

UNIVERSITÉ LIBRE DE BRUXELLES

MA1 COMPUTER SCIENCE & ENGINEERING

2015-2016

---

## Advance database : Voldemort

---

*Author:*

Benjamin AKBOKA

Nazar FILIPCHUK

*Professor:*

Esteban ZIMANYI

December 28, 2015



# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>A brief presentation of distributed databases</b>	<b>5</b>
2.1	Distributed databases characteristics and objectives . . . . .	5
2.2	Distributed Databases Management Systems . . . . .	8
<b>3</b>	<b>Project Voldemort</b>	<b>9</b>
3.1	Design . . . . .	9
3.1.1	Key-value storage . . . . .	9
3.1.2	System architecture . . . . .	10
3.1.3	Data Format and Queries . . . . .	13
3.1.4	Consistency and Versioning . . . . .	13
3.1.5	Versioning in a distributed system . . . . .	14
3.2	Brewer's theorem . . . . .	14
<b>4</b>	<b>Performances and benchmarking</b>	<b>15</b>
4.1	Performance vs. Scalability . . . . .	15
4.2	Performances . . . . .	15
4.2.1	Build Times . . . . .	15
4.2.2	Read Latency . . . . .	16
4.2.3	Solid State Drive . . . . .	17
4.3	Benchmarks . . . . .	17
4.3.1	Workload impact on latency . . . . .	17
4.3.2	Failover Characteristics . . . . .	18
<b>5</b>	<b>Creating distributed database servers with Voldemort</b>	<b>21</b>
5.1	Installing the source code . . . . .	21
5.2	Server configuration . . . . .	21
5.2.1	Cluster configuration . . . . .	22
5.2.2	Stores configuration . . . . .	23
5.2.3	Each server properties . . . . .	23

<b>6</b>	<b>Using the database from the client side</b>	<b>25</b>
<b>7</b>	<b>Conclusion</b>	<b>27</b>

# List of Figures

2.1	Homogeneous plan for distributed databases . . . . .	6
2.2	Heterogeneous plan for distributed databases . . . . .	6
2.3	Comparison of distributed database design strategies . . . . .	7
2.4	DBMS schema . . . . .	8
3.1	Architecture of the database [7] . . . . .	10
3.2	The physical architecture . . . . .	11
3.3	The hash ring for 3 nodes and 12 partitions [7] . . . . .	12
4.1	The time to complete the build for the random data set [7, p. 10] . . . . .	16
4.2	Single node median read latency taken at 1 minute intervals since the swap[7, p. 10] . . . . .	16
4.3	Single node read latency after warming up the cache[7, p. 10] . . . . .	17
4.4	Client-side median latency with varying data size[7, p. 11] . . . . .	17
4.5	Workload A-E for Cassandra 2.0.2[1, p. 7] . . . . .	18
4.6	Workload A-D and f for Voldemort 1.3.0 [1, p. 8] . . . . .	19
4.7	Maximum throughput for 1M records [1, p. 8] . . . . .	19
4.8	Maximum throughput for 50M records[1, p. 8] . . . . .	19
4.9	Update heavy workload with one node failure at 50% of maximum throughput for Cassandra 2.0.2 [1, p. 11] . . . . .	20
4.10	Update heavy workload with one node failure at maximum throughput for Cassandra 2.0.2 [1, p. 11] . . . . .	20
4.11	Update heavy workload with one node failure at maximum throughput for Voldemort 1.3.0 [1, p. 11] . . . . .	20
6.1	Establishing the connection to both clusters of the database from same client . . . . .	25
6.2	Inputting information into first cluster and retrieving from the other. . . . .	26
6.3	Few basic commands on string data-set. . . . .	26

# Chapter 1

## Introduction

With the globalisation created by the Internet, companies can no longer have a database on a unique machine. With a unique machine managing the database, there are huge drawbacks. Firstly, it is a single point of failure. Meaning that if the data center goes offline for any reason then the application stops working. Another drawback is that the user of the database will suffer delay due to the distance between him and the data center.

In order to avoid those drawbacks, another type of database has been created because it need to be *distributed* on multiple machine. Those new databases have requirement in order to keep each database in sync with the main one, if there is any, or just avoid incoherence between them. So when one database is modified, that modification needs to be propagated into the network.

With the growing utilisation and generation of data, maintaining those distributed database is revealing to be a challenge. In response to this challenge, the popular website LinkedIn created a project who's name is inspired by a villain from Harry Potter : The project Voldemort. It is an open source highly scalable distributed database created to suit their needs. The structure of Voldemort is inspired by the Amazon's Dynamo paper[5]. Their goal was to create a low-latency, high-availability database [6].

In this report, a short introduction of distributed databases will be made. Next will come a detailed description of the inner working of Voldemort. Then some discussion about the performance and some benchmark will be presented. Finally, the backend implementation will be presented for the readers that intend to use this database. As the main user of Voldemort is LinkedIn, most of our references comes from them but this report will try to not be limited by what is done by them.

## Chapter 2

# A brief presentation of distributed databases

### 2.1 Distributed databases characteristics and objectives

Databases have various implementations depending on company's needs, one may have their sieges dispersed over all the country or even abroad. To keep the integrity of overall database system few options are possible: you either centralize all data on main server or distribute the database locally and gather the data in further needs. Here is the point where distributed databases show them-self use-full, according to the definition - **a distributed database** is a single logical database that is spread physically across computers in multiple locations that are connected by a data communications network. Emphasizing that a distributed database is truly a database, not a loose collection of files. The distributed database is still centrally administered as a corporate resource while providing local flexibility and customization. [10, p. 1]

A distributed database requires multiple instances of a database management system or several of them, running at each remote site. The degree to which these different DBMS instances cooperate, or work in partnership, and whether there is a master site that coordinates requests involving data from multiple sites distinguish different types of distributed database environments. [10, p. 2]

Two major groups of distributed databases are **homogeneous** 2.1 and **heterogeneous** 2.2. The first set has data distributed across all nodes and managed by the distributed DBMS, same DBMS is used at each location. All users access the database through one global schema or database definition, the global schema is simply the union of all the local database schema. The heterogeneous plan for for DDBs characteristics have following variations - different DBMSs may be used on each node even though all data is as well distributed across all nodes. Users that require only local access may get it by using local DBMS and schema, but global schema still persists.[10, p.3-4] Common objectives are **location transparency**, meaning that user does not need to know the location of data, and **local autonomy**, where database operates even if few nodes failed. The **synchronous** or **asynchronous** data management are both applicable with DDBs.[10, p.5]

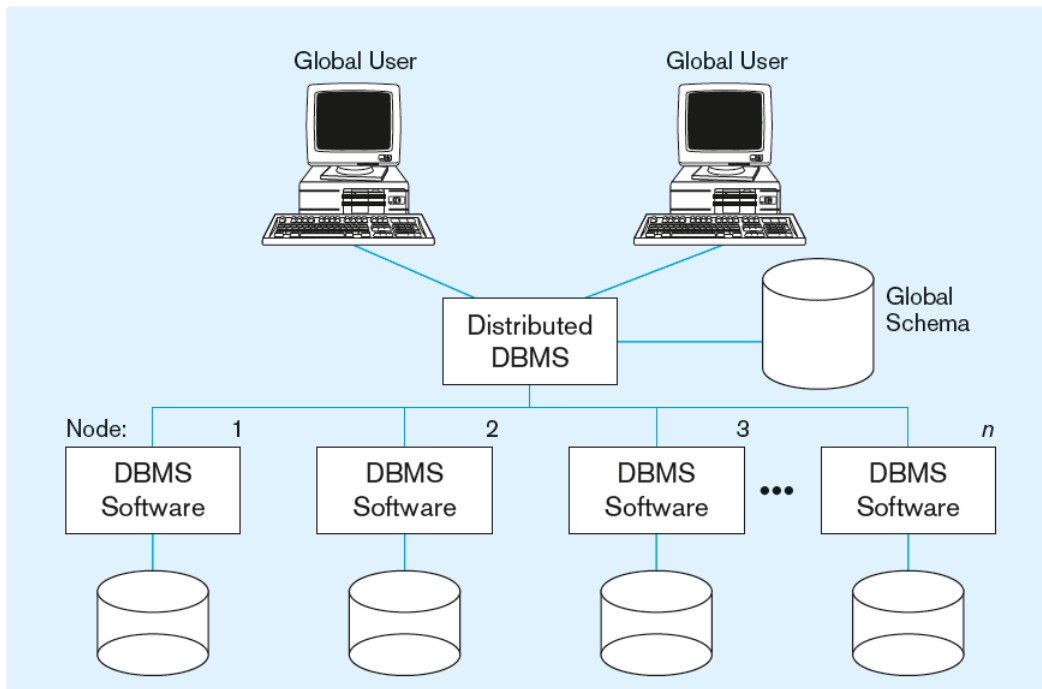


Figure 2.1: Homogeneous plan for distributed databases

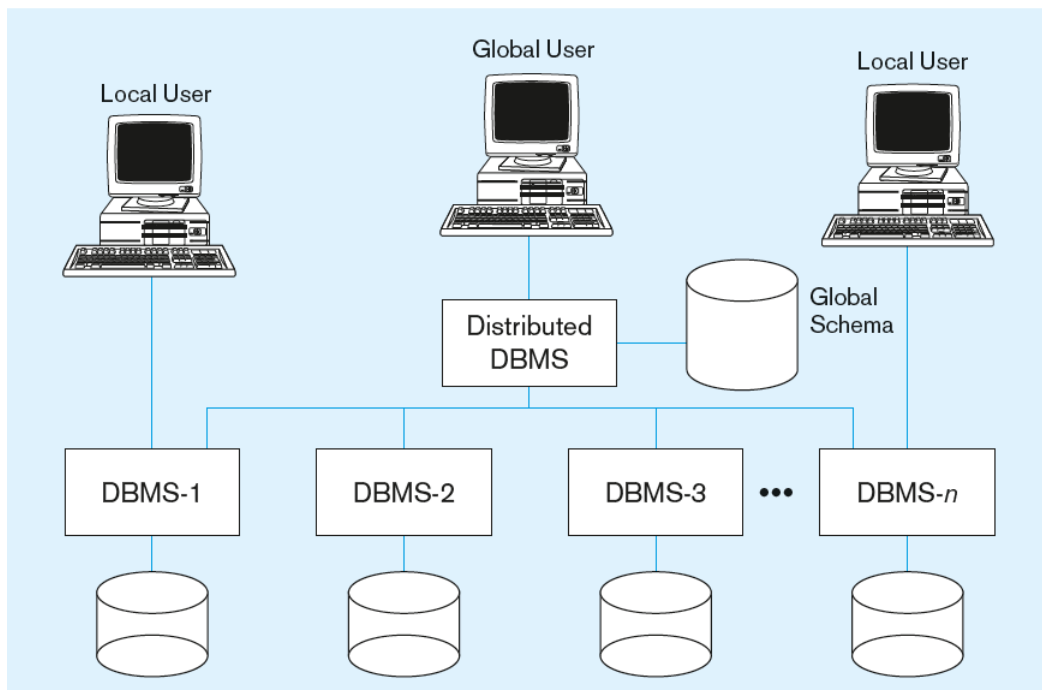


Figure 2.2: Heterogeneous plan for distributed databases

There are four main distribution strategies for relational databases as example but it works as fine for other types:

- **Data replication** - putting copies of the database at each two or more sites, making it closer to the user. There are several types of update synchronization, such as: snapshot replication, near-real-time replication for pushing and pull replications.
- **Horizontal partitioning** - putting some of the rows of a table into a base relation at one site, and other rows are put into a base relation at another site.
- **Vertical partitioning** - some of the columns of a relation are projected into a base relation at one of the sites, and other columns are projected into a base relation at another site. The relations at each of the sites must share a common domain so that the original table can be reconstructed.
- **The hybrid** - is a combination of vertical and horizontal slicing in order to have the most comfortable and optimized partitioning of the database.[10, p.6-13]

All pros and cons for specific application are summarized in the table 2.3

Strategy	Reliability	Expandability	Communications Overhead	Manageability	Data Consistency
Centralized	<b>POOR:</b> Highly dependent on central server	<b>POOR:</b> Limitations are barriers to performance	<b>VERY HIGH:</b> High traffic to one site	<b>VERY GOOD:</b> One monolithic site requires little coordination	<b>EXCELLENT:</b> All users always have the same data
Replicated with snapshots	<b>GOOD:</b> Redundancy and tolerated delays	<b>VERY GOOD:</b> Cost of additional copies may be less than linear	<b>LOW to MEDIUM:</b> Not constant, but periodic snapshots can cause bursts of network traffic	<b>VERY GOOD:</b> Each copy is like every other one	<b>MEDIUM:</b> Fine as long as delays are tolerated by business needs
Synchronized replication	<b>EXCELLENT:</b> Redundancy and minimal delays	<b>VERY GOOD:</b> Cost of additional copies may be low and synchronization work only linear	<b>MEDIUM:</b> Messages are constant, but some delays are tolerated	<b>MEDIUM:</b> Collisions add some complexity to manageability	<b>MEDIUM to VERY GOOD:</b> Close to precise consistency
Integrated partitions	<b>VERY GOOD:</b> Effective use of partitioning and redundancy	<b>VERY GOOD:</b> New nodes get only data they need without changes in overall database design	<b>LOW to MEDIUM:</b> Most queries are local, but queries that require data from multiple sites can cause a temporary load	<b>DIFFICULT:</b> Especially difficult for queries that need data from distributed tables, and updates must be tightly coordinated	<b>VERY POOR:</b> Considerable effort; and inconsistencies not tolerated
Decentralized with independent partitions	<b>GOOD:</b> Depends on only local database availability	<b>GOOD:</b> New sites independent of existing ones	<b>LOW:</b> Little if any need to pass data or queries across the network (if one exists)	<b>VERY GOOD:</b> Easy for each site, until there is a need to share data across sites	<b>LOW:</b> No guarantees of consistency; in fact, pretty sure of inconsistency

Figure 2.3: Comparison of distributed database design strategies



## 2.2 Distributed Databases Management Systems

The existence of distributed databases involves a management system to access and use that distributed data. There are several mandatory criteria the management systems must obey, such as - keeping track of where data are located, determine the location from which to retrieve the requested data, translating request from node with local DBMS to another with the different one, providing data management functions and consistency among copies across remote sites, present a single local database that is distributed. Additional options like being scalable, running different version of applications and DBMS on different nodes, replicate data and stored procedures across the nodes and finally transparently use residual computing power to improve processing performances. The general schema is presented on figure 2.4.[10, p.13-14]

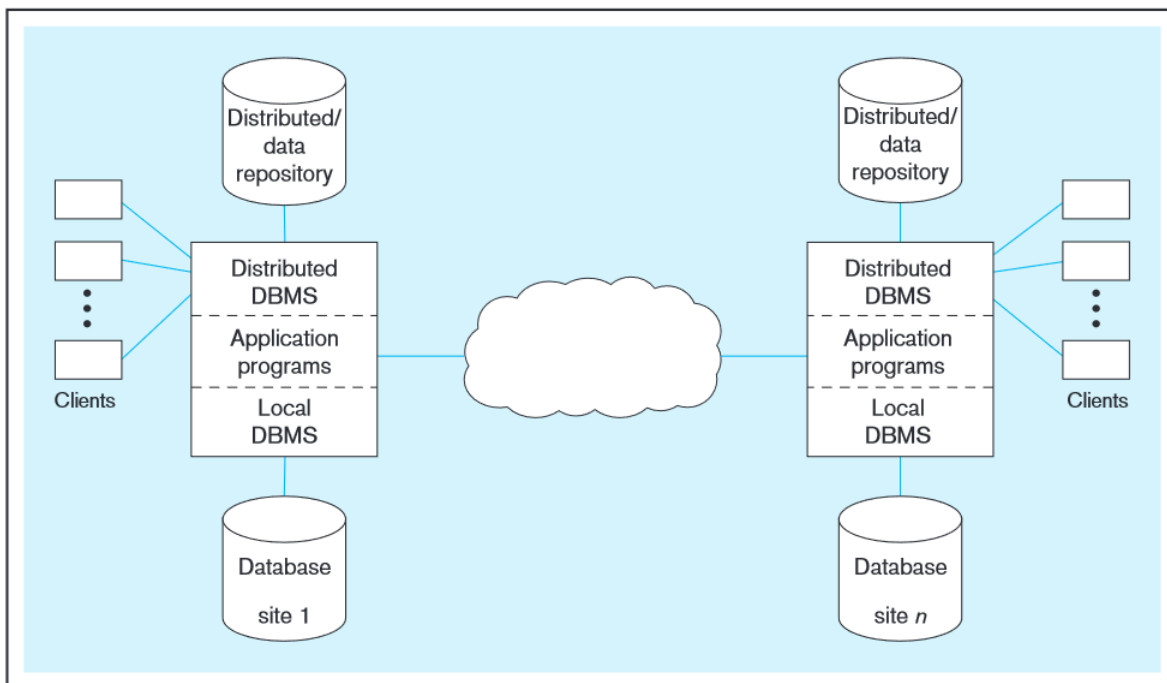


Figure 2.4: DBMS schema

Each node has its local DBMS managing local DB as well as a copy of distributed DBMS with associated data dictionary/ directory. Client's request first is processed by distributed DBMS to know if the request is a local transaction and can be managed by local DBMS or it is a global transaction that needs reference to data from several sites. For global transactions DBMSs exchange messages to coordinate it. [10, p.15]

## Chapter 3

# Project Voldemort

Project Voldemort is an open source database created by the company LinkedIn. This chapter will explain in detail how this database is functioning. The description will be in general detail for people who want to use this database for online purposes..

### 3.1 Design

Voldemort is constituted of cluster that holds multiple nodes who have a unique id. A single machine can hold multiple nodes if needed. All those nodes need to have the same number of *stores* which is basically a table in the database. This is needed as the data will be spread on multiple cluster and thus a store will hold part of the original information. Each store must have a list of configurable parameters [7, p. 3] :

- Replication factor (N) : the number of nodes in which the data will be replicated
- Required Reads (R) : number of parallel reading that Voldemort has to make to execute a get request
- Required writes (W) : number of nodes that respond to a put request
- Key/Value serialization and compression : Voldemort can have different serialization schemas for key and value
- Storage engine type : Voldemort can be combined with multiple read-write storage

Every node in a cluster holds the complete cluster topology and the store definitions

In this section, the underlying technologies of Voldemort will be exposed. Those explanations come mainly from the Voldemort website [13].

#### 3.1.1 Key-value storage

Project Voldemort uses a key-value storage. Key-values store uses a dictionary[3, p. 56] to find information. This allow the client to simply search through the database per key. The api is very

simple[13] :

```
value = store.get(key)
store.put(key, value)
store.delete(key)
```

The intent was to enable high performance and availability. But it has his pros and cons[13].

### Pros

- It is easy to distribute across a cluster
- Clean separation of storage and logic
- No object-relational miss-match
- Only efficient queries are possible, very predictable performance

### Cons

- No complex query filters
- All joins must be done in code
- No foreign key constraints
- No triggers

## 3.1.2 System architecture

### Logical architecture

The system is composed of several layers as seen on fig 3.1 completely independent from each other. Each layer implement the same simple interface and is in charge of only one functionality in the database. This make it very easy to add or remove them on the fly as the needs change. For example, a module can be add to manage partitioning and replication. Another one can manage conflict resolution or can implement a repair mechanism.

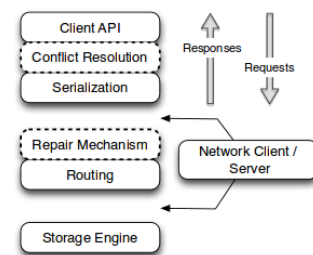


Figure 3.1: Architecture of the database [7]

## Physical architecture

There are three options proposed by the project [13] to implement physically a database using Voldemort. The physical architecture will try and handle the load distribution on each cluster. On the figure 3.2, Those three types of architecture can be seen. On this figure, Load bal is a hardware loadbalancer and partition aware routing in the internal storage routing. Having those steps on the server side does let the choice of the language of the frontend. To implement a routing on the frontend the client must use java or implement Voldemort library[13].

Those are the two first architecture and represented on the left of figure 3.2. The left most use server routing in order to not constraint the frontend but loses latency and throughput. The middle one is the client side routing which impose the frontend to use java. The right most architecture represent an architecture where a request is directly routed to the right machine containing the data that the user wants.

The flexibility made by the choice of the architecture make high performance configuration possible. The biggest hit to performance is the disk access just before the number of hop. We can avoid the latter by choosing for example the httpd configuration. Disk access time can be avoided by partitioning dataset and make use of cache. Performance and its optimization will be discussed further in this paper in chapter 4

### Physical Architecture Options

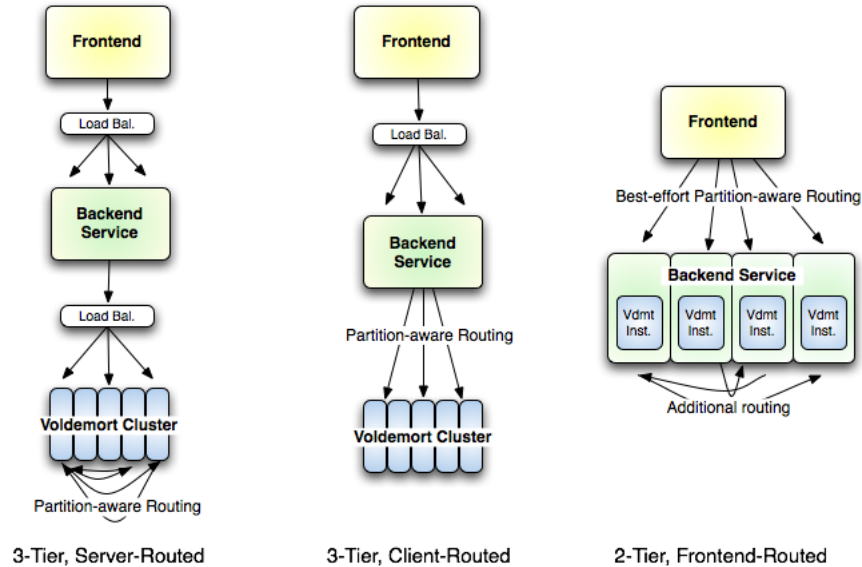


Figure 3.2: The physical architecture

## Data partitioning and replication

As explained in the last section, data need to be partitioned on multiple servers in order to avoid disk access time which is the biggest loss of time. Even if the data can hold on a single disk, it is needed to spread it on multiple servers. Retrieval of data is dominated by the seeking time of the disk, so by partitioning this data it is hoped that the smaller chunk will hold in memory reducing the seeking time of the next one. This mean that server can not be interchanged and that routing needs to be implemented.

From time to time, for whatever reason a server can go offline for a short period of time (overheat,ddos,...) or for an extended period of time if it simply die on us. This mean that the data needs also to be replicated on multiple server to avoid losing information. If a server S as a probability  $p$  to go offline, then the probability of losing at least one server is  $1 - (1 - p)^S$ . This is extra motivation to replicate the data.

A simple way[13] to accomplish this, is to cut our data into  $S$  partition (one per server) and store copies of a given key  $K$  on  $R$  servers. A way to associate  $R$  serveur and  $K$  key is to compute  $a = K \bmod S$  and store this value on server  $a, a+1, \dots, a+r$ .  $R$  must be chosen to have a low probability of data loss. With this method, the system can use peer-to-peer search because everyone can compute the location of the data knowing they key. The downside is the addition of knew server that can force data to shift around. This can be solved using what is called *consistent hashing*

Using consistent hashing when a server fails, the load will be distributed equally amongst the servers. When a new server is added, only  $1/(S + 1)$  values will be moved to the new machine. This is due to the propriety of this hash method. For further reading on this, the reader can look at this paper[4].One way to visualize this method is by using a ring containing each hash values as represented on figure 3.3. The ring is divided into  $Q$  equally sized partitions with  $Q \gg S$ . A key is mapped into the ring using a hashfunction, here md5.

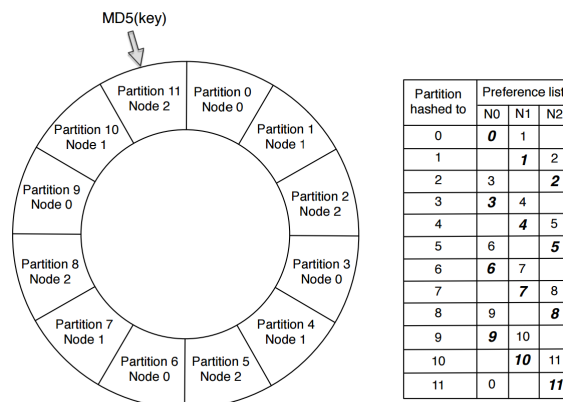


Figure 3.3: The hash ring for 3 nodes and 12 partitions [7]

### 3.1.3 Data Format and Queries

Voldemort does not have table but rather stores. Each key is mapped to one and only one store.

#### Queries

Voldemort supports hashable semantics, this means that a single value can be modified and that the search is done by primary key. There is no one to many relationship in this database but a value can be a list which has the same results. It is a huge performance improvement[13] because it will only have one disk seek. But for huge queries, it is not the case, rendering this method inefficient and the query must be broken into smaller sub-queries.

#### Data Model and Serialization

On the lowest level, the keys and values in the database are just byte-arrays. Higher-level of data format can be configure within each store but a serializer need to be implement in order to have a good translation between bytes and the high-level data format. The following types are supported "out of the box" : json, string, java-serialization, protobuf, thrift, avro-generic and identity. It is however recommended to use json format as it produce human readable content.

### 3.1.4 Consistency and Versioning

"The value for a given key is consistent if, in the absence of updates, all reads of that key return the same value"[13]. In a read only world, the data is created consistent then is never changed thus staying consistent. But when the data can change, it must change on every server it is replicated on and this is hard when server failure is a reality. When server partition is added into the mix, the task becomes impossible.

There exists different methodologies to reach consistency doing performance trade off. Those definitions comes from [13]:

- Two-Phase Commit : This is a locking protocol that involves two rounds of co-ordination between machines. It is perfectly consistent, but not failure tolerant, and very slow
- Paxos-style consensus : This is a protocol for coming to agreement on a value that is more failure tolerant.
- Read-repair : The first two approaches prevent permanent inconsistency. This approach involves writing all inconsistent versions, and then at read-time detecting the conflict, and resolving the problems. This involves little co-ordination and is completely failure tolerant, but may require additional application logic to resolve conflicts

Voldemort uses versioning and read-repair as it guarantees the best availability.

### 3.1.5 Versioning in a distributed system

On a centralized database, a simple versioning system can use a clock that is update with each write and store the time at which it was modified. But for a distributed database where servers are added and some are removed frequently this method fails. Here is an example : If two sever fetch the same values (triples) but each modify one different attribute then the server cannot decide which one to keep and which one to throw away.

An answer to this problem can be found by using a vector clock : [1:45,2:3,5:55]. The first number is the version of the document for which the server was the "master" and the second the number of writes done. So a version  $v1$  succeeds a version  $v2$  if for all  $i$ ,  $v1_i > v2_i$

## 3.2 Brewer's theorem

The CAP theorem[8], also known as the Brewer's theorem, state that for a distributed system, here a NoSQL data store, only two attributes from Consistency, Availability and Partition tolerance can be achieved. Consistency means that server must return the *right* response to each request. A database can (will) be inconsistent if it can not keep track of modification and so returning an old value. Availability means that every query will eventually receive a response. Partition tolerance means that the system will continue to work even when a failure occurs when two or more nodes try to connect to each other.

The creators of Voldemort has chosen availability and partition tolerance hence rejecting consistency. In section 3.1.4, consistency was discussed and this report went directly to the solution. But those solution to avoid consistency are made at read time leaving the database inconsistent. In order to implement versionning, Voldemort utilize a vector clock. Using the definition[13] it is possible to show this inconsistency. The definition is : "A version  $v1$  succeeds a version  $v2$  if for all  $i$ ,  $v1_i > v2_i$ ". So, if neither  $v1 > v2$  nor  $v1 < v2$ , then  $v1$  and  $v2$  co-occur, and they are in conflict. Voldemort has what is called *Eventual consistency*. It can be defined as followed : "In a steady state, the system will eventually return the last written value"[9].

## Chapter 4

# Performances and benchmarking

Performance is pivotal point of any software. It can make or break an idea or reduce its utility into niche areas. The reader will find feature performance and benchmark of this distributed database.

### 4.1 Performance vs. Scalability

Performance and scalability although often used together are different notions. A service is scalable if when added with more resources it results in an proportional increase of performance. In a distributed system, like Voldemort, that system is called scalable if when resources are added, for example to improve redundancy, it does not result in a loss of performance [14].

Performance, in another hand, is the ability for a service to have a certain response time [2]. It is something that can be defined with numbers. For our database, it will be the ability to respond to a query rapidly. Voldemort is a scalable database, the focus for the rest of this chapter will be around performance.

### 4.2 Performances

In this section, performance of Voldemort will be discussed. Those are mainly based on a paper [7] written from LinkedIn about read only store.

#### 4.2.1 Build Times

The main goal for voldemort is rapid data deployment [7, p. 10]. This mean that the build phase must be fast. They define the build time by "the time starting from the first mapper to the end of all reducers". For MYSQL, the build time is the completion time of the LOAD DATA INFILE command on an empty table.



On figure 4.1, a comparison between SQL and Voldemort is made for the build time of different sizes. SQL is drastically behind because it buffers changes in the index before writing them on the disk. Also, SQL does about 1.4 more I/O than voldemort due to the incremental index.

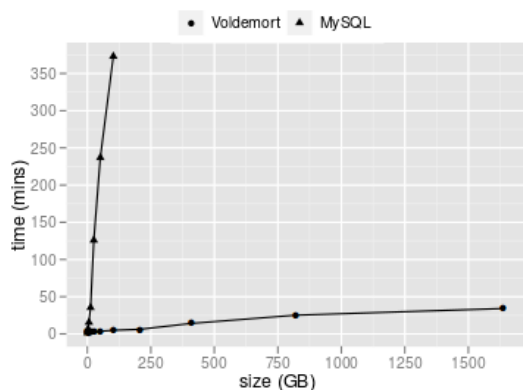


Figure 4.1: The time to complete the build for the random data set [7, p. 10]

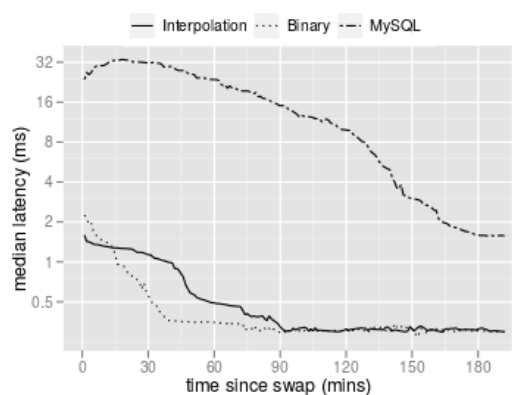


Figure 4.2: Single node median read latency taken at 1 minute intervals since the swap [7, p. 10]

## 4.2.2 Read Latency

The read latency must be acceptable and it must scale with the number of nodes. LinkedIn made some experiment with 10 million requests with simulated keys following a uniform distribution between 0 to number of tuples in the data set [7, p. 10]

First was measured the speed of loading pages into the operating system cache. A test was run with 100GB of data and the median latency after swap was taken. The results can be seen on figure 4.2. For MySQL, they created a view on an existing table, bulk loaded into a new table, and swapped the view to the new table without stopping the request. MySQL starts slowly because it need to build his cache. Voldemort build his cache much more rapidly due to his binary search that does a bit more look up than an interpolation one.

On the figure 4.3 is a comparison with SQL on the same 100GB bit with different throughput. This test shows that Voldemort scale to twice the amount of query that MySQL can handle while maintaining the same latency median.

The last Test will determine if the database scales with the number of nodes. The test is the same as the percent ones but this time the data is spread over 32 machines and a store. The query were for one following a uniform distribution and the other one the Zipfian distribution. The latter distribution will query some keys more than other to simulate a real world utilization of the database. The result can be seen on figure 4.4. This figure shows that indeed Voldemort scale with nodes.

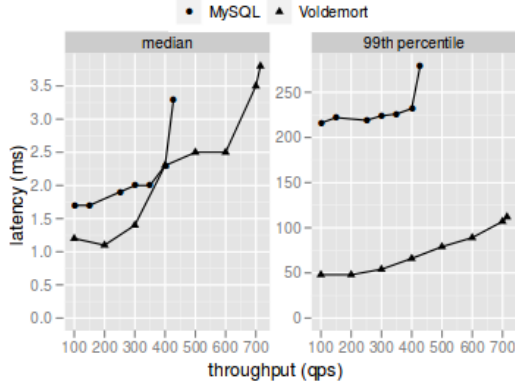


Figure 4.3: Single node read latency after warm-

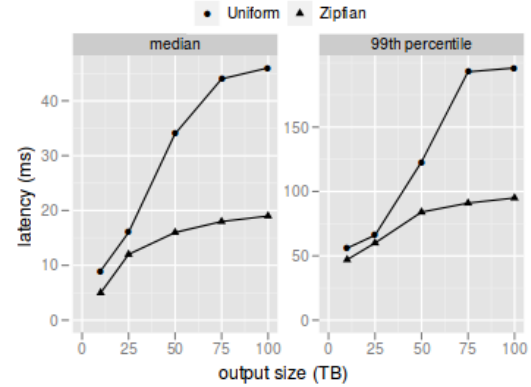


Figure 4.4: Client-side median latency with varying data size

### 4.2.3 Solid State Drive

Using solid state drive (ssd) offer huge boost over Serial Attached SCSI [11]. The average latency drop from 30ms to 2ms and the speed of cluster expansion improved 10x. But for the 5% top percentile, there is an increased latency from Java garbage collector. Those were due to linux stealing pages from JVM heap. It is important to use `mlock()` on the server heap in order to avoid this.

## 4.3 Benchmarks

This section will firstly benchmark the performance of voldemort against Cassandra in term of throughput and failover.

### 4.3.1 Workload impact on latency

The benchmark will be created using five workloads[1]:

**Workload A: Update heavy workload** This workload consists of 50% read operations and 50% update operations

**Workload B: Read mostly workload** This workload consists of 95% reads and 5% updates

**Workload C: Read only workload** This workload consists of 100% read operations

**Workload D: Read latest workload** This workload consists of 95% read and 5% insert operations. The most recently inserted records are the most popular

**Workload E: Short ranges workload** This workload consist of 5% insert and 95% scan operations new records are inserted and short ranges of records are queried using scan operations

**Workload F: Read-modify-write workload** This workload consists of 50% read and 50% read-modify-write operations implemented as sequential read and update operations on the same key.

On figure 4.5, can be seen the result for the workload A-E for Cassandra. Those show that the latency increase with throughput. For the workload A, C, D and F, it increase almost linearly with throughput although the throughput of F is less than the other ones. For workload E, the latency increases almost exponentially. This is due to the fact that the scan operation is more resource intensive than a read or write.

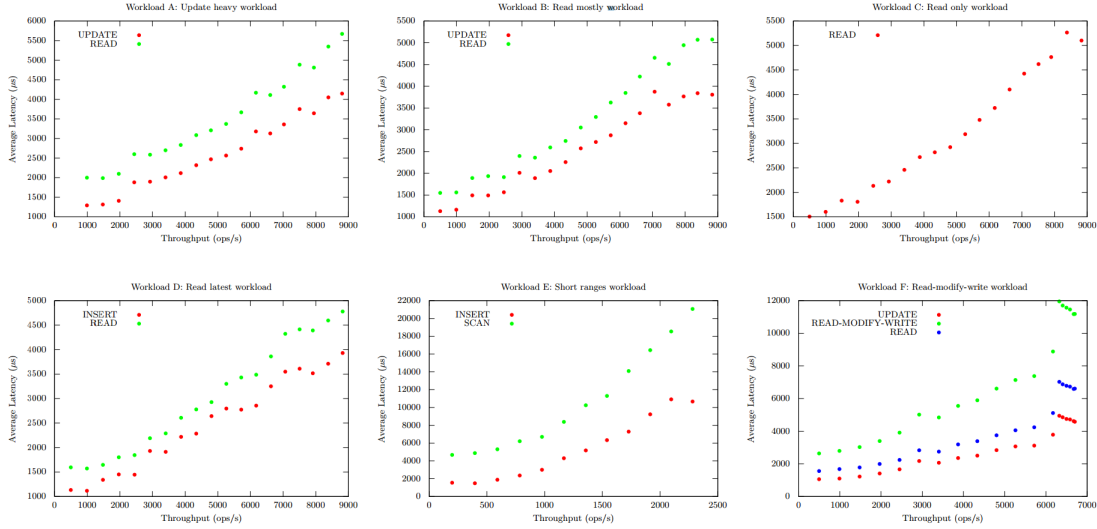


Figure 4.5: Workload A-E for Cassandra 2.0.2[1, p. 7]

On the figure 4.6, are displayed the result for the same tests but applied to Voldemort. The most important different between those two result is that for Voldemort it seems that there is no direct relation between latency and throughput. Only for workload C a relationship is clearly visible.

Maximum achieved throughput for both database are displayed on figure 4.7 and figure 4.7. We can see that for 1 million records cassandra is the clear winner in terms of throughput and manage to keep a "low" amount of server load. This change a bit with 50 millions records, Cassandra cretae more server load than Voldemort but can still handle more throughput than its counterpart.

### 4.3.2 Failover Characteristics

The goal here is to monitor the latency and throughput over time when a node fails and then come back up. The test is the following. Both database have a workload of 50% read and % write operations. After 30s a node will be killed and then will be bring back only after 300s. The graphs shown on 4.10 and 4.11 show what happens to those database in this scenario and with a set of 50 millions records. Other test has been done but will not be explained here. For more information and graph about this benchmark, the interested reader can read the complete paper [1].

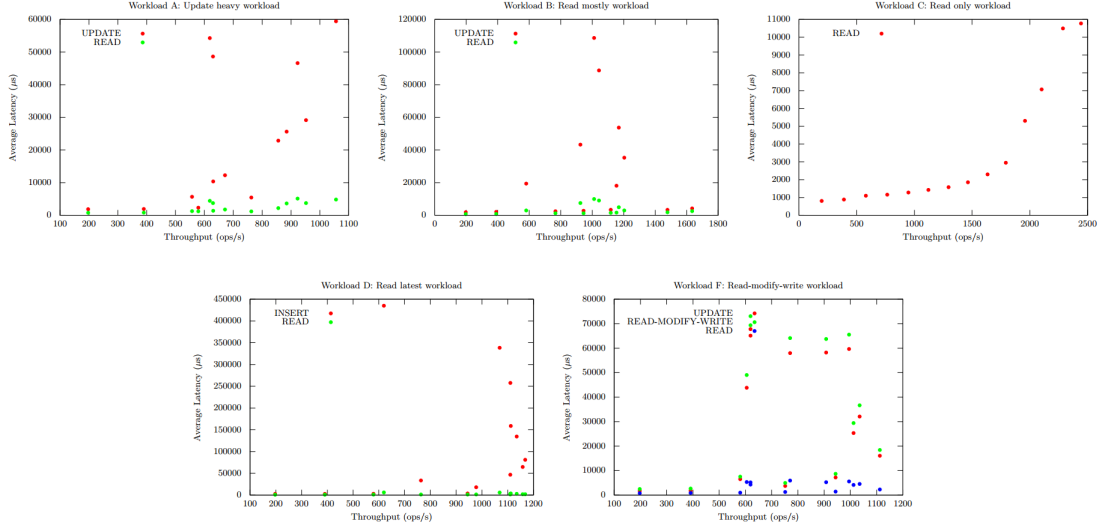


Figure 4.6: Workload A-D and f for Voldemort 1.3.0 [1, p. 8]

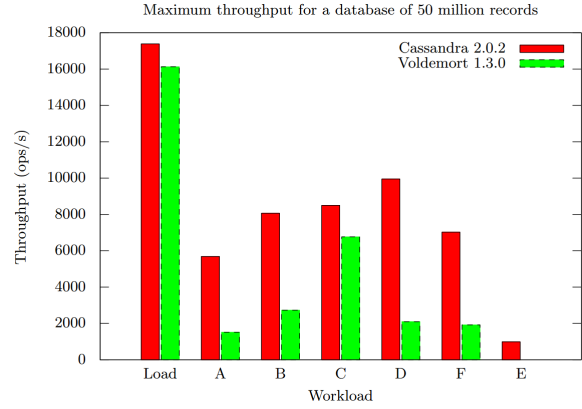
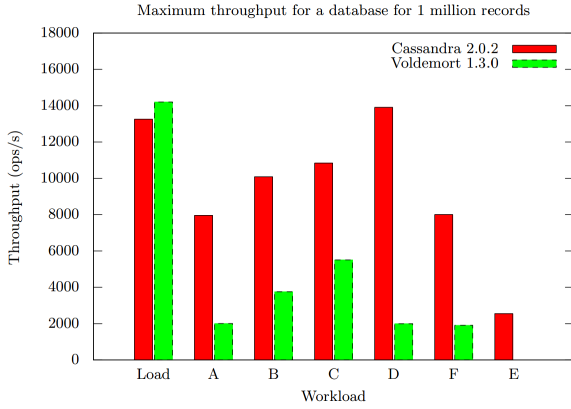


Figure 4.7: Maximum throughput for 1M recordsFigure 4.8: Maximum throughput for 50M records[1, p. 8]

For most of the lighter test, Cassandra performed well throughout the entire test even when Voldemort had issues with latency. It is important to note that Voldemort even though had some latency issues manage to keep a high stable throughput. Similar characteristics can be seen on both graphs. A spike a latency occurs when the node goes down. This is mostly due to the fact that the client must manage all the transaction that the node was working on. Once the node comes back up, a huge spike of latency arise, this comes from the fact that adding a node to the pool is much more expensive then removing one. Every node must update any stale data before the new node can join them. Throughput shows also some interesting trend in those graphs. When Cassandra's nodes start updating their stale data there is a big drop in the throughput. It is as all the nodes stopped worked for a moment before the new node comes back.

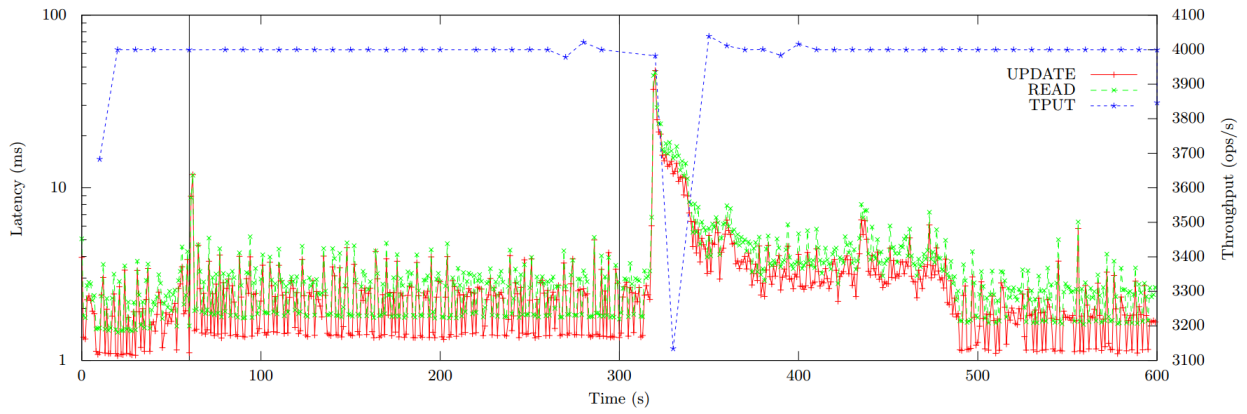


Figure 4.9: Update heavy workload with one node failure at 50% of maximum throughput for Cassandra 2.0.2 [1, p. 11]

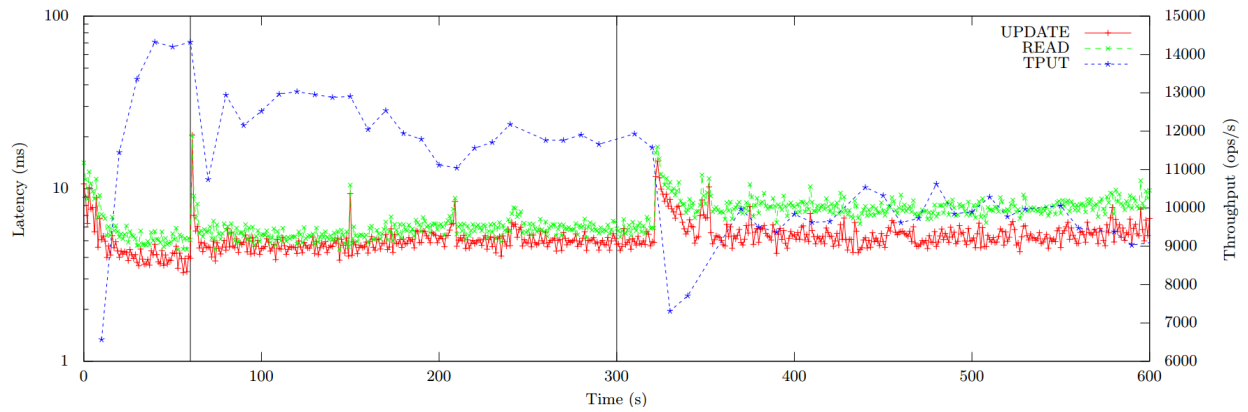


Figure 4.10: Update heavy workload with one node failure at maximum throughput for Cassandra 2.0.2 [1, p. 11]

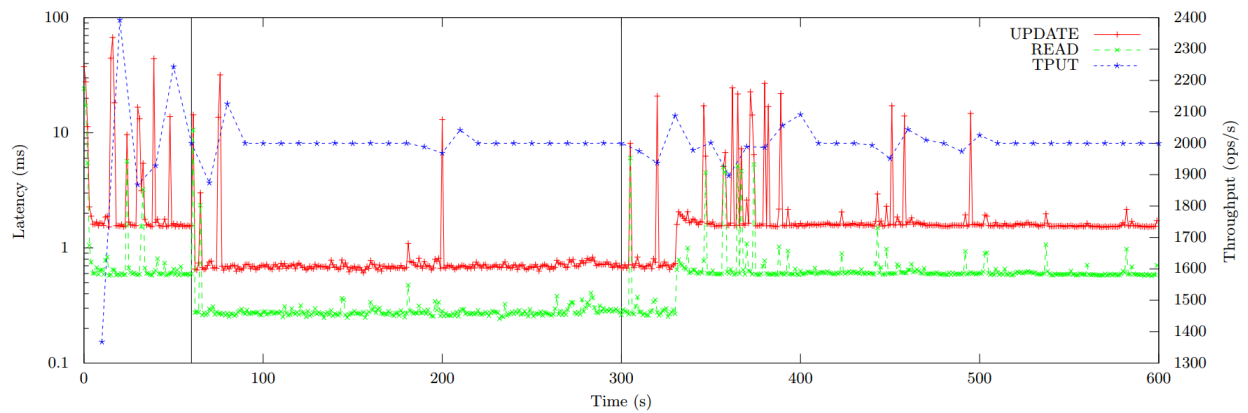


Figure 4.11: Update heavy workload with one node failure at maximum throughput for Voldemort 1.3.0 [1, p. 11]

## Chapter 5

# Creating distributed database servers with Voldemort

This chapter will present the creation of clusters for server side of Voldemort database. The server will be installed on clean *ubuntu-server-14.04.3* running inside a virtual machine.

### 5.1 Installing the source code

In order to install and build Voldemort source code, following applications have to be added:

```
sudo apt-get install git
sudo apt-get install default-jre
sudo apt-get install default-jdk
sudo apt-get install python2.7
```

Next steps are copying the source code from git and compiling it with existing java makefile named *gradlew*:

```
git clone https://github.com/voldemort/voldemort.git
cd voldemort
./gradlew clean jar
```

To start the server itself navigate to folder */bin* and run the script *voldemort-server.sh* by specifying the configuration options from */config* folder. Here is an example for single node cluster:

```
bin/voldemort-server.sh config/single_node_cluster
```

### 5.2 Server configuration

Server configuration consists of 3 files:

- **cluster.xml** - holds the information about all the nodes/servers in the cluster, what hostname they are at, the ports they use, etc. It is exactly the same for all voldemort nodes. It does not hold tuning parameters or data directories for those nodes, since that is not information public to the cluster but is specific to that particular nodes configuration.
- **stores.xml** - holds the information about all the stores/tables in the cluster. This includes information about the required number of successful reads to maintain consistency, the required number of writes, as well as how keys and values are serialized into bytes. It is the same on all nodes in the cluster.
- **server.properties** - contains the tuning parameters that control a particular node. This includes the id of the local node so it knows which entry in cluster.xml corresponds to itself, also the thread pool size, as well as any configuration needed for the local persistence engine such as BDB or mysql. This file is different on each node.

[12] Two virtual machines are going to be deployed to show the implementation of distributed databases across multiple nodes. Following paragraphs are going to show the configurations needed. Two servers are run on virtual machines: *192.168.1.4* and *192.168.1.5* - the client will be using the local machine *192.168.1.2*, all bridge connected. We navigate to config folder and define the *2\_ - node\_cluster/config/* folder to put the following config files inside.

### 5.2.1 Cluster configuration

As said before this file will be duplicated on both virtual servers. This file defines all the servers IPs and ports running the distributed database and partitions they have access to. Each line name is intuitive and does not require further explanations:

```
<cluster>
  <name>db_cluster</name>
  <server>
    <id>0</id>
    <host>192.168.1.4</host>
    <http-port>8081</http-port>
    <socket-port>6666</socket-port>
    <partitions>0, 1</partitions>
  </server>
  <server>
    <id>1</id>
    <host>192.168.1.5</host>
    <http-port>8085</http-port>
    <socket-port>6670</socket-port>
    <partitions>2, 3</partitions>
```

```

    </server>
</cluster>

```

### 5.2.2 Stores configuration

We add the *STORES/* directory with **xml** files representing the stores (tables for SQL) the database contains. Here is the testing table:

```

<store>
  <name>test</name>
  <persistence>bdb</persistence>
  <description>Test store</description>
  <owners>nfilipch@ulb.ac.be,bakboka@ulb.ac.be</owners>
  <routing-strategy>consistent-routing</routing-strategy>
  <routing>client</routing>
  <replication-factor>2</replication-factor>
  <required-reads>1</required-reads>
  <required-writes>1</required-writes>
  <key-serializer>
    <type>string</type>
  </key-serializer>
  <value-serializer>
    <type>string</type>
  </value-serializer>
  <hinted-handoff-strategy>consistent-handoff</hinted-handoff-strategy>
</store>

```

Some descriptive data is added first but implementation choices have to be added as well, such as routing, replication and data type used for storing. This file as well has to be duplicated over all servers.

### 5.2.3 Each server properties

The last file is specific for each node and indicates what part of the cluster they represent:

```

# The ID of *this* particular cluster node
node.id=0

max.threads=100

##### DB options #####

```



```
http.enable=true
socket.enable=true
```

```
# BDB
```

```
bdb.write.transactions=false
bdb.flush.transactions=false
bdb.cache.size=1G
bdb.one.env.per.store=true
```

```
# Mysql
```

```
mysql.host=localhost
mysql.port=1521
mysql.user=root
mysql.password=3306
mysql.database=test
```

```
#NIO connector settings.
```

```
enable.nio.connector=true
```

```
request.format=vp3
```

```
storage.configs=voldemort.store.bdb.BdbStorageConfiguration, voldemort.store.readonly
```

The node id and number of threads describe server functions, some additional options for server connectivity may be added as shown in this example.

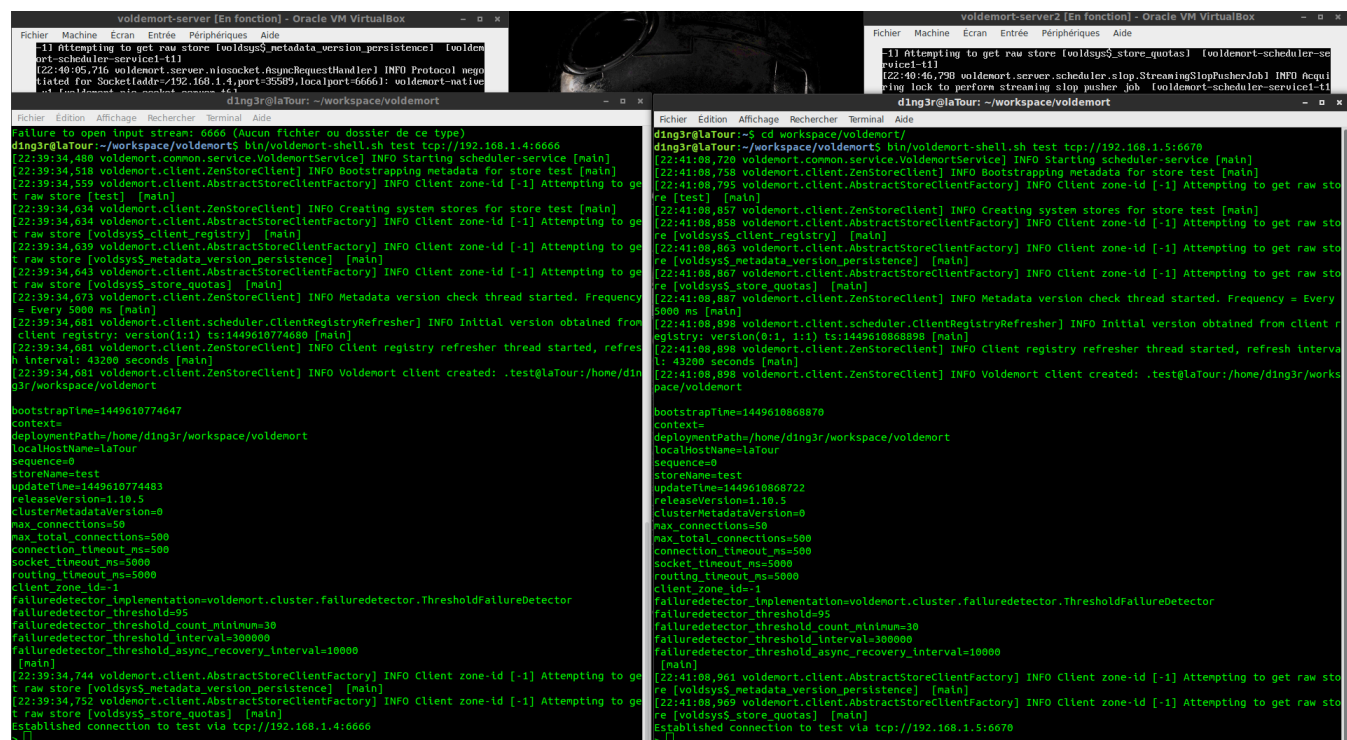
## Chapter 6

# Using the database from the client side

From the client side we have to install and compile same voldemort source code on the host machine. After doing so, navigate to the Voldemort folder and run the voldemort shell:

```
bin/voldemort-shell.sh test tcp://192.168.1.4:6666/
```

Where we give the following parameters - the name of the store we are working on, one of the servers IP address and the port on this one. After the connection is established we can use the database from the shell.



```
voldemort-server [En fonction] - Oracle VM VirtualBox
Fichier Machine Écran Entrée Périphériques Aide
[1] Attempting to get raw store [voldsys$metadata_version_persistence] [voldemort-scheduler-service-t1]
[22:40:05,716 voldemort.server.niosocket.AsyncRequestHandler] INFO Protocol negotiated for Socket[addr=192.168.1.4, port=35589, localport=6666]: voldemort-native
ding3r@laTour: ~/workspace/voldemort

voldemort-server2 [En fonction] - Oracle VM VirtualBox
Fichier Machine Écran Entrée Périphériques Aide
[1] Attempting to get raw store [voldsys$store_quotas] [voldemort-scheduler-service-t1]
[22:40:46,798 voldemort.server.scheduler.slop.StreamingSlopPusherJob] INFO Acquiring lock to perform streaming slop pusher job [voldemort-scheduler-service-t1]
ding3r@laTour: ~/workspace/voldemort

ding3r@laTour:~/workspace/voldemort$ bin/voldemort-shell.sh test tcp://192.168.1.4:6666/
[22:39:34,488 voldemort.common.service.VoldemortService] INFO Starting scheduler-service [main]
[22:39:34,518 voldemort.client.ZenStoreClient] INFO Bootstrapping metadata for store test [main]
[22:39:34,559 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [test] [main]
[22:39:34,634 voldemort.client.ZenStoreClient] INFO Creating system stores for store test [main]
[22:39:34,634 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$client_registry] [main]
[22:39:34,639 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$metadata_version_persistence] [main]
[22:39:34,643 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$store_quotas] [main]
[22:39:34,673 voldemort.client.ZenStoreClient] INFO Metadata version check thread started. Frequency = Every 5000 ms [main]
[22:39:34,681 voldemort.client.scheduler.clientRegistryRefresher] INFO Initial version obtained from client registry: version(1:1) ts:1449610774680 [main]
[22:39:34,681 voldemort.client.ZenStoreClient] INFO Client registry refresher thread started, refresh interval: 43200 seconds [main]
[22:39:34,681 voldemort.client.ZenStoreClient] INFO Voldemort client created: .test@laTour:/home/ding3r/workspace/voldemort
bootstrapTime=1449610774647
context=
deploymentPath=/home/ding3r/workspace/voldemort
localHostName=laTour
sequence=0
storeName=test
updateTime=1449610774483
releaseVersion=1.10.5
clusterMetadataVersion=0
max_connections=50
max_total_connections=500
connection_timeout_ms=500
socket_timeout_ms=5000
routing_timeout_ms=5000
client_zone_id=-1
faileddetector_implementation=voldemort.cluster.faileddetector.ThresholdFailureDetector
faileddetector_threshold=95
faileddetector_threshold_count_minimum=30
faileddetector_threshold_interval=300000
faileddetector_threshold_async_recovery_interval=10000
[main]
[22:39:34,744 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$metadata_version_persistence] [main]
[22:39:34,752 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$store_quotas] [main]
Established connection to test via tcp://192.168.1.4:6666
> [1]

ding3r@laTour:~/workspace/voldemort$ bin/voldemort-shell.sh test tcp://192.168.1.5:6670/
[22:41:08,728 voldemort.common.service.VoldemortService] INFO Starting scheduler-service [main]
[22:41:08,758 voldemort.client.ZenStoreClient] INFO Bootstrapping metadata for store test [main]
[22:41:08,795 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [test] [main]
[22:41:08,857 voldemort.client.ZenStoreClient] INFO Creating system stores for store test [main]
[22:41:08,858 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$client_registry] [main]
[22:41:08,863 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$metadata_version_persistence] [main]
[22:41:08,867 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$store_quotas] [main]
[22:41:08,887 voldemort.client.ZenStoreClient] INFO Metadata version check thread started. Frequency = Every 5000 ms [main]
[22:41:08,898 voldemort.client.scheduler.clientRegistryRefresher] INFO Initial version obtained from client registry: version(0:1, 1:1) ts:1449610868898 [main]
[22:41:08,898 voldemort.client.ZenStoreClient] INFO Client registry refresher thread started, refresh interval: 43200 seconds [main]
[22:41:08,898 voldemort.client.ZenStoreClient] INFO Voldemort client created: .test@laTour:/home/ding3r/workspace/voldemort
bootstrapTime=1449610868870
context=
deploymentPath=/home/ding3r/workspace/voldemort
localHostName=laTour
sequence=0
storeName=test
updateTime=1449610868722
releaseVersion=1.10.5
clusterMetadataVersion=0
max_connections=50
max_total_connections=500
connection_timeout_ms=500
socket_timeout_ms=5000
routing_timeout_ms=5000
client_zone_id=-1
faileddetector_implementation=voldemort.cluster.faileddetector.ThresholdFailureDetector
faileddetector_threshold=95
faileddetector_threshold_count_minimum=30
faileddetector_threshold_interval=300000
faileddetector_threshold_async_recovery_interval=10000
[main]
[22:41:08,961 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$metadata_version_persistence] [main]
[22:41:08,969 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$store_quotas] [main]
Established connection to test via tcp://192.168.1.5:6670
> [1]
```

Figure 6.1: Establishing the connection to both clusters of the database from same client

```

ding3r@laTour: ~/workspace/voldemort
LocalHostName=laTour
sequence=0
storeName=test
updateTime=1449610774483
releaseVersion=1.10.5
clusterMetadataVersion=0
max_connections=50
max_total_connections=500
connection_timeout_ms=500
socket_timeout_ms=5000
routing_timeout_ms=5000
client_zone_id=1
failedetector_implementation=voldemort.cluster.failedetector.ThresholdFailureDetector
failedetector_threshold=95
failedetector_threshold_count_minimum=30
failedetector_threshold_interval=300000
failedetector_threshold_async_recovery_interval=10000
[main]
22:39:34,744 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$metadata_version_persistence] [main]
22:39:34,752 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$store_quotas] [main]
Established connection to test via tcp://192.168.1.4:6666
> put "hello" "world"
> []

ding3r@laTour: ~/workspace/voldemort
sequence=0
storeName=test
updateTime=1449610868722
releaseVersion=1.10.5
clusterMetadataVersion=0
max_connections=50
max_total_connections=500
connection_timeout_ms=500
socket_timeout_ms=5000
routing_timeout_ms=5000
client_zone_id=1
failedetector_implementation=voldemort.cluster.failedetector.ThresholdFailureDetector
failedetector_threshold=95
failedetector_threshold_count_minimum=30
failedetector_threshold_interval=300000
failedetector_threshold_async_recovery_interval=10000
[main]
22:41:00,961 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$metadata_version_persistence] [main]
22:41:00,969 voldemort.client.AbstractStoreClientFactory] INFO Client zone-id [-1] Attempting to get raw store [voldsys$store_quotas] [main]
Established connection to test via tcp://192.168.1.5:6670
> get "hello"
version(1:1) ts:1449610900216: "world"
> []

```

Figure 6.2: Inputting information into first cluster and retrieving from the other.

```

Established connection to test via tcp://192.168.1.5:6666/
> put "Bruxelles" "16C"
> put "Paris" "18C"
> put "Budapest" "20C"
> get "Bruxelles"
version(1:1) ts:1451061510541: "16C"
> get "Budapest"
version(0:1) ts:1451061545118: "20C"
> delete "Budapest"
> get Budapest
Error serializing values: voldemort.serialization.SerializationException: Unacceptable initial character
    at voldemort.serialization.json.JsonReader.read(JsonReader.java:111)
    at voldemort.VoldemortClientShell.parseObject(VoldemortClientShell.java:240)
    at voldemort.VoldemortClientShell.parseKey(VoldemortClientShell.java:273)
    at voldemort.VoldemortClientShell.processGet(VoldemortClientShell.java:327)
    at voldemort.VoldemortClientShell.evaluateCommand(VoldemortClientShell.java:382)
    at voldemort.VoldemortClientShell.processCommands(VoldemortClientShell.java:353)
    at voldemort.VoldemortClientShell.process(VoldemortClientShell.java:152)
    at voldemort.VoldemortClientShell.main(VoldemortClientShell.java:229)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:606)
    at jline.ConsoleRunner.main(ConsoleRunner.java:69)
> get "Budapest"
null
> []

```

Figure 6.3: Few basic commands on string data-set.

## Chapter 7

# Conclusion

Voldemort has proven to be a highly available and partition safe while maintaining high throughput. It manages to avoid inconsistency but using trade off like the two-commit method. But it is a rather new software that is mainly used by the company LinkedIn which make little to no documentation about it. This is a major problem appears when a problem occurs during implementation. A solution can be hard to found has there is not that much community around it.

In terms of performance, it holds it own against its competitors. His main one is Cassandra who manage to have better latency when a node die but also maintain higher throughput at almost any time than Voldemort. But it has been created to manage large set of data and do it well. When Cassandra would crash during heavy load and failure it manages to carry on with the workload.

# Bibliography

- [1] Pokluda Alexander and Sun Wei. Benchmarking failover characteristics of large-scale data storage applications: Cassandra and voldemort.
- [2] Reitbauer Alois. <http://apmblog.dynatrace.com/2008/09/11/performance-vs-scalability/>. 2008. <http://apmblog.dynatrace.com/2008/09/11/performance-vs-scalability/>.
- [3] Strauch Christof. Nosql databases.
- [4] Karger David, Lehman Eric, Leighton Tom, Levine Matthew, Lewin Daniel, and Rine Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web.
- [5] DeCandia Giuseppe, Hastorun Deniz, Jampani Madan, Kakulapati Gunavardhan, Lakshman Avinash, Pilchin Alex, Sivasubramanian Swaminathan, Voshall Peter, and Vogels Werner. Dynamo: Amazon’s highly available key-value store.
- [6] Kreps Jay. Project voldemort: Scaling simple storage at linkedin. 2009. <http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin/>.
- [7] Sumbaly Roshan, Kreps Jay, Gao Lei, Feinberg Alex, Soman Chinmay, and Shah Sam. Serving large-scale batch computed data with project voldemort. 2012.
- [8] Gilbert Seth and A. Lynch Nancy. Perspectives on the cap theorem. <https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf>.
- [9] Loch Todd. Design patterns for distributed non-relational databases. 2009. <http://www.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelationaldatabases>.
- [10] Unknown. Distributed databases. [http://wps.pearsoned.co.uk/wps/media/objects/10977/11240737/Web%20chapters/Chapter%2012\\_WEB.pdf](http://wps.pearsoned.co.uk/wps/media/objects/10977/11240737/Web%20chapters/Chapter%2012_WEB.pdf).
- [11] Chandar Vinoth, Gao Lei, and Tran Cuong. Voldemort on solid state drives. May 2012. <http://www.slideshare.net/vinothchandar/voldemort-on-solid-state-drives>.

- [12] Project Voldemort. Configuration. <http://www.project-voldemort.com/voldemort/configuration.html>.
- [13] Project Voldemort. Design. <http://www.project-voldemort.com/voldemort/design.html>.
- [14] Vogels Werner. A word on scalability. 2006. [http://www.allthingsdistributed.com/2006/03/a\\_word\\_on\\_scalability.html](http://www.allthingsdistributed.com/2006/03/a_word_on_scalability.html).