Cloud Databases for Internet-of-Things Data

Thi Anh Mai Phan

Kongens Lyngby 2013
IMM-M.Sc.-2013-48

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-48

Summary (English)

The vision of the future Internet of Things is posing new challenges and opportunities for data management and analysis technology. Gigabytes of data are generated everyday by millions of sensors, actuators, RFID tags, and other devices. As the volume of data is growing dramatically, so is the demand for performance enhancement. When it comes to this Big Data problem, much attention has been paid to cloud computing and virtualization for their unlimited resource capacity, flexible resource allocation and management, and distributed processing ability that promise high scalability and availability.

On the other hand, the types and nature of data are getting more and more various. Data can come in any format, structured or unstructured, ranging from text and number to audio, picture, or even video. Data are generated, stored, and transferred across multiple nodes. Data can be updated and queried in real time or on demand. Hence, the traditional and dominant relational database systems have been questioned whether they can still be the best choice for current systems with all the new requirements. It has been realized that the emphasis on data consistency and the constraint of using relational data model cannot fit well with the variety of modern data and their distributed trend. This led to the emergence of NoSQL databases with their support for a schema-less data model and horizontal scaling on clusters of nodes. NoSQL databases have gained much attention from the community and are increasingly considered as a viable alternative to traditional databases.

In this thesis, we address the issue of choosing the most suitable database for Internet of Things big data. Specifically, we compare NoSQL versus SQL databases in the cloud environment, using common Internet of Things data

types, namely, sensor readings and multimedia data. We then evaluate their pros and cons in performance, and their potential to be a cloud database for the Internet of Things data.

Preface

This thesis was prepared at Aalto University, School of Science, Finland in partial fulfillment of requirements for the M.Sc. double degree in Security and Mobile Computing (NordSecMob) in Aalto University, School of Science, Finland and Technical University of Denmark (DTU), Denmark.

The thesis deals with extensive experiments to evaluate and compare the performance of two classes of database, namely, SQL and NoSQL as cloud databases for Internet of Things data. The focus is on two popular types of data, that is, sensor scalar data and multimedia data.

Lyngby, 28-June-2013

Thi Anh Mai Phan

Acknowledgements

First of all, I would like to express my deepest gratitude to my supervisor at Aalto University, Prof. Jukka K. Nurminen. Without his invaluable guidance and advices that came from his immense knowledge and experiences in the field, it would not be possible to finish this thesis.

I would like to thank my supervisor from my host university DTU, Prof. Nicola Dragoni, who was always willing to help and give his best guidance and support.

I am grateful to Dr. Mario Di Francessco, my instructor, for his useful instructions and suggestions that helped me throughout the process of the thesis.

In addition, a special thank to Mr. Mikael Latvala from There corporation, for his introduction to the Home Energy Management System of There corporation, and his practical comments that played an important role in my work.

Last but not least, I would like to thank my parents, my elder sister, and my friends, without whom I would not have been able to complete this thesis.

Contents

Summary (English) i

Preface iii

Acknowledgements v

1 Introduction 1
- 1.1 Problem statement . . . 2
- 1.2 Contribution . . . 3
- 1.3 Structure . . . 3

2 Databases 5
- 2.1 CAP Theorem, ACID vs. BASE . . . 5
- 2.2 SQL Databases . . . 7
- 2.3 NoSQL Databases . . . 8
  - 2.3.1 NoSQL Properties . . . 8
  - 2.3.2 NoSQL Categories . . . 13
- 2.4 Tested databases . . . 16
  - 2.4.1 MySQL . . . 16
  - 2.4.2 CouchDB . . . 19
  - 2.4.3 MongoDB . . . 21
  - 2.4.4 Redis . . . 25

3 Cloud databases for the Internet of Things 29
- 3.1 Internet of Things . . . 29
  - 3.1.1 Internet of Things vision . . . 30
  - 3.1.2 Internet of Things data . . . 31
- 3.2 Cloud Databases . . . 33
  - 3.2.1 Amazon Web Services . . . 33

3.2.2 Scalability . . . 34
3.3 Literature Review . . . 36

4 Experimental Methodology and Setup 41
4.1 Experiment overview . . . 41
4.2 Experiment environment . . . 43
4.2.1 Hardware and Software . . . 43
4.2.2 Database configuration . . . 43
4.2.3 Libraries and drivers . . . 44
4.3 Sensor scalar data benchmark . . . 44
4.3.1 System description . . . 44
4.3.2 Data structure . . . 45
4.3.3 Parameters . . . 48
4.4 Multimedia data benchmark . . . 48
4.4.1 System description . . . 48
4.4.2 Data structure . . . 49
4.4.3 Parameters . . . 50

5 Experimental Results 51
5.1 Sensor scalar data benchmark results . . . 51
5.1.1 Bulk insert . . . 51
5.1.2 MongoDB index . . . 54
5.1.3 Write latency . . . 56
5.1.4 Read latency . . . 58
5.1.5 Database size . . . 61
5.2 Multimedia data benchmark results . . . 63
5.2.1 Write latency . . . 63
5.2.2 Read latency . . . 64

6 Conclusion 67

Bibliography 71

Chapter 1

Introduction

The development of pervasive computing, RFID technology, and sensor networks has created the ground for the so-called *Internet of Things* (IoT) [AIM10]. The major idea behind it is that the Internet will exist as a seamless network of interconnected "smart" objects that forms a global information and communication infrastructure. The vision of the Internet of Things consists in a huge, dynamic, and expandable network of networks, involving billions of entities. These entities simultaneously generate data and communicate with each other. A side effect is then the massive volume of data that comes to the network. Everyday, systems of different fields including manufacturing, social media, or cloud computing crank out gigabytes of data, which can contain text, pictures, videos and more. According to IDC 2012 Digital Universe Study [GR12], the world information is doubling every two years. By 2020, the size of information is 50 times more than that of 2010, reaching 40 trillion gigabytes, 40% of which will be either processed or stored in a cloud. This is not only due to the number of IoT objects, but also because of the massive data generation. Besides, many of these data are updated in real-time and across multiple nodes. Hence, the challenge is to handle this large amount of things and data, all in a global information space, with a performance that can meet real-time requirement.

When it comes to Big Data, *cloud computing* is closely involved. According to the related paradigm [AFG+10], hardware and software resources are placed remotely and accessible over a network. The physical infrastructure is virtual-

ized and abstract to users, providing virtually unlimited resource capacity on demand. Cloud databases [MCO10] are an important part in the cloud infrastructure. To deal with huge data volumes, cloud databases use cloud computing to optimize scalability, availability, multitenancy, and resource usage.

The rapid growth in the amount of data is tightly coupled with radical changes in the data types and how data is generated, collected, processed, and stored. With all the new sources of data, data tend to be distributed across multiple nodes and no longer conform to some predefined schema definition. In fact, unstructured and semi-structured data make up 90% of the total digital data space [GR11], including text messages, log files, blogs, media files and more.

The problem has boosted the creation of new technologies to handle the data growth while improving system performance. Several database management systems (DBMS) have been developed and characterized to this end. Even though SQL databases have been the classic and dominant type among database systems so far, questions have been raised whether the traditional relational databases fit well with all the new data types and performance requirements. Here comes a new class of database referred to as NoSQL databases. NoSQL databases store data very differently from the traditional relational database systems. They are meant for data of schema-free structure, and are claimed to be easily distributed with high scalability and availability. These properties are actually needed to realize the vision behind Internet of Things data.

1.1 Problem statement

Regarding Internet of Things data, a big question is how to manage the data system in an efficient and cost-effective way. That depends on a proper planning on which DBMS is used to store the concerned data and how it is configured to provide adequate performance. As mentioned before, a variety of databases are currently available, including SQL and NoSQL databases. However, which model and solution best fit IoT data is still an open problem. As far as we know, there has not been much research on a general database solution for IoT data that provides a practical, experimentally-driven characterization of the efficiency and suitability of different databases, especially in the cloud environment. Hence, the thesis addresses this problem and looks for a solution that can provide the best performance for the various types and the large amount of IoT data.

1.2 Contribution

This thesis investigates different types of cloud databases. The focus is to evaluate and compare NoSQL databases against traditional SQL databases, in order to point out their differences in performance, usage and complexity. Along with that, the thesis characterizes the typical types of IoT data, and then abstracts the most common ones to be used in testing the databases, namely, sensor readings and multimedia data. Extensive tests have been performed under four popular databases: MySQL, MongoDB, CouchDB and Redis, with the focus on MongoDB versus MySQL. Besides, all the database servers were located on the cloud by using the Amazon EC2.

1.3 Structure

The rest of the thesis is organized as follows. Chapter 2 introduces the background information of databases, including their characteristics and classification. Chapter 3 explains the main concepts used in the thesis, that is, Internet of Things and cloud databases, and also reviews the related works. Chapter 4 describes the methodology and setup of the experiments performed to compare the performance of the different database systems considered. Chapter 5 presents the results and the evaluation of the experiments. Finally, Chapter 6 summarizes and concludes the work done, with directions for future research.

Chapter 2

Databases

This chapter gives an introduction about the two classes of databases presenting in the thesis: SQL and NoSQL databases. The chapter starts with a brief definition of CAP theorem, which has been used as the paradigm to explore the variety of distributed systems as well as database systems. Thereafter, SQL and NoSQL databases are presented along with the main differences between them. In the end, the chapter describes the main features of the four databases that are targeted in the performance tests.

2.1 CAP Theorem, ACID vs. BASE

The CAP theorem proposed by Eric Brewer [Bre00] states that a *shared data system* cannot guarantee all of the following three characteristics at the same time:

- ***Consistency*** means that once an update operation is finished, everyone can read that latest version of the data from the database. A system which not all the readers can view the new data right away does not have strong consistency and is normally *eventual-consistent*.

- ***A**vailability* is achieved if the system always provides continuous operation, normally achieved by deploying the database as a cluster of nodes, using replication or partitioning data across multiple nodes so if one node crashes, the other nodes can still continue to work.
- ***P**artition tolerance* means that the system can continue to operate even if a part of it is inaccessible (e.g. due to network connection, or maintenance purpose). This can be accomplished by redirecting writes and reads to nodes that are still available. This property is meaningless for a system of one single node though.

Most traditional RDBMSes were initially meant to be on a single server and focus on *Consistency*, thus having the so-called ACID properties [Bar10]:

- ***A**tomicity*: the transactions are all-or-nothing.
- ***C**onsistency* (different from **C** in CAP): the system stays in a stable state before and after the transaction. If a failure occurs, the system reverts to the previous state.
- ***I**solation*: transactions are processed independently without interference.
- ***D**urability* guarantees that committed transactions will not be lost. The database keeps track of the changes made (in logs) so that the system can recover from an abnormal termination.

Databases used for banking or accounting data are examples of systems where consistency is essential. However, there are ones that favour availability and partition-tolerance over consistency, for instance social networks, blogs, wikis, and other large scale web sites with high traffic and low-latency requirement. For such systems, it is hard to achieve ACID, and hence BASE approach is more likely to be applied:

- ***B**asic **A**vailability*
- ***S**oft-state*
- ***E**ventual consistency*

The idea is that the system does not have to be strictly available and consistent all the time, but is more fault-tolerant. Even though clients may encounter an inconsistent data as updates are in progress (during replication process), the data will eventually reach the expected consistent state.

Even though relational databases have been considered the classic kind of databases for years, NoSQL databases have been using the CAP theorem as an argument against the traditional ones. As system scale is getting larger, it is difficult to leave out the partition tolerance. In the end, the goal is to find the best combination of consistency and availability to optimize specific applications. The point of NoSQL databases is to focus on availability first then consistency, while SQL databases with the ACID properties go for the opposite direction.

2.2 SQL Databases

Back in the 1970s, *SQL* (Structured Query Language) was developed by IBM when Edgar Codd introduced the so-called *relational model* of data [Cod70]. Since then, SQL has become the standard query language for relational database management systems (RDBMS).

In the relational model [Cod70], data are organized into *relations*, each is represented by a *table* consisting of *rows* and *columns*. Each column represents an attribute of the data, the list of columns makes up the header of the table. The body of the table is a set of rows, one row is an entry of data which is a *tuple* of its attributes. Another important concept in the relational model is *key*, which is used to order data or to map data to other relations. *Primary key* is the most important key of a table, it is used to uniquely identify each row in the table.

To access a relational database, SQL is used to make queries to the database such as the CRUD basic tasks of Creating, Reading, Updating and Deleting data. SQL supports indexing mechanism to speed up reading operations, or creating views which can join data from multiple tables, and other features for database optimization and maintenance. There are many relational databases available, such as MySQL, Oracle, SQLServer, and all are using SQL. Although the concrete syntax for each database can be slightly different, switching from one to another does not require a significant change in system programs.

One important attribute of SQL databases is that they follow the ACID rules to ensure the reliability of data at any point of time. This is one of the key differences between SQL and NoSQL databases. To achieve the data integrity, SQL databases usually support isolated transactions, with two-phase commit and roll back mechanism [FL05]. This feature, however, contributes to the processing overhead. The normal sources of processing overhead are [Vol10]:

- *Logging*: To ensure system durability and consistency, SQL databases write everything twice, once to the database itself and once to the log, so

the system can recover from failures.

- *Locking*: Before making a change to a record, a transaction must set a lock on it and the other transactions can not interfere before the lock is released.
- *Latching*: A latch can be understood as a "lightweight, short-term lock" used to prevent data from unexpected modification. However, while locks are kept during the entire transaction, latches are only maintained during the short period when a data page is moved between the cache and the storage engine.
- Besides, *index* and *buffer* management also require significant CPU and I/O operations, especially when used on shared data structures (e.g., index B-trees, buffer pool). Hence, they also cause processing overhead.

Originally designed to focus on data itegrity, relational databases are nowadays facing challenges of scaling to meet the growing data volume and workload demand.

2.3 NoSQL Databases

NoSQL (not only SQL) is another type of DBMS that can be used on the cloud. Different from SQL databases, NoSQL databases do not divide data into relations, nor do they use SQL to communicate with the database.

2.3.1 NoSQL Properties

When it comes to NoSQL definition, it is likely that SQL is put into perspective. That is not only because SQL is widely considered as the traditional and popular type of database, but also the origin for NoSQL movements is to eliminate the weak points of relational databases. Below is the main characteristics of common NoSQL databases, which reflect the motivations for the rise of such databases and how they are different from relational ones.

Non-relational

The types of NoSQL databases are various, including *document*, *graph*, *key-value*, and *column family* databases, but the common point is that they are *non-relational*. Jon Travis, a principal engineer at Java toolmaker SpringSource,

said “Relational databases give you too much. They force you to twist your object data to fit a RDBMS” [Lai09]. The truth is the relational model only fits a portion of data, many data need a simpler structure, or a flexible one. For example, a database is built to store student information and the courses that each student takes. A possible design in the relational model for this data is to have one table for student, one for course, and one that maps a student with his courses (Figure 2.1).

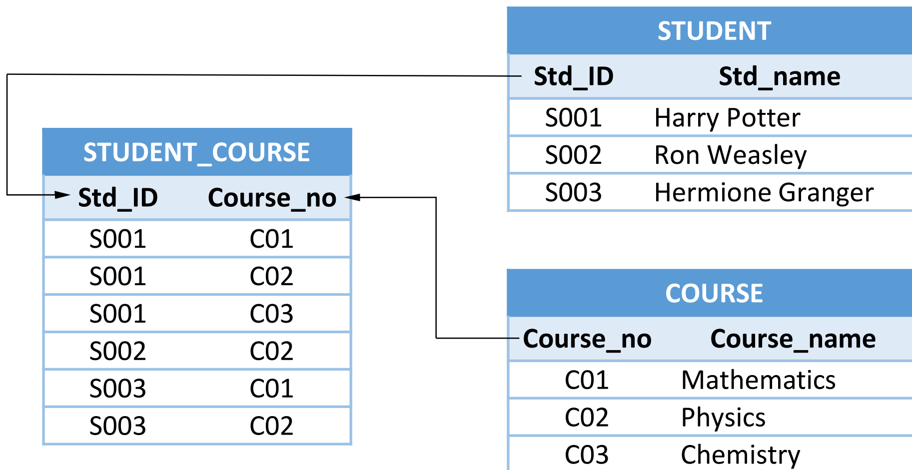


Figure 2.1: SQL database - student example

One problem with this design is that it contains extra duplicated data, in this case the mapping table *STUDENT_COURSE* repeats the *Std_ID* multiple times for each different course. NoSQL approach, however, is flexible enough to map one student with a list of courses in only one record without this duplicated data. Figure 2.2 shows the solution using a document-store database.

In fact, NoSQL databases generally do not have many limitations on data structure. Apart from the normal primitive types, more data types can be supported, for instance, nested documents or multi-dimensional arrays. Unlike SQL, each record does not necessarily hold the same set of fields, and a common field can even have different types in different records. Hence, NoSQL databases are meant to be schema-free and suitable to store data that is simple, schema-less, or object-oriented [SSK11]. This seems to be the case in many current applications. For example, for a library database, each item can have a different schema, depending on the type of item. In this case, it might be a good idea to follow a flexible schema-less design instead of creating an SQL table with all the possible columns and not using all of them for an item.

COURSE
No : C01 Name: Mathematics
No : C02 Name: Physics
No : C03 Name: Chemistry

STUDENT
Std_ID : S001 Name: Harry Potter Courses: {C01, C02, C03}
Std_ID : S002 Name: Ron Weasley Courses: {C02}
Std_ID : S003 Name: Hermione Granger Courses: {C01, C02}

Figure 2.2: NoSQL database - student example

- *Book*: Author(s), Book title, Publisher, Publication date.
- *Journal*: Author(s), Article title, Journal title, Publication date.
- *Newspaper*: Author(s), Article title, Newspaper name, Section title, Publication date.
- *Thesis*: Author, Thesis title, School, Supervisor, Instructor, Date.

Hence, NoSQL databases can handle *unstructured* data (e.g., email body, multimedia, metadata, journals, or web pages) more easily and efficiently. Moreover, the benefit of a schema-free data structure also stands out when it comes to data of *dynamic* structure. Since it is costly to change the structure of relational tables[1], how data will change (e.g., form and size) over time should be taken into consideration. However, relational or non-relational also depends on the kind of queries to be performed. Continue the example of students and courses, we want to add a new field for *grade*. Figure 2.3 shows two possible solutions using SQL and NoSQL databases.

In this example, if the user wants to query the average grade of all students together, that is one simple work for the SQL table, which only works on one column *grade* and gets the average value of all grades. Meanwhile, the operation will be much more complicated with the nested layers in NoSQL collection. On the other hand, if the system only serves displaying the data, meaning listing the courses and grades for each student (including student name) then the opposite

[1]SQL tables store data as one row after another. If a new column is added, there will be no space for it. Consequently, the entire table needs to be copied to a new location, and for the time of the copying, the table is locked.

SQL_GRADE		
Std_ID	Course_no	Grade
S001	C01	3
S001	C02	5
S001	C03	4
S002	C02	2
S003	C01	3
S003	C02	5

SQL_STUDENT	
Std_ID	Std_name
S001	Harry Potter
S002	Ron Weasley
S003	Hermione Granger

NoSQL_GRADE
Std_ID : S001 Name: Harry Potter Grades: { {C: C01, G: 3}, {C: C02, G: 5}, {C: C03, G: 4}}
Std_ID : S002 Name: Ron Weasley Grades: { {C: C02, G: 2}}
Std_ID : S003 Name: Hermione Granger Grades: { {C: C01, G: 3}, {C: C02, G: 5}}

Figure 2.3: NoSQL vs SQL database - student example

is true. In this case, SQL database has to perform a JOIN query, which is an expensive operation, on table *grade* and *student* to get the student name. As said by Curt Monash, a blogger and database-analyst, "SQL is an awkward fit for procedural code, and almost all code is procedural. For data upon which users expect to do heavy, repeated manipulations, the cost of mapping data into SQL is well worth paying...But when your database structure is very, very simple, SQL may not seem that beneficial"[Lai09].

Horizontal scalability

Most classic RDBMSes were initially designed to run on a single large server. Joining data over several servers is a difficult work that makes it uneasy for relational databases to operate in a distributed manner [Lea10]. The idea of "one size fits it all", however, is not feasible to fulfill current demand. A better idea is to partition data across multiple machines.

Unlike SQL databases, most NoSQL databases are able to scale well horizontally and thus not relying much on hardware capacity. Cluster nodes can be added or removed without causing a stop in system operation. This can provide higher availability and distributed parallel processing power that increase performance, especially for systems with high traffic. Some NoSQL databases can provide

automatic *sharding*[2] (Section 3.2.2). For example, MongoDB [mon13] can auto shard data over multiple servers and keep the data load balanced among them, thus distributing query load over multiple servers.

Availability over Consistency

One main characteristic of SQL databases is that they conform to ACID rules (Section 2.1), which mainly focus on consistency. Many NoSQL databases have dropped ACID and adopted BASE. That is to compromise consistency for higher availability and performance. Applications used for bank transactions, for example, require high reliability and therefore, consistency is vital for each data item. However, in some cases, that merely complicates and slows down the process unnecessarily. Social network applications such as Facebook do not require such high data integrity. The priority here is to be able to serve millions of users at the same time with the lowest possible latency. One method to reduce query response time for database systems is to replicate data over multiple servers, thus distributing the load of reads on the database. Once a data is written to the master server, that data will be copied to the other servers. An ACID system will have to lock all other threads that are trying to access the same record. This is not an easy job for a cluster of machines, and will lengthen the delayed time. BASE systems will still allow queries even though the data may not be the latest. Hence, it can be said that NoSQL databases drop the expense for data integrity when it is not highly necessary to trade for better performance.

Map Reduce model

Relational databases put computation on reads. For large scale applications, that will cause long delays for responses. NoSQL databases, however, normally do not provide or avoid complex queries (e.g., join operations). While SQL databases all use SQL as their query language, NoSQL databases are so different that there is no such common API among them. Nevertheless, many NoSQL databases adopt Google's *Map-Reduce* model [DG08] in querying. The model provides an effective method for big data analysis. It supports parallel and distributed processing on clusters of nodes. The main idea is to divide the computation work into smaller sub-problems, distribute them to smaller nodes (map), then aggregate individual results into a final one (reduce). This is suitable for sensor data analytic, for example. Generally, sensor data structure is repetitive and the typical computations are linear, such as sum, average, min, and max.

In the end, what makes NoSQL differ from SQL is its flexibility and variety. Applications for business intelligence, e-commercial, document processing, or

[2] sharding: horizontal partitioning data across a number of servers

social network go with different data schemas and have different requirements for consistency, performance, and scalability. NoSQL with various capabilities and purposes gives users more choices to pick the most suitable database that meets their needs. Numerous companies have chosen NoSQL over the rich but unnecessary SQL platform as their solution. Many NoSQL databases were initially built as a specialized tool, and later released as open source. For instance, Facebook first developed Cassandra data store for their Inbox Search feature. The motivation was to build a highly available data store that can handle large data and process a lot of random reads and writes. According to Facebook engineers Avinash Lakshman et al., "No existing production ready solutions in the market meet these requirements", and Cassandra can write 50GB data in 0.12 milliseconds, that is 2500 times faster than MySQL does [LMR08].

Table 2.1 summarizes the main differences between general SQL databases and NoSQL databases.

SQL	NoSQL
Relational model	Non-relational data (schema-less,unstructured,simpler)
Tables	Key-value, Document, Graph, Column family stores
ACID	BASE
Consistency	Availability, Performance
Single server	Cluster of servers (Horizontal scalability)
SQL query	Simpler and different API

Table 2.1: SQL vs NoSQL

2.3.2 NoSQL Categories

NoSQL databases can be classified into four major categories [Tiw11]:

- Key-Value stores
- Document stores
- Column Family stores
- Graph databases

This thesis, however, just focuses on the first two types.

Key-Value stores

This is the simplest kind of NoSQL databases (in term of API). As its name, key-value databases [Tiw11] store data in pairs of *key* and *value*. The value is just a block of data of any type and any structure. No schema needs to be defined, but the user defines the semantics for the values and how to parse the data himself. The advantage of key/value stores is that it is simple to build, easy to scale, and tends to have good performance.

Basically, the way to access data in a key/value database is by the key. The basic API to manipulate data are:

- *put(key, value)*
- *get(key)*
- *remove(key)*

Figure 2.4 is an example of a key-value data structure. Database *Student* consists of a list of student information, identified by *student ID*.

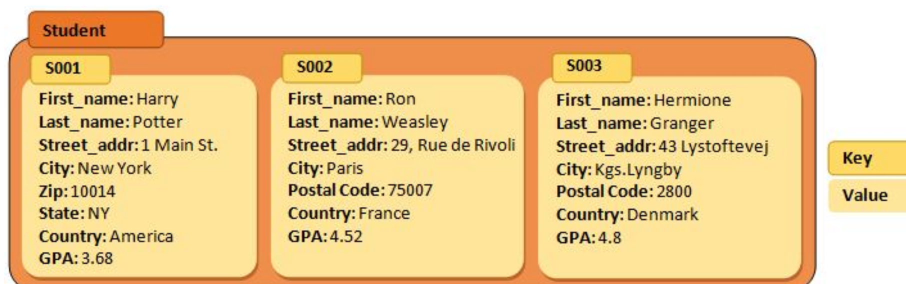


Figure 2.4: Key-Value stores - student example

Examples of available key-value stores are Redis [red13], Project Voldermort [pro13], Amazon Dynamo [Voe12]. If an application fits this data structure, for example Amazon's shopping carts and user sessions, and its major query is key lookup, then significant performance benefits can be achieved. That is because key lookup can be highly enhanced by using *hash* or *tree*. Besides, queries are easy to handle (one request to read, one to write), and so are conflicts (only one single key to be resolved).

Document stores

A document database [Tiw11] is a higher step of a key/value store where a database is a collection of *document*. Each document consists of multiple *named fields*, one of which is a unique *documentID*. A named field is actually a key-value pair where the *key* is the *name* of the field. Document databases are schema free. The data can be of any structure and different among each document. Therefore, it allows users to store arbitrarily data, from primitive types such as strings, numbers, dates to more complex data such as trees, dictionaries, or nested documents. However, it should be noted that the field name of the same field will be repeated in multiple documents, so one good practice is to make the field name as short as possible to save storage space.

Unlike key/value stores, the content of the document (the value) is not just a block of data. Documents are normally stored in a specific format, which can be XML, JSON, or BSON. With such format, the server supports not only simple key-value lookup but also queries on the document contents. Besides, the known format also makes it easier to build tools to display and edit the data. Examples of document-store databases are CouchDB [cou13], and MongoDB [mon13].

The *Student* database example shown in Figure 2.4 is converted into a document store, as in Figure 2.5.

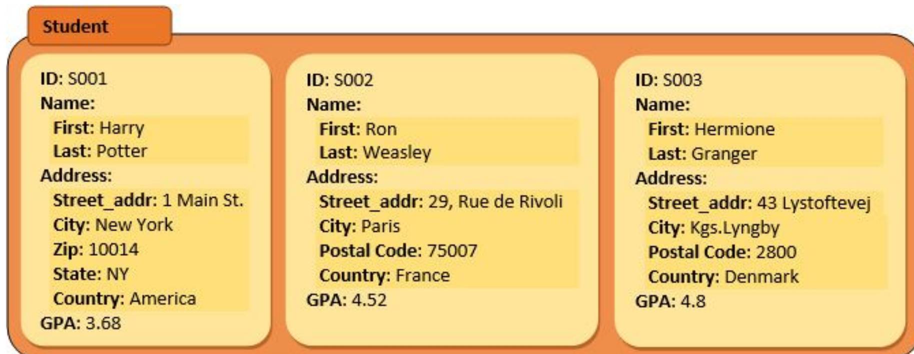


Figure 2.5: Document stores - student example

Document stores can easily be the most popular among developers. The document format can map nicely to programming language data types. Complicated join operations can be avoided thanks to the use of embedded documents, reference documents, and arrays. At the same time, it still provides rich query capability and high scalability.

2.4 Tested databases

This section gives a description about the particular databases that are to be tested. Their performance will be recorded and compared.

2.4.1 MySQL

MySQL [MyS13a] is the most popular open-source SQL database in business currently. The database was developed by MySQL AB, now owned by Oracle.

SQL statements

As a typical relational database, MySQL organizes data in the relational model with tables, rows and columns, and uses SQL to access databases. MySQL provides a very rich set of other statements for manipulating data. The basics are INSERT, SELECT, UPDATE, DELETE, which correspond to the CRUD operations. Besides, MySQL supports other functionalities such as *join*, *group by* and *views* for data aggregation over multiple tables; or *stored procedures*, *functions*, *triggers*, and *events* that can be run according to schedule or user's requests.

Take advantage of the fact that database applications often process a lot of similar statements repeatedly, MySQL provides server-side *prepared statements*. These statements only need to compile once while different values for the parameters can be passed each time the statement is executed. If properly used, it can help to increase efficiency.

Buffering and Caching

MySQL uses *storage engines* to store, handle, and get data from database tables. MySQL supports different storage engines, which have different features and performance characteristics. *InnoDB* is the default for versions after 5.5. InnoDB is ACID compliant. It supports *transactions* with commit, roll back, crash-recovery, and foreign key constraints to maintain data integrity.

InnoDB uses *buffer pool* to cache data and indexes in memory, thus improving performance. A buffer pool is a linked list of pages, keeping heavily accessed data at the head of the list by using a variation of the least recently used (LRU) algorithm. To prevent bottleneck as multiple threads access the buffer pool at once, users can enable multiple buffer pools to the maximum of 64 instances.

Additionally, MySQL uses a *query cache* to store SELECT statements and their results. If the same statement is queried again, the result will be retrieved from the cache rather than being executed again. The query cache is shared among sessions. If a table is modified, all cached queries using the table will be removed.

Indexes

Instead of searching through the whole table, users can create indexes on a single or multiple columns in a table to increase query performance. InnoDB supports the following types of index and stores the indexes in B-trees:

- Normal Index: the basic type of index.
- Unique Index: all values must be different (or null).
- Primary Key: all values must be unique and not null.
- Fulltext Index: used in full-text searches.

Each InnoDB table has a *clustered index* where the rows are actually stored. The clustered index is the primary key if there is one, which means data is physically sorted by the primary key. If primary key is not specified, InnoDB chooses an unique index where all values are not null. If there is no such unique index, InnoDB generates a hidden index on a synthetic ID column, where the ID is incremented as insertion order.

Indexes that are not clustered index are *secondary indexes*. Except from the columns defining the index, each secondary index record includes the primary key columns as well.

Compared to secondary indexes, queries by the clustered index has optimal performance because searching through the index means searching through the physical pages where the real data reside, while with the other indexes, data and index records are stored separately.

Replication

Replication in MySQL follows the *master-slave* model. Changes in the master will be recorded to a *binary log* as *events*. Each slave receives a copy of the log and continues reading and executing the events. The slaves do not need to connect to the master permanently. Each will keep track of the position before which the log has been processed and so the slave can catch up with the master whenever it is ready. Besides, users can configure the master to specify which

databases to write to this log, and configure each slave to filter which events from the log to execute. Hence, it is possible to replicate different databases to different slaves.

Replication in MySQL is *asynchronous* by default which means the master does not know when the slaves get and process the binary log. Nevertheless, *semi-synchronous* replication can be enabled on at least one slave. In this case, after a transaction has been committed on the master, the thread blocks and waits until receiving a receipt from the slave indicating that the binary log has been copied to the slave.

MySQL does not provide an official solution for *auto failover* between master and slaves. That means in case of failure, the user is responsible for checking whether the master is up, and switching the role to a slave.

Sharding

MySQL supports *partitioning* an individual table into portions, and then distributing the storage to multiple directories and disks. As a result, queries can be performed on a smaller set of data. This might also help to reduce I/O contention as multiple partitions are placed on different physical drives.

For MySQL, *sharding* is external to the database. Auto-sharding is supported by MySQLCluster [MyS13b]. However, basic MySQL does not provide an official sharding feature. An alternative is to perform sharding at *application level*. The approach works by having multiple databases of the same structure in multiple servers, and dividing the data across these servers based on a selected *shard key* (a set of columns of the table). The application is in charge of coordinating data access across multiple shards, directing read and write requests to the right shard. This approach, however, adds a lot of complexity to database development and administration work. First, it is a difficult job to manually ensure load balance between the shards. Second, MySQL features that are to ensure data integrity such as foreign key constraints or transactions are incapable across multiple shards. Additionally, horizontal queries (such as sum or average) that need to be resolved against all of these nodes can have a significant latency as data access time increases along with the number of nodes. MySQL does not have a proper asynchronous communication API (such as MapReduce) that can parallelize the operation and aggregate the results. Consequently, implementation can be highly complicated and unsafe with a lot of forking and connections in the child processes.

Sharding can also be done at *MySQL Proxy layer*. MySQL Proxy is an application that is placed between MySQL servers and clients, able to intercept and direct queries to a specific server. However, MySQL Proxy is currently at Alpha

version and not used within production environment.

2.4.2 CouchDB

CouchDB [ALS10] is a NoSQL database. It is an open source project done by Apache, written in Erlang and first released in 2005.

Document-store

CouchDB falls on the category of document-stores. A CouchDB database is a set of documents which are schema-free. Data is stored in JSON format [Cro06], which is a lightweight, human readable format. Each document is basically a collection of key-value pairs (fields), including a unique *_id* field. If the _id is not explicitly specified by the user, the database will automatically generate one. Arrays and nesting are supported in the documents.

RESTful API

CouchDB was designed as "A Database for the Web" [cou13]. The database provides a RESTful API [Rod08], that is to use HTTP methods POST, GET, PUT, and DELETE for the four CRUD operations on data. Hence, users can access data using a web browser. Web applications can also be served directly from a CouchDB database.

MVCC

CouchDB implements Multiversion concurrency control (MVCC) method [BHG87] to manage concurrent access to the database. That is for each write on a data item (insert or update), the system creates a new *version* of that item. Hence, in each document there is a *_rev* field that stores the revision number. All revisions of a document will be kept even if the document is deleted, and users can retrieve any version they ask for. That way, the system can avoid the need to use locks, operations can be performed in parallel, thus increasing speed. The system provides automatic conflict detection, it stores all the versions of the concerned document and marks it as *conflicted*. It is then up to the application to handle the conflicts. However, one major drawback of this approach is the growth in storage space.

Querying by views

CouchDB does not support adhoc queries. Data is queried using *views* (except for single query-by-IDs which can be a GET HTTP request). Each view is de-

fined with Javascript functions, using MapReduce paradigm. View definition is stored in a special document called *design* document. However, this view mechanism can put more burden on the programmers than normal query language, and the storage needed is increased to store the view indexes.

CouchDB uses append-only B+ trees [Hed13] to store documents and view indexes, with the idea of trading space for speed. The views are updated on read requests. In fact, all views in the same design document are indexed as a group, and so they will be updated together even though only one is accessed. The first time the view is read, CouchDB takes some time to build the B-tree. On subsequent reads, it will check for the changed documents (using revision numbers) and update the view indexes incrementally. As a result, the more changes there are, the longer the view query takes. Since CouchDB keeps all data versions, the changes made by insert, update or even delete operations will be appended to the database file, this also applies for view files. The result is that the data files grow constantly. In this case, *compaction* can be run, which removes all the old revisions and deleted documents. The procedure can be configured to run periodically or when the database file exceeds a threshold.

Bulk Document Inserts

CouchDB provides a bulk insert/update feature via the *_bulk_docs* endpoint. This is the fastest way to import data into the database. Users can send a collection of documents in a single POST request and only one single index operation needs to be done. With CouchDB usage of append-only B+ tree, this also means saving a lot of storage space.

Consistency

ACID properties is ensured by the system. Since data is stored in an append-only B-tree, the existing data is never overwritten and stays stable. The changes are added to the end of the database file, then added a file footer (with a checksum) storing the new length twice. This keeps the database file robust in case of corruption. If there is a failure when flushing data to disk, the old length will be kept and the system stays as before the update.

Scalability

Scalability in CouchDB is achieved by *incremental replication*. The changes can be periodically copied among servers, or when a device turns back to online after offline time (*multi-master* replication). Hence, the database is *eventually consistent*. This makes CouchDB a good choice to be used as a mobile embedded database, also for the fact that CouchDB does not cache anything internally (although it can make use of the file system cache, for example, when loading

the B-trees).

However, basic CouchDB does not support sharding. If users want to partition their data, they need to do it manually, or using a project called CouchDB Lounge [ALS10], which provides sharding on top of CouchDB.

2.4.3 MongoDB

MongoDB [mon13] is an open source document store database, developed by 10gen and written in C++. The database is meant to work with large amount of data, thus being scalable and fast.

Document-store

As CouchDB, MongoDB is a document-oriented database, meaning that its data has a flexible schema. The database contains multiple *collections*, each in turn contains multiple documents. In practice, the documents in a collection normally have similar structure, representing one kind of application-level object.

Data is stored in BSON format, which is a binary-encoded format of JSON. The format makes the data easily parsable as JSON, highly traversable, fast to encode and decode [bso13].

MongoDB does not keep different versions of the data. Therefore, there is no *_rev* field needed as in CouchDB. Each document in a collection is identified by an unique *_id*. If the user does not assign a value to *_id*, the system will automatically generate it with an *ObjectID*. ObjectID is 12 bytes, structured as shown in Figure 2.6.

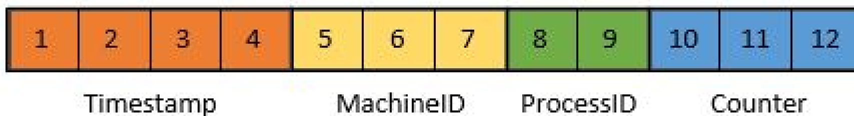


Figure 2.6: MongoDB ObjectID

Querying

Unlike CouchDB, MongoDB supports a very rich set of *adhoc* queries. Users do not need to write MapReduce functions for simple queries. It comes with a

JavasScript shell (the *mongo* shell) which is actually a stand-alone MongoDB client that can interact with the database from the command line.

CRUD operations are executed with the commands *insert, find, update, remove* respectively. There is no need for an explicit *create* command for databases and collections, they are automatically created once the collection is referred to. MongoDB supports search by fields, range of values, and regular expressions. Users can choose which fields to be returned in the result. The results are returned in batches, through *cursors*. A cursor is automatically closed after some configured time, or once the client iterates to its end.

For aggregation tasks, clients can use either *MapReduce* operations or the simpler *aggregation framework* which is similar to GROUP BY in SQL.

Indexes

Indexes in MongoDB are on a per-collection level. MongoDB automatically creates an unique index on the *_id* field. It also supports secondary index, which means users can create indexes on any other fields in the documents, including compound index, index on sub-document, and index on sub-document fields.

Capped Collections and Tailable Cursor

A capped collection is a fixed-size collection that works similarly to a circular buffer. Data are stored on disk in the insertion order. Therefore updates that increase document size are not allowed. When the space for the collection runs out, the round turns over and the new documents automatically replace the oldest ones. Hence, capped collections are suitable for queries based on insertion order. That is analogous to *tail* function to get the most recently added records, for example for logging service. Because of its natural order, capped collection cannot be sharded. Different from normal collections, capped collections require an explicit *create* command in order to preallocate the space. The command can be time consuming, but it is only needed in the first run.

Capped collections allow the use of *tailable cursors* which stay open even after the cursors have been exhausted. If there are new documents added, the cursors will continue to retrieve these documents. Tailable cursors do not use indexes. Therefore, it might take some time for the initial scan, but subsequent retrievals are inexpensive.

GridFS

While MySQL uses BLOB data type, MongoDB provides *GridFS* to store and retrieve data files of large size. The GridFS database structure is shown in

Figure 2.7. A GridFS *bucket* (default named *fs*) comprises of two collections: *files* collection stores the file metadata, and *chunks* collection stores the actually binary data, divided into smaller chunks. This approach makes storing the file easier and more scalable, also possible for range operations (such as getting specific parts of a file).

MongoDB GridFS	
fs	
fs.files	**fs.chunks**
"_id" : <ObjectID>	**“_id”**: <ObjectID>
"length" : <num> (file size in KiB)	**“files_id”:** <string> (_id of the “parent” *file* document)
"chunkSize" : <num> (default 256 KiB)	**“n”**: <num> (sequence number of the chunk)
"uploadDate" : <timestamp>	**“data”**: <binary> (the chunk’s payload)
"md5" : <hash>	
"filename" : <string> (optional)	
"contentType" : <string> (optional)	
"aliases" : <string array> (optional)	
"metadata" : <dataObject> (optional)	

Figure 2.7: GridFS structure

Storage

MongoDB does not implement a query cache but it uses *memory mapped files* for fast accessing and manipulating data. Data are mapped to memory when the database accesses it, thus being treated as if they are residing in the primary memory. This way of using operating system cache as the database cache yields no redundant cache. Cache management is, therefore, different depending on the operating system. MongoDB automatically utilizes as much free memory on the machine as possible [Tiw11]. Hence, the database is at its best performance if the working set can fit in RAM.

Data are stored in several preallocated files, starting from 64 MB, 128 MB and so on, up to 2 GB, after that all files are 2 GB. That way small databases do not take up so much space while preventing large databases from file system fragmentation. Hence, there can be space that is unused but for large databases, this space is relatively small.

Consistency

MongoDB is not ACID compliant but *eventually consistent*. It writes all update operations to a write ahead logging called *journal*. If an unexpected termination occurs, MongoDB can re-run the updates and maintain the system in a consis-

tent state. By default, changes in memory are flushed to data files once every minute. Users can configure a smaller sync interval to increase consistency with the expense of decreased performance.

Sharding

MongoDB offers *automatic sharding* as a solution for horizontal scaling. Sharding is enabled on a per-database basis. It partitions a collection and distributes the partitions to different machines. Data storage is automatically balanced across the shards.

Data are divided according to the ranges of a *shard key*, which is a field (or multiple fields) existing in all the documents in the collection. In each partition (or *shard*), data are divided further into *chunks*. Chunk size can be specified by users. Small chunks lead to a more even data distribution while large chunks limit data migration during load balancing. The choice of the shard key can directly affect the performance. The shard key should be easily divisible, likely to distribute write operations to multiple shards, but route the search queries to a single one (query isolation). Queries that do not involve the shard key will take longer time as it must query all shards.

A minimal shard cluster includes:

- Several *mongod*[3] server instances, each serves as a shard.
- A *mongod* instance to become a *config server*, maintaining the shard metadata.
- A *mongos*[4] instance acts as a single point of access to a sharded cluster. It appears as a normal single MongoDB server instance.

The *mongos* instance receives queries from clients, then uses metadata stored in the *config server* to route the queries to the right *mongod* instances.

Replication

Replication in MongoDB is used to provide backup, distributing read load, and automatic failover. Replication copies data to a group of servers, forming a *replica set*. A replica set is a cluster of two or more *mongod* instances, one is the (only) *primary*, the others are *secondary* instances. Write operations can only be

[3] mongod: the primary daemon process for the MongoDB system, handling data requests and background management operations [mon13].

[4] mongos: provides routing service for MongoDB shard clusters.

performed on the primary, data will then be copied to the secondaries. For read operations, users can choose a preference to read from primary or secondaries or the nearest machine. In case the primary is unreachable, one secondary will be automatically chosen to become the new primary. This process is called *failover*. This way, MongoDB can provide high availability.

In production, the system usually combines both replication and sharding to increase reliability, availability, and partition tolerance. Figure 2.8 shows an example of a system architecture in practice. The system provides no single point of failure with multiple points of access, data are partitioned across three shards, each is a replica set.

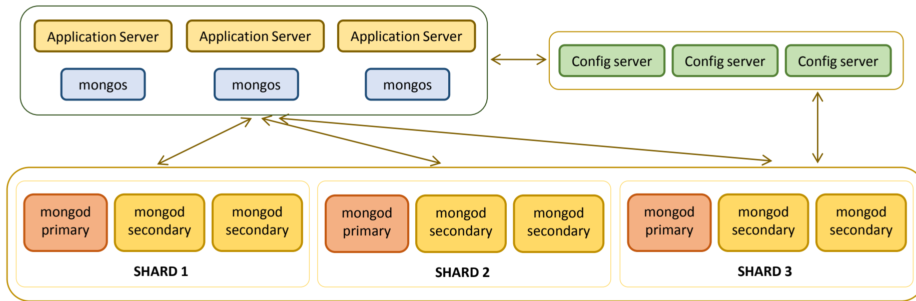


Figure 2.8: Scalable system architecture of MongoDB

2.4.4 Redis

Redis [Seg10] is an opensource in-memory key-value store. The database promises very fast performance, and more flexibility than the basic key-value structure.

Data model

In Redis, a database is identified by a number, the default database is number 0. The number of databases can be configured but default is 16 databases. Basically, a Redis database is a *dictionary* of *key* and *value* pairs. Nevertheless, apart from the classic key-value structure where *value* is a string and users are responsible to parse it at the application level, Redis offers more choices of data structures, where a value can be stored as:

- A *string*
- A *list* of strings: Insertions at either the head or tail of the list are sup-

ported. Besides, querying for items near the two ends of the list is extremely fast, while querying for one in the middle of a long list is slower.

- A *set* of strings: This is a non-duplicated collection of strings which means adding the same string repeatedly yields only one single copy. Add and remove operations only take constant time (O(1)).
- A *sorted set* of strings: Similar to *set* but in a *sorted set*, each string is associated with a *score* specified by clients. This score is used as the criteria for sorting and can be the same among multiple members of the set.
- A *hash*: In this case, each value itself is a map of *fields* and *values*. This data type is very useful for representing objects. For example, a *student* object will have multiple fields, for example, *name*, *age*, and *GPA*.

Querying

Each type of data structures has its own set of commands available [red13]. Redis does not support secondary index, all queries are based on the *keys*, which means it is impossible to query for students that are at the age of 20 (i.e., *students* whose value of field *age* is 20).

Persistence

To achieve high performance, Redis stores the entire data set in memory. However, an obvious drawback is that this makes Redis depend highly on RAM and limits the database storage capacity, as RAM is an expensive piece of hardware.

On the other hand, Redis persists data on disk as well. Hence, the dataset can be reloaded to the memory at server startup. Nevertheless, data persistence can be disabled in case users only need to keep the data while the server is active, for instance, for cache purpose.

There are two main methods for data persistence, that is, by *snapshots* or by *append-only file*, or a combination of the two. As regards snapshots which is the default option, Redis can be configured to save dataset snapshots periodically if a specified number of keys changed. For example, the configuration *save 300 10* will automatically save the database after 300 seconds if at least 10 keys have changed. A disadvantage of this approach is the weak durability, for data can be lost before the snapshot is taken if a sudden termination occurs. The alternative is to use the append-only file that logs all the write operations. However, the append-only file is normally bigger than the snapshot file, plus it can be slower depending on how often the file is configured to be dumped on disk.

Scalability

Redis databases can be replicated using the *master-slave* model. However, it does not support automatic failover, which means if the master crashes, a slave has to be manually promoted to replace it. A slave can have other slaves of its own, so it can also accept write requests, though a slave is in read-only mode by default.

At the time being, sharding is not officially supported, although it is provided by some particular drivers. Nevertheless, a project called Redis Cluster is now being developed which promises horizontal scalability along with other useful features for a distributed Redis system.

Chapter 3

Cloud databases for the Internet of Things

This chapter explains the two main concepts of the thesis: *Internet of Things* and *cloud databases*. Additionally, it gives an overview of the previous related work done on the topic of Cloud Databases for IoT data.

3.1 Internet of Things

The phrase "*Internet of Things*" started life in 1999 by Kevin Ashton, co-founder and executive director of Auto-ID Center [SGFW10].

> "*Internet of Things (IoT) is an integrated part of Future Internet and could be defined as a dynamic global network infrastructure with self configuring capabilities based on standard and interoperable communication protocols where physical and virtual things have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network.*"

To make it simpler, IoT refers to a world of physical and virtual objects (*things*) which are uniquely identified and capable of interacting with each other, with people, and with the environment. It allows people and things to be connected at *anytime* and *anyplace*, with *anything* and *anyone*. Communication among the things is achieved by exchanging the data and information sensed and generated during their interactions.

3.1.1 Internet of Things vision

The broad future vision of IoT is to make the things able to react to physical events with suitable behavior, to understand and adapt to their environment, to learn from, collaborate with and manage other things, and all these are autonomous with or without direct human intervention. To achieve such a goal, numerous researches have been carried out, which emphasize on different aspects of the IoT. The followings are the three main concrete visions of the IoT that most of the researches are focusing on [AAS13] [AIM10]:

Things-oriented Vision

Originally, the IoT started with the development of *RFID* (Radio Frequency Identification) tagged objects that communicate over the Internet. RFID along with the *Electronic Product Code* (EPC) global framework [TAB+05] is one of the key components of the IoT architecture. The technology targets a global EPC system of RFID tags that provide object identification and traceability.

However, the vision is not limited to RFID. Many other technologies are involved in the things-vision of IoT, including *Universally Unique IDentifier* (UUID) [LMS05], *Near Field Communications* (NFC) [Wan11], and *Wireless Sensor and Actuator Networks* [VDMC10]. Those in conjunction with RFID are to be the core components that make up the Internet of Things. Applying these technologies, the concept of *things* has been expanded to be of any kind: from human to electronic devices such as computers, sensors, actuators, phones. In fact, any everyday object might be made *smart* and become a thing in the network. For example, TVs, vehicles, books, clothes, medicines, or food can be equipped with embedded sensor devices that make them *uniquely addressable*, be able to collect information, connect to the Internet, and build a network of networks of IoT objects.

Internet-oriented Vision

A focus of the Internet-oriented vision is on the *IP for Smart Objects* (IPSO) [VD10] which proposes to use the Internet Protocol to support smart objects

connection around the world. As a result, this vision poses the challenge of developing the Internet infrastructure with an IP address space that can accommodate the huge number of connecting things. The development of IPv6 has been recognized as a direction to deal with the issue.

Another focus of this vision is the development of the *Web of Things* [GT09], in which the Web standards and protocols are used to connect embedded devices installed on everyday objects. That is to make use of the current popular standards such as URI, HTTP, RESTful API to access physical devices, and integrate those objects into the Web.

Semantic-oriented Vision

The heterogeneity of IoT things along with the huge number of objects involved impose a significant challenge for the interoperability among them. Semantic technologies [BWHT12] have shown potential for a solution to represent, exchange, integrate, and manage information in a way that conforms with the *global* nature of the Internet of Things. The idea is to create a standardized description for heterogeneous resources, develop comprehensive shared information models, provide semantic mediators and execution environments [Sen10], thus accommodating semantic interoperability and integration for data coming from various sources.

3.1.2 Internet of Things data

With its powerful ability, the scope of the Internet of Things is wide. It can provide applicability and profits for users and organizations in a variety of fields, including environmental monitoring, inventory and product management, customer profiling, market research, health care, smart homes, or security and surveillance [MSPC12]. For instance, digital billboards use face recognition to analyze passing shoppers, identify their gender and age range, and change the advertisement content accordingly. A smart refrigerator keeps track of food items' availability and expiry date, then autonomously orders new ones if needed. A sensor network used to monitor crop conditions can control farming equipments to spray fertilizer on areas that are lack of nutrients. Examples for such applications of IoT are countless. Therefore, the types of data transmitted in the Internet of Things are also unlimited. It could be either discrete or continuous, input by humans or auto-generated. Generally, IoT data include, but not limited to, the following categories [CJ+09][CLR10].

RFID Data. Radio Frequency Identification [Wan06] systems are said to be a main component of the IoT [AIM10]. The technique uses radio wave for

identification and tracking purposes. An RFID tagging system includes several RFID tags that are uniquely identified and can be attached to everyday objects. The tag can store information internally and transmit data as radio waves to an RFID reader through an antenna. Hence, the technology can be used to monitor objects in real time. For example, it can replace bar codes in supply chain management, stock control, or used to track livestock and wildlife. In healthcare, VeriChip [GH06] is an RFID tag that can be injected under human's skin. It is used to biometrically identify patients and provide critical information about their medical records.

Sensor Data. Sensor networks [ASSC02] have been widely spread nowadays from small to large scale. They are also a key component in the Internet of Things. Their usage varies from recording and monitoring environment parameters or patient conditions in real time to tracking customer behavior and other applications. Several common parameters are temperature, power, humidity, electricity, sound, blood pressure, and heart rate. Data format can also be different, from numeric or text based to multimedia data. For this data type, the normal question is how often the data is to be captured, whether continuously, periodically, or when queried. In any case, the result could be an enormous volume of data, which in turn raises a challenge of storage as well as how to do querying, data mining, and data analysis on such an amount of data with a real-time demand. Additionally, sensor data generation tends to be continuous. As time goes by, some data become old and less valuable. Hence, the system is responsible to decide which data to keep, when to remove or archive old data, or how to distribute new data to active data warehouses used for frequent querying.

In the thesis, one of our focus is on sensor scalar data. The context for the sensor data benchmark is based on the Home Energy Management System (HEMS) developed by There corporation [the13]. The system uses smart metering sensors to monitor the electric energy consumption of households. The energy is periodically measured and recorded data are sent to a central database. Customers can then get the real time report about the energy usage in their house via a provided web service.

Multimedia Data. The term refers to the convergence of text, picture, audio, and video into a single form. Multimedia data finds its application in numerous areas including surveillance, entertainment, journalism, advertisement, education and more. As a result, it can easily contribute a large source of data to the Internet of Things.

Positional Data. This data represents the location of an object within a positioning system, for example a global positioning system (GPS). Positional data is highly relevant in the work of mobile computing where objects are either static or mobile, or geographical information system.

Descriptive Data and Metadata about Objects (or Processes and Systems). This kind of data describes the attributes of a certain object, to help identify the object type, to address the object, and to differentiate it with other objects. For example, an IoT object might have data *"TV"*, *"Samsung"*, *"40 inches"* and the corresponding metadata for it are *"Type"*, *"Brand"*, *"Size"*.

Command Data. Some of the data coming into the network will be command data which are used to control devices such as actuators. The interfaces of each system are different, and so the format of command data will be different as well.

3.2 Cloud Databases

By its name, a cloud database [MCO10] is a database that runs on a cloud computing platform, such as Amazon Web Services, Rackspace and Microsoft Azure. The cloud platform can provide databases as a specialized service, or provide virtual machines to deploy any databases on. Cloud databases could be either relational or non-relational databases. Compared to local databases, cloud databases are guaranteed higher scalability as well as availability and stability. Thanks to the elasticity of cloud computing, hardware and software resources can be added to and removed from the cloud without much effort. Users only need to pay for the consumed resource while the expenses for physical servers, networking equipments, infrastructure maintenance and administration are shared among clients, thus reducing the overall cost. Additionally, database service is normally provided along with automated features such as backup and recovery, failover, on-the-go scaling, and load balancing.

3.2.1 Amazon Web Services

The most prominent cloud computing provider these days is Amazon with its Amazon Web Services (AWS) [ama13]. Clients can purchase a database service from a set of choices:

Amazon RDS. Amazon Relational Database Service is used to build up a relational database system in the cloud with high scalability and little administration effort. The service comes with a choice of the three popular SQL databases including MySQL, Oracle, and Microsoft SQL Server.

Amazon DynamoDB, Amazon SimpleDB. These are the key-value NoSQL databases provided by Amazon. The administrative work here is also minimal. DynamoDB offers very high performance and scalability but simple query capability. Meanwhile, SimpleDB is suitable for a smaller data set that requires query flexibility, but with a limitation on storage (10GB) and request capacity (normally 25 writes/second).

Amazon S3. The Simple Storage Service provides a simple web service interface (REST or SOAP) to store and retrieve unstructured blobs of data, each is up to 5 TB size and has a unique key. Therefore, it is suitable for storing large objects or data that is not accessed frequently.

Amazon EC2 (Amazon Elastic Compute Cloud). When clients require a particular database or full administrative control over their databases, the database can be deployed on an Amazon EC2 instance, and their data can be stored temporarily on an Amazon EC2 Instance Store or persistently on an Amazon Elastic Block Store (Amazon EBS) volume.

3.2.2 Scalability

Scalability is one key point of cloud databases that make them more advantageous and suitable for large systems than local databases. Scalability is the ability of a system to expand to handle load increases. The dramatic growth in data volumes and the demand to process more data in a shorter time are putting a pressure on current database systems. The question is to find a cost-effective solution for scalability, which is essential for cloud computing and large-scale Web sites such as Facebook, Amazon, or eBay. Scalability can be achieved by either scaling vertically or horizontally [Pri08]. *Vertical* scaling (scale up) means to use a more powerful machine by adding processors and storage. This way of scaling can only go to a certain extent. To get beyond that extent, *horizontal* scaling (scale out) should be used. That is to use a cluster of multiple independent servers to increase processing power.

Currently, there are two methods that can be used to achieve horizontal scalability, that is, *replication* and *sharding*.

Replication

Replication is the process of copying data to more than one server. It increases the robustness of the system by reducing the risk of data loss and one single

point of failure. The nodes can be distributed closer to clients, thus reducing latency in some cases, but also making the nodes far away from each other lengthens the data propagating process. Besides, replication can effectively improve read performance as read queries are spreaded across multiple nodes. However, write performance normally decreases as data have to be written to multiple nodes. Depending on the database system, replication can be *synchronous, asynchronous,* or *semi-synchronous* [Siv13]. A database using synchronous replication only returns a write call when it has finished on the slaves (usually a majority of the slaves) and received their acknowledgements. On the other hand, in asynchronous replication, a write is considered complete as soon as the data is written on the master while there might be a lag in updating the slaves. Semi-synchronous replication is in between which means an acknowledgement can be sent as soon as the write operation is written to a log file.

Replication can either be *master-slave* or *multi-master.* In case of master-slave replication, one single node of the cluster is designated as the *master*, and data modifications can only be performed on that node. Allowing one single master makes it easier to ensure system consistency, and that node can be dedicated to write operations while the others (slaves) take care of read operations. Meanwhile, multi-master replication is more flexible as all nodes can receive write calls from clients, but they are responsible for resolving conflicts during data synchronization.

Replication can provide some useful features, such as automatic *load balancing* (for example on a round-robin basis), or *failover* which is the ability to automatically switch from a primary system component (for example server, database) to a secondary one in case of a sudden failure.

Sharding

In short, *sharding* [Cod13] means horizontal partitioning data across a number of servers. One database (or one table) is divided into smaller ones, all have the same or similar structure. Each partition is called a *shard.* Partitioning is done with a *"shared-nothing"* approach, that is, the servers are CPU, memory and disk independent. Hence, sharding solution is needed for systems that have data sets larger than the storage capacity of a single node, or systems that are write-intensive in which one node cannot write data fast enough to meet the demand.

Scalability by sharding is achieved through the distribution of processing (both reads and writes) across multiple shards. A smaller data set can also outper-

form a large one. Moreover, sharding is cost-effective as it is possible to use commodity hardware rather than an expensive high-end multi-core server.

On the other hand, sharding also poses several challenges. First and foremost is choosing an effective sharding strategy, since using a wrong one can actually inhibit performance. A database table can be divided in many ways, for example, based on the value ranges of one or several fields that appear in all data items, or using a hash function to perform on an item field. The ideal option is the one that can distribute data and load evenly, take advantage of distributed processing while avoiding cross-shard joins. However, which solution to choose highly depends on the query orientation, data structures, and key distribution nature of the system. At the same time, sharding increases the complexity of a system. The system highly relies on its coordinating and rebalancing functionalities. Scattered data complicates the process of management, backup, and maintaining data integrity, especially when there is a change in the data schema. Besides, partitioning data causes single points of failure. Corruption of one shard due to network or hardware can lead to a failure of the entire table. To avoid this, large systems usually apply a combination of sharding and replication, where each shard is a replicated set of nodes.

3.3 Literature Review

Database characteristics of SQL vs. NoSQL

A lot of work has been done on studying the characteristics and features of different kinds of databases. Many reviews and surveys comparing SQL versus NoSQL as well as comparing multiple NoSQL databases are available [TB11][HHLD11][Ore10].

Padhy et al. [PPS11] characterized the three main types of NoSQL databases, that is, key-value, column-oriented, and document stores. Simultaneously, the authors gave detailed description about the data model and architecture of several popular databases, namely, Amazon SimpleDB, CouchDB, Google Big Table, Cassandra, MongoDB, and HBase.

Hecht et al. [HJ11] evaluated the four NoSQL database classes, the three above along with the graph databases. The underlying technologies were compared from different aspects, from data models, queries, concurrency controls, to scalability, but all were evaluated under the consideration of the database applicability for systems of different requirements.

Meanwhile, Jatana1 et al. [JPA+12] studied the two broad categories of databases: relational and non-relational. The authors gave an overview of each database class, along with their advantages and disadvantages. Several widely used databases were also briefly introduced. Finally, the paper highlighted the key differences between the two classes of database.

Database performance of SQL vs. NoSQL

As regards database performance measurement, countless tests have been run, but most are individual, small scale, or case specific local tests. Nevertheless, there are several researches have been carried out in an attempt to demonstrate the performance with real world loads.

Datastax Corporation examined three NoSQL databases: key-value store Apache Cassandra, column oriented Apache HBase, and document store MongoDB on Amazon EC2 m1 extra large instances [Dat13]. The results showed that Cassandra outperformed the other two by a large margin, while MongoDB was the worst. However, there was no SQL database involved, nor was there any other document database to compare with MongoDB, which is one of the main focus in our tests.

In the paper written by Konstantinou et al. [KAB+11], the elasticity of NoSQL databases, including HBase, Cassandra, and Riak, was verified and compared as the authors examined the changes in query throughput when the server cluster size changed. The results showed HBase as the fastest and most scalable when the system was read intensive, whereas Cassandra performed and scaled well in a write intensive environment, and nodes could be added without a transitional delay time. Apart from that, the authors proposed a prototype for an automatic cluster resize module that could fit the system requirements.

Meanwhile, Rabl et al. [RGVS+12] addressed the challenge of storing application performance management data, and analyzed the scalability and performance of six databases including MySQL and five NoSQL databases. The benchmark showed the latency and throughput of those databases under different workload test cases. Again Cassandra was the clear winner throughout the experiments, while HBase got the lowest throughput. When it comes to sharding, MySQL achieved nearly as high throughput as Cassandra. Although a standalone Redis outperformed the others when the system was read intensive, its performance of a sharded implementation dropped with an increasing number of nodes. The same case applied for VoltDB in a sharded system, thus Redis and VoltDB did not scale very well.

Tudorica et al. [TB11] compared MySQL, Cassandra, HBase, and Sherpa. The experiments concluded that the SQL database was not as efficient as the NoSQL ones when it comes to data of massive volume, especially on write intensive systems. However, MySQL could have relatively high performance on read intensive systems.

One common point of those papers is their use of Yahoo! Cloud Serving Benchmark (YCSB), a generic framework for the evaluation of key-value stores, which helped to perform the tests with sets of big data, hundreds of millions of records, hundreds of clients and multiple data nodes, simultaneously they provide the setup for different read and write intensity. However, little attention was paid on the structure and variety of data.

Internet of Things storage

As regards more types of data related to IoT data, van der Veen et al. [vdVvdWM12] compared PostgreSQL, Cassandra, and MongoDB as a storage for sensor data. The tests were not run in a cloud environment, but run in a comparison between a physical server and a virtual machine. The paper is closely related to our work for the similar used data structure. The result did not show a solely winner as MongoDB won at single writes and PostgreSQL at multiple reads. The impact of virtualization was unclear for it was different in each case.

Other solutions for storing IoT data have also been proposed. One is a storage management solution called IOTMDB which is based on NoSQL [LLT+12]. The system came with the strategies for a common IoT data expression in the form of key-value, as well as a data preprocessing and sharing mechanism.

Pintus et al. [PCP12] introduced a system called Paraimpu, a social scalable Web-based platform that allowed clients to connect, use, share data and functionalities, and build a Web of Things connecting HTTP-enabled smart devices like sensors, actuators with virtual things such as services, social networks, and APIs. The platform uses MongoDB as the database server, provides models and interfaces that help to abstract and adopt different kinds of things and data.

Another solution is the SeaCloudDM [DXY12], a cloud data management framework for sensor data. The solution addressed the challenges that the data are dynamic, various, massive, and spatial-temporal (i.e., each data sampling corresponds to a specific time and location). To provide a uniformed storage mechanism for heterogeneous sensor sampling data, the system combined the use of the relational model and the key-value model, and was implemented with PostgreSQL database. Its multi-layer architecture was claimed to reduce the

amount of data to be processed at the cloud management layer. Besides, the paper also came with several experiments that showed a promising result for the performance of the system when storing and querying a huge volume of data.

Meanwhile, Di Francesco et al.[DFLRD12] proposed a document-oriented data model and storage infrastructure for IoT heterogeneous as well as multimedia data. The system used CouchDB as the database server, taking advantage of its RESTful API, and supporting other features such as replication, batch processing, and change notifications. The authors also provided an optimized document uploading scheme for multimedia data that showed a clear enhancement in performance.

In this thesis, we also target the same data types as [DFLRD12], but we will provide more intense experiments, focus on sensor data and run on the cloud with multiple databases.

CHAPTER 4

Experimental Methodology and Setup

In this chapter, we describe in details the methodology used to assess the performance of SQL and NoSQL databases. Many tests were carried out on four databases: *MySQL, MongoDB, CouchDB*, and *Redis*. The set-up of the tests, including the test environment, materials, design, and procedure, will be presented. By the end of the chapter, we will thoroughly explain what, how, and why these tests were taken.

4.1 Experiment overview

The main goal of the tests was to compare the performance of different databases as cloud databases. Hence, the database servers were placed on the cloud. A system was implemented to play the role of the database clients. The system can perform very basic read/write operations on the databases. The tests were to run these operations with different workload, measure the average request latency, and compare them among the databases.

The choice of the databases was based on the fact that those were among the most popular databases available, and that they were the representatives for

their kinds. Many large organizations have been using them in production, such as Facebook, Google, Wikipedia, LinkedIn, Instagram, and more. On the other hand, each database has its own promising strength that is worth exploring. MySQL so far has been the most popular open source SQL database. MongoDB was built to work with very large sets of data. CouchDB has its user-friendly RESTful API. Meanwhile, Redis is said to be very fast thanks to its in-memory storage.

To benchmark the database performance, many works in the literature have used Yahoo! Cloud Serving Benchmark (YCSB) [Dat13][KAB+11][RGVS+12][TB11]. The databases themselves also provide their own benchmarks. However, these benchmarks only allow setting the record size without specifying its actual data types, which we believe to have a great impact on the database performance. Our tests targeted two particular data types: *sensor scalar data* and *multimedia data* which were expected to contribute a large portion to the whole Internet of Things data. Therefore, we built our own system with two benchmarks to test on the two specific data structures.

As mentioned before, the tests were to evaluate the performance of the basic read and write operations. Each operation was assessed separately, meaning at any point, only one kind of database was tested, only one test was running, and the system was under 100% read load or 100% write load. A test was a single or a continuous series of either read or write requests sending from clients to a database. Different tests were setup using different values of parameters, which could be the number of records, the number of concurrent clients (simulated by multi-threads) and so on. More details will be given later for each benchmark.

The performance measurement was done on the client side. The time taken to complete the requests in each test was measured. Separated time was recorded for connecting to the database and for actually executing the requests. One limitation was that the network connection between clients and servers was not dedicated to the tests and could not be controlled. Hence, in order to increase reliability, each test of the same input were run multiple times (at least 10 times). The final result for a test was then the average of these individual runs.

4.2 Experiment environment

4.2.1 Hardware and Software

The test system was implemented in Java and ran on the Java 7 Virtual Machine (JVM). The system included machines for database servers and their clients that either wrote data to or read data from the server:

- *Server*: Each server was deployed on a virtual machine instance running 64-bit Ubuntu 12.10 on an Amazon EC2 m1.large instance (7.5 GiB memory, dual-core with total 4 Compute Units, 840 GiB instance store volume, moderate I/O performance, European region [ama13]).
- *Client*: a local machine running 64-bit Ubuntu 12.04, with 7.5 GiB memory and Intel Core 2 Q6600 2.40GHz quad-core processor.

4.2.2 Database configuration

Used versions of the databases:

- MySQL 5.6.10
- MongoDB 2.4.3
- CouchDB 1.2.0
- Redis 2.6.12

All databases ran with default configurations, except for the followings:

- *Data* and *log* file path changed to the Amazon instance store volume.
- *bind-address = 0.0.0.0* (to enable connections from any remote client and not have to reconfigure when the server IP address is dynamically assigned).
- MongoDB *nssize = 100* (This is to increase the namespace file size to 100MB, thus increasing the limitation of namespaces including the number of collections and indexes. By default the size is 16MB which corresponds to about 24000 namespaces)

4.2.3 Libraries and drivers

In order to connect and interact with the database servers, the following libraries and drivers were used in the implementation of the database clients:

- MySQL JDBC[1] Connector version 5.1.22
- MongoDB Java Driver[2] 2.10.1
- Ektorp[3] 1.2.2 for CouchDB
- Jedis[4] 2.1.0 for Redis

4.3 Sensor scalar data benchmark

This benchmark is the focus of the thesis. It was used to test the efficiency of *MySQL*, *MongoDB*, *CouchDB*, and *Redis* when it comes to storing scalar readings generated by sensors. The benchmark was built based on the Home Energy Management System developed by There corporation [the13].

4.3.1 System description

The system implemented for the tests simulated a sensor network, the architecture is shown in Figure 4.1. The network comprised of one central *database server* (located on the cloud) and multiple uniquely identified *sensor nodes*. The nodes were divided into smaller groups and each group was monitored by a *data sender*. Each node generated a reading record once in an interval. The interval was common for all nodes and could be set by the user. The data sender was responsible for collecting individual records and sending them all to the database server. The user could configure the data sender to send the data individually or in a bulk, send once every interval or store the data in an internal buffer and send later when the buffer is full. Multiple database clients can read these data from the server. In the implementation, instead of multiple sensor nodes per node group, we used a *data generator* that periodically created a series of sensor records with random values, simulating the interval readings coming from the

[1] http://dev.mysql.com/downloads/connector/j/
[2] http://docs.mongodb.org/ecosystem/drivers/java/
[3] http://www.ektorp.org/
[4] http://code.google.com/p/jedis/

nodes. The data series would then be passed to the data sender to be inserted in the database.

For such system, we made the following assumptions:

- The data had the same structure for all records.
- Performance and availability had higher priority than data integrity.
- The system was highly write intensive, i.e., data were sent continuously at a short interval.
- In practice, the writing thread is meant to run continuously without disconnection. However, in the tests, we only measured the time taken to execute a particular number of writes.
- The system was expected to serve clients in real time which means that once the data were generated, they would be sent immediately to the database and ready for clients to query.
- Queries were simple, possible queries were: fetching all data in the database, fetching all data belonging to one node, continuouly fetching new data of one node. Note that the queries were considered finished when the returned list of records had been iterated through.
- Update and delete requests rarely happened and so were not considered.

4.3.2 Data structure

The common data structure for all records is shown in Table 4.1. When stor-

Name	Type
nodeID	String
time	Date
value	double

Table 4.1: Sensor scalar data structure

ing this data type in different databases, there was a slight difference in the database storage structures. Figure 4.2 shows how the databases appeared to users in each case. The databases were grouped based on their structure. We tested MongoDB with both types of structure (denoted as Mongo_1set and Mongo_mset from now on).

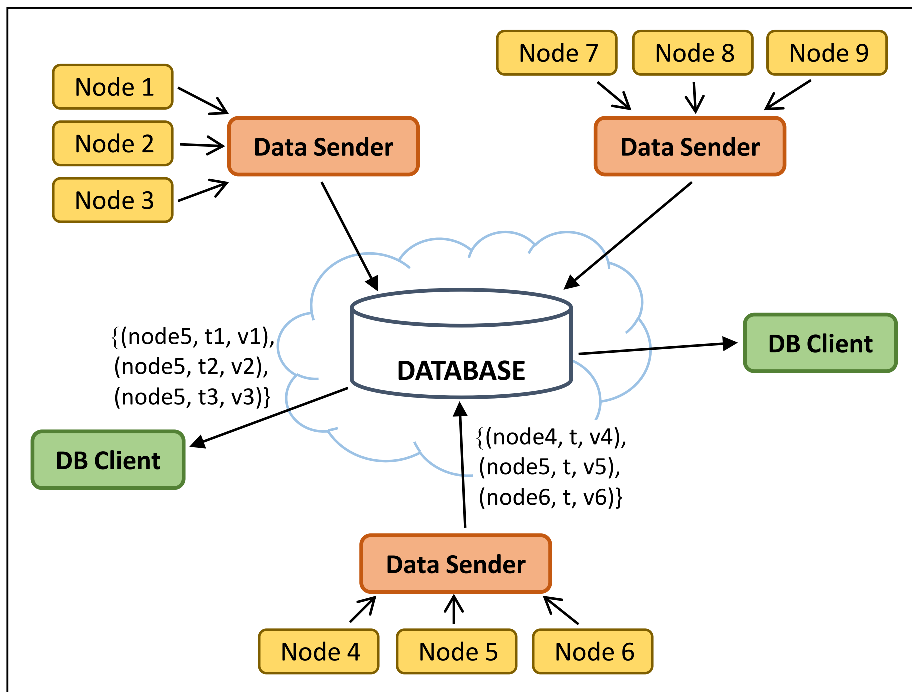


Figure 4.1: System architecture of the Sensor Scalar Data Benchmark

One data set: MySQL, Mongo_1set, CouchDB. For these databases, records of all nodes were stored as one common set only. The advantage of this structure is that it was easy to make use of bulk insert and improve write performance, as there was only one destination storage.

- *MySQL:* The database contained only one table *data* of structure {*nodeID, time, value*} where {*nodeID, time*} was the primary key. It is worth noting that the primary key was automatically indexed.
- *Mongo_1set:* The database contained only one collection *data*. In each document, a field *_id* of ObjectId type was automatically added to uniquely identify the documents. The *_id* field was automatically indexed as well. For Mongo_1set, we could have created a structure more similar to the one in MySQL by grouping {*nodeID, time*} as a nested document and making it the *_id*. However, we decided to discard this approach, for it would complicate and slow down indexing as well as querying for data of a single node.
- *CouchDB:* The database itself was a set of all documents. *_id* and *_rev*

fields were automatically added by the system.

Multi data sets: Mongo_mset, Redis. In this case, one database consisted of multiple subsets, each dedicated to one node, the subset name was the *nodeID*. Data sent to the database were distributed to the corresponding subset. This design reduced the duplication of *nodeID* field in every record. Besides, querying for data of a single node, which we considered the most popular query, was simpler and only worked on a small set of data rather than all the data.

- *Mongo_mset:* Each node corresponded to a collection. Inside a collection, a document had type {*_id, value*}. Here the *time* itself ensured uniqueness, therefore we used it as the *_id*, thus saving up the space used for *ObjectIds*.
- *Redis:* The database was made up of multiple hashes, the hash keys were the *nodeIDs*. Each hash was a map of all the *time* fields and their corresponding *values*.

MYSQL		
data		
nodeID	**time**	**value**
nodeID1	2013-05-19 22:34:06.338	16.1
nodeID1	2013-05-19 22:34:11.163	58.96
nodeID2	2013-05-19 22:34:06.338	46.59
nodeID2	2013-05-19 22:34:11.163	13.57

MONGO_1set
data
_id: ObjectId("519bf6ede4b04f58cf3d290f") **nodeID**: "nodeID1" **time**: ISODate("2013-05-19T22:34:06.338Z") **value**: 16.1
_id: ObjectId("519bf6ede4b04f58cf3d2911") **nodeID**: "nodeID2" **time**: ISODate("2013-05-19T22:34:06.338Z") **value**: 46.59

COUCHDB
_id: "0cd28453657d5d6715666f336c577ec3" **_rev**: "1-054e4fbe7931bfbac63207db22208efa" **nodeID**: "nodeID1" **time**: "2013-05-19T22:34:06.338+0000" **value**: 16.1
_id: "0cd28453657d5d6715666f336c57876d" **_rev**: "1-82f63be6c7b04217602a08b0c566b46b" **nodeID**: "nodeID2" **time**: "2013-05-19T22:34:06.338+0000" **value**: 46.59

MONGO_mset	
nodeID1	**nodeID2**
_id: ISODate("2013-05-19T22:34:06.338Z") **value**: 16.1	**_id**: ISODate("2013-05-19T22:34:06.338Z") **value**: 46.59
_id: ISODate("2013-05-19T22:34:11.163Z") **value**: 58.96	**_id**: ISODate("2013-05-19T22:34:11.163Z") **value**: 13.57

REDIS	
nodeID1	**nodeID2**
"2013-05-19 22:34:06.338" "16.1" "2013-05-19 22:34:11.163" "58.96"	"2013-05-19 22:34:06.338" "46.59" "2013-05-19 22:34:11.163" "13.57"

Figure 4.2: Sensor database structure for the different solutions considered

4.3.3 Parameters

In the following, we list the parameters that can be tuned to assess the performance from different aspects:

- Type of database: MySQL, Mongo_1set, Mongo_mset, CouchDB, Redis.
- The number of clients, i.e., the number of concurrent threads doing the same task.
- For Mongo_1set only: whether to create an index on *nodeID*.
- For write operations:
  - The number of records to be sent to the database, i.e., the number of nodes multiplied by the number of data generations.
  - The size of bulk insert, i.e., the number of records that were to be sent together in one request.
- For read operations:
  - The query to perform: get database size, get all data, get all data of one node.
  - NodeID: the nodeID to be used when querying for all data of one node. In case of multi-clients, each queried for a different node.

4.4 Multimedia data benchmark

The purpose of this benchmark is to evaluate the performance of SQL and NoSQL databases when used for multimedia storage on the cloud. *MySQL* and *MongoDB* were chosen as representative databases in this case.

4.4.1 System description

The system architecture is illustrated in Figure 4.3, which is similar to that of the sensor scalar data benchmark. The system consisted of multiple *media senders* and multiple *database clients* connecting to the central cloud database. The media senders were capable of sending multimedia files to the server, one at a time, while the database clients could query for the content of the files. The multimedia file could be of any type: audio, video, picture or text.

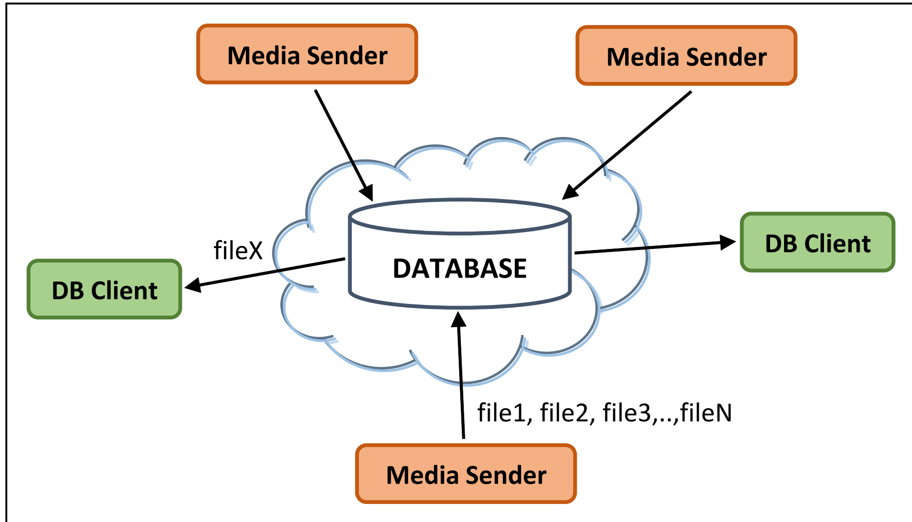


Figure 4.3: System architecture of the Multimedia Data Benchmark

The following assumptions were made during the tests:

- All data files had the same size and format.
- The writing threads ran without disconnection, sending the data files continuously, one after another.
- Queries were simple, the tested query was the one querying for one particular file.
- Update and delete requests rarely happened and so were not considered.

4.4.2 Data structure

MySQL and MongoDB provide different data types for storing data of large size. Since multimedia data can be larger than normal database items, we used those special data types to store multimedia files.

- *MongoDB*: the database contained one *GridFS bucket* in which data files were automatically divided and stored in small chunks (refer to Section 2.4.3). Each file was set with a unique *filename* which was a counter incremented by 1 for each file inserted.

- *MySQL*: the database consisted of one table with the structure shown in Table 4.2. Here the whole file content was stored as a BLOB (a binary large object of various size), and each file was uniquely identified by an *id* (the primary key) which corresponded to the *filename* in MongoDB case.

Name	Type
id	integer
data	longblob

Table 4.2: MySQL multimedia data structure

4.4.3 Parameters

The following parameters can be configured differently for different tests:

- Type of database: MySQL, MongoDB
- The number of clients, i.e., the number of concurrent threads doing the same task.
- For write operations:
  - The data file to be inserted.
  - The number of inserts (the above file will be written repeatedly and continuously)
- For read operations:
  - The query to perform: get database size, get all files, get one file of a particular ID. Note that the get file operation was considered finished when the data file was queried and written to a local file.
  - FileID: the file ID to query. In case of no FileID specified or multi-clients, each thread would query for a different random file.

Chapter 5

Experimental Results

In this chapter, we detail the results of all the tests taken. Additionally, evaluation for each result will also be presented.

‘

5.1 Sensor scalar data benchmark results

5.1.1 Bulk insert

Instead of inserting data to the database once at a time, clients can use a feature called *bulk insert*, which means to insert multiple records at the same time in one operation. Bulk insert is expected to improve write performance. Hence, the purpose of this test is to examine the efficiency of bulk insert over individual inserts, also to check the write latency when it comes to bulk insert with different number of records.

Bulk insert is supported by all the considered databases. However, between the two data types introduced in Section 4.3.2, we chose to implement bulk insert feature only on the group of databases with one data set (MySQL, Mongo_1set,

CouchDB). This is reasonable because: first, bulk insert only works on one table/collection (or one data set), meaning a list of data cannot be distributively inserted to multiple data sets at once; second, each node generated one record at a time but all nodes generated data at the same time; finally, these data were meant for real time query.

Parameters

The benchmark was carried out on MySQL, Mongo_1set and CouchDB. Except from the bulk size (i.e., the maximum number of records to be inserted at once) which was changed for each test, the parameters shown in Table 5.1 were set the same for all databases and for all tests.

Parameter	Value
The total number of records	10,000
The number of nodes	100
The number of data generations	100
The number of concurrent writing threads	1
MongoDB index for NodeID	True

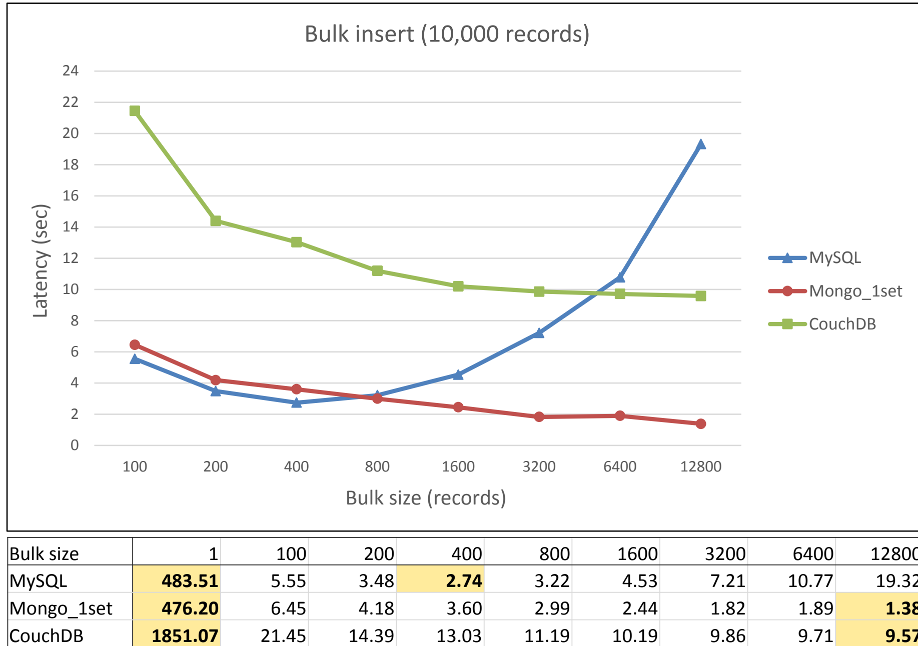
Table 5.1: Parameters of the bulk insert test

Results and Evaluation

Each test was run 10 times. The average values for the write latency (excluding connection time) are shown in Figure 5.1.

The latency values for individual insert (where bulk size equals to 1) were not put into the graph due to the huge difference with the other values, but they are included in the data table. As can be seen, applying bulk insert has a great impact on the performance of the three databases, as individual inserts cost the greatest latency compared to bulk inserts. In the table, the maximum and minimum latency values for each database were marked. The biggest difference between maximum and minimum values was in MongoDB with approximately 350 times, for it took only more than 1 second to insert 10000 records in one operation, but it took nearly 8 minutes to insert the same amount of data individually. Those differences for MySQL and CouchDB were around 175 times and 200 times respectively.

According to the graph, among the three databases, MongoDB had the best results in general. The insert time decreased gradually along with the increase



Bulk size	1	100	200	400	800	1600	3200	6400	12800
MySQL	**483.51**	5.55	3.48	**2.74**	3.22	4.53	7.21	10.77	19.32
Mongo_1set	**476.20**	6.45	4.18	3.60	2.99	2.44	1.82	1.89	**1.38**
CouchDB	**1851.07**	21.45	14.39	13.03	11.19	10.19	9.86	9.71	**9.57**

Figure 5.1: Bulk insert latency

in bulk size. A similar pattern applied for CouchDB, but CouchDB was outperformed by MongoDB in all cases. In the meantime, MySQL produced a decrease until reaching the lowest at the bulk size of 400 records, after this point, the latency significantly climbed up again. However, in the beginning with a smaller bulk size, MySQL achieved a slightly better latency than MongoDB, although the reverse was applied when it comes to individual insert.

It is worth noting that individual insert in MySQL was implemented using *prepared statements*. Hence, performance can be greatly improved compared to using normal statements. This is because all the inserts had the exact same syntax, therefore using prepared statements would eliminate the cost of constructing and compiling the statement at every request. Meanwhile, we chose not to implement MySQL bulk insert with prepared statements. The reason is that while bulk insert functions in CouchDB and MongoDB can accept a data list of any size, doing the task by a prepared statement in MySQL will limit the number of records inserted in one statement to a fixed bulk size. As a result, if a node fails to generate its data or the number of records per each write is different, the prepared statement cannot be fulfilled, thus leading to a failure in writing data of the whole group of nodes. Therefore, a bulk insert query in MySQL was built dynamically as a normal statement each time it was

called. The construction of such a statement normally takes time, consequently, the insert latency for MySQL started to grow after some point as the bulk size increased.

Apart from affecting the write latency, bulk insert also caused an effect on the data size of CouchDB, although it was not the case for the other databases. As mentioned in Section 2.4.2, this was due to the use of *append-only Btree* to store documents. As can be seen in Figure 5.2, the difference is obvious between individual insert and the rest, for the data storage when using individual insert was around 14 times more than that of using a bulk insert of size 100. The data size afterward also reduced as the bulk size increased although with a very slight difference.

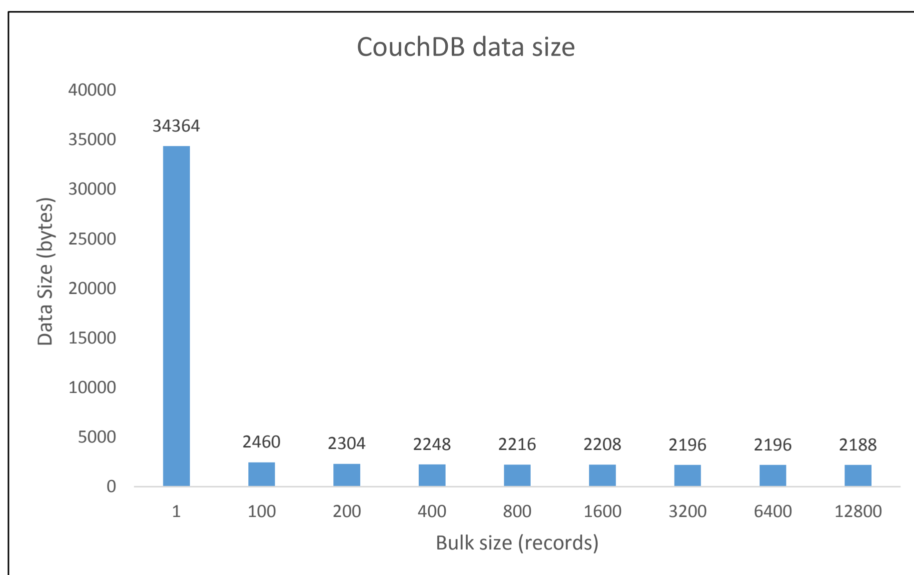


Figure 5.2: CouchDB data size for bulk insert

5.1.2 MongoDB index

In this scenario, the main query that we focused on is to query data by *nodeID*. Hence, it was expected that creating an index on the *nodeID* field would make an enhancement in the query performance. The following test was to check the impact of creating the index on read and write latency.

Parameters

The test configuration was shown in Table 5.2.

Parameter	Value
The total number of records	1,000,000
The number of nodes	1000
The number of data generations	1000
The number of inserting threads	1
The number of querying threads	multiple
Bulk insert size	1000

Table 5.2: Parameters of the MongoDB index test

Results and Evaluation

The average values for read and write latency were illustrated in Figure 5.3.

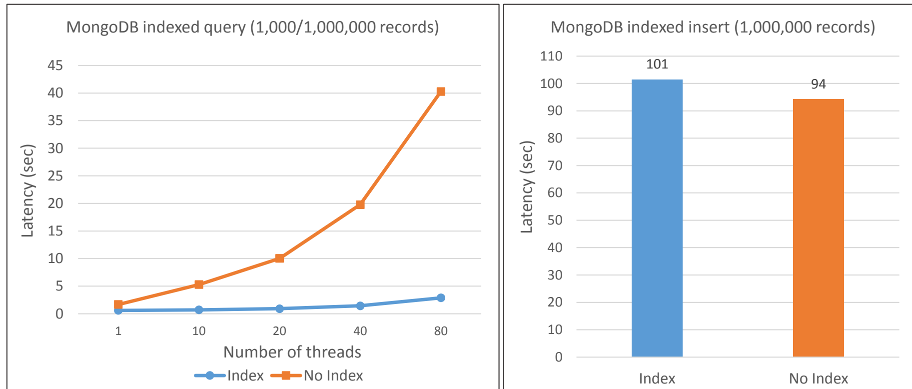


Figure 5.3: MongoDB index latency

As expected, the index had a great impact on the quey performance, as it dramatically reduced the read latency. The effect was clearer when more concurrent querying threads were added. For instance, with 80 threads, the reading latency with the index was less than 3 seconds on average, whereas it was more than 40 seconds without it. Meanwhile, the write latency (excluding connection time) showed much less difference between the two cases, which we considered acceptable for this system to compromise for the improvement in querying. From this

point onwards, the index for *nodeID* in Mongo_1set was created in all later tests. However, more indexes will cause a greater degradation in the write performance. Hence, in practice, it is necessary to analyze the system use cases and requirements to decide whether to sacrifice writes for reads or vice versa before creating secondary indexes in the database.

5.1.3 Write latency

This is one of the main benchmarks that were carried out. As its name, the benchmark is used to evaluate the write performance of the system when implemented with different databases.

Parameters

The write performance was to be assessed in a multi-threaded environment (not only multiple *sensor nodes* but multiple *data senders*), therefore different numbers of threads were set for different test suites. Apart from that, the other parameters were set as in Table 5.3 for all the tests.

Parameter	Value
The number of nodes per thread per generation	1000
The number of data generations	100
MongoDB index for NodeID	True
MySQL, Mongo_1set, CouchDB bulk insert size	1000
Mongo_mset, Redis bulk insert size	1 (individual insert)

Table 5.3: Parameters of the write latency test

Results and Evaluation

Figure 5.4 illustrates the average values of write latency with respect to different numbers of concurrent writing threads. The values exclude the connection time, since we expected the sensor nodes to send data continuously without disconnection in real life.

The graph shows an obvious discrimination between the *multi data set* group that used individual insert (Mongo_mset, Redis) with higher latency and the *one data set* group that used bulk insert. In the previous section, the bulk insert

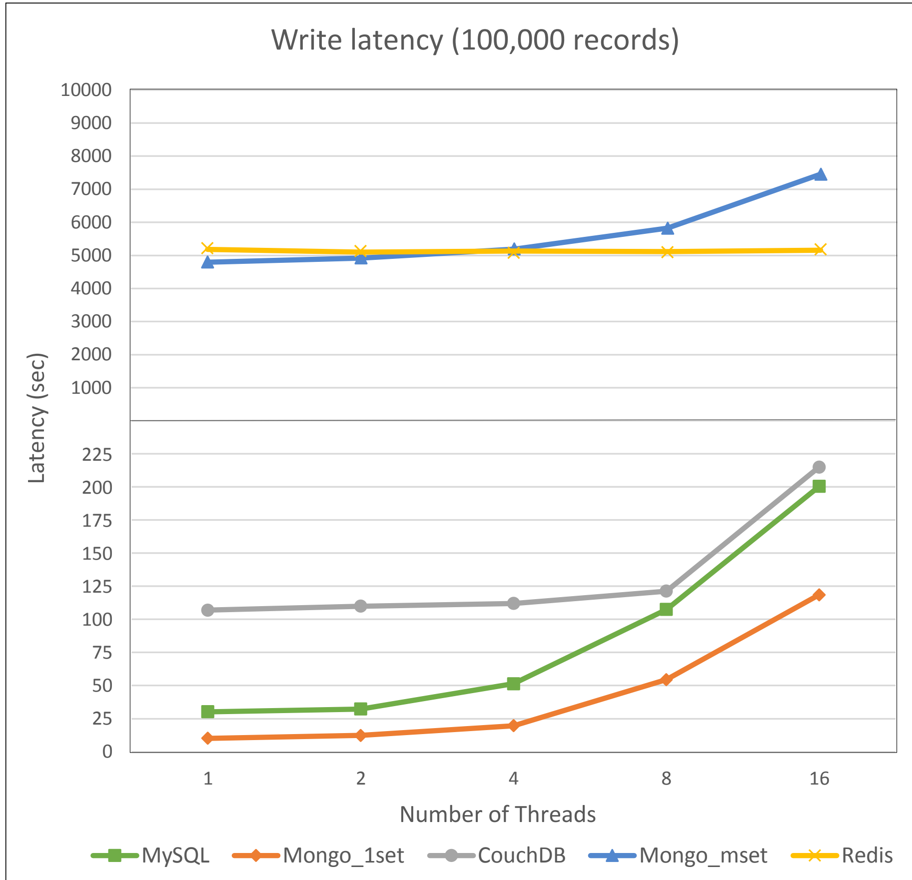


Figure 5.4: Write latency

benchmark has shown the important role of the bulk size to insert performance, although its effectiveness only applied to a subset of database types with the current system design. In this benchmark, the choice for bulk size was made as a compromise that could reflect how the system could be used in real life, while providing a general comparable view among the databases. That is to say the result does not mean that Redis writes much slower than the other databases, but it has an disadvantage for this particular type of system and data.

In the lower group of databases with one data set, Mongo_1set got the best result out of the three. With a single thread, the database took about 10 seconds to insert 100,000 records, followed by MySQL which was three times slower, while the latency of CouchDB was 10 times larger. Meanwhile, Mongo_mset

and Redis results were close at more or less 5000 seconds with a small number of threads, but as there were more threads, the former became slower than the latter.

Regarding concurrency, Redis, for which all operations were performed in RAM, was hardly affected by the number of threads. However, the case was different for the other databases as the latency increased along with the number of threads. In the benchmark implementation, in order to avoid conflicts, each thread wrote data of a different node. That means for Redis and Mongo_mset writes were distributed to different data sets, but for the others the writes still concentrated on one data set. In MySQL, the default InnoDB storage engine uses *row* locking which helps to eliminate the possible overhead of *table* locking (for example in MyISAM storage engine) in this case. In case of MongoDB, not only the system processing power was shared among the threads, but also MongoDB has locks on a per-database basis, which means only one write operation can be performed in the database at a time, causing some overhead for both Mongo_1set and Mongo_mset.

5.1.4 Read latency

The following set of tests is to assess the query performance of the databases on different loads, including data of different volumes and different number of concurrent queries. For this system, we assumed that the most popular query was the one to get all the data of one particular node, hence, we used that query for all the tests.

Parameters

As mentioned before, the number of threads and the number of records varied for each test suite, but the configuration in Table 5.4 was kept the same. Note that each thread queried for a different node and all nodes had the same amount of data.

Parameter	Value
Query	Fetch all data of one node
The number of nodes	1000
MongoDB index for NodeID	True

Table 5.4: Parameters of the read latency test

Results and Evaluation

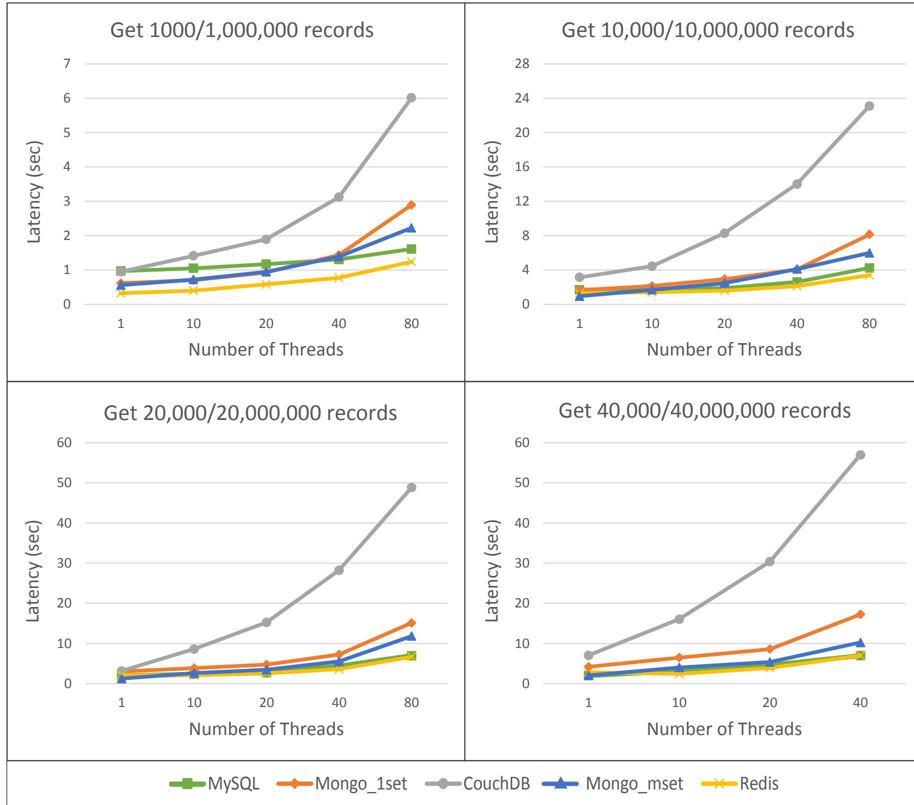


Figure 5.5: Query latency as a function of the number of threads

Figure 5.5 shows the average query latency (including connection time) with respect to different numbers of threads in four test suites, each with a different database size. From the first glance, it can be seen that CouchDB was significantly outperformed by all the other databases while the differences among the rest were less remarkable. For example, a data set of total 20 million records, with 80 concurrent clients, to retrieve 20000 records of a node, it took CouchDB nearly 50 seconds but only less than 7 seconds for MySQL and Redis, with Redis slightly faster. This latency of Mongo_1set and Mongo_mset were 15 and 12 seconds respectively.

In general, Redis got the best results of all thanks to its in-memory storage and the nature of querying by key in this case. Simultaneously, in cases where the database size was large (10,000,000 records or more), MySQL closely kept

up with Redis and also queried very quick. As mentioned in Section 2.4.1, querying by the primary key index in MySQL, as in this case, would receive optimal performance since the data were physically sorted by the primary key.

Meanwhile, between the two types of MongoDB, the change in data structure from Mongo_1set to Mongo_mset (meaning switching from all data in one collection to multiple collections) improved the performance of the query in all cases as espected. Moreover, in some cases with a single thread, Mongo_mset even slightly read faster than the others.

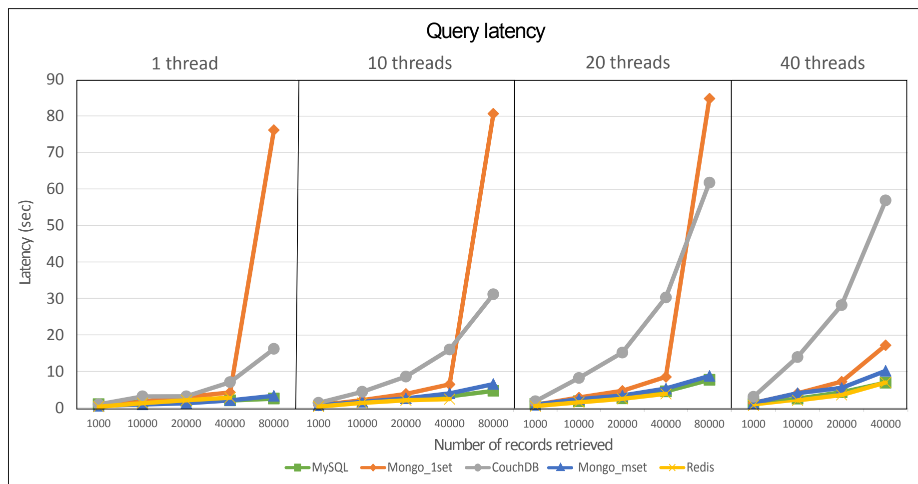


Figure 5.6: Query latency as a function of the database size

It is worth noting that all the queries were run after the databases had been "*warmed up*", which means they were run once at the server side before. This was to make the data ready to be queried in the quickest mode. The warm-up did not have an impact on Redis since all data had already been in the RAM. Nevertheless, it had great effect on CouchDB, since the view index was created in the first run, which took up a lot of time. For the other databases, the first run loaded the data set of MongoDB into memory and the query results of MySQL into the query cache, which highly boosted up the performance afterward. However, in practice, the impact of this warm-up can be not as much as in the tests because data are bound to be changed and added continuously, or the amount of data can not fit in the cache. The latter case happened for MongoDB as shown in Figure 5.6. The figure is another view of the data displayed in Figure 5.5, now showing the latency with respect to different number of records retrieved, with the add of the case where the database size is 80 million records, and the query returns 80000 of them. In this case, the working set was too big to be loaded into memory, as a result, queries had to read from disk,

causing a sudden degradation of performance for Mongo_1set. For example, for one single thread, querying for 40000 records took only 4 seconds, while that of querying for 80000 records climbed up to more than 75 seconds which was 20 times slower. Hence, MongoDB query performance is very poor when the working set cannot fit into memory, in this case, a horizontal scaling solution should be considered.

On the other hand, the latency trend for all databases shared the same pattern in all cases in the sense that the latency grew as the number of threads or the database size grew. However, if we leave out the case where MongoDB and Redis could not fit the RAM, CouchDB performance dropped the most when serving more clients or more data, while MySQL and Redis stayed more stable under the increasing load.

5.1.5 Database size

Although storage space is a cheap factor compared to performance as the capacity of hard disks is ascending dramatically while the price is descending, for the issue of Big Data, storage space is still an aspect that is worth considering. The bar chart in Figure 5.7 illustrates the database size made up of 10 million records for different database types. The database size was calculated as the sum of the actual *data size* plus the *index size*.

In the chart, CouchDB was shown to take up much more space than the rest, nearly 20 GiB for the 10 million records, which was around 40 times more than that of MySQL, the one with the smallest size. A large portion of CouchDB huge size came from the view indexes, that was nearly 90% of the total. Moreover, in this test, there was only one index (built for the query of fetching data of each node), the size would have been greater if more views had been created.

Meanwhile, there was no extra index in Redis or MySQL. That was because Redis did not support secondary index, and for this system, there was no need to build a secondary index for MySQL apart from the automatic indexed primary key which physically ordered the data on disks. However, since Redis is an in-memory database, it has the greatest disadvantage in storage compared to the other ones, as the storage capacity is limited by RAM, not to mention the fact that the database size itself was larger than that of MySQL, and Mongo_mset which had the same design of multiple data sets.

For MongoDB, the design of Mongo_mset saved some space compared to Mongo_1set, since it eliminated the duplication of the *nodeID* field and promoted the *time* to be the unique *_id* instead of using a system-generated *ObjectId*. This made

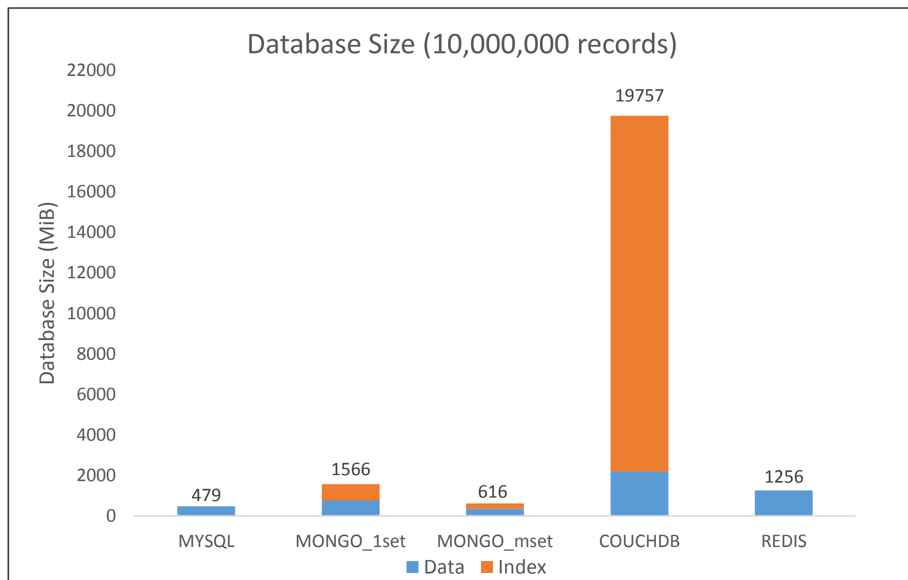


Figure 5.7: Database size

Mongo_mset quite efficient in space usage, only slightly larger than MySQL. Besides, Mongo_1set, in this case, took up some space for the index of nodeID, although it could be turned off, it was necessary for a better query performance. Even without nodeID index, all MongoDB databases will have at least one index automatically created for the *_id* field.

Besides, compared to MySQL, document stores like CouchDB or MongoDB caused an extra space for storing the field names repeatedly (for instance: *"time"*, *"value"*) and the *_id* or *_rev* fields generated in all documents. In practice, those kinds of databases can follow some tips to improve the storage usage, which in turn can help improve the performance since the smaller the data set the quicker. Firstly is to use short field names, for example if the string name *"nodeID"* is 6 bytes, making it just *"id"* will save 4 bytes for each document. Secondly, in MongoDB, an auto-generated *_id* field of type *ObjectId* contains itself a 4-byte value of *timestamp*. Although the granularity is only to second, this value can be used instead of an extra time field. For the implemented system, the time field is of type BSON *Date*, a 8-byte representation of time in milliseconds.

5.2 Multimedia data benchmark results

5.2.1 Write latency

The tests below were run to compare the write performance of the two databases MySQL and MongoDB when storing multimedia files. Since we assumed that all the files had the same size and format, the two parameters that could influence the result and therefore were set with various values in different tests were:

- The size of each data item
- The number of concurrent inserting threads

In addition, the write latency was measured as the time taken to finish inserting 1000 mp4 video files per thread (excluding connection time, since we assumed that the system generated data continually).

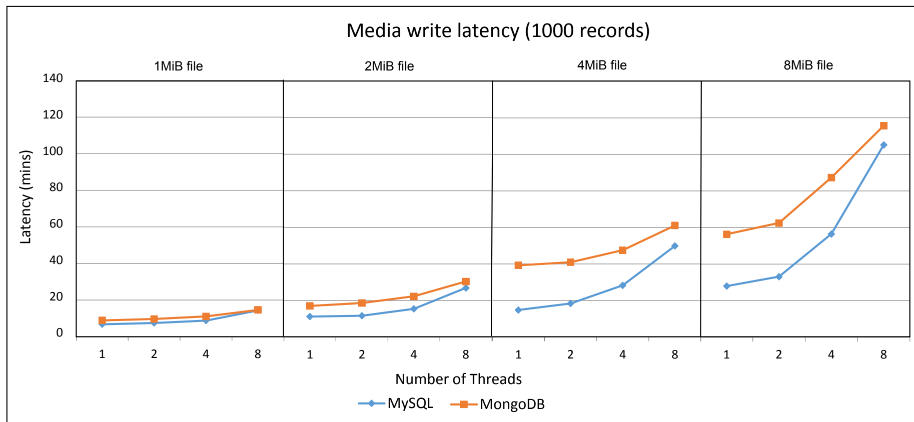


Figure 5.8: Media write latency

The results for the tests are shown in Figure 5.8. The graph demonstrates a clear discrimination between MySQL and MongoDB, where the former outperformed the latter. It can also be seen that the performance of both databases highly depended on the file size and the number of threads, as the latency increased proportionally with the increase of the item size and threads. Nevertheless, the graph shows that MongoDB shortened the difference with MySQL when the servers served more clients at the same time. In case where the file size was 8 MiB, the latency of one thread inserting was 28 and 56 minutes for MySQL

and MongoDB, whereas the response time was 105 and 115 respectively when there were 8 concurrent threads. While MySQL database stored the file as one single blob, MongoDB's GridFS divided and stored it as small chunks along with the file metadata, which added extra information and some more latency to the operation. However, the latter approach benefits more if the clients want to query for a sub-part of the file, or if the system requires high scalability.

5.2.2 Read latency

Among the parameters that could be configured in the system, these three are potential to have an impact on the query performance of the database:

- The size of each data item
- The total number of items in the database
- The number of concurrent querying threads

The purpose of the following tests is, therefore, to find out the effect of these parameters on the latency of querying for a specific file in the database. The latency recorded in the tests includes both the connecting and querying time, remind that the querying time itself consists of the time to scan through the database to find the right record (for MongoDB it means searching through both the *files* and *chunks* collections) plus the time to read the binary data from the database and write it to a local file.

Impact of file size

Figure 5.9 illustrates the different read latency when it comes to querying for files of different size. The data set in all cases contained 1000 mp4 video files of the same size. The latency was grouped according to the different number of concurrent querying threads, each queried for a *random* file.

From a first glance, the graph does not show a great difference between MySQL and MongoDB in this case, for the two graph lines closely follow each other. The graph, however, does show a clear impact of the data item size on the read performance, since the time taken for reading one item roughly doubled as the file size doubled. For instance, for 40 concurrent threads querying for data of 1 MiB, it took MongoDB and MySQL more than 4 and 5 seconds, while the

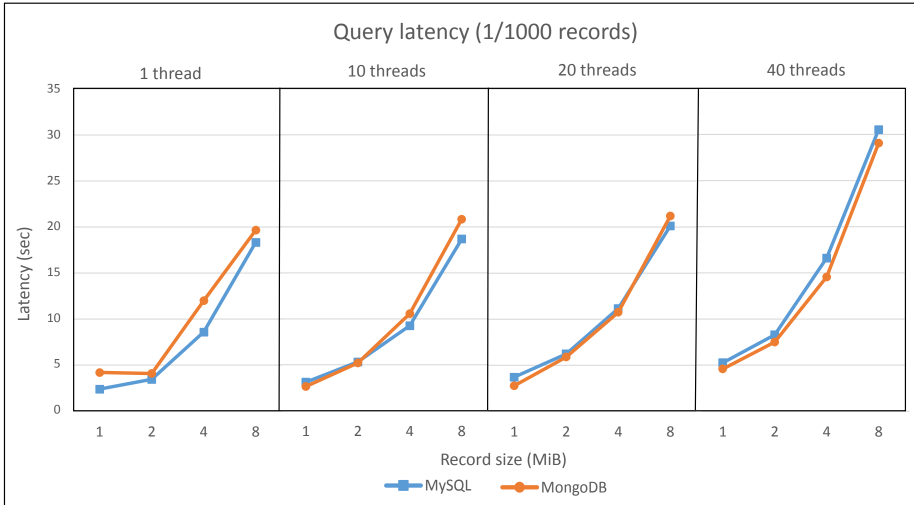


Figure 5.9: Media query latency as a function of the file size

latency when the data size raised to 8 MiB was more than 29 and 30 seconds respectively.

Impact of database number of items

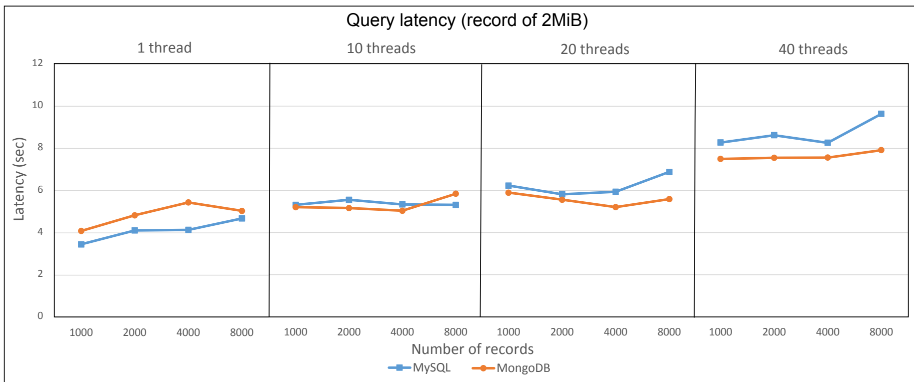


Figure 5.10: Media query latency as a function of the total number of files

The graph in Figure 5.10 shows the time taken to search for one file among a database consisting of a different number of files. Despite the number of records, each item in all the tests was an mp4 video file of 2 MiB. Now that the queried

data were of the same size, the difference in latency was purely due to the time taken to locate and read the chunks of data. However, the graph does not show a consistent difference when it comes to different total number of items, either between MongoDB and MySQL, or when only one database was concerned. Hence we believe that the total database size does not have a great impact on querying for a file, or the time taken to locate the file is very small compared to the actual time of reading the binary data and writing it to a local file.

In general, the two graphs share several common points. First, it can be seen that the performance dropped when there were more threads querying at the same time. Second, the difference between MySQL and MongoDB was inconsistent and small (most of the cases it was less than 1 second). Although it seems that MongoDB slightly performed better when there were more threads, it is hard to compare and conclude about the query performance of the two databases.

CHAPTER 6

Conclusion

The purpose of this thesis work is to investigate how different database systems can effectively handle the heterogeneous and large amount of data of the Internet of Things on the cloud, in order to meet the increasing demand on load and performance. Two classes of databases were studied, namely, SQL and NoSQL databases. While SQL databases are relational and focus on data consistency, NoSQL databases are normally schema-less and provide higher scalability and availability.

In order to assess the performance of each type of databases, several benchmarks were conducted on four different solutions: MySQL, MongoDB, CouchDB, and Redis. They represent the most widely used database systems in different contexts, and each of them has its own advantages and disadvantages. The benchmarks evaluated and compared the read and write performance of the databases as a storage for two popular kinds of IoT data: sensor scalar data and multimedia data.

The sensor scalar data benchmark showed good results for NoSQL databases, especially MongoDB. With respect to write performance, MongoDB got the smallest latency by using bulk insert with the design of all data stored in one collection, followed by MySQL, CouchDB, and Redis. However, the performance was close in query tests. Although Redis managed to achieve the best results in general, MySQL performed nearly as fast in most cases, while MongoDB lagged

behind. In contrast, the performance of CouchDB was very poor in this test as well, not to mention its huge database size compared to the others. Redis also had similar issues. This key-value in-memory database, although being very fast for querying, is limited by the database size, data structure, and query capabilities. Using a key-value store like Redis for IoT data may cause excessive computational overhead, since the variety of possible queries is not restricted to keys. Hence, the two solutions do not appear to be good candidates for a system serving IoT big data and real-time queries.

On the other hand, although MongoDB had greater query latency than MySQL, the difference was acceptable, especially considering that the system was write-intensive and MongoDB outperformed the rest when executing data insertions. In those tests, MongoDB was applied with two different designs, between which the design of one collection is more suitable for this system than the other one with multiple collections. That is because switching from the former to the latter may result in a slight improvement in querying but cause a huge lost in write performance. The lesson learned is to take advantage of the schemaless and flexible data model and consider the best fit for the system, since the change in the data model can make a huge change in performance.

Based on the results of the sensor scalar data benchmark, we conducted a similar benchmark with multimedia data on the two potential databases MySQL and MongoDB. The results show a reversed win for MySQL using BLOB storage against MongoDB's GridFS when it comes to inserting multimedia files. For query performance, the difference between the two was less pronounced, though MongoDB was slightly faster when serving more clients simultaneously. However, since multimedia systems tend to be large, the approach of MongoDB's GridFS makes it easier to shard the database across several machines, thus distributing the loads and increasing scalability.

In conclusion, it is hard to point out a clear winner for the best cloud database of IoT data, since the data types are various and the scope of use cases is vast. Moreover, each database has its own pros and cons, and its own area of application. Which database to choose therefore highly depends on the properties and requirements of the specific system. However, for such systems that were studied here, the thesis has shown the potential of NoSQL databases against the popularity of traditional relational database systems.

There are still much more room for future research about this problem. One is to expand the current benchmarks to further explore the performance of the databases with other more complicated types of IoT data, for example an object-oriented data model that involves multiple object types. That is to investigate the strength of the schema-free data model against the powerful (but expensive) use of joining data across multiple SQL tables. Another direction of research is to

assess the efficiency of scaling the system by sharding and replication, also when dealing with system failures, which was mentioned but is limited in this thesis. Scalability is actually one key point that can potentially make NoSQL win over SQL databases, considering the fact that most NoSQL databases were originally designed to scale out seamlessly to meet the growing demand of Internet data.

Bibliography

[AAS13] Charu C Aggarwal, Naveen Ashish, and Amit Sheth. The internet of things: A survey from the data-centric perspective. In *Managing and Mining Sensor Data*, pages 383–428. Springer, 2013.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[ALS10] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax*. O'Reilly Media, 2010.

[ama13] Amazon web services. http://aws.amazon.com/, Accessed: 14.05.2013.

[ASSC02] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

[Bar10] Daniel Bartholomew. Sql vs. nosql. *Linux Journal*, 2010(195):4, 2010.

[BHG87] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[Bre00] Eric A Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000.

[bso13] Bson specification. http://bsonspec.org, Accessed: 17.03.2013.

[BWHT12] Payam Barnaghi, Wei Wang, Cory Henson, and Kerry Taylor. Semantics for the internet of things: early progress and back to the future. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 8(1):1–21, 2012.

[CJ+09] Joshua Cooper, Anne James, et al. Challenges for database management in the internet of things. *IETE Technical Review*, 26(5):320, 2009.

[CLR10] Michael Chui, Markus Löffler, and Roger Roberts. The internet of things. *McKinsey Quarterly*, 2:1–9, 2010.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod13] CodeFutures Corporation. Database sharding. http://www.codefutures.com/database-sharding/, Accessed: 17.05.2013.

[cou13] Couchdb, a database for the web. http://couchdb.apache.org/, Accessed: 16.05.2013.

[Cro06] D Crockford. Rfc 4627-the application/json media type for javascript object notation. Technical report, Technical report, Internet Engineering Task Force, 2006.

[Dat13] Datastax Corporation. *Benchmarking Top NoSQL Databases*. Datastax, 2013.

[DFLRD12] Mario Di Francesco, Na Li, Mayank Raj, and Sajal K Das. A storage infrastructure for heterogeneous and multimedia data in the internet of things. In *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pages 26–33. IEEE, 2012.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DXY12] Zhiming Ding, Jiajie Xu, and Qi Yang. Seaclouddm: a database cluster framework for managing and querying massive heterogeneous sensor sampling data. *The Journal of Supercomputing*, pages 1–25, 2012.

[FL05] Steve Fogel and Paul Lane. Oracle database administrator's guide, 2005.

[GH06] Simson Garfinkel and Henry Holtzman. Understanding rfid technology. *RFID*, pages 15–36, 2006.

[GR11] John Gantz and David Reinsel. Extracting value from chaos. *IDC iView*, pages 1–12, 2011.

[GR12] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. Technical report, Technical report, IDC, 2012.

[GT09] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, 2009.

[Hed13] Martin Hedenfalk. How the append-only btree works, 2011. http://www.bzero.se/ldapd/btree.html, Accessed: 17.03.2013.

[HHLD11] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[HJ11] Robin Hecht and Stefan Jablonski. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341. IEEE, 2011.

[JPA+12] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering*, 1(6), 2012.

[KAB+11] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. On the elasticity of nosql databases over cloud management platforms. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2385–2388. ACM, 2011.

[Lai09] Eric Lai. No to sql? anti-database movement gains steam. *Computerworld Software, July*, 1, 2009.

[Lea10] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[LLT+12] Tingli Li, Yang Liu, Ye Tian, Shuo Shen, and Wei Mao. A storage solution for massive iot data based on nosql. In *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pages 50–57. IEEE, 2012.

[LMR08] Avinash Lakshman, Prashant Malik, and K Ranganathan. Cassandra: Structured storage system over a p2p network, 2008.

[LMS05] Paul J Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. 2005.

[MCO10] Vladimir Mateljan, D Cisic, and D Ogrizovic. Cloud database-as-a-service (daas) – ROI. In *MIPRO, 2010 Proceedings of the 33rd International Convention*, pages 1185–1188. IEEE, 2010.

[mon13] The mongodb manual. http://docs.mongodb.org/manual, Accessed: 16.05.2013.

[MSPC12] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications & research challenges. *Ad Hoc Networks*, 2012.

[MyS13a] MySql Developer. Mysql documentation: Mysql 5.6 reference manual. http://dev.mysql.com/doc/refman/5.6/en/, Accessed: 16.05.2013.

[MyS13b] MySql Developer. Mysql documentation: Mysql cluster. http://www.mysql.com/products/cluster/, Accessed: 17.03.2013.

[Ore10] Kai Orend. Analysis and classification of nosql databases and evaluation of their ability to replace an object-relational persistence layer. *Master's thesis, Technische Universität München*, 2010.

[PCP12] Antonio Pintus, Davide Carboni, and Andrea Piras. Paraimpu: a platform for a social web of things. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 401–404. ACM, 2012.

[PPS11] Rabi Prasad Padhy, Manas Ranjan Patra, and Suresh Chandra Satapathy. Rdbms to nosql: Reviewing some next-generation non-relational databases. *International Journal of Advanced Engineering Science and Technologies*, 11(1):15–30, 2011.

[Pri08] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[pro13] Project voldemort, a distributed database. http://www.project-voldemort.com/voldemort/, Accessed: 11.03.2013.

[red13] Redis. http://redis.io/, Accessed: 04.06.2013.

[RGVS+12] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012.

[Rod08] Alex Rodriguez. Restful web services: The basics. *Online article in IBM DeveloperWorks Technical Library*, 36, 2008.

[Seg10] Karl Seguin. The little redis book. *Karl Seguin*, 2010.

[Sen10] Jaydip Sen. Internet of things-a standardization perspective. *This article is property of Tata Consultancy Services*, 2010.

[SGFW10] Harald Sundmaeker, Patrick Guillemin, Peter Friess, and Sylvie Woelfflé. Vision and challenges for realising the internet of things. *Cluster of European Research Projects on the Internet of Things (CERP-IoT)*, 2010.

[Siv13] Swami Sivasubramanian. Synchronous vs. asynchronous replication strategy: Which one is better? http://scalingsystems.com/, Accessed: 17.05.2013.

[SSK11] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011.

[TAB+05] Ken Traub, Greg Allgair, Henri Barthel, L Bustein, John Garrett, Bernie Hogan, Bryan Rodrigues, Sanjay Sarma, Johannes Schmidt, Chuck Schramek, et al. The epcglobal architecture framework. *EPCglobal Ratified specification*, 2005.

[TB11] Bodgan George Tudorica and Cristian Bucur. A comparison between several nosql databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*, pages 1–5. IEEE, 2011.

[the13] There corporation. http://www.therecorporation.com/en/products/, Accessed: 04.06.2013.

[Tiw11] Shashank Tiwari. *Professional NoSQL.* Wrox, 2011.

[VD10] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting smart objects with ip: The next internet.* Morgan Kaufmann, 2010.

[VDMC10] Roberto Verdone, Davide Dardari, Gianluca Mazzini, and Andrea Conti. *Wireless sensor and actuator networks: technologies, analysis and design.* Academic Press, 2010.

[vdVvdWM12] Jan Sipke van der Veen, Bram van der Waaij, and Robert J Meijer. Sensor data storage performance: Sql or nosql, physical or virtual. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 431–438. IEEE, 2012.

[Voe12] W Voegels. Amazon dynamodb–a fast and scalable nosql database service designed for internet-scale applications. *Retrieved July*, 30:2012, 2012.

[Vol10] VoltDB LLC. Voltdb technical overview, 2010.

[Wan06] Roy Want. An introduction to rfid technology. *Pervasive Computing, IEEE*, 5(1):25–33, 2006.

[Wan11] Roy Want. Near field communication. *Pervasive Computing, IEEE*, 10(3):4–7, 2011.