Consistency Models for Mobile Games with Cloud Databases

Benjamin Norlin

August 26, 2015
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Johan Tordsson
Examiner: Fredrik Georgsson

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Developing games on smartphones with online features requires careful designing in order to scale to a large amount of users. In this thesis, selected databases are evaluated for use in mobile games, to store game data locally, in terms of performance and usability. The databases are compared to the current storage implementation of a game developed for iOS and Android devices. It is found that the databases perform better when reading or writing small parts of a larger data set, and scale well as the data set increases in size. However, the size of the data in the game studied is not large enough for the increase in complexity to outweigh the gained performance. In addition to this, a library is written to simplify synchronizing state between server and client on mobile games.

Acknowledgements

I would like to thank the people at Turborilla, for giving me the opportunity to do this thesis, their hospitality, and their help. Also, I would like to thank my supervisor, who gave me valuable feedback throughout this thesis. Last but foremost, thanks to my family and Lisa, for your support.

Contents

1 Introduction **1**

2 Problem Description **3**
2.1 Problem Statement . . . 3
2.1.1 Storage . . . 3
2.1.2 Synchronization . . . 3
2.2 Goals and Purposes . . . 4
2.2.1 In-depth study . . . 4
2.2.2 Benchmarking . . . 4
2.2.3 Prototyping . . . 4
2.3 Methods . . . 5
2.3.1 Research . . . 5
2.3.2 Benchmarking . . . 5
2.3.3 Prototype . . . 5

3 Databases **7**
3.1 Why We Use Databases . . . 7
3.2 Relational Databases . . . 7
3.3 NoSQL Databases . . . 8
3.3.1 Key-Value . . . 9
3.3.2 Document . . . 9
3.3.3 Column-family . . . 9
3.3.4 Graph . . . 9

4 Consistency Models in Distributed Systems **11**
4.1 Consistency in Distributed Systems . . . 11
4.1.1 Safety and Liveness . . . 11
4.1.2 Strong and Weak Consistency . . . 11
4.1.3 Eventual Consistency . . . 12
4.2 CAP Theorem . . . 12
4.3 PACELC . . . 13

4.4 ACID . . . 13
4.5 BASE . . . 14
4.6 Metrics . . . 14
4.6.1 Harvest and Yield . . . 14
4.6.2 Probabilistically Bounded Staleness . . . 14

5 Google App Engine 17
5.1 Runtime environments . . . 17
5.2 Applications . . . 17
5.3 Bigtable . . . 18
5.4 Datastore . . . 18
5.4.1 Kinds . . . 18
5.4.2 Entities . . . 19
5.4.3 Properties . . . 19
5.4.4 Indexes . . . 19

6 Mad Skills Motocross 2 21
6.1 Overview . . . 21
6.2 Implementation . . . 22
6.3 Storage . . . 22
6.4 Synchronization . . . 23
6.4.1 Client . . . 23
6.4.2 Server . . . 24

7 Evaluation of Databases for Smartphones 25
7.1 Chosen Databases . . . 25
7.1.1 SQLite and SQLCipher . . . 25
7.1.2 Siaqodb . . . 25
7.1.3 DBreeze and UnDBreeze . . . 25
7.2 Benchmarking Method . . . 26
7.2.1 Batch Performance . . . 26
7.2.2 Random Access Performance . . . 26
7.3 Benchmarking Results . . . 29
7.3.1 Batch Performance . . . 29
7.3.2 Random Access Performance . . . 30
7.3.3 Disk Usage . . . 33
7.3.4 Build Size . . . 33
7.4 Feature Comparison . . . 34
7.4.1 Supported Platforms . . . 34
7.4.2 Fault tolerance . . . 35
7.4.3 API . . . 35

8 A Synchronization Library for Mad Skill Motocross 2 **37**
8.1 Goals . . . 37
8.2 Design . . . 37
8.2.1 Merging Operators . . . 37
8.2.2 Forward Compatibility . . . 39

9 Conclusion **41**
9.1 Performance Evaluation . . . 41
9.2 Synchronization . . . 42
9.3 Limitations . . . 42
9.3.1 Benchmarks . . . 42
9.3.2 Synchronization . . . 42
9.4 Future work . . . 43

List of Figures

3.1 Visualization of relational database terminology. Taken from Wikipedia [1] . 8

4.1 An example of partitioning where server nodes N_1 and N_2 are unable to communicate with each other as a result of a lost connection. The clients of N_1 and N_2 can still communicate with the node they are connected to. 13

6.1 Screenshot from MSM2 showing in-game footage. Screenshot is copyright Turborilla AB. [2] 21

6.2 UML class diagram showing selected methods of the Synchronizable interface. 22

6.3 The flow of serialization and deserialization of storable objects in MSM2. . . 23

7.1 Comparison of batch read performance between the different storage formats. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values. 29

7.2 Comparison of batch write performance between the different storage formats. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values. 30

7.3 Comparison of single lookup performance between the different storage formats. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values. 31

7.4 Comparison of single update performance between the different storage solutions. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values. 31

7.5 Comparison of single update performance between the different storage solutions with a logarithmic y axis. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values. 32

7.6 Comparison of disk usage of the different storage solutions. 33

List of Tables

5.1 Comparison of Datastore terminology to that of the relational database model. Note that the terms are not entirely equivalent, but similar in concept. 18

6.1 Methods of the Synchronizable interface in MSM2. 22

6.2 Some of the different storage classes that implement Synchronizable in MSM2. 22

7.1 An approximation of the increase in build size due to using different storage back-ends. 34

7.2 Comparison of platform support between databases. 35

8.1 A description of the supported operators in the synchronization library 38

Chapter 1

Introduction

Turborilla AB [3] develops and maintains games for iOS [4] and Android[5] devices. In their latest game, Mad Skills Motocross 2 (MSM2)[6], they make use of Google App Engine [7] to store data pertaining to the players of the game, such as their progress and account information. JSON-formatted [8] files are used locally on the clients and the contents are synchronized with a server running on the Google App Engine cloud platform. This storage model has evolved over time, and the amount of files has increased and new features such as encryption have been added. Because of this, the code has become cumbersome to maintain and some usage scenarios lead to errors and inconsistencies in the data. Turborilla's wish is to eliminate their current problems associated with synchronizing data between the servers on Google App Engine and the clients on the mobile devices, and to make their system easier to maintain.

Certain challenges appear when creating applications with online features such as those developed by Turborilla. These include achieving a responsive application that also scales with a large number of users. Cloud services such as Google App Engine are great tools which are built for overcoming these challenges. However, applications must be designed to fit the cloud paradigm and deal with the limitations that come with it in order to reach these design goals. This report covers some of the implementation details of cloud services, such as Google App Engine, and distributed systems to examine why the limitations exist and some of the history behind them. Mobile devices, the target platforms of MSM2, are limited in processing power and other resources. This makes performance an important consideration when creating applications that have timing requirements, such as games.

Chapter 2

Problem Description

2.1 Problem Statement

This section describes the problems that have been identified with Turborilla's current setup.

2.1.1 Storage

Currently, the application stores data as standard text files, one file per category of user data. Examples of categories are purchases and unlocked achievements. The files are also encrypted to make tampering with them more difficult. When data of a certain category is updated and needs to be persisted, the entire file is rewritten, including both the updated data and any parts that have not changed since the last write which is redundant. These writes decrease performance and can thus potentially impact user experience in a negative way.

The source code for handling local storage in MSM2 has evolved over time which has led to each category being handled in a slightly different way. This makes it hard to maintain and add upon in some cases. A more unified design would remove these obstacles and improve usability and maintainability.

2.1.2 Synchronization

Most of the data that is stored locally is also synchronized with servers running on Google App Engine. The data is synchronized periodically and at key times. When synchronization happens, a client downloads all the data from the server and merges it with the local data. Any files with more recent local data are uploaded to the server with the latest data.

One of the problems with the current design is that it is difficult to maintain because the merging of data is handled differently in each category. A unified design of this behaviour would reduce the amount of code needed when adding features, and thus increase maintainability and decrease development times.

Another problem with the current design is that there are issues with concurrency. Two clients who are synchronizing concurrently read from the server, merge and update their data locally, and then upload any changes back to the server. This means that whichever client finishes last overwrites the other clients' changes as the server does not know how the data should be merged.

2.2 Goals and Purposes

There are three goals of this project.

1. Perform an in-depth study of consistency models in distributed systems
2. Evaluate the use of a database in MSM2
3. Design and implement a prototype storage and synchronization library

2.2.1 In-depth study

The in-depth study of this project covers consistency models in distributed web scale systems in order to gain a theoretical understanding of the related problems. Creating reliable distributed systems, including synchronizing state and achieving consensus, is hard. The in-depth study helps us avoid common errors and learn from already proven methods. In this study, different consistency models, as well as different database designs are studied and compared. Google App Engine and the Datastore is researched in more detail, with regards to how it is used and its consistency model.

A fundamental aspect of distributed systems is that, according to the CAP theorem [9], they can only simultaneously achieve two out of the three properties consistency, availability and partition tolerance. This aspect and its consequences is also examined in more detail in the in-depth study.

2.2.2 Benchmarking

The benchmarking is done in order to answer the question if using a database locally on the devices would improve the application. In order to answer this question, different database solutions are researched and compared with the original version. For each solution candidate, we measure what effect it has on device performance, and what other features it has that can affect the application. The following metrics are compared in the benchmark:

- Execution time;
- Size of the data stored on disk;
- Estimated addition to build size.

In addition to the benchmarks, the usability of the candidates are compared as well as their fault tolerance and support for different platforms. It is important to state that deciding which storage system to use can often be a business decision and not necessarily a question of optimizing performance. This means that measuring the performance of the databases is only a part of what needs to be done in order to make a decision.

2.2.3 Prototyping

A prototype is implemented in order to simplify the synchronization of data between clients and servers in the application. This is to answer the questions of what problems the current solution has, and how these can be rectified.

2.3 Methods

This project consists of three parts: an in-depth study, evaluation of storage back-ends, and the building of a prototype.

2.3.1 Research

An in-depth study is made on the subject of consistency models in order to understand the underlying problems in applications that make use of distributed systems. This phase consists of a literature study, which includes research papers and publications on related subjects. It also includes searching for available methods and tools for solving the problem at hand and identifying their qualities and their suitability for synchronization of mobile game data.

2.3.2 Benchmarking

A set of selected database solutions are evaluated and compared with the current storage implementation using the metrics described in the goals section. A benchmark application is written that produces measurable data on the performance of the different storage formats. The results of the benchmarking application being run are then presented and compared. As a result, one solution is chosen for the prototype implementation.

2.3.3 Prototype

The prototype is developed as a modification of Turborilla's application MSM2 based on the findings of the in-depth study, benchmarks and the requirements of the prototype. Because MSM2 is currently being maintained and is continuously updated, the prototype is based on the latest version of the software.

Turborilla's developers use Scrum [10] in their daily software development, which is an agile and iterative process. The development of the prototype in this project is also done in a Scrum-like process so that it fits in with the workflow of the rest of the organization at which the project is carried out. This also enables Turborilla to give feedback and help in a structured way, to increase the chances of the prototype being a success.

Chapter 3

Databases

This chapter briefly covers the subject of databases and why they are used. Also, we compare traditional relational databases to the more modern term NoSQL. We look at the origin of the term NoSQL, and examples of data models commonly found in NoSQL databases. We will briefly explain some of the history that has led to the current situation of databases in distributed systems.

3.1 Why We Use Databases

Primary storage is volatile and typically lost when the application stops. To address this, we use secondary memory such as hard drives, flash memory, and other storage mediums that maintain data even when powered off. Storing data on secondary storage can be done in a number of ways, and we commonly choose between storing data in files using various file formats, databases or some other type of storage.

Databases are one way to store data in an organized fashion, and the advantages of this are many. If used correctly, the advantages include flexibility, security, ease-of-use, performance and reliability. On the other hand, if used incorrectly we would experience the opposite of those benefits. Databases are a fundamental building block of many applications, in many different business fields to store application and business data. Until recently, choosing a database often meant choosing one of many relational database systems, as this has been the industry standard since they first became popular.

While there are many advantages to databases, they do not fit every storage problem. Depending on the problem, they can introduce an overhead in development complexity and requirements on storage space and processing power. This is perhaps the most obvious with a relatively small amount of data, when we must ask if it is worth the added effort. At a certain point as the data size increases, the advantages of using a database will outweigh the overhead. Where this point lies exactly is difficult to predict, and is specific to every situation.

3.2 Relational Databases

The dominating type of database is the relational database [11]. The relational database model was invented by Codd [12] in 1970. Relational databases are built on schemas which define relations, attributes and tuples. These can be visualized as tables with rows and

columns, see Figure 3.1. Tuples are unique and are often, but not necessarily, identified using a primary key. Relations between tables are built using keys to the primary keys of other relations, so-called foreign keys.

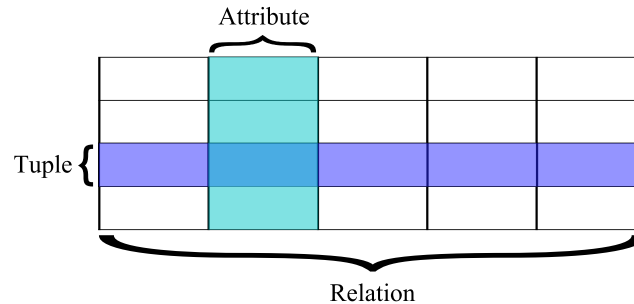

Figure 3.1: Visualization of relational database terminology. Taken from Wikipedia [1]
.

Relational databases are managed using a relational database management system (RDBMS). Examples of RDBMSs include Microsoft SQL Server [13], MySQL [14] and PostgreSQL [15]. The standard way of communicating with relational database systems is with a language called Structured Query Language (SQL) defined by ISO/IEC 9075 [16]. Many RDBMSs support SQL with minor variations.

Big data and huge web scale systems have outgrown the capacity of what is feasible for a traditional relational database. Relational databases are not suited for handling large volumes of read-write intensive data.

This is due to scaling, and scaling relational databases, at least traditionally, means adding more resources to a computer in order to handle a greater load. This is called scaling up, or scaling vertically. There is a limit to how much processing power that can be put into one machine and powerful machines are expensive. It is cheaper to buy large amounts of commodity hardware to run as a cluster. This is called scaling out, or scaling horizontally. This creates a new set of problems, as the database must be replicated or split up in some way, so called sharding [17]. This is something most traditional relational databases were not designed for. It also requires applications to be aware of how the database is sharded, which adds complexity to the application.

3.3 NoSQL Databases

NoSQL is used to describe some types of nonrelational databases. According to Sadalage and Fowler [17] there is no strict definition of NoSQL, but they give a description of some common characteristics of NoSQL databases. Wikipedia gives the definition "A NoSQL (often interpreted as Not only SQL) database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases" [18]. The modern usage of the term "NoSQL" was first used to name a meetup on "open-source, distributed, nonrelational databases" in 2009 [19]. Commonly mentioned examples of NoSQL databases are Cassandra [20], Riak [21], MongoDB [22] and Neo4j[23].

Sadalage and Fowler [17] claim that NoSQL databases were introduced to address some of the problems that exist with relational databases. More specifically, the need to handle massive amounts of data required distributed databases. Contrary to relational databases, NoSQL databases are commonly designed to run on clusters. The authors also note that,

even though NoSQL was conceived to handle large amounts of data and running in clusters, there are advantages to using NoSQL databases even in single node systems. According to Sadalage and Fowler, there is a so called impedance mismatch between the relational data model and many in-memory data structures. By not limiting the choice to relational databases, we can choose a database that supports a data structure that is closer to our in-memory data structure. This can simplify applications by removing the need for translation between data models.

Sadalage and Fowler refer to the notion of aggregates, which originates in Domain-Driven Design [24]. Aggregates are collections of related objects that should be treated as one unit. Within the field of NoSQL databases, different databases have different types of aggregates. These different types are put into categories and are described below.

3.3.1 Key-Value

One example of an aggregate is the key-value database. These types of databases store key-value pairs with no particular notion of what the value contains or of what type it is. This gives us the freedom to store anything as a value. The downside is that we can only query the database for the entire value belonging to a certain key, and not for parts of the values or base a query on the values. Amazon's Dynamo [25] is an example of a key-value store.

3.3.2 Document

A similar category of databases is document databases, which could be seen as a key-value store where the value has a structure. This structured data is called documents. Since the database has knowledge of the structure of the documents, we have the opportunity to retrieve or update parts of the documents as well as make queries based on the value.

3.3.3 Column-family

Another category is column-family databases. In these databases, rows are the aggregate and each row maps to column families. These column families are groups of key-value pairs pertaining to some category of data for the aggregate. As an example, we could want to store customer data in a column-family database. We could assign an id to the customer and use that as the row key. In that row we could store, for example, profile information and purchases in one column-family each. Examples of column-family stores are Bigtable by Google [26] and Cassandra [27], which was originally developed at Facebook.

3.3.4 Graph

The last category is graph databases. As the name suggests, they store data as graphs. In graph databases data is stored as nodes and with edges between the nodes representing relations. This is useful for large sets of simple data items with a complex web of relations between them. This category stands out as it is not primarily a result of the need to distribute a database, but rather to to handle queries with a large amount of joins, which is slow on traditional RDBMSs [17].

Chapter 4

Consistency Models in Distributed Systems

This chapter introduces the subject of consistency models in distributed systems. In order to explain this concept, a few illustrative examples of consistency models are described.

4.1 Consistency in Distributed Systems

A distributed system is a system in which components of networked computers communicate and coordinate their actions only through message passing [28]. In other words, maintaining system state across the different processes requires communication. Consistency is a term used to describe how much the processes agree on the state. In a perfectly consistent distributed system, all processes agree on the same state. As we have learned from history, this is not always possible because of the many challenges of distributed computing [29].

4.1.1 Safety and Liveness

Safety and liveness are two important notions in distributed systems. Informally, if a system guarantees safety, it guarantees that nothing bad happens. This could, for example, be that the application crashes or some important data is lost. Liveness is a property that guarantees that, eventually, something good happens. For example, the server we are communicating with responds to our messages. All properties of a distributed system can be described as an intersection of safety and liveness [30].

4.1.2 Strong and Weak Consistency

Consistency models range from strong to weak. In a strict consistency model, all processes always see the most recent values when reading. It is the strongest form of consistency and is equivalent in behaviour to having only a single process. This model of consistency is costly, reduces the ability to do work in parallel, and can be difficult to implement in a distributed system. Because of this, it is not widely used in web-scale systems that often need to be highly available.

There are other consistency models that are weaker than strict consistency. These models relax constraints in some way in order to allow for increased concurrency or other benefits.

4.1.3 Eventual Consistency

Eventual consistency is a form of weak consistency that, given that no updates are made to an item, all reads of that item will eventually see the same value [31]. In other words, it guarantees liveness, without specifying when it happens. This is widely used in web-scale applications because of the latency and availability benefits [31]. The fact that it is widely used may be surprising considering that it gives no guarantees on *when* the system will be consistent. However, as Bailis and Ghodsi [31] has shown, it can and often does in practice behave strongly consistent within a small time frame, with higher performance than a strongly consistent counterpart. Strong consistency is also not needed in many applications, especially in those that deal with non-critical data such as social networks and chats. Having different clients seeing events in slightly different orderings does not break these systems and allowing this to happen can enable huge performance improvements.

4.2 CAP Theorem

According to Brewer [32], he invented the CAP Theorem in 1998 and it was first published in Fox and Brewer [33]'s paper in 1999. Brewer conjected that it is impossible for a web service to guarantee consistency (C), availability (A) and partition-tolerance (P) simultaneously. In other words, we must choose two of these properties when designing distributed applications. Originally called the CAP principle, it was later modeled and presented as a theorem in a paper by Gilbert and Lynch [9], after which it has been known as the CAP theorem, Brewer's CAP theorem, or simply Brewer's theorem. The definitions of the three CAP properties from Gilbert and Lynch are shown below.

Consistency: There must exist a total order on all operations such that each operation looks as if it were completed at a single instant.

Availability: Every request received by a non-failing node in the system must result in a response.

Partition-tolerance: No set of failures less than total network failure is allowed to cause the system to respond incorrectly.

The theorem defines consistency as atomic, or linearizable, consistency. In this definition, operations must appear to have been run on a single node. The reason for using this definition is that it provides an easy model to understand [9].

To illustrate the theorem, we consider a distributed system containing two server nodes, N_1 and N_2, providing a service that allows clients to read and update data. N_1 and N_2 replicate this data so that if a client connected to N_1 updates the data, N_1 propagates this update to N_2 and vice versa. The nodes each have their respective connected clients. This system is defined to be consistent in the CAP sense because there is only one version of the data in any part of the system. If the connection between the servers is lost, N_1 and N_2 are said to be in different partitions, see Figure 4.1. In this scenario, each server can still communicate with their respective clients but there is no communication between servers N_1 and N_2. If we allow clients to update the data on the server they are connected to, the system is no longer consistent as the servers do not see the same data. On the other hand, if we do not allow clients to update, the system is by definition no longer available.

While the applicability and usefulness of the CAP theorem has been discussed [34] [35], CAP has become widely spread and cited as a means of reasoning about different trade-offs

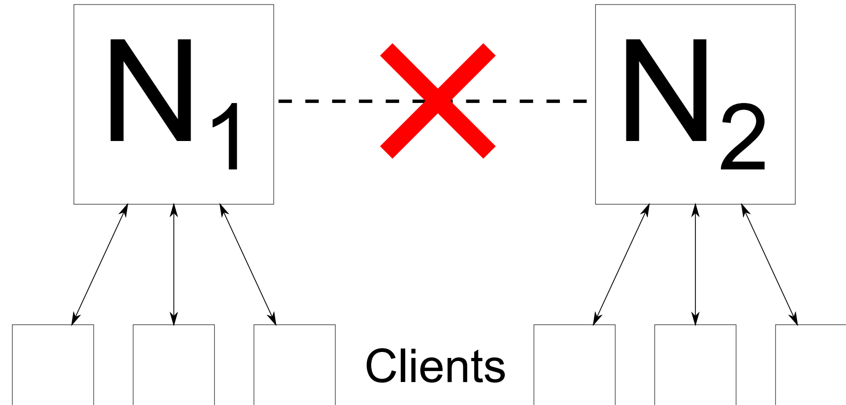

Figure 4.1: An example of partitioning where server nodes N_1 and N_2 are unable to communicate with each other as a result of a lost connection. The clients of N_1 and N_2 can still communicate with the node they are connected to.

needed in distributed systems. One problem of the theorem is that it does not include a notion of latency, which is a real problem [29] in distributed systems. Brewer [32] later added to the discussion about CAP that it can be confusing to think about it in terms of choosing two out of three of these properties, and that this notion is outdated. Instead, we should consider true partitions to be a rare occurance, and in normal operation there is no point in forfeiting consistency or availability. When partitions do occur, we should detect and handle this explicitly. Brewer [32] also states that, in practice, we do not choose between consistency and availability in terms of one or the other, exclusively. Instead, there is a continuous scale of choices between consistency and availability. Different applications, and even operations within a single application, can have different demands on consistency, thus creating the opportunity to achieve high availability even in face of partitions.

4.3 PACELC

In response to confusion about the CAP theorem and its disregard of latency, Abadi [35] suggests another acronym; PACELC. The meaning of PACELC is that if there is a partition (P), we must trade off availability (A) and consistency (C). Else (E), with no partitions, we must trade off latency (L) and consistency (C). This clears up the misconception, which is common according to Abadi, that the CAP theorem states that we must give up consistency or availability even in the absence of partitions. Abadi also clarifies that the trade-off between latency and consistency only applies to systems that use replication of data, otherwise any node failure always means a loss of availability.

4.4 ACID

ACID [36] is an acronym for Atomicity, Consistency, Isolation, and Durability, and it describes properties of a database transaction. These properties have been under a lot of research and are used to characterize databases and database replication as follows.

Atomicity: A transaction is either completed fully, or aborted.

Consistency: A transaction must put the database in a valid state, not breaking any rules.

Isolation: A transaction can read only committed writes.

Durability: If a transaction completes, it must persist (and never be undone by a crash, for example).

Consistency can be defined differently depending on who we ask and in which context. When referring to ACID transactions in databases, it is usually said to mean that no integrity constraints may be violated. These constraints are defined and checked in the database using aspects like foreign keys and triggers. Outside of transactions, consistency can also mean that the business rules of an application such as, never having a negative account balance, must not be violated. A general-purpose database only knows about constraints specifically specified within the database.

4.5 BASE

BASE is an acronym (or backronym) that stands for Basically Available, Soft state and Eventually consistent. Pritchett [37] claims that BASE can be seen as the opposite of ACID, where the ACID model gives a pessimistic view of consistency, forcing the database to make every operation consistent, upholding the illusion of serial execution. On the other hand, BASE models are optimistic in that they assume that things will work correctly most of the time, and when bad things do happen we handle the problems explicitly.

In practice, this means that we can achieve higher perceived availability if we accept that consistency fluctuates. Pritchett [37] gives an example in which users are partitioned over five database servers. He claims that BASE encourages us to design the system so that the failure can only be seen by those users connected to the server that failed, thus maintaining availability for the majority of users. This is in contrast to an ACID system, which would make sure no further operations succeed until all five servers are up.

4.6 Metrics

In order to measure the correctness of a distributed system we need performance metrics. This section lists some metrics suggested by different researchers that can be used to characterize systems based on correctness.

4.6.1 Harvest and Yield

Fox and Brewer [33] suggested the two metrics harvest and yield. Yield is defined as the probability of completing a request, and harvest as the fraction of the data reflected in the response. In their paper, Fox and Brewer claim that yield is the common metric, often measured in *nines*, such that *four-nines availability* means that the probability of a completed response is 0.9999.

4.6.2 Probabilistically Bounded Staleness

Another metric is Probabilistically Bounded Staleness [38] (PBS). Bailis and Ghodsi [31] state the usefulness of being able to measure a running system by monitoring its performance, and being able to predict how certain configurations or workloads affects the system.

We can measure the time it takes for writes to become visible to readers. The versions metric is applied to read operations and measures how old the data returned is. For example, say we read a data item that has been updated three times, but we only see the two first updates because of slow propagation. That read is then said to be one version old.

PBS consists of t-visibility, which is a measure of time, and k-staleness which is a measure of versions. k-staleness is the probability of reading a version at most k versions old, whereas t-visibility is the probability that a read operation at t seconds after the latest write sees the latest value. These can be combined into PBS $\langle k, t \rangle$-staleness which models the probability that a read will an at most k versions old value if the last write was at least t seconds ago.

Chapter 5

Google App Engine

Google App Engine is a Platform-as-a-Service (PaaS) offered by Google [7]. This chapter describes, in some detail, the parts of Google App Engine that are used in MSM2. Portions of this chapter are reproduced from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License. This applies to the documentation of Google's Cloud services [39].

5.1 Runtime environments

Google App Engine supplies a number of runtime environments in which uploaded apps are run. At the time of writing it supports applications written in Java, Python, Go and PHP. The environments are sandboxed, i.e. isolated from other applications, and the engine provides automatic load balancing and scaling. In addition to the runtime environments, Google offers specialized services such as storage and message queues that help developers to build complete applications running on Google App Engine.

5.2 Applications

Applications on the Google App Engine are developed using one of the supported languages and are then uploaded to Google's servers. Every application must have an id which is a string that is created by the developer or randomly generated at the time of creation. The id is used as a key for identifying the application and the objects stored in the datastore. Applications are served on the `appspot.com` domain as `<application-id>.appspot.com` by default, or they can be set up to be served on a custom domain.

5.3 Bigtable

Bigtable is a distributed storage system designed by Google for structured data, and is used by internal systems within Google's infrastructure [40]. It has not been released to the public and there is no documentation of the implementation other than the paper released by Google. There is, however, an open source implementation called HBase which is modeled after Bigtable but on Apache infrastructure instead [41].

According to Google's paper on Bigtable it was designed to meet the varied storage needs of their internal systems. The main points brought up are high applicability, scalability, performance and availability. While Bigtable itself is not offered to the public, it is used by many services that Google supplies, specifically the Datastore in Google App Engine [42].

5.4 Datastore

The Datastore on Google App Engine is an object store [43]. There are two configurations of the Datastore, the Master/Slave Datastore and the High Replication Datastore (HRD), where the former is deprecated as of April 4, 2012 [44]. The HRD uses replication across datacenters and the Paxos algorithm for consensus among replicas [43] [45].

The Datastore stores data objects called entities. Entities have one or more properties which are named values of supported data types. Kinds are used to categorize and identify entities. Table 5.1 compares this terminology to that of relational databases. In the sections below we describe these concepts and how they are used.

Table 5.1: Comparison of Datastore terminology to that of the relational database model. Note that the terms are not entirely equivalent, but similar in concept.

Datastore	**Relational Database**
Kind	Table
Entity	Row
Property	Column

5.4.1 Kinds

Kinds are the equivalent of tables in relational databases. They describe the intent of what kinds of objects are stored in it. In practice, they are strings which represent some category of objects, for example "Person" or "Product". Kinds do not need to be created explicitly as they are created when an entity of a certain kind is created. They are also schemaless, in that they do not require entities of a certain kind to conform to any form of schema.

5.4.2 Entities

Entities are the objects stored in the datastore, analogous to rows in a relational database. All operations that are done on an entity are done in atomic transactions. Entities can be grouped into entity groups enabling transactions on an entire entity group. Entity groups are created using a hierachical model, where each entity either has a parent node or is itself a parent (with or without children).

All entities related to App Engine applications are stored in one Bigtable [42]. Entities are identified by the application id and the entity path, see Listing 5.1. The path is a concatenation of the keys of all of the entity key's ancestors.

Listing 5.1: The listing shows the structure of a Datastore entity key using JSON.

```
{
  "partitionId": {
    "datasetId": string,
    "namespace": string
  },
  "path": [
    {
      "kind": string,
      "id": long,
      "name": string
    }
  ]
},
```

5.4.3 Properties

Entities can have any number of properties, which are named values of a given type, similar to that of columns in relational databases. The properties must be of a type supported by the Datastore, which include types such as strings, integers and references to other entities. Data types of a property do not have to be the same for all properties of a given entity.

5.4.4 Indexes

Because of the structure of Bigtable and the Datastore, the operations and queries supported by the Datastore are limited compared to a traditional relational database. Datastore supports create, read, update and delete (CRUD) operations, which are also run as atomic transactions within an entity group. However, there is a very limited support for queries and no support for joins, which is heavily used in relational databases.

Queries are supported through the use of indexes, of which there are four: the kind index, two single-property indexes and the composite index. The kind index supports queries on kinds, i.e. "give me all entities of kind X". Single properties can also be queried, which is enabled by the single-property index. This allows us to ask queries such as "give me all entities whose property 'name' has the value of John", or those matching a certain range of values for a property. It also supports ordering, using one index stored in ascending order and another in descending order. Lastly, the composite index supports queries on multiple properties or on ancestors, e.g. "give me all entities whose first name is X and last name is Y".

These indexes are stored in their own Bigtables and are updated along with any updates to the main Bigtable. All of these tables, except for the composite index, are automatically managed by App Engine and added for each individual property. As we can see, Google have opted to support queries by using additional storage and updates of indexes. One write to the datastore using the API results in multiple writes to the datastore as each involved index must be updated. The benefit of this is that because of the underlying Bigtable implementation, all queries are implemented as dense scans, without the need for filtering. This results in a constant number of disk seeks per query and the performance scales based on the size of the results, and not the total size of the tables.

Chapter 6

Mad Skills Motocross 2

This chapter describes the parts of MSM2 which were looked at for this project. We describe the behaviour of the storage and synchronization parts and their limitations.

6.1 Overview

Figure 6.1: Screenshot from MSM2 showing in-game footage. Screenshot is copyright Turborilla AB. [2]

MSM2 is a 2D side-scrolling motocross racing game developed by Turborilla AB. It is part of a series of extreme sport games called Mad Skills. Figure 6.1 shows an in-game screenshot in which two players are racing against each other. The game objective is to race to the finish against the computer in career mode or other players in the multiplayer modes. As a part of the game, players can unlock new bikes and bike parts, such as skins and colors for the rider and the bikes. This can be done either by fulfilling the required achievements or by paying. In the single player modes, the players can use rockets for a burst of speed while racing. These rockets can also be bought in the in-game store.

The game runs on iOS and Android phones and is backed by a server on Google App Engine. User data and game data are mostly stored in the Datastore. This enables players to compete worldwide, and the server can keep global high scores.

6.2 Implementation

The game is made using Unity, which is a game engine and a tool for developing games for a variety of platforms [46]. MSM2 is being developed and maintained for iOS and Android platforms and is written in C#. Unity supports a few different programming languages, C# being one of them. Support for C# is achieved using the Mono framework which is an open source implementation of the .NET framework [47].

6.3 Storage

The parts of the stored data that we are interested in are stored in both the server and the clients. In the original solution, storable objects are implemented as singleton classes that implement an interface called Synchronizable. Any class that needs to be stored and synchronized with the server may implement this interface. Figure 6.2 shows selected methods of the Synchronizable interface as an UML class diagram and Table 6.1 gives a brief description of each of these methods. Table 6.2 shows some of the classes in MSM2 which implement the interface and what they store.

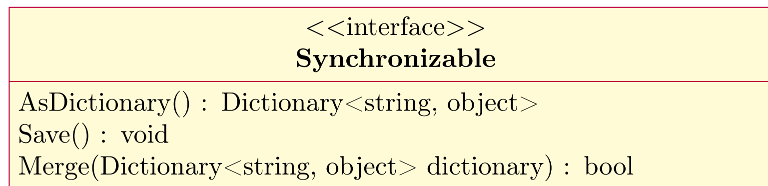

Figure 6.2: UML class diagram showing selected methods of the Synchronizable interface.

Table 6.1: Methods of the Synchronizable interface in MSM2.

Method	Description
AsDictionary	Returns a JSON-friendly dictionary that represents the object.
Save	Saves the object to local storage.
Merge	Merges the object with the contents of the specified dictionary. Returns true if current state is newer than the merged dictionary.

Table 6.2: Some of the different storage classes that implement Synchronizable in MSM2.

Class	Description
Purchases	The user's purchased and unlocked items in the game, such as rockets and bikes.
AchievementSystem	The user's unlocked achievements.
PrivateProfile	The user's private profile.
PublicProfile	The user's public profile.

One implication of this interface is that all storable objects are converted to a dictionary before serialization. This is because a dictionary can be converted into JSON in a straightforward way, and vice versa. Figure 6.3 describes the process of serialization and deserialization of storable objects in MSM2.

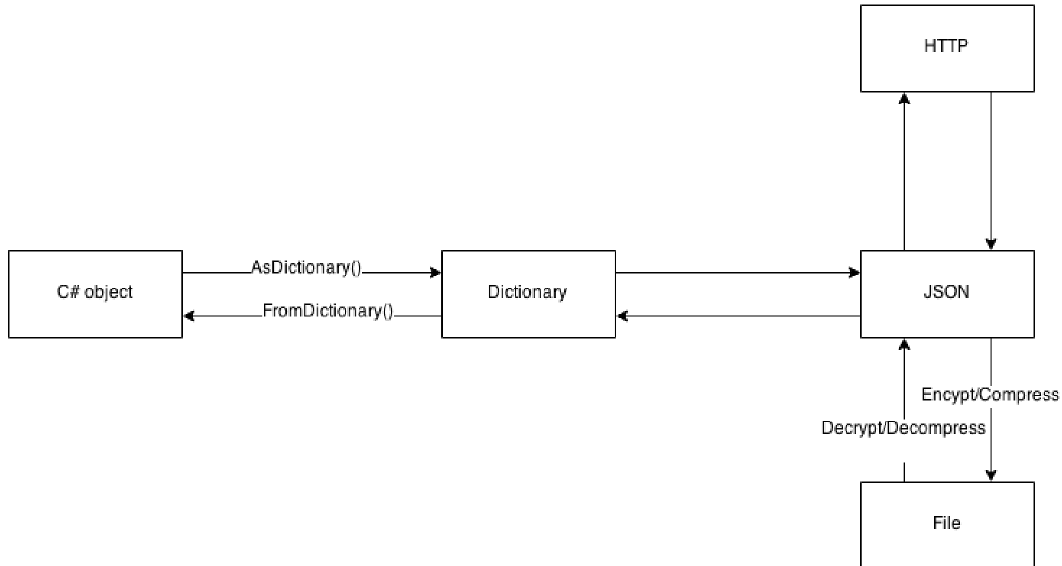

Figure 6.3: The flow of serialization and deserialization of storable objects in MSM2.

6.4 Synchronization

The synchronization of data between the server and the clients is set up to happen periodically and at chosen times. We describe below the behaviours of the client and server when synchronizing their data.

6.4.1 Client

When synchronization occurs, the client asks the server for all the data it has on the user. When the client receives this data, it merges the contents of the local and server data. If the client detects that the local data is more recent than the data on the server, it sends the merged data to the server. All the merging logic is done in the clients, as the server acts as a simple data store and has no way of ensuring application-level consistency. This means that if two clients concurrently update the same data, the values of the latest writes are used which can result in lost updates.

The merging of data is handled on a per-class basis in the Merge method of the Synchronizable interface. This is shown in Table 6.2 and Table 6.1. The implementation of Merge must make sure that each item in the dictionary is merged correctly according to the application logic. For example, merging after having unlocked an achievement on the client or making a purchase must preserve these facts. The merging is made using simple rules like always keeping either the client's or the server's data in case of conflict, or keeping the lowest or highest value and so on. This is done explicitly (with or without loops) for each field in many of the storage classes.

6.4.2 Server

The server is implemented as servlets in Java that listen to requests from clients via HTTP. One servlet on the server represents a task which is typically to update or retrieve some set of data. For example, there is one servlet for retrieving a user's profile and another servlet for updating a user's profile. The data is persisted on Google's Datastore.

Chapter 7

Evaluation of Databases for Smartphones

This chapter describes the methods used to benchmark the different databases and the original storage solution. The results of the benchmark and the conclusions drawn are also presented.

7.1 Chosen Databases

This section describes the different databases that were examined and benchmarked in this project.

7.1.1 SQLite and SQLCipher

SQLite is an open-source embedded SQL database engine written in C. [48] It has a small footprint and needs no external server process, making it useful for embedded systems and mobile devices. It is supported on both iOS and Android.

SQLCipher is an open-source extension to SQLite that enables encryption of the database file. [49] It builds upon the SQLite source code and adds methods for encrypting and decrypting databases using keys. SQLCipher does not have the widespread adoption as SQLite, but since the source code is available we can build the necessary binaries to support any of the platforms in Unity that support so called native plugins.

7.1.2 Siaqodb

Siaqodb is an embedded object and document database engine. [50] It supports multiple platforms, including .Net and Mono, and has specific support for Unity3D. It supports encryption using two built-in algorithms or a custom algorithm.

7.1.3 DBreeze and UnDBreeze

In addition to the databases described above, we examined a few more databases which were not included in the final benchmarks. One of these databases is DBreeze. DBreeze is an open-source embedded key-value store written in C# for .NET and Mono. [51] UnDBreeze is an asset in the Unity asset store that claims to use a modified version of DBreeze patched

and packaged to work with Unity and support encryption. UnDBreeze was evaluated and tested during the benchmark but was not included in the final runs. This is because the encryption used in UnDBreeze was not considered good enough as only the values were encrypted before storing it in the database. Using this technique, any database could be considered to support encryption.

DBreeze is mentioned here because it is the only database found that has the option to store the database in memory that would make it usable on platforms that do not support writing to a file system, such as Unity's Web Player. The limitations of Unity projects running in web browsers are described in more detail in Section 7.4.1 Supported Platforms.

7.2 Benchmarking Method

The purpose of the benchmark is to examine how the use of a database differs in performance compared to storing data in files. We are interested in finding out when or if there are situations when databases are worse or better in performance.

We constructed two categories of performance tests, one for batch reads and writes one for randomly updating or retrieving a single item in a populated database. These tests are described in detail in their respective sections below.

In all the tests we used a very simple C#-class called `KeyValuePair` described by C#-code similar to the following:

```
class KeyValuePair {
    public string Key;
    public string Value;
}
```

Getting accurate benchmarking results on multitasking systems can be difficult because there are many factors that affect execution time. We can mitigate this by repeating the tests multiple times and calculating the average execution time. In addition to sharing CPU time with other applications, applications developed in Unity have their memory managed and garbage collected. The garbage collector can run at any time and cannot be turned off reliably. If the garbage collector runs when we are measuring execution time the results are skewed. In order to reduce the chance of this happening, we run the garbage collector between serializations when we are not measuring execution time. This might reduce the likelihood of it running during a test, but there are still no guarantees.

7.2.1 Batch Performance

By batch performance we mean writing and reading an entire category of data, in other words a table in the database or an entire file using the original solution. This is the behaviour of the original solution, as all writes cause a rewrite of the entire file. We expect to see the database to have lower performance in this case, but we want to know by how much and how the different databases behave.

Algorithm 1 shows, in pseudocode, how the batch serialization benchmarks are run and measured. Algorithm 2 shows the same for the batch deserialization.

7.2.2 Random Access Performance

In the random access performance tests we write or read one randomly selected item instead of everything at once. We expect the original solution to perform worse than a database

Algorithm 1 Measuring batch serialization

```
1:  // N is the maximum number of items to serialize
2:  while n < N do
3:      items ← n objects using random values
4:      for r ← 0, repetitions do
5:          for all S ∈ Serializers do
6:              Start a timer
7:              S.Serialize(items)
8:              Stop the timer and save the elapsed time
9:          end for
10:     end for
11:     increase n
12: end while
13: Calculate the average execution time for each serializer.
```

Algorithm 2 Measuring batch deserialization

```
1:  // N is the maximum number of items to deserialize
2:  while n < N do
3:      for r ← 0, repetitions do
4:          Serialize n objects to disk
5:          for all D ∈ Deserializers do
6:              Start a timer
7:              // The previously serialized data is used here
8:              D.Deserialize(n)
9:              Stop the timer and save the elapsed time
10:         end for
11:     end for
12:     increase n
13: end while
14: Calculate the average execution time for each serializer.
```

when writing and reading as the total number of items grows, because it always writes or read the entire file. Reading just a single item is not possible in the original solution so it is implemented by reading the entire data set into a dictionary and then performing a lookup by key on that dictionary. Algorithm 3 shows, in pseudocode, how the single inserts benchmark is constructed and Algorithm 4 shows the same for the retrieval of a single item.

Algorithm 3 Benchmark for updating a single item

1: // N is the maximum number of items to serialize
2: **while** $n < N$ **do**
3:     **for** $r \leftarrow 0, repetitions$ **do**
4:         Serialize n items to disk
5:         // Choose existing item to update
6:         $item \leftarrow$ RandomItem
7:         $item.value \leftarrow$RandomString
8:         **for all** $S \in$ Serializers **do**
9:             Start a timer
10:             // Write the updated item to disk
11:             S.Update($item$)
12:             Stop the timer and save the elapsed time
13:         **end for**
14:     **end for**
15:     increase n
16: **end while**
17: Calculate the average execution time for each deserializer.

Algorithm 4 Benchmark for reading a single item

1: **while** $n < N$ **do**
2:     // N is the maximum number of items to deserialize
3:     **for** $r \leftarrow 0, repetitions$ **do**
4:         Serialize n items to disk
5:         $key \leftarrow$ RandomExistingKey
6:         **for all** $D \in$ Deserializers **do**
7:             Start a timer
8:             // Read one item from disk
9:             D.Lookup(key)
10:             Stop the timer and save the elapsed time
11:         **end for**
12:     **end for**
13:     increase n
14: **end while**
15: Calculate the average execution time for each deserializer.

7.3 Benchmarking Results

In this section, the results of the benchmarks are presented. The results presented here are from running the benchmark on a Samsung Galaxy S5 Android device, using the internal storage of the phone. The S5 has 2 GB of RAM, a Quad-core 2.5 GHz Krait 400 CPU and was running Android 4.4.3. The tests were run using a data set of size of 50 items up to 1000 items with an increment of 50. They were repeated 10 times for each storage format and data size. The average, min, max, and the first and last quartiles of the execution times were recorded and are presented in the plots below.

7.3.1 Batch Performance

Figure 7.1 shows a comparison of the batch reading performance of all storage formats as the number of items read increases. We can see from this comparison that reading from files in the MSM2 format is faster than all of the databases. The difference between the different methods is dependent of the size of the data being read. We can also see that the slowest storage format is Siaqodb.

Figure 7.2 shows a comparison of the batch writing performance between storage formats as the number of items written increase. The results are similar to the reading test and shows that the MSM2 format is the fastest, and Siaqodb the slowest.

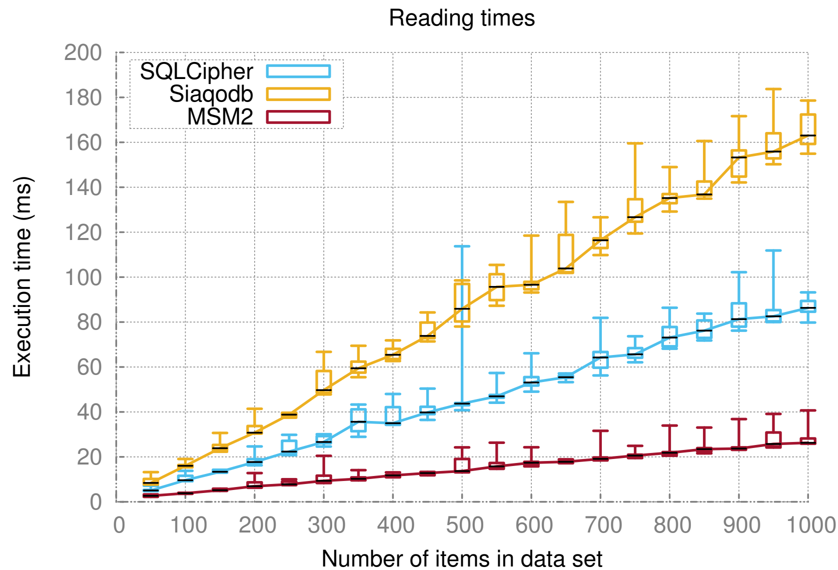

Figure 7.1: Comparison of batch read performance between the different storage formats. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values.

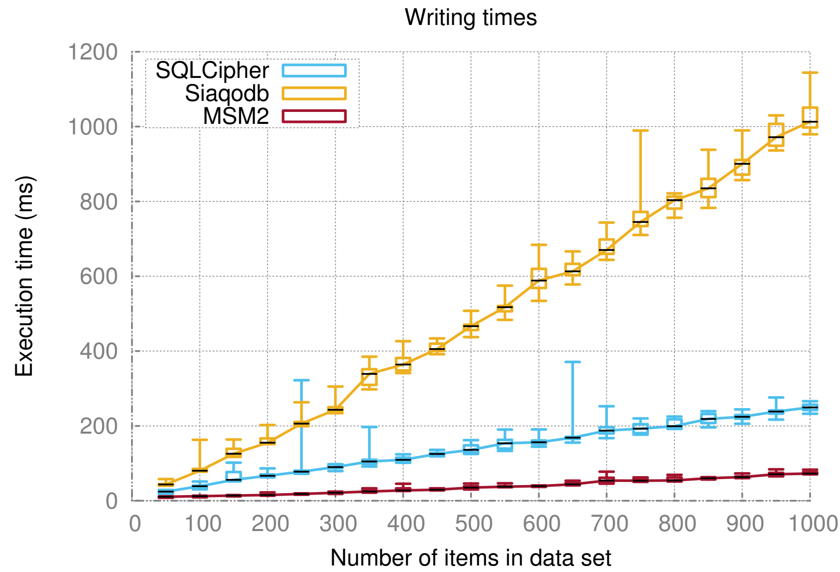

Figure 7.2: Comparison of batch write performance between the different storage formats. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values.

7.3.2 Random Access Performance

Figure 7.3 shows a comparison of the single item reading performance between storage formats as the total number of items in the data set increases. We can see from the results that the performance of the MSM2 format depends on the number of items in the data set. This is because it has to read all items in order to find the specified item among them. The databases are implemented in such a way that enables lookups in constant time, not dependent of the size of the data being searched. We can also see that Siaqodb is slightly slower than SQLCipher.

Figure 7.4 shows a comparison of the single item writing performance between storage formats as the total number of items in the data set increases. Figure 7.5 shows the same data but with a logarithmic y scale. We can see that the MSM2 format suffers from having to write all items even when only one item needs to be updated. From these results it seems that when the number of items reaches around 250 and above, the SQLCipher method starts outperforming the MSM2 method. Siaqodb seems to outperform the others by a large amount, but we cannot be sure that this is accurate.

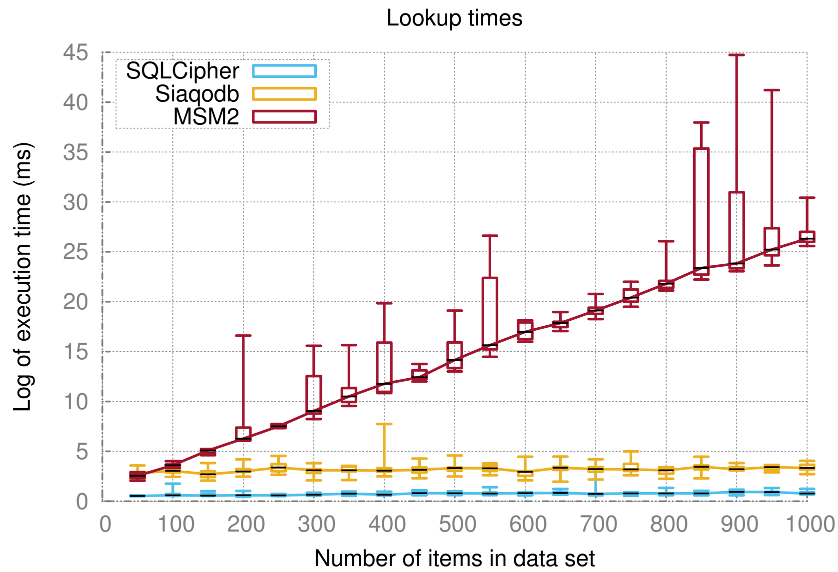

Figure 7.3: Comparison of single lookup performance between the different storage formats. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values.

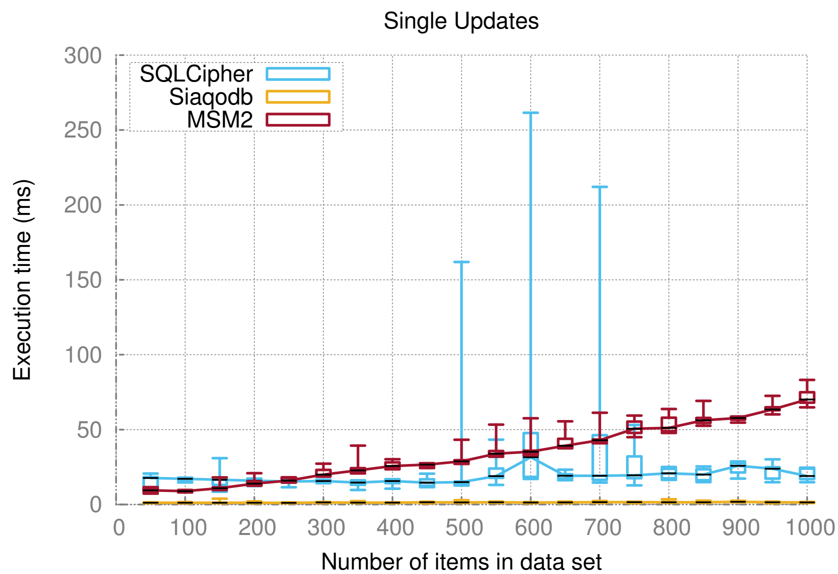

Figure 7.4: Comparison of single update performance between the different storage solutions. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values.

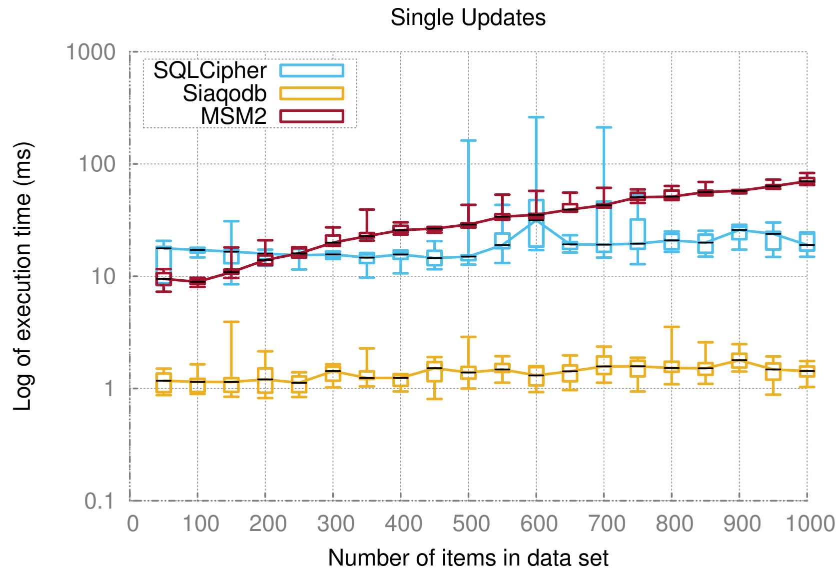

Figure 7.5: Comparison of single update performance between the different storage solutions with a logarithmic y axis. The edges of the boxes represent Q1 and Q3, and the whiskers represent the min and max values.

7.3.3 Disk Usage

Because space is limited on mobile devices, it can be of interest to see how much space the stored data consumes using the different storage formats. Figure 7.6 shows a comparison of the disk space used by different storage formats as the number of items increases. As we can see, Siaqodb grows considerably faster than the other formats. According to the Siaqodb documentation [52] the database does not claim free space automatically when objects are deleted or resized, but can be manually shrunk to reduce the size of the database on disk. However, in these benchmarks no objects are deleted and using the shrink utility method increased the size of the database instead of reducing it. The game itself uses a few hundred megabytes of storage in total when installed, so in relation to this the stored data makes a negligible difference.

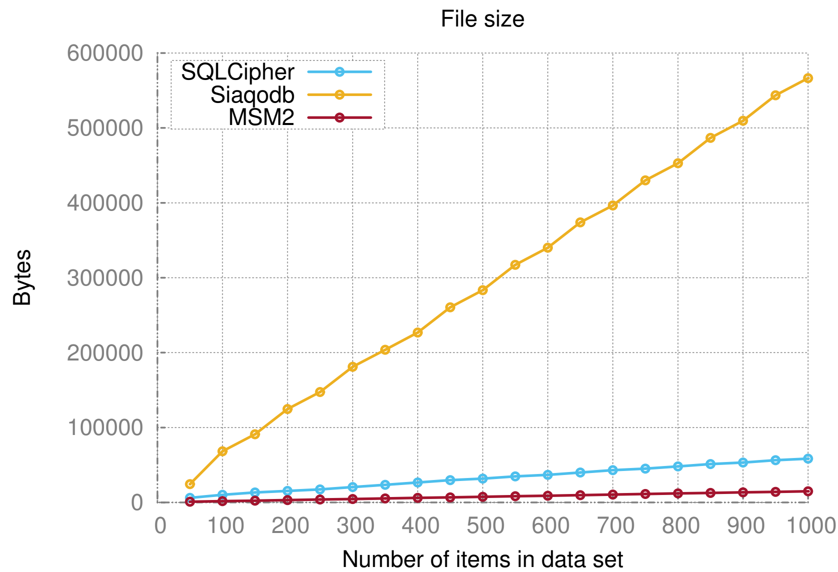

Figure 7.6: Comparison of disk usage of the different storage solutions.

7.3.4 Build Size

The build size of the application is an important measurement because of the limitations that Apple and Google have put on the distribution of applications. On Google Play, a single .apk file must be under 50 MB. [53] If the application needs more than this, the extra data can be put in so called expansion files. Most of the time, these expansion files are downloaded along with the main application file but in some cases the application needs to handle the downloading which increases the complexity of the application. Applications distributed on the App Store for iOS devices must be under 100 MB in order for them to be downloadable over a cellular network. [54] If application files are larger than 100 MB, users must be connected to a WiFi network in order to download the application.

On Android, MSM2 is built into a single .apk which is under 50 MB containing all the files necessary to play the game. In addition to this, the application can download optional high resolution textures after the game has been installed. On iOS, MSM2 is under 100

MB. Any external libraries added to the MSM2 project adds to the build size, so care must be taken not to go over the limits.

Table 7.1 shows the approximated uncompressed build size of each of the chosen storage formats. Application files on both iOS and Android platforms are compressed before being distributed. The numbers in the table are the sizes before compression and are thus only approximations of the final build size, which depends on how much the files are compressed. The assumption made here is that the final size added to the build is directly related to the size when uncompressed. This is a simplification that ultimately may or may not hold true, as the compression is probably also affected by all other files that are included in the final application file. Siaqodb seems to have the smallest footprint, attributed to the fact that it uses no external libraries. MSM2 uses external libraries for JSON and compression, which add to the total size. SQLCipher is the largest, but can possibly be reduced in size by compiling using different compiler options and encryption library.

Table 7.1: An approximation of the increase in build size due to using different storage back-ends.

Storage back-end	**Build size**
SQLCipher	2.48 MB
MSM2	1.23 MB
Siaqodb	0.35 MB

7.4 Feature Comparison

In this section, we compare a few select features of the databases that we examined.

7.4.1 Supported Platforms

Turborilla's MSM2 runs on iOS and Android devices, so for a database to be useful it must also support at least these platforms. In addition to these platform, it is interesting to know which other platform the databases support, should Turborilla want to add to their list of supported platforms. Unity currently supports 21 different platforms for mobile devices, desktops, web browsers, and consoles. In our case, the interesting platforms are mainly the mobile platform, as that is what MSM2 is developed for. However, it is also of interest to know which other platforms that are supported.

Table 7.2 shows which platforms each database supports. We can note that most of the storage solutions do not support the web platform. The reason for this is that web applications in Unity are restricted from accessing the filesystem directly, for security reasons. In our case, we are left with a few different options. One option is to use an in-memory database, such that nothing needs to be written to the local filesystem. In order for the data to be persistent, we would need to make use of an external database. Another option is to use the limited storage capabilities that are provided via Unity for the web platforms. DBreeze is the only database tested in this project that supports storing the database in memory, but this feature was not tested.

Unity currently supports two different web platforms, the Web Player and WebGL. The Web Player is a browser plugin that uses the Netscape Plug-in API (NPAPI). Plugins that use NPAPI have the same permissions as the user running the browser and put no restrictions on the usage of the machine's resources. However, the Web Player in Unity does

put restrictions on the application in terms of local storage. Unity has a simple storage system called PlayerPrefs which works on all platforms, but is restricted to 1 MB of storage on Web Player builds [55]. HTML5 supports local storage, which typically allows at least 5 MB of storage depending on the browser implementation [56] [57]. Another major drawback of using PlayerPrefs for MSM2 or a game with similar requirements is that it has no support for encryption.

Table 7.2: Comparison of platform support between databases.

	SQLCipher (SQLite)	**Siaqodb**	**MSM2**	**DBreeze**
iOS	✓	✓	✓	✓
Android	✓	✓	✓	✓
Desktops	✓	✓	✓	✓
Web				✓

7.4.2 Fault tolerance

Any storage solution used in MSM2 must handle that the application crashes. If this happens while a file is being written to, the file and its data can become corrupt or lost. Writing to an existing file is dangerous because of this. In MSM2, this is currently handled by writing to a temporary file first, and then replacing the existing file with the updated file. If the app crashes there is a large chance that at least one of the original or the updated file is intact.

SQLite is designed to be fault tolerant and claims to implement ACID transactions even in the face of crashes and power loss. [58] Siaqodb also claims to implement ACID transactions [50]. No tests have been made in this project to verify any of these claims.

7.4.3 API

Usability is an important aspect to consider when choosing a storage back-end. Measuring this can be difficult as it often becomes a matter of taste. In this section, we describe and discuss the APIs of the different database back-ends with usability in mind.

In MSM2, the storage solution uses the JSON format through the use of a dictionary-to-JSON conversion. The actual conversion to JSON is done automatically through the use of a third party library. However, the game's logical model does not necessarily use dictionaries to store the game data in memory. This means that any game object that needs to be persisted must first be represented as a dictionary, which is done on a per-category basis.

Siaqodb uses reflection, inspecting C# objects to decide how to store it. The different members of a class can also be decorated using attributes in order to achieve things like indexes or uniqueness constraints. In contrast to the current storage back-end of MSM2, developers do not need to explicitly convert game objects into any special format before storing an object.

SQLCipher is a library written in C, and is consequently accessed through a low level C-API. However, thanks to SQLite being so widely spread, there are multiple higher-level wrappers in a multitude of languages, including C#. Any SQLite wrapper with support for the Mono version used in Unity can be used as a wrapper for SQLCipher. One way to do this is to modify the wrapper library to include the added functions that control the encryption of the database. Another way that does not involve modifying the wrapper is to just replacing the SQLite library with a SQLCipher library and use SQLite #pragma

commands to set the encryption key. The benchmarks on SQLCipher were run using a wrapper called SQLite4Unity3d [59] which uses reflection similarly to Siaqodb.

From a usability aspect, it would seem that there are ways to improve on MSM2's storage solution. The major benefit of using reflection in the storage back-end seems to be that the usability increases. It removes the need for explicit conversion to a storage friendly format, as the reflection code can be written once, for all object types.

Chapter 8

A Synchronization Library for Mad Skill Motocross 2

In this chapter we investigate the implementation of a synchronization library for MSM2. We investigate the benefits of using it as well as some limitations and possible improvements and alternative implementations.

8.1 Goals

The main goal of implementing a synchronizing library for MSM2 is to decrease the amount of code needed to make changes in the storage model of MSM2 and thus increase development efficiency. While the library is developed to be a part of MSM2, the rewriting of MSM2 to include this library in the released products is not a part of this project because of the risks involved and the time constraints of the project. Instead, the library is considered a prototype for use in future projects and thus we discuss the potential benefits and problems discovered in the library, and how it might be improved.

To reach the goal, we analyze the current code base to find the flaws that are considered the most prominent by the developers of MSM2. The biggest issues found are in the way synchronization of state between the server and the clients is done. We identified the potential to use reflection in C# to automatically merge local and server data, based on simple rules decided at compile time.

8.2 Design

The library is designed around the JSON format, so that all objects that can be represented as JSON objects should be compatible with the library. To use the library, we need an object that has a valid JSON representation that we decorate with C# attributes. The object can then be merged with a matching JSON string using one method call.

8.2.1 Merging Operators

To be able to choose how an object is merged, C# attributes and reflection are used. The attributes can be used to specify how a field or property of a class should be merged, by choosing between a set of operators. Supported operators in the prototype are shown

in Table 8.1. These operators where chosen to be able to implement all the behaviour of the merging code in MSM2, only automatically. The local and remote operators are not used in MSM2 per se and might seem redundant, but they are used internally in the library when timestamps are used. If a field or a property has the newest operator, the timestamps of the remote and local objects are compared and the operator is then changed to local or remote based on which is the newest timestamp.

Table 8.1: A description of the supported operators in the synchronization library

Operator	**Description**
Min	The field or property that has the lowest value is chosen.
Max	The field or property that has the highest value is chosen.
Newest	The field or property that has the latest timestamp is chosen.
Local	The local value is chosen.
Remote	The remote value is chosen.

```
class MyObject {
    int anInteger;
    string aString;
    Dictionary<string, object> dict;
}
```

Listing 8.2.1: An example of a class that represents an object in the storage model.

```
MyObject myobj = new
   MyObject {
    anInteger = 1,
    aString = "two",
    dict = new
       Dictionary<string
       , object> {
        {"key1" : 1},
        {"key2" : "two"}
    }
};
```

a: Example code for creating a C# object.

```
{
   "anInteger":1,
   "aString":"two",
   "dict":{
      "key1":1,
      "key2":"two"
   }
}
```

b: Corresponding JSON representation.

Listing 8.2.2: An example C# object with the corresponding JSON representation

Figure 8.2.2 shows an example of a C# object with a corresponding JSON representation. Note that JSON objects within the root structure correspond to a dictionary object in C#. This expands the available data types slightly from being just the primitive data types.

8.2.2 Forward Compatibility

The format, or schema, of the stored data changes periodically when new features and content are added. These things are practically impossible to predict when the schema is first conceived. This is a common problem when using schemas, and is used to argue for the use of schemaless databases. In our case, the class declaration in C# can be viewed as the schema of the data model. The JSON representation must have the same schema to be used together with the C# object.

The schema changes, for example, when some new member is added to the C# class. This can cause problems if an older client gets JSON data that has an updated schema that contains data not part of the C# class. This is common as the application is constantly updated and it is impractical to try to force clients to be up to date. Because of this, the library needs to handle these situations by not causing the application to fail, and keep the unknown data intact so that it is not lost when uploading data to the server.

This is done by saving all the unknown JSON data in a dictionary, which is unaltered on the client and is uploaded with the rest of the data. Effectively, this means that any data from a newer schema is ignored by the client but kept intact. The library accomplishes this by forcing all storage models to inherit from an class called Mergeable that contains the necessary bits to achieve compatibility with future versions. Listing 8.2.3 shows a simplification of the C# declaration of this class. The merging of objects can only be done on objects of classes that inherit from Mergeable. The unknown data is automatically put inside the dictionary when merging, so the need to handle this explicitly has been removed.

```
class Mergeable
{
    public Dictionary<string, object> unknownData;
    public DateTime unknownDataTimestamp;
}
```

Listing 8.2.3: The class that all storage classes need to inherit from.

Chapter 9

Conclusion

In this chapter we draw some conclusions based on the results, and discuss restrictions, limitations and possible future work.

9.1 Performance Evaluation

The benchmarks have given us a way to measure the overhead of using two different databases compared to the file format used in MSM2. We see from the results that writing and reading a batch of items is much faster using the MSM2 format, especially as the number of items increases. When comparing reading or writing a single item between storage methods, we can see how the lack of support for this makes the MSM2 format slower than using a database. Compared to using SQLCipher, the point where MSM2 starts to perform worse seems to be around 250 items. This number could depend on many factors, like the device, the operating system, the storage medium etc., that it is not possible to state that it would be the same in the general case for all devices. However, we can assume that the general case does not diverge substantially from this result which at least gives us something to base our decisions on.

Figure 7.4 suggests that Siaqodb outperforms SQLCipher almost by a factor of 20. It is difficult to know how accurate this number is, because we cannot be sure of what actually happens when we call upon the database to update an object. It is possible that the new data is simply buffered and not physically written to disk until a later time which could explain why it is so fast. Since the source code of Siaqodb was not available we cannot know for sure. However, it is reasonable to assume that the performance of single writes and reads are not hugely affected by the total number of items in the database.

In the batch writing and reading tests the MSM2 file format is much faster. This is expected as the databases tested are not optimized for these kinds of operations. The writing tests show the greatest difference in performance which is interesting. The behavior that is tested is precisely what the MSM2 application currently uses which means that simply changing the storage format would be detrimental to the application's performance. In order for it to be beneficial to change the storage format, the behavior of the app would need to change to not rewrite data that has not changed.

The amount of data stored in MSM2 is not estimated to grow to very large numbers for any user and the data is already split into multiple files. The largest growing data category adds the equivalent of approximately 30 to 40 items every week the user plays in a tournament race. MSM2 suffers from the performance degradation measured in the

single update performance tests (Figure 7.4), but since size of the data grows so slowly it is unlikely that it creates a noticeable effect. A simpler fix to this problem is to delete old records, which happens if the app is uninstalled and then reinstalled.

In conclusion, the benchmarks give us a reference that we can use to reason about when the chosen databases perform faster than storing game data in the MSM2 file format. In practice, the fastest solution is not always the best suited solution which is why we need to look at the other factors. Judging by performance alone there are benefits to using a database, if the data set is large enough. However, it requires some additional refactoring in order to obtain these performance benefits.

9.2 Synchronization

There was less time than planned to work with the synchronization problems, but a working prototype was built that decreases the complexity of merging data with the server. The key to making the merging simpler is in the use of reflection which allows the class declaration to be used as a schema, and attributes which enables customization of the merging behaviour.

9.3 Limitations

This section describes the limitations of the methods and results of the thesis.

9.3.1 Benchmarks

The benchmarks were designed to be as fair as possible, but achieving this is hard because of the many factors that come into play. All of the different formats performed a similar task of persisting a data set to permanent storage. The times recorded were on the same set of data to increase fairness. In order to get results that were relevant to MSM2, the data set was similar in size and type to what is being used in the application. However, the different storage formats have different performance depending on many factors, such as the structure in the database (in the case of SQLite) and the format of the data. Very little time was put into optimizing each storage format, which could have potentially large effects on performance. The results are clear for this particular use case, but we must be careful with what conclusions are made from it. We can never be fully certain that the results would be the same in a different setting without actually running additional tests.

9.3.2 Synchronization

There are a few limitations to the synchronization library that could make it less useful for certain applications. The library is written for the MSM2 application and storage model specifically, and is therefore only useful for an application that has the same or a similar storage model. JSON is used in many applications as an interchange format, so it is possible that this library or something similar can be useful, but there are no guarantees.

One limitation is that the Mergeable class must be inherited for the library to work. This forces developers to create a storage class for their object which may or may not be a small task depending on the size of the project. Arguably, having separate classes for your storage model is a good thing, but there might exist situations where this is not wanted or even impossible.

Another limitation is in which data types that are supported. Currently, all primitive C# data types are supported as well as `String`, `DateTime` and dictionaries with strings as keys. Additionally, any Mergeable object can be nested inside another object. This covers the most common use cases in MSM2, although it has not been tested as a complete replacement of the older solution. It is possible to add support for more data types by modifying the library source code. This must be done explicitly for each new data type, unless the library is rewritten to allow all data types implicitly. Perhaps the largest missing piece is support for JSON arrays, which has not been added or tested.

The library needs more testing in order to guarantee that it handles errors gracefully. There are many things that can go wrong, and very little testing is done to ensure graceful error handling.

9.4 Future work

No attempts have been made to optimize the library for speed or memory footprint. The library uses reflection which can be slow in certain situations and it is possible that there are ways to improve the performance of the library by rewriting it. The library expects a dictionary to merge with a C# object. This dictionary is assumed to have been converted from JSON, and the code for doing this is already in MSM2. However, this is not necessarily a good assumption and the library could include a JSON-to-dictionary converter or simply use the JSON data directly. Skipping the dictionary could possibly improve performance. However, dealing with a string containing JSON data is a lot more complicated than a dictionary with string keys, so the added complexity is probably not worth the performance gain.

As mentioned in the limitations section, the synchronization library does not support JSON arrays which could be useful. A C# List is possibly the closest representation that would be useful as a corresponding data type.

The current implementation uses timestamps to decide the newest value. This is a simple approach that should work most of the time. The timestamps are stored in ISO-8601 formatting as UTC time which enables lexicographical ordering and should minimize the risk of confusion between timezones. However, timestamps may not be a good solution because of the difficulties in synchronizing the server and client clocks. Vector clocks or similar techniques could be used which would probably decrease the amount of errors because of timing discrepancies. One clock could be added per device, as the amount would only increase when a user plays the game on a new device with the same account, which is not likely to increase massively in size within the lifetime of the game.

Another possible route is to implement conflict-free replicated data types in the library to investigate their usefulness in this kind of usage scenario. It would be interesting to know how hard it would be to implement and what benefits it would have, if any. It could potentially be possible to write a library that is useful in a wider context.

Bibliography

[1] Wikipedia, “Relational database — wikipedia, the free encyclopedia,” 2015, (visited 2015-03-03). [Online]. Available: http://en.wikipedia.org/wiki/Relational_database

[2] Turborilla AB. Mad skills motocross 2. [Online]. Available: http://www.madskillsmx.com/media-kit

[3] ——. Turborilla. [Online]. Available: http://www.turborilla.com/

[4] Apple, Inc. Apple - ios 8. [Online]. Available: http://www.apple.com/ios/

[5] Google, Inc. Android. [Online]. Available: https://www.android.com/

[6] Turborilla AB. Mad skills motocross 2. [Online]. Available: http://www.madskillsmx.com/

[7] Google, “What is google app engine?” 2015, (visited 2015-03-06). [Online]. Available: https://cloud.google.com/appengine/docs/whatisgoogleappengine

[8] json.org. Json. [Online]. Available: http://www.json.org/

[9] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *ACM SIGACT News*, vol. 33, pp. 51–59, 2002.

[10] Scrum.org. Scrum.org | the home of scrum > home. [Online]. Available: https://www.scrum.org/

[11] db engines.com, “Db-engines ranking,” 2015, (visited 2015-03-03). [Online]. Available: http://db-engines.com/en/ranking

[12] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, pp. 377–387, 1970.

[13] Microsoft Corporation. Sql server 2014 | microsoft. [Online]. Available: http://www.microsoft.com/en-us/server-cloud/products/sql-server/

[14] Oracle Corporation. Mysql :: The world’s most popular open source database. [Online]. Available: https://www.mysql.com/

[15] The PostgreSQL Global Development Group. Postgresql: The world’s most advanced open source database. [Online]. Available: http://www.postgresql.org/

[16] iso.org. Iso/iec 9075-1:2011 - information technology – database languages – sql – part 1: Framework (sql/framework). [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681

[17] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.*

[18] Wikipedia, “Nosql — wikipedia, the free encyclopedia,” 2015, (visited 2015-03-03). [Online]. Available: http://en.wikipedia.org/w/index.php?title=NoSQL&oldid=647474824

[19] Last.fm, “Nosql meetup,” 2015, (visited 2015-03-03). [Online]. Available: http://nosql.eventbrite.com

[20] The Apache Software Foundation. The apache cassandra project. [Online]. Available: http://cassandra.apache.org/

[21] Basho Technologies. Riak. [Online]. Available: http://basho.com/riak/

[22] MongoDB. Mongodb. [Online]. Available: https://www.mongodb.org/

[23] Neo Technology, Inc. Neo4j, the world’s leading graph database. [Online]. Available: http://neo4j.com/

[24] E. Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Boston: Addison-Wesley, 2003.

[25] A. G. DeCandia et al., “Dynamo: Amazon’s highly available key-value store,” *SOSP ’07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pp. 205–220, 2007.

[26] I. F. Chang et al., Google, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, 2008.

[27] ——, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, pp. 35–40, 2010.

[28] T. K. G. B. G. Coulouris, J. Dollimore, *Distributed Systems: Concepts and Design, 5th Edition.* Boston: Addison-Wesley, 2012.

[29] A. Rotem-Gal-Oz, “Fallacies of distributed computing explained,” 2006. [Online]. Available: http://www.rgoarchitects.com/Files/fallacies.pdf

[30] B. Alpern and F. B. Schneider, “Recognizing safety and liveness,” Cornell University Ithaca, NY, USA, Tech. Rep., 1986.

[31] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *ACM Queue*, vol. 11, 2013.

[32] E. Brewer, “CAP Twelve Years Later: How the ’Rules’ Have Changed,” (visited 2015-02-24). [Online]. Available: http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

[33] A. Fox and E. A. Brewer, *HOTOS ’99 Proceedings of the The Seventh Workshop on Hot Topics in Operating ystems*, 1999.

[34] M. Burgess, “Deconstructing the ‘cap theorem’ for cm and devops,” (visited 2015-02-24). [Online]. Available: http://markburgess.org/blog_cap.html

[35] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, pp. 37–42, 2012.

[36] J. Gray, "The transaction concept: Virtues and limitations," *VLDB '81 Proceedings of the seventh international conference on Very Large Data Bases*, vol. 7, pp. 144–154, 1981.

[37] D. Pritchett, "Base: An acid alternative," *ACM Queue*, vol. 11, 2008.

[38] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 776–787, Apr. 2012. [Online]. Available: http://dx.doi.org/10.14778/2212351.2212359

[39] Google, "Google cloud platform," 2015, (visited 2015-05-26). [Online]. Available: https://cloud.google.com/docs/

[40] ——, "Mastering the datastore - google app engine," 2015, (visited 2015-03-06). [Online]. Available: https://cloud.google.com/appengine/articles/datastore/overview

[41] The Apache Software Foundation, "Hbase - apache hbase home," 2015, (visited 2015-03-09). [Online]. Available: http://hbase.apache.org/

[42] Google, "How entities and indexes are stored - google app engine," 2015, (visited 2015-03-09). [Online]. Available: https://cloud.google.com/appengine/articles/storage_breakdown

[43] ——. (2015) Java datastore api. [Online]. Available: https://cloud.google.com/appengine/docs/java/datastore/

[44] ——, "Using the master/slave datastore," 2015, (visited 2015-03-09). [Online]. Available: https://cloud.google.com/appengine/docs/java/datastore/usingmasterslave

[45] I. T. Chandra et al., Google, "Paxos made live – an engineering perspective," *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*, 2007.

[46] U. Technologies, "Unity - game engine, tools and multiplatform," 2015, (visited 2015-08-26). [Online]. Available: http://unity3d.com/unity

[47] Mono Project. Home | mono. [Online]. Available: http://www.mono-project.com/

[48] sqlite.org, "About sqlite," 2015, (visited 2015-04-20). [Online]. Available: http://sqlite.org/about.html

[49] Z. LLC, "Sqlcipher," 2015, (visited 2015-04-20). [Online]. Available: https://www.zetetic.net/sqlcipher/

[50] D. SRL, "Siaqodb," 2015, (visited 2015-04-20). [Online]. Available: https://www.siaqodb.com/

[51] dbreeze.tiesky.com, "Dbreeze - c# acid nosql embedded object database management system." 2015, (visited 2015-04-27). [Online]. Available: https://www.dbreeze.codeplex.com/

[52] D. SRL, "Shrink and repair - siaqodb," 2015, (visited 2015-04-20). [Online]. Available: http://siaqodb.com/shrink-and-repair/

[53] Google, "Apk expansion files," 2015, (visited 2015-04-24). [Online]. Available: http://developer.android.com/google/play/expansion-files.html

[54] Apple, "Now accepting larger binaries," 2015, (visited 2015-04-24). [Online]. Available: https://developer.apple.com/news/?id=02122015a

[55] U. Technologies, "Unity - scripting api: Playerprefs," 2015, (visited 2015-04-20). [Online]. Available: http://docs.unity3d.com/ScriptReference/PlayerPrefs.html

[56] S. González, "Web storage support test," 2015, (visited 2015-04-20). [Online]. Available: http://dev-test.nemikor.com/web-storage/support-test/

[57] W3C, "Web storage - editor's draft 14 may 2014," 2015, (visited 2015-04-20). [Online]. Available: http://dev.w3.org/html5/webstorage/#disk-space

[58] sqlite.org, "Sqlite is transactional," 2015, (visited 2015-04-20). [Online]. Available: http://sqlite.org/transactional.html

[59] Roberto Huertas. codecoding/sqlite4unity3d github. [Online]. Available: https://github.com/codecoding/SQLite4Unity3d

[60] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems*. New York, NY, USA: McGraw-Hill, Inc., 1994.