allowing for unstructured and semi-structured data. They usually disregarded powerful query languages, such as SQL, and often opted for greater availability and performance over ACID properties, strict guarantees, and consistency. They were termed NOSQL systems: Not Only SQL.

NOSQL systems have several key characteristics, making them optimal for highly distributed applications, where data is accessed extremely often[4]. They provide simple horizontal scalability, making it possible to easily expand the system by adding additional machines or processes. They deliver high availability by replicating data over several nodes, making it possible to handle more requests concurrently. NOSQL systems also employ *eventual consistency*. This allows the system to defer replication of updates for later, increasing performance and availability, but reducing the consistency of the system.

As mentioned, they are often schema-less, and many focus on providing an API for CRUD operations, rather than a powerful query language[4]. The data stored in NOSQL systems rarely require complex joins or aggregations, making such tools unnecessary.

As previously discussed, NOSQL systems often adopt eventual consistency[5]. In terms of the CAP theorem, most choose availability over consistency. NOSQL systems are often used in applications that require continuous availability; it is more acceptable to read potentially stale data than to read no data at all. Due to their focus on performance and availability over consistency, many NOSQL systems sacrifice transactions and ACID properties. These systems are instead described as BASE: Basically Available, Soft state, Eventual consistency.

2.2.1 Categories of NOSQL databases

Most NOSQL systems can be split into four main categories, based on the data model they employ: key-value stores, column-based systems, graph-based systems, and document stores[4]. Some systems may not fit these categories; these can often be described as hybrids.

A key-value store has a quite simple datamodel, as the name implies, and allows fast access by the key to a value[17]. The value can typically store any kind of data or structure. Many of these systems lack query languages, but provide great architectures for efficient distribution and access of data. DynamoDB, Redis and Voldemort are all well known KV-stores[18].

Column-based systems, such as Cassandra, HBase and Bigtable, can be seen as two-dimensional key-value stores[19, 20]. Data is organized by tables, but is grouped by columns instead of rows[17]. A key can consist of the table name, the row key, the column identifier, and a timestamp, for instance. Columns are often grouped by *families*: columns that are related or often used together.

Graph-based systems use nodes and edges to represent and store data[4]. The graph is used to describe relationships between different data points[21]. The system might not provide direct access to all nodes, but allows for efficient processing of certain queries, such as steps from one node to another. Neo4j is the most prominent graph-based system[22].

Document stores, the main focus of this thesis, are expanded upon in the following section.

2.3 Document stores

This type of NOSQL system uses well-known formats, such as JSON, to describe data as documents[17]. Such documents are reminiscent of complex objects or XML documents, and are specified as self-describing data. These systems often employ efficient index structures to quickly access the data stored within documents. This is done by building an index on some commonly accessed field within the documents. Popular examples include MongoDB and Couchbase[23].

Documents are often grouped in *collections* of similar documents. While object structures may be similar to relational systems, document stores typically do not require a schema to be defined for the collection; each document is allowed to have different attributes, and new documents may contain new elements that aren't found in any other document in the collection. Thus, index structures will only contain documents that contain the indexed field.

Chapter 3

Foundation DB

FoundationDB is an open source distributed database, with a focus on providing a high-performance [9, 10] minimalistic core that can be adapted to any use case. At its heart, FDB is an ordered key-value store, which employs ACID transactions for all operations [7]. Users are able to use any data model or query language they desire, through customizable plugins known as *layers*[24].

The team behind FoundationDB has developed and released several layers themselves [25]. The capabilites of FDB related to document storage, a focus point of this thesis, will be tested using the FoundationDB Document Layer, presented in section 3.7.

This chapter provides a description of the history of FoundationDB as a database and as a company. Furthermore, the architecture of FDB is discussed in detail, together with the key features of FDB. The chapter also includes sections on *anti-features*, i.e. features that were deliberately not included, known limitations and drawbacks. Finally, the chapter elaborates on the concept of layers, and provides a brief presentation of the FoundationDB Document Layer. These sections are based on the author's own findings, from a project report written in the fall of 2018.

3.1 History

This section details the history of FoundationDB, from 2009 to 2018. It includes some reactions from the user base as the story progresses, and concludes with a summary of how the software itself has evolved since it first released.

3.1.1 Early beginnings and mission

In 2009 the company behind FoundationDB was founded[26]. The creators believed that the split between NOSQL databases and transactions was unnecessary[15], and that the issues were based on misunderstandings of the CAP theorem and NOSQL as a concept[16]. A NOSQL database does not require you to relinquish the concept of ACID-compliant transactions, but the creators of FDB felt that many were under the impression that this was the case. In 2012 the first alpha version of FDB was released[27], providing the best of both worlds with highly scalable, ACID-compliant transactions in a distributed key-value store.

FoundationDB also published a white paper to go along with their database, *The Transaction Manifesto*[15]. It further explains the misunderstandings around the usefulness of transactions, and argues why transactions are both a requirement and a cornerstone for future development of NOSQL databases. FDB grew and evolved over the following years, releasing both a public beta[27] and a general release in 2013[28].

3.1.2 Acquisition by Apple

In 2015, two years after the initial release, news broke of Apple acquiring the company behind FoundationDB, and the technology itself[26, 29]. All software from the company was removed from GitHub, and further down-

loads were no longer possible[30, 31]. A notice was posted on their website outlining a shift in direction and mission[32], and users were left in disarray. The sudden halt in further development and lack of support caused a huge backlash in the community[31], and users scrambled to find a fork of the project that was relatively up to date.

After the acquisition, Apple said nothing of their plans for the technology. There was great uncertainty in the community, with many concerned that Apple bought the company for their developers and their technology, not for the software itself[30]. This would have led to the death of the software, potentially disappointing many users. However, there was also a rumour floating around that Apple planned to replace their Cassandra implementation, used for ”iMessage, iTunes passwords and a bunch of stuff”, with FoundationDB[31].

3.1.3 Re-release of the software as an open source project

In April 2018 Apple released a blog post announcing the return of FoundationDB as an open source software, proving that they had kept developing the technology over the past three years[8][33]. This opened the database up for use to other companies, and allowed the community to help develop the software even further. The news were met with rejoice among users, who were happy that the promising software was resurrected[34].

Whether or not Apple actually decided to implement FDB in their own systems was unknown at this point, but the release of the FoundationDB Record Layer in January 2019 revealed that FDB is used in Apple’s CloudKit[35]. CloudKit is an iOS developer tool used for connecting an application to storage in iCloud, and FDB is used to manage ”billions of databases” for CloudKit[36].

3.1.4 Technical developments

FoundationDB has received many new features over the years, and most releases were published before the acquisition happened. Realease notes can be found in [37], [38], and on GitHub[8]. In Alpha 5 a new system for scheduling backups of snapshots of the database was introduced, and Alpha 6 brought cross platform clients and development servers.

The first beta improved FoundationDB’s distributed capabilities, by making it possible to safely remove servers from a cluster. The second beta release enabled support for databases up to 100TB, and the third released a new API to find the physical location of keys, as well as functionality to watch for changes in keys, and to cancel transactions.

The official release brought no major changes, and mainly served to move the software from beta to a general public release. Version 2 however, brought support for PHP and Go, and introduced encrypted network traffic, using TLS. The third major release, 3.0.0, brought huge performance improvements. It increased the maximum writes per second from 200 000 to 11 000 000, as well as many other upgrades.

Version 4.0.1 and beyond kept improving the performance, and added many new features, such as improved transaction handling and backup systems. The first new version after the acquisition (4.4.1) brought streaming writes, better logging of statistics, and improved reporting tools. Many of the features developed internally at Apple clearly point towards a shift in direction, with a higher focus on DevOps and enterprise use. Apple also implemented a new storage engine, further increasing performance. Version 5.2.0 marks the first release after the product was re-released to the public, and version 6.0.0 was released in November 2018.

3.2 Architecture

The architecture of a FoundationDB cluster can be split into three major parts[1, 7]. First, the most significant component would be the distributed key-value store itself. Second, the FDB client provides a minimal API for interacting with the KV store. Finally, layers provide a third and optional component to the architecture, allowing users to customize FDB to their needs[24]. Section 3.6 expands upon the concept of layers. A general and simplified overview of the system, as described by the developers themselves, can be seen in figure 3.1.

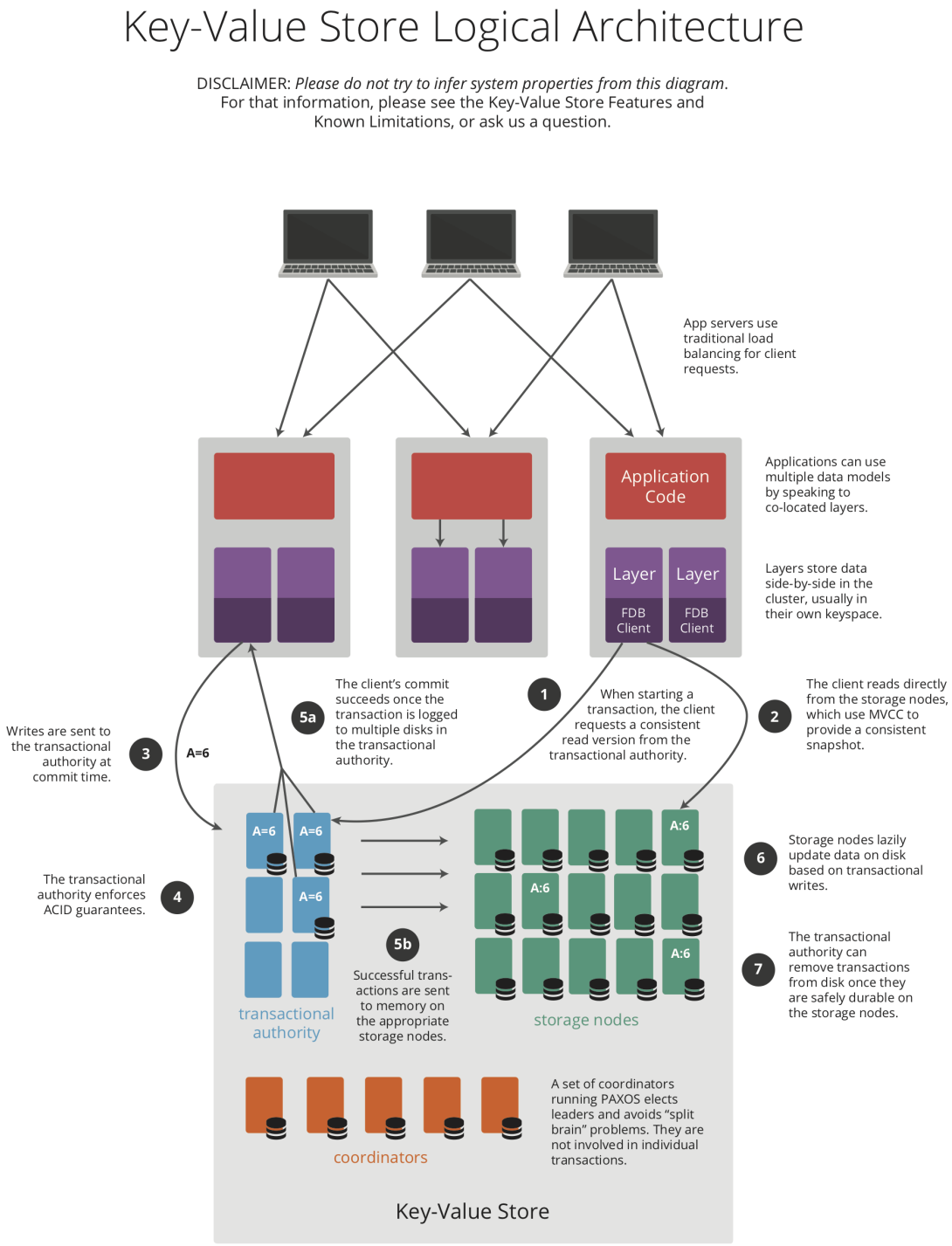

Figure 3.1: A simple overview of FoundationDB's architecture, as described by FDB documentation[1]

3.2.1 Client

User applications can talk to layers on top of clients, or directly to the clients themselves [1]. The client API is used to communicate with a cluster of FoundationDB servers. The client is used for reading data, sending writes to the cluster, and is responsible for wrapping operations in transactions. A client is able to connect to a cluster using a *clusterfile*, which contains an identifier for the cluster, and the location (IP address and port) of all *coordination servers* in the cluster [39]. This file should contain the same information for all servers and clients in the cluster [40].

Clients start every transaction by acquiring a read version. All reads during a transaction is done at that version[41]. The client sends transactions to *proxies* in the transactional authority[40], and is responsible for retrying transactions in case of conflicts [41].

3.2.2 Distributed key-value store

The main component of FoundationDB is the distributed database, which is an ordered key-value store[7]. The lexicographic ordering of keys ensures that scanning ranges of key-value pairs is efficient, and the store does not require any specific format for keys or values[42, 43]. The FDB documentation does however encourage users to keep the ordering in mind when constructing keys[42], so as to allow for fast range reads.

A FoundationDB cluster consists of many FDB servers. More precisely, each node in a cluster is an instance of the `fdbserver`-process, running on some machine[44]. This means that it's possible to run several FDB servers on a single machine. All nodes act as storage nodes, and some nodes are recruited to perform management tasks. It is possible to configure which kinds of tasks a node prefers[44].

As seen in figure 3.1, a cluster is made up of typical storage nodes, a transactional authority, and coordinators. The following sections will provide an overview of each subsystem. Further explanations and more in-depth figures can be found in [40] and [45], two forum threads where several of the developers go into more detail about the system.

Coordinators

A set of nodes known as coordinators are defined by the clusterfile[44]. Being a coordinator has negligible impact on performance, any process can be used for this role. Coordinators are not involved in committing transactions at all.

Any process or client connecting to the cluster must first ask the coordinators for information about the cluster controller[40, 41], so that they can register with the controller. Coordinators communicate a small amount of shared state, and at least half the coordinators should return the same information about the controller. If no controller exists, a connecting process will attempt to become the cluster controller. Coordinators use a modified PAXOS algorithm to elect the controller[1, 16].

Should the cluster become partitioned, FoundationDB will select the partition containing a majority of coordinators as the partition that will remain available[44]. As described in the FDB documentation, one should configure a cluster to have an odd number of coordinators, and to choose coordinators based on their physical location, so as to maximize fault tolerance[44].

Cluster controller

The cluster controller is a single server responsible for registering workers and monitoring failures[40, 41]. It is elected by the coordinators, and acts as an entry point to the cluster for both processes and clients. Any client connecting to the cluster is directed to the controller, which will send the addresses for all *proxies* (see the section on the transactional authority) in the cluster[40].

All processes joining the cluster must register with the controller, if one exists. If not, the process will attempt to become the controller[41]. The cluster controller must also determine when a process has failed, and must pass system information between all processes. Additionally, the controller will tell processes which role they should assume.

Master

The master node is recruited by the controller, and is responsible for coordinating the transactional authority[41]. The master keeps a centralized version controller, which provides commit versions for batches of transactions. It also receives information from the transaction logs and storage nodes[40]. An algorithm uses this information to dynamically distribute data across the storage nodes[46]. This information is also used to run a ratekeeper[47], which limits the transaction request rate, based on the system overhead. This limit is passed to the proxies, and is used to determine the number of transactions which can be processed, and how they should be batched.

Transactional authority

As shown in figure 3.1, the transactional authority is responsible for processing incoming transactions, ensuring that the ACID properties are upheld[1]. The authority is responsible for detecting conflicting updates; transactions execute using optimistic concurrency control, any conflicts require the transaction to be retried[48]. The authority is comprised of a set of nodes, recruited to perform different transaction processing tasks[41]. This process involves batching transactions, checking for conflicts, logging transactions and durably storing the updates[48].

Proxy nodes receive transaction requests from clients, based on a round-robin sequence [41, 40, 45]. To start a transaction, clients will first request a read version. These requests are handled in batches by proxies, who acquire the latest read version by asking the log system and all other proxies for their latest commit version. The highest of these is returned as the latest read version. A read transaction can now be handled by communicating with the storage nodes directly, while a write transaction requires the client to contact the proxies again [1, 40].

A write transaction is sent to the proxies[40, 45]. Such requests include the aforementioned read version, read and write ranges, as well as mutations. These transactions are also handled in batches. Proxies send the affected ranges to the resolvers, who check each transaction for conflicts. The proxy will then ask the master for the commit version, before pushing the mutations to the transaction logs. Once the transaction log has written the mutation to their disk, it replies to the proxy, who then returns the commit version to the client.

Resolvers have the important responsibility of conflict-checking transactions[41]. It receives read and write ranges of transactions from the proxies, and returns a conflict if it receives a key that has been written between the read and commit version of the transaction. This is done by holding the past five seconds of writes in memory.

The transaction logs are nodes that write mutations to a log on disk[41]. This enables fast commit latencies, because mutations don't have to be written to storage to be considered durably stored. Transaction logs receive commits from the proxy, and forward the data to the responsible storage servers. It typically takes a storage node several seconds to actually write these changes to disk. To avoid such long wait times, transaction logs respond to the proxy once the mutations have been written to an append only log on their local disk, which is done right after the data is passed to storage nodes.

Additionally, the proxies will periodically create empty commits, ensuring that the commit version keeps increasing[40].

Storage

Most nodes in a cluster act simply as storage nodes[41]. Using the data distribution algorithm, the master assigns storage nodes with key ranges that they are responsible for. Storage nodes will periodically read mutations from the transaction logs that are to be made durable[1, 40]. These mutations are read into memory, and eventually written to storage. At this point the storage node will notify the transaction logs that the mutation has been written to disk, so that the logs can discard the mutation. Storage nodes will also read data from other nodes and write it to their local disk, to ensure replication. All this is done in the background.

Storage nodes keep recent mutations (the past five seconds) in memory, and a copy of the data as of five seconds ago on disk[41]. If a client provides a read version that is less than five seconds old it is allowed to read from a storage node, who will scan for the data in memory before checking the disk. If a transaction consists of a read and a write on the same key, the write value will be returned.

Storage nodes can use one of two storage engines[44]. One is optimized for SSDs, and writes data to a b-tree on disk. The other is for use with spinning disks; it writes data to memory, and to an append only log that is stored on disk. This log is only read if the process is rebooted. A third storage engine, *Redwood*, is currently in development[49].

3.3 Features

As the name implies, FoundationDB is designed to provide a core for users to build their metaphorical house upon. It can be used as a fully fledged key-value store, as a storage engine for other applications, or as the basis for custom user-designed database systems. This sections outlines the key features of this core, i.e. the distributed key-value store that is FDB.

3.3.1 The foundation

FoundationDB is an ordered key-value store, which is optimized for many small reads and writes[43]. The sorted keyspace allows for fast range scans, and the system is well suited for both OLTP and OLAP workloads. All operations are accomplished using transactions, which are fully ACID across multiple machines, with high performance. The ACID-compliant transactions provide the highest available isolation, and the strongest durability. Transactions appear to happen in sequence, and all changes are stored to disk before they are considered to be committed.

Clients are also able to set watches on keys, ensuring that FDB will push notifications back to the client upon changes, without polling[43]. Finally, FDB supports atomic operations, such as ”ADD” or ”XOR”. These low-latency operations can read values, perform some change to them, and write the result, without creating conflicts with other transactions.

3.3.2 Operations, concurrency and performance

The core of FoundationDB provides automatic redundancy, partitioning, and caching through the use of horizontal scaling [43, 50]. It can run as a single process, partially using a single core, as a giant cluster, spanning across several data centers, or as anything in between. Processes and machines are easily added or removed from the cluster, and data is automatically redistributed across nodes based on data size and request loads, allowing for immediate load balancing. Requests can be redirected from busy processes to replicas on less busy machines. The system is also able to absorb work for several minutes, deferring background work for later, ensuring low latencies even when the load is far higher than usual. Caching is also handled in a distributed manner, by using the memory of the entire cluster. The cache is completely synchronized to the database, and provides all the same ACID guarantees as the key-value store itself.

To avoid having to lock data, FoundationDB uses multiversion concurrency control to provide isolated reads for transactions[43]. This optimistic approach makes sure that deadlocks can’t occur, and protects the system performance from suffering at the hands of slow or failing clients. Additionally, the threadless communications and concurrency model makes FDB capable of handling ”hundreds of thousands of in-flight requests”, without being slowed down.

3.3.3 Fault tolerance

”No single point of failure” is a key feature in FoundationDB, and the developers claim that the system has been designed to go much further than that[43, 46]. Data is replicated across several physical servers, and the system will automatically create new replicas in the case of node failures. Servers can be grouped based on locality, ensuring that replicas are distributed across machines, racks, data halls or even data centres. FDB also provides an integrated backup system, which can schedule complete snapshots of the entire database stored to a remote file system.

All these systems, including network and I/O, are rigorously tested by complex deterministic simulation tools[43, 46, 51]. Catastrophic and random failures are emulated and evaluted, so as to ensure that FDB is capable of operation even while enduring major problems and failures. [52] provides an interesting talk on these testing measures and how they affect development at FoundationDB.

3.4 Anti-features and drawbacks

By design, there are some things FoundationDB deliberately does not provide. As discussed, FDB aims to provide a minimal core; this section outlines some features that are intentionally not provided by FDB, as well as some design limitations. These are fundamental design choices, and are not likely to change in the future.

3.4.1 Excluded features

FoundationDB markets itself as a multi-model data store, while actually only providing a key-value store[6, 7]. The multi-model support comes from the use of layers, where users can construct their own models, which are mapped onto the key-value store[24]. Similarly, FDB does not provide any query language or index structures, these can both be implemented through layers[53]. Although the core is well suited for scanning large amounts of data, it does not provide any sort of analytic framework for processing the data. Both batch and real-time processing is implementable through layers.

The FDB system provides no form of access control to differentiate between clients [54]. If a client is allowed to connect, it is allowed to modify and read any data in the store. Thus, to protect the database, users must implement external protections. Finally, FDB does not support disconnected operation, where clients connect to a database for periodic synchronization, but remain available even when disconnected from the database [53]. While this is popular for many mobile applications, such as notekeeping apps, it is not directly provided by FDB, as that would sacrifice ACID properties.

3.4.2 Technical issues

As a result of the design of FoundationDB, the database faces some technical drawbacks [54]. FDB does not respond well to transactions larger than 10 million bytes; this can reduce both performance and availability. Similarly, keys and values have a max size; FDB will raise errors if keys exceed 10 000 bytes, or if values exceed 100 000 bytes. Additionally, large offsets in key selectors are resolved quite slowly, but the API provides tools to work around this.

Finally, FoundationDB does not support transactions longer than five seconds[53, 54]. FDB uses multiversion concurrency control, and must keep information about previous changes for a set period of time to check for conflicts. This could arguably be defined as both an antifeature and a technical drawback, and FDB suggests splitting workloads into smaller transactions instead.

3.5 Current limitations

Some of FoundationDB's disadvantages are not inherently related to design choices. While important to acknowledge and be aware of, these are more likely to be resolved in future releases. As previously discussed, transactions in FDB are limited to a maximum runtime of five seconds. While the concept of long-running transactions is not something that is supported, the developers are looking to increase the time limit[54].

FoundationDB has been tested and tuned with clusters up to 500 processes, with a database size of up to 100 TB[54]. Clusters larger than this may face issues when scaling out, such as sub-linear scaling, and will face higher disk-space requirements after replication and overhead. While FDB does balance reads across servers, it does not increase the replication factor of frequently accessed data[54]. This means that aggregate read performance of certain keys may be directly related to the system wide replication factor (default factor is 3).

3.6 Layers in FoundationDB

This section explains the concept of *layers*, and how they are used in FoundationDB to expand upon the features of the core database system. The section will describe what layers are capable of, how to use them, and how to design and create them.

3.6.1 The idea behind layers

FoundationDB is designed to provide minimalist storage substrate[55]. The core aims to be as strong as possible, without any additional features[24]. This was done for several reasons. First, FDB was created as a startup almost ten years ago; time and money were scarce resources, and priorities had to be made. The team opted for a lean approach to development, focusing on providing a robust and high-performance platform, while stripping away as many features as possible[52, 56].

Second, FoundationDB argues that providing a complete database system creates tougher decisions for users [24]. If a database system includes a storage engine, an overlaid technology, a specific data model and a tailored query language then finding the perfect database for all relevant use cases becomes increasingly difficult. In fact, it could lead to systems where several database systems are necessary. Following the trend of microservices, FDB provides a separated data storage technology with an adaptable data model.

3.6.2 Capabilities of layers

Supplementary components, such as specialized data models or query languages, are implemented as layers[53, 24]. A layer communicates with the FBD client API to access and manipulate data, create and commit transactions, set watches or waits, or any other communication with the database[57]. This allows for endless possibilities. The ordered key-value model can be adapted to any data model, and the API makes it possible to implement query languages, analytic frameworks, additional monitoring and much more[53].

As mentioned, layers provide countless opportunities. Through implementing additional data models, structures, and query languages, FoundationDB's core can act as a common storage system for all the needs of an application. In other words, by combining layers FDB could act as a single replacement for applications where several different database systems are in use today. It is even possible to implement relational models and SQL through layers[42].

Another example would be the use of indexes, which are not provided by FoundationDB. They are quite simple to implement through layers, however. The documentation for FDB provides an example for this[24], which is detailed in section 3.6.3.2. In summary, layers are a way to mold FDB to the needs of the user, while the core provides robust structure and performance.

3.6.3 Design and implementation

Layers are an essential concept for the FoundationDB system, and implementing them is quite simple. In fact, any code or service interacting with the FDB client API can be defined as a layer. The API exposes methods for defining transactions, accessing or updating data and keys, key selectors, watches, namespace management and much more[57]. A built-in layer for mapping tuples to keys is also included[42]. FDB provides API bindings for several languages, including Python, Go, and C, allowing developers to implement layers in their language of choice. The FDB documentation also provides API references[57], extensive guidance for data modeling[42], as well as tutorials for implementing some common data models through layers[58, 59].

Through stacking layers and defining methods for reads, writes, queries, and so on, just about any structure can be created. Keys in the key-value store can be used as IDs for sets of data, as a structure for objects and attributes, and anything in between; everything is up to the developer[42]. An example for how keys can be used to represent JSON objects can be found in section 3.6.3.1. Layers should be implemented as stateless services[7], and can be deployed both as a distributed service, or as local instances on the same host as each application instance[60]. A distributed layer service will make for easier load balancing, while the coupled approach provides much stronger consistency guarantees, since all requests are guaranteed to be handled by the same instance of the layer[60].

Namespaces in FoundationDB are essentially prefixes to a key, but the API handles *subspaces* and *directories* as objects, with methods for lookup, ranges and management, that can be passed around and modified[59]. Subspaces (and nested subspaces) are recommended to define different categories of data, while directories manage related subspaces. This allows layers to reason about the structure of data, without managing state[57, 59].

Sections 3.6.3.1 and 3.6.3.2 provide two simple examples of how layers can be used to implement other data models or structures, without going into specific details on the implementation itself. Both are found in the FoundationDB documentation.

Considerations

Some deliberations have to be made when designing layers[59]. Transactions provide concurrency control, and all database operations in a layer must be encapsulated in a transaction. The API will automatically retry failed transactions, and asynchronous operations will return *futures*, representing a value that will be returned by the key-value store at a later time. This allows operations to execute in parallel, instead of blocking the thread. A transaction may experience some latencies, specifically during set-up and as the transaction commits, but also in cases where reads are done sequentially. Ideally, reads should be done in parallel.

While FoundationDB does provide fully ACID transactions, it is useful to minimize conflicts between transactions, thus reducing the amount of transactions that must be retried[59]. Conflicts reduce the performance of the system, so developers should spread frequently updated data across large amounts of keys. This could be done by splitting values over several keys, based on the structure of the data stored in the value, or to implement atomic operations wherever possible (atomic operations cause no conflicts). Generally, it is also an option to keep a list of changes to the value, instead of modifying the value directly. Any transaction can then insert new elements in the list.

Additionally, maintaining optimal sizes for both keys and values is quite desirable[59]. Performance tests have found that keys should be kept below 32 bytes, and that values larger than 1 kB benefit from being split over multiple keys.

3.6.3.1 Example of data model mapping

This example, found in the documentation for data modeling in FoundationDB[42], explains how layers can be used to move from hierarchical JSON objects to a key-value model, without losing the structure of the data. This is done via the built-in tuple layer. Consider the JSON object seen in figure 3.2.

```
{
  "people": {
    "alice" : {
      "eye_color": "blue",
      "age": 30,
      "friends": ["jim"]
    },
    "jim" : {
      "eye_color": "green",
      "age": 25,
      "friends": ["alice, john"]
    }
  }
}
```

Figure 3.2: A simple JSON object, containing to persons and their attributes.

Instead of storing the objects directly as an entire document, under a single key, we wish to store each object and their attributes separately. By deserializing the JSON object, and implementing a method for mapping the dictionary to tuples, we are left with the structure seen in fig 3.3. Notice how the list of friends keeps its ordering by representing the index as an element in the tuple.

```
("people", "alice", "eye_color", "blue"),
("people", "alice", "age", 30),
("people", "alice", "friends", 0, "jim"),
("people", "jim", "eye_color", "green"),
("people", "jim", "age", 25),
("people", "jim", "friends", 0, "alice"),
("people", "jim", "friends", 1, "john")
```

Figure 3.3: Tuples representing the deserialized JSON object.

Using the built-in tuple layer, these tuples can be mapped to keys and values, which can be stored in the database. This is shown in figure 3.4. Notice how they can be stored in two different ways; one approach uses the final element of the tuple as the value, while the other simply stores an empty string as the value, using the entire tuple as the key. The latter approach means that the existence of a certain key, such as `people/jim/eye_color/green`, implies that the attribute represented by the key is true.

```
"people/alice/eye_color" = "blue",
"people/alice/age" = "30",
"people/alice/friends/0" = "jim",
"people/jim/eye_color/green" = "",
"people/jim/age/25" = "",
"people/jim/friends/0/alice" = "",
"people/jim/friends/1/john" = ""
```

Figure 3.4: Keys and values, representing the tuples seen in figure 3.3

As a result, reading any key starting with `people/alice/` will return the attributes of Alice. This can be done efficiently, since the key-value store is kept in a sorted order. Additionally, one could create methods for altering and accessing specific attributes for a given person.

3.6.3.2 Example of a simple index implementation

This short example shows how a simple index can be implemented in a layer, allowing for multiple ways to retrieve the same data. The FoundationDB documentation provides several guides for creating both simple[61] and more complex indexes[58]; this example can be found in [24].

Suppose we wish to store Alice's eye color in our key-value store. Using the structure seen in the previous example, this would be set as `people/alice/eye_color = blue`. We'd like to have the opportunity to fetch a list of people by eye color as well. To achieve this, we implement a method that retrieves the eye color of a person from the deserialized JSON object, so that we can store the color under an additional key, `eye_color/blue/alice = true`.

By calling the method in the same transaction as the one used to store a person in the previous example, the atomicity of the transaction will ensure that both the index and the person are stored to the database, as long as the transaction succeeds. A range read of keys starting with `eye_color/blue` would quickly show that Alice has blue eyes. Alternatively, it could be possible to create a separate method for setting the eye color of a person, which would store the value to both keys simultaneously.

3.7 FoundationDB Document Layer

On the 30th of November 2018 FDB released the open-source Document Layer, which exposes a document-oriented database API[12, 11]. This section outlines some key features from this layer.

3.7.1 Overview

The FoundationDB Document Layer is a stateless microservice, capable of receiving requests using the MongoDB API, via any MDB driver[12, 11]. This means that users don't have to rewrite their existing applications, except

for some minor differences. FDB claims that applications currently using MDB should be able to ”have a lift-and-shift migration” to this new layer[11]. It uses the FDB key-value store to durably persist its data. FDB-DL can be seen as a replacement for the MDB query language, with a focus on CRUD operations, transactions and indexes, whereas FDB replaces the storage engine[60]. Naturally, FDB-DL inherits the strong guarantees of FDB, and alleviates the need for replica sets and sharded operation.

While the key features of the Document Layer are well defined, and the project is completely open to the public, it is important to keep in mind that it is still very much a work in progress.

3.7.2 Key differences to MongoDB

As mentioned, the Document Layer is compatible with the existing MongoDB API. However, there are certain notable changes, both positive and negative, users should be aware of[62]. On the positive side, FDB-DL avoids the use of locks. It is also able to provide index builds, backgrounded or not, without locking the database. Several commands related to sharding and replication are unimplemented, as they are irrelevant to FDB-DL. Similarily, sessions are unsupported by FDB-DL; the core of FDB already provides better guarantees by default.

The implementation of the Document Layer has some drawbacks as well. It does not implement any aggregation framework at all, and it does not support certain complex queries. Geospatial and text queries, as well as certain kinds of indexes, are unsupported in FDB-DL. There is no support for the MongoDB authentication system or access control either.

The Document Layer has built-in transactions, thanks to the FoundationDB backend [63]. These acted as an addition to the MongoDB API from version 3.0, and worked quite differently from the transactions added in version 4.0 of MDB. The current transactions in MDB are tracked through sessions, and only one transaction at a time can be open per session. The FDB-DL implementation allowed transactions to run in parallel across the same connection. Transactions were available across the entire cluster; they were not limited by the sharded configurations of MDB. However, transactions in FDB-DL faced the same limitations as transactions in FDB.

Additionally, FDB-DL used to have a feature known as ”explicit transactions”, which was removed in version 1.7.0. This allowed users to explicitly control how operations were batched into transactions, trough simple modifications to their application code, reducing overhead and increasing performance greatly. While these are unavailable in the latest release of FDB-DL, due to some connection-related bugs, the developers have stated that they intend to reimplement them by exploiting the transactions found in MongoDB 4.0[64].

3.7.3 Deployment and usage

The Document Layer should be deployed with an instance of the layer on the same host as each application instance, for a tight coupling[60]. Any commands from the application’s MongoDB driver is passed to the layer service, which communicates with the key-value store. It is also possible to deploy the layer as a distributed service, which allows for load balancing between instances of the layer. This approach has a major drawback, however; application instances only see the load balancer, and reconnections to the same IP could be directed to a different instance of the layer. FDB-DL does not provide correct consistency guarantees in this setup.

Chapter 4

MongoDB

This chapter contains a short presentation of MongoDB, and a short discussion of the key features of MDB. The final section of the chapter provides a rough comparison between the features of FoundationDB and MDB. The intention is to provide some insight into the workings of MDB, so as to better understand how the differences may affect subsequent benchmarks. These sections are based on the author's own findings, from a project report written in the fall of 2018.

4.1 What is MongoDB?

MongoDB is a popular NOSQL database, initially released in 2009, developed by MongoDB Inc. (previously known as 10gen)[65, 66]. MDB provides a scalable and flexible document store, that is free and open source[67]. The schema-less structure allows for adaptable and simple storage, and the extensive documentation and guidance provided makes it easy for new users to get started[68].

MongoDB also provides more than 10 different drivers[69], supporting different languages and procedures, as well as several tools for management, cloud and operational services, and analytical tools and connectors[68]. It has built-in tools for queries, indexing, and aggregation, and has a focus on providing a highly scalable and available distributed system[67, 70].

4.2 Key features

This section outlines the most important features of MongoDB. Some discussion around the features and their purpose is included, more in-depth explanations can be found in the MongoDB documentation[70].

4.2.1 Structures and query operations

MongoDB uses JSON-like documents to store data, allowing for variations in structure between documents, and for data structure to change over time[70]. Related documents are grouped within *collections*. Such objects often correspond to the data structures used in many applications and even native data types in several programming languages. It is possible to embed documents within each other, reducing the need for joins and cross-document operations. The MDB API exposes a rich query language, allowing for single and multi-document CRUD operations[71], and much more.

The query language also enables other complex interactions with the database. It can be used to search for text within the fields of a document, or even for geospatial queries[70]. It also provides an aggregation framework for documents, which provides a simpler alternative to map-reduce[72]. This framework exposes several operators and stages, which are useful for quick and powerful data analysis.

Another important feature of MongoDB is the use of indexes[73]. These can improve the efficiency of queries, and may even include multiple keys or keys from embedded structures. MDB indexes can also be enhanced with certain properties. They can be used to guarantee uniqueness for the indexed field, and *partial indexes* will only index documents matching a specified filter. Sparse indexes will only index documents containing the specified field. TTL indexes can be used to automatically purge documents when a given time has passed, based on the value of the indexed field. Indexes can be defined as the collection is created, or built at a later point in time.

4.2.2 Scalability and availability

A core feature of MongoDB is to provide high availability[67, 70]. Replication ensures both data redundancy and availability; multiple copies can provide fault tolerance, and the opportunity to load balance reads[74]. MDB uses *replica sets* to achieve redundancy; these are groups of database instances that store the same datasets. One instance acts as a primary node, receiving all write operations, while other instances (secondary nodes) will replicate data asynchronously. Reads are by default handled by the primary as well, but clients can specify if they wish to allow reads from secondary nodes.

MongoDB aims to provide high scalability as well[67, 70]. This is done through *sharding*[75]. This technique distributes the documents within a collection over several nodes, allowing operations to be distributed across shards for higher throughput. This also increases the storage capacity of the cluster, and makes it possible for collections to grow larger than the storage capacity of a single node. Additionally, a sharded cluster increases the availability of the cluster. Even if shards containing parts of a collection are unavailable, the rest of the collection will still be available on the functioning shards.

The distribution of documents is done by setting a *shard key*, which are made from one or more immutable fields that exist across all documents in the collection[75]. These keys are mapped to chunks, which are spread across shards in the cluster. This mapping can be done by hashing the key, producing evenly distributed documents, or by creating key ranges, which lead to better performance for scanning reads.

4.2.3 Storage engines

MongoDB can be used with three different storage engines, specialized for different use cases[76]. One of these, MMAPv1, is deprecated as of version 4.0. The In-Memory Storage Engine can allow for more predictable latencies, as it avoids disk I/O, but it does not persist any data to disk. It keeps all data in memory, and relies on replicas for fault tolerance. The third, and default, storage engine is the WiredTiger engine.

WiredTiger allows for multiple clients to modify different documents at the same time, using document-level concurrency control[77]. Most reads and write operations are handled using multiversion concurrency control, but operations at the global, database, and collection levels use intent locks. Some important operations require instance-wide locks, or even exclusive database locks. Data is made persistent and durable through the use of write-ahead transaction logs and checkpoints, which are snapshots of in-memory data written to disk. Finally, WiredTiger supports the use of LSM-trees, which allows for faster writes and less overhead, while approaching the read performance of b-trees[78].

4.2.4 Transactions

Multi-document transactions, released during the summer of 2018 (version 4.0), are a new feature in MongoDB[79]. The implementation guarantees ACID-compliant transactions, across multiple operations, collections, databases, and documents, within the same replica set. Multi-document transactions across a sharded cluster are not implemented yet, but is scheduled for version 4.2. This feature is only available for the WiredTiger storage engine.

While all CRUD operations are available for use within a transaction, some operations are not allowed. A full list can be found in [79]. A multi-document transaction may also suffer from greater performance costs, and all transactions are tightly coupled to a session. Finally, a client must have sufficient privileges for all operations in a transaction.

4.3 Comparing MongoDB and FoundationDB

As previously discussed, MongoDB provides a complete service, including everything from analytical frameworks and aggregation pipelines to several storage engines. FoundationDB on the other hand, provides a minimal core system, stripped of additional features. As mentioned in previous chapters, several key features of MDB are deliberately not included in FDB. Query languages, indexes and aggregation tools are instead intended to be implemented by the users themselves, through layers. This makes a comparison between the two similar to comparing apples and oranges, but there are still several overlapping features worth discussing.

While FoundationDB sports a key-value store capable of being adapted to any data model, MongoDB provides a specialized document store. Each approach has different strengths and weaknesses. The configurable solution of FDB opens many opportunities, but if it is to handle other data structures, all data related operations are forced to go through a mapping to fit the key-value model[58]. All such transformations must be handled in layers, which may add extra cost to any operation. MDB can store data, perform queries, use indexes, and even access individual fields within the document, without performing such mappings, although it only supports the document model.

As mentioned in the previous section, MongoDB employs replica sets and sharding to achieve data redundancy, fault tolerance, high availability and scalability[70]. FoundationDB only has the concept of storage nodes, and uses the master node to automatically distribute data across nodes in an efficient manner[41]. This ensures that data is redundantly stored, and highly available, without having to worry about managing replica sets or choosing shard keys. The ordered structure of the key-value store ensures that range reads are efficient, avoiding the choice between hashed or ranged sharding.

The WiredTiger engine is responsible for managing and storing data in MongoDB[76]. This approach is entirely different to FoundationDB, where one could argue that all it aims to deliver is a storage engine[24]. The ACID properties of FDB provides stronger guarantees than WiredTiger, and FDB avoids the use of locks completely. However, FDB has some limits, as mentioned in previous chapters. Transactions are limited both in runtime and data size, and keys and values have size limits as well[43, 53]. Additionally, WiredTiger leverages LSM-trees to achieve better write performance, whereas FDB uses b-trees[78, 44].

Transactions are also handled quite differently in the two systems. The concept is a new addition to MongoDB, albeit it does not cover all operations, and cannot be used acrossed sharded clusters[79]. In comparison, FoundationDB has transactions as a core concept; any database operation, across any amount of nodes, is done through ACID-compliant transactions. Additionally, transactions in MDB are tightly coupled to a session, allowing for only one transaction per session at a time. FDB supports the parallelization of transactions, allowing for higher throughput. It is however important to remember the drawbacks and limits of transactions in FDB, as mentioned in the previous paragraph. They cannot exceed a runtime of five seconds, and are limited to writing only 10 MB of data per transactions[54]. In fact, it is encouraged to keep transaction sizes below 1 MB for performance reasons.

In summary, MongoDB and FoundationDB are quite different databases. Where one provides great tools for quickly accessing, aggregating and analyzing data, the other provides a robust core, with strong guarantees for data storage and management. However, the customizable layers of FDB enable it to ”learn” key features from MongoDB, without sacrificing its strong guarantees, and hopefully without sacrificing its performance.

Chapter 5

Preliminary benchmarks

This chapter presents two methods used to acquire a baseline benchmark for the performance of FoundationDB, in comparison to MongoDB. The methods were chosen for preliminary benchmarks, with the intent of becoming more familiar with the intricacies of FoundationDB and the concept of performance tests for databases. These methods helped shape an image of FDB's capabilites under ideal conditions, and how it compared to MongoDB under standardized conditions.

The first section details the built-in test suite of FoundationDB[51, 10], while the second section describes *YCSB*: a standardized benchmarking tool[80, 81]. Both sections also briefly discuss why these methods were insufficient in terms of evaluating the viability of FDB and the Document Layer.

5.1 FoundationDB's built-in test framework

FoundationDB provides an integrated testing suite, complete with predefined workloads and templates for creating custom test runs[82]. This can be used by contributors as a way to ensure that changes or additions don't negatively affect system performance. It is also used extensively by Apple developers: an automated suite of performance tests is used for systematic testing every single night[51]. The system also provides tools for testing system correctness, and allows for *deterministic* end-to-end simulations of the system while generating random failures.

Each workload has a set of parameters, outlining the test to be performed[83]. These include a title and name for the test, and a duration. The amount of transactions to be generated per second can be defined, as well as the contents of a transaction (e.g. 90% reads and 10% writes). It is possible to define two kinds of transactions per workload: transaction *A* and *B*. Each may have different contents. There is also a parameter used to set their relative rarity, i.e. the percentage chance that the next generated transaction is of type A or type B. The size of keys and values, and the amount of records can also be defined.

5.1.1 Drawbacks

While the built-in testing suite has a wide array of tools and use cases, it is not particularly suited for this research. As the integrated test framework is native to FoundationDB, and designed with the intricacies of FDB in mind, the results of the test suite may have some inherent statistical bias. Even if MongoDB had its own test suite, which it currently does not, it would irrelevant to compare the results. It would be quite difficult to ensure that two such frameworks were operating on equal premises.

The FoundationDB Document Layer does not have a built-in testing framework. The test suite only works for the standard FDB key-value store, making the results irrelevant for the main problem: comparing the Document Layer with an established competitor, namely MongoDB.

Finally, the test suite is not well-supported on macOS. Running the tests requires building the `fdbserver`-binaries manually, which is known to cause issues on macOS [84]. The accepted workaround is to build the

binaries inside a Docker container[85] running Ubuntu, and to use that container for testing. As one might guess, this does hamper performance slightly.

5.1.2 Benefits

In spite of the drawbacks discussed above, using the integrated test framework does provide some value. The results describe a kind of baseline benchmark for FoundationDB, and they give an idea of what FDB is capable of under ideal conditions on a single machine. Ideally, benchmarks of the FoundationDB Document Layer should be able to achieve results close to the baseline result of the native testing suite.

5.1.3 Method

The tests were performed on a mid-2017 MacBook Pro running macOS 10.14.4, with a 3.1 GHz dual-core Intel Core i5 Kaby Lake (7267U) CPU, 8 GB of RAM, and 256 GB of SSD storage. As previously discussed, the tests were performed within a Docker container[85]. The Docker service was configured to limit containers to two CPU cores, 2 GB of memory, and 1 GB of swap.

The container was created and the binaries were compiled using the guidelines for building FoundationDB in Docker. These are found in the FDB repository README[8]. Additionally, the FDB client and server were installed according to the guidelines found in the FoundationDB documentation[7], as that generates some handy default settings. Then, the FDB service was stopped using `$ service foundationdb stop`. Two instances of the `fdbserver`-process were used to set up the test system, using the following commands for initialization, respectively:

```
$ ./bin/fdbserver -p auto:4500 -d /var/lib/foundationdb/data/4500 -L
/var/log/foundationdb -l public

$ ./bin/fdbserver -c test -p auto:4501 -d /var/lib/foundationdb/data/4501
-L /var/log/foundationdb -l public
```

The first command starts a basic server, the starting point for the cluster. The second command starts a server with `class=test`, which connects to the cluster and is used as the test manager. To actually perform tests, a third server was initiated, using the following command:

```
$ ./bin/fdbserver -c test -p auto:4502 -d /var/lib/foundationdb/data/4502
-L /var/log/foundationdb -l public -r multitest -f tests/file --num_testers 1
> tests/runs/file-date-time.txt
```

The command defines the server's role as a multitest runner, and points to the file defining the test to be run. In this research, `tests/ReadRandom.txt` was used. It is a simple test where 50 000 nodes are read continuously for 10 seconds, with 10 reads in each transaction. It aims to generate 2500 transactions per second. Finally, the output was written to a `.txt`-file; *file*, *date* and *title* should be replaced with the relevant data, so as to easily manage test results. The `runs`-directory must be created manually before storing test results there. All three commands are called from within the root directory of the FoundationDB repository, and from within the Docker container.

5.2 YCSB

In 2010, Yahoo! released a standardized benchmarking framework, the Yahoo! Cloud Serving Benchmark (YCSB)[80, 81, 86], aiming to provide a testing tool for common workloads across different NOSQL systems. Initially, API bindings were only provided for five database systems, but the open source project has since grown to support over 30 different NOSQL databases.

The tool has three parts: workload definitions and templates, the YCSB client, and API bindings for the supported systems. The workload definitions describe a set of operations for the client to perform. It is possible to configure the number of records and operations, the proportion of basic operations (reads, inserts, updates and scans), to define the size of keys and values, the distribution of requests across the dataset, and much more. The

client generates actual workloads based on these definitions, mapping proportions and properties to randomly generated datasets and database operations. The API bindings implement the four basic operations defined in the workloads, as well as a method for deleting records. A binding connects the client to the database being tested.

5.2.1 Standard workloads

YCSB provides seven predefined workloads, useful for quickly comparing database systems at a glance. Each workload has different characteristics, testing different aspects of the database, and how it operates during different situations[87]. Before a workload can be performed, the database must be loaded with the related dataset. This is done by running a corresponding loading procedure for the workload, effectively doubling the number of predefined benchmarks.

Workload A is an update heavy workload, with a 50/50 mix of reads and writes.

Workload B focuses on read performance with minor updates, with a 95% read ratio.

Workload C is a read only workload.

Workload D has a 95% read ratio as well. It inserts some new records, and the latest records are preferred when reading, making recent records more popular.

Workload E focuses on short range reads, with a 95/5 ratio of scans and inserts.

Workload F will read, modify, and write about 50% of records, while simply reading the remaining 50%.

TimeSeries Workload A generates a large amount of records containing numeric values and timestamps. The operation count is much greater for this kind of workload: it has almost 3 million operations, while the other predefined workloads typically have 1000 operations. 90% of these operations are inserts, while the remaining 10% are reads.

5.2.2 Configuration, installation and application versions

All YCSB benchmarks were performed on a a mid-2017 MacBook Pro running macOS 10.14.4, with a 3.1 GHz dual-core Intel Core i5 Kaby Lake (7267U) CPU, 8 GB of RAM, and 256 GB of SSD storage.

Software

- Version 0.15.0 of YCSB was used for all benchmarks, and it was installed by following the guidelines in the README in the YCSB repository[81].
- Version 5.2.0 of FoundationDB was used for all benchmarks. It was installed by following the guidelines found in the FDB documentation[7]. Both client and server was installed.
- Version 3.6.2 of MongoDB was used for all benchmarks[68]. It was installed using Homebrew for macOS.
- Version 1.6.3 of the FoundationDB Document Layer was used for all benchmarks. It was installed by following the guidelines found in the FDB-DL documentation[12]. It was configured to run on port 27017 when active, the default port for MongoDB servers.

Each of the following benchmarks consists of the same ”test suite”, where all 7 default workloads are performed in order, with the corresponding loading workload preceding each respective workload. Additionally, all benchmarks were completed with the `batchsize`-parameter set to 100. This parameter ensures that all operations are performed in batches of 100, instead of a single operation at a time.

5.2.3 Benchmarking FoundationDB

FoundationDB was benchmarked using the API binding included in YCSB. The benchmark was performed on a default FDB cluster, i.e. a cluster comprised of a single, local server-process. The benchmark provides a baseline result for the default FoundationDB configuration, under standardized conditions. It provides a more prudent and cautious benchmark of FDB's capabilities, compared to the native testing suite.

However, just as the native testing suite, benchmarking the default FoundationDB configuration is not strictly relevant to the key problem of this thesis.

Method

The following commands, called from within the root folder of YCSB, were used to perform the benchmark:

```
$ ./bin/ycsb load foundationdb -s -P workloads/workload -p
foundationdb.batchsize=100 > runs/fdb/date/load-workload-time.csv
```

```
$ ./bin/ycsb run foundationdb -s -P workloads/workload -p
foundationdb.batchsize=100 > runs/fdb/date/run-workload-time.csv
```

The first command generates and loads the corresponding dataset for the workload, while the second generates and performs the workload. Both commands store their results in specific output files. $workload$ should be replaced with the chosen workload, while $date$ and $time$ should be replaced for easy output file management. A short script was written to run these commands for all seven workloads, and to create the directory structure for the output files.

5.2.4 Benchmarking MongoDB

This benchmark was used to generate a baseline result for the default MongoDB configuration.

The baseline result from this standardized benchmark is useful for comparisons with the results from benchmarks of the FoundationDB Document Layer. However, the results of this benchmark are not directly comparable to those of the benchmark of the default key-value store of FoundationDB, as mentioned previously.

Method

First, the MongoDB daemon must be started. This is done by calling `$ mongod`.

The following commands, called from within the root folder of YCSB, were used to perform the benchmark of MongoDB:

```
$ ./bin/ycsb load mongodb -s -P workloads/workload -p mongodb.upsert=true -p
mongodb.batchsize=100 > runs/mdb/date/load-workload-time.csv
```

```
$ ./bin/ycsb run mongodb -s -P workloads/workload -p mongodb.upsert=true -p
mongodb.batchsize=100 > runs/mdb/date/run-workload-time.csv
```

The commands are essentially equal to those described in section 5.2.3. All references to FoundationDB are replaced with MongoDB, and an additional parameter, `upsert`, was added. This ensures that MongoDB is allowed to update existing records when attempting to insert a record that is already found in the database. This allows for several workloads to be performed in succession, without deleting the MongoDB collection between each workload. A short automation script was written here as well.

5.2.5 Benchmarking MongoDB with stricter write concern

A second benchmark was performed for MongoDB, this time with a stricter write concern. The default write concern of MongoDB only guarantees in-memory consistency, while FoundationDB and the Document Layer has much stricter consistency by default[88]. This benchmark was performed to generate a result that could be used for a more "fair" comparison, under more similar premises.

Method

As described in section 5.2.4, the MongoDB daemon must be started first.

The following commands where used to perform the benchmarks, in the same way as previous benchmarks.

```
$ ./bin/ycsb load mongodb -s -P workloads/workload -p mongodb.upsert=true -p
mongodb.batchsize=100 -p
mongodb.url='mongodb://localhost:27017/ycsb?w=w&journal=j' >
runs/mdb/date/load-workload-time.csv

$ ./bin/ycsb run mongodb -s -P workloads/workload -p mongodb.upsert=true -p
mongodb.batchsize=100 -p
mongodb.url='mongodb://localhost:27017/ycsb?w=w&journal=j' >
runs/mdb/date/run-workload-time.csv
```

The only difference from the previous commands is the addition of a new parameter: the connection string. It uses the default address and port, and the default collection name (*YCSB*). Additionally, it has two query parameters used to set the write concern: w and j. They should be replaced with `majority` and `true`, respectively, to enforce a write concern that is more similar ot that of FoundationDB[89].

5.2.6 Benchmarking FoundationDB Document Layer via MongoDB

The final benchmark used YCSB to test the performance of the FoundationDB Document Layer. It used the YCSB API bindings for MongoDB, and the Document Layer replaced the MongoDB daemon when it came to actually processing the operations. It acted as a connection between the YCSB API and the standard FoundationDB key-value store.

By default, the Document Layer has a strict write concern, and it will wrap all operations in transactions, just like the regular FoundationDB[62, 63].

Method

Instead of starting the MongoDB daemon, this benchmark requires starting the Document Layer service. This is done by calling the following command on macOS:

```
$ launchctl load
/Library/LaunchDaemons/com.foundationdb.fdbdocmonitor.plist
```

See the Document Layer documentation for instructions on how to start the service when using other operating systems[12]. The commands used to perform the actual benchmark were the same as the commands used in section 5.2.4. This is because the Document Layer simply replaces the underlying storage engine, as explained previously.

Chapter 6

Results of preliminary tests

In this chapter, results from both preliminary testing methods are presented. The first section details the results from the built-in testing framework found in FoundationDB, while the second section presents results from the standardized YCSB benchmarks.

6.1 Integrated test suite

Using the native test framework of FoundationDB, and the `RandomRead`-test, yielded quite predictable results. As seen in table A.1, found in appendix A, the benchmark was able to achieve approx. 2140 transactions per second, and 21 398 operations per second. Table 6.1 shows some key results from the benchmarks.

Metric	Output value
Transactions/sec	2139.80
Operations/sec	21398.0
Retries	0.0
Read rows	213980.0
Write rows	0.0
Mean Latency (ms)	230.903154
Median Latency (ms, averaged)	248.797894
Mean Row Read Latency (ms)	11.064244
Median Row Read Latency (ms, averaged)	10.884285
Mean Total Read Latency (ms)	11.172971
Median Total Read Latency (ms, averaged)	11.0633
Max Total Latency (ms, averaged)	43.063879
Mean GRV Latency (ms)	219.657581
Median GRV Latency (ms, averaged)	237.621784
Read rows/sec	21398.0
Bytes read/sec	684736.0

Table 6.1: Key results from the native test framework of FoundationDB

6.2 YCSB benchmarks

This section briefly presents some key results from the YCSB benchmarks of all four database configurations, with a focus on throughput (operations per second), and average latencies for operations (milliseconds). Due to similarities between the workloads, only four of the seven workloads are presented. Similarly, only one of seven loading procedures is presented. See appendix B for complete outputs of each workload and loading procedure, for all four configurations.

6.2.1 Workload A

Workload A is a 50/50 mix of reads and updates. As seen in 6.1, FoundationDB performed surprisingly well when inserting this data. Figures 6.2 and 6.3, showing the throughput and latencies when runnning the workload, display somewhat more predictable results.

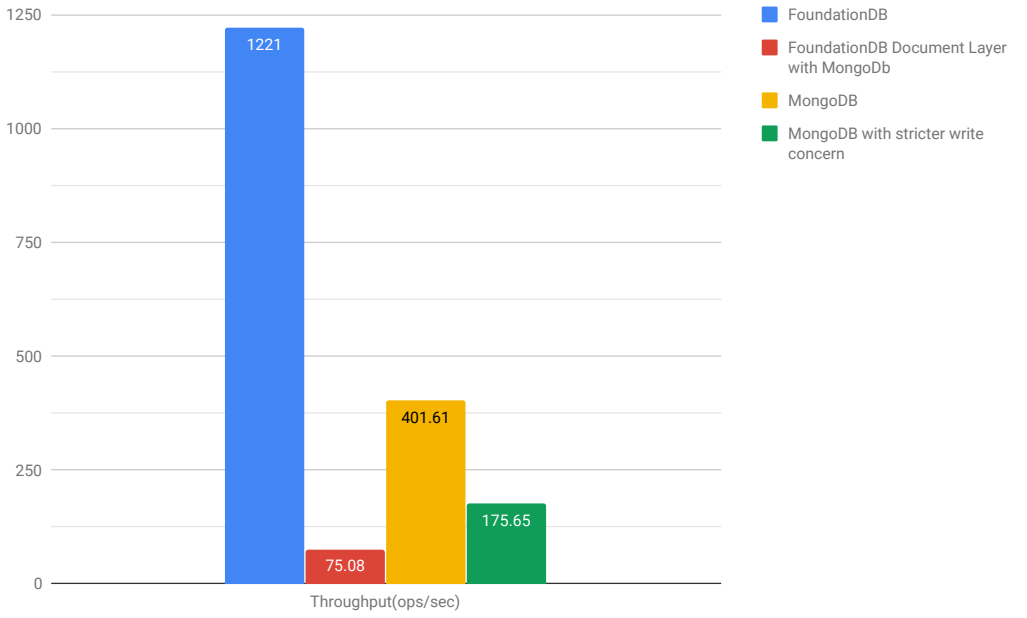

Figure 6.1: Throughput when loading workload A

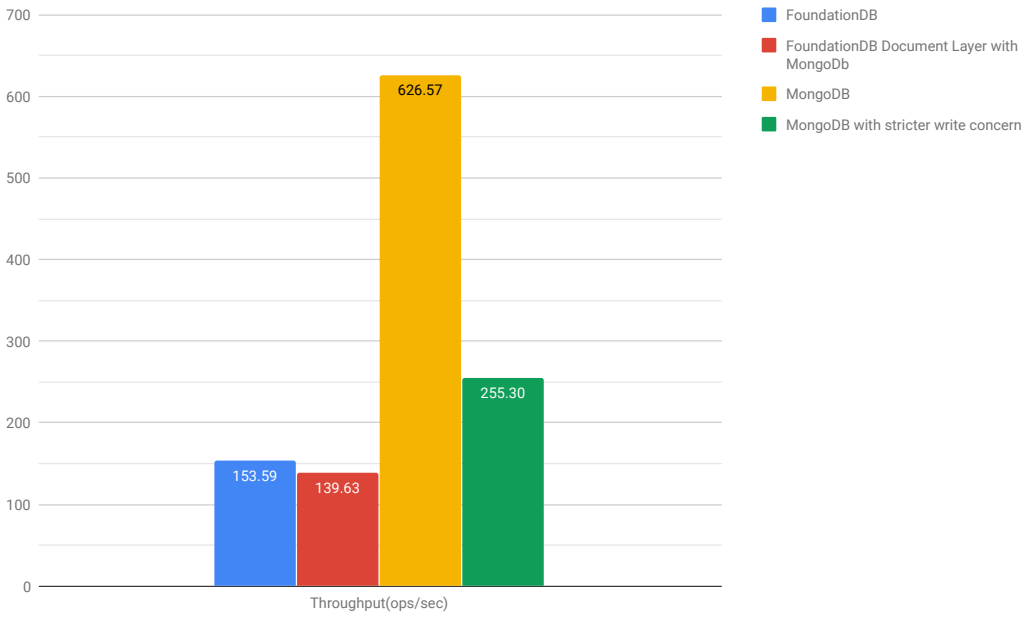

Figure 6.2: Throughput when running workload A

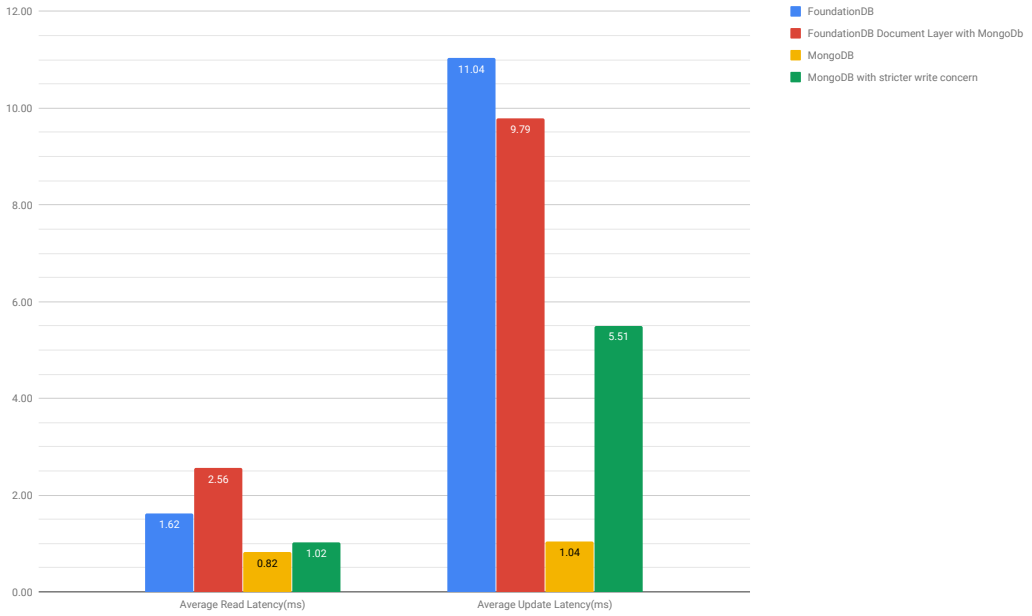

Figure 6.3: Average latencies when running workload A

6.2.2 Workload B

The second workload focuses on read performance, with 95% reads and 5% updates. The performance of FDB was much better when running this workload, both for the default configuration and the Document Layer. The stricter configuration of MDB also performed better during workload B. This is shown in figure 6.4.

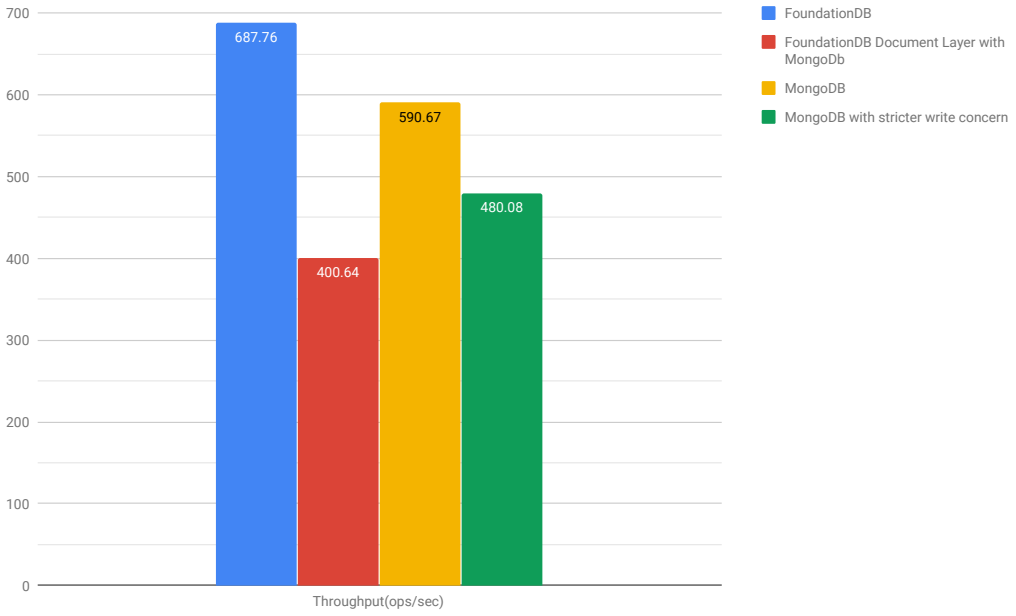

Figure 6.4: Throughput when running workload B

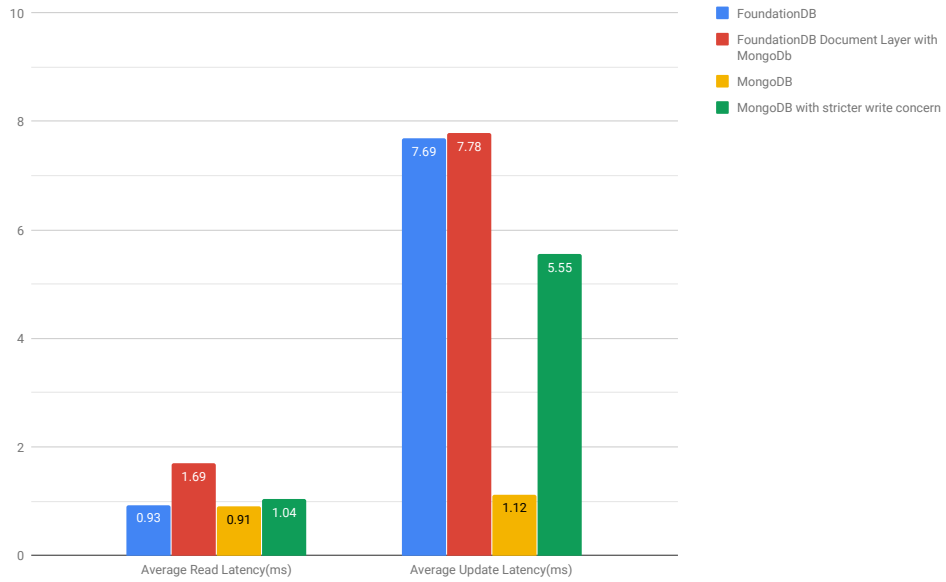

Figure 6.5: Average latencies when running workload B

Figure 6.5, displaying average latencies, is quite similar to figure 6.3. However, where the latencies of the two different MDB configurations are practically equal to those found in workload A, the two FDB and FDB-DL configurations show a reduction in latencies of approximately 30% compared to workload A. This matches the improved throughput.

6.2.3 Workload E

Workload E employs short range reads, with a 95/5 ratio between scans and inserts. This seemed to severely affect the performance of the Document Layer. It also reduced the performance of FDB and regular MDB, while the stricter MDB configuration was improved moderately.

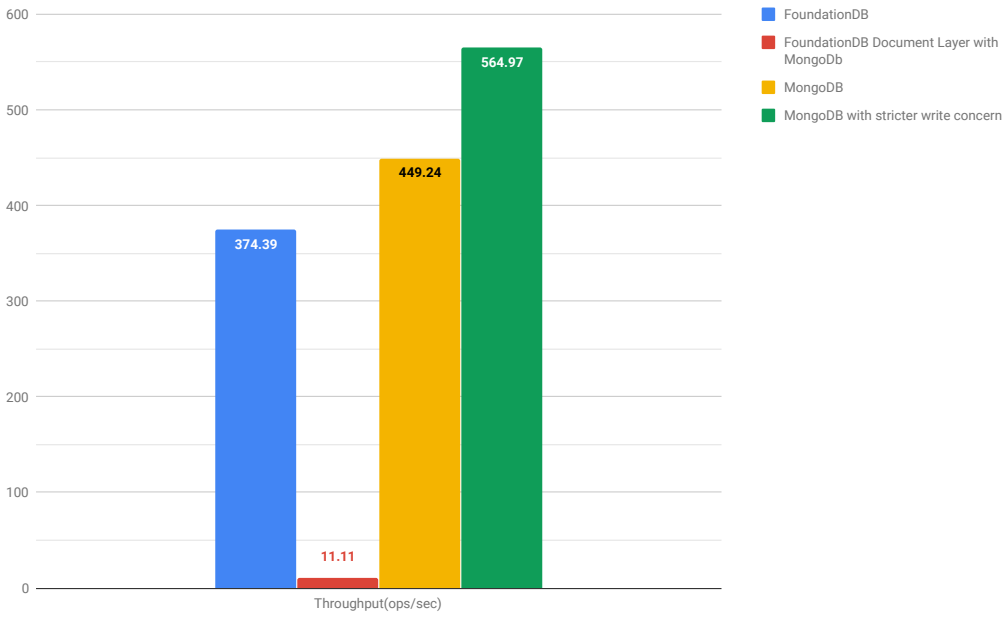

Figure 6.6: Throughput when running workload E

6.2.4 TimeSeries Workload A

The TimeSeries workload generates a large amount of records, and the operation count is much increased compared to the regular workloads. It has almost 3 million operations, while the other predefined workloads typically have 1000 operations. These operations have a insert/read ratio of 90/10. This allowed the default configurations of FDB and MDB to achieve much higher throughput: it was improved by a factor of 10 compared to previous workloads. FDB-DL and the stricter MDB configurations did not reap the same results, however, as seen in figure 6.7.

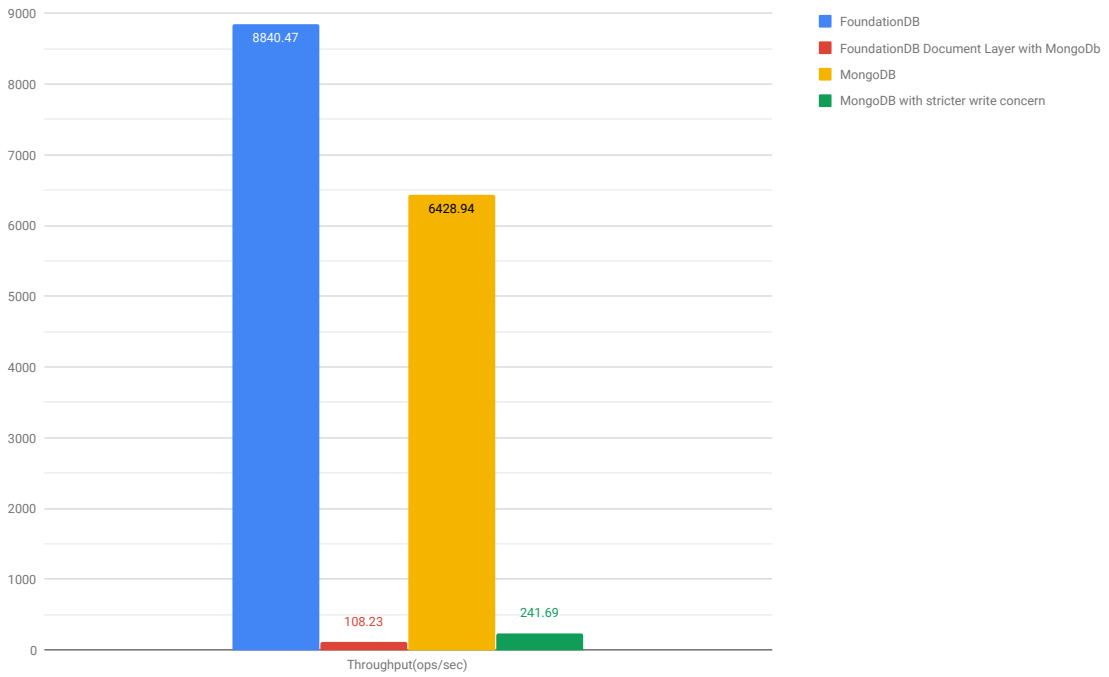

Figure 6.7: Throughput when running TimeSeries workload A

Chapter 7

Discussion of preliminary results

The following sections briefly analyze the results presented in chapter 6. The first two sections discuss the results from the integrated test suite and the YCSB benchmarks, while the last two review the limitations of the preliminary test methods, and how the information learned may be used when creating a custom testing method.

7.1 Integrated test suite

The native test framework of FoundationDB delivered formidable results for a single core benchmark, seen in table 6.1, which are relatively consistent with the developers' own results[90]. This gives a decent picture of what FDB should able to achieve under the ideal conditions for this specific configuration.

However, the test was not able to achieve the targeted transactions per second (tps); the test defines its goal as 2500 tps, while the benchmark only achieved 2139.8 tps. As mentioned by developers on the FDB forums[83], a FDB system under saturation pushes most latency into getting read versions (GRV), which is evident in table 6.1. Note how the median GRV latency equals about 95% of the median latency. Thus, this limited system configuration is likely saturated before it is able to achieve the targeted 2500 tps.

While the integrated test suite of FDB showed promising results, they are inherently biased. The `RandomRead`-test plays to FDB's great read performance, and is configured by default to use operations that are well suited to FDB. While representative of a specialized application, it may not be representative of every-day, real-world use.

7.2 YCSB benchmarks

This section briefly discusses the results of each workload, and each configuration. The section is based on the results presented in chapter 6, while complete results may be found in appendix B. This analysis is used to uncover information and details that may prove useful when developing a custom benchmark later on.

7.2.1 Workload A

A large amount of inserts is quite costly for several of these configurations, which should harm their throughput when loading data for this workload. This is due to the conflict checking required when using ACID-compliant transactions. However, as seen in figure 6.1, the default configuration of FoundationDB performed quite well when loading data. This could be explained by the batch size used when packing operations into transactions, reducing the overhead for FDB. Also, inserting data into an empty database would not cause many conflicting transactions.

The poor perfomance of the Document Layer is somewhat troubling, The low throughput seems to be a recurring factor when performing inserts in FDB-DL, and it displays very high average latencies compared to the other configurations upon insertion (see table B.43 for further details). On the other hand, it is similar to the performance of MDB with stricter write concern. This is a good sign, as the two configurations operate quite similarly in terms

of consistency, although the version of MDB used in YCSB does not support transactions.

When running Workload A, as seen in figure 6.2, results matched expectations. Configurations with costly updates were slow, while the direct inserts of standard MongoDB outperformed all others, due to the extra overhead incurred by transactional configurations. The average latencies, seen in figure 6.3, really show how a large amount of updates will slow FDB and FDB-DL to a halt. Once again, this may be due to transactional overhead.

The configuration of MDB with stricter write concerns delivered double the performance of FDB-DL when running workload A, and even outperformed the default FDB configuration. Again, this likely comes down to the transactional overhead. While the stricter write concern of MongoDB should reduce performance, it is less noticeable in a single core environment. The `majority`-setting requires the data to propagate to a majority of data-bearing members before notifying the client that the write is completed[89], which in this case is only one member.

7.2.2 Workload B

The read-heavy operations of workload B lend themselves quite well to both FDB and FDB-DL, as seen in figure 6.4. The performance was vastly improved over workload A, and FDB even outperformed the default MDB configuration. The MDB configuration with stricter write concern also showed greater throughput than in workload A. This is simply explained by the reduced amount of updates, which are quite costly when using transactions or a stricter consistency level.

The latencies of both FDB and FDB-DL configurations were also improved, as seen in figure 6.5. Interestingly, the average latencies of both MDB configurations remained relatively equal to those seen in workload A, despite the throughput being different.

7.2.3 Workload E

Workload E displayed vastly different results than the previous workloads, as seen in figure 6.6. Both FDB and MDB had a slightly lower throughput than previously seen, which is somewhat surprising. FDB claims to provide high performance range reads, but was unable to deliver this in this standardized benchmark. This may be due to the implementation used in the YCSB client binding.

Additionally, the strict configuration of MDB outperformed the default configuration. This is particularly odd, as the stricter configuration should be slower in all regards. This may simply be due to errors in the research method or randomness in the system.

The performance of FDB-DL was quite abysmal in this workload, dipping below 10% of the throughput seen in all other workloads. The configuration seems almost incompatible with YCSB's implementation of scanning reads.

7.2.4 TimeSeries Workload A

TimeSeries workload A, an insert-heavy workload, showed some truly remarkable results. Default FDB and MDB both performed quite well, delivering ten times higher throughput than in previous benchmarks. In fact, FDB was almost able to achieve the same throughput as in the native test framework, seen in table 6.1. It is kind of surprising that FDB performed so well in such a insert-focused benchmark; it seems that the large amount of records and operations is able to truly push the system towards saturation, which is key when achieving high performance in FDB[90].

On the other hand, FDB-DL and the strict MDB configuration showed results similar to those seen in the benchmark of workload A (figures 6.1, 6.2). While not much worse than previous benchmarks, it is quite evidently much slower than FDB and MDB in this workload. This may be due to the large amount of inserts, causing large latencies for each transaction (see table B.56).

7.2.5 Configurations

The YCSB benchmark is not particularly well suited to the intricacies of FoundationDB. The documentation of FDB describes several properties to keep in mind when designing transactions, record-sizes, and more. For instance, FDB shines when transactions are of a certain length, and when they are designed to avoid conflicts. The standardized tests do not take this into account, naturally. Also, FDB achieves its highest throughput when the system is saturated[90], which only the final workload was able to achieve.

The default configuration of FoundationDB is a key-value store, while MongoDB is a document store. Thus, some of these benchmarks may not be directly comparable. Additionally, the results may be slightly biased in favor of MDB when comparing to FDB, as direct insertions should be much faster than the transaction-based operations that are required by FDB. Despite this, FDB actually outperformed MDB in a few workloads. This may be due to the great read performance of FDB, as reads are able to avoid a lot of transactional overhead.

The strict configuration of MDB performed as expected, being slower than default MDB in practically all benchmarks. This is due to the stronger consistency, which costs more. It is surprising that a more costly configuration outperformed default MDB in workload E, but this may come down to error and randomness.

The FoundationDB Document Layer performed quite poorly compared to default MongoDB, which is expected, but it was not able to equal the performance of the stricter MDB either. This may be due to the fact that FDB-DL must wrap operations in transactions, while MDB can quickly push through insertions. It is important to keep in mind that YCSB does not support MongoDB 4.0, which implements transactions[79].

Another cause for the low performance may be that FDB-DL is a work in progress, and there is still much room for improvement when it comes to the performance of certain operations, including how they are translated from MongoDB operations to FoundationDB operations. This is particularly noticeable when testing the insert-performance of FDB-DL, and when using range reads.

7.3 Limitations and caveats

As previously mentioned in section 5.2, the tests were performed on a MacBook Pro, which limits performance slightly. Additionally, the integrated tests of FoundationDB were built and performed within a Docker container. Due to the preliminary nature of these benchmarks, not much time has been spent on configuring the benchmarks and database systems to achieve maximum throughput in each test case. Creating custom workloads in YCSB and the native framework are also possibilities that went unexplored.

Finally, results are limited by the YCSB client bindings, which may not be ideal for each database, and are dependant on the contributions made by developers to the open source project.

7.4 Findings

This section details certain useful findings that may affect the development and implementation of the custom test methods.

Several details and issues uncovered by the preliminary benchmarks proved useful when implementing the custom test method later on. For instance, while the Document Layer can be implemented in a ”plug-and-play”-manner, some thought must be put into the design and use of transactions to achieve great results. Similarly, key and record sizes, namespaces and other performance considerations must be made.

The default configuration of FoundationDB was omitted from further benchmarks. The goal was to compare MongoDB and a competing document store, and the default key-value store of FoundationDB was not particularly relevant in that case. Also, adapting the custom tests to fit both data structures would have made the benchmark quite generic, which was not the intention.

The version of MongoDB supported by YCSB does not support transactions, as this is a relatively new feature in MongoDB. A stricter configuration of MongoDB was benchmarked to emulate the stronger consistency provided by ACID transactions, and the increased cost. Creating a custom benchmark opened the opportunity to use any version of MongoDB. Thus, the strict configuration of MongoDB 3.0 was replaced in the custom benchmark by a transactional configuration of MongoDB 4.0.

In summary, the custom benchmark had to support the Document Layer, and MongoDB with transactions, as well as a default configuration of MongoDB to use as a baseline result. It would also be useful to develop several implementations of each workload, specialized for each configuration, while still managing to keep the operations of the implementations so similar that the benchmarks are comparable, and can be viewed as one. This would enable the benchmarks to make the ideal design choices for each configurations, and to use the different kinds of transactions and operations to their full extent.

While the Document Layer was unable to achieve the same result as MongoDB in these standardized benchmarks, the results show that a properly configured and designed system may allow the FoundationDB Document Layer to reach the same level of throughput.

Chapter 8

Research method

This chapter introduces the custom testing tool, implemented to benchmark and compare MongoDB and the FoundationDB Document Layer. The first section discusses the inner workings and implementation of the tool. The second section presents each workload, and how they were implemented. Lastly, the final section discusses the method used for benchmarks, and limitations and caveats to this method.

8.1 Document store benchmarking tool

To evaluate and benchmark the performance of FoundationDB Document Layer and MongoDB, a custom test tool was written (hereinafter: *the tool*). The source code can be found in appendix C, and in [91]. The tool, which is implemented in Python, intends to provide a simple performance testing suite that is more specialized towards MongoDB and FoundationDB Document Layer than the standardized tests seen in YCSB.

The six workloads used by the tool mimic the workloads of YCSB in terms of operations, i.e. workload A of the custom tool has the same request distribution and operation ratios as workload A in YCSB, and so on. However, each workload in the custom tool has access to several implementations when performed: one for each database configuration. This allows it to be more specialized for each database configuration (known as *runners* in the tool), while still being comparable across configurations. Additionally, the mimicry of YCSB workloads enables easy comparisons with the preliminary results from chapter 6.

While the workloads are similar to those of YCSB, one element is deliberately different. Where YCSB executes all reads at once, followed by all insertions, the custom tool randomizes the order of operations. This is more similar to real-world situations, and makes for more realistic batching of operations. Where YCSB can batch hundreds of inserts in a single operation, the custom tool must create batches of mixed operations on the fly. The specific workloads and their implementations are described in detail in section 8.2.

Upon execution of the tool it is possible to define which runners to test, which workloads to perform, and how many runs to complete of each workload. The tool will perform all chosen workloads using all chosen runners, as many times as defined upon execution. The average throughput per workload per runner is written to separate `.csv`-files, along with some other interesting metrics. The available runners are the FoundationDB Document Layer, the default MongoDB configuration, and MongoDB 4.0 with session-based transactions.

8.1.1 Technical considerations from preliminary findings

As discussed in section 7.4, certain design considerations must be made when designing workloads that are optimized for MDB and FDB-DL. For instance, where YCSB will wrap each operation in a separate transaction, which causes large amounts of overhead, the tool makes a conscious decision about how to batch operations into fewer transactions. The amount of operations in each transaction is based on the considerations to transaction design presented in the documentation of FoundationDB[48, 53, 54], as transaction size and runtime may affect the performance of FDB.

The tool also ensures that the size of inserted documents is kept in mind when working with transactions, while the Document Layer itself handles key sizes and namespace structure. These factors are also relevant when improving the performance of FDB. All these considerations for transaction design are also implemented in the benchmarks of transactional MDB (MongoDB 4.0), and should benefit both runners.

8.1.2 Implementation

All benchmarks begin with the execution of `test.py`, which handles the configuration of each runner, and the execution of workloads. It acquires Mongo Clients, connects to the appropriate database and collection, and acquires an instance of the active workload through helper methods defined in `setup.py`. It executes benchmarks by acquiring a client for the specific runner, fetching the correct database and collection, and then acquiring and executing workloads sequentially. Once a runner has completed all runs for all workloads, the next runner is initiated.

Workloads are implemented as classes, which all inherit the abstract `Workload`-class. This base class defines quite a few attributes, counters and initial values that are used during benchmarking, and defines abstract benchmarking methods for each runner, that each specific workload must implement. The entry point for all workloads is the `benchmark()`-method, which is implemented in the base class. This method acquires the appropriate benchmarking method for the active runner, and performs the benchmark as many times as was defined upon execution of the tool. It is also responsible for loading data into the database before benchmarking, and building an index on the field used for lookups in the workloads (a simple field was defined in all documents, storing a unique, small integer, acting as a makeshift primary key which is easily accessed programatically), and it handles timing and logging of several metrics during benchmarks. Finally, it is also responsible for writing all output to a `.csv`-file after all runs of a workload has been completed for the active runner.

The specific workloads define the amount of records that are to be loaded before their execution, and the number of operations to execute. Also, all workloads implement the required benchmarking methods: `benchmark_mongo3`, `benchmark_mongo4`, and `benchmark_fdbdl`. Each method implements and performs the operations of the relevant workload, while being specialized for their respective runners. Each workload emulates some situation or ruleset, such as 50/50 read/update with a certain distribution of requests, and each method is tasked with implementing the operations according to this ruleset. Further details about the implementation of workloads can be found in section 8.2

Configuration

Clients for standard MongoDB were acquired with the default connection string, while the following strings were used for the Document Layer and transactional MongoDB (MongoDB 4.0 with session-based transactions), respectively:

```
"mongodb://localhost:27016/"
```

```
"mongodb://localhost:27017/fdb-benchmark?replicaSet=rs"
```

For transactional MongoDB, connection to the database was acquired with the parameters for write and read concern both set to `"majority"`. The same parameters were used when starting transactions. For FDB-DL, a wrapper function was implemented to define explicit transactions, as described in [63].

8.1.3 Installation and application versions

Setup of the tool was performed as described in the `README.md`-file (see appendix C.1). This involves setting up and preparing all database configurations, installing required Python packages, and preparing the tool itself.

Software

- Version 6.0.15 of FoundationDB was used for all benchmarks. It was installed by following the guidelines found in the FDB documentation[7]. Both client and server was installed.
- Version 4.0.6 of MongoDB was used for all benchmarks[68]. It was installed using Homebrew for macOS.

- Version 1.6.3 of the FoundationDB Document Layer was used for all benchmarks. It was installed by following the guidelines found in the FDB-DL documentation[12]. It was configured, by default, to run on port 27016 when active.

8.1.4 Limitations and caveats

As mentioned in previous sections, this study focuses on the single core performance of the databases. Thus, this tool does not consider the intricacies of a distributed system, and has not been tested in a distributed environment.

As discussed in section 3.7.2, usage of explicit transactions was removed in version 1.7.0 of FDB-DL. As this tool relies heavily on the use of such explicit transactions, it is important to keep in mind when studying the results that the latest versions of FDB-DL may not be able to achieve similar performance. The developers of FDB-DL have outlined their intentions of reintroducing explicit transactions in the future, and while it may be reasonable to expect future versions to perform at least as good as the previous implementation, no such guarantees have been made.

8.2 Workloads

In this section, each workload is presented, along with the details of each runner-specific implementation.

All benchmarking methods within each workload generate a set of operations randomly, with weighted choices. This means that each execution may not achieve exactly the intended ratio of reads and writes, but repeated runs seem to put the variance at less than 1%. Across hundreds of benchmarking runs, this variance is relatively insignificant for this research. The distribution of requests, i.e. which documents to retrieve or update, is also generated by each benchmarking method. In terms of request distribution, five out of six workloads follow Zipf's law[92], which is the same distribution as most workloads in YCSB[87].

In general, each runner-specific method follows the same pattern across all workloads:

Standard MongoDB: The `mongo3`-runner, used for standard MongoDB, typically begins by generating a set of operations, as described in previous sections. It then iterates over the set of operations, performing each one in order, while fetching which IDs to read or write from the request distribution generated during setup.

Transactional MongoDB: The transactional configuration of MongoDB 4.0, using runner `mongo4`, will begin by initiating a session for the run, and defining a batch size. This variable can vary from workload to workload, and should be tuned to achieve the highest throughput for each workload. The runner initiates a loop, which generates a set of operations the size of the defined batch size, and performs all these operations within a new transaction. Transactions are initiated using the PyMongo API for sessions: `session.start_transaction()`. The loop continues until the total number of operations has been reached.

FoundationDB Document Layer: The Document Layer, using runner `fdbdl`, follows a pattern similar to transactional MongoDB. A batch size is defined, and operations are generated and performed within a loop, just like the `mongo4`-runner. The key difference is that transactions are initiated and performed by calling a separate method. This method is responsible for performing operations within the transaction, and must be annotated by the transactional wrapper discussed in 8.1.2.

The tool implements six workloads, mimicking the six core workloads of YCSB. It is quite simple to add additional workloads, although it has to be done programmatically. The tool does not currently support template files or similar solutions, as used by YCSB. Currently, all workloads load 1000 records into the database on setup, and all workloads consist of 10000 operations. Additionally, all workloads use a batch size of 1000 operations for the transactional MongoDB configuration, and for FoundationDB Document Layer. This can be fine-tuned per workload, if necessary. Each workload works as follows:

Workload A is a basic workload, with a 50/50 ratio of reads and updates. Each update consists of changing the contents of a single field in the document, and each read retrieves an entire document. Updates are implemented through `collection.update_one()`, and reads through `collection.find_one()`. Requests are distributed according to Zipf's law.

Workload B is quite similar to workload A, except for the ratio of reads and updates: this workload consists of 95% reads.

Workload C is another variant of workloads A and B. Configurations and request distributions are equal, but workload C consists solely of reads.

Workload D has a 95/5 ratio of reads and inserts. Each insert consists of a small document with four fields, each containing different, small values. Inserts are implemented using `collection.insert_one()`. The reads used in this workload will always access the document that was most recently inserted, making these documents quite popular.

Workload E focuses on short scanning reads, with a 95/5 ratio of range reads and inserts. The length of each scan was generated randomly from all integers between 0 and 10 (non-inclusive). Scans are implemented using `collection.find()`. For both MongoDB configurations, the method queried all documents with a key greater than or equal to a value generated by the request distribution, and was limited by the length of the scan. For FDB-DL, this approach uncovered several issues during development. Instead, FDB-DL queries for all keys matching a range of IDs, generated by the request distribution and the random scan length.

Workload F has a 50/50 ratio of reads and *read-modify-writes*. This requires fetching and reading the document before updating it, and is implemented using `collection.find_one_and_update()`.

8.3 Method and execution

All benchmarks using the tool were performed on a a mid-2017 MacBook Pro running macOS 10.14.4, with a 3.1 GHz dual-core Intel Core i5 Kaby Lake (7267U) CPU, 8 GB of RAM, and 256 GB of SSD storage. Prior to executing the benchmarks, all databases were set up and initiated according to the guidelines outlined in the `README.md` of the tool (see appendix C.1 or [91]).The following command was used to execute the benchmarks:

```
python test.py -runners mongo3 mongo4 fdbdl -workloads a b c d e f -num_runs
100
```

This initiated a benchmark of all six workloads, for all three runners, with 100 runs per workload per runner, for a total of 1800 runs. As each workload finished its 100 runs, a `.csv`-file of metrics and logs was written to disk.

Chapter 9

Results

In this chapter, results from benchmarks using the tool introduced in chapter 8 are presented. The total results of the benchmarks can be seen in figure 9.1, which shows the average throughput generated for each runner, across all workloads. The following sections break the results down in further detail; the first section details the results from each workload, while the second section groups results based on each database configuration.

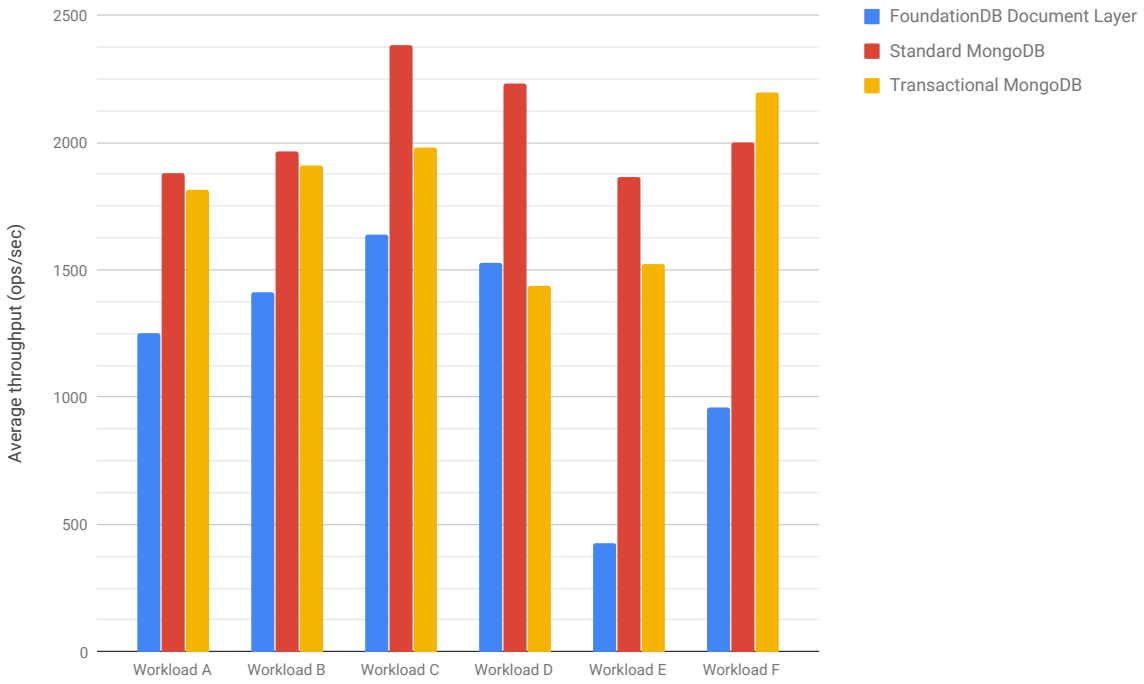

Figure 9.1: Average throughput for each database configuration, across all workloads

9.1 Workloads

This section briefly presents the average throughput generated from benchmarks of all six workloads on all three database configurations. See appendix D for complete outputs of each workload, for all three configurations.

9.1.1 Workload A

The first workload has a 50/50 ratio of reads and updates. Figure 9.2 shows the throughput achieved in workload A, for each database configuration. Standard MongoDB outperformed both other runners, as expected, but trans-

actional MongoDB is not far behind. The Document Layer delivered approximately two thirds of the throughput of the other runners. This is much improved over the preliminary results, where FDB-DL achieved less than a third of standard MongoDB.

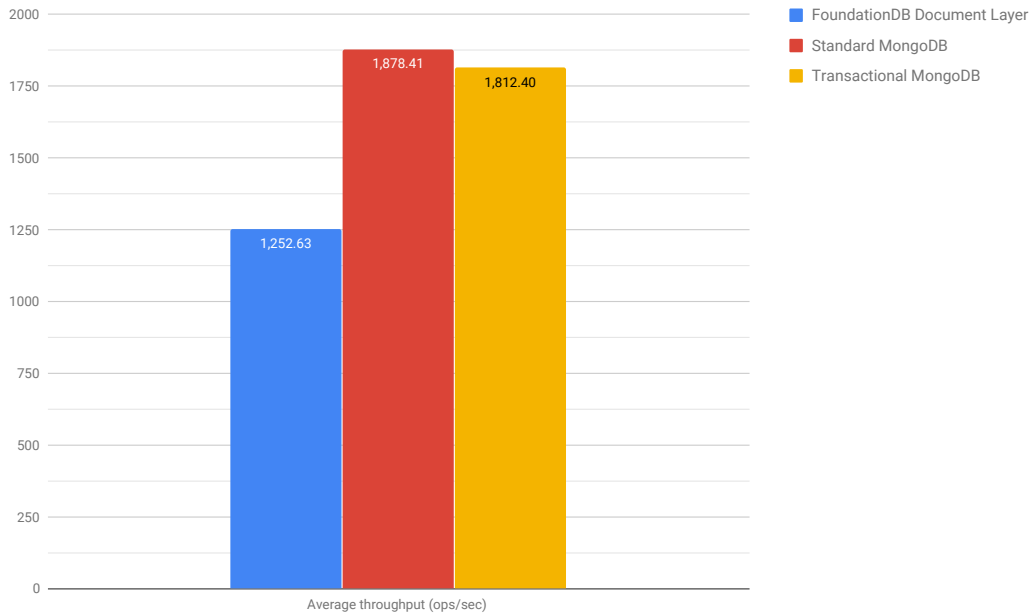

Figure 9.2: Average throughput when running workload A

9.1.2 Workload B

Workload B consists of 95% reads, and 5% updates. The results, seen in figure 9.3, are quite similar to those of Workload A. The throughput was somewhat higher for all configurations, likely due to the reduced amount of updates.

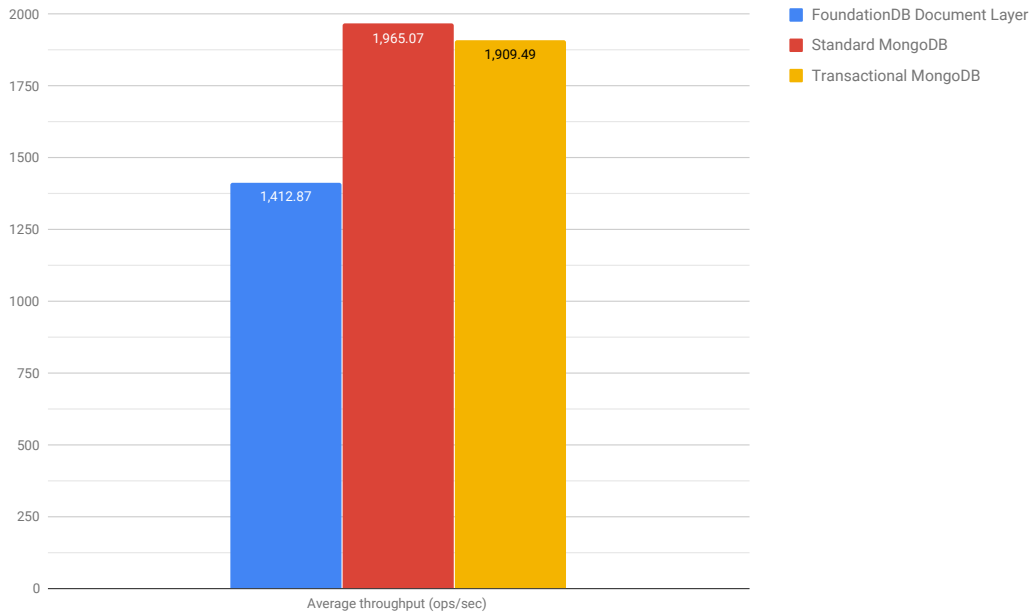

Figure 9.3: Average throughput when running workload B

9.1.3 Workload C

The results of workload C, a read-only workload, can be seen in figure 9.4. All runners achieved even higher throughput, due to the lack of costly updates. However, transactional MongoDB only achieved a marginal improvement, compared to the results of workload B.

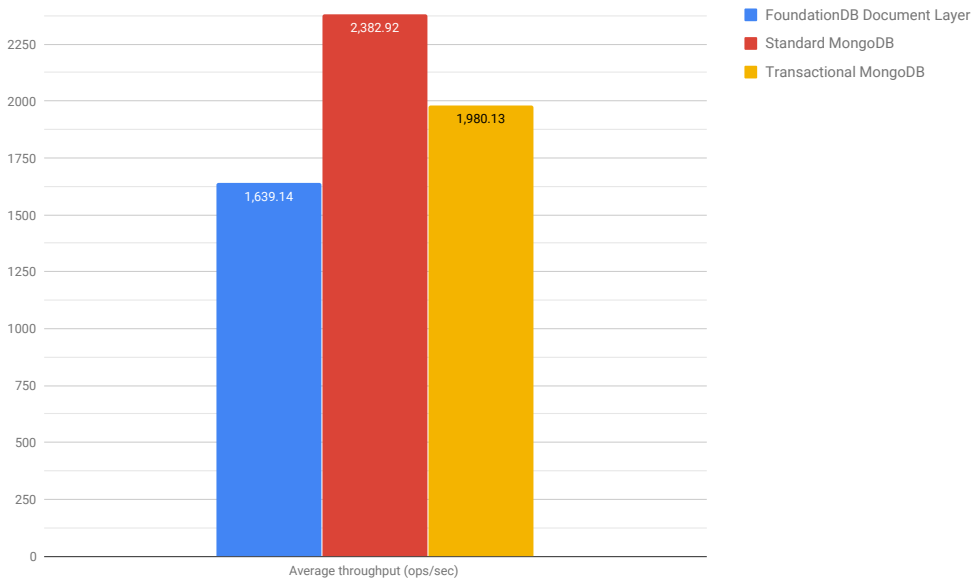

Figure 9.4: Average throughput when running workload C

9.1.4 Workload D

The results of workload D, seen in figure 9.5, are quite interesting. This workload consists of 95% reads, and 5% inserts. While the results of FDB-DL outperformed those seen in workload B (95/5 read/update), standard MongoDB achieved almost the same throughput as in workload C (100% reads). Transactional MongoDB, on the other hand, seemed to struggle with insertions, and was outperformed by FDB-DL.

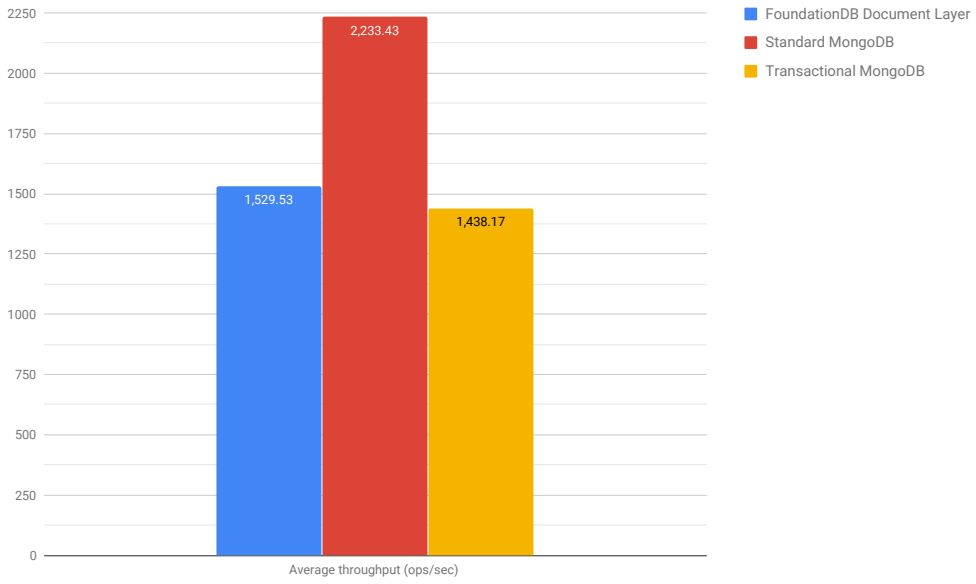

Figure 9.5: Average throughput when running workload D

9.1.5 Workload E

Workload E has a 95/5 ratio of short scans and inserts. FDB-DL seemed to struggle with range reads, just as in the preliminary tests, albeit not as much as when using YCSB (section 6.2.3). Scans slightly reduced the performance of standard MongoDB as well, whereas transactional MongoDB was left unaffected, as seen in figure 9.6.

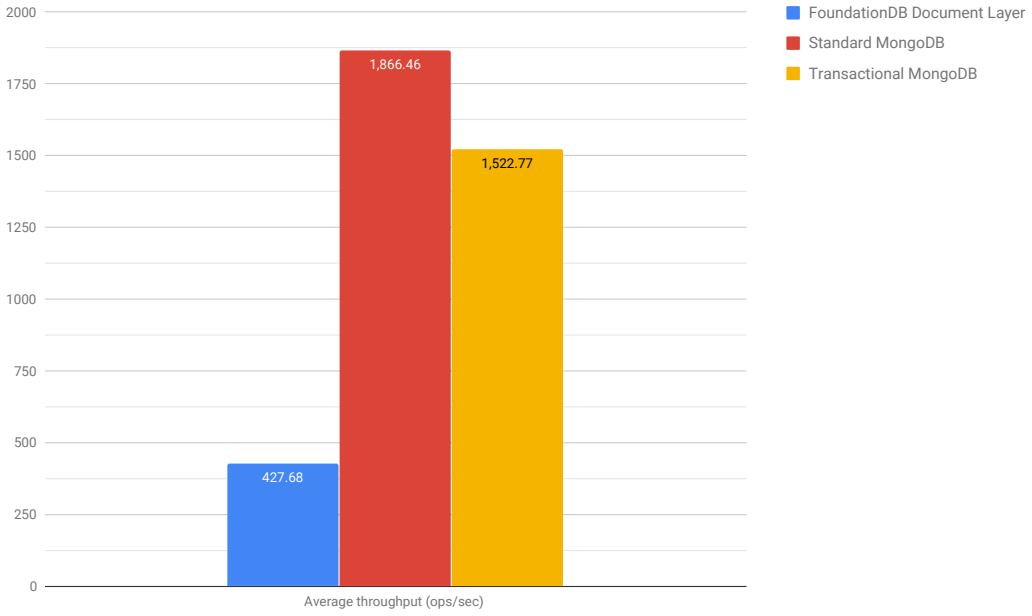

Figure 9.6: Average throughput when running workload E

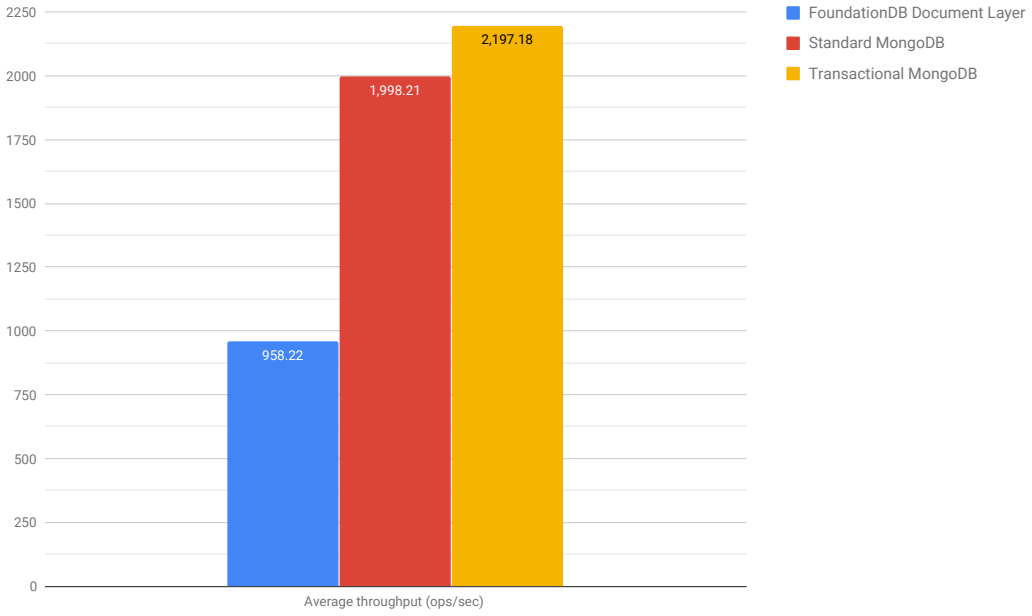

Figure 9.7: Average throughput when running workload F

9.1.6 Workload F

The final workload has a 50/50 ratio of reads to read-modify-writes (*rmw*), and results are shown in figure 9.7. An rmw requires the application to fetch a document, read it, modify a field, and then push this modification as an update. This seems to be quite costly for FDB-DL, whereas the other configurations performed quite well. It is somewhat suprising that transactional MongoDB outperformed standard MongoDB in this workload, as the rmw-operations should come with some transactional overhead.

9.2 Database configurations

This section presents the presents the outputs of all workloads, grouped by configurations, making it simple to visualize the differences between each workload on a single configuration.

9.2.1 FoundationDB Document Layer

Figure 9.8 displays how FoundationDB Document Layer performed across all workloads. It is quite evident that performance improves as the ratio of updates decreases, and it is interesting to note that inserts outperform updates. Additionally, FDB-DL struggles with certain operations, such as range reads and read-modify-writes.

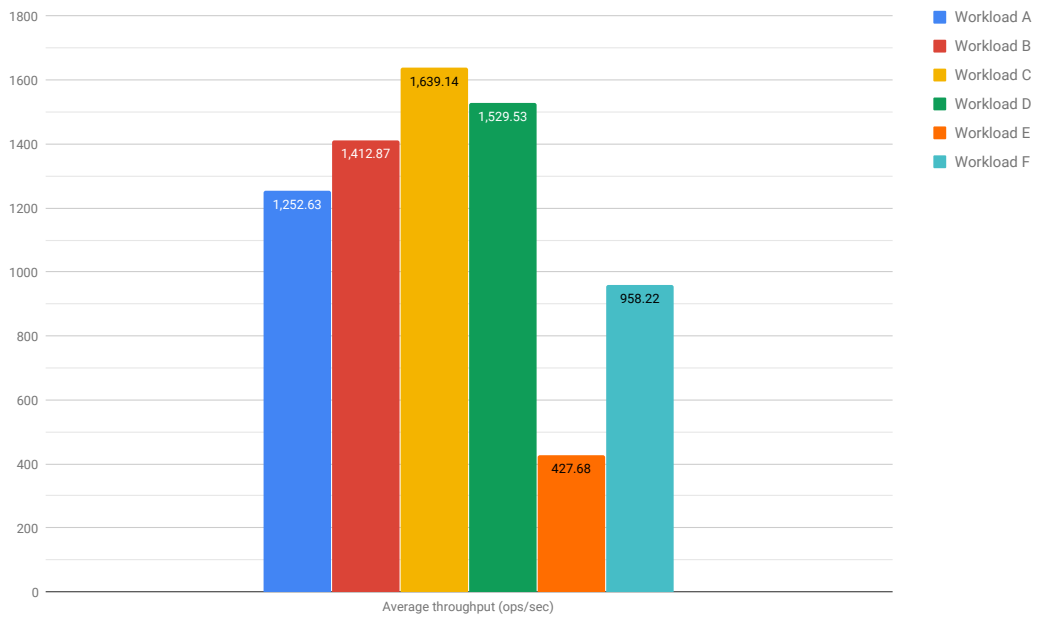

Figure 9.8: Average throughput of all workloads for runner `fdbdl`, i.e. the FoundationDB Document Layer

9.2.2 Standard MongoDB

The performance of standard MongoDB across all workloads is presented in figure 9.9. It may be interesting to note that workload F, which uses read-modify-writes, slightly outperformed workloads A and B, which make use of direct updates. Additionally, similarly to FDB-DL, inserts seemed to outperform updates.

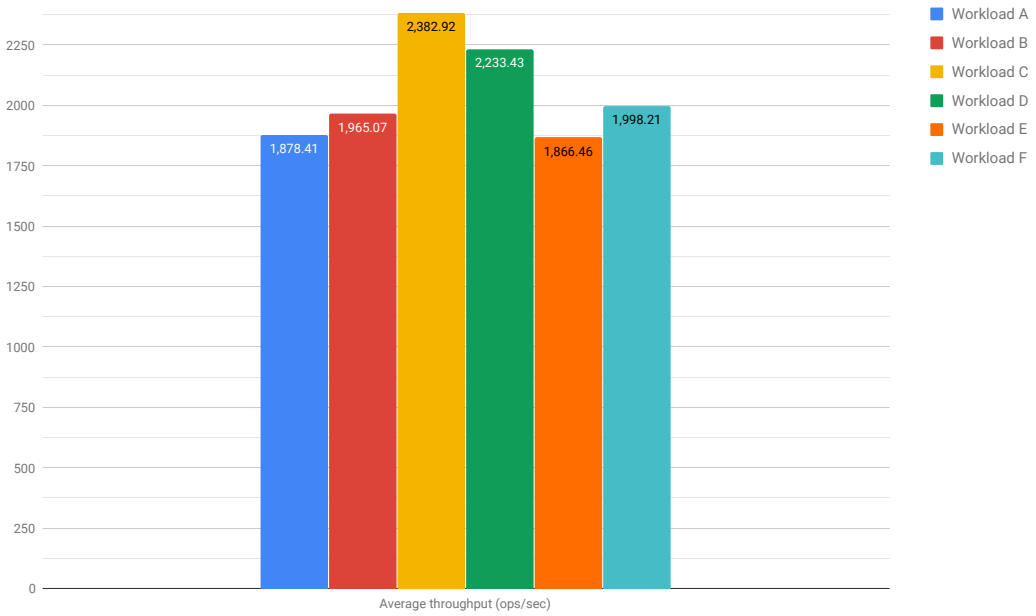

Figure 9.9: Average throughput of all workloads for runner `mongo3`, i.e. standard MongoDB

9.2.3 Transactional MongoDB

Finally, figure 9.10 presents the results of transactional MongoDB across all workloads. Inserts performed far worse than updates for this configuration, and workload F outperformed all other workloads, suprisingly.

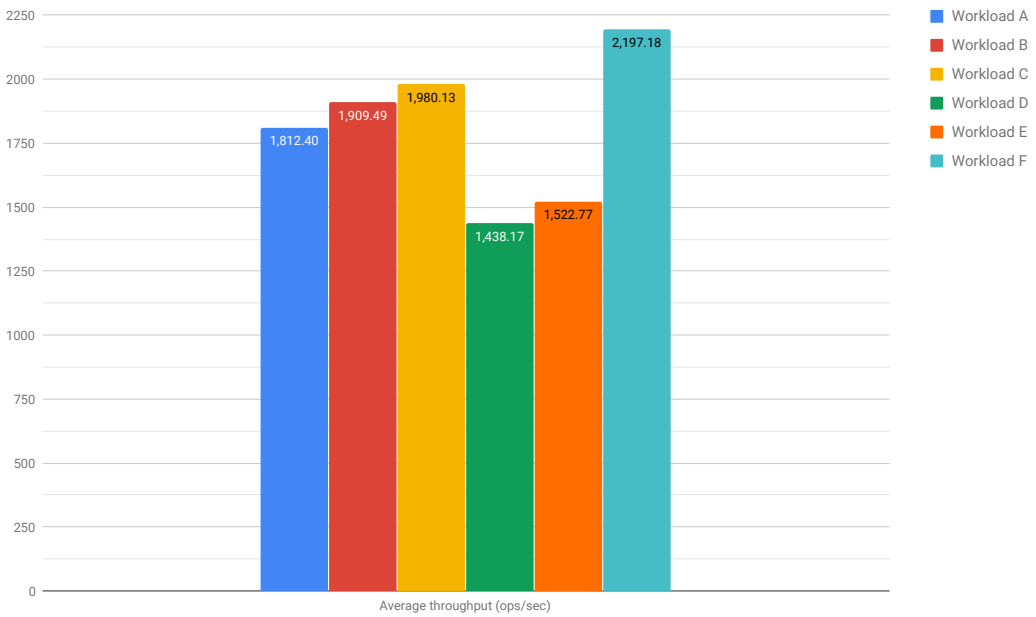

Figure 9.10: Average throughput of all workloads for runner `mongo4`, i.e. transactional MongoDB

Chapter 10

Discussion and evaluation

This chapter analyzes and evaluates the results presented in chapter 9, aiming to understand how the FoundationDB Document Layer performs in comparison to MongoDB, and perhaps to uncover how and why these results were achieved.

The first section evaluates the results of each workload, looking at the operations used and how they are handled by each configuration. The second section evaluates FDB-DL as a whole, across all workloads, while briefly summarizing the performance of transactional and standard MongoDB. The final section presents some limits and caveats to the research presented.

10.1 Evaluation of workloads

10.1.1 Workloads A, B, and C

The first three workloads generated quite predictable results: read-heavy workloads will naturally outperform workloads with many costly update-operations, as seen in figures 9.2, 9.3 and 9.4. This is even more evident in the transaction-based configurations, FDB-DL and transactional MDB, where updates also bring some overhead in terms of conflict checking. Standard MDB has no transactional overhead, and thus is only concerned with finding the correct document before issuing an update. Similarly, standard MDB far outperforms the other configurations when performing 100% reads, as FDB-DL and transactional MDB still have some overhead due to transactions.

Even with the tweaks and adaptations suggested in section 7.4, FDB-DL is not truly able to achieve the same performance as the two MongoDB configurations in these three basic workloads. The performance is improved for all three configurations, but while FDB-DL is closer to MDB in terms of performance in the custom benchmarks, there is still a ways to go. This may come down to the additional overhead created by FoundationDB's transactions, or from mapping MongoDB operations to FoundationDB key-value operations[88, 93].

10.1.2 Workload D

The results presented for workload D, seen in figure 9.5, show the difference between update-based and insertion-based workloads. Workload D is similar to workload B, but uses inserts instead of updates, and this is reflected in the throughput generated by each runner. The Document Layer and standard MongoDB show slightly better performance, compared to workload B. This is due to the fact that insertions are cheaper than updates; updates require scanning the collection for a matching document before issuing a modification, whereas inserts can generate a new primary key and create a new document straight away.

It is noteworthy that the performance of transactional MDB in workload D is ca. 25% lower than in workload B, despite the fact that workload D is insertion-based. One could argue that the Document Layer is able to compete with MongoDB due to this result, but it's more likely an issue in MongoDB 4.0 than a strong result for FDB-DL. While it would be interesting to investigate this further, it is outside the scope of this thesis. It is not related to the

evaluation of FDB-DL, and would require additional time. Also, multi-document transactions are a relatively new feature in MongoDB, and this may just be a sign that the feature is not mature yet.

10.1.3 Workload E

The fifth workload consists of 95% short range reads, and 5% insertions. Standard MongoDB performs as expected, being slightly slower than in workload D, as seen in figure 9.6. This is because the tool implements range reads as a query for a range of IDs, which is slower than a query for a single ID.

Querying for several IDs at once seems to hurt the performance of FDB-DL quite a lot, albeit less than when using YCSB (see figure 6.6). As mentioned in section 8.2, several issues with transaction design were uncovered when implementing a scanning read for FDB-DL in the benchmarking tool, and this is likely what hurt performance in YCSB as well. The replacement solution seems unable to solve the issues completely, as the complex query seems to be quite costly.

While regular FoundationDB excels at range reads[43], the Document Layer is unable to access this operation directly, and seems to struggle with mapping the MongoDB operations to appropriate FDB operations. This seems to match statements made by FDB-DL developers, outlining certain improvements that are yet to be implemented[88, 93]. Thus, scanning reads likely generate a lot of overhead.

Transactional MongoDB is interesting yet again; insertions still seem to hurt performance. However, it outperforms its own results from workload D, despite reads being cheaper than scanning reads. As previously mentioned, investigating this is outside the scope of this thesis, and may likely be due to the immaturity of transactions in MongoDB.

10.1.4 Workload F

The final workloads has a 50/50 ratio of reads and RMWs. It is quite similar to workload A, except that documents must be fetched and read before an update is issued. As seen in figure 9.7, this seems to hurt the throughput of FDB-DL by about 25%. This is likely due to the increased cost of the RMW-operation, and is to be expected.

What is more surprising, are the results of standard and transactional MongoDB. Both outperform the throughput generated in workload A, despite using supposedly more costly operations. Also, transactional MongoDB is able to outperform standard MongoDB, despite incurring a lot of transactional overhead. This may come down to how these operations are handled under the hood of MongoDB, and how they are managed within MongoDB transactions, but investigating this would be time-consuming, and outside the scope of this thesis.

10.2 Evaluation of configurations

10.2.1 FoundationDB Document Layer

Across the different workloads, seen in figure 9.8, it is quite evident that the performance of FDB-DL is heavily connected to the ratio of writes. Updates and inserts generate more transactional overhead, and require more conflict checking than a simple read, so as to maintain the ACID properties of the transactions. Additionally, there is some overhead as the Document Layer maps MongoDB operations to FoundationDB operations[88, 93].

The throughput in workload E and F is far lower than in other workloads. As previously discussed, this is likely due to poor performance in the mapping from MongoDB to FoundationDB. The performance of scanning reads in FDB-DL, in both preliminary and concluding results, is abysmal, and shows great room for improvement. It is likely that FDB-DL handles this operation as a series of separate reads, instead of a direct range read. RMW-operations show similar issues, and it is likely that a single RMW-operation must be mapped to several operations within the Document Layer. Investigating the cause for this would be interesting, but outside the scope of this thesis, as it isn't strictly relevant to whether or not FDB-DL can replace MDB in its current state.

When compared to the preliminary results, it is evident that the adaptations and customizations iplemented by the benchmarking tool improved the performance of FDB-DL. While all configurations saw performance improve-

ments compared to the YCSB results, FDB-DL was clearly the configuration that benefited the most, bringing it closer to the performance of MDB. Despite this, there is still a ways go for it to become truly competetive. It is slower in all workloads, by approximately a third, as seen in figure 9.1. Additionally, it has several use cases where it plainly is not ready yet, such as scanning reads and RMWs. While it does provide stronger consistency, and uses a multi-model database at its core, the project does not seem mature enough to replace MDB yet.

It is also noteworthy that FDB-DL only outperforms transactional MongoDB in a single workload. One would think that transactions would be something FDB-DL excelled at, seeing as it is the core, primitive feature of FoundationDB. However, there seems to be much room for improvement before it is able to truly leverage this to its benefit.

In summary, FoundationDB Document Layer seems not to be mature enough to act as a replacement for MongoDB yet. While certain adaptations to an application may bring FDB-DL closer to MDB in terms of performance, and it provides stricter consistency guarantees with a robust storage engine to back it up, there are still several key issues which must be resolved before it can truly act as a replacement for the popular document store.

10.2.2 Standard MongoDB

The results of standard MongoDB, seen in figures 9.1 and 9.9, are somewhat expected. It outperforms both other configurations, is faster when performing read-heavy workloads, and handles most use cases well. The most surprising result is seen in workload F, where it is outperformed by its own, session-based transactions.

10.2.3 Transactional MongoDB

The stricter, more consistent configuration of MongoDB performs similarly to standard MDB. It delivered a slightly lower throughput, due to transactional overhead. However, as seen in figure 9.10, there is a considerable dip in throughput when working with insertions. Also, it outperforms all other configurations when working with RMW-operations, as shown in figure 9.1. These anomalies may come down to how operations are managed and processed within the transactions of MDB, but investigating would be outside the scope of this research, as mentioned previously.

10.3 Limitations and caveats

10.3.1 Performance and environments

All benchmarks using the custom tool were performed on a single MacBook Pro, which naturally limits the performance of all runners. A single client communicating with a single server is not necessarily representative of what these databases are capable of under ideal circumstances. Also, as mentioned in section 1.3, this research focused on comparing the configurations in a single core environment. A benchmark of the databases in a highly distributed environment may reveal other intricacies to be studied. Despite this, the author is confident that the results presented by the custom benchmarks are sufficient for understanding the current capabilities and limitations of FoundationDB Document Layer.

10.3.2 Explicit transactions and future updates to the Document Layer

As discussed in sections 3.7 and 8.1, the benchmarking tool is reliant on a feature of FDB-DL known as explicit transactions to achieve the generated throughput presented in chapter 9. This feature was removed in version 1.7.0 of FDB-DL, with developers outlining a plan for its reintroduction in the future[64].

Thus, the current version of FDB-DL will not be able to match the results seen in chapter 9. The tool uses explicit transactions to batch several operations within a single transaction, reducing overhead greatly. Without this feature, each operation must be wrapped in their own transaction, causing a lot of extra work for FDB-DL, and reducing throughput by large amounts. However, it would not be unrealistic to assume that the performance of FDB-DL is at least as good, if not better, when the feature is re-implemented in the future.

10.3.3 Insight and scope

As discussed several times throughout this thesis, it is difficult to attain deep insight into the inner workings of each database, especially within the time constraints presented by a master's thesis. The information available is often presented through documentation written by the developers themselves, and the project depends on the author's understanding of this documentation.

Thus, there may be many other ways to implement the benchmarking tool, and likely some optimal solution for adapting the workloads to the intricacies of FDB-DL. While it would be interesting to truly understand *why* FDB-DL performs as it does, and to figure out certain suprising results generated by the two MongoDB configurations, such research is outside the scope of this thesis.

Chapter 11

Conclusion and further work

This chapter concludes the thesis, outlining the insight produced from the research goals presented in section 1.2, and whether or not FoundationDB Document Layer fulfilled its promise as a viable replacement for MongoDB. The conclusion is followed by sections on related work, and further research to be done.

11.1 Conclusion

This thesis has explored and evaluated the architecture and features of FoundationDB and its Document Layer. By guaranteeing ACID properties without sacrificing performance, scalability, and availability, the core of FDB has taken a new and different approach to delivering a NOSQL system. The robust and minimalistic key-value store makes no assumptions about the needs of the user, making specialized configurations available to the user through customized *layers*. This multi-model data store can be adapted to each use case, making it a possible replacement for several systems at once. Compared to MongoDB, FoundationDB provides stronger guarantees, without the use of locks, and a simpler, yet efficient, approach to redundancy and scaling. However, it lacks several of MDB's most compelling features, such as a rich query language, index structures, and an aggregation framework.

The Document Layer provides a document store on top of FoundationDB, with all the perks and strong guarantees of the robust key-value store. It acts as a replacement for MongoDB in existing systems, uses the MongoDB protocol, and is compatible with all official MongoDB drivers. It employs ACID-compliant transactions, and provides simpler horizontal scaling. It has tools for building indexes, but lacks certain key features from MongoDB, such as an aggregation framework and complex queries.

In terms of performance, the FoundationDB Document Layer falls short. While it does provide ACID-compliant transactions, it incurs a large amount of overhead when doing so. While one could argue that the difference in throughput is small enough to justify replacing MongoDB with the Document Layer, there are several other factors that contest this. NOSQL systems rose to prominence due to a need for highly distributed, high performance systems. Although making adaptations in an application will help improve the performance of the Document Layer, it does not yet achieve the high performance expected of such a system. More importantly, the implementation of several operations seems to be lacking or not yet mature, making the Document Layer struggle with certain workloads.

Therefore, this thesis concludes that the FoundationDB Document Layer is not yet viable as a replacement for MongoDB. A system built from the ground up as a document store will likely always outperform a system that must be adapted and modified to perform the same tasks. Such customizations generate too much overhead for it to remain competitive. To become truly effective, FDB-DL would likely have to be completely incorporated into the core of FDB. However, this would be a far cry from the design principles of FDB, which aims to design a focused, minimalist core, without extraneous features. It would likely be more prudent for FDB to focus on what it does well (a robust, scalable key-value store, with ACID transactions), than to weaken its core with several expansions.

11.2 Related work

11.2.1 MongoDB and DynamoDB transactions

As mentioned throughout this thesis, MongoDB released fully ACID transactions in version 4.0, achieving much of what FDB-DL intends to deliver. However, the implementation is not yet complete: it does not work across sharded clusters yet[79]. Also, transactions in MDB rely on locks and sessions, unlike the approach used in FoundationDB.

DynamoDB has also implemented ACID transactions recently[94]. However, they lose some guarantees when operating across several regions, and they do not support mixed operations within a single transaction. In other words, transactions can either be write-transactions or read-transactions.

Additionally, MDB and DynamoDB do not provide the customizability of FDB. A user is forced to select a data model, query engine and storage engine at once, instead of separating the systems into separate decisions[24].

11.2.2 ArangoDB

ArangoDB is a multi-model NOSQL database, which supports ACID transactions[95]. However, these can only be multi-document or multi-collection when running on a single instance[96]. When using a cluster, transactions are only ACID on a single-document basis. While ArangoDB is multi-model, it does not support custom data models[95].

11.3 Further work

The most obvious choice for further work would be to amend the flaws and poor implementations uncovered in FDB-DL by this research. The software is open source, and open for contributions from everyone on Github[8, 25]. This would help FDB-DL move closer to MongoDB in terms of performance, making it more viable as a replacement, and this research can hopefully help identify where to focus further development.

Testing and benchmarking the Document Layer in a distributed environment is also desirable. This would make it possible to compare how the two databases function in a more realistic setting, and how they deal with operational issues. It could reveal even more about FDB-DL's current state, and help guide further development as well.

While FoundationDB Document Layer proposes a certain solution for replacing MongoDB, it could still be interesting to look at another approach. FDB-DL acts as a middleman between MongoDB drivers and the FDB core; a complete, stand-alone replacement has not yet been developed.

It would also be relevant to explore other NOSQL databases, and investigate how FDB can be used to create replacement for these. This would provide legitimacy to the assertion that FDB can be used as a single replacement system for several databases. Additionally, it would be useful to carry out performance tests of MDB and FDB-DL, or even an implementation of the proposed approach.

FoundationDB also provides some interesting fields of research and further development, unrelated to its viability as a replacement for other systems. For instance, FDB has undergone extensive testing and simulation to ensure that ACID properties are preserved, even under extreme edge cases and failures[51]. The deterministic simulation tool, which is capable of generating random failures in all parts of the system, could possibly be extracted for more general use cases, or studied to investigate the validity of FDB's consistency and performance claims.

There are also some known limitations to FoundationDB, which would be interesting to solve[54]. As an example, FDB currently does not increase the replication factor of frequently accessed data. Implementing this would help balance loads between storage nodes, potentially increasing the read performance of FDB.

Bibliography

[1] Apple Inc. and the FoundationDB project authors: *Architecture*. https://apple.github.io/foundationdb/architecture.html. Accessed: 2018-11-28.

[2] Cisco Systems: *Cisco Visual Networking Index: Forecast and Methodology, 2008-2013*. https://www.cisco.com/c/dam/global/pt_br/assets/docs/whitepaper_VNI_06_09.pdf, 2009. Accessed: 2018-11-24.

[3] Cisco: *Cisco Visual Networking Index: Forecast and Methodology, 2016-2021*. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf, 2017. Accessed: 2018-11-24.

[4] Ramez Elmasri and Sham Navathe: *Fundamentals of database systems, 7th edition*, pages 883–888, 909. Pearson London, 2016.

[5] Ramez Elmasri and Sham Navathe: *Fundamentals of database systems, 7th edition*, pages 888–890. Pearson London, 2016.

[6] Apple Inc.: *FoundationDB*. https://www.foundationdb.org/. Accessed: 2018-11-15.

[7] Apple Inc. and the FoundationDB project authors: *FoundationDB Documentation*. https://apple.github.io/foundationdb/. Accessed: 2018-11-15.

[8] Apple Inc. and the FoundationDB project authors: *FoundationDB GitHub Repository*. https://github.com/apple/foundationdb. Accessed: 2018-11-15.

[9] Megha Katkar: *Performance analysis for nosql and sql*. International Journal of Innovative and Emerging Research in Engineering, 2(3):12–17, 2015. http://www.ijiere.com/FinalPaper/FinalPaper20153153146465.pdf.

[10] Apple Inc. and the FoundationDB project authors: *Performance*. https://apple.github.io/foundationdb/performance.html. Accessed: 2018-11-28.

[11] Apple Inc. and the FoundationDB project authors: *Announcing FoundationDB Document Layer*. https://www.foundationdb.org/blog/announcing-document-layer/. Accessed: 2018-12-3.

[12] Apple Inc. and the FoundationDB project authors: *FoundationDB Document Layer Documentation*. https://foundationdb.github.io/fdb-document-layer/. Accessed: 2018-11-30.

[13] Patrick zsu, M. Tamer; Valduriez: *Principles of Distributed Database Systems, Third Edition*, pages 335–344. Springer, 2011.

[14] Patrick zsu, M. Tamer; Valduriez: *Principles of Distributed Database Systems, Third Edition*, pages 344–349. Springer, 2011.

[15] FoundationDB: *The Transaction Manifesto*. https://apple.github.io/foundationdb/transaction-manifesto.html. Accessed: 2018-11-28.

[16] FoundationDB: *CAP Theorem*. https://apple.github.io/foundationdb/cap-theorem.html. Accessed: 2018-11-28.

[17] Ramez Elmasri and Sham Navathe: *Fundamentals of database systems, 7th edition*, pages 890–909. Pearson London, 2016.

[18] solid IT: *DB-Engines Ranking of Key-value Stores*. https://db-engines.com/en/ranking/key-value+store. Accessed: 2018-12-6.

[19] solid IT: *DB-Engines Ranking of Wide Column Stores*. https://db-engines.com/en/ranking/wide+column+store. Accessed: 2018-12-6.

[20] solid IT: *Wide Column Stores*. https://db-engines.com/en/article/Wide+Column+Stores. Accessed: 2018-12-6.

[21] solid IT: *Graph DBMS*. https://db-engines.com/en/article/Graph+DBMS. Accessed: 2018-12-6.

[22] solid IT: *DB-Engines Ranking of Graph DBMS*. https://db-engines.com/en/ranking/graph+dbms. Accessed: 2018-12-6.

[23] solid IT: *DB-Engines Ranking of Document Stores*. https://db-engines.com/en/ranking/document+store. Accessed: 2018-12-6.

[24] Apple Inc. and the FoundationDB project authors: *FoundationDB Documentation - Layer Concept*. https://apple.github.io/foundationdb/layer-concept.html. Accessed: 2018-11-28.

[25] Apple Inc. and the FoundationDB project authors: *FoundationDB GitHub Organization*. https://github.com/foundationdb. Accessed: 2019-03-12.

[26] Matthew Panzarino — TechCrunch: *Apple acquires durable database company FoundationDB*. https://techcrunch.com/2015/03/24/apple-acquires-durable-database-company-foundationdb/. Accessed: 2018-11-28.

[27] Klint Finley — WIRED Magazine: *Database house wants you to stop dropping ACID*. https://www.wired.com/2013/03/foundationdb/. Accessed: 2018-11-28.

[28] Saroj Kar — Silicon Angle: *FoundationDB Goes Public, Releases NoSQL Database with Unique ACID Transactions*. https://siliconangle.com/2013/08/21/foundationdb-goes-public-releases-nosql-database-with-unique-acid-transactions/. Accessed: 2018-11-28.

[29] Julie Bort — Business Insider: *Apple just bought a tiny database company called FoundationDB*. https://www.businessinsider.com/apple-just-bought-a-tiny-database-company-called-foundationdb-2015-3?r=US&IR=T&IR=T. Accessed: 2018-11-28.

[30] Hackernews Forum: *Apple Acquires FoundationDB*. https://news.ycombinator.com/item?id=9259986. Accessed: 2018-11-28.

[31] Julie Bort — Business Insider: *A lot of people are mad that Apple bought this tiny company and shut it down*. https://www.businessinsider.com/why-apple-bought-foundationdb-2015-3?r=US&IR=T&IR=T. Accessed: 2018-11-28.

[32] FoundationDB: *FoundationDB Community*. https://web.archive.org/web/20150325003206/http://community.foundationdb.com/. Accessed: 2018-11-28, via Internet Archive.

[33] Apple Inc. and the FoundationDB project authors: *FoundationDB is Open Source*. https://www.foundationdb.org/blog/foundationdb-is-open-source/. Accessed: 2018-11-28.

[34] Hackernews Forum: *Apple open-sources FoundationDB*. https://news.ycombinator.com/item?id=16877395. Accessed: 2018-11-28.

[35] Apple Inc. and the FoundationDB project authors: *Announcing the FoundationDB Record Layer*. https://www.foundationdb.org/blog/announcing-record-layer/. Accessed: 2019-3-12.

[36] Christos Chrysafis, Ben Collins, Scott Dugas, Jay Dunkelberger, Moussa Ehsan, Scott Gray, Alec Grieser, Ori Herrnstadt, Kfir Lev-Ari, Tao Lin, Mike McMahon, Nicholas Schiefer, and Alexander Shraer: *FoundationDB Record Layer: A Multi-Tenant Structured Datastore*. https://www.foundationdb.org/files/record-layer-paper.pdf. Apple. Inc, Accessed: 2019-3-12.

[37] Apple Inc. and the FoundationDB project authors: *Release notes*. https://apple.github.io/foundationdb/release-notes.html. Accessed: 2018-11-28.

[38] Apple Inc. and the FoundationDB project authors: *Earlier release notes*. https://apple.github.io/foundationdb/earlier-release-notes.html. Accessed: 2018-11-28.

[39] Apple Inc. and the FoundationDB project authors: *Administration*. https://apple.github.io/foundationdb/administration.html. Accessed: 2018-11-28.

[40] *FoundationDB Forum Post: Technical overview of the database*. https://forums.foundationdb.org/t/technical-overview-of-the-database/135. Accessed: 2018-11-20.

[41] Apple Inc. and the FoundationDB project authors: *FoundationDB Architecture*. https://apple.github.io/foundationdb/kv-architecture.html. Accessed: 2018-11-28.

[42] Apple Inc. and the FoundationDB project authors: *FoundationDB Documentation - Data modeling*. https://apple.github.io/foundationdb/data-modeling.html. Accessed: 2018-11-25.

[43] Apple Inc. and the FoundationDB project authors: *Features*. https://apple.github.io/foundationdb/features.html. Accessed: 2018-11-28.

[44] Apple Inc. and the FoundationDB project authors: *Configuration*. https://apple.github.io/foundationdb/configuration.html. Accessed: 2018-11-28.

[45] *FoundationDB Forum Post: Understanding inter communication*. https://forums.foundationdb.org/t/understanding-inter-communication/745. Accessed: 2018-11-25.

[46] Apple Inc. and the FoundationDB project authors: *Fault Tolerance*. https://apple.github.io/foundationdb/fault-tolerance.html. Accessed: 2018-11-28.

[47] Apple Inc. and the FoundationDB project authors: *Engineering*. https://apple.github.io/foundationdb/engineering.html. Accessed: 2018-11-28.

[48] Apple Inc. and the FoundationDB project authors: *Transaction Processing*. https://apple.github.io/foundationdb/transaction-processing.html. Accessed: 2018-11-28.

[49] Steve Atherton — The Linux Foundation via Youtube: *Future of FoundationDB Storage Engines - Steve Atherton, Apple*. https://www.youtube.com/watch?v=nlus1Z7TVTI. Accessed: 2019-3-12.

[50] Apple Inc. and the FoundationDB project authors: *Scalability*. https://apple.github.io/foundationdb/scalability.html. Accessed: 2018-11-28.

[51] Apple Inc. and the FoundationDB project authors: *Simulation and Testing*. https://apple.github.io/foundationdb/testing.html. Accessed: 2018-11-28.

[52] Will Wilson — Strange Loop via YouTube: *"Testing Distributed Systems w/ Deterministic Simulation" by Will Wilson*. https://www.youtube.com/watch?v=4fFDFbi3toc. Accessed: 2018-11-28.

[53] Apple Inc. and the FoundationDB project authors: *Anti-Features*. https://apple.github.io/foundationdb/anti-features.html. Accessed: 2018-11-28.

[54] Apple Inc. and the FoundationDB project authors: *Known limitations*. https://apple.github.io/foundationdb/known-limitations.html. Accessed: 2018-11-28.

[55] Alex Handy — SDTimes: *FoundationDB brings transactions to NoSQL*. https://sdtimes.com/databases/foundationdb-brings-transactions-to-nosql/. Accessed: 2018-11-30.

[56] Gavin Clarke — The Register: *NoSQL's CAP theorem busters: We don't drop ACID*. https://www.theregister.co.uk/2012/11/22/foundationdb_fear_of_cap_theorem/. Accessed: 2018-11-30.

[57] Apple Inc. and the FoundationDB project authors: *API Reference*. https://apple.github.io/foundationdb/api-reference.html. Accessed: 2018-11-30.

[58] Apple Inc. and the FoundationDB project authors: *Design recipes*. https://apple.github.io/foundationdb/design-recipes.html. Accessed: 2018-11-30.

[59] Apple Inc. and the FoundationDB project authors: *Developer guide*. https://apple.github.io/foundationdb/developer-guide.html. Accessed: 2018-11-30.

[60] Apple Inc. and the FoundationDB project authors: *FoundationDB Document Layer Architecture*. https://foundationdb.github.io/fdb-document-layer/architecture.html. Accessed: 2018-11-30.

[61] Apple Inc. and the FoundationDB project authors: *Simple index*. https://apple.github.io/foundationdb/simple-indexes.html#. Accessed: 2018-11-30.

[62] Apple Inc. and the FoundationDB project authors: *Known differences*. https://foundationdb.github.io/fdb-document-layer/known-differences.html. Accessed: 2018-12-3.

[63] Apple Inc. and the FoundationDB project authors: *Transactions*. https://github.com/FoundationDB/fdb-document-layer/blob/d81badfec7abaccf50844b382f0712de85a442a5/docs/transactions.md. Accessed: 2019-5-1.

[64] Apple Inc. and the FoundationDB project authors: *Remove explicit transactions*. https://github.com/FoundationDB/fdb-document-layer/pull/150. Accessed: 2019-5-1.

[65] DB-Engines: *MongoDB System Properties*. https://db-engines.com/en/system/MongoDB. Accessed: 2018-12-1.

[66] Derrick Harris — GIGAOM: *10gen embraces what it created, becomes MongoDB Inc.* https://gigaom.com/2013/08/27/10gen-embraces-what-it-created-becomes-mongodb-inc/. Accessed: 2018-12-1.

[67] MongoDB Inc.: *What is MongoDB?* https://www.mongodb.com/what-is-mongodb. Accessed: 2018-12-1.

[68] MongoDB Inc.: *MongoDB Documentation*. https://docs.mongodb.com/. Accessed: 2018-12-1.

[69] MongoDB Inc.: *MongoDB Drivers and ODM*. https://docs.mongodb.com/ecosystem/drivers/. Accessed: 2018-12-1.

[70] MongoDB Inc.: *Introduction to MongoDB*. https://docs.mongodb.com/manual/introduction/. Accessed: 2018-12-1.

[71] MongoDB Inc.: *MongoDB CRUD Operations*. https://docs.mongodb.com/manual/crud/. Accessed: 2018-12-1.

[72] MongoDB Inc.: *Aggregation Pipeline*. https://docs.mongodb.com/manual/core/aggregation-pipeline/. Accessed: 2018-12-1.

[73] MongoDB Inc.: *Indexes*. https://docs.mongodb.com/manual/indexes/. Accessed: 2018-12-1.

[74] MongoDB Inc.: *Replication*. https://docs.mongodb.com/manual/replication/. Accessed: 2018-12-1.

[75] MongoDB Inc.: *Sharding*. https://docs.mongodb.com/manual/sharding/. Accessed: 2018-12-1.

[76] MongoDB Inc.: *Storage engines*. https://docs.mongodb.com/manual/core/storage-engines/. Accessed: 2018-12-1.

[77] MongoDB Inc.: *WiredTiger storage engine*. https://docs.mongodb.com/manual/core/wiredtiger/. Accessed: 2018-12-1.

[78] MongoDB Inc.: *WiredTiger: Log-Structured Merge trees*. http://source.wiredtiger.com/3.1.0/lsm.html. Accessed: 2019-4-30.

[79] MongoDB Inc.: *Transactions*. https://docs.mongodb.com/manual/core/transactions/. Accessed: 2018-12-1.

[80] Yahoo! Inc.: *Yahoo Cloud Serving Benchmark*. https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/. Accessed: 2019-4-17.

[81] Brian Frank Cooper, YCSB project authors, and contributors: *YCSB GitHub Project*. https://github.com/brianfrankcooper/YCSB. Accessed: 2019-4-17.

[82] Apple Inc. and the FoundationDB project authors: *FoundationDB Test Suite source code*. https://github.com/apple/foundationdb/tree/master/tests. Accessed: 2019-4-17.

[83] *FoundationDB Forum Post: ANN: benchmark FoundationDB with Go YCSB*. https://forums.foundationdb.org/t/ann-benchmark-foundationdb-with-go-ycsb/318. Accessed: 2019-4-17.

[84] *FoundationDB Forum Post: How to compile and debug FDB (on mac os)?* https://forums.foundationdb.org/t/how-to-complie-and-debug-fdb-on-mac-os/176/11. Accessed: 2019-4-17.

[85] Docker Inc: *Enterprise Container Platform for High Velocity Innovation*. https://www.docker.com/. Accessed: 2019-4-17.

[86] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears: *Benchmarking Cloud Serving Systems with YCSB*. https://www2.cs.duke.edu/courses/fall13/cps296.4/838-CloudPapers/ycsb.pdf, 2010. Accessed: 2019-4-17.

[87] Brian Frank Cooper, YCSB project authors, and contributors: *Core Workloads*. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. Accessed: 2019-4-17.

[88] *FoundationDB Forum Post: 1 M rows write via Mongo client using document layer*. https://forums.foundationdb.org/t/1-m-rows-write-via-mongo-client-using-document-layer/1157. Accessed: 2019-4-17.

[89] MongoDB Inc.: *Write Concern*. https://docs.mongodb.com/manual/reference/write-concern/. Accessed: 2019-5-3.

[90] Apple Inc. and the FoundationDB project authors: *Benchmarking*. https://apple.github.io/foundationdb/benchmarking.html. Accessed: 2019-4-24.

[91] Edvard Viggaklev Bakken: *Custom benchmarking tool - fdb-benchmarks Github repository at latest commit at time of delivery*. https://github.com/edvardvb/fdb-benchmarks/tree/0ab29ff539f6a8a0fd81a58c71909c1def30bdaa. Accessed: 2019-5-6.

[92] William L. Hosch: *Zipf's Law*. https://www.britannica.com/topic/Zipfs-law. Accessed: 2019-5-9.

[93] *FoundationDB Forum Post: Document Layer Performance*. https://forums.foundationdb.org/t/document-layer-performance/1255/2. Accessed: 2019-5-7.

[94] Danilo Poccia — AWS News Blog: *New Amazon DynamoDB Transactions*. https://aws.amazon.com/blogs/aws/new-amazon-dynamodb-transactions/. Accessed: 2018-12-4.

[95] ArangoDB: *What you can't do with MongoDB*. https://www.arangodb.com/why-arangodb/arangodb-vs-mongodb/. Accessed: 2018-12-4.

[96] ArangoDB: *Limitations*. https://docs.arangodb.com/3.3/Manual/Transactions/Limitations.html. Accessed: 2018-12-4.

Appendix A

Native test suite output

This table shows the complete output from running the test `RandomRead.txt` in the built-in test framework of FoundationDB.

Metric	Output value
Transactions/sec	2139.80
Operations/sec	21398.0
A Transactions	21398.0
B Transactions	0.0
Retries	0.0
Mean load time (seconds)	0.0
Read rows	213980.0
Write rows	0.0
Mean Latency (ms)	230.903154
Median Latency (ms, averaged)	248.797894
90% Latency (ms, averaged)	266.732454
98% Latency (ms, averaged)	276.864529
Max Latency (ms, averaged)	289.610386
Mean Row Read Latency (ms)	11.064244
Median Row Read Latency (ms, averaged)	10.884285
Max Row Read Latency (ms, averaged)	43.063879
Mean Total Read Latency (ms)	11.172971
Median Total Read Latency (ms, averaged)	11.0633
Max Total Latency (ms, averaged)	43.063879
Mean GRV Latency (ms)	219.657581
Median GRV Latency (ms, averaged)	237.621784
Max GRV Latency (ms, averaged)	285.488605
Mean Commit Latency (ms)	0.0
Median Commit Latency (ms, averaged)	0.0
Max Commit Latency (ms, averaged)	0.0
Read rows/sec	21398.0
Write rows/sec	0.0
Bytes read/sec	684736.0
Bytes written/sec	0.0

Table A.1: Formatted output from the native test framework of FoundationDB

Appendix B

YCSB benchmark outputs

The following sections contain tables with complete outputs for all benchmarks described in section 5.2.

B.1 FoundationDB benchmark

This section contains solely YCSB benchmarks of the default FoundationDB configuration.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	1221.001221001221
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	5
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.6105006105006106
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	5
[TOTAL˙GC˙TIME˙%]	Time(%)	0.6105006105006106
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	42.0
[CLEANUP]	MinLatency(us)	42
[CLEANUP]	MaxLatency(us)	42
[CLEANUP]	95thPercentileLatency(us)	42
[CLEANUP]	99thPercentileLatency(us)	42
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	391.649
[INSERT]	MinLatency(us)	1
[INSERT]	MaxLatency(us)	85631
[INSERT]	95thPercentileLatency(us)	32
[INSERT]	99thPercentileLatency(us)	2315
[INSERT]	Return=OK	1000

Table B.1: Complete output from loading workload A on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	687.757909215956
[TOTAL˙GCS˙PS˙Scavenge]	Count	0
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.0
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	0
[TOTAL˙GC˙TIME]	Time(ms)	0
[TOTAL˙GC˙TIME˙%]	Time(%)	0.0
[READ]	Operations	953
[READ]	AverageLatency(us)	934.5582371458552
[READ]	MinLatency(us)	329
[READ]	MaxLatency(us)	16671
[READ]	95thPercentileLatency(us)	2389
[READ]	99thPercentileLatency(us)	5107
[READ]	Return=OK	953
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	52.0
[CLEANUP]	MinLatency(us)	52
[CLEANUP]	MaxLatency(us)	52
[CLEANUP]	95thPercentileLatency(us)	52
[CLEANUP]	99thPercentileLatency(us)	52
[UPDATE]	Operations	47
[UPDATE]	AverageLatency(us)	7692.510638297872
[UPDATE]	MinLatency(us)	4812
[UPDATE]	MaxLatency(us)	18063
[UPDATE]	95thPercentileLatency(us)	14095
[UPDATE]	99thPercentileLatency(us)	18063
[UPDATE]	Return=OK	47

Table B.2: Complete output from running workload A on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	1295.3367875647668
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	6
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.7772020725388601
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	6
[TOTAL˙GC˙TIME˙%]	Time(%)	0.7772020725388601
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	47.0
[CLEANUP]	MinLatency(us)	47
[CLEANUP]	MaxLatency(us)	47
[CLEANUP]	95thPercentileLatency(us)	47
[CLEANUP]	99thPercentileLatency(us)	47
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	452.166
[INSERT]	MinLatency(us)	1
[INSERT]	MaxLatency(us)	104191
[INSERT]	95thPercentileLatency(us)	37
[INSERT]	99thPercentileLatency(us)	2585
[INSERT]	Return=OK	1000

Table B.3: Complete output from loading workload B on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	687.757909215956
[TOTAL˙GCS˙PS˙Scavenge]	Count	0
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.0
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	0
[TOTAL˙GC˙TIME]	Time(ms)	0
[TOTAL˙GC˙TIME˙%]	Time(%)	0.0
[READ]	Operations	953
[READ]	AverageLatency(us)	934.5582371458552
[READ]	MinLatency(us)	329
[READ]	MaxLatency(us)	16671
[READ]	95thPercentileLatency(us)	2389
[READ]	99thPercentileLatency(us)	5107
[READ]	Return=OK	953
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	52.0
[CLEANUP]	MinLatency(us)	52
[CLEANUP]	MaxLatency(us)	52
[CLEANUP]	95thPercentileLatency(us)	52
[CLEANUP]	99thPercentileLatency(us)	52
[UPDATE]	Operations	47
[UPDATE]	AverageLatency(us)	7692.510638297872
[UPDATE]	MinLatency(us)	4812
[UPDATE]	MaxLatency(us)	18063
[UPDATE]	95thPercentileLatency(us)	14095
[UPDATE]	99thPercentileLatency(us)	18063
[UPDATE]	Return=OK	47

Table B.4: Complete output from running workload B on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	2105.2631578947367
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	4
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.8421052631578947
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	4
[TOTAL˙GC˙TIME˙%]	Time(%)	0.8421052631578947
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	45.0
[CLEANUP]	MinLatency(us)	45
[CLEANUP]	MaxLatency(us)	45
[CLEANUP]	95thPercentileLatency(us)	45
[CLEANUP]	99thPercentileLatency(us)	45
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	330.481
[INSERT]	MinLatency(us)	1
[INSERT]	MaxLatency(us)	63423
[INSERT]	95thPercentileLatency(us)	33
[INSERT]	99thPercentileLatency(us)	3657
[INSERT]	Return=OK	1000

Table B.5: Complete output from loading workload C on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	108.23682216690118
[TOTAL˙GCS˙PS˙Scavenge]	Count	0
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.0
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	0
[TOTAL˙GC˙TIME]	Time(ms)	0
[TOTAL˙GC˙TIME˙%]	Time(%)	0.0
[READ]	Operations	1000
[READ]	AverageLatency(us)	8972.663
[READ]	MinLatency(us)	465
[READ]	MaxLatency(us)	646143
[READ]	95thPercentileLatency(us)	14415
[READ]	99thPercentileLatency(us)	225023
[READ]	Return=OK	1000
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	206.0
[CLEANUP]	MinLatency(us)	206
[CLEANUP]	MaxLatency(us)	206
[CLEANUP]	95thPercentileLatency(us)	206
[CLEANUP]	99thPercentileLatency(us)	206

Table B.6: Complete output from running workload C on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	440.33465433729634
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	4
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.17613386173491855
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	4
[TOTAL˙GC˙TIME˙%]	Time(%)	0.17613386173491855
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	49.0
[CLEANUP]	MinLatency(us)	49
[CLEANUP]	MaxLatency(us)	49
[CLEANUP]	95thPercentileLatency(us)	49
[CLEANUP]	99thPercentileLatency(us)	49
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	1458.026
[INSERT]	MinLatency(us)	1
[INSERT]	MaxLatency(us)	689151
[INSERT]	95thPercentileLatency(us)	34
[INSERT]	99thPercentileLatency(us)	17855
[INSERT]	Return=OK	1000

Table B.7: Complete output from loading workload D on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	652.7415143603133
[TOTAL˙GCS˙PS˙Scavenge]	Count	0
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.0
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	0
[TOTAL˙GC˙TIME]	Time(ms)	0
[TOTAL˙GC˙TIME˙%]	Time(%)	0.0
[READ]	Operations	955
[READ]	AverageLatency(us)	1186.9895287958116
[READ]	MinLatency(us)	349
[READ]	MaxLatency(us)	88255
[READ]	95thPercentileLatency(us)	2527
[READ]	99thPercentileLatency(us)	7899
[READ]	Return=OK	955
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	41552.0
[CLEANUP]	MinLatency(us)	41536
[CLEANUP]	MaxLatency(us)	41567
[CLEANUP]	95thPercentileLatency(us)	41567
[CLEANUP]	99thPercentileLatency(us)	41567
[INSERT]	Operations	45
[INSERT]	AverageLatency(us)	80.26666666666667
[INSERT]	MinLatency(us)	17
[INSERT]	MaxLatency(us)	1568
[INSERT]	95thPercentileLatency(us)	259
[INSERT]	99thPercentileLatency(us)	1568
[INSERT]	Return=OK	45

Table B.8: Complete output from running workload D on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	2873.5632183908046
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	5
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	1.4367816091954022
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	5
[TOTAL˙GC˙TIME˙%]	Time(%)	1.4367816091954022
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	140.0
[CLEANUP]	MinLatency(us)	140
[CLEANUP]	MaxLatency(us)	140
[CLEANUP]	95thPercentileLatency(us)	140
[CLEANUP]	99thPercentileLatency(us)	140
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	229.037
[INSERT]	MinLatency(us)	1
[INSERT]	MaxLatency(us)	57567
[INSERT]	95thPercentileLatency(us)	25
[INSERT]	99thPercentileLatency(us)	2681
[INSERT]	Return=OK	1000

Table B.9: Complete output from loading workload E on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	374.3916136278547
[TOTAL˙GCS˙PS˙Scavenge]	Count	7
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	45
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	1.6847622613253461
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	7
[TOTAL˙GC˙TIME]	Time(ms)	45
[TOTAL˙GC˙TIME˙%]	Time(%)	1.6847622613253461
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	33744.0
[CLEANUP]	MinLatency(us)	33728
[CLEANUP]	MaxLatency(us)	33759
[CLEANUP]	95thPercentileLatency(us)	33759
[CLEANUP]	99thPercentileLatency(us)	33759
[INSERT]	Operations	45
[INSERT]	AverageLatency(us)	35.44444444444444
[INSERT]	MinLatency(us)	12
[INSERT]	MaxLatency(us)	153
[INSERT]	95thPercentileLatency(us)	71
[INSERT]	99thPercentileLatency(us)	153
[INSERT]	Return=OK	45
[SCAN]	Operations	955
[SCAN]	AverageLatency(us)	2460.6376963350785
[SCAN]	MinLatency(us)	408
[SCAN]	MaxLatency(us)	55679
[SCAN]	95thPercentileLatency(us)	7063
[SCAN]	99thPercentileLatency(us)	13591
[SCAN]	Return=OK	955

Table B.10: Complete output from running workload E on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	2590.6735751295337
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	5
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	1.2953367875647668
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	5
[TOTAL˙GC˙TIME˙%]	Time(%)	1.2953367875647668
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	44.0
[CLEANUP]	MinLatency(us)	44
[CLEANUP]	MaxLatency(us)	44
[CLEANUP]	95thPercentileLatency(us)	44
[CLEANUP]	99thPercentileLatency(us)	44
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	262.015
[INSERT]	MinLatency(us)	1
[INSERT]	MaxLatency(us)	62655
[INSERT]	95thPercentileLatency(us)	28
[INSERT]	99thPercentileLatency(us)	7463
[INSERT]	Return=OK	1000

Table B.11: Complete output from loading workload F on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	194.74196689386562
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	5
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.09737098344693282
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	5
[TOTAL˙GC˙TIME˙%]	Time(%)	0.09737098344693282
[READ]	Operations	1000
[READ]	AverageLatency(us)	1178.272
[READ]	MinLatency(us)	420
[READ]	MaxLatency(us)	21007
[READ]	95thPercentileLatency(us)	2679
[READ]	99thPercentileLatency(us)	7931
[READ]	Return=OK	1000
[READ-MODIFY-WRITE]	Operations	494
[READ-MODIFY-WRITE]	AverageLatency(us)	9005.971659919029
[READ-MODIFY-WRITE]	MinLatency(us)	5144
[READ-MODIFY-WRITE]	MaxLatency(us)	106623
[READ-MODIFY-WRITE]	95thPercentileLatency(us)	14615
[READ-MODIFY-WRITE]	99thPercentileLatency(us)	31695
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	70.0
[CLEANUP]	MinLatency(us)	70
[CLEANUP]	MaxLatency(us)	70
[CLEANUP]	95thPercentileLatency(us)	70
[CLEANUP]	99thPercentileLatency(us)	70
[UPDATE]	Operations	494
[UPDATE]	AverageLatency(us)	7660.384615384615
[UPDATE]	MinLatency(us)	4564
[UPDATE]	MaxLatency(us)	100863
[UPDATE]	95thPercentileLatency(us)	12767
[UPDATE]	99thPercentileLatency(us)	21999
[UPDATE]	Return=OK	494

Table B.12: Complete output from running workload F on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	12653.800276321323
[TOTAL˙GCS˙PS˙Scavenge]	Count	622
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	463
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.397319168289983
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	622
[TOTAL˙GC˙TIME]	Time(ms)	463
[TOTAL˙GC˙TIME˙%]	Time(%)	0.397319168289983
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	5862.0
[CLEANUP]	MinLatency(us)	5860
[CLEANUP]	MaxLatency(us)	5863
[CLEANUP]	95thPercentileLatency(us)	5863
[CLEANUP]	99thPercentileLatency(us)	5863
[INSERT]	Operations	1474560
[INSERT]	AverageLatency(us)	78.00698852539062
[INSERT]	MinLatency(us)	0
[INSERT]	MaxLatency(us)	294143
[INSERT]	95thPercentileLatency(us)	0
[INSERT]	99thPercentileLatency(us)	2561
[INSERT]	Return=OK	1474560

Table B.13: Complete output from loading TimeSeries workload A on FoundationDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	8840.47327132164
[TOTAL˙GCS˙PS˙Scavenge]	Count	1270
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	1008
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.3021646137658764
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1270
[TOTAL˙GC˙TIME]	Time(ms)	1008
[TOTAL˙GC˙TIME˙%]	Time(%)	0.3021646137658764
[READ]	Operations	0
[READ]	AverageLatency(us)	NaN
[READ]	MinLatency(us)	9223372036854775807
[READ]	MaxLatency(us)	0
[READ]	95thPercentileLatency(us)	0
[READ]	99thPercentileLatency(us)	0
[READ]	Return=NOT˙FOUND	295486
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	7730.0
[CLEANUP]	MinLatency(us)	7728
[CLEANUP]	MaxLatency(us)	7731
[CLEANUP]	95thPercentileLatency(us)	7731
[CLEANUP]	99thPercentileLatency(us)	7731
[INSERT]	Operations	2653634
[INSERT]	AverageLatency(us)	69.94728248130677
[INSERT]	MinLatency(us)	0
[INSERT]	MaxLatency(us)	136831
[INSERT]	95thPercentileLatency(us)	1
[INSERT]	99thPercentileLatency(us)	4567
[INSERT]	Return=OK	2653634
[READ-FAILED]	Operations	295486
[READ-FAILED]	AverageLatency(us)	487.4802021077141
[READ-FAILED]	MinLatency(us)	288
[READ-FAILED]	MaxLatency(us)	159615
[READ-FAILED]	95thPercentileLatency(us)	812
[READ-FAILED]	99thPercentileLatency(us)	1661

Table B.14: Complete output from running TimeSeries workload A on FoundationDB

B.2 MongoDB benchmark

The following tables contain the output of the YCSB benchmark of MongoDB.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	401.60642570281124
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	12
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.48192771084337355
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	12
[TOTAL˙GC˙TIME˙%]	Time(%)	0.48192771084337355
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	4278.0
[CLEANUP]	MinLatency(us)	4276
[CLEANUP]	MaxLatency(us)	4279
[CLEANUP]	95thPercentileLatency(us)	4279
[CLEANUP]	99thPercentileLatency(us)	4279
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	917.261
[INSERT]	MinLatency(us)	229
[INSERT]	MaxLatency(us)	126591
[INSERT]	95thPercentileLatency(us)	1860
[INSERT]	99thPercentileLatency(us)	5539
[INSERT]	Return=OK	1000

Table B.15: Complete output from loading workload A on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	590.6674542232723
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	10
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.5906674542232723
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	10
[TOTAL˙GC˙TIME˙%]	Time(%)	0.5906674542232723
[READ]	Operations	944
[READ]	AverageLatency(us)	905.9014830508474
[READ]	MinLatency(us)	227
[READ]	MaxLatency(us)	65983
[READ]	95thPercentileLatency(us)	2103
[READ]	99thPercentileLatency(us)	4931
[READ]	Return=OK	944
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2709.0
[CLEANUP]	MinLatency(us)	2708
[CLEANUP]	MaxLatency(us)	2709
[CLEANUP]	95thPercentileLatency(us)	2709
[CLEANUP]	99thPercentileLatency(us)	2709
[UPDATE]	Operations	56
[UPDATE]	AverageLatency(us)	1121.8214285714287
[UPDATE]	MinLatency(us)	422
[UPDATE]	MaxLatency(us)	10447
[UPDATE]	95thPercentileLatency(us)	3029
[UPDATE]	99thPercentileLatency(us)	4923
[UPDATE]	Return=OK	56

Table B.16: Complete output from running workload A on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	406.00893219650834
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	9
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.3654080389768575
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	9
[TOTAL˙GC˙TIME˙%]	Time(%)	0.3654080389768575
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2961.0
[CLEANUP]	MinLatency(us)	2960
[CLEANUP]	MaxLatency(us)	2961
[CLEANUP]	95thPercentileLatency(us)	2961
[CLEANUP]	99thPercentileLatency(us)	2961
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	1127.523
[INSERT]	MinLatency(us)	250
[INSERT]	MaxLatency(us)	92159
[INSERT]	95thPercentileLatency(us)	3261
[INSERT]	99thPercentileLatency(us)	10455
[INSERT]	Return=OK	1000

Table B.17: Complete output from loading workload B on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	590.6674542232723
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	10
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.5906674542232723
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	10
[TOTAL˙GC˙TIME˙%]	Time(%)	0.5906674542232723
[READ]	Operations	944
[READ]	AverageLatency(us)	905.9014830508474
[READ]	MinLatency(us)	227
[READ]	MaxLatency(us)	65983
[READ]	95thPercentileLatency(us)	2103
[READ]	99thPercentileLatency(us)	4931
[READ]	Return=OK	944
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2709.0
[CLEANUP]	MinLatency(us)	2708
[CLEANUP]	MaxLatency(us)	2709
[CLEANUP]	95thPercentileLatency(us)	2709
[CLEANUP]	99thPercentileLatency(us)	2709
[UPDATE]	Operations	56
[UPDATE]	AverageLatency(us)	1121.8214285714287
[UPDATE]	MinLatency(us)	422
[UPDATE]	MaxLatency(us)	10447
[UPDATE]	95thPercentileLatency(us)	3029
[UPDATE]	99thPercentileLatency(us)	4923
[UPDATE]	Return=OK	56

Table B.18: Complete output from running workload B on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	729.3946024799417
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	7
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.5105762217359592
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	7
[TOTAL˙GC˙TIME˙%]	Time(%)	0.5105762217359592
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	3113.0
[CLEANUP]	MinLatency(us)	3112
[CLEANUP]	MaxLatency(us)	3113
[CLEANUP]	95thPercentileLatency(us)	3113
[CLEANUP]	99thPercentileLatency(us)	3113
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	687.324
[INSERT]	MinLatency(us)	300
[INSERT]	MaxLatency(us)	57535
[INSERT]	95thPercentileLatency(us)	1142
[INSERT]	99thPercentileLatency(us)	3149
[INSERT]	Return=OK	1000

Table B.19: Complete output from loading workload C on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	761.6146230007616
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	9
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.6854531607006854
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	9
[TOTAL˙GC˙TIME˙%]	Time(%)	0.6854531607006854
[READ]	Operations	1000
[READ]	AverageLatency(us)	678.283
[READ]	MinLatency(us)	288
[READ]	MaxLatency(us)	57439
[READ]	95thPercentileLatency(us)	1185
[READ]	99thPercentileLatency(us)	3419
[READ]	Return=OK	1000
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2077.0
[CLEANUP]	MinLatency(us)	2076
[CLEANUP]	MaxLatency(us)	2077
[CLEANUP]	95thPercentileLatency(us)	2077
[CLEANUP]	99thPercentileLatency(us)	2077

Table B.20: Complete output from running workload C on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	733.6757153338225
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.586940572267058
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.586940572267058
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2971.0
[CLEANUP]	MinLatency(us)	2970
[CLEANUP]	MaxLatency(us)	2971
[CLEANUP]	95thPercentileLatency(us)	2971
[CLEANUP]	99thPercentileLatency(us)	2971
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	718.422
[INSERT]	MinLatency(us)	307
[INSERT]	MaxLatency(us)	40895
[INSERT]	95thPercentileLatency(us)	1336
[INSERT]	99thPercentileLatency(us)	3255
[INSERT]	Return=OK	1000

Table B.21: Complete output from loading workload D on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	770.4160246533128
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.6163328197226503
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.6163328197226503
[READ]	Operations	954
[READ]	AverageLatency(us)	643.3427672955975
[READ]	MinLatency(us)	210
[READ]	MaxLatency(us)	57375
[READ]	95thPercentileLatency(us)	1159
[READ]	99thPercentileLatency(us)	2321
[READ]	Return=OK	954
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2449.0
[CLEANUP]	MinLatency(us)	2448
[CLEANUP]	MaxLatency(us)	2449
[CLEANUP]	95thPercentileLatency(us)	2449
[CLEANUP]	99thPercentileLatency(us)	2449
[INSERT]	Operations	46
[INSERT]	AverageLatency(us)	855.2608695652174
[INSERT]	MinLatency(us)	389
[INSERT]	MaxLatency(us)	7331
[INSERT]	95thPercentileLatency(us)	1621
[INSERT]	99thPercentileLatency(us)	7331
[INSERT]	Return=OK	46

Table B.22: Complete output from running workload D on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	766.2835249042146
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	9
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.6896551724137931
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	9
[TOTAL˙GC˙TIME˙%]	Time(%)	0.6896551724137931
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	3079.0
[CLEANUP]	MinLatency(us)	3078
[CLEANUP]	MaxLatency(us)	3079
[CLEANUP]	95thPercentileLatency(us)	3079
[CLEANUP]	99thPercentileLatency(us)	3079
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	696.502
[INSERT]	MinLatency(us)	292
[INSERT]	MaxLatency(us)	58399
[INSERT]	95thPercentileLatency(us)	1210
[INSERT]	99thPercentileLatency(us)	2981
[INSERT]	Return=OK	1000

Table B.23: Complete output from loading workload E on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	449.23629829290206
[TOTAL˙GCS˙PS˙Scavenge]	Count	5
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	63
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	2.8301886792452833
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	5
[TOTAL˙GC˙TIME]	Time(ms)	63
[TOTAL˙GC˙TIME˙%]	Time(%)	2.8301886792452833
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2909.0
[CLEANUP]	MinLatency(us)	2908
[CLEANUP]	MaxLatency(us)	2909
[CLEANUP]	95thPercentileLatency(us)	2909
[CLEANUP]	99thPercentileLatency(us)	2909
[INSERT]	Operations	51
[INSERT]	AverageLatency(us)	1162.5098039215686
[INSERT]	MinLatency(us)	453
[INSERT]	MaxLatency(us)	11111
[INSERT]	95thPercentileLatency(us)	1575
[INSERT]	99thPercentileLatency(us)	6635
[INSERT]	Return=OK	51
[SCAN]	Operations	949
[SCAN]	AverageLatency(us)	1618.6733403582718
[SCAN]	MinLatency(us)	328
[SCAN]	MaxLatency(us)	109951
[SCAN]	95thPercentileLatency(us)	3329
[SCAN]	99thPercentileLatency(us)	9183
[SCAN]	Return=OK	949

Table B.24: Complete output from running workload E on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	651.8904823989569
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	7
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.45632333767926986
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	7
[TOTAL˙GC˙TIME˙%]	Time(%)	0.45632333767926986
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2851.0
[CLEANUP]	MinLatency(us)	2850
[CLEANUP]	MaxLatency(us)	2851
[CLEANUP]	95thPercentileLatency(us)	2851
[CLEANUP]	99thPercentileLatency(us)	2851
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	894.645
[INSERT]	MinLatency(us)	287
[INSERT]	MaxLatency(us)	58047
[INSERT]	95thPercentileLatency(us)	2063
[INSERT]	99thPercentileLatency(us)	4247
[INSERT]	Return=OK	1000

Table B.25: Complete output from loading workload F on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	587.5440658049354
[TOTAL˙GCS˙PS˙Scavenge]	Count	2
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	16
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.9400705052878966
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	2
[TOTAL˙GC˙TIME]	Time(ms)	16
[TOTAL˙GC˙TIME˙%]	Time(%)	0.9400705052878966
[READ]	Operations	1000
[READ]	AverageLatency(us)	728.47
[READ]	MinLatency(us)	188
[READ]	MaxLatency(us)	56447
[READ]	95thPercentileLatency(us)	1516
[READ]	99thPercentileLatency(us)	3357
[READ]	Return=OK	1000
[READ-MODIFY-WRITE]	Operations	501
[READ-MODIFY-WRITE]	AverageLatency(us)	1244.1017964071857
[READ-MODIFY-WRITE]	MinLatency(us)	487
[READ-MODIFY-WRITE]	MaxLatency(us)	11143
[READ-MODIFY-WRITE]	95thPercentileLatency(us)	2829
[READ-MODIFY-WRITE]	99thPercentileLatency(us)	6715
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	5818.0
[CLEANUP]	MinLatency(us)	5816
[CLEANUP]	MaxLatency(us)	5819
[CLEANUP]	95thPercentileLatency(us)	5819
[CLEANUP]	99thPercentileLatency(us)	5819
[UPDATE]	Operations	501
[UPDATE]	AverageLatency(us)	636.0698602794412
[UPDATE]	MinLatency(us)	265
[UPDATE]	MaxLatency(us)	8743
[UPDATE]	95thPercentileLatency(us)	1190
[UPDATE]	99thPercentileLatency(us)	4215
[UPDATE]	Return=OK	501

Table B.26: Complete output from running workload F on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	6187.674670381778
[TOTAL˙GCS˙PS˙Scavenge]	Count	705
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	705
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.2958381240925533
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	705
[TOTAL˙GC˙TIME]	Time(ms)	705
[TOTAL˙GC˙TIME˙%]	Time(%)	0.2958381240925533
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	7594.0
[CLEANUP]	MinLatency(us)	7592
[CLEANUP]	MaxLatency(us)	7595
[CLEANUP]	95thPercentileLatency(us)	7595
[CLEANUP]	99thPercentileLatency(us)	7595
[INSERT]	Operations	1474560
[INSERT]	AverageLatency(us)	159.312894015842
[INSERT]	MinLatency(us)	113
[INSERT]	MaxLatency(us)	1719295
[INSERT]	95thPercentileLatency(us)	234
[INSERT]	99thPercentileLatency(us)	518
[INSERT]	Return=OK	1474560

Table B.27: Complete output from loading TimeSeries workload A on MongoDB

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	6428.935791736243
[TOTAL˙GCS˙PS˙Scavenge]	Count	1509
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	1377
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.3001791919359269
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1509
[TOTAL˙GC˙TIME]	Time(ms)	1377
[TOTAL˙GC˙TIME˙%]	Time(%)	0.3001791919359269
[READ]	Operations	294971
[READ]	AverageLatency(us)	159.4121422105902
[READ]	MinLatency(us)	110
[READ]	MaxLatency(us)	78015
[READ]	95thPercentileLatency(us)	237
[READ]	99thPercentileLatency(us)	476
[READ]	Return=OK	294971
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	4638.0
[CLEANUP]	MinLatency(us)	4636
[CLEANUP]	MaxLatency(us)	4639
[CLEANUP]	95thPercentileLatency(us)	4639
[CLEANUP]	99thPercentileLatency(us)	4639
[INSERT]	Operations	2654149
[INSERT]	AverageLatency(us)	152.62485715760494
[INSERT]	MinLatency(us)	113
[INSERT]	MaxLatency(us)	161663
[INSERT]	95thPercentileLatency(us)	225
[INSERT]	99thPercentileLatency(us)	453
[INSERT]	Return=OK	2654149

Table B.28: Complete output from running TimeSeries workload A on MongoDB

B.3 MongoDB benchmark with stricter write concern

This section contains the output of YCSB benchmarks of the official MongoDB driver, with a write concern more similar to that of FoundationDB.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	175.65431231336729
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	12
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.21078517477604075
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	12
[TOTAL˙GC˙TIME˙%]	Time(%)	0.21078517477604075
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	5350.0
[CLEANUP]	MinLatency(us)	5348
[CLEANUP]	MaxLatency(us)	5351
[CLEANUP]	95thPercentileLatency(us)	5351
[CLEANUP]	99thPercentileLatency(us)	5351
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	5068.726
[INSERT]	MinLatency(us)	3096
[INSERT]	MaxLatency(us)	60959
[INSERT]	95thPercentileLatency(us)	6747
[INSERT]	99thPercentileLatency(us)	10143
[INSERT]	Return=OK	1000

Table B.29: Complete output from loading workload A on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	480.07681228996637
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.3840614498319731
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.3840614498319731
[READ]	Operations	952
[READ]	AverageLatency(us)	1039.6334033613446
[READ]	MinLatency(us)	239
[READ]	MaxLatency(us)	79551
[READ]	95thPercentileLatency(us)	2603
[READ]	99thPercentileLatency(us)	6387
[READ]	Return=OK	952
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	3095.0
[CLEANUP]	MinLatency(us)	3094
[CLEANUP]	MaxLatency(us)	3095
[CLEANUP]	95thPercentileLatency(us)	3095
[CLEANUP]	99thPercentileLatency(us)	3095
[UPDATE]	Operations	48
[UPDATE]	AverageLatency(us)	5551.791666666667
[UPDATE]	MinLatency(us)	3370
[UPDATE]	MaxLatency(us)	27359
[UPDATE]	95thPercentileLatency(us)	9303
[UPDATE]	99thPercentileLatency(us)	27359
[UPDATE]	Return=OK	48

Table B.30: Complete output from running workload A on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	153.30369461904033
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.12264295569523226
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.12264295569523226
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2767.0
[CLEANUP]	MinLatency(us)	2766
[CLEANUP]	MaxLatency(us)	2767
[CLEANUP]	95thPercentileLatency(us)	2767
[CLEANUP]	99thPercentileLatency(us)	2767
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	5335.939
[INSERT]	MinLatency(us)	3208
[INSERT]	MaxLatency(us)	62911
[INSERT]	95thPercentileLatency(us)	9383
[INSERT]	99thPercentileLatency(us)	16047
[INSERT]	Return=OK	1000

Table B.31: Complete output from loading workload B on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	480.07681228996637
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.3840614498319731
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.3840614498319731
[READ]	Operations	952
[READ]	AverageLatency(us)	1039.6334033613446
[READ]	MinLatency(us)	239
[READ]	MaxLatency(us)	79551
[READ]	95thPercentileLatency(us)	2603
[READ]	99thPercentileLatency(us)	6387
[READ]	Return=OK	952
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	3095.0
[CLEANUP]	MinLatency(us)	3094
[CLEANUP]	MaxLatency(us)	3095
[CLEANUP]	95thPercentileLatency(us)	3095
[CLEANUP]	99thPercentileLatency(us)	3095
[UPDATE]	Operations	48
[UPDATE]	AverageLatency(us)	5551.791666666667
[UPDATE]	MinLatency(us)	3370
[UPDATE]	MaxLatency(us)	27359
[UPDATE]	95thPercentileLatency(us)	9303
[UPDATE]	99thPercentileLatency(us)	27359
[UPDATE]	Return=OK	48

Table B.32: Complete output from running workload B on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	159.4896331738437
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	10
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.1594896331738437
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	10
[TOTAL˙GC˙TIME˙%]	Time(%)	0.1594896331738437
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2693.0
[CLEANUP]	MinLatency(us)	2692
[CLEANUP]	MaxLatency(us)	2693
[CLEANUP]	95thPercentileLatency(us)	2693
[CLEANUP]	99thPercentileLatency(us)	2693
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	5075.049
[INSERT]	MinLatency(us)	3278
[INSERT]	MaxLatency(us)	76159
[INSERT]	95thPercentileLatency(us)	7415
[INSERT]	99thPercentileLatency(us)	10735
[INSERT]	Return=OK	1000

Table B.33: Complete output from loading workload C on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	800.0
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.64
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.64
[READ]	Operations	1000
[READ]	AverageLatency(us)	714.417
[READ]	MinLatency(us)	221
[READ]	MaxLatency(us)	66111
[READ]	95thPercentileLatency(us)	1321
[READ]	99thPercentileLatency(us)	2957
[READ]	Return=OK	1000
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2595.0
[CLEANUP]	MinLatency(us)	2594
[CLEANUP]	MaxLatency(us)	2595
[CLEANUP]	95thPercentileLatency(us)	2595
[CLEANUP]	99thPercentileLatency(us)	2595

Table B.34: Complete output from running workload C on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	173.19016279875302
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	11
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.19050917907862833
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	11
[TOTAL˙GC˙TIME˙%]	Time(%)	0.19050917907862833
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	3495.0
[CLEANUP]	MinLatency(us)	3494
[CLEANUP]	MaxLatency(us)	3495
[CLEANUP]	95thPercentileLatency(us)	3495
[CLEANUP]	99thPercentileLatency(us)	3495
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	5028.363
[INSERT]	MinLatency(us)	3350
[INSERT]	MaxLatency(us)	66111
[INSERT]	95thPercentileLatency(us)	6267
[INSERT]	99thPercentileLatency(us)	9903
[INSERT]	Return=OK	1000

Table B.35: Complete output from loading workload D on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	766.8711656441718
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.6134969325153374
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.6134969325153374
[READ]	Operations	953
[READ]	AverageLatency(us)	572.392444910808
[READ]	MinLatency(us)	186
[READ]	MaxLatency(us)	69759
[READ]	95thPercentileLatency(us)	1066
[READ]	99thPercentileLatency(us)	2189
[READ]	Return=OK	953
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	3123.0
[CLEANUP]	MinLatency(us)	3122
[CLEANUP]	MaxLatency(us)	3123
[CLEANUP]	95thPercentileLatency(us)	3123
[CLEANUP]	99thPercentileLatency(us)	3123
[INSERT]	Operations	47
[INSERT]	AverageLatency(us)	4792.595744680851
[INSERT]	MinLatency(us)	3318
[INSERT]	MaxLatency(us)	15543
[INSERT]	95thPercentileLatency(us)	5839
[INSERT]	99thPercentileLatency(us)	15543
[INSERT]	Return=OK	47

Table B.36: Complete output from running workload D on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	182.2821728034998
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	7
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.12759752096244986
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	7
[TOTAL˙GC˙TIME˙%]	Time(%)	0.12759752096244986
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	3179.0
[CLEANUP]	MinLatency(us)	3178
[CLEANUP]	MaxLatency(us)	3179
[CLEANUP]	95thPercentileLatency(us)	3179
[CLEANUP]	99thPercentileLatency(us)	3179
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	4918.853
[INSERT]	MinLatency(us)	3244
[INSERT]	MaxLatency(us)	57855
[INSERT]	95thPercentileLatency(us)	6167
[INSERT]	99thPercentileLatency(us)	9015
[INSERT]	Return=OK	1000

Table B.37: Complete output from loading workload E on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	564.9717514124294
[TOTAL˙GCS˙PS˙Scavenge]	Count	5
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	36
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	2.0338983050847457
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	5
[TOTAL˙GC˙TIME]	Time(ms)	36
[TOTAL˙GC˙TIME˙%]	Time(%)	2.0338983050847457
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	3645.0
[CLEANUP]	MinLatency(us)	3644
[CLEANUP]	MaxLatency(us)	3645
[CLEANUP]	95thPercentileLatency(us)	3645
[CLEANUP]	99thPercentileLatency(us)	3645
[INSERT]	Operations	40
[INSERT]	AverageLatency(us)	5555.275
[INSERT]	MinLatency(us)	3564
[INSERT]	MaxLatency(us)	12111
[INSERT]	95thPercentileLatency(us)	9863
[INSERT]	99thPercentileLatency(us)	12111
[INSERT]	Return=OK	40
[SCAN]	Operations	960
[SCAN]	AverageLatency(us)	1034.709375
[SCAN]	MinLatency(us)	216
[SCAN]	MaxLatency(us)	67967
[SCAN]	95thPercentileLatency(us)	1820
[SCAN]	99thPercentileLatency(us)	4607
[SCAN]	Return=OK	960

Table B.38: Complete output from running workload E on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	166.44474034620507
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	7
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.11651131824234355
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	7
[TOTAL˙GC˙TIME˙%]	Time(%)	0.11651131824234355
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	2637.0
[CLEANUP]	MinLatency(us)	2636
[CLEANUP]	MaxLatency(us)	2637
[CLEANUP]	95thPercentileLatency(us)	2637
[CLEANUP]	99thPercentileLatency(us)	2637
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	5430.848
[INSERT]	MinLatency(us)	3278
[INSERT]	MaxLatency(us)	65439
[INSERT]	95thPercentileLatency(us)	7127
[INSERT]	99thPercentileLatency(us)	12415
[INSERT]	Return=OK	1000

Table B.39: Complete output from loading workload F on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	291.5451895043732
[TOTAL˙GCS˙PS˙Scavenge]	Count	2
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	15
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.43731778425655976
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	2
[TOTAL˙GC˙TIME]	Time(ms)	15
[TOTAL˙GC˙TIME˙%]	Time(%)	0.43731778425655976
[READ]	Operations	1000
[READ]	AverageLatency(us)	665.283
[READ]	MinLatency(us)	165
[READ]	MaxLatency(us)	58879
[READ]	95thPercentileLatency(us)	1116
[READ]	99thPercentileLatency(us)	1966
[READ]	Return=OK	1000
[READ-MODIFY-WRITE]	Operations	483
[READ-MODIFY-WRITE]	AverageLatency(us)	5345.946169772256
[READ-MODIFY-WRITE]	MinLatency(us)	3380
[READ-MODIFY-WRITE]	MaxLatency(us)	88127
[READ-MODIFY-WRITE]	95thPercentileLatency(us)	7023
[READ-MODIFY-WRITE]	99thPercentileLatency(us)	10007
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	3639.0
[CLEANUP]	MinLatency(us)	3638
[CLEANUP]	MaxLatency(us)	3639
[CLEANUP]	95thPercentileLatency(us)	3639
[CLEANUP]	99thPercentileLatency(us)	3639
[UPDATE]	Operations	483
[UPDATE]	AverageLatency(us)	4574.304347826087
[UPDATE]	MinLatency(us)	3206
[UPDATE]	MaxLatency(us)	19151
[UPDATE]	95thPercentileLatency(us)	5839
[UPDATE]	99thPercentileLatency(us)	9015
[UPDATE]	Return=OK	483

Table B.40: Complete output from running workload F on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	208.9941645590934
[TOTAL˙GCS˙PS˙Scavenge]	Count	729
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	1280
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.018141854562421303
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	729
[TOTAL˙GC˙TIME]	Time(ms)	1280
[TOTAL˙GC˙TIME˙%]	Time(%)	0.018141854562421303
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	9844.0
[CLEANUP]	MinLatency(us)	9840
[CLEANUP]	MaxLatency(us)	9847
[CLEANUP]	95thPercentileLatency(us)	9847
[CLEANUP]	99thPercentileLatency(us)	9847
[INSERT]	Operations	1474560
[INSERT]	AverageLatency(us)	4779.209121365017
[INSERT]	MinLatency(us)	2744
[INSERT]	MaxLatency(us)	388863
[INSERT]	95thPercentileLatency(us)	6151
[INSERT]	99thPercentileLatency(us)	9743
[INSERT]	Return=OK	1474560

Table B.41: Complete output from loading TimeSeries workload A on MongoDB with a stricter write concern

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	241.69033753317063
[TOTAL˙GCS˙PS˙Scavenge]	Count	1566
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	2771
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.022709280236287974
[TOTAL˙GCS˙PS˙MarkSweep]	Count	4
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	418
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0034256510785883694
[TOTAL˙GCs]	Count	1570
[TOTAL˙GC˙TIME]	Time(ms)	3189
[TOTAL˙GC˙TIME˙%]	Time(%)	0.02613493131487634
[READ]	Operations	294205
[READ]	AverageLatency(us)	429.07787767033193
[READ]	MinLatency(us)	123
[READ]	MaxLatency(us)	24575
[READ]	95thPercentileLatency(us)	699
[READ]	99thPercentileLatency(us)	1320
[READ]	Return=OK	294205
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	90656.0
[CLEANUP]	MinLatency(us)	90624
[CLEANUP]	MaxLatency(us)	90687
[CLEANUP]	95thPercentileLatency(us)	90687
[CLEANUP]	99thPercentileLatency(us)	90687
[INSERT]	Operations	2654915
[INSERT]	AverageLatency(us)	4541.985827041543
[INSERT]	MinLatency(us)	2692
[INSERT]	MaxLatency(us)	711679
[INSERT]	95thPercentileLatency(us)	6075
[INSERT]	99thPercentileLatency(us)	9911
[INSERT]	Return=OK	2654915

Table B.42: Complete output from running TimeSeries workload A on MongoDB with a stricter write concern

B.4 FoundationDB Document Layer benchmark via MongoDB

This section contains the output of YCSB benchmarks of the FoundationDB Document Layer, using YCSB MongoDB binding.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	75.08071176514753
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.060064569412118025
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.060064569412118025
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1929.0
[CLEANUP]	MinLatency(us)	1929
[CLEANUP]	MaxLatency(us)	1929
[CLEANUP]	95thPercentileLatency(us)	1929
[CLEANUP]	99thPercentileLatency(us)	1929
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	12266.634
[INSERT]	MinLatency(us)	5816
[INSERT]	MaxLatency(us)	174207
[INSERT]	95thPercentileLatency(us)	31743
[INSERT]	99thPercentileLatency(us)	63327
[INSERT]	Return=OK	1000

Table B.43: Complete output from loading workload A on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	400.64102564102564
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.3205128205128205
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.3205128205128205
[READ]	Operations	951
[READ]	AverageLatency(us)	1692.6056782334385
[READ]	MinLatency(us)	875
[READ]	MaxLatency(us)	44639
[READ]	95thPercentileLatency(us)	3067
[READ]	99thPercentileLatency(us)	4395
[READ]	Return=OK	951
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1823.0
[CLEANUP]	MinLatency(us)	1823
[CLEANUP]	MaxLatency(us)	1823
[CLEANUP]	95thPercentileLatency(us)	1823
[CLEANUP]	99thPercentileLatency(us)	1823
[UPDATE]	Operations	49
[UPDATE]	AverageLatency(us)	7780.571428571428
[UPDATE]	MinLatency(us)	5580
[UPDATE]	MaxLatency(us)	17615
[UPDATE]	95thPercentileLatency(us)	11439
[UPDATE]	99thPercentileLatency(us)	17615
[UPDATE]	Return=OK	49

Table B.44: Complete output from running workload A on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	110.60723371308484
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	10
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.11060723371308484
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	10
[TOTAL˙GC˙TIME˙%]	Time(%)	0.11060723371308484
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1832.0
[CLEANUP]	MinLatency(us)	1832
[CLEANUP]	MaxLatency(us)	1832
[CLEANUP]	95thPercentileLatency(us)	1832
[CLEANUP]	99thPercentileLatency(us)	1832
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	8524.606
[INSERT]	MinLatency(us)	5628
[INSERT]	MaxLatency(us)	64927
[INSERT]	95thPercentileLatency(us)	11127
[INSERT]	99thPercentileLatency(us)	14695
[INSERT]	Return=OK	1000

Table B.45: Complete output from loading workload B on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	400.64102564102564
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.3205128205128205
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.3205128205128205
[READ]	Operations	951
[READ]	AverageLatency(us)	1692.6056782334385
[READ]	MinLatency(us)	875
[READ]	MaxLatency(us)	44639
[READ]	95thPercentileLatency(us)	3067
[READ]	99thPercentileLatency(us)	4395
[READ]	Return=OK	951
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1823.0
[CLEANUP]	MinLatency(us)	1823
[CLEANUP]	MaxLatency(us)	1823
[CLEANUP]	95thPercentileLatency(us)	1823
[CLEANUP]	99thPercentileLatency(us)	1823
[UPDATE]	Operations	49
[UPDATE]	AverageLatency(us)	7780.571428571428
[UPDATE]	MinLatency(us)	5580
[UPDATE]	MaxLatency(us)	17615
[UPDATE]	95thPercentileLatency(us)	11439
[UPDATE]	99thPercentileLatency(us)	17615
[UPDATE]	Return=OK	49

Table B.46: Complete output from running workload B on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	112.00716845878136
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	7
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.07840501792114696
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	7
[TOTAL˙GC˙TIME˙%]	Time(%)	0.07840501792114696
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1891.0
[CLEANUP]	MinLatency(us)	1891
[CLEANUP]	MaxLatency(us)	1891
[CLEANUP]	95thPercentileLatency(us)	1891
[CLEANUP]	99thPercentileLatency(us)	1891
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	8394.18
[INSERT]	MinLatency(us)	5664
[INSERT]	MaxLatency(us)	49023
[INSERT]	95thPercentileLatency(us)	11231
[INSERT]	99thPercentileLatency(us)	14423
[INSERT]	Return=OK	1000

Table B.47: Complete output from loading workload C on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	458.9261128958238
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	9
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.4130335016062414
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	9
[TOTAL˙GC˙TIME˙%]	Time(%)	0.4130335016062414
[READ]	Operations	1000
[READ]	AverageLatency(us)	1667.635
[READ]	MinLatency(us)	965
[READ]	MaxLatency(us)	44255
[READ]	95thPercentileLatency(us)	3139
[READ]	99thPercentileLatency(us)	5015
[READ]	Return=OK	1000
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1656.0
[CLEANUP]	MinLatency(us)	1656
[CLEANUP]	MaxLatency(us)	1656
[CLEANUP]	95thPercentileLatency(us)	1656
[CLEANUP]	99thPercentileLatency(us)	1656

Table B.48: Complete output from running workload C on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	111.07408641563923
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	7
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.07775186049094747
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	7
[TOTAL˙GC˙TIME˙%]	Time(%)	0.07775186049094747
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1935.0
[CLEANUP]	MinLatency(us)	1935
[CLEANUP]	MaxLatency(us)	1935
[CLEANUP]	95thPercentileLatency(us)	1935
[CLEANUP]	99thPercentileLatency(us)	1935
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	8455.8
[INSERT]	MinLatency(us)	5700
[INSERT]	MaxLatency(us)	58847
[INSERT]	95thPercentileLatency(us)	11023
[INSERT]	99thPercentileLatency(us)	15039
[INSERT]	Return=OK	1000

Table B.49: Complete output from loading workload D on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	379.9392097264438
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.303951367781155
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.303951367781155
[READ]	Operations	946
[READ]	AverageLatency(us)	1781.1289640591965
[READ]	MinLatency(us)	960
[READ]	MaxLatency(us)	53247
[READ]	95thPercentileLatency(us)	3147
[READ]	99thPercentileLatency(us)	4851
[READ]	Return=OK	946
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1734.0
[CLEANUP]	MinLatency(us)	1734
[CLEANUP]	MaxLatency(us)	1734
[CLEANUP]	95thPercentileLatency(us)	1734
[CLEANUP]	99thPercentileLatency(us)	1734
[INSERT]	Operations	54
[INSERT]	AverageLatency(us)	8165.185185185185
[INSERT]	MinLatency(us)	5592
[INSERT]	MaxLatency(us)	17295
[INSERT]	95thPercentileLatency(us)	10271
[INSERT]	99thPercentileLatency(us)	12863
[INSERT]	Return=OK	54

Table B.50: Complete output from running workload D on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	110.37527593818984
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	8
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.08830022075055188
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	8
[TOTAL˙GC˙TIME˙%]	Time(%)	0.08830022075055188
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1961.0
[CLEANUP]	MinLatency(us)	1961
[CLEANUP]	MaxLatency(us)	1961
[CLEANUP]	95thPercentileLatency(us)	1961
[CLEANUP]	99thPercentileLatency(us)	1961
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	8535.326
[INSERT]	MinLatency(us)	5508
[INSERT]	MaxLatency(us)	52959
[INSERT]	95thPercentileLatency(us)	11399
[INSERT]	99thPercentileLatency(us)	15423
[INSERT]	Return=OK	1000

Table B.51: Complete output from loading workload E on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	11.108395725489325
[TOTAL˙GCS˙PS˙Scavenge]	Count	7
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	70
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.07775877007842527
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	7
[TOTAL˙GC˙TIME]	Time(ms)	70
[TOTAL˙GC˙TIME˙%]	Time(%)	0.07775877007842527
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1711.0
[CLEANUP]	MinLatency(us)	1711
[CLEANUP]	MaxLatency(us)	1711
[CLEANUP]	95thPercentileLatency(us)	1711
[CLEANUP]	99thPercentileLatency(us)	1711
[INSERT]	Operations	56
[INSERT]	AverageLatency(us)	10748.82142857143
[INSERT]	MinLatency(us)	6128
[INSERT]	MaxLatency(us)	77119
[INSERT]	95thPercentileLatency(us)	11423
[INSERT]	99thPercentileLatency(us)	25279
[INSERT]	Return=OK	56
[SCAN]	Operations	944
[SCAN]	AverageLatency(us)	94163.71080508475
[SCAN]	MinLatency(us)	1847
[SCAN]	MaxLatency(us)	1187839
[SCAN]	95thPercentileLatency(us)	165759
[SCAN]	99thPercentileLatency(us)	176511
[SCAN]	Return=OK	944

Table B.52: Complete output from running workload E on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	112.00716845878136
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	7
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.07840501792114696
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	7
[TOTAL˙GC˙TIME˙%]	Time(%)	0.07840501792114696
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1772.0
[CLEANUP]	MinLatency(us)	1772
[CLEANUP]	MaxLatency(us)	1772
[CLEANUP]	95thPercentileLatency(us)	1772
[CLEANUP]	99thPercentileLatency(us)	1772
[INSERT]	Operations	1000
[INSERT]	AverageLatency(us)	8409.85
[INSERT]	MinLatency(us)	5716
[INSERT]	MaxLatency(us)	50559
[INSERT]	95thPercentileLatency(us)	11071
[INSERT]	99thPercentileLatency(us)	14455
[INSERT]	Return=OK	1000

Table B.53: Complete output from loading workload F on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	144.52955629426216
[TOTAL˙GCS˙PS˙Scavenge]	Count	1
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	12
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.1734354675531146
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1
[TOTAL˙GC˙TIME]	Time(ms)	12
[TOTAL˙GC˙TIME˙%]	Time(%)	0.1734354675531146
[READ]	Operations	1000
[READ]	AverageLatency(us)	2133.857
[READ]	MinLatency(us)	954
[READ]	MaxLatency(us)	58303
[READ]	95thPercentileLatency(us)	3743
[READ]	99thPercentileLatency(us)	8647
[READ]	Return=OK	1000
[READ-MODIFY-WRITE]	Operations	508
[READ-MODIFY-WRITE]	AverageLatency(us)	10411.311023622047
[READ-MODIFY-WRITE]	MinLatency(us)	6416
[READ-MODIFY-WRITE]	MaxLatency(us)	81343
[READ-MODIFY-WRITE]	95thPercentileLatency(us)	18575
[READ-MODIFY-WRITE]	99thPercentileLatency(us)	28063
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	1946.0
[CLEANUP]	MinLatency(us)	1946
[CLEANUP]	MaxLatency(us)	1946
[CLEANUP]	95thPercentileLatency(us)	1946
[CLEANUP]	99thPercentileLatency(us)	1946
[UPDATE]	Operations	508
[UPDATE]	AverageLatency(us)	8390.944881889764
[UPDATE]	MinLatency(us)	5048
[UPDATE]	MaxLatency(us)	78399
[UPDATE]	95thPercentileLatency(us)	13935
[UPDATE]	99thPercentileLatency(us)	25327
[UPDATE]	Return=OK	508

Table B.54: Complete output from running workload F on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	112.76842208867987
[TOTAL˙GCS˙PS˙Scavenge]	Count	666
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	744
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.005689812963458782
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	666
[TOTAL˙GC˙TIME]	Time(ms)	744
[TOTAL˙GC˙TIME˙%]	Time(%)	0.005689812963458782
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	74400.0
[CLEANUP]	MinLatency(us)	74368
[CLEANUP]	MaxLatency(us)	74431
[CLEANUP]	95thPercentileLatency(us)	74431
[CLEANUP]	99thPercentileLatency(us)	74431
[INSERT]	Operations	1474560
[INSERT]	AverageLatency(us)	8861.910639105903
[INSERT]	MinLatency(us)	4432
[INSERT]	MaxLatency(us)	1352703
[INSERT]	95thPercentileLatency(us)	12727
[INSERT]	99thPercentileLatency(us)	19599
[INSERT]	Return=OK	1474560

Table B.55: Complete output from loading TimeSeries workload A on the FoundationDB Document Layer.

Operation	Metric	Output value
[OVERALL]	Throughput(ops/sec)	108.23056466054364
[TOTAL˙GCS˙PS˙Scavenge]	Count	1311
[TOTAL˙GC˙TIME˙PS˙Scavenge]	Time(ms)	2311
[TOTAL˙GC˙TIME˙%˙PS˙Scavenge]	Time(%)	0.008481202356313625
[TOTAL˙GCS˙PS˙MarkSweep]	Count	0
[TOTAL˙GC˙TIME˙PS˙MarkSweep]	Time(ms)	0
[TOTAL˙GC˙TIME˙%˙PS˙MarkSweep]	Time(%)	0.0
[TOTAL˙GCs]	Count	1311
[TOTAL˙GC˙TIME]	Time(ms)	2311
[TOTAL˙GC˙TIME˙%]	Time(%)	0.008481202356313625
[READ]	Operations	294668
[READ]	AverageLatency(us)	2848.609367152185
[READ]	MinLatency(us)	854
[READ]	MaxLatency(us)	625151
[READ]	95thPercentileLatency(us)	5083
[READ]	99thPercentileLatency(us)	8479
[READ]	Return=OK	294668
[CLEANUP]	Operations	1
[CLEANUP]	AverageLatency(us)	264064.0
[CLEANUP]	MinLatency(us)	263936
[CLEANUP]	MaxLatency(us)	264191
[CLEANUP]	95thPercentileLatency(us)	264191
[CLEANUP]	99thPercentileLatency(us)	264191
[INSERT]	Operations	2654452
[INSERT]	AverageLatency(us)	9940.734236671073
[INSERT]	MinLatency(us)	4624
[INSERT]	MaxLatency(us)	3164159
[INSERT]	95thPercentileLatency(us)	15063
[INSERT]	99thPercentileLatency(us)	35231
[INSERT]	Return=OK	2654452

Table B.56: Complete output from running TimeSeries workload A on the FoundationDB Document Layer.

Appendix C

Source code for custom benchmarking tool

The complete source code for the custom benchmarking tool that was developed for the research of this thesis has been included as a `.zip`-file upon delivery. Additionally, the code has been published to Github, and can be found in [91]. The link points to the latest commit at time of delivery. Finally, the entirety of the tool's `README.md`-file has been included in the following section, as it is referenced several times in the thesis.

C.1 README.md - fdb-benchmarks

Benchmarking tool for comparing MongoDB and FoundationDB Document Layer, written in Python.

Sets up an environment where a set of workloads are performed X amount of times, for each specified database runner. Workloads define different sets of operations to execute on the database, with patterns that emulate real-world scenarios (e.g. Zipf distribution, or read-latest-insert). Each workload implements a method for performing the operations that is adapted and specialized to each database. Outputs the average throughput for each workload per database runner to file, as well as the number of operations performed, grouped on operation type.

C.1.1 First time setup

- Install FoundationDB client and server, and FoundationDB Document Layer (https://www.foundationdb.org/download/)
- Install MongoDB client and daemon (https://docs.mongodb.com/manual/installation/)
- Clone repo
  - SSH: `git clone git@github.com:edvardvb/fdb-benchmarks.git`
  - HTTPS: `git clone https://github.com/edvardvb/fdb-benchmarks.git`
- Enter directory
  - `cd fdb-benchmarks`
- Create virtual environment for Python
  - `python3 -m venv venv`
- Install requirements
  - `source venv/bin/activate`
  - `pip install -r requirements.txt`
- Ensure FDB and FDB-DL is running (see FDB and FDB-DL documentation)
- Start mongo daemon with a replica set
  - `mongod --replSet rs`
- Initiate replica set in mongo client:
  - `mongo`
  - `rs.initiate()`

C.1.2 Regular setup

- Activate virtual environment
  - `source venv/bin/activate`
- Ensure FDB and FDB-DL is running (see FDB and FDB-DL documentation)
- Start mongo daemon with a replica set
  - `mongod -replSet rs`

C.1.3 Usage

- `python test.py`
- **Flags and arguments**
  - `-runners`
    - Required argument, one or more
    - Options:
    - `fdbdl`, `mongo3`, `mongo4`
    - Example: `-runners fdbdl mongo3 mongo4`
    - Performs chosen workloads on FoundationDB Document Layer, standard MongoDB, and MongoDB with transactions.
  - `-workloads`
    - Required argument, one or more
    - Options:
    - `a`, `b`, `c`, `d`, `e` or `f`
    - Example: `-workloads a e f`
    - Performs workloads A, E and F on chosen runners.
  - `-num_runs`
    - Optional argument
    - Takes an integer, signifying the number of runs to perform per workload. Default is 5 runs.
  - `-no_write`
    - Optional flag
    - When set, outputs are not written to file, but still printed in the console.
- Example command:
  - `python test.py -runners fdbdl mongo3 mongo4 -workloads a b c d e f -num_runs 100`
  - Output: Six `.csv`-files for each runner (one for each workload), containing the average ops/sec, as well as total ops grouped on operation type. Prints status per run in console.

Appendix D

Custom tool benchmark outputs

The following sections contain tables with complete outputs for all benchmarks described in chapter 9. They were produced by following the execution described in section 8.3, using the tool described in section 8.1.

D.1 FoundationDB Document Layer benchmark

This section contains solely benchmarks of FoundationDB Document Layer, using the tool described in section 8.1.

Metric	Output value
Number of runs	100
Average runtime	7.983203930854797
Operations per run	10000
Average throughput	1252.6299073170808
Total number of reads	500378
Total number of updates	499622
Read percentage	50.0378%

Table D.1: Complete output from running workload A on FoundationDB Document Layer

Metric	Output value
Number of runs	100
Average runtime	7.077793030738831
Operations per run	10000
Average throughput	1412.8697966400027
Total number of reads	950107
Total number of updates	49893
Read percentage	95.0107%

Table D.2: Complete output from running workload B on FoundationDB Document Layer

Metric	Output value
Number of runs	100
Average runtime	6.100765535831451
Operations per run	10000
Average throughput	1639.1385542137764
Total number of reads	1000000
Read percentage	100.0%

Table D.3: Complete output from running workload C on FoundationDB Document Layer

Metric	Output value
Number of runs	100
Average runtime	6.537935025691986
Operations per run	10000
Average throughput	1529.5349312440717
Total number of reads	949791
Total number of inserts	50209
Read percentage	94.9791%

Table D.4: Complete output from running workload D on FoundationDB Document Layer

Metric	Output value
Number of runs	100
Average runtime	23.381800849437713
Operations per run	10000
Average throughput	427.68305420069817
Total number of reads	950005
Total number of inserts	49995
Read percentage	95.0005%
Average scan length	4.998652638670323

Table D.5: Complete output from running workload E on FoundationDB Document Layer

Metric	Output value
Number of runs	100
Average runtime	10.436050989627837
Operations per run	10000
Average throughput	958.216859034014
Total number of reads	499006
Total number of read-modify-writes	500994
Read percentage	49.9006%

Table D.6: Complete output from running workload F on FoundationDB Document Layer

D.2 Standard MongoDB benchmark

This section contains solely benchmarks of standard MongoDB, using the tool described in section 8.1.

Metric	Output value
Number of runs	100
Average runtime	5.32366206407547
Operations per run	10000
Average throughput	1878.4062323341034
Total number of reads	499830
Total number of updates	500170
Read percentage	49.983%

Table D.7: Complete output from running workload A on standard MongoDB

Metric	Output value
Number of runs	100
Average runtime	5.088876066207885
Operations per run	10000
Average throughput	1965.0704536516198
Total number of reads	950178
Total number of updates	49822
Read percentage	95.0178%

Table D.8: Complete output from running workload B on standard MongoDB

Metric	Output value
Number of runs	100
Average runtime	4.196538217067719
Operations per run	10000
Average throughput	2382.916461794403
Total number of reads	1000000
Read percentage	100.0%

Table D.9: Complete output from running workload C on standard MongoDB

Metric	Output value
Number of runs	100
Average runtime	4.477418222427368
Operations per run	10000
Average throughput	2233.4299596830256
Total number of reads	949693
Total number of inserts	50307
Read percentage	94.9693%

Table D.10: Complete output from running workload D on standard MongoDB

Metric	Output value
Number of runs	100
Average runtime	5.357735698223114
Operations per run	10000
Average throughput	1866.4601173433184
Total number of reads	949967
Total number of inserts	50033
Read percentage	94.9967%
Average scan length	5.497245693797785

Table D.11: Complete output from running workload E on standard MongoDB

Metric	Output value
Number of runs	100
Average runtime	5.0044755625724795
Operations per run	10000
Average throughput	1998.2113759907427
Total number of reads	499775
Total number of read-modify-writes	500225
Read percentage	49.9775%

Table D.12: Complete output from running workload F on standard MongoDB

D.3 Transactional MongoDB benchmark

This section contains solely benchmarks of transactional MongoDB (MongoDB 4.0 with session-based transactions), using the tool described in section 8.1.

Metric	Output value
Number of runs	100
Average runtime	5.517560932636261
Operations per run	10000
Average throughput	1812.3950278193038
Total number of reads	500785
Total number of updates	499215
Read percentage	50.078500000000005%

Table D.13: Complete output from running workload A on transactional MongoDB

Metric	Output value
Number of runs	100
Average runtime	5.2370065450668335
Operations per run	10000
Average throughput	1909.4877796973199
Total number of reads	949977
Total number of updates	50023
Read percentage	94.9977%

Table D.14: Complete output from running workload B on transactional MongoDB

Metric	Output value
Number of runs	100
Average runtime	5.050165932178498
Operations per run	10000
Average throughput	1980.1329568761882
Total number of reads	1000000
Read percentage	100.0%

Table D.15: Complete output from running workload C on transactional MongoDB

Metric	Output value
Number of runs	100
Average runtime	6.9532847547531125
Operations per run	10000
Average throughput	1438.169203866449
Total number of reads	949967
Total number of inserts	50033
Read percentage	94.9967%

Table D.16: Complete output from running workload D on transactional MongoDB

Metric	Output value
Number of runs	100
Average runtime	6.566993496417999
Operations per run	10000
Average throughput	1522.7668499221984
Total number of reads	949738
Total number of inserts	50262
Read percentage	94.9738%
Average scan length	5.502579658811167

Table D.17: Complete output from running workload E on transactional MongoDB

Metric	Output value
Number of runs	100
Average runtime	4.551295847892761
Operations per run	10000
Average throughput	2197.176438141233
Total number of reads	499572
Total number of read-modify-writes	500428
Read percentage	49.9572%

Table D.18: Complete output from running workload F on transactional MongoDB

Edvard Viggaklev Bakken

Evaluating FoundationDB as a replacement for MongoDB

NTNU
Norwegian University of
Science and Technology