DATA DE-DUPLICATION IN NoSQL DATABASES

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree Masters of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

NICOLETA C. BRAD

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan (S7N 5C9)

Canada

ABSTRACT

With the popularity and expansion of Cloud Computing, NoSQL databases (DBs) are becoming the preferred choice of storing data in the Cloud. Because they are highly de-normalized, these DBs tend to store significant amounts of redundant data. Data de-duplication (DD) has an important role in reducing storage consumption to make it affordable to manage in today's explosive data growth. Numerous DD methodologies like chunking and, delta encoding are available today to optimize the use of storage. These technologies approach DD at file and/or sub-file level but this approach has never been optimal for NoSQL DBs.

This research proposes data De-Duplication in NoSQL Databases (DDNSDB) which makes use of a DD approach at a higher level of abstraction, namely at the DB level. It makes use of the structural information about the data (metadata) exploiting its granularity to identify and remove duplicates. The main goals of this research are: to maximally reduce the amount of duplicates in one type of NoSQL DBs, namely the key-value store, to maximally increase the process performance such that the backup window is marginally affected, and to design with horizontal scaling in mind such that it would run on a Cloud Platform competitively. Additionally, this research presents an analysis of the various types of NoSQL DBs (such as key-value, tabular/columnar, and document DBs) to understand their data model required for the design and implementation of DDNSDB.

Primary experiments have demonstrated that DDNSDB can further reduce the NoSQL DB storage space compared with current archiving methods (from 17% to near 69% as more structural information is available). Also, by following an optimized adapted MapReduce architecture, DDNSDB proves to have competitive performance advantage in a horizontal scaling cloud environment compared with a vertical scaling environment (from 28.8 milliseconds to 34.9 milliseconds as the number of parallel Virtual Machines grows).

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Ralph Deters for his continuous guidance, support and patience throughout the entire period of my studies. Also, I would like to thank my family (my husband Daniel and my daughter Arina) for their continuous support. And last, I would especially like to thank my mother for all her help in the last two years so that I can be successful.

TABLE OF CONTENTS

page

LIST OF TABLES

Table page

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACID	Atomicity, Consistency, Isolation, Durability
AM	All Metadata
API	Application Programming Interface
AWG	Average
BASE	Basically Available, Soft state, Eventually consistent
CAP	Consistency, Availability, and Partition tolerance
CC	Cloud Computing
CP	Cloud Platforms
CS	Cloud Services
DBs	Databases
DBMS	Database Management Systems
DC	Distributed Computing
DD	De-Duplication
DDR	De-Duplication Ratio
DDNSDB	Data De-duplication in NoSQL Databases
DS	Data Store
EC2	Amazon Elastic Cloud Computing
EVM	Erlang Virtual Machine
IaaS	Infrastructure as a Service
JSON	Java Script Object Notation
LAN	Local Area Network
MD5	Message-Digest Algorithm

NAS	Network-Attached Storage
NM	No Metadata
NoSQL	Not Only SQL
PaaS	Platform as a Service
RDBMS	Relational Databases Management Systems
REST	Representational State Transfer
SaaS	Software as a Service
SM	Some Metadata
SSTable	Sorted String Table
SQL	Standard Query Language
T1-3	Topology1,2, and 3
VLT	Virtual Tape Library
WAN	Wide Area Network
XML	Extensible Markup Language

CHAPTER 1
INTRODUCTION

Cloud computing is transforming computing into a utility like service, changing the scale of computing operations. The cloud platforms composed of storage, computational power, and web access are becoming the choice for deploying highly available and scalable systems, changing the data landscape. At the same time the rapid growth of data pushed by Web 2.0 companies, social networking and user contributed content brings new challenges to the DB management systems, compelling them to consider data storage options beyond the traditional SQL-based relational DBs. Properties like elasticity and high availability are becoming increasingly important for these systems.

To be able to store and analyze the rich information, through custom engineering development at large web sites and services like Amazon, Google, Yahoo!, Facebook, etc., new types of DBs have emerged called NoSQL (Not Only SQL). More specifically, they are designed with horizontal scaling, availability, cost, and performance in mind. NoSQL DBs introduce new storage architectures that scale horizontally and parallel algorithms designed to efficiently process the distributed data (MapReduce being the most popular example). Many of the NoSQL DBs are open source and run on commodity hardware, making them significantly less expensive per terabyte than traditional DBs from vendors. NoSQL DBs introduce also new data structures, more appropriate to the web based data [14].

Three main types of NoSQL DBs are becoming more popular these days: the key-value DBs, columnar/tabular DBs, and document DBs. One problem that they all have in common is the amount of redundant data which they store because of their highly de-normalized structure.

One can argue that storage is cheap and getting cheaper but besides the price of acquiring the storage, companies also have to be able to efficiently store and maintain this data. The amount of processing power and energy needed to handle and manage the data, the network resources for transmitting of the data to different locations, the time and resources dedicated to backup and replication all add to the *"cheap storage"* cost, making it not *"cheap"* anymore.

Data DD has received broad attention both in industry [21; 29; 30; 39; 28; 22], and academia [37; 23; 33; 36; 43; 53; 13; 26] in recent years as the method to optimize storage capacity. The way DD works is by detecting exact copies of data blocks or by detecting similar or near duplicate blocks and storing the difference.

Some more mature NoSQL DBs like Amzon's BigTable have implemented optional archiving - a combination of different DD algorithms combined with archiving – which is applied to each SSTable (Sorted String Table) block [12] to help reduce the duplicated data. The SSTable is *"…an on-disk file format that represents a string-to-string mapping"* [49] and consists of immutable key-value pairs.

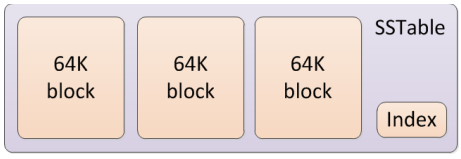

Figure 1.1: SSTable internal structure.

An intuitive assumption is to approach DD at the DB level and to make use of the structural information about the data to locate the duplicate data. This is another layer which can be added to help reduce duplicate data in NoSQL DBs. The effectiveness of this approach

depends on the granularity of structural information available in the DBs. There are reasons why we should not have too much structure, like fast response, and there are also reasons why we should have more structure, like reducing duplicate data. Although these two concepts seem very contradictory in the sense that you can only have one at a time, there is no reason why we should not try and take advantage of both at the same time. The way data is stored with minimal structure does not have to be affected, maintaining the quick access response. Instead, a new layer is introduced for storing separately the additional structural information, and making use of it only at the time of backup when the DD process needs it.

The key contributions of this research are:

- Introduces of a new concept of approaching DD at the NoSQL DB level using the metadata for chunking.
- Creates a comparison between the three main types of NoSQL DBs.
- Develops a DD tool using the MapReduce architecture for the key-value DBs as a proof of concept.

The benefit of such a DD tool is that it can easily be adapted for other types of NoSQL DBs, and that there are no major changes required by the specialized backup tools to integrate with.

The rest of the document is organized as follows: Section two presents the duplicate data problem present in the NoSQL DBs. Section three reviews the related work about the four areas of interest involved in this research namely: Cloud Computing and MapReduce principals, NoSQL DBs, data DD, and DB backups. Section four introduces the data model of the three main types of NoSQL DBs (key-value, columnar, and document based), and explains why the key-value DB was chosen for the prototype. Furthermore, it investigates and points out the importance of structural information used in the chunking for DD in NoSQL DBs. Then, it presents how the design considerations for DDNSDB share the MapReduce programming

principals for performance and scalability of the DD process. Section five describes the architectural design and execution overview of DDNSDB. Section six describes the evaluation approach, experimental setup, and the datasets used in the experiments. Section seven describes the experiments and the evaluations of the DDNSDB. Section eight summarizes the research contribution and presents potential future work.

CHAPTER 2
PROBLEM DEFINITION

2.1 Applying Data De-duplication to NoSQL Database Backup

NoSQL DBs generally follow a simple data model with dynamic control over data layout and form. There are three main types of NoSQL DBs based on their data model: key-value DBs, columnar DBs, and document DBs. Due to their simpler design compared with the relational databases which are highly normalized, they tend to have a large amount of duplicated data. In distributed environments where NoSQL database are used, redundancy is desired but that is done in a controlled manner generally through replication. What we are referring to is the uncontrolled duplicate data which is a result of highly de-normalized structures. This duplicate data is then further propagated into the backups increasing the storage requirements even more. While it made no sense to approach DD at the DB level for RDBMS, it makes a lot of sense to approach it for NoSQL DBs.

This research is using the key-value DB as the representative structure for its De-Duplication of NoSQL Databases (DDNSDB) implementation. The key-value DB is the most popular type of NoSQL DB used today though it has the least complex data model. Like an associative array composed of a collection of unique keys and a collection of values where each key is associated with one value or a set of values, the values of a Key-value DB can be simple attributes or a vector of attribute-value pairs. This type of

structure gives flexibility to create very complex schema-less structures with a very fast retrieval, based on the unique key.

DD approaches can be divided into two broad categories: hardware DD approaches and software DD approaches. For backup systems at software level, two other categories can be distinguished based on the placement: source DD where duplicate data is identified at the server being backed up, and before it is sent across the network or target where DD is presented to the backup server as a Network-Attached Storage (NAS) share or Virtual Tape Library (VLT). For the target option modern storage technology (e.g. large computational resources at disk array controllers like Network-Attached Storage (NAS) share, and Virtual Tape Library (VTL) controllers) has been the choice of placement for the DD technology. This option however, rules out the possibility of using DD algorithms which are content aware, plus all the data has to cross over the LAN/WAN contributing to the increase of network traffic. DD at the target/client reduces the network traffic, and the technology can be embedded in the backup architecture. The internal DB information is only available at the client/DB level, making the placemat of the DD process an easy choice for this research.

Existing DD technology reduces the cost of storage and network traffic making it more affordable but many companies are still struggling to complete their backups on a regular scheduled basis. Data loss is similar to hurricanes, leaving behind devastated enterprises which may not be able to ever recover. Because all data is mostly organized in a form of files, existing technologies mostly use file and sub-file DD strategies. Various chunking strategies provide higher or lower DD ratios depending on the type of data and how much the strategies are aware of the content. But this has been proven as not enough.

This novel idea of approaching DD at the databases level where metadata information can be made available to help identifying duplicate data, can be used in parallel with existing DD approaches. This will increase the overall DD ratio and implicitly reduce the data footprint.

There are two main challenges in the DD process of NoSQL database:

Challenge 1- Granularity of the structural information – The structural information is the key element used in identifying duplicate data. Generally, the finer the granularity, the higher the probability of finding duplicate data but it can become very costly compared with the increase in the DD ratio obtained.

Challenge 2 - Scalability – The backup window is very limited regardless of the data growth. At the same time, the backup process is a very intensive I/O process, while the DD process is a CPU and memory intensive operation. By adding a DD process as part of the backup process, it can slow down the backup, and increase the backup window. In such situations, the backup may not be able to finish in time interfering with other processing which needs to happen, and potentially causing business loss. In these circumstances, the DD process will need to be able to scale horizontally, parallelizing the processing to reduce the overhead.

2.2 Research Goal

The focus of this research is to find a scalable architecture for DD of NoSQL DBs backup.

Goal 1 – adapt the file and sub-file base DD approaches to the NoSQL DB DD.

Goal 2 – explore the use of structural information about the data and its granularity to reduce the uncontrolled duplicate data in NoSQL DBs.

Goal 3 – develop a scalable architecture for the DD tool to minimize the time of the processing.

CHAPTER 3
LITERATURE REVIEW

The work in this research combines ideas from different industry and research fields: cloud computing, NoSQL DBs, and DD methods. Combing these areas the research tries to minimize storage challenges in the new emerging NoSQL DBs by using existing DD techniques at a different level.

3.1 Cloud Computing

Cloud Computing has generally been defined either by what it is considered to be made of (components), by its purpose, and sometimes using a combination of the two. Looking at what it is made of, Pinase et al. [42] define Cloud Computing (CC) as a distributed logical entity with managed computing resources deployed in big data centers around the globe and connected using public networks, like the Internet. As for the purpose of such entity, Maia et al. [35] define CC as a service with remote access to hardware and software in a highly reliable and transparent way like the electrical network. The increase in popularity of concepts like Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), presented by Luis et al. [51] was the next step in the evolution of CC. This challenged the Web 2.0 companies which were already facing huge data and infrastructures growths. Companies like Google (through Google App Engine – GAE [25]) and Amazon (through Amazon Elastic Compute Cloud - EC2 [1]) provide users with hardware resources, computational

resources, and software resources which have properties like elasticity, availability, and cost effectiveness in mind.

Elasticity is required for flexibility in scaling these systems easily, allowing applications with fast growth like the social networking applications to embrace it. Availability was a must when one thinks about computing as a service provided to millions of users and/or businesses for which downtime may mean revenue loss. And last but not least, cost effectiveness comes into play. It has to be worth paying for these services instead of owning your own.

In summary, CC is a new computing paradigm providing elasticity, availability, and cost effectiveness. These are important infrastructure characteristics to efficiently run a scalable Data De-duplication in NoSQL Databases (DDNSDB).

With the emergence of CC and its inherent properties, other areas have been challenged as well to meet these requirements, and that is the DB management Systems [47].

3.2 Databases

DBs at high level can be split into two categories: relational DBs and distributed DBs providing alternatives on architecture and management systems, depending on the type of data one needs to store and manipulate.

3.2.1 Relational DBs

The relational data model is based on the mathematical concept of a relation, which in this case is the notion of table. In a relational model, the data is stored in tables with columns and rows which imply a rigorous structure. The relational model is very

popular because it maps very well to a large variety of real-world data storage needs from the organization of information point of view. They fit best the structured type of data. Relational DBs also follow the ACID (Atomicity, Consistency, Isolation, and Durability) properties for transactions with which one can achieve extensive power, flexibility and reliability [7]. In 1983, Harder and Reuter [27] created the acronym ACID to describe them. In order for a transaction to achieve indivisibility it has to have the ACID properties: Atomicity (all-or-nothing), Consistency (only valid data will be written to the database), Isolation (events within a transaction must be hidden from other transactions running concurrently), and Durability (ability to recover the committed transactions against any kind of system failure) [27].

Normalization – is the process of organizing data to minimize redundancy in the relational DB world. The concept of normalization and what we know now as the First Normal Form (1NF) was introduced by Edgar F. Codd, the inventor of the relational model. Today there are six normal forms defined but generally, a relational DB table is often described as "normalized" if it is in the Third Normal Form [16]. Normalization involves dividing large, badly-formed tables into smaller, well-formed tables and defining relationship between them. This information about table's structures and their relations is called metadata (or data about the data). Depending on the degree of normalization, we have more or less information about the DB structure.

However, some modeling disciplines such as the dimensional modeling approach to data warehouse design, explicitly recommend non-normalized designs. The purpose of such systems is to be intuitive and have high-performance retrieval of data [32].

3.2.2 NoSQL DBs

NoSQL DBs use a similar but more extreme approach in their design. These DBs have a simple data model - “large, badly-formed tables” - for the purpose of having dynamic control over data layout and form, and high-performance retrieval against very large amounts of data. At the same time, they tend to have extensive amounts of duplicated data. While there was no reason to do DD at the DBs level for relational DBs, it makes a lot of sense to do DD at the DB level for NoSQL DBs.

The concepts behind non-relational DBs and the DBs themselves like hierarchical, graph, and object oriented have been around for more than 20 years. One common characteristic of these DBs is that they are not relational and they are used best for unstructured and semi structured data or data that changes form and size often.

These DBs do not have a unified Standard Query Language (SQL), instead they use their own APIs, libraries, and preferred languages to interact with the data they contain, hence the name Not Only SQL (NoSQL) DBs.

In pursuing the need for high availability and abundance of data which needs to scale horizontally across multiple nodes, old concepts emerged into these new Data Store (DS) technologies.

Some of the features of NoSQL DBs including seemingly large scalability (dynamic growth with no downtime), extensive fault tolerance and high availability (through partitioning and replication of data across nodes, and dynamically repair node failures capabilities), and integration of REST-ful and cloud computing technologies (web standards, ability to syndicate information directly to/from web sites and, replicate data directly to/from other DBs) are built in from the beginning [47].

3.2.2.1 MapReduce

MapReduce - is a very successful programming model adopted for implementation of data-intensive applications to support distributed computing. Jeffrey et al. [17] introduces MapReduce as a master-slave model. The failure of a slave is managed by re-assigning its task to another slave, while master failures are not managed as considered unlikely to happen. Users specify a map and a reduce function. The map function processes key/value pairs and generates a set of intermediate key/value pairs. The reduce function merges all intermediate values associated with the same intermediate key and produces a result as a list of values [17]. The main advantage of MapReduce is that it allows for distributed processing of the map and reduces operations. All map processes can potentially perform in parallel and all reduce processes can potentially perform in parallel; provide that their operations is independent of the others. Figure 3.1 illustrates the execution phases in a generic MapReduce programming model.

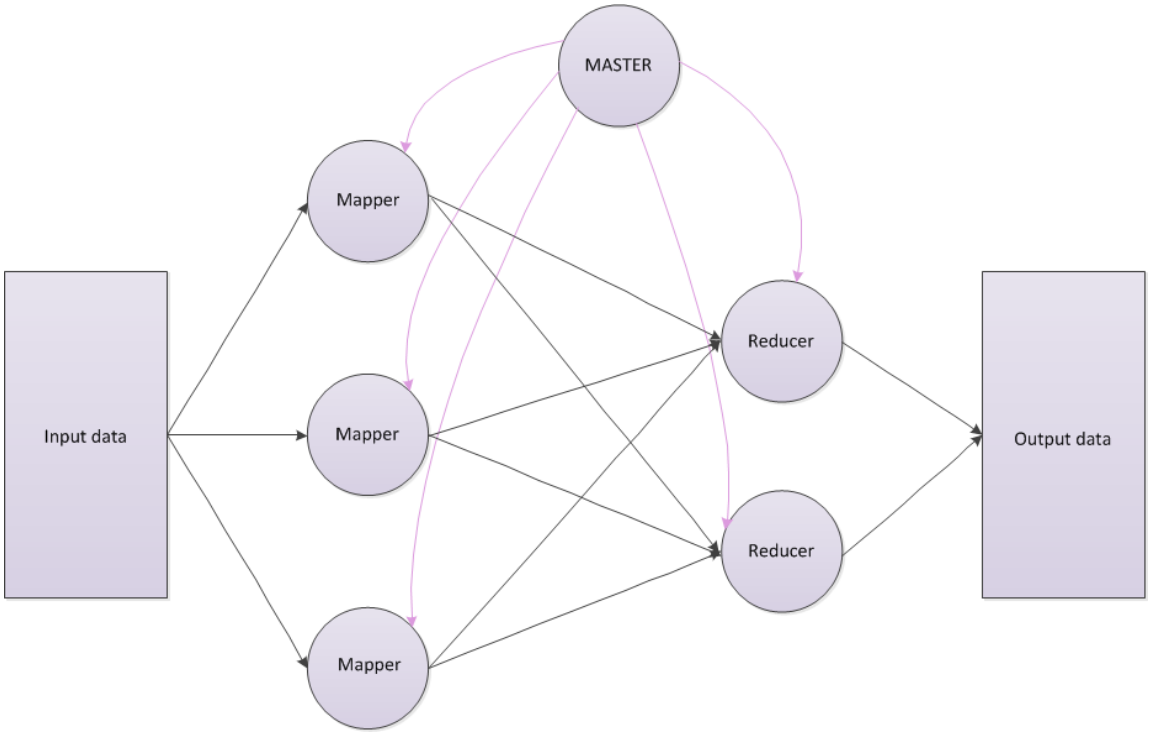

Figure 3.1: Generic MapReduce execution phases

The current market of NoSQL is a *"hodge-podge"* of vendors and open source projects with different levels of maturity. CC prompted some companies like Amazon and Google to develop new management systems with elasticity, availability, and cost effectiveness as core features. These companies impelled other companies and the open source world into the same direction. Some of these systems which are today available are Google Bigtable [12], Amazon's Dynamo [18], MongoDB [38], and others.

There are two aspects that need to be taken into consideration when looking at the NoSQL DBs, namely the CAP theorem and the data model of the different NoSQL DBs.

Following the CAP theorem (also called Brewer's Theorem), which states that *"in a distributed environment it is impossible to achieve all three properties: Consistency,*

Availability, and Partition tolerance" [24] different NoSQL DBs focus on different properties. Some DBs focus on Consistency and Availability. Consistency here is implemented with the *"eventual consistency"* [52] concept which is based on the idea that *"every change will be propagated to the entire DB eventually but some nodes may not have the latest data at a given time"* [7]. Some DBs focus on the Availability and Partition tolerance compromising on Consistency. They converge mainly to provide low latency and high throughput. Some DBs are in between the traditional RDBMS and NoSQL focusing on Consistency and Availability. They provide data consistency guarantees by supporting some types of transactions. Das et al. [15] proposes Elastrans where transactions are allowed but only at partition level. A second solution proposed by Francisco M. et al. [35] builds on top of the former, expanding the consistency to a group of partitions by introduction of a new layer of replication which also ensures higher availability.

Based on the data model, the different NoSQL DBs can be organized in the following categories: Key-value DS, Tabular/Columnar DS, Document DS [47; 34], graph DBs, object DBs, XML DBs, multi-value DBs, and other NoSQL related DBs [20]. The most popular are the first three categories.

3.2.2.2 Key-Value DBs

Key-value DBs have the least complex structure out of the NoSQL DBs. They store values indexed for retrieval by programmer-defined keys, and can hold structured and unstructured data. Some are built to run in-memory, some write to disk, and some do both to provide high-performance, scalable, and reliable DS. They have the flexibility to add new attributes that only apply to certain records at any point in time, without having

to rebuild tables or indices. Some of them follow the immediate or strong consistency model; others follow the eventually consistent model. The access is done through APIs (SOAP, REST-ful) and integrity is guaranteed by the application itself.

Key	Value
K1	{name => 'alfred', age => '32', sex => 'male'}
K2	{name => 'bob', age => '22'}
K3	{name => 'mary', age => '28', nickname => 'maria'}
K4	{name => 'dag', age => '45'}
K5	{name => 'Lille', sex => 'female'}

Figure 3.2: Representation of key-value store with arbitrary data (no schema)

Some of today's more popular key-value stores are: Amazon's SimpleDB [2] which is mostly used for small projects due to limitations (10 GB per domain, 100 domains per account, 256 attribute name-value pairs per item, manual partitioning), Oracle's Berkeley DB [40] which now provides SQLIite-compatible SQL APIs, Scalaris [48] which offers multiple concurrent transactions across multiple keys, and Project Voldemort [44] a mature project and open source version of Amazon Dynamo [18] supporting versioning and eventual consistency.

3.2.2.3 Column-Based DBs

Tabular or Columnar DBs are based on the concept of grouping closely related data into one extendable column [34]. In particular, they offer advantages to compute aggregate values on a limited number of columns. They emerged as implementations designed to meet certain needs (e.g. small footprint, highly compressible distribution of data or spare matrix emulation) rather than provide a general purpose column-oriented DBs. Like any new technology, they evolved to become more mature products. Google's BigTable model represented in figure 3.3 [12] was used for most DS in this class. BigTable can be described as a *"...distributed storage system organized as a sparse, multi-dimensional sorted map"* [12]. Logically, data is organized in tables with rows and columns. The tables are indexed based on a row key, a column key, and a timestamp:

(row: string, column: string, time:int64) -> string.

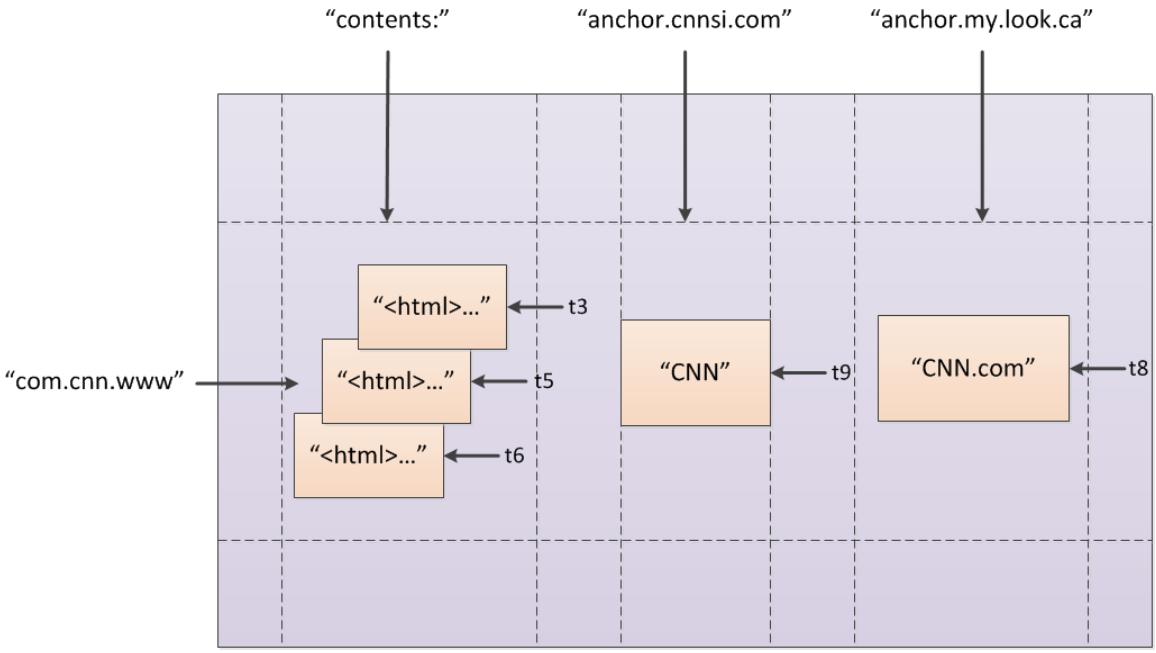

Figure 3.3: Google's BigTable structure, used to store Web pages

In figure 3.3 a row is a reserved URL where "*contents:*" is a column family to store versions of the page content and *"anchor:"* is another column family represented here by two names to store the text of the anchors which reference the page.

Partitioning is dynamic at the row range level. Data is stored in lexicographic order based on the row key. The rows of one table can have an arbitrary number of columns. Columns keys are grouped into column families ("following syntax: family: qualifier") in order to store data of the same type together. Multiple versions of the same data can reside in the same BigTable cell, each versioned with a timestamp.

BigTable is a more mature proprietary DB that uses compression at the SSTable (Sorted Strings Table which is a file of key/value pairs sorted by keys) block level, yet not all columnar DS do that. The compression can be set by the clients and is usually a two pass compression. The first pass uses a long common string technique called also the Bentley and McIlroy scheme [8] and the second pass is a fast compression algorithm based on repetitions of small blocks (16 KB window of data). This scheme achieves a significant 10-to-1 reduction of space. This low level block compression is fast and avoids the decompression of the entire file when reading one small portion of an SSTable [12]. By introducing DD at the database level as well, the footprint of the data can be further reduced. Compared with the key-value data model, the columnar data model has by default some structural information available, like the column families where each cell can have multiple version of the same piece of data. This also gives valuable information where the potential duplicate data is located.

Some of the columnar DSs are: Google BigTable [12] used for many of Google's applications like Google Maps and Google's search engine; Facebook which created the high-performance Cassandra [4] which uses a gossip protocol to easily scale (the nodes in a cluster are aware of the state of each node), it also provides durability (writes once completed will survive permanently) by appending writes to a commit log first then, is fsync'd; Apache's Hbase [6] is a distributed versioned store following Google's Bigtable capabilities on top of their Hadoop Distributed File System (HDFS) [3]. HDFS is a distributed file system which operates on common hardware for a low cost implementation solution. The tables in Hbase can be used as the input and output for MapReduce jobs which run in Hadoop.

3.2.2.4 Document-Based DBs

Document–based DBs store and organize complex documents/objects which commonly refer to data items. The documents are indexed providing efficient queries, mostly rely on a new principle called BASE (Basically Available – appears to work all the time; Soft state – it doesn't have to be consistent all the time; Eventual consistent – at some stage it will reach consistency) which trades some amount of consistency for availability. While ACID is pessimistic and forces consistency for all operations, BASE has an optimistic view and assumes that inconsistent operation will occur but will reach a consistent state at some point. Document-base stores support multiple types of documents and multiple indices per DB. The fact that they support multiple indices is the main difference between document and key-value DS. They also provide flexibility to add any numbers of fields of any length to any document at any time. This also means that some

structural information is available at the database level represented by the document attributes.

In CouchDB, the documents are the primary unit of data and consist of any number of fields and attachments. Metadata is also associated with the documents. The uniquely named document fields can contain values of varying types with no limit on text size or element count. JavaScript can be used for queries and indexes in a MapReduce fashion [5]. In MongoDB, the primary unit of data is an object; data is organized as one database collection for each top level objects; classes with embedded objects are used. The rule is that objects which follow an object modeling relationship should generally be embedded. There are limits on single object sizes [38].

In figure 3.3 there is a student collection where the student document embeds the address document and the score document.

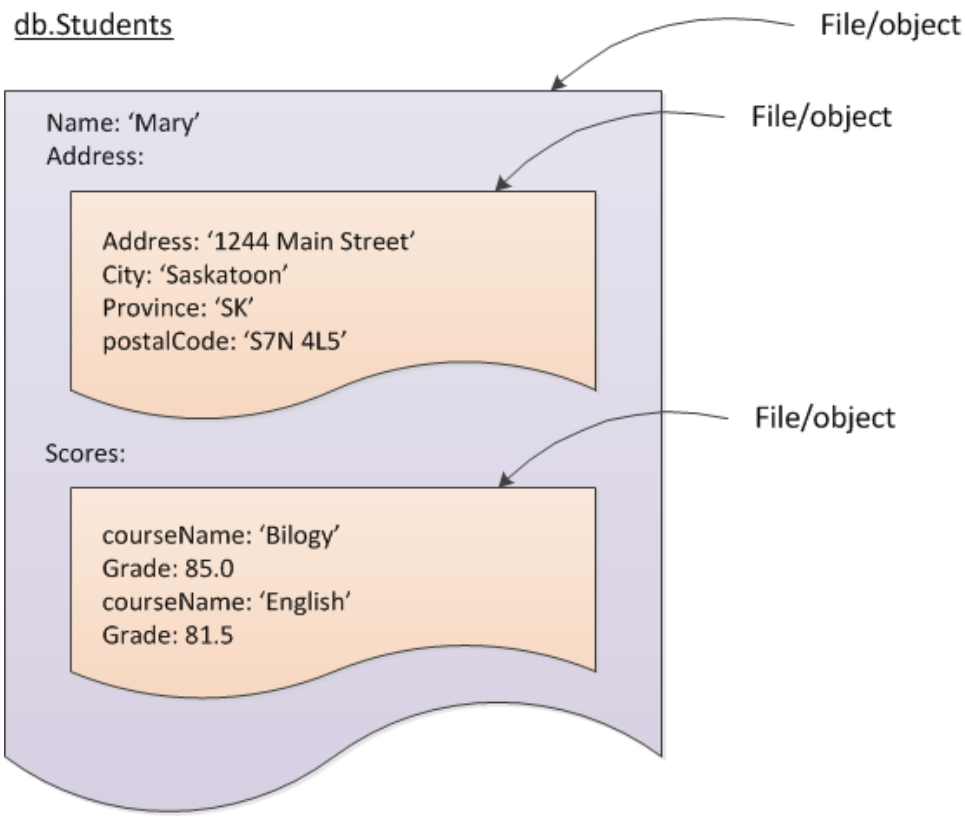

Figure 3.4: Structural representation of a Document DS for MongoDB.

Some of the more prominent projects are: MongoDB [38] – manages JSON (Java Script Object Notation) object collections with full index support, provides auto-partitioning and fault tolerance, uses sharding (a method of horizontal partitioning) and replication for distributed environments, and it also offers commercial support. CouchDB [5] – uses multi-master support for master-slave replication, works well in distributed environments, provides REST-ful (Representational State Transfer) HTTP APIs for reading and updating DB documents featuring ACID properties, and provides auto-partitioning and fault tolerance. Riak [46] – was inspired from Amazon's Dynamo,

provides capability for pluggable storage, and uses the eventual consistency concept but it is a less mature project compared to MongoDB and CouchDB.

In conclusion, there are three main types of NoSQL DBs Key-value, Columnar, and Document DBs used for managing unstructured and partially structured data, providing high scalability, availability, and fast retrieval requirements. One downside of these databases is that they tend to store large amounts of duplicate data which gets further propagated into the backup footprint.

3.3 Data De-Duplication

DD is *"…the process of identifying duplicates information using different methods and, eliminates them by applying pointers to those duplicates instead of storing the same data multiple times"* [23]. In the context of optimizing storage capacity, DD is one method of reducing storage consumption [23].

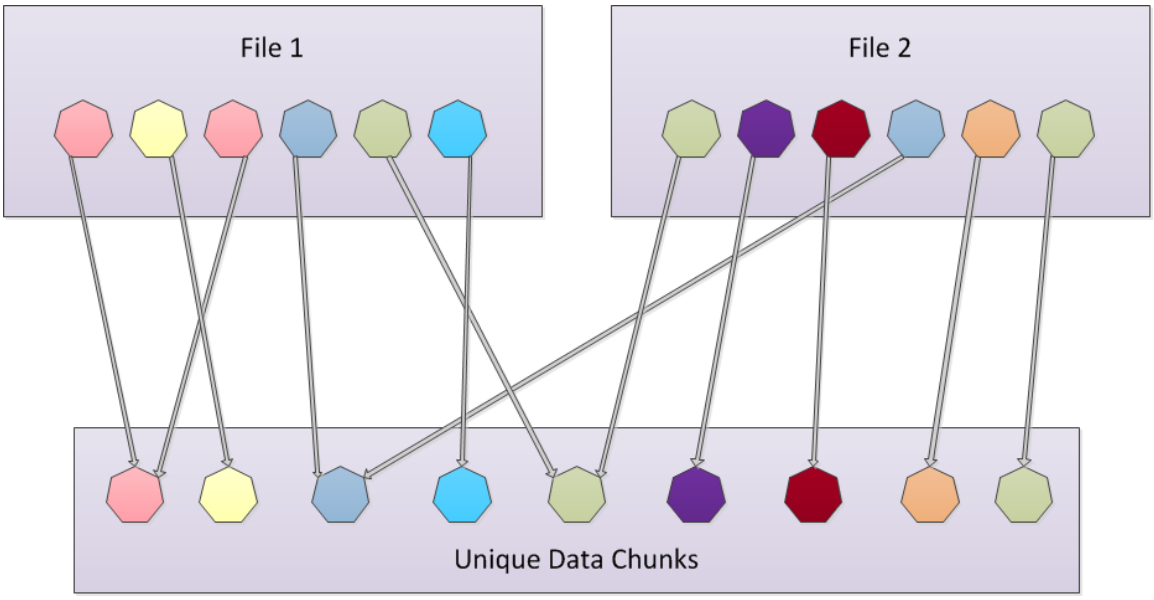

Figure 3.5: DD for two files split in chunks.

Figure 3.5 illustrates how the DD process will retain only one chunk of the same color, where the same color represents duplicate chunks.

Nagapramod at al. [37] developed a taxonomy to characterize and classify the different DD technologies available. They used three dimensions for their classification: the placement of the DD functionality, the timing, and the algorithm used, and created a comprehensive picture of the different aspects involved in the DD process. The choices of one dimension influence the choices of the other two dimensions. The three main DD algorithms presented are whole file hashing, sub file hashing, and delta encoding [37]. As their naming suggests, different types of hashing are used for a faster byte comparison. Sub file hashing has been further divided into fix sized blocks and variable sized blocks also called content defined blocks. Nagapramod at al. [37] experimented with different chunking techniques against real life data to investigate the DD inherent (changes of data where multiple backups were not taken into consideration) to conclude that no one algorithm can fit all.

One of the most popular algorithms to identify repetitions in strings is the Rabin Fingerprint algorithm. [45] It has been used to create the content-defined chunks of identical data or to discover near-duplicate documents in large collection of files [33; 36].

Policroniades at al. [43] evaluates three alternatives of identifying identical portions of data (whole file content, fixed size blocks, and Rabin fingerprints) against five different types of real-world data sets of different sizes. Some data sets were more prone to have a high level of duplicate data, some less. For the whole file and fixed size blocks granularity to identify the sharing patterns, they calculated the SHA-1 [50] digest

of the individual files, and correspondingly of each non-overlapping fixed size chunks of the files. For the third method of variable size chunks, they used Rabin's fingerprinting which has the advantage that the chunks are created according to their contents. They conclude that the content defined chunking algorithm identifies the most redundant data, but when file access patterns, overhead storage, and computation are considered, the fixed block size strategy may be a better solution.

Different types of semantic information about the data have been also used to increase the percentage of duplicate data detection and narrow down the search space to reduce the total disk access.

Yujuan et al. [53] experimented with one type of semantic information, namely the data ownership and built a three layered DD approach which includes user level, group-level, and global-level DD. The system makes use of data stream locality, Bloom filters [10], and hash chunking.

Chuanyi et al. [13] experimented with two types of semantic information, file type and file format to direct the file chunking along with Rabin fingerprinting. They define these types of chunks as "variable sized, self-identifying, and self-describing logical units". The files are divided into representative semantic chunks, implementing different file dividing algorithms for different file types. Because the semantic chunks have different length and variable size, they also implemented a storage scheme to alleviate the fragmentation and random disk access problems. Their results show a range between 20% and 50% of better compression ratio than the current conventional methods used in the archival storage.

Other hybrid DD approaches have been explored as well where two or more different algorithms are combined for enhancing the data DD ratio.

Guanlin et al. [26] proposes a two-step process which uses first the more common content defined chunk algorithm as a more coarse-grained chunking mechanism. It divides the files into content defined chunks and removes the duplicates identified. Second it applies a more finer-grain chunking mechanism using resemblance detection to perform delta-encoding and remove the duplicates detected. For the rest of the chucks, compression is applied.

Different chunking algorithms at file and sub-file level have been extensively explored, proving to make a big difference in the duplicate elimination process. The main steps followed by these algorithms are as follow:

- Each chunk of data is processed using hash algorithms like MD5, SHA-1 etc.. A unique number for each chunk gets generated with this algorithm which is then stored in an index.
- When duplicate data is detected, by comparing the hash number generated for a chunk of data with the ones existing in the index, it is not retained; instead a *"data pointer"* is modified so that the system references an exact copy of the data object already stored on the disk.

A potential problem with DD is hash collisions. In very rare cases, the hash algorithm may produce the same hash number for a different chunk of data. This is also called *"false positive"*, and can result in data loss. A solution to avoid hash collision would be to combine different hash algorithms. Another solution would be to examine the metadata to identify data and prevent collision [9].

Collisions tend to happen more often when we deal with big chunks of data like an entire file. By using smaller chunks the probability is lower. In the context of this research, due to the need to split the attributes - *"big unstructured data"* – in smaller

chunks using the metadata, the probability of hash collision will be very low therefore it will not be taken into consideration.

In relational DBs, duplicate data was very minimal or almost nonexistent but with the increased interest in NoSQL DBs in the cloud environment, there is a new need to exploit possibilities of identifying duplicates within the DBs. While it made no sense to do DD at the databases for highly normalized data, it makes a lot of sense to approach it for NoSQL DBs which have highly de-normalized data, and where structural information may have a key role.

3.4 Database Backup

DB backups can have different purposes: to recover data after its loss (deleted, corrupted), and to recover data from an earlier time. Data loss is a very common experience but at the same time can be catastrophic if there is no way of getting it back. DBs can store sensitively personal and financial information and institutions, companies, and enterprises make sure that they have an option to recover lost data.

There are two main types of DB backups: consistent backups also called *"cold backups"* and inconsistent backups also called *"hot backups"*. Consistent backups have the advantage that they take less time to perform and the DBs can be consistently recovered to the time of backup. This requires that the DB has to be down and most companies cannot tolerate such downtime windows. The alternative is the inconsistent backup. A backup that is made when the DBs is open, is inconsistent. When a DB is restored from an inconsistent backup, media recovery is required before the DBs can be opened. Any pending changes which were committed but did not have a chance to be written to the data files are applied. Usually, there are some requirements that need to be

met by the DBs to be able to perform inconsistent backups which more or less consist of generation transaction logs.

"Hot backups" can also be of multiple types depending on the needs. They can be full backups (all the data is backed up), incremental backups (based on a full backup; only the changed blocks since the last full or incremental are backed up leading to smaller backups and backups window), and cumulative backups (based on a full backup; only the changed blocks since the last full backup are backed up; used to reduce the recovery time since only two backups are required to be restored, the last full and the last cumulative) [41].

The performance of backup tools is generally higher than using manual backups as they have in-depth knowledge of the format of data blocks, the order in which blocks will be read to be able to capture a known good checkpoint for the file, etc. [41]. The backups have usually a proprietary format which can only be read through those specific tools, and there is not much information available for the researchers as to how internally the backups are performed. Due to these restrictions, this research is using the manual backup option; hence the focus is on DD ratio and performance of the DD process through horizontal scaling.

In the context of backup, DD can occur at the source or target. Source DD is reducing the size of backup data at the client (e.g. exchange, file server, DB server) so that only unique data is sent across the local wide area network during the backup process. In these situations, the DD technology is embedded in the backup application. Target DD is reducing the size of backup after it crosses the local area network when it reaches a DD storage system. Each has its own advantages and disadvantages and

depends on the needs. To take advantage of the metadata form within the databases, the source DD is the choice for DDNSDB which besides the fact that it reduces the amount of data backed up, it also helps to optimize network bandwidth.

3.5 Conclusion

Although still maturing, the different types of NoSQL DBs are becoming more popular in the context of CC and web programming. When dealing with new needs of storing and retrieving large amounts of data, NoSQL DBs tend to become the choice. However, their highly de-normalized structures retain a lot of duplicate data. Because ultimately data is represented into a file, the current research in DD focuses mainly on algorithms implemented at file and sub-file level to help reduce the data footprint. Because of the dependencies between the placement of the DD process, timing of DD, and algorithm used to find and reduce redundancies in the data there is no one solution which fits all. It depends, and generally it depends on the type of data. For NoSQL DBs, the current DD algorithms can be brought at a different level where additional information about the data can be made available to help find and reduce the duplicate data in a highly efficient and scalable fashion.

Table 3-1 is listing and summarizing the reviewed literature grouped by the different area of interest for this research.

Area	Papers	Notes
Cloud concepts	Pinase et al. [42]	Cloud computing components.
	Maia et al. [35]	Cloud computing purpose.
	Luis et al. [51]	Infrastructure as a resource, Platform as a resource and Software as resources
	Mathew D. [47]	Effects of Cloud computing in the DBMS world
NoSQL databases	Mathew D. [47]	NoSQL databases general characteristics and classification
	Daniel B. [7]	Comparison between RDBMS and NoSQL database
	Werner V. [52]	Eventually consistent
	Das et al. [15]	Elastras - NoSQL database with minimal transaction support
	Seth et al. [24]	CAP theorem
	Francisco M. et al. [35]	Higher level of transaction support for Elastras
	Neal L. [34]	NoSQL database characterization and classification
	Stefan E. [20]	NoSQL database extensive classification
	Fay et al. [12]	Google BigTable details
	[2, 40, 4, 5, 6, 3, 44, 46, 48]	Dedicated web sites for various NoSQL database with details about structure and functionality
Data De-duplication	David G. [23]	Data De-duplication in storage systems
	Nagapramod et al. [37]	Taxonomy of data de-duplication technologies
	Rabin M. [54]	Rabin fingerprint algorithm
	Purushottam et al. [33]	De-duplication at file level to discover near-duplicate documents in large collection of files
	Udi M [36]	
	Policroiades et al. [43]	Evaluations of three de-duplication techniques against real-world data sets
	Yujuan et al [53]	Use of one type of semantic information in detecting duplicate data
	Chuanyi et al. [13]	Use of multiple types of semantic information in data de-duplication
	Guanlin et al. [26]	Hybrid de-duplication approaches combining different de-duplication algorithms.
	Stephen et al. [9]	Detailed description of how chunk comparison is done using hash values and the hash collision problems.

	[21, 29, 30, 39, 28, 22]	Implementation of data de-duplication in the industry.
Database Backup	[41] Theo et al. [27]	RMAN backup concepts in Oracle RDBMS. Transaction oriented database recovery principals.
MapReduce	Jeffrey et al. [17]	MapReduce framework, components and functionality.

Table 3.1: List of research solutions by area

In summary, the existing researches show the following:

- CC, through properties like elasticity, reliability, and cost-effectiveness, provides a scalable platform for running distributed process applications.
- MapReduce is a successful programming model used to support distributed computing.
- All the different types of NoSQL DBs compared with the RDBMS have a highly de-normalized data structure therefor they store large amounts of duplicate data.
- DD concepts focus only on the file and sub-file level algorithms.
- Hash collision can lead to data loss, but in very rare circumstances, mostly when big chunks of data are dealt with.
- DD has been implemented for backups both as synchronous and asynchronous processes.

However, there are still open questions namely:

- Can one DD approach easily fit different types of NoSQL DBs?
- How to use the metadata in the NoSQL DBs to identify duplicate data?
- How to improve the DD ratio in NoSQL DBs?
- How to design a DD process to scale horizontally in a distributed environment using the MapReduce principles?

CHAPTER 4
DE-DUPLICATION APROACH

This section describes the data model of the three main types of NoSQL DBs, and why the key-value DB was chosen as the representative structure for the DD process. Then it presents the role of the NoSQL DBs metadata in identifying duplicate data, and how it can improve the DD ratio. The last part presents a high level architectural design for implementing DD in a key-value data store using a programming model called MapReduce [17] which supports distributed computing.

4.1 NoSQL DBs Data Model

Tables are one of the most commonly used conventions throughout many disciplines to organize and represent data, and for DBs, it is *"the format"*. NoSQL DBs are not considerable different from this perspective than relational DBs. Depending on what they are designed for, the difference is in their physical layout. Row oriented representations store row values together while Column oriented representations store column values together. Their logical representations can still be as a table with column and rows, although one cell may be a very complex object with its own structure as well. In their logical representation, based on what was the need for which they were designed, NoSQL DBs have one or more columns, sometimes more complex grouping of columns, etc..

4.1.1 Key-value DBs

Key-value DBs have a very simple data model and store their data by row. Entries are stored as key-value pairs in large hash tables. The data domains (possible values of an attribute) are similar to relational DBs tables but no specific schema is defined. Keys are arbitrary while values are big large objects. There are no explicit relationships between data domains. In consequence, to have this level of flexibility which is reflected as no structural information about the data in the DB itself requires that the applications have that knowledge. Implicitly, the lack of structure at the DB level results in lots of duplicate data.

4.1.2 Column-Based DBs

Column-based DBs store their data by column. This allows for big performance uplift when you need to query many rows for smaller sets of data (not all columns). There are three critical concepts which describe the column oriented DBs:

- Column Family - logically can be represented as a table because it requires predefining column names from the beginning. Each column family is stored in a separate file with the purpose of grouping data of the same type together.
- Super Column - is a dictionary; it is a column that contains other columns (but not other super columns).
- Column - is a tuple of name, value, and timestamp.

The entries are indexed based on a row key and the data is stored based on the sort order of the column family. The sort order, unlike in a relational DB, isn't affected by the columns values but by the column names. Considering a twitter model as an example, figure 4.1 illustrates three column families which can be defined: Users (a), Tweets (b), and Users Tweets (c) as a super column family.

Users

Key	"@nid"	
Columns	Location	Canada
	Name	Nicoleta Brad
	Profession	DBA

(a)

Tweets

Key	Tweets/000000000001	
Data	Application	TweetDeck
	Private	TRUE
	Text	"Hi there!"
Key	Tweets/000000000002	
Data	Application	TweetDeck
	Private	TRUE
	Text	"Hello World!"!"
	Version	1.2

(b)

UserTweets

Key	"@nid"		
Data			
	Timeline	Timeline/000000000003	Tweets/000000000001
		Timeline/000000000004	Tweets/000000000002

(c)

Figure 4.1: Twitter model in a column oriented structure

Based on the representation form figure 4.1, each column family or super column can logically be represented as a table with multiple columns or as a more flattened version as a key-value table. The tuple structures would look like this:

Users:

{{key,@nid},{columns,{location, Canada},{name, Nicoleta Brad},{profession, DBA}}}.

Tweets

[

{{Key,Tweets/000000000001},{Data,{Application, TweetDeck},{Private, true},{Text, Hi there!}}},

{{Key,Tweets/000000000002},{Data,{Application, TweetDeck},{Private, true},{Text, “Hello World!”}, {Version, 1.2}}}

].

UserTweets:

{{Key,@nid},{Timeline,{Timeline/000000000003,Tweets/000000000001},

{Timeline/000000000004, Tweets/000000000002}}}.

The attributes in a key-value DB can incorporate multiple column complexities therefore allowing for situations presented in figure 4.1 (b). In conclusion, the Columnar DBs can ultimately be represented as a set of key-value structure if needed.

4.1.3 Document DBs

Document based DBs are at their core, Key-value DBs, where each record is stored as a document/object (e.g. JSON) and can be identified by a unique ID. The objects consist of named fields which can be strings, numbers, dates, or more complex structures like associative maps, ordered lists, etc.. The benefits of document based DBs over the key-value DBs are that they allow multiple indices based on the uniquely name fields available, and offer additional query capabilities. They were designed to better deal with larger objects where key-value DBs were designed to deal with primarily smaller objects. There is no big difference in the data-model itself but in the way the data is manipulated for their corresponding needs.

In conclusion, the document DBs can ultimately be represented as sets of key-value structures as well, if needed.

Columnar and document DBs are logically built with certain features in mind to manipulate data in different ways and serve different needs. At the same time, their structure can be flattened to the key-value structure. Hence, with minimal adaptation, key-value DD concept could be easily converted to fit the other main types of NoSQL DBs, namely columnar and document DBs. The analysis of the data-model of the three main types of NoSQL DBs answers one of the open questions presented at the end of Chapter 3 namely "*Can one DD approach easily fit different types of NoSQL DBs?*".

4.2 NoSQL DBs Metadata

Today, chunking based data DD is the dominant technology to reduce the space requirements for both primary file systems and data backups. This technology approaches DD at file and sub-file level. There are two steps involved in this DD technology: chunking - splitting the data into non-overlapping data blocks – and duplicate detection – each chunk is compared with all other stored chunks to detect if they have the exact same content. Various elaborate chunking techniques have been developed to better identify duplicate data, and applied against data files with reasonable results. In the DB context, making use of information about the data structure (metadata) corresponds to the "chunking" technique.

Chunking - the key-value data store is very similar to a file system structure. The key is the name/inode of the file and the value is the content of the file. The chunking mechanism proposed in this research for the key-value store is also similar with the chunking mechanisms used for files. For example, some of the files chunking algorithms

use the semantic information about the file (e.g file structure) to obtain *"semantically meaningful data chunks"* [13]. An email file consists of several semantic segments: sender, attachment, receiver, and so on embraced by tags. Using the semantic information, the file is chunked in variable sized chunks using the tags as delimiters. This results in identifying larger duplicate chunks for a better DD ratio and less burden on the management of future file retrieval. The metadata in the key-value store is used in a similar manner. A row in a key-value store has metadata associated with the value (e.g position and/or identifier - 3/application, 5 /text). Using the metadata information, data is chunked in variable sized chunks, using the position/identifier as delimiters.

```
(
        {key, Tweets/000000000001},
        {data,
                [{application, TweetDeck},
                {private, true},
                {text, Hi there!}]
        }
).
```

Duplicate detection – stands for finding the duplicate values and replacing them with pointers. The duplicate detection step can be split into two sub steps: (1) computing a hash for each data chunk; (2) comparing the hash values of data chunk to detect duplicates. In this research, the same steps are used for detecting the duplicates for a key-value store.

To avoid confusions between the file chunks used as a unit of comparison in the file/sub-file DD process, this research will refer to the unit of comparison for key-value store as *"data set"*. For example the tuple {f,g} can be a data set, meaning that the data stored in that tuple position and format will be compared across table rows.

An important aspect for comparison accuracy is selecting a collision resistant hash function. This makes the probability of two different inputs to produce the same output so low that it is practical to assume that each chunk has a unique hash value. This research is using MD5 (Message-Digest Algorithm), a widely used cryptographic hash function that produces a 128-bit (16-byte) hash value. For the purpose of this research, the risk of losing data has no major consequences, and this algorithm was considered acceptable.

In conclusion, it is a natural/easy way to apply the existing DD steps used at file and sub-file level for the DB level as well, and use the metadata for chunking and defining how the duplicate data search should happen efficiently. This answers another open question presented in Chapter 3, namely *"How to use the metadata in the NoSQL DBs to identify duplicate data?"*.

4.3 De-duplication Ratio

As explained by Duch [19], DD ratio or *"Space reduction Ratio"* is represented by the fraction:

$$Ratio = \frac{\text{Bytes In}}{\text{Bytes Out}}$$

This represents the number of bytes input to a data DD process divided by the number of bytes output, and are typically described as *"ratio:X"*. For example, if 100 GB of data consumes 10 GB of storage capacity, the space reduction ratio is 10:1 [19].

An intuitive assumption is that the more information about the data structure there is, the smaller the unit of comparison can be used implying a higher potential of identifying duplicate data. NoSQL DBs have very limited structure for various legitimate reasons. The lack of structural information at the DB level is compensated by incorporating it into the applications. Depending on the application needs, this information can have different granularity, varying form having no information at all to having all the information. There are different ways that this metadata could be collected to be made available to the DD process e.g: impose some level of structure automatically, ask for information about the structure at the DB creation time, collect information about the potential structure after the DB creation and presented to the user next time for confirmation, etc.. This research will assume that this information is available and is provided to the DD process through a configuration file.

4.4 Data DD with MapReduce

The architecture of the DDNSDB shares most of the design principals of MapReduce. It follows the master-slave design where the master node is responsible for managing the jobs, i.e., start the worker nodes, and assign the map/reduce tasks. Each worker can run a map or a reduce task at any given time. The job execution begins by splitting the input data and assigning it to individual map tasks. When a worker finishes executing a map task, it stores the map results as intermediary key-value tables in memory. The intermediaries results of each map task are assigned to the existing reduce

workers. A reduce task begins by retrieving its corresponding intermediary results from all map outputs and then it apply the reduce function.

Figure 4.2 illustrates the DDNSDB execution phases following the MapReduce master-slave model.

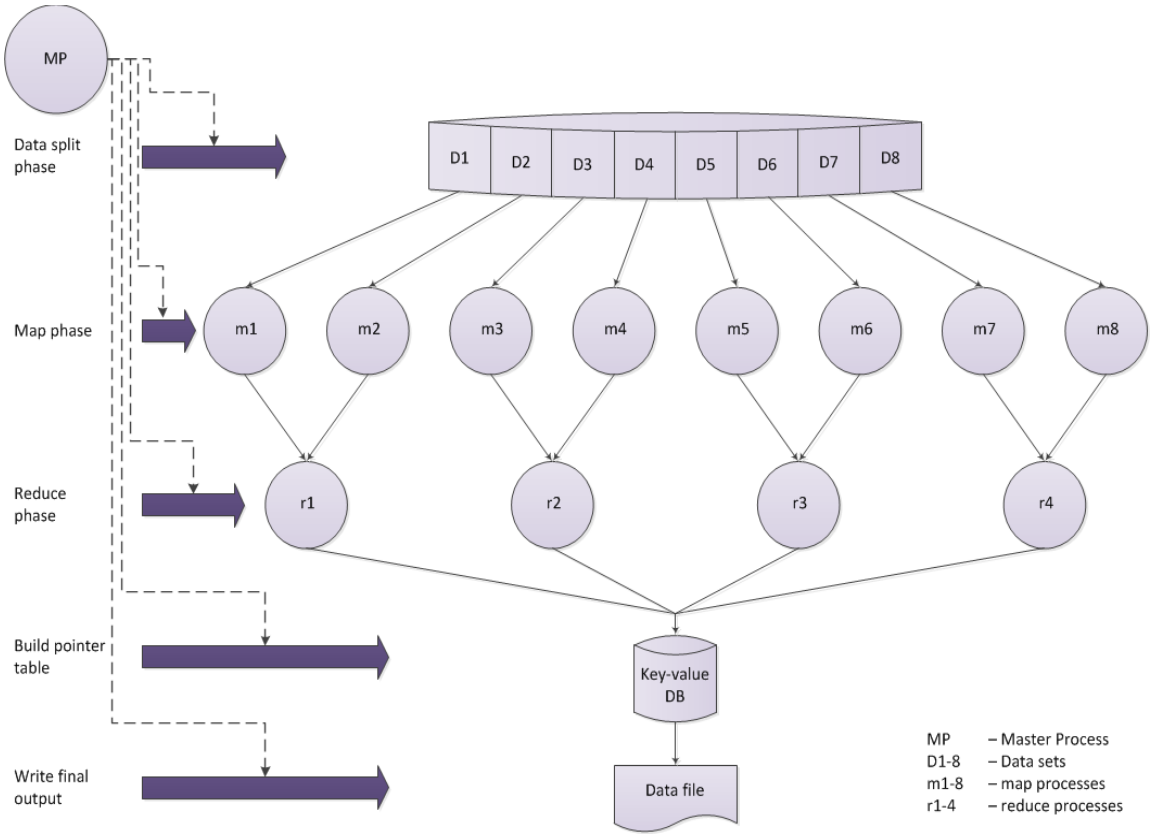

Figure 4.2: Overview of the execution phases

For the purpose of improving the DD ratio, the architecture leverages a two layer hierarchical reduction. Conceptually, the map and reduce tasks are organized as a tree where each level waits for all the tasks at the previous level to finish, before the work in the next level begins. Figure 4.3 outlines the map and the two reduce layers architecture.

The last layer of DD is executed by one reducer worker, which aggregates all the partially reduced results and creates a single output table with the data references resulted from the DD process.

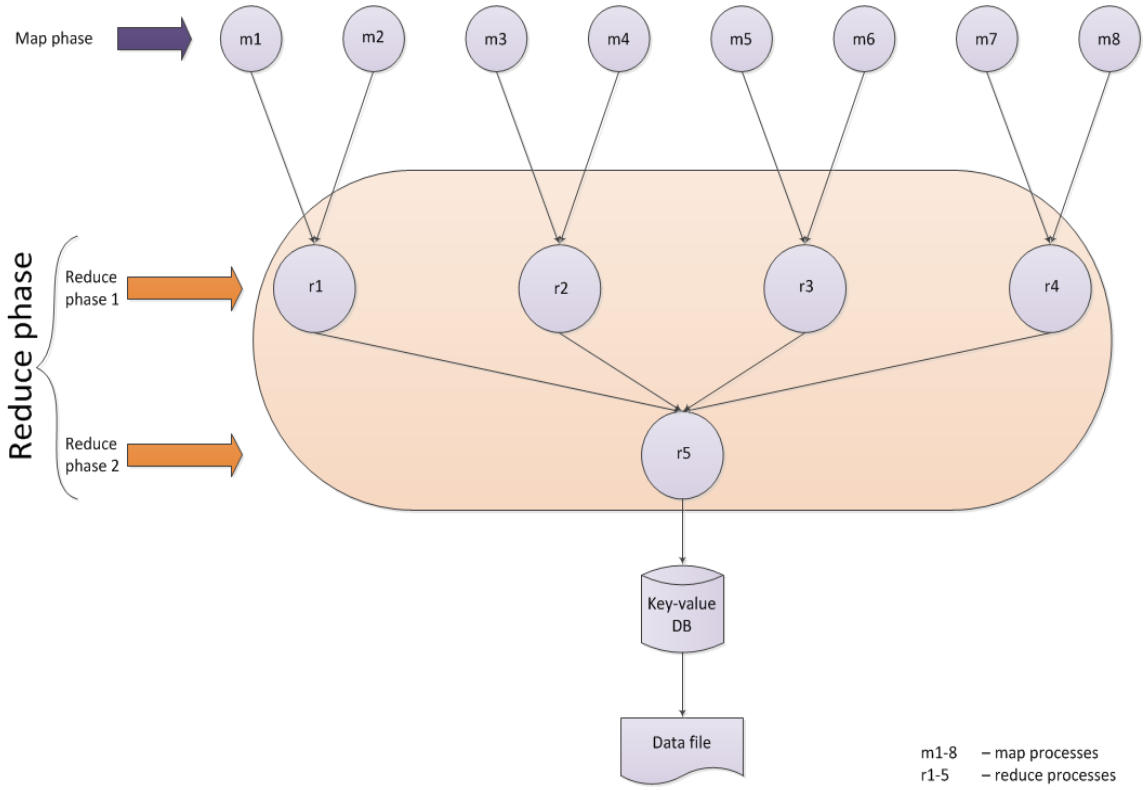

Figure 4.3: Hierarchical reduction

Traditionally, the map function processes a key-value pair and returns a list of intermediate key-value pairs: map (k1, v1) -> list (k2, v2). The reduce function merges all intermediate values having the same intermediate key: reduce (k2, list (v2)) -> list (v3).

In this implementation, to increase the efficiency of the DD process instead of materializing the intermediate key-value pairs within every map task, the data is kept in

memory and then directly pipelined to the reduce tasks. The same principal is applied between the two layers of reduce tasks. Moreover, between the separate reduce layers the data is not transformed into key-value pairs but pipelined raw as is manipulated by the reduce function. This way 6 extra back and forth transformation and materializations steps are eliminated for each pair of data-sets.

In conclusion, the MapReduce framework has been adapted to the DD process of key-value DB to optimize the processing and increase the DD ratio. The metadata is the key element used for chunking the records into semantically aware data-sets, and to better identify duplicates by comparing only identical data-set types. There is a tight relationship between the degree of normalization and the amount of duplicate data. In the key-value store, it can be said that data is completely de-normalized.

This analysis demonstrates **Goal 1** of this research, namely how we *"adapt the file and sub-file base DD approaches to the NoSQL DB DD"*. Subsequently this is implemented into DDNSDB.

CHAPTER 5
IMPLEMENTATION

This chapter presents the prototype implementation of a DD process in the key-value DB, the representative NoSQL DB, using a MapReduce approach.

The implementation of this tool uses Erlang (http://www.erlang.org), a functional, concurrent programming language, and development platform. It was chosen because of its easy to implement scalability and the built in key-value store. For concurrency, Erlang uses light-weight "processes" and message passing, and comes with its own in memory key-value store called ETS, and the more permanent storage version of a key-value store called DETS.

Execution overview – depending on the amount of data which needs to be processed, the required number of worker processes can be passed in as parameters along with the data file. The number of mapping worker processes "*M"* is passed separately from the number of reducing worker processes "*R"*. There is no relationship between the number of mapping and reducer processes. The input data is read sequentially and partitioned into a set of *"M"* data-sets.

The map function parses each record, and based on the metadata information produces a sequence of {key,value} pairs which is then stored into a set of in memory temporary tables. There is one temporary table for each data-set type defined in the metadata. For example, if the metadata available for a key-value store which stores FaceBook profile data has the following information: {{key,1}, {name,2}, {location,3},

{hometown,4}, {gender,5},{birthday,6},{languages,7},{picture,8}}…., the map function will split the record based on all attributes keeping their respective key as identifier, calculate the hash value, and insert it into ETS tables. Figure 5.1 illustrates how the structure of each table will look like. For the readability purpose, figure 5.1 (a) has the actual data represented, and figure 5.1 (b) has the calculated md5 hash values represented.

key	name
{key,1122334455}	Nicoleta Brad
{key,1122334466}	Marin Ioan
…	

key	location
{key,1122334455}	Canada
{key,1122334466}	Hungary
…	

key	hometown
{key,1122334455}	Sinaia
{key,1122334466}	Budapest
…	

key	birthday
{key,1122334455}	Nov-73
{key,1122334466}	May-71
…	

key	gender
{key,1122334455}	female
{key,1122334466}	male
…	

key	languages
{key,1122334455}	Romanian
{key,1122334466}	Hungarian
…	

key	picture
{key,1122334455}	file1.jpg
{key,1122334466}	file2.jpg
…	

Figure 5.1 (a): Metadata base records split without hashing

key	name
{key,1122334455}	<<221,238,54,78,206,129,0,15,141,17,231,162,53,224,219,199>>
{key,1122334466}	<<149,212,119,25,58,101,235,249,78,178,213,16,83,179,49,202>>
…	

key	location
{key,1122334455}	<<68,93,51,123,92,213,222,71,111,153,51,61,246,176,194,167>>
{key,1122334466}	<<250,121,195,0,93,174,196,126,207,248,74,17,106,9,39,161>>
…	

key	hometown
{key,1122334455}	<<199,207,37,28,91,129,33,45,243,189,145,125,37,136,138,90>>
{key,1122334466}	<<159,237,93,174,134,227,3,13,155,227,116,14,250,25,134,89>>
…	

key	birthday
{key,1122334455}	<<136,217,34,219,72,62,103,129,133,112,38,97,127,180,3,162>>
{key,1122334466}	<<109,218,83,98,221,166,98,106,96,86,181,42,168,31,210,88>>
…	

key	gender
{key,1122334455}	<<39,59,154,229,53,222,83,57,156,134,169,184,49,72,168,237>>
{key,1122334466}	<<7,207,79,143,93,139,118,40,41,23,50,7,21,221,162,173>>
…	

key	languages
{key,1122334455}	<<239,167,57,78,202,167,252,112,118,169,218,19,167,114,54,184>>
{key,1122334466}	<<123,134,17,46,198,64,31,216,240,106,181,37,29,26,104,254>>
…	

key	picture
{key,1122334455}	<<4,145,255,80,239,251,236,55,146,9,31,74,124,215,130,251…..>>
{key,1122334466}	<<110,215,162,212,19,36,108,200,247,194,188,127,110,70,120,84….>>
…	

Figure 5.1 (b): Metadata base records split with hashing

Once the table traverse has finished, each mapping worker receives a message which indicates that the DD process can begin, and initiates the process by sending the data to the reducer worker processes. Each mapping process reads the data from its own temporary tables and sends it off as one message to one available reducer process. Once all the mapping workers have finished sending the data to the first layer of reducer workers, the reducer workers start the DD process. The reducer processes are the ones which perform the actual data DD. Figure 5.2 illustrates a streamlined graphical representation of how the data DD process works where the tuple complexity is three e.g. {name, birthday, avatar}. There are 4 mapping workers which create the data-sets based on the metadata, and 3 reducer processes, 2 on the first layer and one on the second layer.

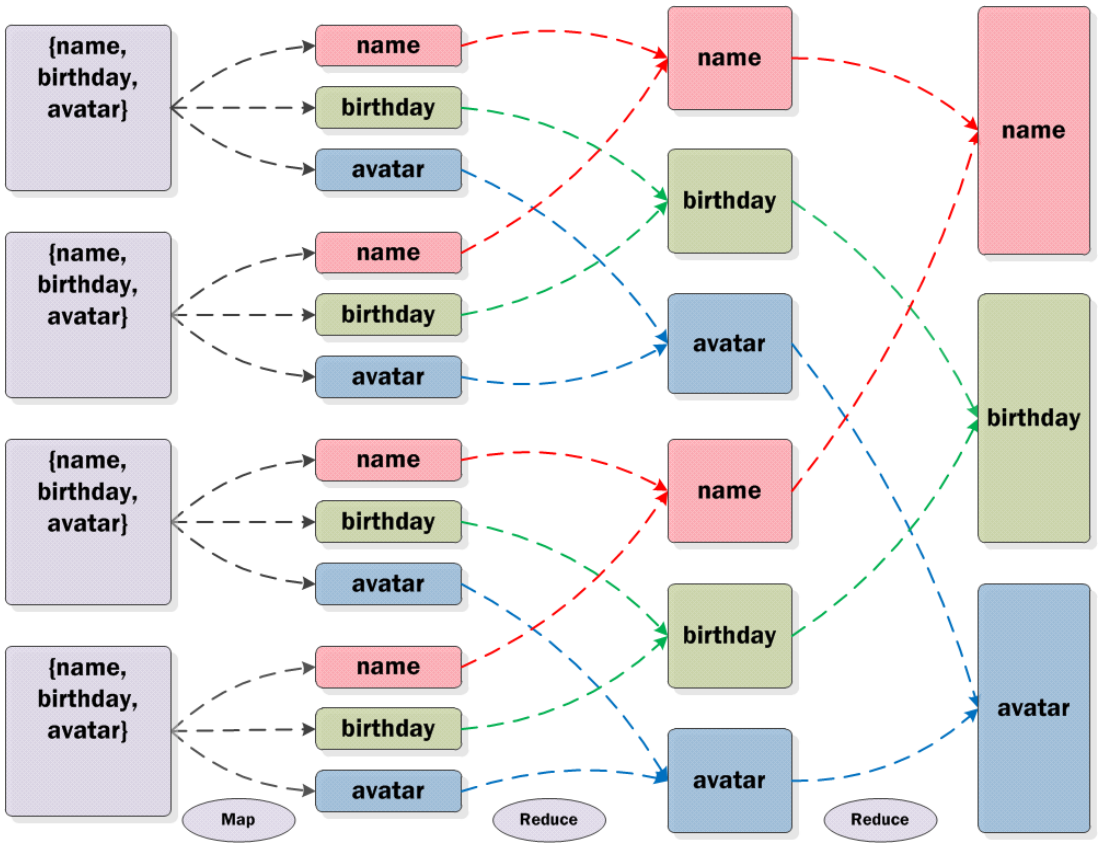

Figure 5.2: De-duplication steps.

Because the probability of finding duplicates is higher where data of the same type is compared, this implementation leverages this type of comparison algorithm. Not every data-set is compared with each other. Based on the metadata, only the data-sets of the same type are compared. For example: a tweeter name is compared with another tweeter name, a birthdate is compared with another birthdate, and an avatar is compared with another avatar. There is no comparison between a tweeter's name and a tweeter's birthdate or avatar.

The master process receives a message from the reducer workers when the DD processing is complete. Once the first layer of reducer workers is done, the next layer of reducer worker receives the intermediary data and proceeds with one more DD process. The intermediary data is then transformed back into key-value pairs and the pointer table is built. The pointer table will consist of the unique data and its corresponding pointers to the data.

The last step is writing the de-duplicated data into a backup file. This implementation uses the manual backup option by writing the data into a text file as a proof of concept; however the DD tool can be easily implemented into other more specialized DB backup tools as a pre-stage for reducing the amount of data to be backed up.

The main design considerations for DDNSDB are:

- Use the map/reduce technology for parallel processing and horizontal scaling.
- Use the metadata in the algorithm of generating semantic chunks.
- Use a hierarchical reducer process to obtain a higher space reduction.

- Compare only data-sets of the same type to minimize the time costs.
- Use calculated hash values in the comparison step to minimize the time cost rather than compare the data chunks byte to byte.

CHAPTER 6
EVALUATION

The evaluation approach of this research focuses on the following two aspects:

- **How much can the DDNSDB reduce storage space?** Because DD was never approached at the NoSQL DB level before, we compare DDNSDB with two other compression algorithms: Windows Zip, and UNIX GZip.
- **How to speed up the DD process?** Because backup generally has a limited time window, performing another high CPU process during this time can easily exceed this window. While the amount of data to be backed up is reduced by applying a DD process against the data before the backup, we can significantly decrease the time of the DD process through parallel processing.

Goal 1 of this research, namely *"Adapt the file and sub-file base DD approaches to the NoSQL DB DD"* is evaluated in Chapter 4 through an analysis process, and implemented for DDNSDB.

Goal 2 namely *"Explore the use of structural information and its granularity to reduce the uncontrolled duplicate data in NoSQL DBs"*, and Goal 3 namely *"Develop a scalable architecture for the DD tool to minimize processing time"* are assessed through the experiments of this research. They measure the degree of compression achieved in the key-value DB by the DDNSDB tool, the time involved in this process and the scalability of the process for potential improved performance and adaptability to the cloud environment.

A high level visual representation of the configurations to be used for the experiments is shown in figure 6.1 and consists of a table generator, different hardware architecture represented by a server with specifically large memory resources and reasonable processing power, and a set of smaller servers with an overall capacity similar to the first larger server. Subsequently, there is a result analyzer which will calculate the redundancy identified in the DB.

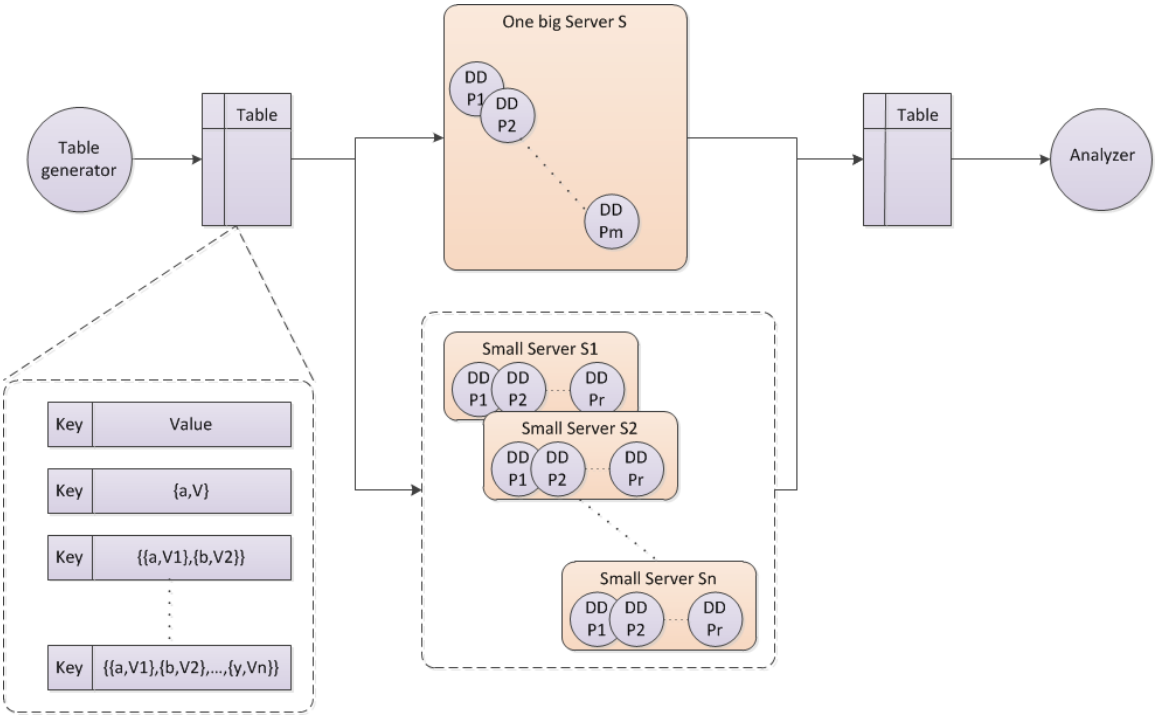

Figure 6.1: Experiments topologies.

6.1 Experiment Goals

The main goals of the experiments will be to find out how to speed up the de-duplication process and in the same time obtain an acceptable de-duplication ratio.

Goal 2 – explore the use of structural information about the data and its granularity to reduce the uncontrolled duplicate data in NoSQL DBs.

- Evaluate how different metadata configurations influence the DD ratio.
- Evaluate how these configurations will perform based on the size of the DB.
- Evaluate how these configurations will perform based on the percentage of redundant data.

Goal 3 – develop a scalable architecture for the DD tool to minimize the time of the processing.

- Evaluate how different number of map and reduce processes influences the time cost;
- Evaluate how different hardware configurations influences the time cost.

There are two sets of experiments to evaluate the two goals involving three architectural topologies. The first set of experiments is used to determine the DD Ratio to evaluate goal 2. The second set of experiments is used to evaluate the performance of the DD process and its scalability in the different architectural topologies. The experiments run in different configurations, and on different types of hardware. Testing was performed on the Amazon Elastic Cloud Computing (EC2) Platform where several hardware configurations are readily available at competitive prices. The experiments ran as batch process, and the following were considered:

- DD ratio;
- Time of DD;
- Network traffic;
- Load on the machine(s) – memory and CPU.

Goal 2

Topology 1: – The topology for these experiments will consist of the following:

- One big machine with extensive memory resources (64 GB RAM) appropriate for the DBs and DD processing which is running in memory;
- Four map processes and three reduce processes used in the configuration of the DD process.

Under this topology there are three factors considered for the experiments: metadata structure (tuple complexity), DB size, and distribution of redundant data in the DB. Table 6.1 shows what values were used for the metadata structure, DB size as number of rows, and the percentage of redundant data generated in the DB. Since there are no benchmarks available for the NoSQL DBs, and in the same time we wanted to have a representation of most popular types of data currently stored in NoSQL DBs (pictures and web pages), the number of rows per DB were selected based on two factors: the size of the objects loaded, and Erlang's limitations of a DB size. Based on the same reason, the values for the duplicate data distribution were selected based on the hypothesis that NoSQL DBs can have high percentages of duplicate data. Hence 80% was selected for the highest percentage, and the rest of the values we selected to keep the proportions comparable. The values for the tuple complexity were chosen based on the similarities with the DD process at file and sub-file. Also each algorithm comes with a relative probability of identifying duplicate data. For example, algorithms which compare entire files can be compared with DB records with no structural information. Algorithms which chunk the file in variable size chunks based on certain algorithms can be compared with a DB records for which we now all the structural information. Algorithms which chunk the files in fixed size chunks, based on the probability to identify duplicate data, can be compared with DB records for which only partial structural information is available.

Nr #	Metadata info
1	No info 0%
2	Some info 50%
3	All info 100%

Nr #	Nr. of rows
1	1,000
2	4,000
3	7,000
4	10,000

Nr#	Redundancy %
1	20%
2	40%
3	60%
4	80%

Table 6.1: Experiments variable and values.

Factor 1 – Structural information:

- How is the DD ratio influenced by the structural information?

Factor 2 – Amount of data:

- How is the DD ratio influenced by the amount of data?

Factor 3 – Distribution of duplicate data:

- How is the DD ratio influenced by the distribution of duplicate data?

Goal 3

Topology 1: - The topology for these experiments consists of the following:

- One big machine with extensive memory resources (64 GB RAM) appropriate for the DBs and DD processing which is running in memory;
- Four sets of numbers of map/reduce processes used in the configuration of the DD process (4/2, 8/4, 16/8, 32/16).

Under this topology, there is only one Erlang Virtual Machine (EVM). The scheduling between the threads and the parallel processes message queues is done internally by Erlang.

Factor 1 – Structural information:

- How is the DD time influenced by the structural information?

Factor 2 – Amount of data:

- How is the DD time influenced by the amount of data?

Factor 3 – Distribution of duplicate data:

- How is the DD time influenced by the distribution of duplicate data?

Factor 4 – Number of parallel workers for the map-reduce processes:

- How is the DD time influenced by the number of parallel processes? In this experiment the traffic between the message queues of the parallel processes is managed internally by Erlang.

Topology 2: – The topology for these experiments consists of the following:

- One big machine with extensive memory resources (64 GB RAM) appropriate for the DBs and DD processing which is running in memory;
- Four sets of numbers of EVM nodes for the map / reduce processes used in the configuration of the DD process (4/2, 8/4, 16/8, 32/16).

Under this topology each EVM node is independent of the others, competing potentially for the same CPU threads. There are no internal message queues scheduling between the nodes, thus everything is left at the operating system level. This configuration will be used along with the same three factors: metadata structure (tuple complexity), DB size, and percentage of redundant data in the DB. This will test the scalability of the DDNSDB and how the parallel EVM nodes influence its performance. The relationship between parallel processes and EVM nodes is 1 to 1. Each EVM node spawns *"n"* number of threads where *"n"* is equal with count of CPUs multiplied with number of threads per CPU.

Factor 1 – Number of workers for the map-reduce processes:

- How is the DD time influenced by the number of worker processes?

Topology 3: – The topology for these experiments will consist of the following:

- Four sets of machines, each set with an approximate overall capacity as the one big machines used in the previous two sets of experiments. The maximum number of machines is less or equal with the number of processes.
- Four sets of numbers of EVM nodes for the map / reduce used in the configuration of the DD process (4/2, 8/4, 16/8, 32/16).

  Topology 3.1: - First set has 4 XXL (extra-extra-large) machines.

  Topology 3.2: - Second set has 2 XL (extra-large) machines.

  Topology 3.3: - Third set has 4 L(large) machines.

  Topology 3.4: - Fourth set has 4 Mi (Micro) machines.

Under this topology a scaled out configuration made of multiple physical/virtual machines will be used along with the same three factors: metadata structure (tuple complexity), DB size, and percentage of redundant data in the DB. The same number of workers used in the previous two topologies will be used for map and reduce processes. This will test the scalability of DDNSDB and the performance influence due to network traffic across multiple commodity hardware machines, configuration specific for Cloud Computing.

Factor 1 – Number of physical/virtual machines:

- How is the DD time influenced by the network traffic across commodity hardware?

6.2 Experimental Setup

The two sets of experiments for this evaluation are performed on Amazon EC2 Platform. EC2 allows scalable deployment of applications by providing a Web service

through which a user can boot an Amazon Machine Image to create a Virtual Machine (VM) also called Instance, containing any software desired. Amazon EC2 is built on commodity hardware, consisting of several different types of physical hardware. It uses the Amazon EC2 Compute Unit (ECU) as a measure to rent compute power rather than a particular processor type. The amount of CPU that is allocated to a particular instance is expressed in terms of ECUs [1]. One ECU *"provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor"* [1]. The hardware configurations used for the experiments are presented in Table 6.2.

Experiment Dataset	**Nr. VMs**	**API name**	**RAM**	**ECUs**	**SMP**	**Platform**	**I/O Performance**
Set1 & Set2	1	M2.4 xlarge	68.4 GB	26 (8 virtual cores * 3.25 ECU each)	smp:8:8 ; rq:8	64-bit	High
Set2	4	M2.2 xlarge	34.2 GB	13 (4 virtual cores * 3.25 ECU each)	smp:4:4 ; rq:4	64-bit	High
Set2	2	M1.xl arge	15 GB	8 (4 virtual cores * 2 ECU each)	smp:4:4 ; rq:4	64-bit	High
Set2	4	M1.la rge	7.5 GB	4 (2 virtual core * 2 ECU each)	smp:2:2 ; rq:2	64-bit	High
Set2	4	T1.mi cro	613 MB	Up to 2 for short periodic bursts	rq:1	64-bit	Low

Table 6.2: EC2 VMs hardware configuration

The instance configurations presented in table 6.2 are part of the Amazon EC2 Standard, Micro, and High-Memory Instance types. SMP represents how many CPU threads Erlang detected, and how many parallel schedulers (generally one per thread) it created at startup.

6.3 Dataset

DD at the NoSQL DB level is a novel concept; therefor there is no research as how it will perform in a controlled or uncontrolled environment. For this reasons, rather than dealing with the uncertainty of the real life data in an uncontrolled environment, this research is generating data to be absolutely certain of its content and structure. This baseline can then be used for future research as a point of reference with not such in-depth knowledge of the data from real life environments.

There are two sets of data used in the experiments. The data was manually generated based on the three factors defined earlier: size, redundancy, and amount of structural information (tuple complexity) available resulting in 48 DB files presented in table 6.3, and table 6.4.

DB Structure	Col1	Col2	Col3
AM	file1.jpg	file2.jpg	file3.htm
SM	null	file1.jpg	file2.jpg || file3.htm
NM	null	null	file1.jpg || file2.jpg || file3.htm

Table 6.3: Tuple complexity

Nr. Rows - Redundancy %	All Metadata	Some Metadata	No Metadata
1000 - 20%	AM100020	SM100020	NM100020
1000 - 40%	AM100040	SM100040	NM100040
1000 - 60%	AM100060	SM100060	NM100060
1000 - 80%	AM100080	SM100080	NM100080
4000 - 20%	AM400020	SM400020	NM400020
4000 - 40%	AM400040	SM400040	NM400040
4000 - 60%	AM400060	SM400060	NM400060
4000 - 80%	AM400080	SM400080	NM400080
7000 - 20%	AM700020	SM700020	NM700020
7000 - 40%	AM700040	SM700040	NM700040
7000 - 60%	AM700060	SM700060	NM700060
7000 - 80%	AM700080	SM700080	NM700080
10000 - 20%	AM1000020	SM1000020	NM1000020
10000 - 40%	AM1000040	SM1000040	NM1000040
10000 - 60%	AM1000060	SM1000060	NM1000060
10000 - 80%	AM1000080	SM1000080	NM1000080

Table 6.4: DB Files structure

The source data is made from JPG and HTM files representing pictures and Facebook Blog page *"Searching for Answers? Ask Facebook Questions."* by Blake Ross. Pictures and web pages are the most common type of data stored in NoSQL DBs, because they generally require fast access and retrieval capabilities. Internally, they are represented as binary large objects (BLOBs) in the DB.

The three scenarios for the tuple complexity represented in table 6.3 stand for All Metadata (AM = {file1, file2, file3}), Some Metadata (SM = {null, file2, file2 ++ file3}), and No Metadata (NM = {null, null, file1 ++ file2 ++ file3}). The distribution of the redundant data is done differently for the two datasets. In the first dataset, the HTM file which is also smaller in size is unique for each row. The two JPEG files which are larger

in size stay the same to generate the redundant data percentage required. Subsequently they become unique for the rows with random data (e.g. the two JPG files size represent 86% of a row size; 20% of redundant data in a 1000 rows table is represented by 229 rows; 229 rows in the DB have the same file1 and file2, everything else is randomized). The same data is initially distributed over the three elements in the tuple presented in table 6.3, than over two elements, and lastly over one element by concatenating the data from the files.

In the second dataset the redundant/duplicate data is generated using identical rows (all the files stay the same), and the random data is applied to all the elements of the tuple. (e.g. 20% of redundant data in a 1000 rows table is represented by 200 identical rows; 200 rows of the DB have the same file1, file2, and file3). The same data is initially distributed over the three elements in the tuple presented in table 6.3, than over two elements, and lastly over one element by concatenating the data from the files.

Based on the three parameters (tuple complexity, amount of data, and distribution of redundant data) 48 different DETS DB tables were loaded, generating an approximate of 36.7 GB of data. There are some size restrictions of DB files in Erlang DETS, which have prevented us at the moment to experiment with larger sizes of tables. The datasets used are still generous enough to represent the characteristics of data in a NoSQL key-value DB, and to prove the concepts behind DDNSDB.

The experiments were run using these files in an ordered manner. The DB files order is represented in Figure 6.4 and shows how the data was split in three sets. The first set consists of the files with AM, the second set consists of the files with SM, and the third set consists of the files with NM. Within each group the files were also sorted base

on the other two parameters descending, first on size and second on distribution of redundant data.

The two data sets are meant to show how different structural information applied to the same set of data can influence the DD Ratio, and how different percentage of duplicate data can influence the DD Ratio and the performance of DDNSDB.

-

CHAPTER 7
RESULTS

7.1 De-Duplication Ratio

A consistent backup of the DBs was performed before and after applying the DD process to record the size of the files. Similarly, the size of the backup files was measured after running the experiments in the first and second set three times. The results remained the same all across. This confirms the accuracy of the DDNSDB and the correctness of the generated data used in the experiments. At the end, the DDNSDB results are compared with the results of two traditional compression methods: Windows Zip and UNIX GZip.

7.1.1 DD Ratio and Structural Information

To determine how the structural information influences the DD ratio, two scenarios with two sets of data are considered in this experiment. The first scenario uses the dataset one where the duplicate data is spread across only two components of the tuple. The data generated with different percentage of redundancy is applied to the three different structures presented in figure 6.3. The results of the DD-Ratios calculated for the three types of file namely, All Metadata, Some Metadata, No Metadata (AM, SM, NM) are described in figure 7.1 and figure 7.2. The DB files are grouped by structural information. Figure 7.1 illustrates the average DD Ratio of all files grouped by structural information and figure 7.2 illustrates the DD ratio for each file.

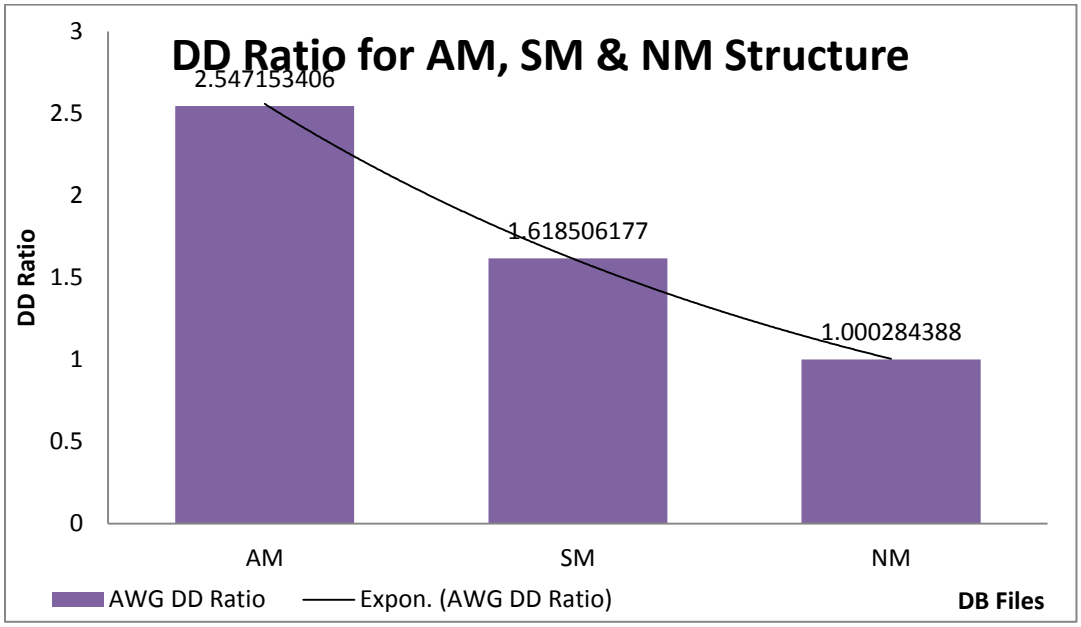

Figure 7.1: Average DD Ratio for the three different DB structure

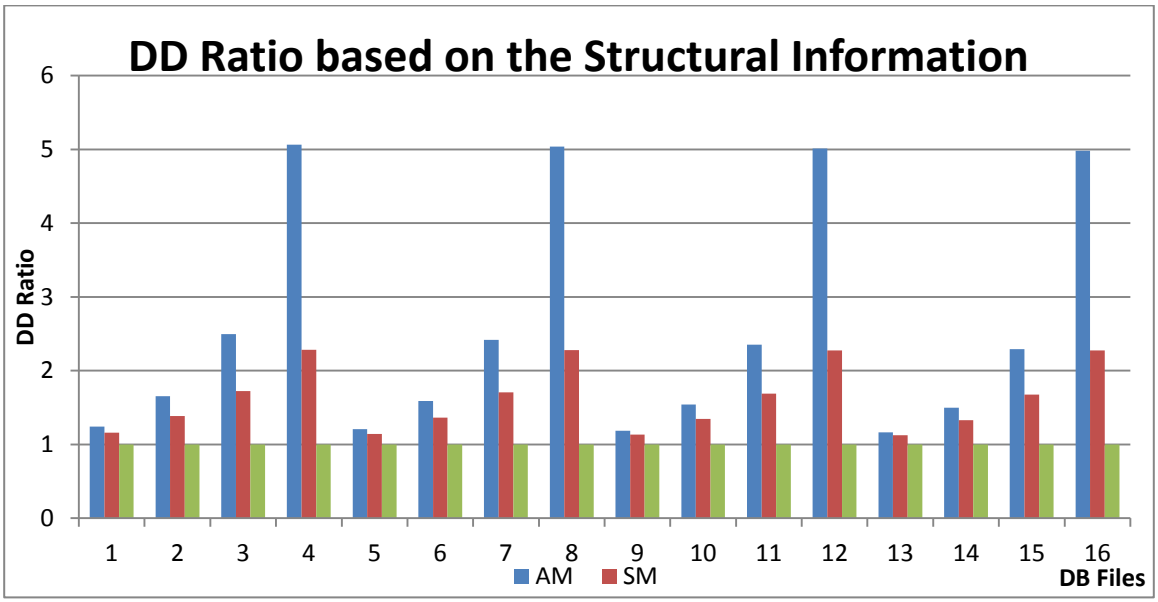

Figure 7.2: DD Ratio based on structural information

There are different levels of granularity for the structural information, starting from zero to knowing everything. The more granular usually means a higher probability of identifying duplicate data. For the DB files with AM, all duplicate data is identified and replaced with pointers by the DDNSDB. For the DB tables with NM, no duplicate data is identified. The results show that the DD ratio for the AM files is significantly higher than for NM files. The average DD Ratio for AM is 1.6 higher than SM and 2.5 higher than NM. This difference in DD Ratio is caused by changing only one character in the HTML file in each DB record.

The second scenario is using dataset two where the duplicate data is spread across all the elements of the tuple creating duplicate rows. By applying this data to the three different structures presented in figure 6.3, the results will show how different sizes of chunks are affecting the DD ratio. In the second set of data the AM structure represents the smallest chunk sizes, the SM structure represents the medium size chunks, and the NM structure represents the largest size chunks. The same measurements were made for the data generated in the second set of experiments. The results of the DD-Ratios calculated for the three types of chunk size (AM-small, SM-medium, and NM-large) are described in figure 7.3 and figure 7.4. The DB files in figure 7.3 are grouped by chunk size showing the average DD Ratio per group. Figure 7.4 shows the DD ratio for each individual file.

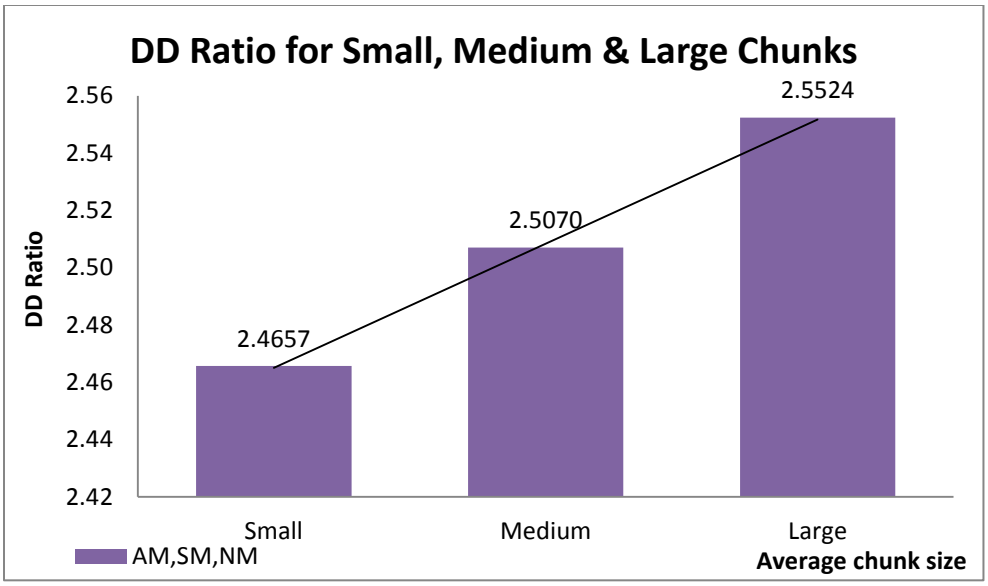

Figure 7.3: Average DD Ratio for the three different data chunks.

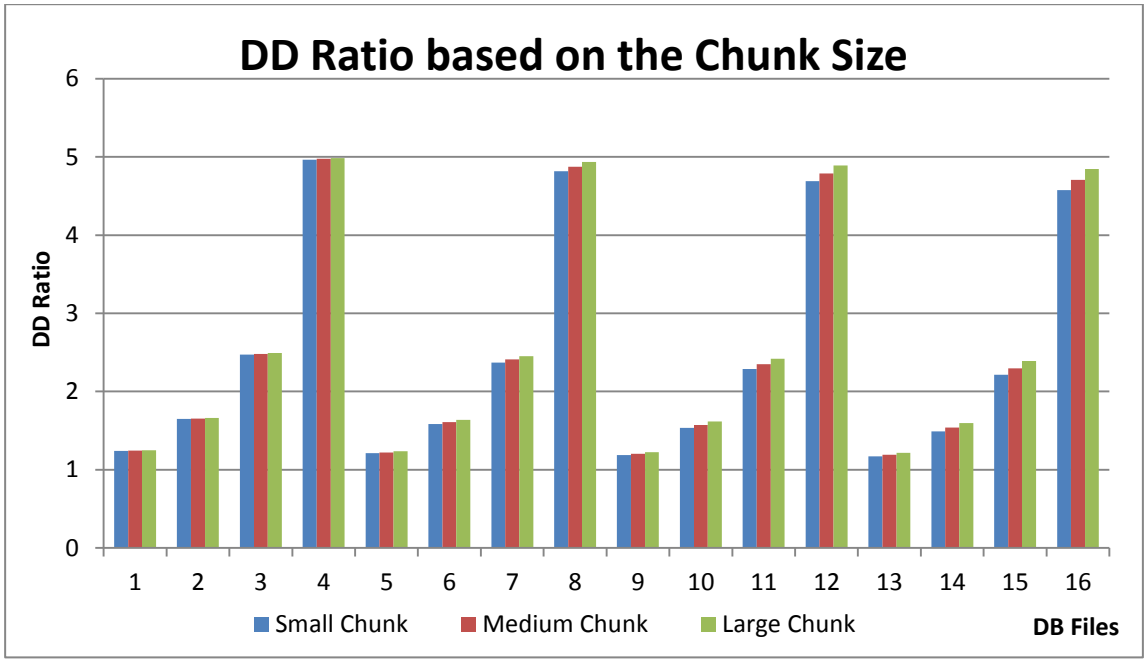

Figure 7.4: DD Ratio based on Chunk size.

Figure 7.3 and 7.4 show that the DD ratio for the files which have bigger duplicate chunks (NM files) is higher than for the ones which have smaller duplicate chunks (AM files). Figure 7.3 also shows that the sizes of the AM files are larger than the sizes of the SM and NM files. The generated data in the second set carries as the *"null"* atom the missing structure in its formatting, keeping pointers in each structural unit. This extra formatting becomes significant when identical rows are involved with different structural information.

Based on the results from the two sets of data, it is observed that with more structural information, a higher DD ratio is obtained. Also, the probability of finding the duplicate data is higher, even if small changes of data occur to some of the fields. If there is no structural information, the smallest change to a row in the table makes it unique, increasing the DB backup footprint. On the other hand, a higher DD ratio is obtained with fewer larger chunks than with more but smaller chunks. The extra formatting for the pointers affects the backup space. However, the results based on chunk sizes, show much smaller differences between the DD ratios than the results based on tuple complexity proportion-wise.

7.1.2 DD Ratio and the Amount of Data

The next parameter taken into consideration in this set of experiments is the amount of data. The DB files have been loaded with four different numbers of rows to see how this parameter is affecting the DD ratio. Table 7.1 and figure 7.5 illustrate the results. The numbers on the x-axis correspond to the number of rows in DB files (from 1000 rows to 10000 rows). The DB files are grouped based on the number of rows and an

average file size for each group was calculated before and after applying the DD process. The average DD ratios were calculated based on the above average DB file sizes.

File Type	AWG File Size	AWG DDNSDB Size	DD Ratio
1000 rows	87165309	62615109	1.7513
4000 rows	364807390	266684146	1.7295
7000 rows	664868022	493178704	1.7116
10000 rows	987516163	742254833	1.6956

Table 7.1: Backup Files and DD Ratio

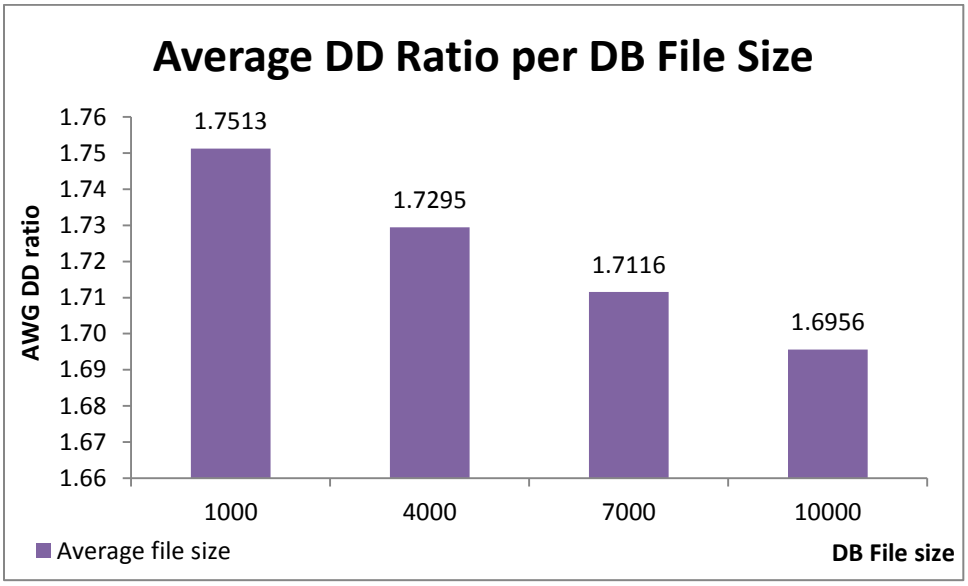

Figure 7.5: Average DD Ratio based on DB file sizes

The results show that the size of the DB files impacts the DD ratio by decreasing it slightly by approximately 3%. This is probably caused by the additional formatting of the data in the backup file. The more data there is, the more formatting information is

stored. The overhead of 3% decrease in the DD ratio while the file size increases 10 times is considered minimal. If the same percentage of duplicate data is present regardless of the size of the DB, we can say that the DD ratio stays the same.

7.1.3 Distribution of Duplicate Data

The third factor taken into consideration in this set of experiments is the distribution of redundant data in the DB files. Four different percentages of redundant data were chosen in the dataset generation: 20%, 40%, 60%, and 80%. The files for this experiment have been grouped based on the distribution of redundant data and size as illustrated in table 7.2. (e.g. the value for 100020 represents the average DD ratio calculated for all the files with 1000 rows and 20% of redundant data). Each peak in graph from figure 7.3 corresponds to the last file in each set of files from table 7.2.

100020	1.1339
100040	1.3473
100060	1.7406
100080	2.7833

400020	1.1177
400040	1.3182
400060	1.7085
400080	1.7735

700020	1.1060
700040	1.2958
700060	1.6816
700080	2.7630

1000020	1.0965
1000040	1.2765
1000060	1.6596
1000080	2.7527

Table 7.2: DD Ratio based on distribution of redundant data

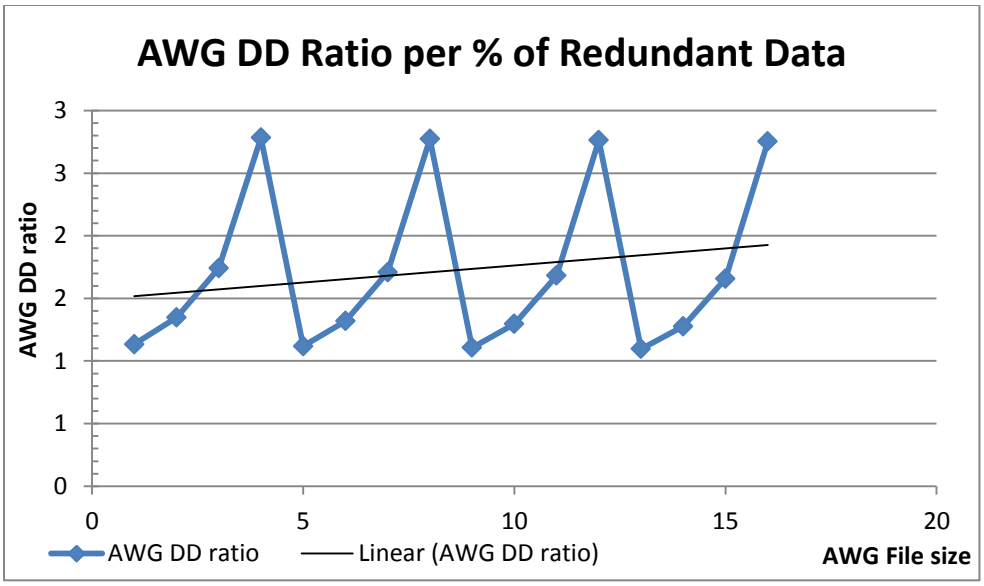

Figure 7.6: DD Ration based on distribution of redundant data

The results show that the DD Ratio grows almost exponentially while the redundant data for each file increases by 20%. The pattern shows the spikes in each DB file group; in this case there are four groups as represented in table 7.2. At the same time, the DD Ratio has an overall slight increase as the amount of data grows. The linear trend line in figure 7.6 highlights this slight growth. The distribution of redundant data has a substantial impact on the DD ratios, making it grow exponentially. Achieving much larger DD Ratios in the 60% to 80% range of redundant data, brings a considerable saving in the backup file storage. For example, if the initial size of the DB backup file was 82 MB, the size of the de-duplicated DB backup file is 33 MB. This reads that with a redundancy of 60% in the DB file, we obtain a 60% smaller file after applying the DDNSDB. This is the case when all the structural information is available.

7.1.4 DD Ratio Comparison

DDNSDB algorithm was applied to the entire data set along with Window Zip and UNIX GZip to better understand the differences between the results. The DB files are grouped first by structural information then by size, and lastly by distribution of redundant data. Figure 7.7 (a,b,c) illustrates the differences between the initial DB file size and the file sizes after applying the three different algorithms, and figure 7.8 (a,b,c) shows the comparison results of DD ratio for the three different algorithms. The intervals on the x-axis correspond to each file in their respective group of DB files (AM, SM, and NM).

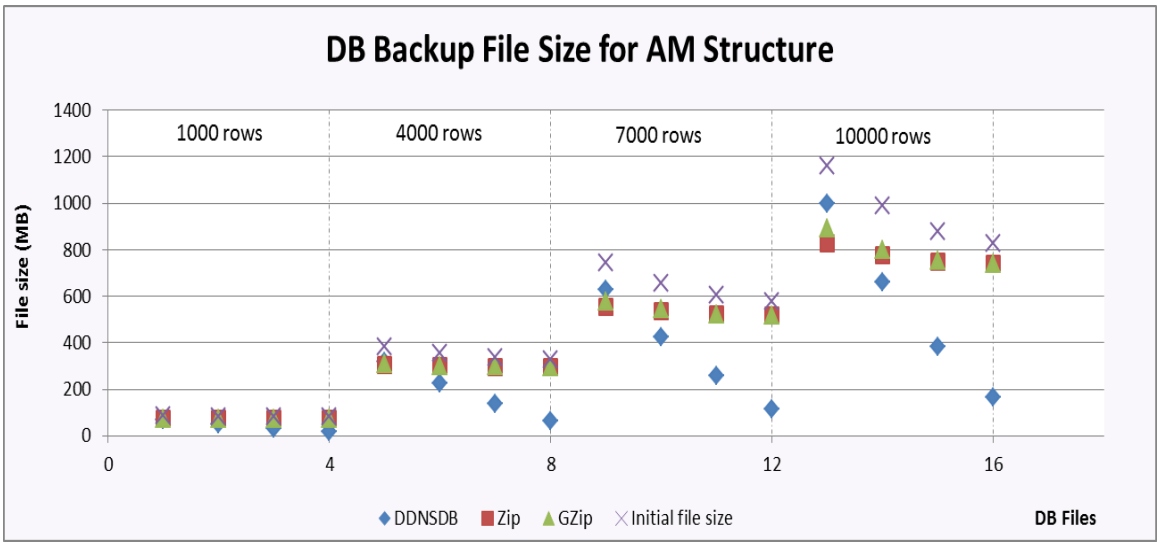

Figure 7.7 (a): DB Backup File sizes comparison for AM structure

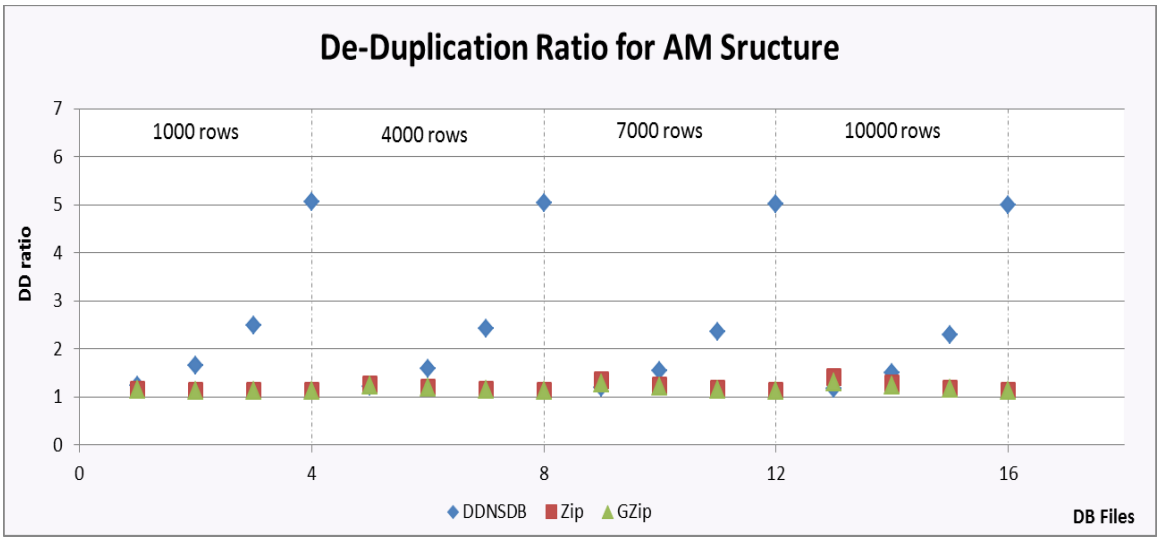

Figure 7.8 (a): DD Ratio for DDNSDB, Zip, and GZip for AM structure

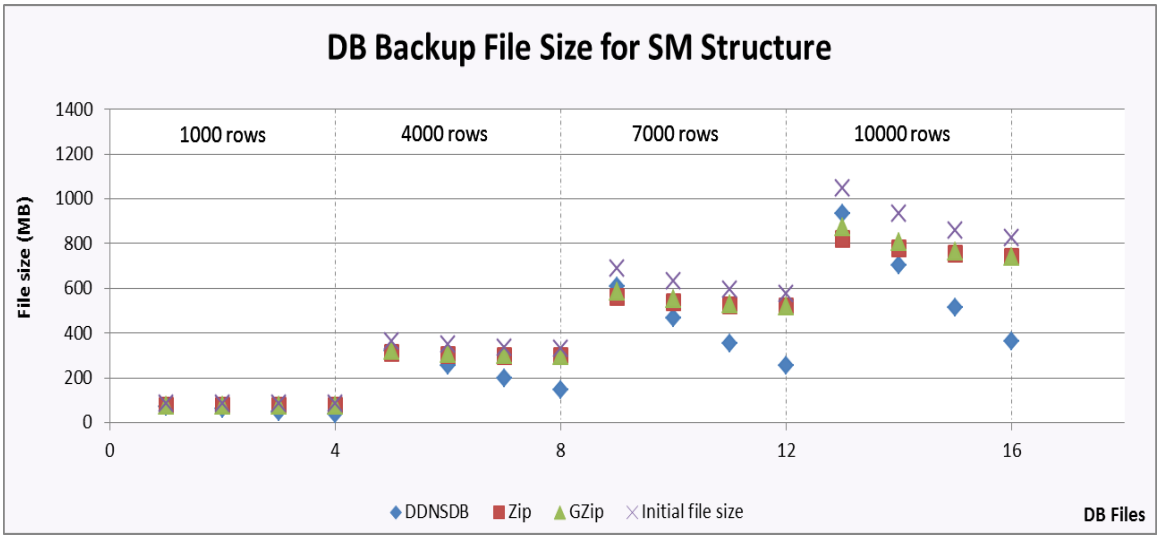

Figure 7.7 (b): DB Backup File sizes comparison for SM structure

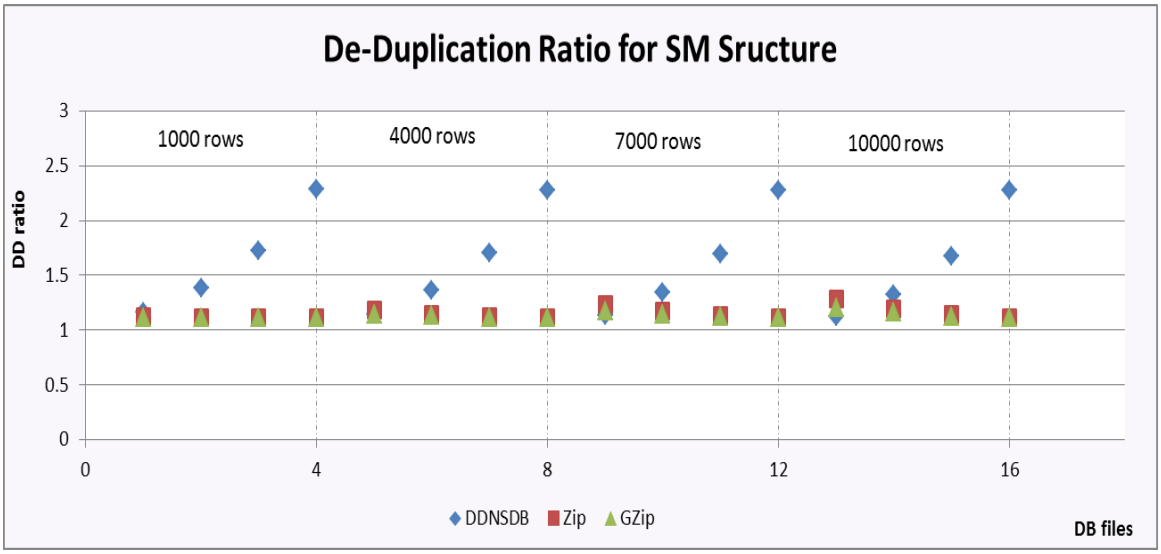

Figure 7.8 (b): DD Ratio for DDNSDB, Zip, and GZip for SM structure

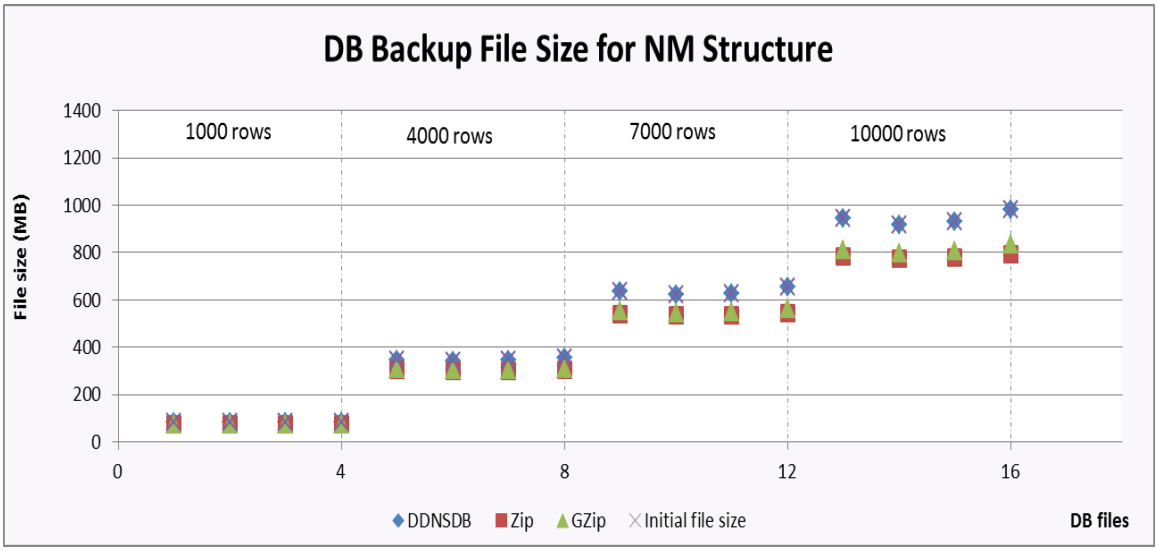

Figure 7.7 (c): DB Backup File sizes comparison for NM structure

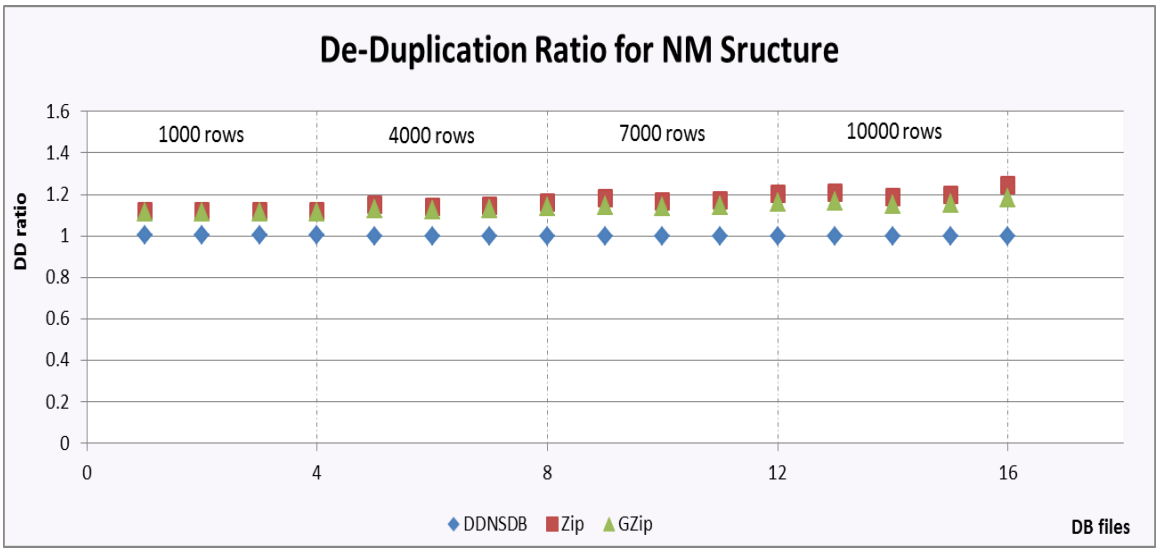

Figure 7.8 (c): DD Ratio for DDNSDB, Zip, and GZip for NM structure

The results show that for AM DB files with high percentage of duplicate data, DDNSDB outperforms considerably Windows Zip and UNIX GZip algorithms (e.g for DB file AM100080 the DD ratio obtained through the 3 algorithms are: DDNSDB = 5.06, Zip = 1.115, and GZip = 1.113). For Zip and GZip algorithms the results are very similar, remaining more or less constant across the files. On the other hand, the space reduction achieved using DDNSDB is 22% higher than the other two algorithms for the DB files with 80% duplicate data and the DD ratio is situated at approximately 5:1. For AM DB files with the lowest percentage of duplicate data (20%), the DD ratio of DDNSDB is very close, slightly larger than the DD ratio obtained by using either Windows Zip compression or UNIX GZip compression algorithms. There is always the internal formatting of data which contributes to the size of the file, directly affecting the measurements of DD ratio. As the structural information decreases (e.g. SM DB Files), the space reduction achieved by DDNSDB compared with Zip and GZip decreases too.

Observing the patterns in each sub-group and group of files, the AM group outperforms the other two, because of the more granular structural information. For the NM files, Zip and GZip algorithms perform slightly better as there is no duplicate data to be detected by DDNSDB. For the SM and AM files, the space reduction goes from 17% to near 69% higher than the other two algorithms, as more structural information is available

DD ratios of 1.5:1 to 5:1 seem reasonable to expect for DB files. Nagapramod et al. [37] conclude in their research about different DD algorithms applied at file and sub-file level that a fold factor of 1:6 to 2.0 is expected for variable sized chunking, out of a single day backup independent of rate of change of data, or backup schedule, or backup

algorithm used. Comparing DD at DB level with DD at file and sub-file level, we obtain similar results. The difference is that DDNSDB is used for NoSQL DBs, where file and sub-file level DD never proved to be as good. DDNSDB is also a fast process and it makes use of structural information at the DB level as chunking algorithm. This information helps identifying larger chunks of duplicate data, producing higher DD ratios in the end.

Figure 7.7 shows the four different file sizes: the initial backup file size, the DDNSDB backup file size, the Windows Zip, and UNIX GZip files. Again, we observe that for low percentage of duplicate data, DDNSDB still outperforms Windows Zip and UNIX GZip. As the percentage of duplicate data increases, DDNSDB produces much smaller files reducing the data footprint significantly for files where structural information is available.

7.2 Scaling the DDNSDB

For all the topologies in the second set of experiments, only the second set of data is relevant. In order to accurately measure the DDNSDB performance, the same amount of duplicate data has to be detected in all the three different scenarios considered for the structural information AM, SM, and NM (e.g. for AM100020, SM100020, and NM100020 there should be 20% of duplicate data detectable by DDNSDB regardless of the tuple complexity, hence the identical duplicate rows structure).

The abbreviations used in the explanation of the results stand as follow:

- T – Topology (e.g. T1 – Topology 1).
- R – Run (e.g. R1 – first run – each experiment was run 3 times).

- The T and R are usually followed by a number which represents the number of map/reduce processes used in the experiment (e.g. T1R1-6 Topology1, Run1, 6 map/reduce processes).
- If R is not present, the average of the 3 runs was used for the table/figure calculations.

There is not much information available regarding the platform on which EC2 is running. To evaluate the performance consistency on the Cloud Platform (CP), each experiment was run 3 times and the average was used for the calculations. Figure 7.9 and Figure 7.10 illustrate the results of the three runs on Topology 1 (T1) for two sets of parallel processes. All the results follow closely the same patterns presented in figure 7.9, and figure 7.10. Like in the previous set of experiments, the DB files are ordered first by structural information (AM, SM, NM), than by size (1000, 4000, 7000, and 10000 rows), and lastly by distribution of redundant data (20%, 40%, 60%, 80%).

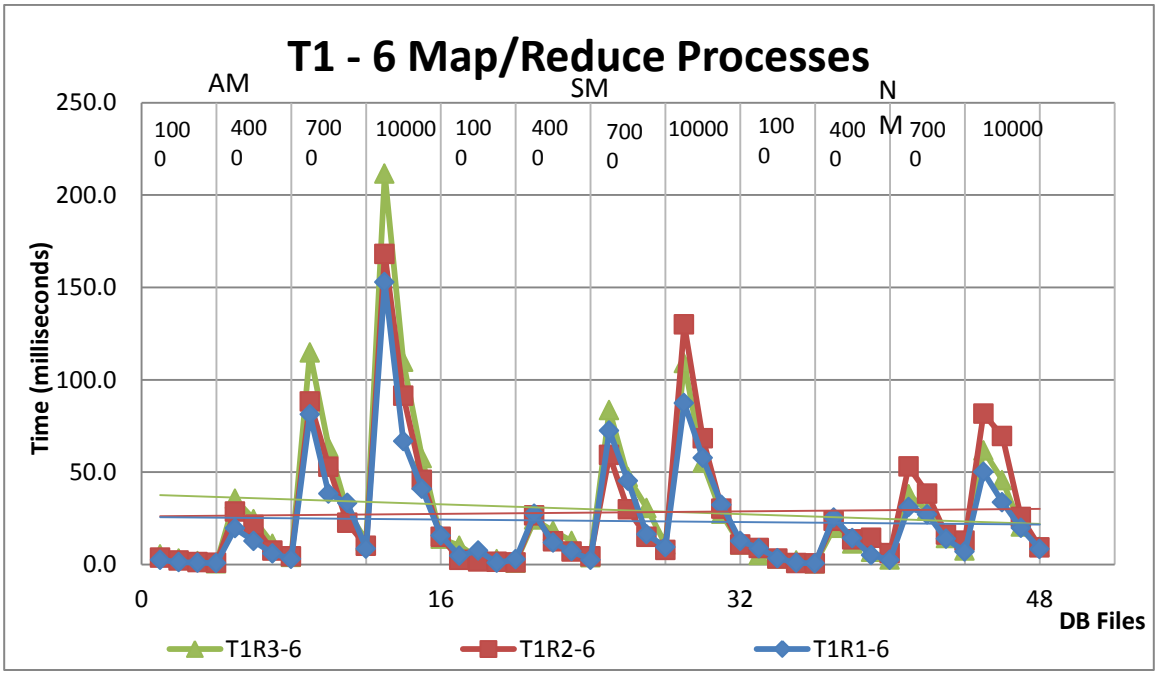

Figure 7.9: DDNSDB Timing for T1 with 6 map/reduce processes

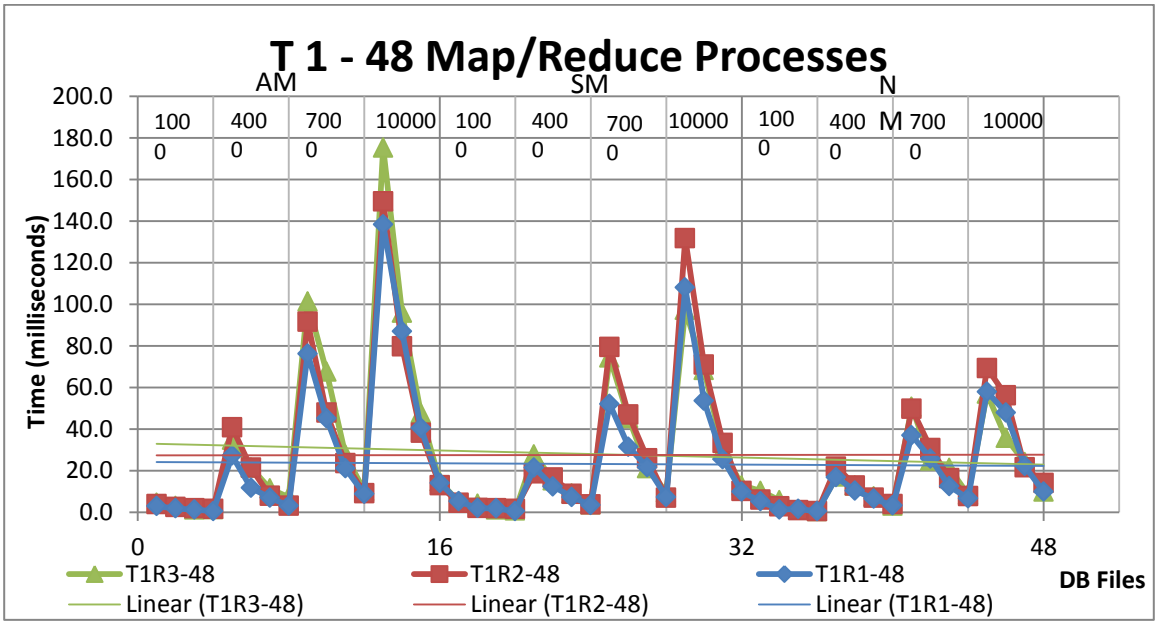

Figure 7.10: DDNSDB Timing for T1 with 48 map/reduce processes

The results show that, overall the first run is slightly faster than the second and third run. The second run is faster than the third run for the first half of the files, after which the third run becomes faster. The possible cause of this behavior is the underlying thin provisioning used by Amazon EC2 at the storage and VM level. Once more space than the initial allocation is required by the growing data, the performance of the process can be affected by the thin provisioning process which needs to allocate and initiate more space. Once more space is allocated the performance starts to improve again (R3). Another factor which can add to this behavior is the fragmentation which occurs after creating and deleting files. The highest time difference between runs obtained was of 0.06 seconds (58 milliseconds) which is very small. These types of results were obtained

almost consistently across all the topologies considered. There are other patterns that are observed in these experiments for each level of grouping and they are explained in more details next.

- At the first level of grouping (structural information), the time cost is reducing as the tuple complexity decreases.
- At the second level of grouping (size of the DB file), the time cost is rising as the number of rows in the DB file increases.
- At the third level of grouping (distribution of redundant data), the time cost is decreasing as the percentage of duplicate data increases.

7.2.1 DDNSDB Performance and Structural Information

In order to measure how the structural information influences the performance of DDNSDB, the same amount of duplicate data is detected for each tuple complexity. The results are presented in figure 7.11 where the DDNSDB was applied to the AM, SM, and NM files. The DB files are ordered first by structural information then by size, and lastly by distribution of redundant data. The average between all runs in T1 was calculated for all AM, SM, and respectively NM in the graphical representation.

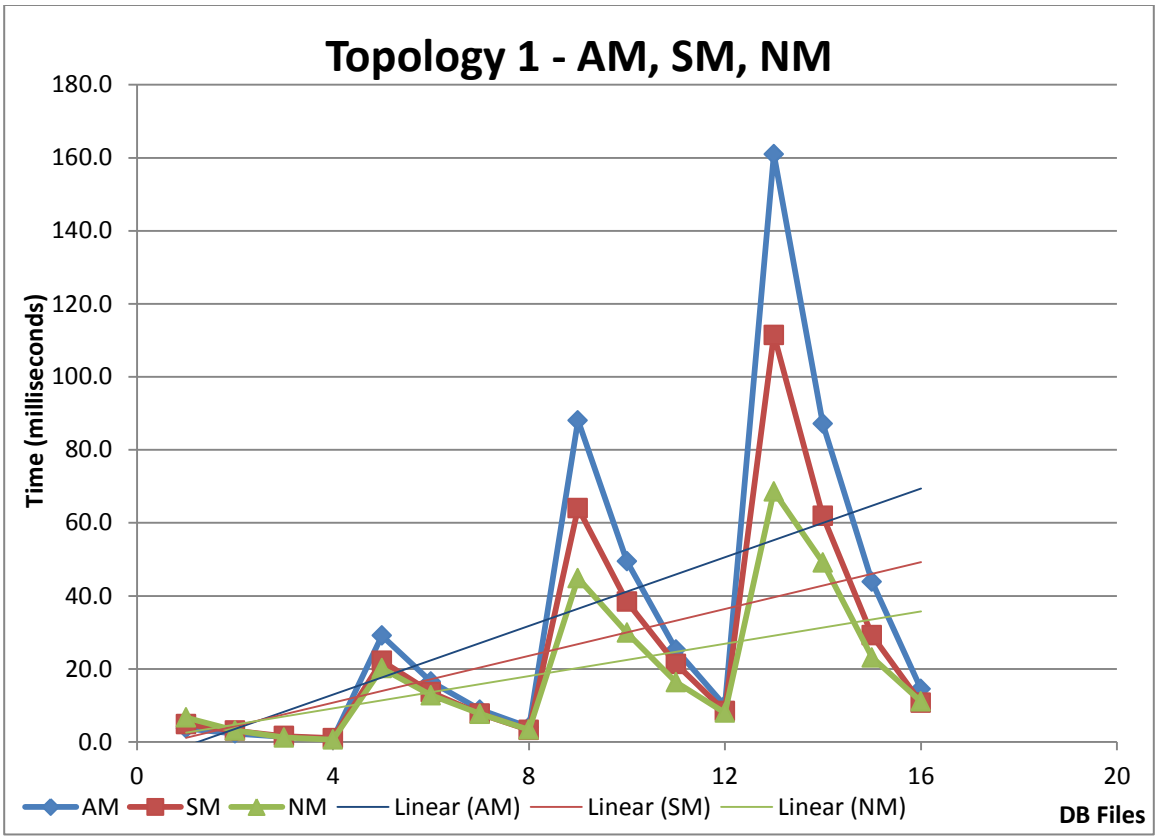

Figure 7.11: DDNSDB performance based on structural information

Based on the structural information, the overall results show that with the decrease of the amount of structural information, the performance of DDNSDB raises. Figure 7.11 also shows the linear trend lines of the growth for each group of structural information. The overall average difference between them is 6 milliseconds. The cause of this behavior can be attributed to the fact that the size of the chunks to be compared grows, making parsing of an AM file to be slower than parsing of a SM file, or even more a NM file. To compare the chunks, DDNSDB makes use of hash values rather than comparing them byte to byte, generating a much faster process. Under these circumstances, the size of the chunks does not affect the performance. Calculating the

hash value of a larger chunk may take slightly longer but being fewer of them, overall the performance increases.

7.2.2 DDNSDB Performance and Amount of Data

The average time cost of DDNSDB is different for different size of DB Files. This is expected as more data gets processed. The tests were conducted for all the DB files in the second dataset, and the results are presented in figure 7.12. The values represented on the X axis represent the number of rows in a DB file where the DB files are grouped based on the number of rows. The average time cost for each group was calculated.

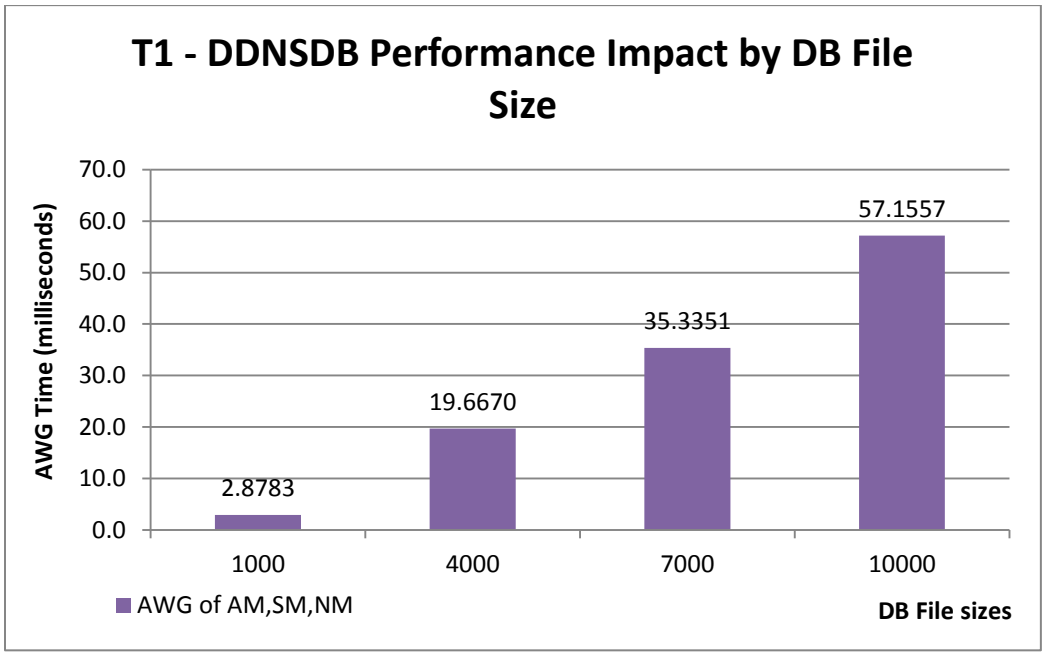

Figure 7.12: DDNSDB Performance based on the amount of data

The results illustrate the relationship between the size of the files and the time it takes DDNSDB to complete the DD operations. As expected, the larger the DB File size, the longer the operations take to complete. For the smaller size DB files (1000 rows – 102,113,391 bytes) the basic cost of the DD operation is 20 times smaller than for larger size DB files (10,000 rows – 1,132,138,892 bytes) while the file size is only 10 times smaller, making the time cost curve relatively steep. However, the DD process takes in the range of milliseconds (57) to complete for a 1 GB size DB file which is still very fast.

7.2.3 DDNSDB Performance and Distribution of Redundant Data

The distribution of redundant data is another important factor which influences the DDNSD performance. The experiments were run against all the files from dataset two and the measurements for DDNSDB time cost are illustrated in table 7.3, and figure 7.13. The files for this experiment are grouped by the percentage of distributed data and size. An average is calculated for the different structural information based on the above grouping.

100020	**400020**	**700020**	**1000020**
3.921	27.9447	94.755	177.4873
100040	**400040**	**700040**	**1000040**
2.2207	19.49	51.5843	89.3231
100060	**400060**	**700060**	**1000060**
1.383	8.2417	28.3817	48.1367
100080	**400080**	**700080**	**1000080**
0.7853	3.796	10.5093	14.9243

Table 7.3: DDNSDB performance based on the distribution of duplicate data

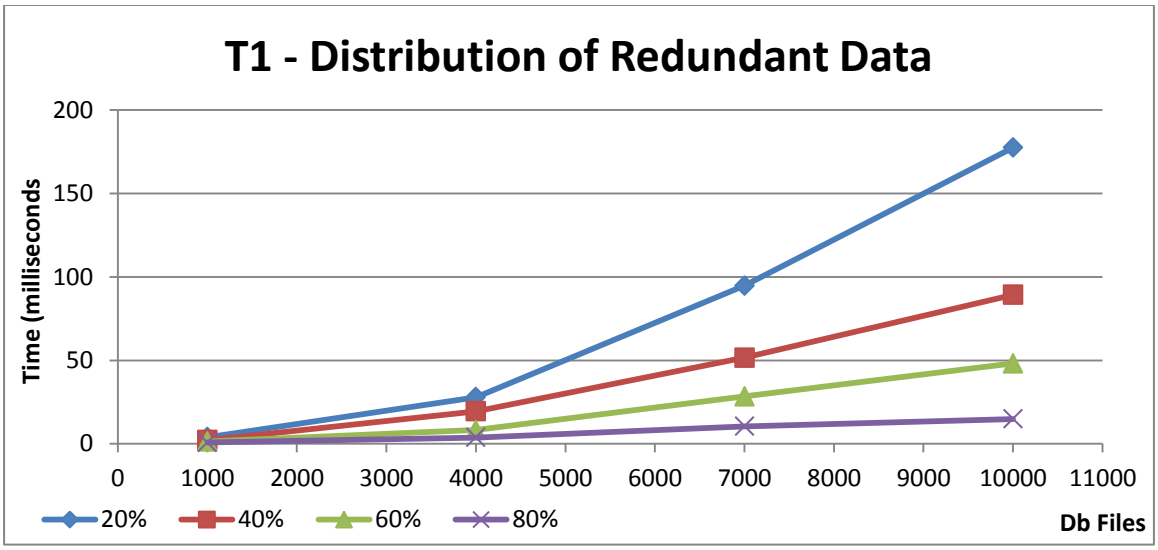

Figure 7.13: DDNSDB performance based on the distribution of duplicate data

The results in figure 7.13 show that, as the amount of redundant data increases, it takes less time for the operation to complete. For DB file with 1000 rows and 20% duplicates, DDNSDB runs for 3.9 milliseconds, and for DB files with 1000 rows and 80% duplicates, DDNSDB only runs for 0.8 milliseconds. In the same time, figure 7.13 also clearly shows that as the amount of data to be processed increases, the proportions between the times for the operations to complete are higher. For DB files with 1000 rows it takes almost 80% less time to complete for 80% redundant data than for 20% redundant data. For DB files with 10000 rows it takes almost 92% less time to complete for 80% redundant data than for 20% redundant data. The cause for why the time cost growth is steeper for the DB files with 20% duplicate data than for the DB files with 80% duplicate data can be attributed to the time it takes to write the data versus just a pointer. Where fewer duplicates are identified, more chunks of data needs to be written causing an

increase number of disk IO. When more duplicates are identified, only the pointers need to be written rather than the entire chunk of data.

7.2.4 DDNSDB Performance and Number of Parallel Map/Reduce Processes

To evaluate how the number of parallel processes used for map/reduce is affecting the DDNSDB performance, the experiments were run on the first and second topology chosen for goal 3. Both topologies involve one big machine, first one uses Erlang's internal message scheduler exclusively, while the second one uses parallel EVM nodes where different scheduler threads may compete for the same resources. The four sets of map/reduce processes combinations used are: 4M+2R=6, 8M+4R=12, 16M+8R=24, and 32M+16R=48. Table 7.6 and figure 7.15 illustrate the comparison results of the experiments run against the two topologies. The values in the table and graph representation indicate the overall averages of all the experiments against each set of map/reduce processes.

T1 - 6 (4+2)	**T1 - 12 (8+4)**	**T1 - 24 (16+8)**	**T1 - 48 (32+16)**
27.1948	26.2258	24.8575	26.3027
T2 - 6 (4+2)	**T2 - 12 (8+4)**	**T2 - 24 (16+8)**	**T2 - 48 (32+16)**
35.7125	32.2353	37.3025	32.9448

Table 7.4: Comparison of average DD Ratio between T1 and T2

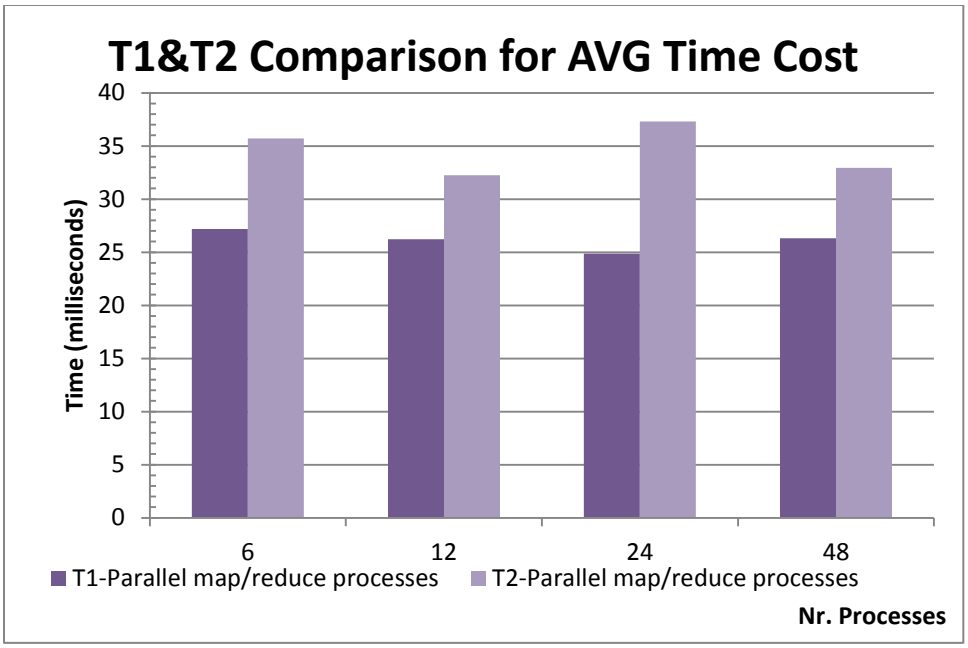

Figure 7.14: Comparison of AVG DD Ratio between T1 and T2

The first observation in table 7.5 is that in T1, the best performance was obtained with 24 map/reduce processes and second best performance was obtained with 12 map/reduce processes. For T2, the best performance was obtained with 12 map/reduce processes and second best performance was obtained with 48 map/reduce processes. The design of DDNSDB is so that the maximum number of parallel processes at a time is equal with the number of map processes plus two (main program and splitter module). The last two processes have very little to do while the mapping occurs, such that their CPU usage consumption can be ignored. This leaves only the map processes to use all the CPU resources. Both topologies use the same hardware resources, more specifically 8CPU with 2 threads each. The set with 24 map/reduce processes fits the best this topology with maximum 16 parallel processes for mapping and 8 parallel processes for reducer. The combinations below this value underuse the resources, and the combinations

above this value have competing resource requirements making the DDNSDB slower in both situations.

The second observation in figure 7.15 is that T1 outperforms T2 in all the experiments. This contributes to the fact that on a big machine with plenty of resources, Erlang's internal message scheduling is more efficient than how the OS tries to evenly distribute the operations across resources (CPUs).

7.2.5 DDNSDB Performance and Number of Physical/Virtual Machines

To evaluate how DDNSDB would perform on commodity hardware and how the network traffic influences the time cost, four hardware configurations were considered in T3. The hardware configurations details are available in table 6.2. Each hardware configuration was mapped to a sub-topology of T3, and again each experiment was run 3 times. The same four sets of map/reduce processes combinations used in the previous experiment are used for this experiment as well. Each experiment was run against the same 48 files from dataset two. The average run for each file in each map/reduce combination was calculated and to better observe the performance trend of DDNSDB in each seat, an average across all the files for that set was also calculated.

In T3.1, T3.3, and T3.4 the same number (four) of EC2 VMs with different hardware configurations were used. Table 7.7 illustrates the average values results for each of these sub-topologies while figure 7.16 illustrates the graphical representation of these values.

T3.1 - 6 (4+2)	T3.1 - 12 (8+4)	T3.1 - 24 (16+8)	T3.1 - 48 (32+16)	T3.1 AWG
59.7412	67.1078	56.2946	60.8778	61.00535
T3.3 - 6 (4+2)	**T3.3 - 12 (8+4)**	**T3.3 - 24 (16+8)**	**T3.3 - 48 (32+16)**	**T3.3 AWG**
62.8784	59.2976	73.4013	77.9474	68.381175
T3.4 - 6 (4+2)	**T3.4 - 12 (8+4)**	**T3.4 - 24 (16+8)**	**T3.4 - 48 (32+16)**	**T3.4 AWG**
242.8378	272.6891	307.7625	332.5421	288.957875

Table 7.5: DDNSDB average run-time for T3.1, T3.3 & T3.4

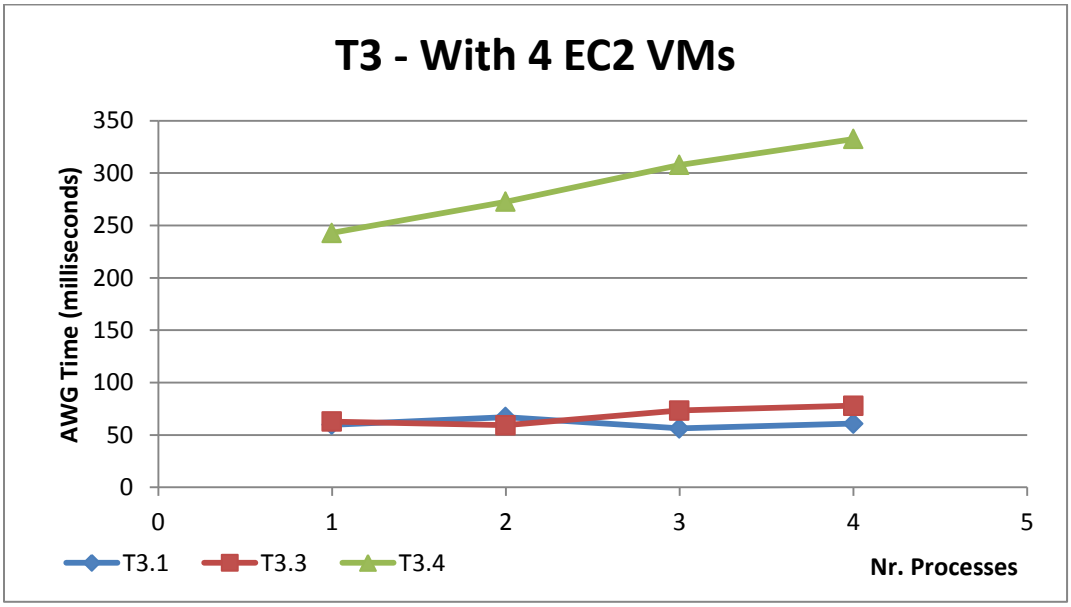

Figure 7.15: DDNSDB performance comparison between T3.1, T3.3, and T3.4

The results show that for T3.1 (13 ECUs over 4 virtual CPUs), the best performance was achieved with the combination of 24 (16/8) map/reduce processes. For T3.3 (4 ECUs over 2 virtual cores), the best performance was achieved with the combination of 12 (8/4) map/reduce processes. For T3.4 (up to 2 ECUs for short periodic bursts), the best performance was achieved with the combination of 6 (4/2) map/reduce

processes. These results further demonstrate the conclusion reached in T2 where the closer the mapping of actual parallel processes to total number of parallel CPU thread, the better DDNSD performs. This is the case also when messages travel across network.

Comparing the overall average performance between the three sub-topologies in table 7.7, the results shows that T3.1 which lies on more powerful EC2 VMs has the best performance. As the VMs performance is reduced, the DDNSDB performance decreases as well. The CPU and RAM capacity used for T3.3 is at least 3.3 times lower than the CPU and RAM capacity used for T3.1 but the maximum performance difference is only by approximately 1.3 times lower (17.1 milliseconds).

For a better understanding of the effects of multiple commodity like hardware over one big machine with plenty of resources, table 7.8 and figure 7.17 illustrates the comparison between T3.1, T3.2, and T1. T3.1 has the best performance out of all T3 sub-topologies with 4 EC2 VMs and T3.2 sub-topology uses only 2 EC2 VMs with a similar overall capacity as T1.

T1 - 6 (4+2)	**T1 - 12 (8+4)**	**T1 - 24 (16+8)**	**T1 - 48 (32+16)**	**T1 AWG**
27.1948	26.2258	24.8575	26.3027	26.1452
T3.1 - 6 (4+2)	**T3.1 - 12 (8+4)**	**T3.1 - 24 (16+8)**	**T3.1 - 48 (32+16)**	**T3.1 AWG**
59.7412	67.1078	56.2946	60.8778	61.00535
T3.2 - 6 (4+2)	**T3.2 - 12 (8+4)**	**T3.2 - 24 (16+8)**	**T3.2 - 48 (32+16)**	**T3.2 AWG**
58.7685	38.6638	59.9429	63.173	55.13705

Table 7.6: DDNSDB average run-time for T1, T3.1, and T3.2

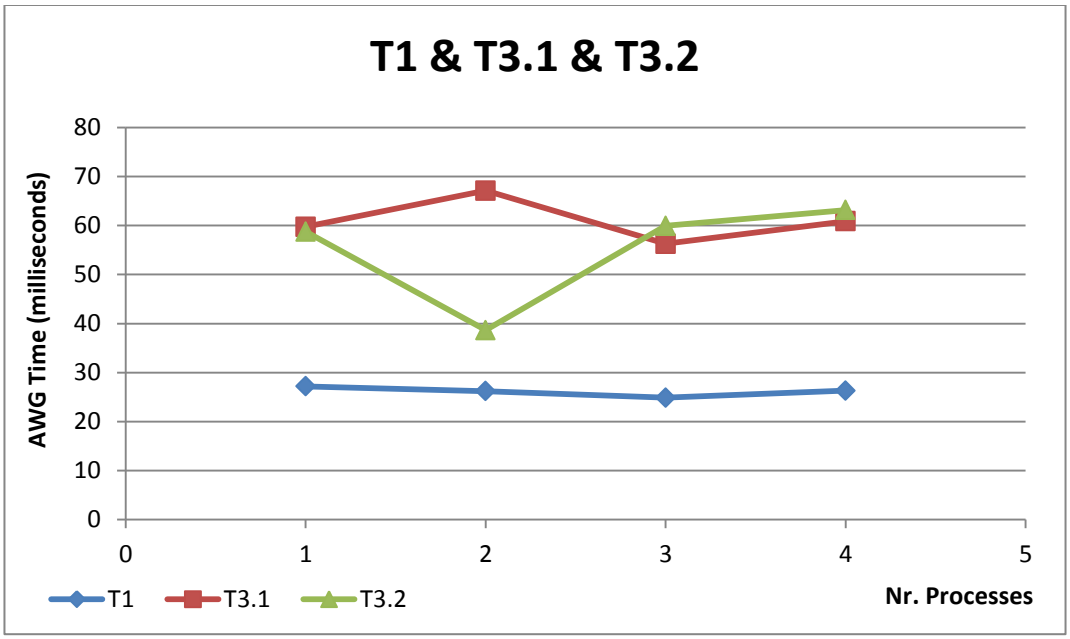

Figure 7.16: DDNSDB performance comparison between T1, T3.1, and T3.2

The results show that T1 outperforms T3.1 by 2.3 orders of magnitude and T3.2 by 2.1 orders of magnitude due to lower resources and the network traffic between the parallel processes within one VM and across VMs. This can be considered significant but in reality the difference is respectively 34.9 milliseconds and 28.8 milliseconds, making the commodity hardware very acceptable for this type of processing. The comparison of the best time cost between the three scenarios shows the minimum time cost differences:

- T1 (24) and T3.1 (24) => 31.3 milliseconds.
- T1 (24) and T3.2 (12) => 13.8 milliseconds.
- T3.1 (24) and T3.2 (12) => 17.6 milliseconds.

These values are calculated based on the average run for all the files for each respective map/reduce set, and show even smaller time cost differences between a one big machine configuration versus several smaller machines. It is clear that the big

machine configuration has higher performance. However, the cost of such machine is also very high. By distributing the processing in parallel across several smaller machines, the performance obtained is very competitive achieving at the same time higher availability and fault tolerance.

7.2.6 Summary

DDNSDB was evaluated to determine the space optimization implications of employing a DD process at the NoSQL DB level and the performance implications of its map/reduce architecture on a Cloud Platform. The analysis and experiments results are summarized for each research goal for an overall view below.

Goal 1. Adapt file and sub-file based DD approaches to the NoSQL DB DD

Analysis Goal 1: Adapted the chunking DD methodology used in file and sub-file based DD to NoSQL DB DD. The analysis presented in chapter 4, exposed how the two steps used in chunking based DD technology can be used for NoSQL DBs as well. The first step – chunking - uses the structural information about the data to chunk the DB records, while the - duplicate detection - is adapted from the initial architecture to only compare the same data-set types, rather than all the data-set types. This adapted chunking technology was implemented for DDNSDB as a proof of concept.

Goal 2. Explore the use of structural information and its granularity to reduce the uncontrolled duplicate data in NoSQL DBs

DD Ratio and Structural Information: Evaluated the implications of structural information on the DD ratio. The experiments proved that significant higher

DD ratios are obtained as more structural information is available (from 1.6 to 2.5 higher). When the same data is spread across multiple smaller chunks to be compared and if one character changes, the probability is higher to detect the other chunks as duplicates than when the data is spread across fewer but larger chunks. At the same time, the experiments show that the larger the duplicate chunks identified, the higher is the DD ratio obtained. This is essentially because of the additional formatting required for the chunks which tend to take space in the backup file.

DD Ratio and the Amount of Data: Evaluated the implications of the amount of data on the DD Ratio. The experiments show that the DD ratio is minimally impacted by the data growth. The slight data growth is caused by the additional formatting required for more data. For a file 10 times bigger the overhead is in the range of 7%.

Distribution of Duplicate Data: Evaluated how the distribution of duplicated data affects the DD ratio. The experiments show a surge of the DD ratio when the amount of duplicate data increases. For a file with 60% redundant data, the backup file after DD is 60% smaller.

DD Ratio Comparison: Evaluated the performance of DDNSDB in comparison with two file compression algorithms, Windows Zip and UNIX GZip. The experiments show that for DB files where duplicate data was detected, DDNSDB outperforms both compression algorithms. It also shows that for DB files with more structural information available for chunking, implying more duplicate data detected, DDNSDB reduces the data footprint overall with 22% more than the other two

algorithms (from 17% to near 69% as more structural information is available). Also the results are comparable with other existing DD technologies used at file and sub-file level.

Goal 3. Develop a scalable architecture for the DD tool to minimize processing time

DDNSDB Performance and Structural Information: Evaluated the performance implications of structural information on DDNSDB. As expected, the experiments show that for larger amounts of structural information, which translates in more chunks to be compared, it takes longer for the operations to complete. Six milliseconds overall difference was registered between the three different types of structural information AM, SM, and NM. The results also show that comparing fewer larger chunks takes less time than comparing smaller chunks but more of them.

DDNSDB Performance and Amount of Data: Evaluated the performance implications of the amount of data on DDNSDB. As expected, the experiments show that it takes longer to process larger amounts of data. For DB file sizes 11 times larger, the time cost is 20 times higher; however the time to process a 1GB file is only approximately 57 milliseconds making this operation still very cheap.

DDNSDB Performance and Distribution of Redundant Data: Evaluated the performance implications of the redundant data on DDNSDB. The experiments show that it takes less time for the operation to complete when there is more redundant data. Also, the growth proportions drop significantly for larger amounts of data e.g for 80% duplicate data it takes 12 times less to process than for 20% duplicate data for the 10000 rows DB

files, while for 1000 rows DB files for 80% duplicate data it takes only 5 times less to process than for 20% redundant data.

DDNSDB Performance and Number of Parallel Map/Reduce Processes: Evaluated the performance implications of the number of map/reduce processes on DDNSDB. The experiments show that the best performance is obtained when the number of parallel processes matches the total number of CPU threads available. At the same time, Erlang's internal message scheduling outperforms the OS scheduling regardless of the number of map/reduce processes.

DDNSDB Performance and the Number of Physical/Virtual Machines: Evaluated the performance implications of the number of machines on DDNSDB. As expected, the experiments show that overall it takes less time for the operations to complete for a lower number of VMs with higher CPU and RAM resources because there is less network traffic and message scheduling. However, the performance differences are not that significant to motivate the significantly more expensive hardware (from 34.9 milliseconds to 28.8 milliseconds as the number of parallel VMs increases). Therefore, the resources provided in a cloud environment which grow horizontally rather than vertically prove to be just as competitive performance wise. They are also significantly cheaper for this type of processing, adding features like 99.9% availability to the package.

CHAPTER 8
CONCLUSION AND FUTURE WORK

8.1 Conclusion

The data landscape has changed and with it emerged the NoSQL DBs allowing for massive concurrent reads and writes, and horizontal scaling. New types of DBs require new ways of saving resources to store data. DD at the DB level was never required because normalizing the data in DBs was *"the norm"*. As more data are produced, new requirements of delivering the data arise changing the norms. An obvious evolution of how to make this data manageable has happened and DD is becoming more popular especially at the storage level.

This research presents a novel approach of data DD for unstructured and semi-structured data stored in the NoSQL DBs. In order to understand the internal data model of the different types of NoSQL DBs used, a description of the three more popular types is presented (key-value, columnar, and document based DBs). Subsequently, the fundamental role of the metadata to overcome the huge duplicate data problem encountered with these types of DBs is pointed out. This duplicate data ultimately gets propagated into the DB backups increasing the data footprint.

This research proposes Data De-duplication for NoSQL Databases (DDNSDB), targeting the key-value DB types, which can be used as a pre-step of the backup process. This allows for easy integration with existing backup tools, rather than having to develop new ones.

DDNSDB makes use of the metadata to divide the data into semantic chunks. The amount of structural information available implies a certain degree of granularity based on which data can be compared. Higher granularity implies higher probability of identifying duplicate data. The experimental results proved a higher DD ratio and a better performance of DDNSDB

for DB files with more structural information available and higher percentage of duplicate data. Contrary to how other DD techniques do the comparison, DDNSDB compares only chunks of the same data-set type, minimizing the resource consumption and processing time.

As CP are becoming more popular because of features like horizontal scalability and availability, so are the NoSQL DBs. One of the design considerations for DDNSDB was to be able to scale horizontaly and run on a CP. The current implementation is using an adapted hierarchical Map/Reduce methodology to allow for scalability and increase performance through parallel processing. This allowed for all the experiments of this research to actually run on a CP, more specifically EC2 CP.

While the performance obtained on one big machine with lots of resource is higher than the performance obtain by running the same process on several smaller machines, the time cost differences are not substantial (average of 6 milliseconds). Horizontal scaling proved to be very elastic on Amazon EC2. It is well known that one machine can grow vertically only so much. Also the cost of a one big machine can be quite high compared to the cost of several smaller machines. Additionally, scaling horizontally also comes with higher availability and fault tolerance which are key requirements in today's businesses.

In conclusion, duplicate data is a major issue for NoSQL DBs. DDNSDB makes use of the structural information of the data to reduce the data footprint significantly in the key-value NoSQL DBs. DDNSDB can easily scale horizontally without significant performance impact to run on commodity hardware specific to CP. In the same time, DDNSDB's standalone design can be used along with existing backup tools without requiring them to be redesigned.

8.2 Future Work

Throughout this research, new characteristics that will need to be addressed in the future came to light:

- Expand on the complexity of structural information. At present, DDNSDB allows only a fix number of structural information complexity which is hard coded. A more flexible approach needs to be implemented to allow for any complexity requirements.
- Create a friendly interface for structural information input. At the moment, DDNSDB does not have an interface or the underlying architecture for collecting, storing, and maintaining structural information. DDNSDB assumes this information is available in three different configurations which are passed in as variables. Having such an interface, the underlying structure will allow DDNSDB to collect, store, and use this information repeatedly without having to provide it every time it runs for the same DB. The details of the structural information can be expanded to the DB level or even further to each type of structure within a DB.
- Dynamic allocations of the number of map/reduce processes. Currently, a fixed number of map/reduce processes can be passed as parameters. The experiments show that DDNSDB best performs when the number of map/reduce process maps close to the number of CPU threads. To take maximum advantage of the CPU resources, these values could be picked up dynamically once the processing starts.
- Extend the capabilities of processing larger DB files. Erlang's key-value DB - DETS – which was used for DDNSDB implementation has some DB file size restrictions. To overcome these restrictions, a different type of key-value DB can

be used (like Mnesia), or distribute the data across multiple DBs and processes them as a group. This will allow DDNSDB to process more real-life data where DB files sizes can be larger than 2GB. The performance expectations for processing larger DBs should be proportionate with the values obtained in the experiments of this research.

- Extend the capabilities of processing other data models. DDNSDB was developed to address one type of NoSQL DB data model as being the most representative, namely the key-value DB. This research presents two other main types of NoSQL DBs, classifying them based on their data model. These DBs also have very basic structures, making them prone for storing duplicate data as well. DDNSDB can be extended to be able to process other data models following the same DD algorithm. This can be achieved in different ways. DDNSDB was built using the Erlang programming language because of its easy way of spawning and managing parallel process. Other NoSQL DBs use different programming languages and have their own APIs through which they allow the interaction with the data. One way to achieve this is by implementing a set of APIs as a web service to interface with other types of NoSQL DBs.
- Evaluate DDNSDB on different public Cloud Platforms. This research focused on implementing a novel idea and performing a proof of concept implementation through DDNSDB. The evaluation focused on the amount of space that can be saved with this approach, and how easily the toll can scale horizontally for a better performance on a CP, namely on Amazon EC2. An interesting thing would be to evaluate the performance of DDNSDB on other existing CPs like Azure

Service Platform or Google App Engine. This will help to identify and improve how DDNSDB performs on different underlying cloud technologies.

- Evaluate DDNSDB on private CP, where there is more control on the type of hardware used and the type of provisioning at storage level and/or virtualization level. Amazon EC2 most likely is using soft provisioning at storage and virtualization level. In a private cloud, it is possible to compare how soft provisioning performs versus hard provisioning, to evaluate if the extra cost is worth it. This can lead to different design improvements of DDNSDB to better make use of the resources.
- Improve fault tolerance. During the experiments, situations were identified when long running processes were altering the results and had to be repeated. Implementing a more fault tolerant design where such processes can be detected and restarted or terminated depending on the nature can help the DD process performance.
- Extend the scalability features, by implementing a *"1 to n"* relationship between the nodes and the number of parallel processes which run on each node, when scaling across multiple machines. This can avoid the potential of competing for the same resource between processes.
- Perform online DD at the NoSQL DB level. DDNSDB was designed as a standalone tool which can be integrated with existing backup tools as a pre-step. Some DB may also require an online DD process because otherwise they may reach sizes which become unmanageable or unresponsive. The algorithm used for DDNSDB can be extended to be able to perform online DD at the DB level.

LIST OF REFERENCES

[1] Amazon elastic compute cloud (amazon EC2). 2012(February/24), Available: http://aws.amazon.com/ec2.

[2] Amazon SimpleDB. 2012(February/24), Available: http://aws.amazon.com/simpledb.

[3] Apache. HDFS architecture. 2012(February/24), Available: http://hadoop.apache.org/common/docs/r0.20.0/hdfs_design.pdf.

[4] Apache. Cassandra. 2012(February/24), Available: http://cassandra.apache.org/.

[5] Apache CouchDB Project. Apache CouchDB project. (February/24), Available: http://couchdb.apache.org/.

[6] Apache HBase. Apache HBase. 2012(February/24), Available: http://hbase.apache.org/.

[7] D. Bartholomew. SQL vs. NoSQL. LINUX Journal 2010. Available: http://www.linuxjournal.com/article/10770?page=0,0.

[8] J. Bentley and D. McIlroy, "Data compression using long common strings," in Data Compression Conference, 1999. Proceedings. DCC '99, 1999, pp. 287-295.

[9] S. J. Bigelow and J. Hawkins. Data deduplication (intelligent compression or single-instance storage). 2012(February/24), 2008. Available: http://searchstorage.techtarget.com/definition/data-deduplication.

[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Commun ACM, vol. 13, pp. 422-426, jul, 1970.

[11] Burlesons-Consulting. Column oriented data storage for oracle. 2012(February/24), .

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," ACM Trans.Comput.Syst., vol. 26, pp. 4:1-4:26, June, 2008.

[13] Chuanyi Liu, Dapeng Ju, Yu Gu, Youhui Zhang, Dongsheng Wang and D. H. C. Du, "Semantic data de-duplication for archival storage systems," in Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific, 2008, pp. 1-9.

[14] Composite Software. Composite data virtualization and NOSQL data stores composite software 2012(February/24), Available: www.compositesw.com.

[15] S. Das, S. Agarwal, D. Agrawal and A. E. Abbadi, "ElasTraS: An elastic, scalable, and self managing transactional database for the cloud," CS, UCSB, 03/2010. 2010.

[16] C. J. Date, "Introduction to transaction processing," in , 8th ed.Anonymous Addison Wesley, 2003, pp. 295.

[17] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Commun ACM, vol. 51, pp. 107-113, January, 2008.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: amazon's highly available key-value store," SIGOPS Oper.Syst.Rev., vol. 41, pp. 205-220, October, 2007.

[19] M. Dutch and L. Freeman. Understanding data deduplication ratios. Storage Networking Industry Assoc. (SNIA). 2009Available: www.snia.org/forums/dmf/news/articles/SNIA_DeDupe_Ratio_Feb09.pdf.

[20] S. Edlich. NoSQL your ultimate guide to the non-relational universe! 2012(February/24), Available: http://nosql-database.org/.

[21] EMC2-DataDomain. Deduplication storage systems for next-generation backup and recovery. 2012(February/24), Available: http://canada.emc.com/backup-and-recovery/data-domain/data-domain.htm.

[22] FalconStor. Virtual tape library with deduplication 2012(February/24), Available: http://www.falconstor.com/.

[23] D. Geer, "Reducing the Storage Burden via Data Deduplication," Computer, vol. 41, pp. 15-17, 2008.

[24] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," SIGACT News, vol. 33, pp. 51-59, June, 2002.

[25] Google. Google app engine 2012(February/24), Available: http://code.google.com/appengine/.

[26] Guanlin Lu, Yu Jin and D. H. C. Du, "Frequency based chunking for data de-duplication," in Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on, 2010, pp. 287-296.

[27] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," ACM Comput.Surv., vol. 15, pp. 287-317, December, 1983.

[28] HP. Deduplication and the data explosion. 2012(February/24), Available: http://h71028.www7.hp.com/enterprise/us/en/solutions/storage-data-protection-with-deduplication.html.

[29] IBM. IBM system storage TS7650, TS7650G, and TS7610. 2012(February/24), Available: http://www.redbooks.ibm.com/abstracts/sg247652.html.

[30] IBM. IBM - tivoli storage manager 2012(February/24), Available: http://www.ibm.com/developerworks/wikis/download/attachments/106987789/TSMDataDeduplication.pdf?version=1.

[31] R. Jones. Anti-RDBMS: A list of distributed key-value stores. 2009. Available: http://www.metabrew.com/article/anti-rdbms-a-list-of-distributed-key-value-stores.

[32] R. Kimball and M. Ross, The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. Wiley, 2002.

[33] P. Kulkarni, F. Douglis, J. LaVoie and J. M. Tracey, "Redundancy elimination within large collections of files," in Proceedings of the Annual Conference on USENIX Annual Technical Conference, Boston, MA, 2004, pp. 5-5.

[34] N. Leavitt, "Will NoSQL Databases Live Up to Their Promise?" Computer, vol. 43, pp. 12-14, feb, 2010.

[35] F. Maia, J. Armendáriz-Iñigo, M. Ruiz-Fuertes and R. Oliveira, "Scalable transactions in the cloud: Partitioning revisited," in Proceedings of the 2010 International Conference on on the Move to Meaningful Internet Systems: Part II, Hersonissos, Crete, Greece, 2010, pp. 785-797.

[36] U. Manber, "Finding similar files in a large file system," in Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, San Francisco, California, 1994, pp. 2-2.

[37] N. Mandagere, P. Zhou, M. A. Smith and S. Uttamchandani, "Demystifying data deduplication," in Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion, Leuven, Belgium, 2008, pp. 12-17.

[38] MongoDB. 2012(February/24), Available: http://www.mongodb.org/.

[39] NetApp - deduplication and compression. 2012(February/24), Available: http://www.netapp.com/us/products/platform-os/dedupe.html.

[40] Oracle. Berkeley DB products. 2012(February/24), .

[41] Oracle. RMAN backup concepts. 2012(February/24), Available: http://download.oracle.com/docs/cd/B28359_01/backup.111/b28270/rcmcncpt.htm#BRADV002.

[42] F. Pianese, P. Bosch, A. Duminuco, N. Janssens, T. Stathopoulos and M. Steiner, "Toward a cloud operating system," in Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP, 2010, pp. 335-342.

[43] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in Proceedings of the Annual Conference on USENIX Annual Technical Conference, Boston, MA, 2004, pp. 6-6.

[44] Project voldemort A distributed database. 2012(February/24), Available: http://www.project-voldemort.com/.

[45] M. O. Rabin, "Fingerprinting by random polynomials," Technical Report TR1581 Center for Research in, pp. 15-18}, 1981.

[46] Riak documentation. 2012(February/24), Available: http://wiki.basho.com/.

[47] M. Sarrel. NoSQL databases – providing extreme scale and flexibility. GigaOmPro. 2010Available: http://pro.gigaom.com/2010/07/report-nosql-databases-providing-extreme-scale-and-flexibility/.

[48] T. Schutt, F. Schintke and A. Reinefeld, "Scalaris: Reliable transactional p2p key/value store," in Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG, Victoria, BC, Canada, 2008, pp. 41-48.

[49] SSTable. 2012(February/24), Available: http://en.wikipedia.org/wiki/SSTable.

[50] U.S. Department of Commerce / National Institute of Standards and Technology. Secure hash standard (SHS). FIPS 1801993. Available: http://securityv.isu.edu/isl/fips180.html.

[51] L. M. Vaquero, L. Rodero-Merino, J. Caceres and M. Lindner, "A break in the clouds: towards a cloud definition," SIGCOMM Comput.Commun.Rev., vol. 39, pp. 50-55, December, 2008.

[52] W. Vogels, "Eventually consistent," Commun ACM, vol. 52, pp. 40-44, January, 2009.

[53] Yujuan Tan, Dan Feng, Zhichao Yan and Guohui Zhou, "DAM: A DataOwnership-aware multi-layered de-duplication scheme," in Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on, 2010, pp. 403-411.