Building and Benchmarking NoSQL DBMS with YCSB

Studienarbeit by Zheng Wang

April 2011

Mentor:  Rico Bergmann
# Contents

1 Introduction 1
   1.1 Schema-Free and Fixed Schema 1
   1.2 SQL, XQuery and JAQL 2
   1.3 Parallel Database and Distributed Database Systems 3
   1.4 Scalability 6
   1.5 MapReduce 6

2 NoSQL 7
   2.1 CAP Theorem and BASE 7
   2.2 NoSQL’s Alternative Data Model 8
      2.2.1 Column Family 9
      2.2.2 Document-Oriented Model 10
      2.2.3 Structured Document-Oriented Data Model 10
      2.2.4 Blob Data Model 11
   2.3 Key-Value Store 11

3 Project YCSB 12
   3.1 Architecture of YCSB 12
   3.2 YCSB Workloads 13
      3.2.1 Test Data for the Core Set of Workloads 13
      3.2.2 Supported Distribution 13
      3.2.3 YCSB DB Interface Layer 15
   3.3 Extensibility 16

4 Overview of Benchmarked Database Systems 17
   4.1 Overview of CouchDB 17
      4.1.1 Architecture of CouchDB 17
      4.1.2 Web-Based APIs 19
      4.1.3 Document-Oriented Storage Strategy 20
      4.1.4 Views and MapReduce 21
      4.1.5 Clustering with CouchDB Lounge 22
   4.2 Overview of Voldemort 24
      4.2.1 System Architecture 24
### CONTENTS

4.2.2 Simple Queries of Voldemort .............................................. 26  
4.2.3 Data Model of Voldemort ................................................ 26  
4.2.4 Clustering and Replication .............................................. 27  
4.2.5 Multi-Version Control of Voldemort ................................. 28  
4.3 Overview of MemcacheDB ................................................. 30  
4.3.1 Memcached ................................................................. 30  
4.3.2 System Architecture ..................................................... 30  
4.3.3 API of MemcacheDB ...................................................... 31  
4.3.4 Replication and Consistency .......................................... 32  

5 Benchmark with YCSB ......................................................... 33  
5.1 The DB Layer Interface of CouchDB ................................... 33  
5.2 The DB Layer Interface of Voldemort ................................. 34  
5.3 The DB Layer Interface for MemcacheDB .............................. 35  
5.4 Benchmark Environment .................................................. 35  
5.5 Results of Benchmark ...................................................... 37  

6 Summary ................................................................. 41  
6.1 Summary of Work .......................................................... 41  
6.2 Conclusion ................................................................. 41  
6.3 Outlook ................................................................. 42
# List of Figures

2.1 An Example for Column Family ........................................... 9  
3.1 Architecture of YCSB .................................................. 12  
3.2 Distribution used by YCSB ............................................ 14  
4.1 Architecture of CouchDB ............................................... 17  
4.2 An Example of No-Delete Policy ..................................... 20  
4.3 An Example of Clustering with CouchDB Lounge .................... 23  
4.4 Logical Architecture of Voldemort ................................. 24  
4.5 Hashing Ring ............................................................ 29  
4.6 Architecture of MemcacheDB .......................................... 31  
5.1 Workload A: Update-Heavy ........................................... 37  
5.2 Workload B: Read-Heavy ............................................. 38  
5.3 Workload C: Read-Only ............................................... 38  
5.4 Workload D: Read-Heavy with Insert ................................ 39  
5.5 Workload F: Read-Heavy with ReadModifyWrite .................... 40
Abstract

In the last 10 years, centralized Relational Database Management Systems (RDBMS) were challenged by applications from some highly successful Web 2.0 companies. Although centralized RDBMS faithfully kept ACID properties, the tasks their applications were best suited to were partitioning data and spreading the workload across servers. Cloud data serving systems have been receiving most attention in this area recently, for example BigTable [CDG⁺06], PNUTS [CRS⁺08], Cassandra [LM10], Hbase [Fou10], Azure [Li09], CouchDB [Len09], SimpleDB [CH10], and so on. Many of them can be categorized with NoSQL [Sto10], a class for non-relational, distributed, open-source, and horizontal scalable data serving systems. Some database architects claimed that ACID properties could be violated and traded for availability and partition-tolerance in the domain of NoSQL. According to their opinion, many NoSQL DBMS were developed with their individual data models to meet various demands of the market. The trade-offs between different systems are always difficult to understand. To choose an appropriate database among NoSQL DBMSs for a workload is therefore a problem for companies.

YCSB (Yahoo! Cloud Serving Benchmark) was developed to provide a framework to benchmark various systems. Four widely used Cloud serving systems are chosen by YCSB, namely Cassandra, Hbase, PNUTS, and simple shared MySQL. Additional cloud benchmarking suites can be developed to make YSCB available for other systems. The workloads generator makes it easy to define new workloads to benchmark systems. In this paper, YCSB was used as a benchmark tool for the performance comparisons among cloud systems and extended to three other systems: CouchDB, Voldemort, and MemcacheDB. The performance of these three systems was studied.
Chapter 1

Introduction

Since the relational model was introduced by E. F. Codd, most commercial and open source databases have followed the principles of the relational model. RDBMS have dominated the database market for decades.

The relational database is conceptually a collection of tables filled with data. A table contains a finite number of columns. Each column contains data in a category; each row of a table presents a unique data item which contains data in all categories defined in this table. The relational model provides both a declarative method for specifying data and a set of methods to manipulate the data with queries. Traditionally, most companies and organizations applied relational databases for their services.

Besides the relational model, there are many other data models such as the network model, the hierarchical model, and the object-oriented model. More and more non-relational databases without SQL interface were developed to provide alternative choices for users. This development in the database industry is called NoSQL Movement.

1.1 Schema-Free and Fixed Schema

A database schema of a database system is a structured plan in a certain formal language supported by database systems to describe the organization of data to demonstrate how the database is structured. The schema policy of a database can be schema-free or fixed schema. A database with fixed schema requires users to give the database the complete information about the structure of the stored data, while schema-free database allows users to store and access their data with only partial knowledge of the schema.

The relational model is inherently restricted to be fixed schema. The fixed schema benefits the relational model with less complexity and more power. If the organization of data is predefined, the number of columns is finite, the order of columns can be specified before constructing a table, and the way to access particular data in a table is determinate.

The fixed schema policy has its weaknesses. A database with fixed schema is unsuitable for a situation in which the user does not know the complete schema or the contents of data are unpredictable. If a user wants to add a new column in a table in an RDBMS, traditionally, the table is rebuilt with the new column by importing original data from the
old table. It is normally an expensive operation. The data integration in the relational model is difficult, too. In this case, two schemas have to be twisted to fit the new schema semantically. Schema-free models allow users to arrange their schema with more flexibility to handle both situations. Because in a schema-free DBMS the number of columns is theoretically unlimited, the user can add a column for some or all rows without changing the remaining data. By data integration, a schema-free DBMS can build a bigger table whose set of columns is the union of the sets of columns of the two tables. Most NoSQL database systems are designed as schema-free database system to handle unstructured data which is difficult to be decomposed into relations such as images, documents, blobs, and so on.

1.2 SQL, XQuery and JAQL

SQL (Structured Query Language) is designed for managing RDBMS, and almost every RDBMS supports SQL. The efficiency of SQL was proven in the last decades in dealing with tabular data, but SQL is difficult to be used to handle unstructured data [Lea10].

XQuery [Cha03] is designed for extracting and manipulating XML data. XML presents its data in trees rather than tabular as the relational model does. An XML data file can be either schema-free or with fixed schema. In the relational model, each row in a table contains data in the same categorie. Therefore, a DBMS can find specific data by reading all rows. However, this method does not work with semi-structured data because a semi-structured data object can theoretically contain an infinite number of columns. XQuery uses XPath (XML Path Language) for selecting the node which describes the path of nodes to address data in an XML document. XQuery searches for data without assuming that all possible addresses return a data item as a response, and only the data item in the path is referred to by XQuery, so it allows nodes to have different structures. XQuery takes all data as an XML file and manipulates the data in a tree-like structure. Structured data in the relational model can be easily presented as a structure in the form of XML; therefore XQuery can be used to handle data in the relational model, too.

JAQL [BES10] is a query language for JSON (JavaScript Object Notation) [JSO], a semi-structured data form. Both JSON and XML are designed for semi-structured data. Compared to XML, JSON is a lightweight data model, but it supports primitive data types and arrays. JSON uses no text markups, so it is normally tighter than XML.

JAQL follows the query method of XQuery: query by path. To gain a high performance of scaled-out architectures with a great amount of data, JAQL allows parallel IOs to enable more parallelism.

The variety of query languages provides alternative opportunities to build databases to handle structured and semi-structured data. Many RDBMS begin to evaluate new functions to query semi-structured data, for example SQLXML from IBM’s DB2 [NvdL05] and Microsoft’s SQL Server 2005 [BS06]. NoSQL databases normally have abandoned the relational model and SQL. Each NoSQL database system has a different approach, and each of them has its own special way to deal with semi-structured data.
1.3 Parallel Database and Distributed Database Systems

According to Moore’s law [Moo00], the number of transistors on a chip doubles every 24 months. Some researchers claimed that this number may be 6 months shorter than Moore’s prediction. Compared to CPUs, the access time of disks has improved more slowly. The growing data volume demands a higher performance of CPUs and a larger capacity of disks. The slow disk throughput becomes the bottleneck. To solve this problem, hardware architects began to use parallelism to improve the performance of computers. Instead of working out a large task in a long time, the parallelism divides a large task into subtasks and each of them is run simultaneously on different processors to make the processing time shorter. The bottleneck of disk throughput is also solved by using multi-processors and disks.

The development of databases inevitably met the disk throughput bottleneck, too. Databases are required to hold an enormous size of data and to be accessed by a large number of connections. DBMS were driven to query large databases and work out large numbers of transactions per second. Parallel database technologies and distributed database technologies were researched to apply parallelism to database systems.

Typically, a parallel database system assumes only a single administrative domain, a homogeneous working environment and a close proximity of data storage [DG92]. Under the term parallel database technologies, four types of parallelism are defined for database processing.

- **Interquery Parallelism** is the parallelism among queries. This means different queries or transactions are processed parallel. With a single-processor computer, a queue is built for queries. The computer processes only one query at a time, other queries are waiting in the queue. The interquery parallelism enables multi-processor computers to take more than one process into the queue. If, for example, each processor takes one query from the queue, the interquery parallelism reduces the waiting time for each process.

- **Intraquery Parallelism** is the parallelism within a query. A query can be divided into small subqueries and each subquery is carried out on a processor. By executing a single query with multiple processors and disks, the evaluation of every single query speeds up.

- **Intra-Operator Parallelism** is the parallelism among subqueries. The intra-operator parallelism introduces a parallelism in each subquery by partitioning it into operations. For example, a table can be partitioned to subsets of rows and each subset is executed by one operation. Therefore the intra-operator parallelism also has another name: partition parallelism.

- **Inter-Operator Parallelism** is a parallelism created by executing different operations of a query parallel and concurrently. For example, a pipelined parallelism is one of this kind. In pipelining, a query is divided into operations in a sequence like an assembly line. The second operation consumes the result of the first operation and so on.
A distributed database is a collection of multiple, logically related databases which are geographically distributed over a network [OV91a]. Each site of a distributed database in the architecture of a distributed database is always assumed as an independent computer which has its own storage components, processors, operating systems and DBMS. The communication between sites is over a network rather than a multiprocessor architecture such as sharing memories. With distributed databases the working environments may be heterogeneous, this means hardware and data model can be different from site to site.

[CSW87] suggests that fragmentation and replication are two methods to distribute a database across data sites. There are two intuitive ways to divide it into fragments: horizontal fragmentation and vertical fragmentation. A horizontal fragment of a table contains only a subset of tuples due to the applied fragmentation predicates, for example, a table about bank customers can be fragmented by their addresses. Vertical fragmentation divides a table into multiple sub-tables, and each of them consists of all tuples but each tuple contains only a subset of columns. For example, we can divide the table about bank customers into two tables: one contains the bank information and one contains only personal information about customers. The usage of fragments benefits a distributed database to store information at the most desired location to reduce transmission costs and the size of the table for queries.

Replication is a method to increase consistency by redundant resources and increase access efficiency by providing more copies in different sites. The algorithm for replication has two major goals: reading and writing data from copies and concurrency control to synchronize operations on different sites. The correctness of replication technology requires the replication algorithm to manage the copies like a single copy [BG83a]. To solve these problems, several protocols were developed. Read-one Write-all (ROWA) [BHG87] allows a transaction to read only one of the copies, but writing happens on all copies. In this protocol, read-only transactions read only one copy because if the value is changed by another transaction, this change will affect all copies, however, the write transaction requires all copies to be available for writing. Another protocol is quorums [BG83a]. It is a problem for ROWA when only a subset of sites is available. To cope with this situation and avoid inconsistencies, transactions have to obtain a subset of copies for reading and writing.

From the architecture perspective of parallel database systems and distributed database systems, they can be classified into three categories: shared-memory (all processors share a common main memory and secondary memory, sometimes cache, too), shared-disk (each processor has its own main memory but they share the secondary memory), and shared-nothing (each processor has its own main memory and secondary memory, they communicate with each other by messages via an interconnected network). The difference between shared-memory and shared-disk is whether the processor has its own main memory. Both of them are strong at load balancing because the data is located in the same place for all processors. Compared to shared-disk, shared-memory is simpler, because with shared-disk, each processor can copy a page into its own memory which needs global locking and protocol of maintenance of memory to avoid conflicts, but it is not as extensible and available as shared-disk. The shared-nothing approach has strong advantages in extensibility and availability. The shared-disk architecture is normally built with SMP (Symmetric
Multi Processor) machines, but a machine with a large number of CPUs is rare due to the scaling-up limitation [TLRG08]. Therefore, the shared-disk approach is limited with high complexity and potential performance. In the shared-nothing approach, each processor has the exclusive authority over its main memory, the secondary memory and the processors are connected with a network. Although the shared-nothing architecture has the highest extensibility and availability, but it suffers from loading balance problems because the local data in each processor may be unequal, so redistribution is sometimes needed. Another problem of the shared-nothing approach is that data in different processors may differ from each other. Although the shared-nothing approach is the most complex one, [DG92] suggests that shared-nothing is the trend for its extensibility and availability.

In the ideal state, if the computing resource doubles, the system doubles its power. In practice, the increase of computing power is not linear. Three major reasons obstacle this process: start-up and consolidation, interference, and skew [TLRG08].

- **Start-Up and Consolidation:** It costs some time for an operation to initiate multiple processes and after that it also costs some time to collect all sub-results from processors for the final result. The time cost by starting up and consolidation is sometimes more than the actual processing time.

- **Interference:** If the multi-processes share some resources exclusively, they have to communicate with each other to obtain the shared resources.

- **Skew:** the total processing time is determined by the slowest process. If the job is not evenly partitioned to all processors, some processors have to wait after their task and some processing power is wasted.

To measure the efficiency of a parallel system, two metrics are used, namely speed-up and scale-up [DG92]. Speed-up measures the performance improved by additional computing units.

\[
\text{Speed-up} = \frac{\text{uniprocessor elapsed time on small system}}{\text{multiprocessor elapsed time on large system}}
\]

Scale-up measures the effect of increasing the degree of parallelism.

\[
\text{Scale-up} = \frac{\text{elapsed time on uniprocessor}}{\text{elapsed time on multiprocessor}}
\]

Ideally, both scale-up and speed-up should be equal to 1. However, in practice, the two values of a database system are influenced by many elements such as the system architecture, parallel algorithm, data models, and so on. [DG92] shows that parallelism benefits relational databases and the relational queries are just relational operations which offer many opportunities for parallelism. Queries represented by SQL can be translated into dataflow graphs. The pipeline parallelism and the partition parallelism are frequently used to execute such dataflow graphs in practice. The pipeline parallelism is limited by the following constraints: (1) Relational queries are rarely long, so the pipeline is always
short. (2) Some operations have to wait for all their input to emit the first result. (3) The cost of one operation can be much greater than that of the others. Compared to pipeline parallelism, the partition parallelism does better by using a divide-and-conquer strategy. It is also suggested that the data should be partitioned in the key for partition execution.

There is no overview of functionality for NoSQL DBMSs. Each NoSQL DBMS uses its special data model and algorithms to provide opportunities for parallelism. Their performance is still to be studied.

1.4 Scalability

Theoretically, a distributed system can be expanded with adding new nodes or upgrading existing nodes. We have mentioned that the scale-up in practice is smaller than 1. The scalability, the ability to increase the power or capacity of system by adding components, is a desirable property of a distributed system.

Additional resources provide larger commodities for growing data volumes or make the computing faster by enlarging the memory capacity to reduce the time cost of IOs or provide more computing units to increase the parallelism. This method is called vertical scaling. The decreasing cost of massive disks, RAMs and CPUs makes vertical scaling one of the simplest and most commercial methods.

Another method is horizontal scaling: adding more machines to the system environment. Horizontal scaling has two advantages: 1) The differences between machines in a distributed system are normally transparent to the user. The user never cares about which data is stored on which machines but the management system guarantees the correctness of data manipulations. 2) It provides tolerance against failures. If one server is shut down, other servers can take over its job until it comes back and is re-available to clients again.

1.5 MapReduce

MapReduce [DG08] is a programming model and an associated implementation for processing and generating large data sets, especially in distributed systems. Two functions (Map and Reduce) are used. The Map function processes a key-value pair to generate multiple intermediate output pairs in the key-value form. The Reduce function consumes the output pairs from a Map function. It merges all intermediate values by keys and outputs aggregated key-value pairs. Normally the output values are smaller than before and typically only zero or one key-value pair is generated by the Reduce function. MapReduce allows users to handle a large volume of data with a small memory. This highly scalable and failure-tolerant programming model is employed widely in commodity machines.
Chapter 2

NoSQL

NoSQL is a group of non-relational, distributed, open sourced, and horizontally scalable databases. NoSQL databases are built for web services by removing many features of traditional relational databases to gain a character essential to web serving. Most of them are schema-free, easily replicated, simple API, lightweight and eventually consistent.

Although RDBMS have many advantages, they also have limitations. For example, building a distributed database system with RDBMS is not so easy because joining tables across nodes may be costly in partitioned data environments. The SQL language limited RDBMS in handling semi-structured data. Transactions obstacle the performance improvement of RDBMSs in some way too. The definition of NoSQL aims at the limitations of RDBMSs to provide alternative choices.

2.1 CAP Theorem and BASE

In 2000, Eric Brewer announced the CAP [Bre00] (Consistency, Availability and Partition-tolerance) concept in a presentation at the ACM symposium on the principles of Distributed Computing. Traditionally, most RDBMS follow ACID (atomicity, consistency, isolation, durability) to guarantee the reliability of transactions. Atomicity requires that modifications of transactions follow the ”all or none” rule; consistency allows transactions only to transform a database from a consistent state to another consistent state; isolation ensures that transactions do not affect each other; durability refers to the requirement that the effect of a transaction is recoverable and survives failures. However, the CAP theorem sees the problem from another angle with three features of DBMS.

- **Consistency:** Most web services and applications traditionally depend on the capacity of providing consistent data. Transactions are employed to handle the interactions between database and web service. This property guarantees that all operations are in total order. Therefore, two essential requirements of distributed databases should be fulfilled: operations work on all copies like a single copy and concurrency control.
CHAPTER 2. NOSQL

- **Availability:** Every request received by a non-failing node must result in a response [GL02]. Availability is essential to web services for commercial and legal reasons. Failure of a single node should not prevent the continuation of a work or lead to the eventual termination of a service.

- **Partition tolerance:** In a distributed DBMS, data is stored at different places geographically and machines are connected with a high speed network. The distributed DBMS permits that the management is transparent to users and applications [OV91b]. When some nodes are not accessible for some reasons, it is important for the system to perform as formerly expected and those failures should still be transparent to the users. This property is called partition tolerance, which also guarantees the availability by requiring all service terminated with a respond.

Brewer’s Theorem proves that these three core requirements of modern web services are impossible to meet together in a distributed system at the same time, but only two of them can be met. Two years later, in 2002, Seth Gilbert and Nancy Lynch of the MIT proved Brewer’s theorem formally in [GL02].

As a consequence of dropping partition-tolerance, DBMS have to drop partitioning. This means they have to store all data on one node. In this case the scale-up performance is limited. Dropping availability means fewer services and lower reliability. It is almost the same thing to drop partition tolerance. They are two sides to this coin. In the mid-nineties database system architects even thought availability was the most essential property to their systems, but they had no idea which property should be traded against it [Vog09]. Dropping consistency finally appears to be the better option to tackle the problem. Noticed that, not all applications ask for a strictly consistent data service, and providing consistent data can be also guaranteed by user applications with a better plan of application logic, so DBMS can be released from strict consistency. The concept of eventual consistency is introduced by Werner Vogels in [Vog09] to define a weak consistency for this problem. Eventual consistency permits that all accesses return the last updated value eventually in a determined inconsistency window time if no new updates are made to the object.

The concept of eventual consistency is supported by the new architectural approach BASE (Basically availability, soft state, eventual consistent) [Pri08]. BASE optimistically accepts that database consistency will be in a state of flux. The BASE consistency model allows partitioned databases to sacrifice some consistency in a time window to gain more availability by increasing the level of scalability.

### 2.2 NoSQL’s Alternative Data Model

Due to the CAP theorem, strict consistency is dropped to build new DBMS with higher availability and partition tolerance. Sharding, holding rows of a database separately on separated databases or physical locations, is usually used to achieve this goal. There is a problem for RDBMS: the join operator does not cope with sharding easily. A join operation with sharding can cause a set of nodes to be too busy to transmit their data between each other to evaluate a query. If the join operation is dropped by RDBMS, the database is
2.2. NOSQL’S ALTERNATIVE DATA MODEL

degraded to a simple data storage system because the relational model has few methods to
describe the interrelationship between tables without join operations in query expressions.
To prevent this problem, some new data storage models were developed to meet the
demand of massive data volume storage and crowd traffic caused by high scalability. The
data models implemented among NoSQL database systems are various, such as Column
Family, the document-oriented data model, Blob and the structured document-oriented
data model.

2.2.1 Column Family

Cassandra (BigTable, Dynamo [DHJ+07]) and HBase (BigTable [CDG+06]) use Column
Family as their data storage model. In Column Family, a column is the elementary unit
of data (e.g. the object alex.users.email in Figure 2.1). A column is formed by three
properties: column name, value and timestamp. All values in a column are provided
by clients, including the timestamp. The value of the timestamp can be used to resolve
version conflicts by ordering the updates. By tracking ordered versions, version conflicts
can be resolved automatically or left to the user’s decision.

```
{
    "adam":{
        "contact":{
            "email":{"name":"adam", "value":"adam@adam.com"},
            "web":{"name":"web", "value":"http://adam.com"}
        },
        "Stats":{
            "visits":{"name":"visits", "value":243"}
        }
    },
    "alex":{
        "Users":{
            "email":"alex",
            "value":"alex2@adam.com",
            "timestamp":398765567
        }
    }
}
```

Figure 2.1: An Example for Column Family

A supercolumn is a special kind of column, it has only two properties: name and
value, with the value defined as a map which can store an arbitrary number of columns.
In Figure 2.1, object adam.contact is a supercolumn with two items in a map as its value,
object stats is a supercolumn with only one item.

A column family, which is referenced by the column name, is a container of an ordered
list of columns or supercolumns. In Cassandra, each column family is stored in a separate
file. The manipulation of data is performed on column families. For example, in Figure
2.1 the object adam is a column family containing two supercolumns contact and stats.

A row with a row key contains exactly one column family. The row key determines
where the referenced column family is stored. In Cassandra, a hash function is employed
here to calculate the hash value for each row key. Then the hash values of row keys show
which machines contain the wanted files. For example, in Figure 2.1, the data file contains the two rows `adam` and `alex` and they are uniquely identified with row keys.

The Column Family data model is similar to a table in the relational data model for its dividing data into rows and columns, but there are some differences. Because a column family is a container of columns or supercolumns, a table in a column family theoretically has an unlimited number of columns. This means in a logic table each row can contain different types of columns. The number of columns can be expanded by inserting new columns into a column family.

### 2.2.2 Document-Oriented Model

The document-oriented data model [AmFL+95] does not limit itself in the framework of tables as other data models do. Compared to the relational data model, the document-oriented data model is content-intensive [SFVM06][SFmFvN05]. Content-intensive applications depend on the domain informational contents and on the explicit representation of the structure of these contents. Although sometimes database systems store multiple documents together as a file, they are logically independent from each other like separate files. Each document in the database can hold different properties. This character benefits database systems with low costs to change the definition of every row in the database. A document-oriented database system usually leaves the content of documents to users and lets users decide which semantics a document has. The elementary unit for transactions is the document for the majority of document-oriented NoSQL databases such as MongoDB, CouchDB and Hypertable. The document-oriented data model also supports structured or semi-structured documents, for example JSON and XML.

The document-oriented data model has three advantages:

- **high comprehensibility:** The information stored in this model is normally readable to the user. All attributes in a document are directly represented and editable.

- **high processability:** Data stored in the document-oriented data model is machine-processable, translation from raw information into processing-oriented data structures is possible and translation tools are available in major programming languages and operating systems.

- **high reusability:** The document-oriented data model does not deliver any business logics.

### 2.2.3 Structured Document-Oriented Data Model

The structured document-oriented data model is a specific document-oriented data model. Although some database systems, such as Voldemort, store values as strings or blobs, they allow users to compose a schema for storage. The database system needs a parser to work with the schema to specify output and input values, and then it can check, maintain, and retrieve the data. The database can create a parser according to the schema, or the user has to provide one.
2.3. **Key-Value Store**

2.2.4 **Blob Data Model**

A blob (binary large object or basic large object) is a collection of binary arrays as a single object. Traditionally, blobs are used for images, audios, and videos in relational databases. The content of a blob is usually not indexed, but some functions are supported to manipulate it. In the NoSQL world, blobs are used to store objects [Li09]. The blob data model provides more freedom for data organization than the document-oriented data model because every serializable data copes with the data structure in the blob. For example, fragments in the main memory can be mapped to blobs.

2.3 **Key-Value Store**

Key-value store is a system that stores its data in the form of key-value pairs. In key-value stores, values can be stored, indexed and retrieved by a key. The content of a value can be structured or unstructured data. Some NoSQL databases enable the schema definition on values and provide parsers to translate values to objects, but the key-value store has no constraints on values.

A table in relational database is a single structure. If a writing operation is going to be carried out on a row, the database system must ensure that there is no other process modifying the row. Each key-value pair in a key-value store is similar to a row in a table of a relational database. Transactions in a relational database may refer to multiple rows in tables because of the join operator. Key-value stores allow join operators, too, but the concurrency management is typically seldomly used across multiple key-value pairs. Each key-value pair is thought to be independent from the others, so there is no need to apply a two-phase locking [BHG87], multi-version concurrency control (MVCC) [BG83b], or optimistic concurrency control (OCC) [KR81] on multiple key-value pairs.

Key-value stores are widely implemented in distributed systems. Fragmentation can be easily implemented by sharding with hashing the key space. The opportunity of replication is offered by the key-value store, but the resolution of versions of values is not provided by the key-value store. Some NoSQL DBMS use additional information to solve this problem, for example Voldemort and Dynamite use *vector clocks* to store updating information, some write the version information into values to identify the version like CouchDB.
Chapter 3

Project YCSB

Many NoSQL databases proposed for cloud serving have been developed recently, such as Hbase, Cassandra, Azure, CouchDB, Voldemort, MemcacheDB, SimpleDB and many others. The booming number of NoSQL databases provides many alternatives for building cloud services, but the study of performance comparisons between them is lagged compared to the progress of NoSQL databases development. Yahoo developed a benchmark tool called Yahoo! Cloud Serving Benchmark (YCSB) \cite{CST10} to meet the performance comparison demand. YCSB provides two essential things: a framework for benchmarking and the definition of a core set of workloads for benchmarking.

3.1 Architecture of YCSB

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{architecture.png}
\caption{Architecture of YCSB \cite{CST10}}
\end{figure}
3.2. YCSB WORKLOADS

As shown in Figure 3.1, the YCSB Client is formed by the components Workload Executor, DB interface layer, client threads, and stats.

The Workload Executor is initialized with user interpreted a series of properties, which define the YCSB Client’s operations. It checks all properties of a benchmark, drives multiple client threads and a status thread. A benchmark is run in two phases: the load phase to load the database and the execution phase to execute the workload.

Each client thread drives a DB layer interface to simulate a connection. It inserts data into the database in the load phase and executes transactions according to the workloads setting in the execution phase. The status thread gathers statistics from client threads and retrieves a aggregated result to the user in a time interval (e.g. every ten seconds).

The YCSB client reports the aggregated result under different traffics. It manages the maximal number of operations per second. If this number is reached in the time interval, YCSB suspends all client threads and restarts them at the beginning of the next time interval. Therefore, YCSB can simulate the running environment under different traffics. If there is no speed limitation, the YCSB client tries to execute as many operations as possible to measure the statistics in an extreme situation.

The DB layer interface calls against the database to execute the requests from the client threads. It tells YCSB how to insert data objects and execute transactions.

3.2 YCSB Workloads

To estimate the performance of a database system, YCSB defines a core set of workloads, which describes the benchmark environment. A workload in YCSB consists of three parts: the property of a scenario, the distribution of test data and the frequency of all kinds of transactions to be executed.

3.2.1 Test Data for the Core Set of Workloads

YCSB manipulates data only in a single table: usertable. By default, each data object contains ten fields whose names are automatically generated. Each field is filled with randomly generated ASCII characters with a default size of 1 KB. YCSB assumes that fields are treated as a blob. So the content of fields is not important for YCSB. Data objects are identified by keys which are indexed or hashed. The performance of a key-value store depends on the management of keys. Each key in YCSB contains two parts. One is the string ”user”; the other is an integer with fixed length. Integers are generated for a given distribution.

3.2.2 Supported Distribution

In practice, the distribution of workloads affects the performance of distributed systems. The distribution algorithm of a database determines which partition contains which data. If the database fails to balance workloads among nodes, some nodes are busy all the time while others have nothing to do. This phenomenon is called ”skew”.

1source: Benchmarking Cloud Serving System with YCSB
The core set of workloads generates keys for objects in different distributions to test the ability of the database's distribution algorithm. There are three distributions available: uniform, Zipfian, and latest.

In the core set of workloads, the user defines the range of integers for keys from zero to an upper bound. In uniform distribution integers are equally distributed by the random function in Java.

The Zipfian distribution [Li92] simulates the situation that keys are related to the natural language. It follows the distribution of natural language. If words are aligned according to the frequency of their appearance in a table, the first word in the table appears \( n \) times more often than the second one. Analogously the second word appears about \( n \) times more often than the third one, and so on. That means the frequency of any word is inversely proportional to its rank in the frequency table. If Zipfian distribution and the range are set, the integers are generated by the rule: the bigger the integer is, the former is the integer in the frequency table.

The latest distribution is used to generate keys which are highly combined with hot topics such as newly generated blogs, photos, news, and stories. The latest distribution in YCSB uses the idea of Zipfian distribution, but the frequency table is built dynamically. The integer generator sets the most recently used integer as the top item in the table. Therefore, the frequency of keys meets the description of the latest distribution.

Figure 3.2: Distribution used by YCSB

In Figure 3.2, diagrams for the three distributions are drawn in the range from 0 to 1000. The X axis of both uniform and Zipfian distribution ranges from 0 to 1000, and the Y axis is the frequency of integers. To illustrate the latest distribution better, we sort the integers by the order of their appearance with the integer last to appear having the largest
3.2. YCSB WORKLOADS

We take the reordered sequence as the X axis and the Y axis is the frequency. The number of items in this latest distribution’s figure is less than 1000 because some integers did not show up.

The final key names are not created directly from generated integers because they still relate to their values. Each integer is mapped to an integer with fixed length in decimal numbers by a hash function.

3.2.3 YCSB DB Interface Layer

A DB interface layer is the abstract interface to connect to databases or cloud serving systems. Each DB interface for a concrete database extends from this abstract interface. There are six functions to be implemented to realize a DB interface:

- **Init:** The Init function initializes a connection to the tested database for each YCSB client. Because the YCSB is a multi-threaded program, several YCSB clients will run simultaneously and each client holds a connection to the database.

- **Read:** Read is the function to read an object from the database. Every returned data from databases is put into a HashMap here. Read returns an integer to indicate whether a query was successful or not, but it has no effect on the progress of the benchmark. All transactions try to query a record with random keys in the same distribution, so many transactions are aborted with warnings or exceptions because they query non-existing records. The benchmarking process should never be blocked or finished unexpectedly.

- **Insert:** Insert is the function to insert a new data object into the database. For the same reason as with Read, there are no guarantees that keys are unique in the table. To solve this problem, YCSB does not require this function to overwrite old values with a new value. To be consistent, insert transactions should never be cascaded to an update transaction.

- **Update:** Update is the function to update the fields of an existing key in the database. Normally an update transaction succeeds only if the key exists, so overwriting the fields for the key is required.

- **Delete:** Delete is the function that deletes an existing data object in the database.

- **Scan:** Scans a given number of objects from the database from a key in ascending order. The results are stored in a HashMap.

The YCSB interface is extensible for other database systems. Users can also implement some side effects in the DB layer interface to meet some special mechanisms of the database system here to compare the performances between database systems.
3.3 Extensibility

There are two ways to extend an YCSB workload. The easiest way is to derive a new workload by writing a new configuration file with different load parameters for the core set of workloads. The second way is to define a new workload. The DB interface layer is ready for further extension. Originally, three DB interface layers for HBase, Cassandra and MongoDB are provided by YCSB.
Chapter 4

Overview of Benchmarked Database Systems

4.1 Overview of CouchDB

CouchDB is a distributed, fault-tolerant and schema-free NoSQL database system. It was first released in 2006. The latest version now is CouchDB1.0.1, which uses JSON as its data format.

4.1.1 Architecture of CouchDB

CouchDB is generally assembled by five components (Figure 4.1). The Erlang HTTP component provides a web based API for querying. The component MOD_Couch is the center module of CouchDB which handles requests from users, drives the store engine, view

\[\text{source: http://couchdb.apache.org}\]
engine and replicator to evaluate queries and retrieves the final results. The view engine provides functions for views to implement the MapReduce functions. The third-party component SpiderMonkey is used here as a JavaScript engine to carry out all algorithms of MapReduce. The storage engine manages the data storage with CouchDB’s strategies: append-only database, streaming data and multi-version concurrency control (MVCC). The replicator performs replication requests.

Append-Only Database

The core implementation of CouchDB is an incremental B-tree [ALS10]. All files in CouchDB in persistent storage are never overwritten. All writing on the database file appears only at the end. Inserting, deleting, and replacing a document are transformed to appending a new document at the end of a database file. The incremental B-tree is a light version of a B-tree with some special properties:

- The performance of writing is improved. Because writing appears only at the end of a database file, the time cost of writing is reduced by skipping the search for the disk heads.
- The backup of a database can be done more easily. All operations do not overwrite the persistent storage, so the truncated database is not corrupted. In this case, the cost of backup of an open database is relatively low.
- The MVCC in CouchDB is very simple. Once a document is written in the persistent file, it will never change. The reading transactions on valid documents are never blocked. The writing transaction is ever serialized due to the append-only rule. Incomplete writes are solved by doubly written headers.
- The append-only strategy orders documents in time series.

Streaming Storage

CouchDB’s raw storage is a stream of data which contains a header and segments for documents. The raw storage is a double-linked list. The header contains meta-information about the database.

Each segment is a structure with header, footer and the data. The location of the former segment and the length of the data are stored in the header, the location of the next segment is stored in the footer. This way allows CouchDB to store data without knowing its length in advance.

CouchDB stores each modification as a segment in the raw storage. The Insert operation appends a new segment to the raw storage. The Update operation does not change any existing data but append a new version of the data as a segment at the end of the database file to announce the former version is out of date. The Delete operation appends a small segment to tell the database that all former versions should be inaccessible to the user.
Because CouchDB does not overwrite any data in persistent storage, old versions of objects and deleted objects are never erased from the raw storage. The percentage of invalidated documents becomes higher and higher. CouchDB provides a function to compact the database. A new stream is created by copying the old stream without invalid entries. Before compacting, all non-deleted documents are accessible to the user. Every document is uniquely identified with a key and its reversion number. After compacting, only documents with the latest version are left. This function takes a risk of loss of data.

**Doubly-Written Footers**

In the raw storage, each segment’s footer is written doubly when the segment is validated by the committing of a transaction. When aborting the data stream it is stored without any footer. When CouchDB continues to append new segments, it scans the database file backwards to find the last two valid footers as the end of the raw storage. So uncommitted segments are overwritten by new segments. If a transaction is interrupted by some reason, the recovery process finds the last pair of footers as the end of raw storage. If one of two is validated and the other is damaged, the recovery process tries to restore the pair of footers by replacing the damaged one with the correct one to rescue the data. Uncommitted data is lost only if both footers are damaged. The rest of raw data is in a consistent state.

### 4.1.2 Web-Based APIs

CouchDB uses RESTful HTTP [RR07] as its API. It allows users to query the database with the HTTP address via web browsers.

RESTful stands for Representational State Transfer. It was first introduced as an architectural style for building large-scale distributed hypermedia systems [PZL08]. CouchDB uses this method to identify all data objects for queries. All RESTful web services have four properties: 1) Resources are identified through URI. It provides a global address space for queries. Most URI identify the web services address. Each data object has its unique address for logical addressing. Therefore, every data object in CouchDB can be described by a RESTful HTTP address. 2) The simple uniform interface of RESTful HTTP accepts only four operations to manipulate data: PUT, GET, DELETE, and POST. 3) The RESTful HTTP delivers self-descriptive messages. The RESTful HTTP address provides the opportunity to represent meta-data about the queried resources by using parameters. 4) RESTful HTTP establishes a stateful interaction between client and server. The stateful interaction allows information to be gathered and stored on both sides, server and client, for future usage, for example URI, cookies, and hidden-form fields can be embedded in RESTful HTTP.

RESTful HTTP is implemented in most programming languages and operating systems. RESTful HTTP is suitable for building lightweight services. Because queries with RESTful HTTP are based on URI and hyperlinks, the access to a centralized site is not necessary to discover and find web resources. RESTful HTTP has its limitations. The input data with POST and GET operations in some systems is limited to 4KB in size. In practice, only GET and POST are needed (GET for idempotent request and POST to
replace PUT and DELETE operations).

The RESTful HTTP used by CouchDB has four operations (GET, PUT, POST and DELETE) as well. To avoid ambiguity between POST and the other two verbs PUT and DELETE, CouchDB defined the four operations as below:

- **GET**: the request method to retrieve data from the database,
- **PUT (POST)**: the request method to insert new data into the database (updating is an insert operation in CouchDB),
- **DELETE**: the request method to delete data from the database.

### 4.1.3 Document-Oriented Storage Strategy

CouchDB focuses on the correctness of storage logic. The correctness of information in a document depends on users or applications. This data model allows users to store their data in its original form. For example, a movie ticket is a normal document in daily life. It may contain the title of a movie, the date of this ticket, the name of the cinema etc. In a relational database, the name of the movie and the cinema may be IDs which are foreign keys pointing to other tables. So if this ticket is queried, three tables would be joined. CouchDB stores its data in the form of documents. This means this ticket is an independent document in CouchDB. Although the data in documents can be joined, the storage strategy of CouchDB does not consider this situation explicitly.

The document-oriented storage strategy of CouchDB is schema-free. For example, tickets from a counter and from a machine may contain information in different schemas. If we want to store them together in a relational database, we have to integrate data with building a greater table or maintain two tables. In practice, a null value is frequently filled into the greater table for nothing to meet the schema. The schema-free storage strategy enables CouchDB to store documents with flexible fields. If the user thinks it is reasonable, every document can be stored together.

```bash
curl -X PUT http://localhost:5984/test/foo -d "\{"count":"1\}"  
{"ok":true,"id":"foo","rev":"1-1ec3c4ed"}
curl -X PUT http://localhost:5984/test/foo -d "\{"count":"2","_rev":"1-1ec3c4ed\}"  
{"ok":true,"id":"foo","rev":"2-e0bddabc"}
curl -X PUT http://localhost:5984/test/foo  
(\"_id\":\"foo\",\"_rev\":\"2-e0bddabc\",\"count\":\"2\")
(\"_id\":\"foo\",\"_rev\":\"1-1ec3c4ed\",\"count\":\"1\")
```

Figure 4.2: An Example of No-Delete Policy

The document-oriented storage of CouchDB is a semi-structured key-value store. Every document in CouchDB is written in JSON. Each of them contains at least two properties: ID and reversion. CouchDB erases no document in the persistent media, so it is possible that multiple documents have the same ID but hold different revisions. Therefore the pair of ID and reversion identifies documents. There is an example (in Figure 4.2): a new document `foo` is created with `count=1`. Then this document is updated with `count=2`. 
4.1. OVERVIEW OF COUCHDB

Both documents can be retrieved from CouchDB, but the older version must be queried by its reversion. In this example, we can see that CouchDB does not delete old documents until the user calls for a compaction.

The key is always given by the user, but the reversion is generated automatically by the database. The user has no knowledge about the reversion before the document is read once. There is the dilemma that the user needs the reversion and ID to read a document to know the reversion. To avoid this dilemma, CouchDB allows the user to access the latest version of a document only with an ID. The reversion is strictly required to update a document. Consider that two users are updating the same document. Both of them are trying to update the latest version of a document. In this process there are two versions: $V_1$ and $V_2$. $V_1$ is created earlier than $V_2$ and appended to the database in this order. According to the storage strategy, $V_1$ is ignored and $V_2$ is announced as the latest version. This means $V_1$ is lost. If reversion is strictly required by updating, the user is always informed which version is going to be updated. If the given reversion does not match the latest version, this updating would be aborted. This means updates only occur to the latest version. So updating by CouchDB has indeed two steps, read and update, unless the user always has a complete knowledge about the reversion.

4.1.4 Views and MapReduce

RESTful HTTP is used to select documents if the user knows the ID. More complex selects cannot be expressed by RESTful HTTP. CouchDB introduced views to handle this problem. Views are created permanently or temporarily by MapReduce. The user can decide whether the result of a view should be stored permanently for further usage or deleted after querying. If a view is stored, a new B-tree is created to store keys and values. After that, views can be queried as a document with RESTful HTTP. Every time a new document is modified or inserted into the database, the view engine will check whether the new document influences the results of a view and updates the B-tree if necessary. This method is also called incremental MapReducing [Len09].

In a relational database each row in a table has the same set of fields, but CouchDB allows documents to contain different fields. If documents are queried, some documents are excluded from the result if they do not have the queried fields. For example there are the three documents Adam, Alex and Adorf:

```json
{ "id":"Adam"," rev":"1-123456","surname":"Britney","title":"Doctor"}
{ "id":"Alex"," rev":"1-172523","familyname":"Wagner","title":"Ph.D"}
{ "id":"Adorf"," rev":"1-134723","surname":"Mueller","title":"Diplom"}
```

To evaluate the query "SELECT surname FROM author", the view engine scans all documents in the database. If a document contains the field "surname", it is emitted and inserted into an indexed list. After all documents are read, CouchDB retrieves the list. This function `emit()` plays the role of the Map function in MapReduce. Each record in the result list is a document with a key and a value. The `emit()` function normally executes a range query. It may take a long time to scan the whole database. Because CouchDB has no management on fields, operation only on the columns is impossible here to reduce the querying time. To avoid double work of `emit()` the result can be saved in CouchDB.
for re-use. Because CouchDB is an append-only database, the indexed list of \texttt{emit()} is
append-only, too. CouchDB only scans for a view once. After that it maintains the index
and list.

To query "SELECT surname FROM auther WHERE title=diplom", the Map function
is needed. The equivalent view is:

\begin{verbatim}
function(doc){
    if(doc.surname&doc.title="diplom") emit(doc._id, doc.surname)
}
\end{verbatim}

If the result for the example above existed in the database, \texttt{emit()} scans only the result
of the last view and emits documents with "title=diplom" one by one and creates new
documents with the two fields \texttt{doc._id} and \texttt{doc.surname}. CouchDB has no management on
the content of a document and every field of a document can be set as a key in the \texttt{emit()} function. So keys can be duplicated or null-valued in a view result.

CouchDB’s Reduce function aggregates the list emitted by the Map function or database
to a single value. For example the following code shows an example for counting the num-
ber of documents which have the field \texttt{surname}:

\begin{verbatim}
Map : function(doc){
    if(doc.surname) emit(doc._id, 1)
}
Reduce: function(key, values, rereduce){
    return sum(values)
}
\end{verbatim}

The Map function emits a pair of \texttt{doc._id} and a constant "1", and then the pair is
stored. The Reduce function takes the resulting list from Map and adds up all numbers.
For example, we change the Map function in this example to \texttt{emit(doc._id, [1,1])}. Then
the values emitted by the Map function are arrays, not literals. The \texttt{rereduce} parameter in
the Reduce function indicates that the database should reduce recursively until the result
cannot be reduced anymore. This means in the first run of Reduce, all arrays are added
up to get a new array as a result and the result can be re-reduced by adding up all literals.

The Reduce function uses the same strategy as the Map function. Each time when a
document is inserted, updated or deleted, a document is appended to the database, and
only those modified documents are reprocessed by the view engine.

\subsection{4.1.5 Clustering with CouchDB Lounge}

CouchDB provides a replicator for clustering. It uses RESTful HTTP for both queries
and views. This architecture provides opportunities to build a clustering framework by
proxies.

Documents and views are identified by their URIs. To build a centralized distributed
database system, a URI to the centralized server can be redirected from URIs to shards.
This idea is realized by a third-party software called CouchDB Lounge [Len09][ALS10]
which provides a proxy-based partitioning and clustering framework for CouchDB. CouchDB
Lounge contains two parts:

\begin{itemize}
    \item Dumbproxy is a proxy based on nginx. It is a lightweight HTTP server which handles
         RESTful HTTP requests by redirecting the requested address to an address on a
4.1. OVERVIEW OF COUCHDB

- Smartproxy is a proxy based on twisted which is an event-driven networking engine to handle view requests. The emit() function is sent to each shard via HTTP proxy, and then the results from the shards are merged together.

CouchDB Lounge allows users to build a cluster with an arbitrary number of shards on an arbitrary number of nodes. Nodes are identified by IP addresses and ports. Each node holds a shard and each shard can be replicated on multiple nodes. A shard map is represented by a list of arrays. Each shard is represented in an array. The first number in the array is the ID of the node which holds the primary shard. The following numbers in the array are the IDs of the nodes to which the primary shard should be replicated. The organization of CouchDB clustering is stored in a JSON object. An example is shown in Figure 4.3 where two nodes are used for clustering and each node is the replication site for the other.

```json
{
    "shard_map": [[0, 1], [1, 0]],
    "nodes": [ ["192.168.1.1", 5984], ["192.168.1.2", 5984] ]
}
```

Figure 4.3: An Example of Clustering with CouchDB Lounge

CouchDB Lounge is transparent to the user. The user accesses the database via a proxy. CouchDB uses a hash function to partition the keys of the documents. The proxy gets an integer from the hash function and chooses a node by a modulo function. The number of nodes is fixed for a cluster. If a host machine is out of service for some reason, the shard map and the number of nodes will not change. If the primary shard fails to be accessed, modifications are carried out on replicated shards. The redirection is stable in this situation because the result of the modulo function does not change. CouchDB Lounge has some disadvantages, for example, MapReduce is slower with CouchDB Lounge because the results from shards must be merged at some nodes again. Range queries are slower, too. CouchDB uses B-trees as an index for keys which supports range queries, but CouchDB Lounge’s hash function distributes keys to different nodes, so the whole B-tree is split into multiple B-trees. The proxy has to merge the results from all shards.
4.2 Overview of Voldemort

Project Voldemort was launched by LinkedIn, a business-oriented social networking site. The first version of Project Voldemort was released in 2009. The most relevant version of Voldemort (release 0.7) released in January 2010 supported rebalancing and entered the area of dynamic clustering. Now the latest version of Voldemort is release 0.81.

4.2.1 System Architecture

There are six layers in Voldemort, and each layer implements an interface for reading, putting and deleting.

- **Client API** performs TCP/IP network communications
- **Conflict Resolution** detects conflicts and reconciles them with multi-version and vector clock,
- **Serialization** translates data objects between the assigned data formats and internal data structures,
- **Routing&Read Repair** is responsible for taking an operation to all storage replicas or the preferred number of nodes or at least the required number of nodes,
- **Failover** (Hinted handoff) handles a situation where a node shuts down or a new node takes part in the service,
- **Storage Engine** is implemented to realize a persistent storage for data. Two persistent storage are supported, namely BerkeleyDB [OBS99] and MySQL [DuB02].

\(^3\)source: http://project-voldemort.com
4.2. OVERVIEW OF VOLDEMORT

Index Management

Voldemort employs a very simple API and all queries have to use the index to accelerate the processing. Each key is transformed into an integer by a hash function and then partitioned. Voldemort manages an index, which is a sorted list, for each partition. Values are stored in a sequential file. Voldemort finds the key for every query in the index to get the location of its corresponding values in the sequential file.

The index of Voldemort has some advantages. The index management is simple to be implemented. The hashing function guarantees that keys are randomly partitioned and the size of each partition is small. The smaller the size, the faster a key can be found. There are also disadvantages. Range queries are not supported by Voldemort. Because keys are hashed and partitioned, the cost for range queries is very high, so Voldemort dropped range queries. In each partition, keys are stored by their hash values and stored in a sorted list which can be stored on multiple pages. Searching algorithms like binary search require that all pages are loaded into the main memory.

Versioning

Voldemort is a multi-version database. It holds a limited number of versions for each document. Different versions of a document are stored in different directories. If Voldemort holds 10 versions, there are 10 directories for data files. Read operations always find the latest version. Write operations try to find the key. If the key does not exist, a new object is inserted into the database, otherwise it finds the right directory to store a new object or overwrite the existing version. The delete operation does not delete the object in the database but writes a new object with empty value to inform the database that this object was deleted. Voldemort performs the MVCC model to manage versions.

Voldemort and Amazon’s Dynamo

One of the most important projects that influenced Voldemort is Dynamo [DHJ+07], Amazon’s highly available distributed key-value storage system. Some ideas of Dynamo were taken to build Voldemort for certain high-scalability storage problems.

The marketing of Amazon requires its cloud serving system to be always available for its clients even if some machines, computing centers or disks are unavailable. The system should allow its clients to access the database at any time.

The CAP theorem proved that the two most important properties chosen by Amazon, availability and partition-tolerance, cannot stand together with consistency. To solve the gap between Amazon’s demand and the service offered by most RDBMS, the new project Dynamo was launched by Amazon. In practice, system architects found that no complex queries are used in most cases of Amazon’s web service, and most queries access the data store directly by primary key. The relational database systems were thought to be too “heavy” for these queries. The lightweight database system Dynamo was designed and developed for this reason. Project Voldemort follows most parts of the ideas of Amazon’s Dynamo.
4.2.2 Simple Queries of Voldemort

Voldemort follows the key-value storage like other typical NoSQL databases. Voldemort has a simple set of functions to access the database.

- `put(key, value)`, e.g. `put("Harry", value-structure)`
- `get(key)`, e.g. `get("Harry")`
- `delete(key)`, e.g. `delete("Harry")`

Every query of Voldemort asks the user to provide a key. Voldemort never queries an object with fields which are not indexed. For example, a query could be: `get("Harry")`. The first thing to do is accessing an index to find the key "Harry" and the corresponding location of its values in the data file. After that, the database reads the value and returns it. The query time is the time to access the index plus the time to read the data file plus the time for preparing to the response. Each query handles only one key, so no scanning happens.

The ability of composing complex queries is limited. Foreign key constraints are not supported by Voldemort because it may cause the database to read multiple data objects.

4.2.3 Data Model of Voldemort

Voldemort uses a similar data model as CouchDB: the structured document-oriented data model. In CouchDB the structure of a document is fixed to a JSON file, however, Voldemort provides more opportunities. The string in the Put function has no constraints. The user can put anything as a value, such as a video, an audio or java object. How to use the values is totally left to the user applications.

Voldemort allows two types of data formats. The value can be an array of binaries. The content of value is free as long as it is serializable. In the other type of data formats, the user is allowed to configure a schema. In this case a serializer is needed to transform a data object into serialized bytes and rebuild the object from serialized bytes according to the schema. The schema must be defined before the database is running. There are seven data formats supported.

- **JSON**: The difference between Voldemort and CouchDB is that CouchDB saves ID, reversion and content in the same document, Voldemort on the other hand uses JSON to handle the format of values. Keys and values are saved separately.

- **String**: In this model anything is taken as a string.

- **Java serializer**: The Java serializer is a serializer to flatten objects to bytes. The Java serializer is useful to store java objects in Voldemort. The size of an object from Java serializer can be large, so it is more reasonable to store data in other formats for re-use.
4.2. OVERVIEW OF VOLDEMORT

- **Protobuf**: Protobuf stands for Protocol Buffer [Inc11] for short. It is a serialized flat data format from Google. It is an excellent language-neutral, platform-neutral data format for communication and storage.

- **Trift**: Trift [SAK07] is a kind of data format designed by Facebook for scalable cross programming language services development. There are software engines for the translation between many programming languages, such as JAVA, C++, Ruby, Python, and so on.

- **avro-generic/avro-specific/avro-reactive**: Avro [Avr11] is a data serialization system which uses JSON as the schema definition to realize its rich structures.

- **Identity**: This data format disables all serializers by Voldemort. The value is faithfully kept to be an array of bytes. It is usually used to save the in-memory data.

The data format must be defined in a `stores.xml` file before the database system is running. Although Voldemort allows the user to define the data format and the structure of documents, Voldemort does not manage the content of documents. It only provides the methods to transform the objects to documents for storage. There are no requirements for fixed schema.

4.2.4 Clustering and Replication

The Clustering and Replication function of Voldemort is very similar to Dynamo. Voldemort is designed for high scalability and availability, so no node in a cluster stores the complete data. Distributing data among nodes makes chunks in every node smaller. Smaller chunks have smaller indexes and gain a higher probability that all pages of an index are loaded into the page cache, especially for frequently queried chunks. The cost of querying a document is dominated by the seek time, so a lower seek time due to small chunks reduces the processing time of a query. The availability is remarkably higher because the system gains the opportunity to re-distribute its data among available chunks.

Another way to raise availability is replication. If data is stored only once on a single machine, once this machine is unavailable, the data is unavailable to the client. Voldemort stores documents in multiple chunks on multiple machines, so the system has a chance that at least one copy of the data survives. If the data on an unavailable machine has copies on other serving machine, there is no data lost in the service.

It is assumed that $N$ is the replication factor, so each document will be stored on $N$ nodes. Theoretically all transactions should hit $N$ copies in the system. If some chunks are unavailable, some transactions may be aborted because they wait too long for the responses from unavailable nodes. If the accessed number of chunks is too high, the system has less availability. If this number is too low, some modifications would not be reflected in a query unless all copies hold the same data, this means writing should be carried out on all copies.

Voldemort allows the user to set the replication factor, the required number and the preferred number of chunks responsible for each query. The required number of copies
determines the minimal number of chunks to be looked up for a query. If this number cannot be met in a time interval, the transaction should be aborted because the quality of a data is considered to be uncertain. The system can commit a transaction immediately if the preferred number of copies is reached.

The settings of these three parameters determine the behavior of the system. In an ideal system, reading can be done on one arbitrary copy and writing on \( N \) (replication-factor) copies. By setting the required number and preferred number of copies to be accessed, the system should read on at least \( R \) copies (required number of read) and write on \( W \) (required number of write) copies. If \( W=0 \), the system has no right to write any copies, in other words, this is a read-only database. If \( R + W > N \), at least one read operation reads the latest versions of a data object. If \( W \) is too low, read and write operations have a great chance that no latest version of a data object is involved.

To solve the problem that the functional specification cannot be simply partitioned into small sub-specifications, Voldemort employs a technique called consistent hashing \[KLL^+97\]. The idea of consistent hashing is that it maintains a hash table which maps keys to slots consistently even if the number of slots changes. Voldemort employs the hash ring, a typical consistent hashing method. A list of slots is chosen randomly and organized in a ring. Each slot appears multiple times in the ring. Keys are mapped to the integer \( K \) by a hash function. The integer \( K \) refers to the \( K \)-th slot on the circle. It assumes that Voldemort has the replication factor \( N \). The hash ring algorithm counts the slots on the circle clockwise until it has found \( N \) unduplicated slots. These \( N \) slots are chosen for the key.

For example, Figure 4.5 shows a hash ring. It is assumed that the key \( k \) has the hash value \( K \). The \( K \)-th slot on the circle is D and the replication factor is three. We count the slots on the circle clockwise. We get DCDBA..., the three non-duplicated slots are DCB (the second D is duplicated, so we ignore it). The final hash value for the key is DCB. Each slot of Voldemort’s hash ring represents a node. In this example three nodes are chosen for the key. Consider that an additional node takes part in the cluster. A new hash ring is created, but for the key \( K \) the \( K \)-th slot and its three following non-duplicated slots may change. This situation makes the system shift some data between nodes to cope with the new hash ring configuration. In expectation, about \( \frac{1}{1+S} \) percent of data has to be shifted, where \( S \) is the number of nodes in the system \[DHJ^+07\].

### 4.2.5 Multi-Version Control of Voldemort

Voldemort uses multi-version concurrency control to manage concurrent accesses to the database. Writing simultaneously across multiple servers is problematic to consistency. For example, there is a document \( D = \{ "\text{name}": "adam", "\text{count}" = "1" \} \) stored on node A and replicated to node B (the replication factor is 2). We set both the required and the preferred number of chunks for reading and writing to 1. Two clients C1 and C2 update the document on different nodes, for example, C1 changes the name to Adam Smith on node A, C2 changes the count to 2 on node B. Then there are two versions referring to the same data object D. Due to the system setting, read and write can be carried out on
one single node. If C1 and C2 update the document on the same node, the conflict will be detected in advance. If they do not, two modifications of one document can produce two conflicting versions. The conflict will be detected sooner or later. To solve this situation Voldemort uses a vector clock.

A vector clock [DHJ+07] is a list of node number and counter pairs. To capture the origin of a conflict between different versions, every evaluation pass of a version is stored in the vector clock. For example, the vector clock of document D={"name":"adam", "count"="1"} is {(A,1)}. It is assumed that document D is written on the node A and replicated on the node B. When a client C1 modifies D to D1={"name":"Adam Smith", "count"="1"} on the node B, the version D1 holds the vector clock {(A,1),(B,2)}. Analogously, when a client C2 modifies D to D2={"name":"Adam", "count"="2"} on the node A, the version D2 holds the vector clock {(A,2)}. Because this version is derived from {(A,1)}, the new vector clock is overwritten to shorten the version list. To detect the conflict of versions, there are two rules:

• If record S is the ancestor of record T, with S and T being the same object, the counter of the i-th pair from the right of S’ vector clock is smaller or equal to the i-th pair from the right of T’s vector-clock. For every pair in S, S is the ancestor of T then.

• If S is neither T’s ancestor nor T is S’ ancestor, then S and T are versions with conflict.

In our case, D is the ancestor of both D1 and D2, but neither is D1 the ancestor of D2 nor is D2 D1’s ancestor, so the conflict between D1 and D2 can be detected. The vector clock {(A,2)} is conflicting with {(A,1), (B,2)}.

When the database has detected a conflict such as D1 and D2, one node is chosen to reconcile it. A new version is created with a vector clock which succeeds all conflicting versions. For example, in our case the new version D3 could be {"name":"Adam Smith", "count"="2"} with vector clock {(A,2), (B,2), (A,3)}. If this solution cannot satisfy clients, the evaluation of D3 is traceable, and it can be corrected manually. This way the conflicting two versions are marked for further version reconciling. The old versions D1 and D2 are not overwritten by Voldemort, so this reconciling causes no data loss.
4.3 Overview of MemcacheDB

MemcacheDB [Chu08] is a blob-oriented storage system. The latest version was released by SINA.com in 2008. Although MemcacheDB is a database system which mostly sacrifices concurrency control, the performance of MemcacheDB is remarkable in high-speed read and write.

4.3.1 Memcached

Memcached is the ancestor of MemcacheDB. Memcached was developed by Brad Fitzpatrick for LiveJournal in 2003, and now more web companies use it to build their high-performance, distributed serving systems, such as Flickr, Wikipedia, Youtube, and so on.

Memcached is a main memory database. To cope with the limited main memory capacity, Memcached manages its memory with the LRU (Least Recently Used) algorithm in which the last recently used data items are deleted to release space for new data items. Every time a page or a data item is deleted it causes a data loss, so Memcached is also thought to be an object caching system. In a cluster, Memcached Servers share their memory with each other, they build a combined cache logically and each node can use the whole cache.

Although Memcached is called a distributed memory object caching system, it provides no distributed algorithms. All algorithms are done by client applications. To build a distributed Memcached System, a hash function is used to compute an integer for each key. With a modulo function the integer is mapped to the node to which the key belongs. The same way, the client application can manage itself to get the corresponding value from the correct server. This algorithm is simple but not optimal for adding and removing a node in the node map. If a server is shut down, the performance of modulo changes violently. For Memcached, the algorithm is not undermined by this situation because all data in Memcached have an expiration time. For more security, better algorithms such as consistent hashing can be applied here to improve the performance. This character is inherited by MemcacheDB, too.

Memcached has the disadvantage that all data objects must have an expiration time. When a data object has expired, the data is not valid anymore. For this reason, Memcached is used to build a serving system to achieve high performance, but at the same time other databases are used to store objects for persistent storage. MemcacheDB is a solution for this problem: high performance and persistent storage.

4.3.2 System Architecture

MemcacheDB (the architecture is shown in Figure 4.6) has three major components: libevent, master thread and threadspool. Libevent is a third-party software which provides the API for MemcacheDB. The master thread manages the connection threads. Each connection thread holds an event handler which handles queries. If the connection thread is not assigned to a task, the event handler waits for the next call. Inside of the event han-
dler, commands are executed according to the user input. Each connection thread holds a connection to BerkeleyDB which provides the persistent storage for MemcacheDB. The threadspool is a container of connection threads. It allocates and releases threads dynamically. If there is not enough memory for new threads, the new request will be denied.

Figure 4.6: Architecture of MemcacheDB

4.3.3 API of MemcacheDB

MemcacheDB inherited most characters from Memcached. Almost all clients for Memcached are compatible with MemcacheDB. The communication with MemcacheDB depends on simple strings. Some commands inherited from Memcached can also be used.

The commands are:

• **GET:** The Get command reads the value by its key. Every time the connection thread tries to read the value, it connects to BerkeleyDB. If there is no such key in BerkeleyDB, the connect thread tries to read the record from the network. The whole memory and persistent disk management is left to BerkeleyDB. Each Get command accesses the database only once if succeeded, otherwise, it will try to read another node in the network one by one.

• **UPDATE:** The Add, Set, Replace, Prepend and Append commands are handled as an update in MemcacheDB. The user’s data is first converted to internal data structures. The Add command tries to get the key from BerkeleyDB to check whether this key exists or not. If there is no such key in BerkeleyDB, the thread is granted to insert a new record into the database. The Set command inserts the key value directly into the database without any checking. If the item exists, it is overwritten. The Replace command reads the BerkeleyDB and changes the value to the new one if this key exists. The Prepend and Append commands read the database and add the value to the existing value if the key exists.

• **DELETE:** The Delete command deletes the key in BerkeleyDB.
After the execution of commands a response is returned to the client and the connection thread is cleaned and sent back to the threadspool.

4.3.4 Replication and Consistency

MemcacheDB provides no replication function. Replication is performed by BerkeleyDB. The master of MemcacheDB drives a BerkeleyDB as a read/write database. Each slave of MemcacheDB drives a read-only BerkeleyDB. The replication between master and slaves is similar to Voldemort’s. The replication behavior is configured by setting the preferred read/write and required read/write. The replication factor is automatically set when master and slaves run. In a cluster of MemcacheDB there are multiple masters, each of them holds a partition of data. The partitioning of data depends on the user applications, MemcacheDB provides no mechanism for partition.

Concurrency control is simple in a MemcacheDB cluster. Write transactions happen only on the master. The data on slaves is synchronized periodically. MemcacheDB has no mechanisms to deal with conflicts. The data on the master has only one version. The user application can store data objects on multiple masters. It is the user applications’ responsibility to handle conflicts.
Chapter 5

Benchmark with YCSB

In the chapter above, three NoSQL databases were introduced. In this chapter, we present the results of benchmarking these three systems.

YCSB DB layer interface of Cassandra, Hbase and MongoDB is offered by YCSB, but they are not involved in this benchmarking because the YCSB DB layer interface for Cassandra has some compiling problems and available versions of HBase and MongoDB are not supported by YCSB anymore.

All of CouchDB, Voldemort and MemcacheDB use key-value storage to manage their data. Each database system has different data models, APIs, indexes and clustering frameworks. In this chapter, we discuss how to design the YCSB DB layer interface for them to benchmark them fairly. The size of each data object sent to the database is different in all three systems. In this experiment we measure the capacity of databases by objects and not by bytes. We chose the most popular APIs for them in this benchmark and tried to build equivalent clusters for all of them.

5.1 The DB Layer Interface of CouchDB

The RESTful HTTP API of CouchDB was introduced in the chapter above. In this benchmark, the RESTful HTTP API is not optimal as a YCSB DB interface. We chose the java-oriented API "jcouchdb". All data used by YCSB is inserted and stored in a particular table, but CouchDB has no table concept. So we decided to store all documents equally in the test database and ignore the table required by YCSB. YCSB queries the test data in only one table, so this adjustment does not influence the result. Voldemort and MemcacheDB have the same situation. To be fair to all database systems, we stored all data objects directly in the test database.

The data given or required by YCSB was a data object stored in a HashMap, but CouchDB accepts documents in JSON. We transformed the data format from HashMaps and JSON documents. To simplify the transformation, all modifications of data were done in the format HashMap. Five functions for YCSB DB interface were implemented:

- **Insert**: The equivalent function to insert in CouchDB’s API is the Put function. Put has two variants, one with reversion and one without. The one with reversion
inserts a document into the database and overwrites it if it already exists. The one without reversion inserts a document into the database if the key does not exist. We chose the Put function without reversion. In this case, if the key exists in the database, the transaction is aborted by version conflict.

- **Read**: The Read function is implemented with the Get function in CouchDB’s API. Although CouchDB allows the user to read older versions, we only read the latest version. Therefore, the Read function can be called without reversion.

- **Scan**: The Scan function is implemented by the query function, which allows the user to give the corresponding RESTful HTTP address for querying the database. The query is carried out by CouchDB’s view engine, which provides range queries by traversing CouchDB’s B-trees.

- **Update**: The Update function is implemented in two steps. It calls the Put function to insert objects, but this time reversion is required, so we have to read the reversion first and then insert.

- **Delete**: The Delete function from CouchDB requires the key and the reversion number to ensure that the latest version of an object is deleted. So the Delete function is executed in two steps. The first step is to query the database to find the reversion of the latest versions. The second step is to perform the deletion. If there are other transactions modifying the object under the key, the delete function is aborted.

### 5.2 The DB Layer Interface of Voldemort

Voldemort is based on JAVA language and it provides a JAVA API. The string data format does not ask a serializer to parse the documents. To avoid unexpected tasks in the database system, we chose the string as a semi-structured data object in this experiment. The string data format has the further advantage that the data object can be represented by JSON strings, which allows YCSB to retrieve the data queried form database. There are five operations:

- **Delete**: The Delete function is implemented by using the Delete function in Voldemort. Voldemort allows the user to delete documents without vector clock, so this function is implemented for Voldemort in one step.

- **Insert**: The Insert function is similar to the Put function in Voldemort’s API. The Put function without vector clock overwrites the document in the database. So we create a new version which has no ancestors. If the key exists in the database, inserting a new version conducts to a version conflict, so it will be aborted.

- **Read**: The Read function calls the Get function to get back data objects by their keys.
5.3. THE DB LAYER INTERFACE FOR MEMCACHEDDB

- **Scan**: Range queries are not supported by Voldemort. To complete this task, we tried a different approach. First, all data objects were gathered from the database. The keys smaller than the start key were ignored and the keys greater or equal to the start key were stored in a set on the client machine. Items in this set were sorted to get the result for a range query. This method is problematic, too, because the data set of the whole database can override the memory of the client machine. Getting the whole data set is not reasonable, so the Scan function is not recommended.

- **Update**: Voldemort provides only Put, Read and Delete functions. The Update function is derived from the Put function. It has three steps. The first step is to read a versioned value, which contains the vector clock and the key-value pair, from the database. In the second step, the vector clock is updated on the query site and the value is updated. Finally the updated document is sent back to the database. If there is a version conflict, it will be aborted. So each update function has to communicate with the database twice.

5.3 The DB Layer Interface for MemcacheDB

We used the Memcached API to build the DB layer interface. Because MemcacheDB supports the memory structure, the HashMap object from YCSB can be used directly in the interface. So we decided to use this memory structure as the data format.

- **Delete**: this function is implemented with the Delete command to delete a given key.

- **Insert**: The Insert function is implemented with the Put command to store the key and HashMap from YCSB.

- **Read**: The Read function reads a HashMap object from the database by the Get command.

- **Scan**: MemcacheDB does not manage any data itself. All data is stored in BerkeleyDB. It provides no range queries. So the Scan function is not implemented for MemcacheDB.

- **Update**: The Update function calls the Replace command, which updates an object only if the object exists. The key and the new value are sent directly to the MemcacheDB.

5.4 Benchmark Environment

For all workloads, we used four virtual machines (VM). Each VM has a 64-bit 3.0GB Intel Xeon Cpu, 1 GB RAM, 9GB disk space. YCSB is configured with 20 threads to run its operations to execute operations. The YCSB DB layer interfaces were built without any side effects, but it might be unfair since MemcacheDB asks user applications to partition
their own data. Most experiments had a connection limitation to study the performances of databases under different traffic situations. If the target number of operations per second is reached, YCSB threads have to wait. To study the unlimited performances of database systems, one experiment for each workload was unlimited. Before each experiment, about 100,000 data objects were inserted into the databases in the required distribution. Because of the character of YCSB’s algorithm of key generation, the number of keys stored in the database is actually less than 100,000.

CouchDB manages the main memory itself. The cache size of Voldemort and MemcacheDB is limited to 512M, the remaining main memory is left for the operating system and the database itself. The JVM (Java Virtual Machine) of Voldemort is configured with 384M and the BerkeleyDB is configured with 128M. MemcacheDB is configured with 512M for the BerkeleyDB’s cache. CouchDB gets more memory than the other two databases because it was allowed to use all available resources. Voldemort holds a great number of files, the recommended cache size is 1GB, but in this experiment only 512M was offered. The performance of Voldemort may be better.

<table>
<thead>
<tr>
<th>Workloads</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Updateheavy</td>
<td>Readheavy</td>
<td>Readonly</td>
<td>Readheavy</td>
<td>Scan</td>
<td>ReadModifyWrite</td>
</tr>
<tr>
<td>Read</td>
<td>50%</td>
<td>95%</td>
<td>100%</td>
<td>95%</td>
<td>--</td>
<td>50%</td>
</tr>
<tr>
<td>Update</td>
<td>50%</td>
<td>5%</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Insert</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>5%</td>
<td>5%</td>
<td>--</td>
</tr>
<tr>
<td>Scan</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>95%</td>
<td>--</td>
</tr>
<tr>
<td>ReadModifyWrite</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>50%</td>
</tr>
<tr>
<td>distribution</td>
<td>Zipfian</td>
<td>Zipfian</td>
<td>Zipfian</td>
<td>latest</td>
<td>uniform</td>
<td>Zipfian</td>
</tr>
</tbody>
</table>

Table 5.1: Properties of Workloads

In this experiment, we used the six original workloads from YCSB to benchmark the database systems. Workload D uses latest distribution and Workload E uses uniform distribution. Zipfian distribution is used for the remaining workloads. The properties of workloads are summarized in Table 5.1. Workload A is an update-heavy workload, in which 50 percent of transactions executed are update transactions and the rest are read transactions. Both Workload B and D are read-heavy workloads. 5 percent of transactions in Workload B are update transactions, but Workload D uses insert transactions instead of update transactions. Workload C is a read-only workload. 95 percent of transactions executed by Workload E are scan transactions, and another 5 percent are insert transactions. Workload F is a read-modify-write workload, in which 50 percent of the transactions read objects from the database, modify their content and then update the objects in the database. Another half transactions are read transactions in Workload F. For each workload, we measured the latency of transactions according to the throughput (operations per second) to show the performance of each database system in different environments,
then we discuss the result of this benchmark.

5.5 Results of Benchmark

In the update-heavy experiment (illustrated in Figure 5.1), the update and read latency of CouchDB increases drastically at a throughput of 150 ops/sec. The latencies of Voldemort and MemcacheDB are more stable. The throughput of MemcacheDB reaches 9225 operations per second in the unlimited experiment. CouchDB is an append-only database system, so it allows only one thread to append new documents to the data file. This limited the performance of CouchDB. When the write requests exceed the throughput of disk, they have to wait in a queue. This can be the most important cause for CouchDB’s drastically raised update latency.

![Figure 5.1: Workload A: Update-Heavy](image)

As Figure 5.2 shows, in the read-heavy experiment Workload B, the update and read latency of CouchDB begin to rise when the throughput reaches 250 ops/sec. In Voldemort, the turning point of update latency is about 500 ops/sec, and the read latency increases steadily in the zone above 550 ops/sec. MemcacheDB has the most stable performance in all three database systems. In an update-heavy situation, CouchDB appends many documents to its data file. So the size of the B-tree increases quickly and CouchDB has to pay more IOs to manage its cache and disks. Voldemort keeps only a few versions for a data object, not all. So in an update-heavy situation Voldemort has a better performance than CouchDB because the update transactions do not change the sizes of indexes and data files after several updates. MemcacheDB keeps only one version, so it pays the least to handle the MVCC.

In Figure 5.3 for Workload C we can see that CouchDB’s line has the same shape as before. Voldemort has a similar trend as CouchDB. The read latency of Voldemort begins to increase after it reaches 800 ops/sec. MemcacheDB has the best performance again with 8982.85 ops/sec and 2.04 ms read latency in the unlimited experiment. CouchDB
has a similar result of read latency as Workload A and B. Because the MVCC employed by CouchDB never blocks the read operations, the write operations influence the read operations very little. Voldemort does better than before. The write operations store new documents in different directories, so the caching of hot data objects may change frequently and Voldemort pays more for updating the cache. In the read-only Workload C the hot data objects have a greater opportunity to be loaded in the cache. MemcacheDB may be the most desirable one among the three for its high speed.

In Workload D (results in Figure 5.4), a read-heavy workload with insert transactions, both insert and read latency increased drastically after its throughput reached 350 ops/sec.
The performance of Voldemort is more stable than CouchDB. MemcacheDB has a similar behavior as Voldemort, but the latencies were higher in most limited throughput runs. We can notice that the performance of MemcacheDB is not the best in this workload. In Workload D the insert transactions insert data objects into the database rather than overwriting by update transactions by MemcacheDB. So MemcacheDB has to pay more IOs to manage its cache. CouchDB appends documents for both insert and update transactions, so we find that the lines of insert and update latency in Workload B and D are almost the same. Voldemort creates only one object for each successful insert transaction and there is no need to do more jobs for MVCC. So the update latency is slightly higher than the insert latency by Voldemort.

MemcacheDB and Voldemort fail the experiment with scan transactions in Workload E. Although there are ways to implement scan transaction for Voldemort, it still failed due to a high demand for memory. CouchDB can execute about 5 scan transactions per second in this experiment.

In Workload F (results in Figure 5.5), latencies of CouchDB increase obviously after 100 ops/sec and the turning point of Voldemort is at 300 ops/sec. MemcacheDB had a flat shape again in this experiment. From the design of their DB interface layers, the update transaction of CouchDB and Voldemort are actually a read-modify-write process by reading data back, updating locally and updating at the server. It is quite reasonable that the update latency line and the read-modify-write latency line have similar shapes for both CouchDB and Voldemort in Figure 5.5.

Generally speaking, in the benchmark CouchDB’s latencies for all workloads are higher than Voldemort’s. In unlimited connection experiments, Voldemort also has a higher throughput than CouchDB. MemcacheDB does the best in most workloads. In all experiments, the performance of CouchDB is stable when the throughput is below a certain value. Voldemort’s performance is more stable than CouchDB’s, and it also has lower la-
tencies and higher throughputs than CouchDB. The performance of MemcacheDB, which has the highest throughput in unlimited experiments and the lowest latencies by most workloads, is most remarkable.

The incremental B-tree allows the user to append documents one after another. This design ensures that the insert and update transactions are serializable, but it permits only one thread to write the data file, so other threads have to wait in the queue. If the throughput reaches a certain value, the increase of latencies is inevitable, as most experiments show. Voldemort uses chunks to store its data, so multiple processes are permitted to write multiple chunks at the same time. So we can see that Voldemort has similar lines as CouchDB, but its lines are much flatter. MemcacheDB is different from both CouchDB and Voldemort. It was noticed that MemcacheDB accesses the database for every transaction only once, and it benefits from this. In Workload D, MemcacheDB cannot perform as good as in other workloads. Insert transactions and latest distribution may be the reason for this phenomenon. MemcacheDB, which has only one version, overwrites data objects by updating, but it has to insert new data objects by insert transactions. Compared to the insert latency of Workload D in the load phase, they are similar.

The incremental B-tree is one of the optimal choices for range queries, but the performance of CouchDB is not as good as expected. The CouchDB lounge distributes keys with a hash function into shards. With range queries, we saw all databases were busy getting objects. After all databases stopped, the queried node began to merge data. The distribution strategies of CouchDB lounge decreased the performance of the incremental B-tree of CouchDB. The algorithm of range queries of CouchDB Lounge can be also improved by adding more parallelisms, for example by adding pipelines to merge the data from all databases. However, the results are delivered by RESTful HTTP from shards, which send all documents back together for range queries. So this improvement may be problematic.
Chapter 6

Summary

6.1 Summary of Work

At the beginning of this paper, we introduced the background knowledge about RDBMS and NoSQL. We discussed the advantages of RDBMS and its limitation and other alternative choices for RDBMS. Then we discussed the parallel database system and distributed database system and the two methods fragmentation and replication for distributed database systems. Later we introduced the concept of scalability and the programming model MapReduce. Based on the background knowledge, we started a new chapter about the NoSQL movement and its theories and technologies.

To begin our benchmarking, we looked into a benchmarking tool called YCSB to see how a database system is benchmarked by YCSB and some details about the workloads used by YCSB. Then we described how to design a database interface layer for YCSB. Then we studied the details about the three to be benchmarked database systems, especially their system architecture, APIs, concurrency controls and their clustering frameworks. We extended the DB interface layers of YCSB to these three database systems and we built clusters with four VMs for each database system. Their performances were benchmarked with six workloads which were defined by the YCSB developing team. We prepared 100,000 data objects for each workload, and we executed each workload for all databases with connection limitations to simulate different connection traffic. For each database and workload, we prepared an unlimited run to benchmark the unlimited throughputs of them. Every time we executed the workload more than 1 minute and less than 30 minutes to get the latencies. The results are illustrated in figures and discussed.

6.2 Conclusion

From the results of this benchmark we studied the performance of the three database systems with respect to their system architectures.

CouchDB is figured as an append-only database system with incremental B-trees. From the benchmark result, we can see that all latencies are very stable before the operations per second reach a turning point. The append-only policy makes CouchDB permit only one
thread to handle all updating, deleting and inserting requests. This algorithm determines that the throughput of CouchDB is limited, and the multi-processor and multi-disk cannot cope with CouchDB very well. Although CouchDB Lounge offers a framework to build a cluster with CouchDB, it splits the B-tree into multiple B-trees and slows down the range queries. The proxy used by CouchDB Lounge uses RESTful HTTP as the API, which delivers the results of range queries form notes all at once, so the proxy can merge the results only after all results were gathered from all nodes.

Voldemort has generally a higher performance and is more stable than CouchDB from the benchmark result. To store data objects in multiple chunks allows Voldemort to use multi-processors to write its data files. We can see that the throughput of update and insert transactions are obviously higher than with CouchDB. The clustering framework of Voldemort distributes data to nodes, and the hash function used by Voldemort balanced the data objects among nodes well, but the hash function also makes the range query by Voldemort very expensive and it is not supported by its API.

MemcacheDB has the simplest implementation among the three, but its performance is remarkable. It scarifies consistency most in the three database systems to gain higher performances. We proved that MemcacheDB is excellent in high speed reading and writing. MemcacheDB has limited function in clustering: MemcacheDB asks the user to distribute their data between nodes. We built a cluster with one VM as the primary server and three slaves. So writing and reading transactions are actually performed on the primary server. Although the way of clustering may be unfair for MemcacheDB, the performance is still outstanding. As we said, MemcacheDB has no guarantee for concurrency control, so the user application has to do more jobs on it, otherwise MemcacheDB is not suitable for some applications which have high requirements for consistency.

### 6.3 Outlook

In this benchmarking, we only benchmarked three database systems, we can extend the YCSB DB layer interface to more database systems in the future. The comparison between different database systems can help us to analyze performances of different approaches. We just used a small number of VMs to build a cluster, so in this benchmark we could not compare the performance of speed-up and scale-up of those database systems. If we have more nodes, we can benchmark database systems with different numbers of nodes, and then we can analyze the improvement of additional sources. Furthermore, we can shut down some nodes while the benchmark is running to see how those database systems recover from the situation that some nodes are suddenly unavailable. The benchmark tool YCSB offers us many opportunities to simulate different situations to study the performances of database systems.
Bibliography


