NTNU
Norwegian University of
Science and Technology

Collecting massive amounts of location data in a NoSQL database

Identifying best practices for high write throughput

Kristian Midtgård

Master of Science in Computer Science
Submission date: July 2017
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

The amount of internet-connected devices is rapidly expanding. Embedded with various sensors, these devices are generating ever increasing amounts of data, causing a shift towards more write-intensive workloads for the underlying database systems. In order to see the extent that existing database systems are able to ingest data at scale, this thesis considers a world where *everything* is connected and continuously sending its location to a central database. From this world, three applications are defined in order to obtain a set of requirements for the database. The applications are designed so that the amount of data points generated exceeds the rate of giants like Facebook and Google. At peak load, around 600 hundred million write requests to the database are generated every second.

This thesis examines the NoSQL landscape, a paradigm with focus on high performance, availability and scalability for databases, in an attempt to identify the best practices for achieving high write throughput. Several different types of NoSQL databases are widely used in production today. Some of the most promising ones will be examined in more detail and evaluated against the requirements for the applications. In addition to these general purpose database, a specific type of database intended for time series data will also be considered.

The most notable similarity between the top NoSQL databases is the use of the LSM-tree data structure. LSM-trees are able to obtain high write throughput by transforming small random writes into larger sequential writes, minimizing the need for expensive disk operations. The performance of these databases grow beyond the capacity of a single machine by partitioning data by rows and automatically distribute the partitions among nodes in a dynamic cluster.

Although most NoSQL databases promise linear scalability, most will encounter practical limitations due to the use of a master server for central coordination. Fully decentralized and eventually consistency NoSQL databases like Cassandra are the candidates most likely able to scale to accommodate 600 million writes per second. Time series databases are shown to provide a much higher write throughput per node than general purpose databases. However, the lack of query functionality and indexing on other attributes than time, makes time series databases less suitable if the applications require a significant portion of reads.

Sammendrag

Antallet internett-tilkoblede enheter vokser hurtig. Sensorer innbygd i disse enhetene genererer stadig økende mengder med data, noe som forårsaker et skifte mot mer skrivebelastede arbeidsmengder for de underliggende databasesystemene. For å se i hvilken grad eksisterende databasesystemer er i stand til å innta massive mengder data, tar denne oppgaven for seg en verden hvor *alle* ting er tilkoblet en sentral database som de kontinuerlig sender plasseringen sin til. Med utgangspunkt i denne verdenen defineres tre applikasjoner for å spesifisere et sett med krav til databasen. Applikasjonene er utformet med et formål om at mengden datapunkter som genereres skal overstige mengdene hos giganter som Facebook og Google. Under høyeste belastning genereres det rundt 600 millioner skriveforespørsler til databasen hvert sekund

Denne masteroppgaven undersøker NoSQL-landskapet, et paradigme med fokus på høy ytelse, tilgjengelighet og skalerbarhet for databaser, i et forsøk på å identifisere beste praksis for å oppnå høy gjennomstrømning av skrivinger. Flere fulike typer NoSQL databaser er mye brukt brukt i produksjon. Noen av de mest lovende vil bli sett nærmere på og evaluert mot kravene til applikasjonene. I tillegg til disse databasene for generelle formål, vil også en spesifikk type database beregnet for tidsseriedata bli vurdert.

Den mest bemerkelsesverdige likheten mellom de største NoSQL databasene er bruken av LSM-trær for organisering av data. LSM-trær er i stand til å oppnå høy skriveytelse ved å transformere små tilfeldige skrivinger til større sekvensielle skrivinger, noe som minimerer behovet for dyre diskoperasjoner. Disse databasene er vanligvis designet til å øke kapasiteten utover en enkelt maskin ved å partisjonere data horisontalt og automatisk distribuere partisjonene mellom noder i en dynamisk klynge av maskiner.

Selv om de fleste NoSQL databaser lover lineær skalerbarhet, vil de fleste se praktiske begrensninger som følge av en eller annen form for sentral koordinering av noder. Fullstendig desentraliserte og eventuell konsistente NoSQL databaser som Cassandra er de kandidatene som mest sannsynlig er i stand til å skaleres til å kunne håndtere 600 millioner skrivinger hvert sekund. Tidsseriedatabaser har vist en mye høyere gjennomstrømning av skrivinger per node enn databaser for generelle formål. Derimot er spørrefunksjonalitet og indeksering på andre data enn tid en stor mangel i tidsseriedatabser, noe som gjør dem dårlig egnet hvis applikasjonene krever en betydelig andel lesninger.

Preface

This Master's Thesis is the final delivery in the Computer Science program at Norwegian University of Science and Technology, Trondheim. The work has been conducted at the Department of Computer and Information Science at NTNU during the spring of 2017 and under supervision of Svein Erik Bratsberg.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

AWS Amazon Web Services

2PC Two-Phase Commit Protocol

ACID Atomicity, Consistency, Isolation, Durability

BASE Basically Available, Soft state, Eventual consistency

CAP Consistency, Availability, Partition tolerance

CPU Central Processing Unit

DBMS Database Management System

FT Fractal Tree

HDD Hard Disk Drive

HDFS Hadoop Distributed File System

GPS Global Positioning System

I/O Input/Output

IoT Internet of Things

IPS Indoor Positioning System

JSON JavaScript Object Notation

LMDB Lightning Memory-Mapped Database

LSM-tree Log-Structured Merge-tree

MVCC Multi-version Concurrency Control

OS	Operating System
OTP	Open Telecom Platform
RAM	Random Access Memory
RDBMS	Relational Database Management System
SSD	Solid State Drive
SQL	Structured Query Language
TSDB	Time Series Database
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Background and Motivation

Big Data and Internet of Things are two increasingly important concepts, which together create opportunities to generate value from analyzing data. The vast amount of Internet-connected smartphones, buildings, vehicles and other everyday objects are generating data at a rate which rapidly outpaces an organizations ability to analyze it. Common workloads had around 10% writes 30 years ago and 33% was a realistic write load 10 years ago [22]. Today, it is not uncommon for businesses to have a majority of writes over reads. This increase is putting higher demands on the write performance of databases and the space required to persistently store all the data.

In recent years, many businesses have moved away from traditional relational databases and instead moved into the NoSQL landscape to accommodate the increasing degree of writes and database scale for their Big Data needs. The NoSQL philosophy favors high performance, loosely structured data, simple interface and efficient horizontally scalable architectures over powerful transactions, ACID guarantees and rigid schemas provided by relational databases. These properties are desirable for managing applications where the structure of data often changes and easy scaling is necessary to meet the demands of increasingly bigger workloads.

Additionally, databases are slowly moving away from read optimized, hard disk drive (HDD) based data structures such as the B-tree and opting for more write efficient alternatives instead. The LSM-tree is one such data structure. Instead of overwriting existing records on updates, an LSM-tree batches updates in memory before flushing the records to disk as immutable files. This approach essentially transforms many small random writes into one sequential write. However, LSM-trees suffer in read performance as records are not stored in sorted order.

Determining the highest theoretical write throughput of a database is a difficult task that depends on many factors beside the implementation of storage structure. The most important aspect for high performance in general is the ability to scale the database and distribute data across multiple servers, or nodes. Besides scalability and storage engine, the write performance is affected by factors such as workload pattern (random vs sequential writes), efficient use of memory and compression. In addition, when designing a database for high write performance, the need to serve reads must be taken into consideration.

The main motivation behind this thesis is to evaluate whether there exists a database system existing NoSQL technologies are capable of ingesting the data from a vast amount of objects in a world where almost everything continuously generate data. Location data is used as example in the proposed theoretical applications because it is an universal property that can quickly be extracted from things using readily available technologies.

1.2 Research Goals

Based on the motivation, the following goals were defined for this thesis:

- **Goal 1:** Review best practices for high write throughput in database systems.
- **Goal 2:** Define a set of theoretical applications that generate massive amounts of data. The applications should be based on the Internet of Things and capture the location of every single thing within a particular domain. Then, from the applications, determine a set of database requirements primarily focusing on write throughput.
- **Goal 3:** Examine existing NoSQL database systems and evaluate which technologies are better suited to meet the demands of the applications.

1.3 Limitations

The write performance of a database is affected by many factors including storage structure, indexing, memory caching, storage medium, hardware capacity, database size and scaling solution. Given the complexity of all these combinations and the infrastructure required to handle such massive amount of data, it is not feasible to perform real tests. Instead, a small selection of factors influencing the performance are investigated and used to identify the best suitable database for the given application based on theoretical information and calculations. Furthermore, several assumptions are made regarding the quantity and behaviour of certain objects in the world as this information is either nonexistent or too complex to estimate.

1.4 Report Outline

The rest of this thesis is divided into the following five chapters:

- **Chapter 2: Background.** Reviews topics relevant for database systems and applications generating and storing massive amounts of data. The main focus of this chapter is to discuss best practices for achieving high write performance in a database. This includes topics such as storage structures and scaling solutions used in database implementations.

- **Chapter 3: Applications.** Presents three example applications for generating huge amounts of location data and the requirements imposed on the database in terms of write throughput and storage capacity.
- **Chapter 4: Database Candidates.** Presents several candidate NoSQL databases that might meet the requirements.
- **Chapter 5: Discussion and Evaluation.** Evaluation of the database candidates with respect to write performance and important features for the proposed applications.
- **Chapter 6: Conclusion.** Final conclusions regarding the ideal database solution for the applications.

Chapter 2

Background

This chapter opens with a short introduction to the Internet of Things (IoT) and good practices for representing location data. It continues with a look at the NoSQL landscape and its importance for IoT applications and Big Data. Following that, important NoSQL concepts and techniques are evaluated in an attempt to identify best practices for achieving high write throughput in a database system.

2.1 Internet of Things

The Internet of Things (IoT) refers to the integration of everyday objects into the Internet, a world where all physical assets and devices are connected and share information. Another definition of IoT is simply the point in time when the amount of things connected to the Internet surpassed the world population, which happened around the explosive growth of smartphones and tablets in 2010 [20]. As the Internet is becoming more widely available and the cost of adding Wi-Fi capabilities to devices are decreasing, the amount of internet connected devices are predicted to reach 40-50 billion in 2020 [20, 31]. With more and more devices being embedded with various sensors and network connectivity, IoT improves the ability to gather and analyze data, which can be turned into information, and ultimately generate value. From coffee makers that brew coffee when your alarm clocks goes off, to broader applications like smart cities that can help reduce waste and save energy, the opportunities with IoT are virtually endless.

2.1.1 Location of Things

Location is a key element in many IoT applications. The location of things is an emerging subcategory of IoT that encompasses devices that sense and share their geographic location. Location-sensing systems date back more than 20 years with the introduction of the Global Positioning System (GPS). GPS was originally developed for the U.S. military and was made available to the public in 2000. However, it took another decade before GPS receivers became affordable and small enough to find their ways into smartphones, wearables and other everyday objects [29]. The breakthrough of location-based services not only allows people to locate things, but it also opens a world where things know where other things are located. Combining location data with timestamps lets IoT applications know when and where something is, which can be used by organizations to optimize various processes such as traffic routing, tracking equipment and reducing theft and loss [28].

Relying on GPS alone has a few drawbacks, most importantly that there are many places on earth where it can't reach accurately, such as inside buildings. That's where indoor positioning systems (IPS) become useful. IPSs can be based on different technologies such as radio waves or magnetic fields, but the most notable IPS is the Wi-Fi based one developed by Skyhook, which is used by customers such as Apple and Samsung [28]. Using a combination of GPS and IPS allows IoT applications to cover more ground and enables the possibility to use the right signal at the right time for a more accurate location of a device.

2.1.2 Database Requirements for IoT

The benefits of IoT relies heavily on the massive amounts of data that applications create. This presents a set of challenges for the database management system used by the application, most importantly regarding scalability and the ability to rapidly ingest data. Data sent from devices are often of great variety, so the ability to handle rapidly changing data is also beneficial for storing IoT data [31].

There are many factors that influence how well a database is able to handle these requirements. Scalability is limited primarily by the sharding and replication strategies used to partition and distribute data in a cluster, the amount of data created and the throughput demanded by the application. The total write throughput of a database system depends on many factors such as scalability, consistency requirements, richness of data model, use of indexes, underlying technologies (e.g. programming language and storage engine) and hardware. It is important that the database powering an IoT application is designed with these factors in mind, in order to satisfy the write requirements without sacrificing important query capability.

2.2 Representing GPS Location Data

An important aspect when dealing with GPS location data is how to represent the data efficiently in order to reduce network traffic and storage requirements. Latitude and longitude coordinates are typically represented using decimal degrees which bound the values by $\pm 90°$ and $\pm 180°$, respectively. The Earth has a circumference of 40,075 km at the equator resulting in approximately 111,300 meters per degree longitude. Moving away from the equator in north/south direction decreases the distance per degree longitude, thus increasing the longitude precision closer to the poles. Table 2.1 lists the longitude precision at different degrees latitude given a certain number of decimal places used to represent the longitude.

Decimal places	**Equator**	**23°N/S**	**45°N/S**	**67°N/S**
0	111.32 km	102.47 km	78.71 km	43.496 km
1	11.132 km	10.247 km	7.871 km	4.3496 km
2	1.1132 km	1.0247 km	787.1 m	434.96 m
3	111.32 m	102.47 m	78.71 m	43.496 m
4	11.32 m	10.247 m	7.871 m	4.3496 m
5	1.1132 m	1.0247 m	787.1 mm	434.96 mm
6	111.32 mm	102.47 mm	78.71 mm	43.496 mm
7	11.132 mm	10.247 mm	7.871 mm	4.3496 mm
8	1.1132 mm	1.0247 mm	787.1 µm	434.96 µm

Table 2.1: Decimal places required for a particular longitude precision.

It is desirable to minimize the number of decimal places used to represent a coordinate while maintaining an acceptable precision for the application using the data. For example, an application that needs to identify a city might accept a deviation of a couple of kilometers, requiring two decimal places. This range of values can be represented as a 16-bit signed integer. On the other hand, if the precision must be down to a meter, five decimal places are required and is better represented as a single-precision floating point using four bytes. For even more precise measurements, a 32-bit signed integer or even a double-precision floating-point can be used.

Representing Location Using Offets

Instead of storing the complete latitude and longitude on every update, devices could send their relative location to the first one in consecutive messages. This value could then be stored as is or transformed into the complete coordinate before being written to the database. Depending on how fast the device is traveling and the desired precision, the size of an offset could be as little as a single byte. For instance, an animal running through a forest might move a few hundred meters every minute before its location is updated. If one creates a grid around the moving animal of one square kilometer, the animal could be tracked with a ~5 meter precision by representing the offset as a 16-bit integer. Alternatively, the device could send the full coordinates periodically. This reduction in message size comes at a cost of increased battery usage by the GPS equipped device and is therefore a very sensitive trade-off option.

2.3 NoSQL

Core concepts of NoSQL have existed in databases since the early 60s, but the term itself was coined almost 40 years later as a response to the emerging trend of database systems not following the traditional RDBMS approach. The term NoSQL can be somewhat misleading when interpreted as "No SQL". Another translation of the term is ”Not Only SQL”, but a more befitting name would be ”non relational”. While traditional relational databases follow a model where data is stored in multiple tables connected through relations, NoSQL databases provide alternative ways to store and access data. While NoSQL systems differ vastly and there is no clear definition, Cattell [12] identifies a set of features generally found in databases labelled as NoSQL:

- The ability to scale horizontally.
- Replication of data over multiple partitions.
- Simpler interface than SQL.
- A concurrency model that is weaker than ACID transactions.
- Efficient use of distributed indexes and in-memory data storage.
- Possibility to add attributes dynamically.

Most importantly is the ability to scale horizontally. In contrast to vertical scaling where additional capacity is obtained by upgrading to better hardware (CPU, RAM) on an existing machine, horizontal scaling means adding additional machines to the system. Horizontal scalability is an attractive feature because it allows the system to scale almost infinitely, in theory. It is also cheap because the capacity can be expanded using commodity hardware. In NoSQL databases, data is typically split by rows across many nodes and replicated over two or more partitions to ensure availability.

Although most NoSQL systems follow the above formula, there are several factors that distinguish one from another. This includes storage architecture, partitioning and replication scheme, and consistency model. NoSQL databases also fall into one of several categories based on the data model.

2.3.1 Data Models

A data model describes the organization of elements in a piece of data and how they related to one another. NoSQL databases adopt widely varying data models which usually fall into one of the following categories: key-value, document, columnar or graph. Some stores implement hybrid or customized versions of these models to better suit certain specialized use cases.

Key-Value Stores

Key-value stores are the simplest form of NoSQL databases where each item is stored together with a value. Values are typically stored as strings, but most modern key-value stores support additional structures like lists and sets enabled by the integration of serialization frameworks such as Thrift. Examples of key-value stores are Redis [39], Berkeley DB [37] and Riak [41].

Document Databases

Document databases pair keys with more complex data structures than key-value stores. These structures are referred to as documents and are typically stored in a format such as XML or JSON. A document has a flexible schema and can contain many different key-value pairs or even nested documents. Two notable document databases are MongoDB [32] and CouchDB [15].

Column-oriented Databases

Column-oriented databases, also called wide column stores or extensible record stores, have a data model that consist of tables with rows and columns similar to relational databases. The main difference is that column-oriented databases are optimized for queries over large datasets by grouping related columns together into column families and storing columns of data together rather than rows. Examples of well-known column-oriented databases include Cassandra [9] and HBase [23].

Graph Databases

Graph databases are useful for storing information about networks where data is stored as nodes and edges in a graph structure. It can for example be used to represent social connections where nodes are people and edges are a two way friendship. Graph databases are excellent at finding patterns and determining relationships between nodes. The most notable graph database is Neo4j [35]. Due to the specific use case and the fact that they are difficult to scale efficiently, graph databases will not be considered here.

2.3.2 CAP Theorem

First introduced in 2000 by Eric Brewer [7], the CAP theorem defines a fundamental property of distributed storage systems. It states that any shared-data system can guarantee at most two out of three following properties:

- **Consistency (C):** The system remains in a consistent state after the execution of an operation. In practice this means that all reads are guaranteed to return the most recent write.
- **Availability (A):** A non-failing node will always respond to the client in a reasonable amount of time.
- **Partition tolerance (P):** The system will continue to operate in the presence of network partitions. Only a complete network failure can cause the system to stop functioning.

The easiest way to explain why the CAP theorem holds true is to imagine two nodes separated by a partition. Updating the state of one node causes the nodes to become inconsistent, thus forfeiting C. In order to maintain consistency, one node must act unavailable, thus giving up A. Preserving both C and A is only possible when nodes communicate, thus forfeiting P. It is important to note that a system can have all three properties at the same time and only under certain conditions has to forfeit one. There is little reason to give up C or A when the system is not partitioned and because partitions are rare, both C and A can be perfect most of the time [6].

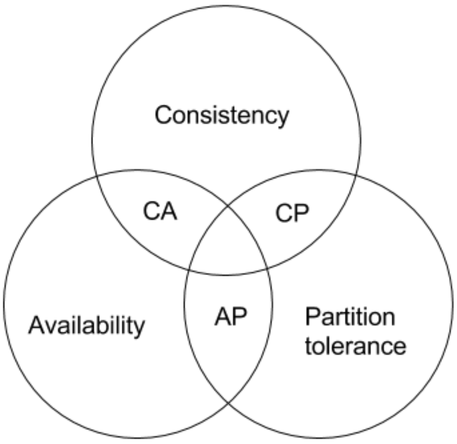


Figure 2.1: Illustration of the CAP theorem

The consequence of CAP is that a distributed database has to choose between three options, giving up either consistency, availability or partition tolerance under certain failure scenarios. A distributed system has to tolerate network partitions given that networks are never completely reliable. This means that for NoSQL systems it becomes a choice between forfeiting availability or consistency. Many NoSQL databases implement a tunable consistency model, meaning the level of consistency can be chosen for each individual operation.

2.3.3 Consistency Model

The ACID properties are fundamental principles in how transactions behave in traditional relational databases. These properties ensure durability of data and that transactions always leave the database in a consistent state. For many use cases the requirements for immediate consistency are not as strict, and properties like availability and scalability are more valuable. BASE is a consistency model that lies on the other side of the consistency-availability spectrum, and was created to capture the emerging design approach focusing on high availability [6]. The acronym stands for Basically Available, Soft state, Eventually consistent. These properties offer the loose guarantee that the database is available most of the time, replicas do not have to be consistent all the time, but will eventually be consistent. Although ACID guarantees are favored by some NoSQL systems, BASE is an essential trade-off for database systems favoring high availability. Table 2.2 shows the key differences between the two approaches.

The BASE trade-off is an important consequence of the CAP theorem. To see this, imagine choosing availability and partitions tolerance from CAP. Consistency can not be guaranteed, so BASE model can be adopted in order to maintain high availability and provide eventual consistency.

ACID	**BASE**
Strong consistency	Weak consistency
Isolation	Availability first
Focus on ”commit”	Best effort
Nested transactions	Approximate answers
Pessimistic	Optimistic
Difficult evolution (e.g. schema)	Easier evolution

Table 2.2: Key differences between ACID and BASE. Source: [7]

2.3.4 NoSQL for IoT

As discussed in Section 2.1, the primary database requirements for IoT applications are scalability and the ability to rapidly ingest data. NoSQL systems are a perfect fit for IoT because they are designed with massive horizontal scalability in mind. Another common feature of NoSQL stores is efficient use of in-memory data storage, which is extremely beneficial for write throughput and latency. Relational database management system technologies tend to fall short because they were not designed to handle the amount of data or the rate that the data is generated [31, 43]. They also employ rigid schemas which make it difficult to adapt to new use cases and standards. In contrast, NoSQL systems have more flexible schemas. A schema can be adapted simply by adding a new field to a document or adding a new column family to a table, making it simple to handle the rapidly changing data from IoT applications [31].

2.4 Storage Structures

The data structure used in a database is perhaps the most important aspect influencing performance. A poorly designed storage structure will quickly limit the read and/or write performance of the database. It is desirable to implement a structure that efficiently utilizes both memory and disk storage so that most workloads experience high throughput.

For the structures presented below it is assumed that the underlying storage medium used is a hard disk drive (HDD). Although solid state drives (SSDs) typically have far superior performance, the use of HDDs is very relevant as NoSQL systems typically rely on cheap commodity hardware. The performance of a data structure will also vary greatly on SSDs compared to HDDs.

2.4.1 B-tree

The B-tree, introduced in 1971, can be seen as a generalization of a binary search tree in that nodes can have any number of children. It was designed to minimize head seek times, making it suitable for read-intensive workloads.

A B-tree has a branching factor k which defines the maximum number of children each node can have. The branching factor used in a database is usually high (in the hundreds) to ensure that the tree remains shallow so that fewer nodes need to be traversed when looking up a record. Every internal node in the B-tree must have at least $k/2$ children and the root must have at least two. All leaf nodes appear at the same level in the tree.

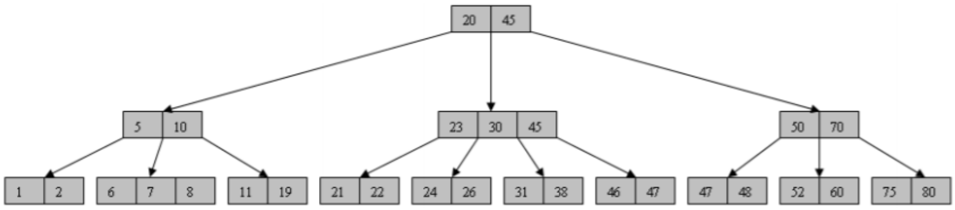


Figure 2.2: Illustration of a B-tree.

Internal nodes in a B-tree contain a number of keys and their associated data. Each key also acts as a separation value which divides its subtrees and helps guide searches through the tree. If an internal node has m subtrees it must have m - 1 keys. For example, a node with three subtrees will have two keys, k_1 and k_2. The subtrees are separated such that the left subtree will have keys lower than k_1, the middle subtree will have keys between k_1 and k_2, and the right subtree will have keys greater than k_2. This is illustrated in figure 2.2.

Performance of a B-tree

Although the B-tree has an $O(\log N)$ search time, it is not necessarily that fast because many operations will involve I/O. This is because the entire tree does not always fit in memory, so only a subset of the nodes will be in memory at any time. Queries to the tree are likely to visit different parts of the tree. The nodes visited by the queries are thus unlikely to be the same and have to be fetched from disk.

2.4.2 B$^+$-tree

A B$^+$-tree is a variation of the B-tree where data is only stored in leaf nodes and not internal nodes. It tries to solve the I/O problems of the B-tree by storing more internal nodes in memory. The internal nodes contain pointers to its children and copies of the keys in the leaf nodes which are used for separation. Consequently, the B$^+$-tree uses slightly more space than a B-tree. However, because the internal nodes do not store any data, they use less space and more of them can fit in memory. With the majority of internal nodes in memory, it is expected maximum one I/O to retrieve data from a leaf node. Additionally, the lead nodes of a B^+-tree are linked, so scanning all the records of the tree only requires one linear pass through the leaf nodes.

Implementations of B^+-trees

The B^+-tree is implemented in the Lightning Memory-Mapped Database (LMDB), an embedded key value store designed around the principles of memory mapped files. Memory mapped files make direct use of the virtual memory system of the OS, which lets LMDB make it appear as if the entire database is in RAM. With the entire database in memory, and the B^+-tree block size equal to the OS page size, LMDB is extremely memory efficient. LMDB never modifies older data by using copy-on-write semantics. Instead, updates to existing records create a completely independent copy of the memory page being written to.

B^+-trees are also widely used in filesystems and relational databases, and the popular document store CouchDB.

2.4.3 LSM-tree

The log-structured merge-tree (LSM-tree) was introduced in 1996 intended to handle high insert volumes [36]. It is composed of two or more tree-like components. One smaller component C_0 resides entirely in memory, and one or more larger components C_1, C_2, ..., C_K are stored on disk. Figure 2.3 illustrates this concept. The on-disk trees have a directory structure comparable to a B-tree, and each tree is larger than the previous one. Frequently accessed pages from the on-disk components are also stored in a memory buffer.

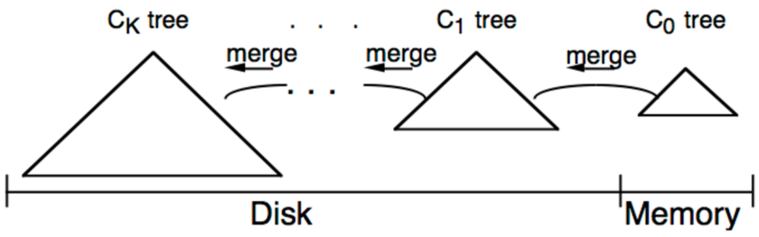


Figure 2.3: Conceptual model of an LSM-tree. Source: [36]

Records are inserted into C_0 which require no I/O, but when it reaches its threshold size it must perform a rolling merge process to merge its contents into C_1 as depicted in figure 2.4. The rolling merge process first reads a multi-page block from C_1 into a buffer called the emptying block and proceeds with a series of merge steps. Each step reads a page sized leaf node from the emptying block, merges its entries with the leaf level entries of C_0, and writes the new leaf nodes to a buffer called the filling block. When the filling block is full, it is written to a new free area on disk. C_1 will then grow until it reaches its maximum size and merges with C_2, and so on. Hence, the newest entries will always be found in C_0 and the oldest in C_K.

To ensure durability of the entries in C_0 before they are merged into C_1, a recovery file on disk logs all the insertions into C_0. When an entry from C_0 has been merged into C_1, it can safely be discarded from the recovery file.

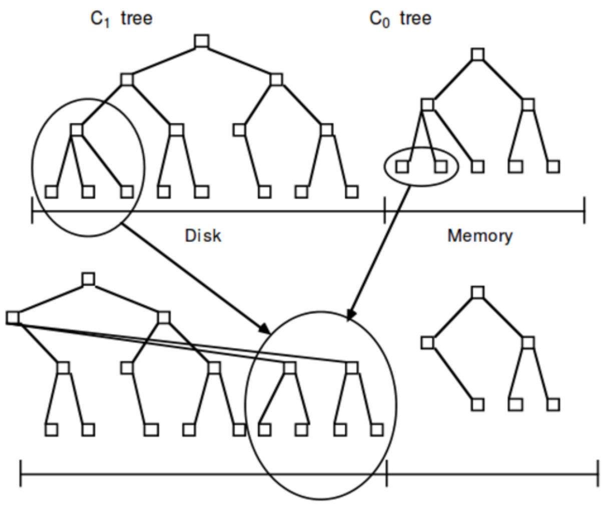


Figure 2.4: Rolling merge in an LSM-tree. Source: [36]

Queries to the LSM-tree will first search the C_0 tree and then continue the search in C_1 up to C_K until the record is found. For exact-match finds, the search is complete once it finds the desired value in an early component. Range queries on the other hand are required to search through all the trees up to C_K. However, the search can be limited to a subset of the components by the use of bloom filters [36].

Comparison of B-tree and LSM-tree I/O costs

To perform an insert in a B-tree, one must first access the position for the entry, insert it, and then write a dirty page. If D_e is the average number of pages not found in memory during a search and the cost of reading a random page is $COST_p$, the cost of inserting a single entry into a B-tree is given by:

$$COST_{\text{B-ins}} = COST_{\text{p}} \times (D_{\text{e}} + 1) \tag{2.1}$$

The LSM-tree amortizes the cost of insertions by batching inserts in C_0. The degree of amortization depends on the batch-merge parameter M during a rolling merge. M is the average number of entries in C_0 inserted into each single page leaf node of C_1 and is given by

$$M = \frac{S_\mathrm{p}}{S_\mathrm{e}} \times \frac{S_0}{S_0 + S_1} \tag{2.2}$$

where S_p is the page size, S_e is the size of a single entry, and S_0 and S_1 are the sizes of C_0 and C_1, respectively. The parameter M increases as the size of C_0 increases in comparison to C_1. Because memory capacity is limited compared to disk, it is desirable to keep C_1 small. To compensate for the resulting small database, it is desirable to break up C_1 into several components to form a multicomponent LSM-tree where the size ratio between adjacent components are adjusted accordingly. Writing a page to the LSM-tree involves reading the C_1 leaf node into memory and then writing it back, a total of two I/O. Thus, the amortized cost of writing a single entry is given by

$$COST_\text{LSM-ins} = \frac{2COST_\pi}{M} \tag{2.3}$$

where $COST_\pi$ is the cost of reading a page as part of a multi-page block I/O. The cost of insertions in an LSM-tree compared to a B-tree is further improved by the fact that $COST_\pi$ is drastically smaller than $COST_p$ as it amortizes the seek time and rotational latency over multiple pages.

Implementations of LSM-trees

The first implementation of LSM-trees was in 2006 with Google's Bigtable [13], a column-oriented datastore built on top of the Google File System. Bigtable introduced a file format called *SSTable* to provide a persistent and ordered immutable map from keys to values, and an in-memory buffer called *memtable* to hold newly committed data. These structures are analogous to the C_0 and C_1, ..., C_K components described in the original paper. The terms SSTable and memtable have been adopted in many other implementations of the LSM-tree, and will from here on be used to refer to the disk and memory components of the LSM-tree, respectively.

Google later developed LevelDB, a fast and embedded key-value with built in Snappy compression, borrowing the LSM-tree structure concepts from Bigtable. LevelDB was in 2012 forked by Facebook to develop RocksDB, designed to improve multithreaded compaction and insertions, and better utilize flash storage. LSM-trees are also implemented in the popular NoSQL databases Cassandra and HBase and in the storage engine WiredTiger used by MongoDB.

2.4.4 Fractal Tree Index

A Fractal tree (FT) index is a data structure similar to a B-tree where each internal node contains a buffer. It can be seen as a refinement of the B^ϵ-tree [5], a write-optimized B-tree based data structure. When inserting data, instead of traversing the entire tree, the record is simply inserting into the buffer at the root node. Eventually the root buffer will fill up, at which point the FT index copies the newly inserting records down a level in the tree. The records eventually reach a leaf node where they are stored as they would be stored in a B-tree. This approach is based on the same principles as LSM-trees in that random writes are transformed into sequential I/O.

Since a part of the internal nodes are used as buffer, there is less space available for separator keys. While an internal node in a B-tree has $k = B$ subtrees, an FT index might have $k = \sqrt{B}$ subtrees, where B is the block size of the disk [26].

Implementations of fractal trees

The fractal tree index was tested by its creators in a open source distribution of MongoDB called TokuMX, where it replaced the MMAPv1 storage engine based on B-trees. Figure 2.5 shows that the FT index achieves a significantly better insert performance than the normal B-tree version of MongoDB as the space requirements exceed the memory capacity. MongoDB still has support for the MMAPv1 storage engine, but now uses the LSM-tree based storage engine WiredTiger by default [8].

In addition to the MongoDB distribution, FT indexes are implemented and commercialized by Tokutek in the TokuDB storage engine for MySQL. Because FT indexes are proprietary to Tokutek, they are not present in any other open source NoSQL databases and performance comparisons to other storage structures are not feasible.

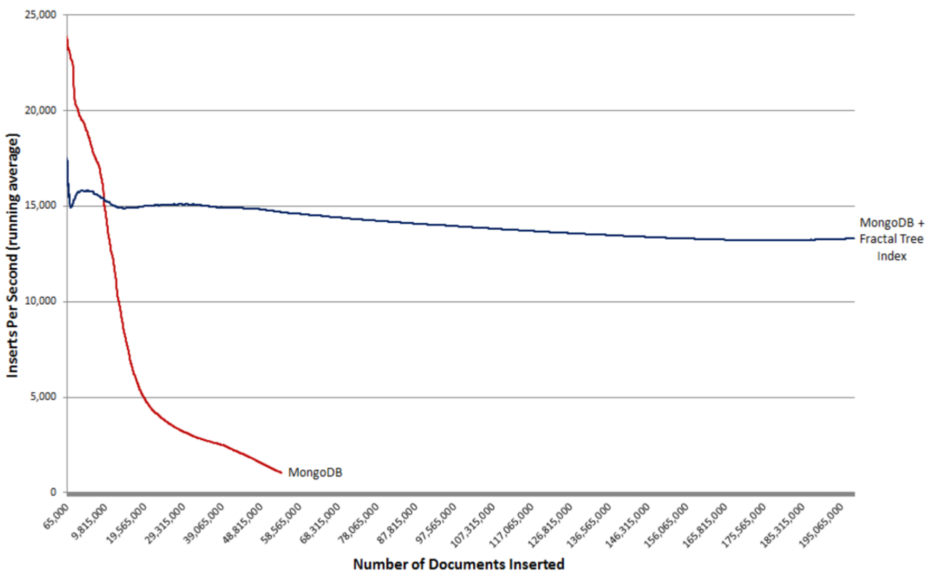


Figure 2.5: MongoDB + FT index insert performance. Source: [8]

2.5 Write Amplification

Write amplification is the amount of data physically written to storage compared to the logical amount intended to be written by the application. For example, writing a row of 100 bytes using a B-tree with 4 KB page size has a write amplification of $4096/100 \approx 40$. Both LSM-trees and fractal tree indexes provide significantly better write amplification than B-trees [26]. Figure 2.3 shows a summary of the write amplification for B-trees, LSM-trees and FT indexes given the block size, fanout and size of the database. High write amplification not only reduces the write performance for both rotating disks and solid-state disks, but also shortens the lifetime for SSDs as it can only be written to a finite number of times.

Depending on data structure, different write patterns can also affect the write amplification. For example, the write amplification will be higher for writes distributed randomly among keys than for a write pattern following a Zipfian distribution [26]. Write amplification can further be reduced by compaction at the cost of increased CPU load. Lower write amplification can reduce the CPU load by reducing the amount of data that needs to be compressed, and hence improve performance. Consequently, lower uncompressed write amplification enables using a more expensive compressor, which in turns improves the write amplification.

2.5.1 B-tree

The write amplification in a B-tree is usually very high because the entire page has to be loaded into memory and written back to disk regardless of how much data is updated in the page. Assuming that the block size of the B-tree matches the block size of the disk and keys, pointers and records are of constant size, each leaf node contains $O(B)$ records. In the worst case, every insertion to the B-tree require writing the leaf node to disk, causing the write amplification to be $O(B)$. Thus, the write performance of a B-tree is better the smaller block size.

2.5.2 LSM-tree

The exact write amplification of an LSM-tree depends on the size between adjacent levels (10 by default in LevelDB and RocksDB), the key range of SSTables undergoing compaction and the type of compaction. Two different compaction strategies are commonly supported: leveled and size-tiered.

Leveled Compaction

In leveled compaction SSTables are stored in sorted runs. A sorted run is stored as a separate level in the LSM-tree, and contains multiple SSTables with non-overlapping key ranges in sorted order. Memtables are flushed to disk as SSTables in level 0. For increased insert performance, the SSTables in level 0 are separately as sub-levels and can overlap in key range. The SSTables in level 0 will eventually be merged into level 1 as a list of non-overlapping SSTables.

If the size of level 0 is equal to the block size, the number of levels in the LSM-tree is $O(\log_k N/B)$. After being merged into a level, data gets remerged back on average about $k/2$ times [26]. The worst case write amplification is thus $O(k \log_k N/B)$.

Size-Tiered Compaction

With size-tiered compaction, the SSTables in a sorted run can contain overlapping keys, meaning a SSTable can cover the entire key range. Additionally, sorted runs are not of fixed size. A compaction merges two or more adjacent SSTables from a sorted run into a sorted run in the next level. Because the size-tiered compaction gives no guarantee of how many SSTables a key is spread across, it only incurs a worst case write amplification of $O(\log_k N/B)$, which is substantially better than leveled compaction. However, size-tiered compaction comes at the cost of increased read amplification and space amplification [26].

Data Structure	**Write Amp**	**Space Amp**
B-tree	$O(B)$	1.33
LSM-tree leveled	$O(k \log_k N/B)$	2
LSM-tree size-tiered	$O(\log_k N/B)$	3
FT Index	$O(k \log_k N/B)$	negligible

Table 2.3: Write amplification and space amplification for various data structures. Source: [26]

2.6 Compression

Data compression involves encoding information into a representation using fewer bits than the original representation. This process can either be lossy or lossless. A lossy compression will lose some information process in the encoding process, while lossless compression allows the original data to be fully reconstructed from the encoded data. Because of the loss of data, lossy compression can achieve higher compression ratios than lossless compression. For database systems, lossless compression is preferable since losing any information is often unacceptable.

2.6.1 Consequences of Compression

The most evident advantage of compression is that it reduces the database size, allowing it to store more data without having to increase its capacity. Secondly, as mentioned in chapter 2.5, compression can improve the write amplification and thus improve write performance. Compression makes it possible to achieve a write amplification less than one.

The benefits of compression come at a cost, as the encoding and decoding data require both processing power and memory. In practice, this means that compression is a trade-off between CPU usage and the amount of I/O required to store a particular amount of data.

2.6.2 Zlib

One of the most popular libraries for lossless compression is zlib. It is based on a compression algorithm called DEFLATE, which uses a combination of the LZ77 algorithm and Huffman coding. This compression model consists of two phases. In the first phase a statistical model is built from the data, which in the second phase is used to create prefix codes such that frequently occurring data produce fewer bits than less frequently occurring data. This algorithm works by exploiting statistical redundancy in the data, so that the process is reversible and avoids losing any information.

While extremely efficient and robust, zlib requires a significant utilization of CPU and can quickly become the bottleneck in the database for write-intensive workloads.

2.6.3 Snappy

Snappy is another popular compression library that was developed by Google as a faster alternative to zlib. It is able compress data at rates of over 500 MB/s on a single core, which is many times faster than zlib. However, the improved compression speed is obtained at the expense of lower compression ratios.

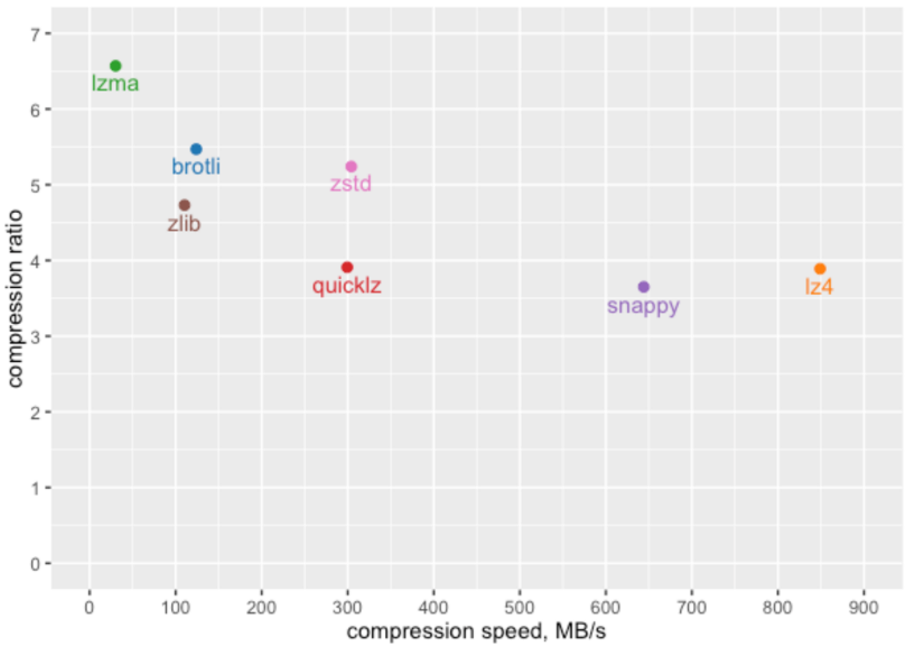


Figure 2.6: Compression ratio and speed for various algorithms. Source: [48]

2.6.4 Comparison of Compression Algorithms

Figure 2.6 shows a benchmark testing both compression speed and ratio of several common compression algorithms including zlib and Snappy. It shows the general trade-off behaviour of speed and size. However, in this benchmark lz4 achieves both better compression ratio and speed than Snappy. The performance of each algorithm depends on other factors such as type of data, patterns and block size. Additionally, high compression speed may come at the cost of decreased decompression speed.

2.7 Concurrency

Allowing multiple threads to read and write from a data structure concurrently requires some form synchronization to avoid race conditions and to ensure that the data structure maintains a consistent state. A common approach to synchronization is the use of mutexes, which requires threads to acquire a lock on the mutex before reading or writing. Typically readers can acquire a shared lock, but writers must acquire an exclusive lock. With many concurrent writers this can quickly cause contention and hence restrict the write throughput of the data structure. In order to efficiently scale the number of concurrent threads it is necessary to use a more sophisticated locking scheme like two-phase locking or a lock-free concurrency control scheme [47].

Two-Phase Locking

In two-phase locking (2PL) threads have to acquire a lock on a particular element (e.g. record, table, partition) before performing a read or write operation. The first phase of 2PL allows threads to acquire as many locks as needed. After a thread has released a lock it enters the second phase where it is prohibited from obtaining additional locks. For systems allowing transaction that modify more than one element at a time, 2PL can cause deadlocks, which require additional mechanisms to solve.

Timestamp Ordering

Every time a thread wants to read or write a record, the DBMS compares the timestamp of the operation with the timestamp of the last operation that accessed the same record. All operations are rejected if the timestamp is less than the timestamp of the previous write to that record. Writes are also rejected if the timestamp is lower than the last read. This basic timestamp ordering scheme avoids the need for locks, but incurs a significant CPU overhead for comparing timestamps.

Multi-Version Concurrency Control

Multi-version concurrency control (MVCC) is another lock-free approach to concurrency control based on timestamps. Under MVCC, every write operation creates a new version of the data and attaches a timestamp to it. Timestamps allow the DBMS to determine which version to be accessed for a read operation, thus ensuring a serializable ordering of all operations and consistent view of the database for readers. MVCC is considered a form of optimistic concurrency control in that it assumes that conflicts rarely happen.

2.7.1 LSM-Trees

In addition to worker threads, an LSM-tree also requires one or more background processes to perform compaction of components. Most LSM engines, including LevelDB, use a single thread for this purpose, while RocksDB has a configurable parameter for the maximum number of compaction threads. For CPU-bound workloads, a single compaction thread typically works 25 - 75% of the time, and will create a bottleneck for disk-bound workloads [21].

Compaction threads introduce addition considerations for concurrency in an LSM-tree. There are three types of conflicts that should be avoided during a compaction process [36]:

- A thread performing a read operation should not access a node of a disk-based component while another thread performs a rolling merge modifying the contents of the same node.
- A thread performing an operation in C_0 should not access the same part of the tree that another thread is simultaneously altering to perform a rolling merge out to C_1.
- The cursor for the rolling merge from C_{i-1} out to C_i might need to move past the cursor for the rolling merge from C_i out to C_{i+1}, since the rate of migration from C_{i-1} is at least as great as the rate of migration out of C_i. Concurrency control must permit this without a process being blocked behind the other at the intersection.

2.8 Database Scalability

In the context of databases, scalability refers to the capability of a system to increase its capacity under an increased load. There are two types of scalability found in databases: vertical and horizontal. Vertical scaling means adding more capacity to a single machine, which includes increasing the amount of CPU cores (requires efficient concurrency control), adding more RAM and disk space. On the other hand, a horizontally scalable system can improve its capacity by adding more machines to its pool of resources. Horizontal scalability is one of the key features found in NoSQL databases.

2.8.1 Sharding

In order to utilize the additional machines in a horizontally scalable system, data must be split into partitions and distributed across nodes in the cluster. A partition refers to the division of a logical database into distinct, independent parts, and can be either vertical or horizontal. Horizontal partitioning, or sharding, works by replicating the database schema and dividing the data across nodes in the cluster based on a shard key. Each logical partition, referred to as a shard, consists of an unique subset of the sharded data.

The shard key used to shard a collection of data must consist of one or more immutable attributes that exist in every entry in the collection. Careful consideration should be taken when choosing the shard key, as the choice will affect the performance, efficiency and scalability of the cluster [34]. It is also important to choose an appropriate sharding strategy for the cluster. The most common sharding strategies are:

- **Ranged sharding:** Each shard is assigned a partition of the data by dividing the data into continuous ranges based on the shard key values. A poorly chosen shard key can cause an uneven distribution of data, which potentially negates the benefits of sharding.
- **Hashed sharding:** A hash function (e.g. modulo of total number of nodes) applied to the shard key. Shards are assigned a range of data based on the computed hash values. Hashed sharding facilitates a more even distribution of data than the ranged based strategy, especially with monotonically increasing shard keys. However, this strategy is inefficient for range based queries.

SHARD 1
SHARD 2
SHARD 3
SHARD N
HORIZONTALLY SCALABLE

Figure 2.7: Sharding in a horizontally scalable system. Source: [33]

2.8.2 Consistent Hashing

Using a simple hash function to shard data presents a few issues. If the cluster experiences a membership change, at least some data must be redistributed to accommodate the joining or leaving node. It is possible that this change triggers almost all the data to be relocated to another node. Consistent hashing is a technique that attempts to mitigate this problem by minimizing the amount of keys that have to be remapped on average [45]. This technique can be seen as a form of automatic sharding in that it provides an adaptive way for partitioning, routing and load balancing in a distributed database system.

Data Partitioning and Replication

The main idea behind the consistent hashing algorithm is to hash both keys and nodes into the same range using the same hash function. The starting and ending points of the range are joined together to form a ring as illustrated in figure 2.8.

Determining which node a key belongs to is simply a matter of moving clockwise in the ring. In the example below, keys 1-3 are mapped to node B while keys 4 and 5 are mapped to node D. Consistent hashing spreads the data almost evenly across the cluster, but unlike naive hashing will not cause the whole data set to be remapped by a membership change. When a node joins the cluster, only the keys between the node itself and its adjacent node in anticlockwise direction need to be relocated.

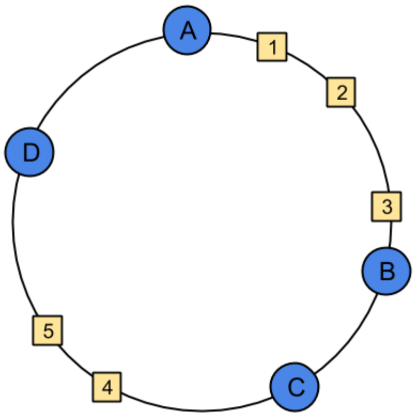


Figure 2.8: Consistent hashing - consistency ring.

In order to provide high reliability of the cluster, data partitions can be replicated across or more other nodes. A replication factor r ensures that each key will be stored on the next r nodes in clockwise direction. In the above example, with r = 3, keys 1-3 will be stored in nodes B, C and D, while keys 4 and 5 will be stored in nodes D, A and B.

The ring data structure can be stored on a dedicated node, duplicated across all nodes in the cluster or partially stored on each node. Having a central point of coordination presents a big drawback in terms of reliability and scalability, and is therefore not suited for massively distributed systems. The second option requires only one hop when serving queries, but causes a lot of gossiping overhead when nodes leave and join the cluster frequently. Keeping partial duplicates of the ring in each node does not provide direct routing of messages, but the amount of gossiping will be reduced for highly dynamic clusters.

Virtual Nodes

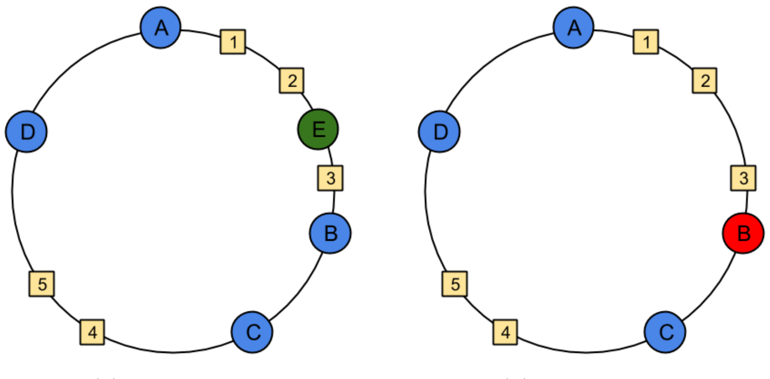
The basic consistent hashing algorithm reduces the amount of data that needs to be rehashed, but presents a few problems regarding load balancing. Realistically, some nodes will carry a larger keyspace than others, and joining or leaving nodes can cause a further imbalance of data. Additionally, the fact that machine may vary in capacity should be taken into consideration when balancing load in the cluster. These issues can be solved by splitting a physical node into a number of virtual nodes.

Virtual nodes minimize the range of each partition by assigning a number of smaller ranges to each node. Because the number of virtual nodes is much higher than the number of physical nodes, the partitions will be more equally sized and the data will be distributed more uniformly. Consequently, the amount of data that needs to be moved from a physical node to others is minimized. It is important when assigning virtual nodes to physical nodes to do so in a manner that prevents a physical node from containing replicas of the same key-ranges.

Membership Changes

In a massively distributed system it is expected that new nodes join or some nodes leave (e.g. crash) the cluster from time to time. When a new node gets hashed into the ring, its neighbours need to adjust their key-ranges to accommodate the new node. Figure 2.9a shows node E joining the ring which triggers a change to the keys in the AB-range. During this process, key 1 and 2 are dropped from node B and obtained by node E. Replica memberships also need to be adjusted such that node E also hosts key 4 and 5, while key 1 and 2 are dropped from C and D. The change of key ownership should be done synchronously, but the transfer of data to the new node can be done asynchronously.

Similarly, when a node leaves the ring, its key-range must be taken over by its adjacent node. If node B crashes as depicted in figure 2.9b, key 1-3 must be relocated to node C. Node C must also update its replica membership to include keys 4 and 5, and node A to include key 1-3.



(a) Joining node (b) Leaving node

Figure 2.9: Consistent hashing - membership changes

2.9 Database Replication Strategies

Data replication is widely used in distributed systems as a cost effective way to increase availability and fault tolerance. One of the challenges of introducing replication to a database is to do so without severely affecting performance. Because of this difficulty, many databases use an asynchronous, or lazy, replication model where updates are propagated to the replicas after it has been applied to the primary node. Lazy replication is very efficient and enables high availability but may result in inconsistencies among replicas. Alternatively, databases can use synchronous, or eager, replication in which an update is applied only after it has been committed by all replicas. Eager replication guarantees consistency but has a prohibitive cost and can ultimately affect availability. Replication therefore introduces a trade-off between consistency and performance [25].

Eager replication strategies can be organized according to three parameters: server architecture, the degree of communication between nodes during a transaction the transaction termination protocol [25]. Lazy replication forfeits the need for a transaction termination protocol as the replicas do not need to be immediately consistent. Because the server can respond to the client directly after receiving an update, the second parameter becomes less important too. Rather than communicating with the other replicas after every update, lazy replication enables updates to be bundled and propagated on an interval basis [46].

2.9.1 Server Architecture

The server architecture of a replicated system is concerned with which node clients perform updates at in the first place. There are two widely used models for the server architecture: primary copy and update everywhere.

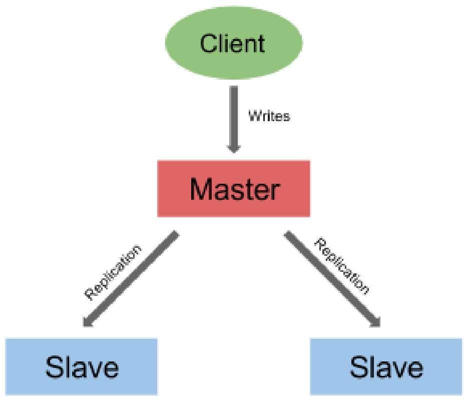


Figure 2.10: Master-slave replication model

Primary Copy

The primary copy (also called master-slave) replication model has one primary node (master) associated with each data item. All writes go to the primary copy where it is processed and then propagated to the other replica nodes (slaves). An illustration of the primary copy model can be seen in figure 2.10. This approach introduces a single point of failure as writes will be blocked if the primary copy crashes. Thus, the primary copy model requires an election protocol to elect one of the other replica nodes as the new primary. Similarly, bottlenecks are avoided by assigning primary copy ownership of partitions to different nodes such that all nodes act as master for some partitions and slave for others. The write workload can therefore be spread evenly across the cluster given an uniform distribution of keys.

Slave nodes in a primary copy system may serve as back-up only or serve read requests. However, the slave nodes may contain stale data if it has not yet received a newly updated value from the master, so reads that require completely up to date data should thus go to the master.

Update Everywhere

Update everywhere (also called multi-master) replication allows both reads and writes to be directed to any replica that holds the data item. This is illustrated in figure 2.11. The main advantage with this model is that other replicas will continue operation in case a master fails, whereas in a primary copy system the partition would become unavailable for writes until a new master is elected. Additionally, the primary copy model is able to spread the write workload for a partition more evenly in case of hotspots in certain key ranges. While inconsistencies may happen in the primary copy model if a new node is elected master before receiving updates from the previous master, they are almost guaranteed to happen with the update everywhere model. With eager replication these inconsistencies can be prevented using some form of commitment protocol such as the two-phase commitment protocol (2PC). Lazy replication requires the use of mechanisms such as vector clocks that can derive a partial ordering of updates to detect conflict among replicas and reconciliation is needed to decide on a winner.

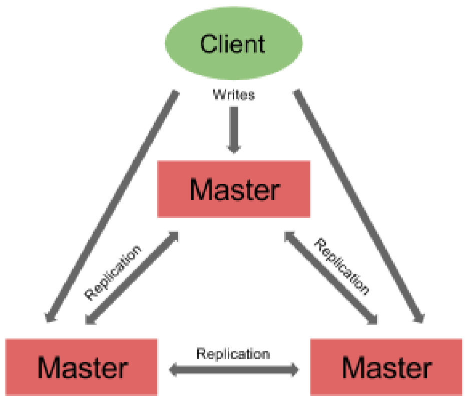


Figure 2.11: Multi-master replication model

Chapter 3

Applications

This chapter presents three example applications where massive amounts of data are generated by a large amount of things. The objects are connected to a positioning system such as GPS for global or relative tracking of location, and will frequently send their position to a central database. For practical purposes the objects are assumed to have a sufficiently long battery life and a reliable internet connection (WiFi, cellular or satellite) regardless of where they are located. It is also assumed that the objects have enough computing power and satellite coverage to be localized with the accuracy that civilian GPS promises, which is approximately 5 meters.

The applications described here are not necessarily meant to depict or solve any real world problems, but rather serve as example use cases for generating and storing massive amount of location data. Furthermore, the amount of a certain thing that are assumed to exist might be exaggerated, as the primary purpose is to generate as much data as possible for the database to ingest.

3.1 Toll Road Payment and Vignettes

In 2010 the total amount of road registered vehicles on earth passed 1 billion and this number is expected to double by year 2035. The concept of toll roads, where vehicles have to pay a fee to pass, exist in countries all over the world. While some countries only impose a fee to pass major highways, expressways or access-controlled roads, other countries toll roads extensively as a way to finance road infrastructure. Certain countries also operate with vignettes, which are coloured stickers affixed to the vehicle allowing it to use a road for a certain period of time. Both vignettes and toll payment is performed electronically in many countries, but it is still done manually in some parts of the world. Every country that implement vignettes or toll payments operate independently, and no cross country road pricing solution currently exist. Given the extensive distribution of vehicles in the world, the concept of a global road pricing system makes for an interesting example application of massive collection of location data in the Internet of Things.

3.1.1 Specification and Requirements

Concurrent Updates

While there are currently a little more than 1 billion road registered vehicles in the world, only a fraction will be driving on the roads at a given time. In addition, not all of these vehicles are driving on vignette imposed roads or toll roads. For the purpose of this problem, it is assumed that up to half of the existing cars will be driving concurrently and that they all operate in or close to areas with vignettes or toll roads. Thus, the underlying database system must be able to ingest, store and process data from at least 200 million vehicles simultaneously.

Update Frequency

The write throughput required by the database is primarily affected by the number of objects and how often they send their location. Vehicles might be travelling at speeds up to 100 km/h or more depending on country and roads. In order to accurately capture the correct road a vehicle is driving on, the update frequency of its location has to be sufficiently high. Similarly, slower moving vehicles need lower update frequency to be accurately identified along a road. Traffic stalls or very slow moving traffic is a common occurrence everywhere, so the update frequency should use a configuration based on time as well as distance. This is similar to how the update rate of locations are handled by the driving directions tool in Google Maps. Using such a configuration means that the location will be updated when the vehicle has travelled a certain distance or after a certain time window has passed, whichever happens first. A distance threshold of 20 meters should account for any potential errors caused by tolls near branching or intersecting roads. Time threshold is not as important as distance and can safely be set to 2 or 3 seconds. With 200 million vehicles this equals a minimum of 100 million location updates per second and a maximum of ~250 million location updates per second if every vehicle travels at 100 km/h.

Location Accuracy and Representation

In addition to update frequency, determining the exact path of a vehicle depends on how accurately its location is represented. Similar to how the location needs to be updated often enough to correctly determine the path of the vehicle, it also needs to be represented accurately while minimizing the storage space needed to store a single point. Accurate representation of location is particularly important in areas with multiple roads intersecting, running in parallel or branching. The margin of error should at least be smaller than the distance travelled between updates. Table 2.1 from chapter 2.2 shows that longitude can be represented with 10 meter precision using four decimal places. Representing the longitude with four decimal places requires $\log_2 3600000 \approx 22$ bits, while the latitude requires $\log_2 3600000 \approx 21$ bits, a total of 43 bits.

Longevity of Data

Registering vehicles passing a specific road for the purpose of claiming payment does not need to happen in real time. The data only need to be written to the database initially, and be processed at some later point in time. However, storing the raw location data at the rate of 250 million updates per seconds requires 1,3 GB of storage every second or 116 TB every day. At this rate a cluster of hundreds of nodes could accumulate data for days before it has to process and then drop the older data.

3.2 Tracking Humans

The single most widespread instance of non-stationary things in the world is the human population, currently at over 7.5 billion. Ignoring the obvious breach of human rights, there are countless of applications that could benefit from knowing exactly where a person is or was located at a given time. While such a tracking service seems extremely far fetched, it might not be such a distant reality considering there are currently more than 2 billion people carrying GPS enabled smartphones. For the purpose of this analysis, it is simply assumed that every single person is enabled with GPS and IPS in order to identify their position both globally and in big buildings or underground systems.

3.2.1 Specification and Requirements

Concurrent Updates

Similar to vehicles, only a fraction of the total amount of humans will be awake and moving simultaneously, but for the purposes of tracking people it is necessary to continuously monitor the entire population of 7.5 billion.

Update Frequency

People travel at widely varying speeds depending on whether they are walking, riding a bike, driving, flying or other means of transportation. It is important for people travelling at higher speeds to update their location more frequently in order to identify their correct location. People also normally stay still during sleep and many other common activities throughout the day, so updates should be triggered after the person has moved a certain distance or some time has passed. The time threshold should not be too small to avoid many meaningless updates from non-moving people. Choosing a minimal suitable distance threshold depends heavily on the use case of the application. Tracking a person in real time would benefit from very frequent updates, while fewer points would suffice for looking up the approximate location history of a person. For practical purposes the distance threshold is set to 20 meters and the time threshold to 1 minute.

Making the same assumptions as in the previous section regarding vehicles, the minimum number of updates per second is only 3 million based on time threshold and 250 million if all vehicles travel at 100 km/h. Vehicles often carry more than one person at the same time, so this number is assumed to be doubled. In addition to up to 200 million concurrent vehicles, there are about 10 million passenger flying daily in the world. Assuming that each flight is on average 2000 km, an additional 12 million updates will be generated per second. An average human walks around 10 km each day for a total of 43 million updates per second. Finally, assuming the average person is still 8 hours of the day, another 42 million updates per second are generated. In total these numbers add up to 600 million updates per second based on worst case assumptions.

Location Accuracy and Representation

Acceptable location accuracy depends on the application use case. While real time tracking would benefit from having as small errors as possible, looking up the location history of a person require only the approximate coordinates for each data point. For most such use cases a precision of around 100 meters should suffice. This precision can be achieved using 3 decimal places, for a total of $19 + 18 = 37$ bits.

Longevity of Data

With potentially 600 million updates every second the database would grow by 240 TB every day for a total of 87 PB every year. At this rate it is extremely difficult to store the location histories of every person for more than a few years. A solution which offers exceptional compression is therefore necessary for this application to be useful.

3.3 Monitoring Farm Animals

With approximately 20 billion chickens, 2 billion sheep and goats, 1,5 billion cattle, 1 billion pigs, the total number of farm animals in the world might be upwards of 30 billion. These numbers vary widely by source, some estimating a total of 70 billion farm animals. Chickens and other animals usually spending all life inside are mostly useless for this sort of application. A possible use case for monitoring farm animals might for example be to detect whether an animal has escaped its enclosure.

3.3.1 Specification and Requirements

Concurrent Updates

Excluding chickens and assuming that a majority of animals are located on a pasture or other open spaces at most time, around 5 billion animals should be monitored simultaneously.

Update Frequency

Farm animals are generally not very active, so updates can safely be triggered based on time intervals. A reasonable time interval is 1 minute given that these animals usually walk very slow and aimlessly. This produces around a total of 80 million updates per second.

Location Accuracy and Representation

For the purpose of identifying that an animal has escaped, its location only needs to be represented for decent accuracy. If an error as high as 1 km is acceptable, it can be represented using only two decimal places. This equals a total of 16 + 15 = 31 bits.

Longevity of Data

An animal is more likely to be dead if its escape is identified after a few days than if it is registered near real time. This type of application therefore does not need to store locations for more than a day or two at most. On the other hand, this application relies on real time monitoring capabilities to be useful.

3.4 Summary of Requirements

The storage requirements defined in each application only accounts for the raw location data. In practice, a timestamp needs to be stored with each value because updates do not happen on regular intervals. Most database systems support Unix timestamps, which represent time as the number of seconds elapsed since 1 January 1970. This requires only 32 bits and works until 2038. 64-bit timestamp are getting more common, but is not strictly necessary here. Each entry also need to represent the object of origin. A 32-bit integer can only represent around 4 billion different values, so a 64-bit integer is more appropriate for this. Furthermore, the supported data types in most databases are restricted to 32 or 64-bit values. This means that only the animal application can store its location using 32 bits while the other two require 64 bits to store a location. Additionally, some overhead is required for each record, e.g. field names and formatting symbols in documents.

Application	**Concurrent clients**	**Write throughput**	**Database size**
Toll Road	200M	250M w/s	432 TB per day
Tracking Humans	7,5B	600M w/s	1 PB per day
Monitoring Animals	5B	80M w/s	110 TB per day

Table 3.1: Summary of key database requirements

Based on this, the minimum storage requirements for each application is shown in figure 3.1 along with concurrent objects performing updates and the resulting write throughput under peak load.

The human tracking application presents the highest requirements of the three. For the purposes that the applications were defined, only the animal monitoring strictly needs to support real time queries, but it could be useful for the other two as well. For this reason, the highest requirement of 600 writes per second will be used in further evaluations.

Implication of Requirements

The implications of the write throughput and database size requirements are straight forward. The total storage space across all nodes must be greater than the rate at which data is generated with respect to the retention policy of data. Likewise, the required write throughput must be satisfied by distributing writes over a large cluster.

However, the high amount of concurrent clients is likely to become the first bottleneck. Since objects are transmitting data continuously, they can be expected to have a TCP connection open at all times. The maximum number of open TCP connections from unique IP addresses is limited to $2^{16} = 65536$, if not less by the operating system. 7,5 billion connections would require more than 114,000 nodes. A node can support more than 65536 clients by closing connections and opening new ones for subsequent writes. Opening and closing thousands of TCP connections every second will have a devastating effect on performance. Therefore, it is assumed that updates from objects are transparently pooled together in a way that allows nodes to receive updates from an indefinite number of sources through a few TCP connections.

Chapter 4

Database Candidates

This chapter examines the suitability of several well known, high performance NoSQL databases for handling the requirements defined in chapter 3. Time Series Databases (TSDBs), which are designed around NoSQL principles, will also be investigated.

4.1 NoSQL Databases

4.1.1 MongoDB

Being the most popular NoSQL database on the market, it makes sense to start this investigation by considering MongoDB first. MongoDB is a document database with an expressive query language, support for secondary indexes and easy accommodation of changes in applications. Data is stored in the BSON format, a binary encoding of JSON, which does not compress data. However, the storage engine WiredTiger provides both Snappy and zlib compression.

The dynamic document data model of MongoDB removes the need for a central catalog describing the structure of documents. Every document is self describing by defining the field names internally, which comes at the cost of greater use of space. Fortunately, repeating values like field names enables very efficient compression [33].

MongoDB provides a pluggable storage engine API allowing developers to tailor the database to their needs. Figure 4.1 shows a benchmark testing insert performance on a single core for MongoDBs WiredTiger, MMAPv1, RocksDB and the FT based storage engine TokuMX. As expected, both LSM-tree based storage engines perform better than MMAPv1. Interestingly, even the best performing storage engine, WiredTiger, has a write throughput of less than 100,000 documents per seconds.

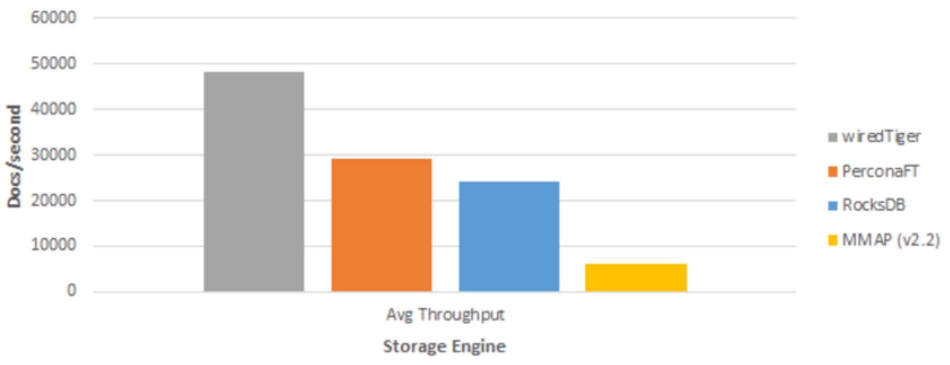


Figure 4.1: Write throughput for MongoDB storage engines. Source: [44]

Partitioning is performed using a standard sharding scheme based on a range or hash of the shard key. The approach is marketed as auto-sharding, meaning that it automatically rebalances data as the cluster grows and undergoes membership changes in a manner similar to consistent hashing. Replication in MongoDB is based on a primary copy model where secondary nodes can serve reads and writes are asynchronously replicated. If the primary node is unresponsive, a secondary will start an election to elect itself the new primary. A replica set may optionally contain an arbiter that holds no data, but can vote in order to provide a quorum in the election process.

Capped Collections

Documents in MongoDB are stored in collections. A capped collection is a fixed-size collection that works in a similar way to circular buffers. Once a collection reaches a threshold, it evicts the oldest documents in the collection to make room for new ones. This automatic removal of older data is extremely useful for applications continuously generating data and eventually fill the database. However, capped collections are not shardable, thus eliminating the usefulness for scalable systems.

Geospatial Indexes

Another useful tool provided MongoDB is geospatial indexes. A geospatial index allows location data to be efficiently retrieved based on longitude and latitude. Note that such an index is useful only if records contain other fields beside location. As with all indexes, maintaining a geospatial index adds additional complexity and I/O. Geospatial indexes cannot be used as the shard key index, but a sharded collection can maintain a geospatial index if it uses other fields than the shard key. An application can therefore not guarantee that data from close locations will be assigned to the same shard.

4.1.2 Cassandra

Cassandra is a column-oriented database initially developed at Facebook to manage its inbox search feature [27]. Today it is an open-source project managed by Apache and is the most popular column-oriented database on the market followed by HBase.

An illustration of Cassandras column-oriented data model is shown in figure 4.2. The outermost container consists of a set of column families. Column families contain a collection of rows which in turn contain columns consisting of a key, a value and a timestamp. Individual rows are not forced to have all the columns in the column family. Because column families are fixed, this model enables flexibility of rows while avoiding some of the storage overhead that document databases experience due to storing all field names in each document. Cassandra allows columns to be sorted by name or time.

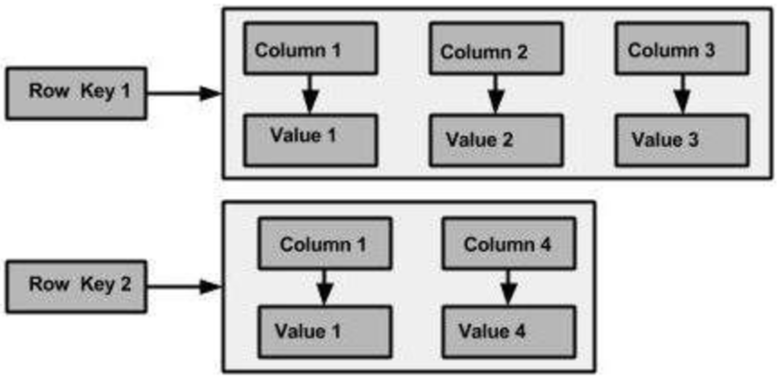


Figure 4.2: Illustration of a column family in Cassandra. Source: [11]

Cassandra's storage engine uses an LSM-tree approach heavily based on Bigtable. This storage engine comes equipped with compression on tables as well as the commit log, and can show up to a 10 percent performance improvement for writes by reducing write amplification [10]. Because SSTables are immutable, there is no need for recompression once the data has been flushed to disk.

Scalability is achieved by partitioning data using a robust consistent hashing scheme based on Dynamo [17] that efficiently handles membership changes and node failures. The replication strategy is based on a primary copy approach where consistency can be ensured for an operation by requiring a quorum of replicas to respond.

4.1.3 HBase

Another popular open source column-oriented database is HBase, which is also an Apache project and follows the Bigtable model. As such, HBase and Cassandra share many characteristics, but also have some important differences.

HBase was developed as part of the Hadoop framework and runs on top of the Hadoop Distributed File System (HDFS). HDFS handles distribution and replication of data in a master-slave fashion, which inflicts a set of limitations on HBase. Additionally, the block size in HDFS is as big as 64 KB, making it particularly unsuitable for random reads.

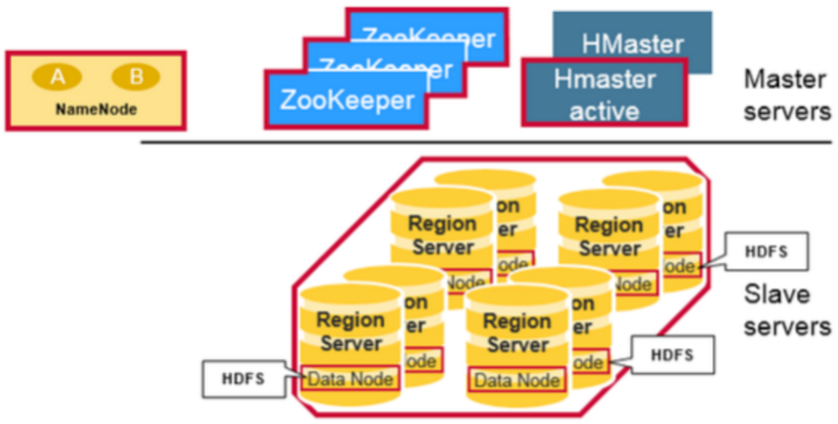


Figure 4.3: Architectural components of HBase. Source: [30]

The complete architecture of HBase is illustrated in figure 4.3. NameNode is a single instance in HDFS that manages name spaces and client access to DataNodes, which in store partitions of data and serve read and write requests. The Region-Server components, part of HBase, implements an LSM-tree structure on top of a DataNode in HDFS. A set of master servers are also present and are responsible for managing the region servers and load balancing. Finally, ZooKeeper is used to coordinate shared state for the masters and the region servers.

HBase provides strong consistency and built in MapReduce functionality through HDFS, but comes with a great deal of overhead with its complex architecture. Additionally, high availability is sacrificed with the single point of failure NameNode.

4.1.4 Redis

Redis is currently the most popular key-value store used in production. Being an in-memory storage solution, it is extremely fast, but not viable for persistent storage. This makes Redis useful as a caching mechanism on top of another persistent database. Redis is not strictly a key-value store, it supports a wide array of data structures including lists, sets, hashes and bitmaps. A single Redis node can easily serve upwards of 100,000 write requests per second [40].

Partitioning is not natively provided by Redis. However, a separate implementation called Redis Cluster provides a sharding scheme similar to that of MongoDB. Redis provides replication in a master-slave configuration, yet it does not guarantee strong consistency.

4.1.5 Insert Benchmark

A benchmark performed by End Point [14] tested the performance of MongoDB, Cassandra and HBase under a variation of different workloads. The benchmark was hosted on Amazon Web Services (AWS) EC2 instances with 7.5 GB RAM, 4 CPU cores and 800 GB of SSD storage on each node. Each benchmark was run on 1, 2, 4 and 8 nodes in a series of different workloads starting with an empty database.

Nodes	**MongoDB**	**HBase**	**Cassandra**
1	985	4,275	12,879
2	2,446	6,418	23,780
4	4,854	8,913	43,945
8	3,642	9,101	85,975

Table 4.1: Insert performance of Cassandra, HBase and MongoDB. Source: [14]

Results from the write-intensive workload are shown in table 4.1. The overall low performance can partially be attributed to the virtualization overhead in AWS and sub-optimal configurations of each database in order to provide a fair testing environment [14]. MongoDB performed much worse than the other two because the test was conducted before WiredTiger became its default storage engine. The results could potentially be around 10x higher if WiredTiger, TokuMX or RocksDB was used instead of MMAPv1. Furthermore, the results for both MongoDB and HBase are inconsistent on more than 2 nodes due to issues with AWS [14].

The results do however show that these NoSQL databases scale near linearly at lower node counts. It is no surprise that Cassandra performs better than HBase and MongoDB since it uses a multi-master replication model.

4.2 Time Series Databases

A time series database (TSDB) is optimized for handling time series data, which is a series of numeric data points of some particular metric (e.g. temperature or coordinate) over time. These databases are designed to be able to collect, store, manage and analyze time series data at scale. Time series databases are also designed to do one of two things: monitor a limited time interval or store historical data. Often, TSDBs are built on top of existing NoSQL technologies to benefit from the scaling and storage solutions that they provide, but these are generally slower than purpose built TSDBs [2]. Another thing to note about TSDBs is that metrics are immutable, meaning that data written will never change. Given this immutability it is extremely rare that conflicting values are generated on different sides of a partition. It is thus little harm in choosing availability from the CAP theorem and opting for an eventual consistent model instead [2, 16].

Queries in Time Series Databases

In addition to rapid insertion of data, TSDBs are also build to efficiently handle SCAN queries and aggregation queries like AVG, SUM, CNT, MAX and MIN. Time series data can optionally include tags consisting of a tag name and a tag value . For example, a time series can represent "location measurements", comparable to a table, while tags are used to define which object or sensor the measurement came from. Aggregation queries can be used to group data together based on time ranges (e.g. querying average values for a day) or on tags. The SCAN operation can be used to retrieve one or more rows in a specific time range. However, scanning all rows for a specific tag requires a secondary index on tags, which will have an effect on the insert performance [4].

4.2.1 InfluxDB

Among the many time series databases on the market, InfluxDB stands as the most popular one, promising millions of writes per second. Key requirements in the design of InfluxDB include the ability to scale to a few thousand nodes, high availability for reads and writes, support for billions of time series and possibility to add or remove nodes dynamically [19]. InfluxDB is a custom built TSDB that can use several different storage engines for the underlying storage of data. The options are LevelDB, RocksDB and HyperLevelDB based on LSM trees and the B+-tree based storage engine LMDB. RocksDB is the default storage engine because it performs best overall in a benchmark testing insertion, deletion and compaction on 100 million values spread over 500k columns (See table 4.2).

Test step	**LevelDB**	**RocksDB**	**Hyper**	**LMDB**
Write 100M	36m8.29s	21m18.60s	**10m45.41s**	1h13m21.30s
DB size	**2.7G**	3.2G	3.2G	7.6G
Query 100M	2m44.37s	**2m44.99s**	13m49.01s	5m24.80s
Delete 50M	3m47.64s	**1m53.84s**	6m0.38s	6m 15.98s
Compaction	3m59.87s	**3m20.27s**	6m33.36s	-
DB size	**1.4G**	1.6G	1.6G	7.6G
Query 50M	12.12s	13.59s	23.98s	**8.48s**
Write 50M	3m5.28s	**1m26.9s**	1m54.56s	3m25.96s
DB size	**673M**	993M	928M	2.5G

Table 4.2: Results from insertions, deletions and compactions in LevelDB, RocksDB, HyperLevelDB and LMDB. Source [18]

InfluxDB provides a clustering design that is split into two systems: a CP system for storing metadata such as node information, shard information and the lifetime of data, and an AP system for handling reads and writes. The CP cluster is responsible for creating shards groups, which define a set of shards in a given time range, and assigning them to nodes ahead of time to avoid contention in the metadata cluster when new time series arrive. Data is hashed to a shard by taking its timestamp and applying the modulo of the number of shards. The sharding strategy does not use a consistent hashing algorithm because it does not need to worry about rebalancing the cluster once a shard group becomes cold [19]. InfluxDB employs a multi-master replication strategy where writes go to either one, all or a quorum of replicas. Techniques like hinted handoff and anti-entropy repair from Dynamo are used to recover from write failures and ensure that replicas eventually hold the same values [19].

InfluxDB supports the data types 64-bit integer, double, boolean and string, and uses only 2.2 bytes per data point after compression. It also provides a nano second precision for metrics and benchmarks show that it can write up to 470k metrics per second [2]. Tags are also indexed in InfluxDB, allowing for efficient queries based on tag values and not only time ranges. The one big downside with InfluxDB is that sharding is only available in the enterprise edition and not in the open source version [4].

4.2.2 DalmatinerDB

DalmatinerDB is a time series database which aims to meet requirements that no other TSDB on the market can provide. It is designed to be fast, scale easily, keep data indefinitely, support a dimensional data model and have an expressive query language [2]. DalmatinerDB uses a custom built storage engine based on a flat binary format and designed around the properties of the ZFS file system, which offers exceptional compression for good storage efficiency. Its implementation is based on Riak Core, which is a single OTP application providing all necessary tools to create distributed applications. Riak Core uses consistent hashing in a master-slave setup to partition and distribute data. It also uses hinted handoff to to handle node failures.

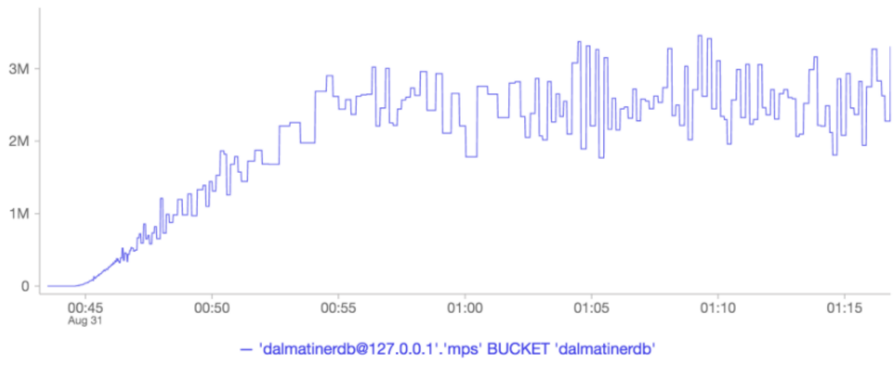


Figure 4.4: DalmatinerDB write performance on a single node. Source: [1]

The supported data types in DalmatinerDB are 62-bit float and 56-bit integer A single data point only requires a single byte after compression. The developers of DalmatinerDB report that a single node setup with 16 cores, 60 GB memory and SSD storage can achieve up to a whopping 3 million writes per second [2, 1].

Chapter 5

Discussion and Evaluation

This chapter will discuss the factors influencing write performance and suitability of the presented databases with regards to the requirements defined in chapter 3.

5.1 Storage Structures

5.1.1 LSM-Tree Performance

Judging by the top three persistent NoSQL databases in production (MongoDB, Cassandra and HBase), it is evident that the LSM-tree is the leading technology for NoSQL storage engines. The popularity of LSM-trees can be attributed to its superior write-performance to traditional B-trees and its variants, while still having acceptable read performance. There are, however, still many areas in which LSM-trees can be improved to achieve higher write throughput and competitive read performance.

Vertical Scalability

One of the key strengths of NoSQL is the ability to scale horizontally. Nevertheless, the LSM-trees powering these databases have room for improvement through vertical scaling. Nowadays, it is not uncommon for commodity servers to have 16 cores, yet most LSM-tree implementations only scale to 4. Golan-Gueta et al. [21] developed cLSM, which aims to improve the scalability of LSM-trees by eliminating blocking in scenarios that do not involve disk accesses. This support for non-blocking atomic operations is enabled by a specific implementation of the memtable using a skip-list data structure and optimistic concurrency control.

Benchmarks based on synthetic CPU-bound workloads were tested against other LSM data stores. The result for the write only scenario with uniformly distributed keys is shown in figure 5.1. cLSM scales to 8 cores and achieves a 80% throughput advantage over its closest competitor. However, for scenarios where the bottleneck was disk compactions, RocksDB was shown to have higher throughput up to 16 cores [21].

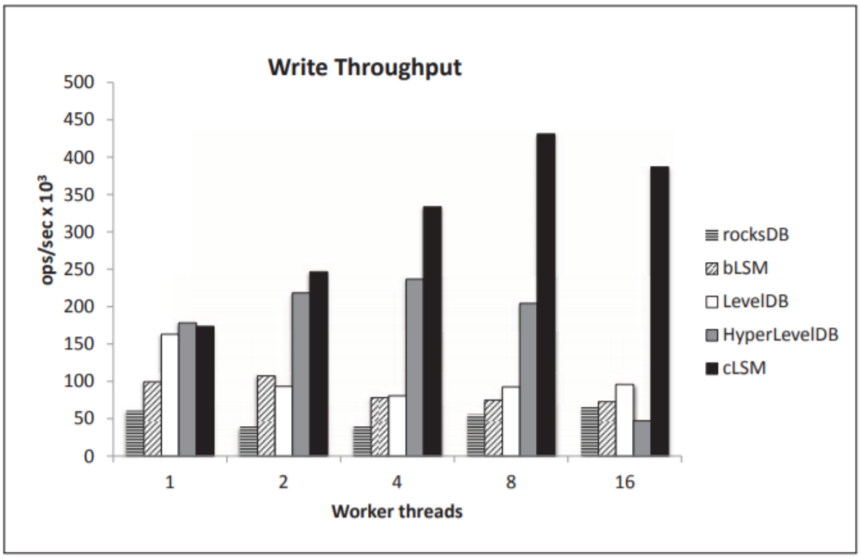


Figure 5.1: Write performance of LSM-trees on multiple cores. Source: [21]

5.1.2 Proprietary Storage Structures

While LSM-trees are considered state of the art, some NoSQL stores are built on proprietary data structures such as the FT indexes by Tokutek introduced in section 2.4.4. Another example of a proprietary storage structure is the Hierarchical B+Tree used by Couchbase [42], which is the second most popular document database behind MongoDB.

These approaches differ from LSM-trees, but have in common that they strive to maximize write throughput by sequential writes. However, it is extremely difficult to compare the performance of these structures because only the vendor of the proprietary solutions may perform such tests. It is therefore common for said vendors to introduce biases towards their own product when performing benchmarks, in order to draw more customers.

5.2 Horizontal Scalability Limitations

Utilizing a storage structure that avoids random disk writes such as LSM-trees is extremely beneficial for handling write-intensive workloads. However, in order to satisfy the enormous storage and write throughput requirements for the applications presented in chapter 3, it is essential that the database scales well. All the major NoSQL databases presented in chapter 4 implement a variation of the consistent hashing algorithm or some form of automatic sharding, which in theory should scale linearly. While there is no practical limitation to the amount of shards or how many nodes can fit on the ring in a consistent hashing scheme, bottlenecks might emerge at higher node counts. For instance, membership changes need to be propagated across all nodes to provide an eventually consistent view of membership [17].

Big companies like Facebook, Google and Apple are using NoSQL technologies store petabytes of data on thousands of nodes. Apple reported in 2014 that it was running over 75,000 Cassandra nodes to store more than 10 petabytes of data and millions of operation every second [3]. This is the highest node count ever reported for a NoSQL database in production. The ability of Cassandra to scale to such a high number of nodes is likely enabled by the completely decentralized failure detection and membership protocols inspired by Dynamo [17].

Assuming that a Cassandra cluster can scale to 75,000 with minimal overhead and each node experiences a write throughput at around 12,000 writes per second (from table 4.1), it is theoretically possible to achieve upwards of 1 billion writes per second across the cluster. However, Apple reports serving only some million operations per second, suggesting that the performance of each Cassandra node degrades as the cluster grows.

In HBase there is a restriction on the amount of partitions, which in turn imposes a limit on the number of nodes. This restriction is caused by the master acting as bottleneck when assigning partitions to nodes and the heavy ZooKeeper usage associated with it [24]. HBase is thus unlikely able to achieve the same level of scalability as a database with completely decentralized shard management like Cassandra. MongoDB also stores information about shards on a single node, and while this configuration server is replicated for availability, it might act as a bottleneck when the number of shards become too high.

5.3 NoSQL and TSDBs Solutions

The databases presented in chapter 4 only represent a fraction of the NoSQL databases available on the market. They were chosen simply because they are the leading database solutions in production and offer at least some variety in design. None of these databases are specially designed to handle the collection of massive amounts of location data, but rather for general storage. It is very likely that there exist a less popular NoSQL database, or even relational database, that better fit the nature of this kind of application. For example, some databases use R-trees for indexing multi-dimensional information such as coordinates, and are excellent for spatial queries. However, R-trees are not well suited for rapid insertions.

Among the databases that were investigated, time series databases were the most promising. Because of their more specific use case, many features that are found in general purpose databases are not present. Most important is the relaxation of consistency and durability. Missing a few data points can be devastating in a normal database, but presents no issues in TSDBs since the missing data in a time series often can be interpolated anyway. Along with a more light weight data model and less storage overhead, this makes TSDBs well suited for storing massive amounts of raw continuous data from millions of sources. While TSDBs are designed to analyze data in real time, they are restricted to simple operations like scanning data within a particular time period and finding maximum or average values. This presents two issues in regards to the proposed applications. First, data should be retrieved based on tags (i.e. object the data belongs to) and not time windows as the interaction between objects is not of interest. Second, if data for a specific tag can be retrieved using a scan, it must be processed on the application side in order to extract the desired information (e.g. if a vehicle passed a specific road).

Only two TSDBs were looked into based on a ranking [2] of the top performing ones overall. It should be noted that, like with most tests including a proprietary candidate, DalmatinerDB likely came out at the top because the survey was performed by someone affiliated with the company behind it. Nevertheless, if DalmatinerDB truly can achieve the promised 3 million writes per second on a single node, it is by far the best candidate for high write throughput applications. In addition, a single metric can be compressed to one byte, making the best solution for storage efficiency too. Unfortunately, no documentation or implementation details are available for DalmatinerDB except for Riak Core which it is built upon. The underlying data structure thus remains a mystery, but is presumably a variant of LSM-tree since that is what Riak uses.

In addition to InfluxDB, there are many open source TSDBs with high write performance. One example is Prometheus [38], which was benchmarked at 800k metrics written per second [2]. Prometheus does, however, not provide any form of sharding and must be run as multiple independent servers to scale, much like the open source version of InfluxDB. Because of the nuisances of the other top performing TSDBs, it is no wonder why InfluxDB is the most popular one. InfluxDB is also likely to scale immensely as it does not rely on a single server to store all shard information. With 350k writes on a single node, it is not unfeasible for InfluxDB to handle the 600 million writes required every second. Additionally, default indexing on tags allows InfluxDB to perform meaningful queries on data from individual objects.

5.4 Database vs. Data Lake

Tracking billions of objects and serving several hundred million updates per second seem to be cutting the edge even for state of the art NoSQL database systems. Part of the reason is the inherent trade-off a database needs to make in order to satisfy reads operations while rapidly ingesting data. If the application does not require queries to be performed while contiguously writing data to the database, it would be beneficial to use a data lake instead. A data lake is a write-only store where all data is stored in its raw format. The data could later be transferred in bulk to a data warehouse or other system suitable for data analysis.

To illustrate the benefits of a data lake, the code in appendix A.1 was used to test the speed at which small records could be written to a single file. Running on a machine with an Intel Core i7 2.0 GHz processor with 4 cores it was able to write 5 million records to a single file in under a second. This shows that, given linear scalability, that achieving 600 million writes per second is feasible in a cluster of only 200 nodes. If the intent of the application is to provide a historical view of the location of things, then this might be an acceptable solution. However, for operational analysis, a database capable of serving queries in real-time is necessary.

5.5 Other Considerations

Concurrency Control

As discussed in section 2.7, using an efficient concurrency control scheme is crucial in order to utilize many threads concurrently writing and reading to a common data structure. However, the nature of IoT ensures that conflicting writes may never happen as long as messages are delivered in order over the network. This is because updates to a data item will only be performed by the object itself. Since the very purpose of the applications is to store new versions without discarding old ones, using an optimistic scheme like MVCC is natural. This also allows multiple readers to access the database simultaneously without the additional overhead of concurrency control. By design, this is how concurrency is handled in many NoSQL databases, especially the Dynamo clones such as Cassandra. Utilization of multiple cores is still an area with room from improvement though.

Timestamp Allocation

An inherent problem of the applications described in section 3 is the potential bottleneck of timestamp allocation. Timestamps are not only needed to reason about the location data, but also for concurrency control and eventual consistency. If each node in the system is expected to handle 3 million writes per second, a significant overhead will incur from timestamp allocation. Furthermore, if a primitive mechanism like a mutex is used to allocate timestamps, the allocation throughput will degrade with the number of cores [47]. One solution is to have timestamps provided by the application, which is perfectly feasible for this type of IoT application. However, this will lead to inaccurate results for queries interested in more than one object's location, since the clocks for each object are unlikely to be synchronized.

Consistency and Availability

While one should be careful labeling NoSQL databases with respect to CAP, it is safe to say that MongoDB and HBase lean more towards the CP side. In the applications described here, inconsistency implies that replicas do not see the most recent locations of things. Replicas not up to do date will however have a consistent view of previous locations. The need for strict consistency boils down to whether the application has need for real time queries. In case of a network partition, a client might read from a replica not receiving the newest updates. Fortunately, it is easy to identify if data is older since it is associated with a timestamp. The client can then perform a read to all replicas to ensure that it finds the newest location of the object. In other words, using an eventual consistency model presents no real harm to this type of application. It is more beneficial to use asynchronous replication and opt for high availability instead.

5.6 Limitations

As mentioned in the introduction, the biggest limitation to this thesis is the impracticality of performing real tests. Although it would be feasible to conduct an unbiased test of open source storage engines on a single node, the real deciding factor is scalability. Testing various databases on a small number of nodes would be feasible, but it would not produce any valuable results because there are no bottlenecks preventing linear scaling at lower node counts. Determining the real scalability limits of a database can only be estimated by looking at actual production settings like Apple's Cassandra cluster. Comparing database scalability this way is not only difficult because these companies do not like to share this information, but also because workload and use case may vary widely between two production settings.

Chapter 6

Conclusion and Further Work

6.1 Conclusion

The motivation behind this thesis was to evaluate whether existing NoSQL technologies are capable of managing the enormous write workloads that would arise from tracking the location of *everything* in the world. An application tracking the location of all humans presented the highest requirements of 600 million writes per second resulting in many hundred terabytes per day.

Although having support for other data structures, it is clear that the LSM-tree is the most prominent storage solution for almost all persistent NoSQL databases. Compared to B-trees, the design of LSM-trees facilitates improved write throughput in many areas, while providing acceptable read performance. The transformation of random writes to sequential I/O combined with efficient compression reduces write amplification substantially, which in turn improves performance for disk-bound workloads on HDDs as well as SSDs. Furthermore, modern LSM-tree based engines such as RocksDB are getting increasingly better at concurrent updates, to the point where vertical scalability could become more relevant in NoSQL systems.

Looking at horizontal scalability, it has been displayed that most of the popular NoSQL databases puts a high value on consistency in replicated environments. It is apparent that including any form of master server in the distributed database will restrict the horizontal scalability of the system, whereas a fully decentralized design can scale much higher in a more linear fashion.

In evaluating existing NoSQL databases, no candidate offered a perfect solution for such applications. Purpose specific databases for multidimensional data have poor insert performance, and general purpose databases typically experience much overhead in storing many small continuous data points. On the other hand, time series databases excel at rapidly ingesting small data points associated with timestamps coming from millions of sources. The downside with time series databases is their lack of complex query functionality.

It is evident that certain trade-offs in terms of consistency or to some extent availability, are necessary in order to attain 600 million writes per second on top of real time query support. The ideal database for this type of application would have a completely decentralized design with asynchronous replication and TSDB-like model with default indexing on tags. From the small set of databases examined in this thesis, Cassandra or InfluxDB seem to be the best fit for collecting massive amount of location data.

6.2 Further Work

Performing real tests of various databases at scale is required to get an absolute answer for which NoSQL technology suits this type of application. While testing on tens of thousands of nodes is highly unrealistic, it is possible to perform small-scale tests and then reason about scalability of the solutions. Some suggestions for small-scale studies are:

- Set up a unbiased environment for testing various NoSQL databases. The databases should be configured to store triples containing a timestamp, a location and a value identifying its origin. Several benchmarks should be performed testing insert only workloads and mixed workloads with reads on the newest location as well as historic locations. Space utilization should also be compared. Tests can first be performed on a single machine and then scaled to a few nodes, but preferably not in the cloud.

- Take one promising database (e.g. Cassandra) and see what optimizations can be done to better facilitate this type of application. This includes finding optimal usage of the data model, tweaking configurations for the LSM-tree and concurrency mechanisms

- Implement a richer query model on top of an open source TSDB (e.g. InfluxDB) and evaluate its performance on a single node against a NoSQL database in a fair testing environment.

Appendix A

Code

A.1 Writing to a Flat File

```
public class FileWritingTest {
    private static final int REC_COUNT = 5000000;
    private static final String RECORD =
        "This␣text␣should␣equal␣32␣bytes\n";
    private static final int BYTES = RECORD.getBytes().length;

    public static void main(String[] args) throws Exception {
        List<String> records = new ArrayList<String>(REC_COUNT);
        int size = 0;
        for (int i = 0; i < REC_COUNT; i++) {
            records.add(RECORD);
        }
        System.out.println(REC_COUNT*BYTES/(1024*2014)+"MB");
        write(records);
    }
    private static void write(List<String> records)
            throws IOException{
        File file = File.createTempFile("test", ".txt");
        try {
            FileWriter writer = new FileWriter(file);
            long start = System.currentTimeMillis();
            for (String record: records) {
                writer.write(record);
            }
            writer.flush();
            writer.close();
            long end = System.currentTimeMillis();
            System.out.println((end-start)/1000f + "␣secsonds");
        } finally {
            file.delete();
        }
    }
}
```

Listing A.1: Write speed test for 5 million records of 32 bytes to a flat file.

Bibliography

[1] Steven Acreman. *DalmatinerDB Write Performance Benchmark.* URL: `https://gist.github.com/sacreman/b77eb561270e19ca973dd5055270fb28` (visited on 05/26/2017).

[2] Steven Acreman. *Top 10 Time Series Databases.* URL: `http://blog.outlyer.com/top10-open-source-time-series-databases` (visited on 05/19/2017).

[3] Matt Asay. *Apple's secret NoSQL sauce includes a hefty dose of Cassandra.* URL: `http://www.techrepublic.com/article/apples-secret-nosql-sauce-includes-a-hefty-dose-of-cassandra/` (visited on 07/07/2017).

[4] Andreas Bader, Oliver Kopp, and Michael Falkenthal. *Survey and Comparison of Open Source Time Series Databases.* 2017. URL: `http://btw2017.informatik.uni-stuttgart.de/slidesandpapers/E4-14-109/paper_web.pdf`.

[5] Michael A. Bender et al. *An Introduction to B^ϵ-trees and Write-Optimization.*

[6] Eric Brewer. *CAP Twelve Years Later: How the Rules Have Changed.* URL: `https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed` (visited on 04/24/2017).

[7] Eric Brewer. *Towards Robust Towards Robust Distributed Systems.* July 2000. URL: `https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`.

[8] Tim Callaghan. *10x Insertion Performance Increase for MongoDB with Fractal Tree Indexes.* URL: `https://www.percona.com/blog/2012/08/23/10x-insertion-performance-increase-for-mongodb-with-fractal-tree-indexes/` (visited on 05/25/2017).

[9] *Cassandra.* URL: `http://cassandra.apache.org/` (visited on 02/09/2017).

[10] *Cassandra Compression.* URL: `http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsAboutConfigCompress.html` (visited on 07/02/2017).

[11] *Cassandra Data Model*. URL: `https://www.tutorialspoint.com/cassandra/cassandra_data_model.htm` (visited on 07/09/2017).

[12] Rick Cattell. *Scalable SQL and NoSQL Data Stores*. Dec. 2011. URL: `http://www.cattell.net/datastores/Datastores.pdf`.

[13] Fay Chan et al. *Bigtable: A Distributed Storage System for Structured Data*. June 2006. URL: `https://www.cs.utexas.edu/~dahlin/Classes/GradOS/papers/chang06bigtable.pdf`.

[14] End Point Corporation. *Benchmarking Top NoSQL Databases*. May 2015. URL: `https://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf`.

[15] *CouchDB*. URL: `http://couchdb.apache.org/` (visited on 02/09/2017).

[16] DalmatinerDB. *Tradeoffs & Design*. URL: `https://dalmatiner.readme.io/docs/tradeoffs-design` (visited on 05/19/2017).

[17] Giuseppe DeCandia et al. *Dynamo: Amazon's Highly Available Key-value Store*. Oct. 2007.

[18] Paul Dix. *Benchmarking LevelDB vs. RocksDB vs. HyperLevelDB vs. LMDB Performance for InfluxDB*. URL: `https://www.influxdata.com/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/` (visited on 05/26/2017).

[19] Paul Dix. *InfluxDB Clustering Design – neither strictly CP or AP*. URL: `https://www.influxdata.com/influxdb-clustering-design-neither-strictly-cp-or-ap/` (visited on 05/26/2017).

[20] Dave Evans. *The Internet of Things: How the Next Evolution of the Internet Is Changing Everything*. Apr. 2011. URL: `http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`.

[21] Guy Golan-Gueta, Edward Bortnikov, and Eshcar Hillel. *Scaling Concurrent Log-Structured Data Stores*. Apr. 2015.

[22] Goetz Graefe. *Write-Optimized B-Trees*. Sept. 2004. URL: `http://dx.doi.org/10.1007/s002360050048`.

[23] *HBase*. URL: `https://hbase.apache.org/` (visited on 02/09/2017).

[24] *HBase Regions*. URL: `https://hbase.apache.org/book.html#regions.arch` (visited on 07/08/2017).

[25] David Karger et al. *Database Replication Techniques: a Three Parameter Classification*. Oct. 2000.

[26] Bradley C. Kuszmaul. *A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees*. Apr. 2014. URL: `http://insideanalysis.com/wp-content/uploads/2014/08/Tokutek_lsm-vs-fractal.pdf`.

[27] Avinash Lakshman. *Cassandra – A structured storage system on a P2P Network*. URL: `https://www.facebook.com/notes/facebook-engineering/cassandra-a-structured-storage-system-on-a-p2p-network/24413138919/` (visited on 02/29/2017).

[28] Karen Lewis. *Where's my stuff? How location and IoT play well together.* URL: https://www.ibm.com/blogs/internet-of-things/location-iot/ (visited on 04/12/2017).

[29] Christian Lundquist. *Location of things: Why location matters in IoT.* URL: http://internetofthingsagenda.techtarget.com/blog/IoT-Agenda/Location-of-things-Why-location-matters-in-IoT (visited on 04/11/2017).

[30] Carol McDonald. *An In-Depth Look at the HBase Architecture.* URL: https://mapr.com/blog/in-depth-look-hbase-architecture/ (visited on 07/02/2017).

[31] MongoDB. *Internet of Things.* URL: https://www.mongodb.com/use-cases/internet-of-things (visited on 04/12/2017).

[32] *MongoDB.* URL: https://www.mongodb.com/ (visited on 02/09/2017).

[33] MongoDB. *MongoDB Architecture.* URL: https://www.mongodb.com/mongodb-architecture (visited on 06/26/2017).

[34] MongoDB. *Sharding.* URL: https://docs.mongodb.com/manual/sharding/ (visited on 05/11/2017).

[35] *Neo4j.* URL: https://neo4j.com/ (visited on 02/09/2017).

[36] Patrick O'Neil et al. *The log-structured merge-tree (LSM-tree).* June 1996. URL: http://dx.doi.org/10.1007/s002360050048.

[37] Oracle. *Berkeley DB.* URL: http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html (visited on 02/09/2017).

[38] *Prometheus.* URL: https://prometheus.io/ (visited on 05/25/2017).

[39] *Redis.* URL: https://redis.io/ (visited on 02/09/2017).

[40] *Redis Benchmark.* URL: https://redis.io/topics/benchmarks (visited on 07/13/2017).

[41] *Riak.* URL: http://basho.com/products/riak-kv/ (visited on 02/09/2017).

[42] Chiyoung Seo. *Next Generation Storage Engine: ForestDB.* URL: https://www.couchbase.com/resources/presentations/next-generation-storage-engine-forestdb.html (visited on 07/07/2017).

[43] Dan Sullivan, James Sulllivan. *Find the IoT database that best fits your enterprise's needs.* URL: http://internetofthingsagenda.techtarget.com/feature/Find-the-IoT-database-that-best-fits-your-enterprises-needs (visited on 04/12/2017).

[44] Jon Tobin. *Myth Busting: MongoDB Scalability.* URL: https://www.percona.com/blog/2016/02/19/myth-busting-mongodbs-scalability/ (visited on 06/26/2017).

[45] Matthias Wiesmann et al. *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.* May 1997. URL: http://dl.acm.org/citation.cfm?id=258660.

[46] M. Wiesmann et al. *Understanding Replication in Databases and Distributed Systems.* Apr. 2000. URL: http://www-users.cselabs.umn.edu/classes/Spring-2016/csci8980-lds/Papers/ProcessReplication/Understanding-Replication-icdcs2000.pdf.

[47] Xiangyao Yu et al. *Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores.* Nov. 2014. URL: http://www.vldb.org/pvldb/vol8/p209-yu.pdf.

[48] Peter Zaitsev and Vadim Tkachenko. *Evaluating Database Compression Methods: Update.* URL: https://www.percona.com/blog/2016/04/13/evaluating-database-compression-methods-update/ (visited on 06/26/2017).