**Assessment of Multiple Ingest Strategies for Accumulo Key-Value Store**

by

Hai Pham

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 7, 2016

Keywords: Accumulo, noSQL, ingest



Approved by

Weikuan Yu, Co-Chair, Associate Professor of Computer Science, Florida State University
Saad Biaz, Co-Chair, Professor of Computer Science and Software Engineering, Auburn University
Sanjeev Baskiyar, Associate Professor of Computer Science and Software Engineering, Auburn University

Abstract

In recent years, the emergence of heterogeneous data, especially of the unstructured type, has been extremely rapid. The data growth happens concurrently in 3 dimensions: volume (size), velocity (growth rate) and variety (many types). This emerging trend has opened a new broad area of research, widely accepted as Big Data, which focuses on how to acquire, organize and manage huge amount of data effectively and efficiently. When coping with such Big Data, the traditional approach using RDBMS has been inefficient; because of this problem, a more efficient system named noSQL had to be created. This thesis will give an overview knowledge on the aforementioned noSQL systems and will then delve into a more specific instance of them which is Accumulo key-value store. Furthermore, since Accumulo is not designed with an ingest interface for users, this thesis focuses on investigating various methods for ingesting data, improving the performance and dealing with numerous parameters affecting this process.

Acknowledgments

First and foremost, I would like to express my profound gratitude to Professor Yu who with great kindness and patience has guided me through not only every aspect of computer science research but also many great directions towards my personal issues. Without his detailed, insightful instructions, as well as other significant helps, I would not have reached to the point today where I can be much confident of carrying out hard works much more accurately and efficiently. Furthermore, I greatly appreciate his style of advising: strict when necessary to keep pace with the harsh timeline while preserving enough freedom and trust to his students. Consequently, I have been very proud of being his advisee.

Second, I would aver my great gratitude to Professor Biaz, my co-advisor. He has shown a great magnanimity and care for his advisee: always approachable, always open, always giving the most practical instructions and always giving his great hands to aid. I am deeply impressed by his guidance, adorable style and feel fortunate to have worked with him.

Third, many collective respects and thanks to Professor Baskiyar in the committee who is always approachable, gentle and supportive to me.

I am also greatly lucky to have worked with great colleagues in Parallel Architecture and System Laboratory (PASL) headed by Professor Yu. To a great extent, we have not only collaborated and worked with each other, but also—not less importantly—we have lived together as a big family and it was definitely a memorable period of time to me. Great thanks to Xinning Wang, Fang Zhou, Kevin Vasko, Hao Zou, Yue Zhu, Lizhen Shi, Micheal Pritchard, Dr. Cong Xu, Teng Wang, Dr. Jianhui Yue, Dr. Hui Chen, Huansong Fu, Dr. Zhuo Liu and Dr. Bin Wang. I learned a lot from each one of you; it has been a great honor and pleasure to be your colleague.

Furthermore, I would express my special thanks to Miller Writing Center in Auburn University, with many writing consultants who have read, edited and given many useful pieces of advice of this thesis' drafts.

Finally, my deepest thankfulness to my beloved family and friends who have been always at my side supporting my decision of pursuing graduate study, one of my toughest decisions in my life.

Table of Contents

List of Figures

List of Tables

# Chapter 1

# Introduction

Big Data has become more and more ubiquitous nowadays with the booming mixed types of data from unstructured (video, image, audio) to the traditional structured data (text, database systems). Amidst such huge amount of data, up to 20 zetabytes by 2020, the unstructured data accounts for as much as 70 - 80% of the total data of any organization [1]. As a result, one of the main challenges and opportunities is to take advantage of this vast amount and make use of various types of unstructured data for research, analysis, business or other purposes. To tackle with such huge amount of unstructured data, the traditional computation and storage approaches have become outdated models as they are not able to keep up with the desired speed and space. The emerging Cloud computing, the MapReduce framework [2], and its particular open–source implementation, Hadoop [3], have been widely employed for dealing with general big data problems.

Concerning MapReduce/Hadoop, there are two separate aspects of concern: the computation and the storage. For computation, the MapReduce dataflow model scatters the data evenly to every node in the distributed environment—in parallel manner—and have them coalesced together to produce the consolidated results. For storage, which is just as important, Hadoop Distributed FileSystem (HDFS) has become the irreplaceable standard for distributed storage with the efficacious capabilities of scalability and resiliency to any type of failure.

While HDFS is becoming more and more essential for any distributed workload, it still has many limitations. First of all, it is shipped with no strong interface for dealing with normal file/directory at Operating System level; users must write their own code to achieve this target. And more importantly, HDFS does not provide a convenient set of application

programming interface for direct services to data science workloads such as data mining, data analysis, machine learning, visualization, etc.

Given such limitation of HDFS towards data science workloads, and with the booming trend of data science in the world in recent years [4], people have now adopted to another type of higher level storage which has more data-processing functionality: noSQL store family. NoSQL systems have converged three important characteristics which are: not being a relational database, not using SQL for manipulating the data and not being built primarily on tables. By this eclectic set of characteristics, it is capable to deal with natural data which is huge and does not require relational model to store and process [5]. Likewise, noSQL system is designed for distributed massive-parallel data processing and large-scale data storage which are needed for taking care of the tasks in Big Data, although it is primitively not equipped with SQL declarative language to query the data. For example, there are two well-known tools for this demand from Apache community: Apache Pig [6] which is of a high-level and flexible scripting language and an even higher level declarative, closer to SQL which is Apache Hive [7]. From their design, Pig has more control over the flow to design a more sophisticated process such as Extract, Transform, Load (a.k.a ETL) whereas Hive is less verbose, much faster for batch jobs such as reporting or data analytics. There are numerous noSQL systems available; each with its own advantages and disadvantages compared with others: Key-Value store, Column store, Document database, and Graph database.

With the emerging role of noSQL key-value store in data science, this work is to investigate the performance of noSQL compared with the original HDFS. The candidate for this study is Accumulo [8], an open-source implementation of sorted, distributed key-value store from National Security Agency and is an emulation of the reputed Google BigTable [9] which serves for many critical production workloads at Google Inc. Not only does Accumulo have a high reported throughput compared with its peers such as Google BigTable, Apache Cassandra [10] or Apache HBase [11] but also it is equipped with state-of-the-art security at cell-level basis, enabling it a potential choice for any strict, security-aware application.

Accumulo was born later than other noSQL stores, but there have been many used cases of Accumulo in various applications. Weber *et al* [12] visualize real-time social media big data with the incorporation with 3D Printing. In particular, they use a program called Twitter Decahose to extract randomly about 10% tweets from Twitter Blog database. The idea is that the majority of users usually tweets through their mobile devices which also share the geographic location information, and then data can be animatedly illustrated in 3D printing for better intuition. From other work regarding cryptography in the same lab, Kepner *et al* [13] introduce a new technique called Computing on Masked Data (CMD) which can directly perform the algebra computation on masked data and allow only authorized personnel to unmask them. Finally, in another disciplinary work towards bioinformatics, Kepner *et al*[14] show that in the area requiring the high performance computation for extreme massive information like such, Accumulo proves to be the optimal choice.

In theory, the basic low-level storage of Accumulo is HDFS, making it a non-negligible overhead on any data application. This work is going to optimize the use of Accumulo store to make it more practical for analytical workloads, and take advantage of its huge capabilities of data–oriented integration and processing. In this work, while trying to optimize the storage functionality of Accumulo, an articulate comparison with HDFS is crucial to demonstrate. The first step is to create a robust interface for ingesting the data into Accumulo which does not exist. Once the ingest of the data is fast enough, it is, by essence, a key to any further efficient data processing which is well–provided by Accumulo.

For the scope, this study will comprise of a broad investigation of Accumulo (in comparison with HDFS) from the normal cases of big or middle-sized files to the more uncommon case of small files. In fact, HDFS is designed to work with big to very big files and likely to behave insufficiently when facing a large number of middle–sized files or even worse in small files—while the demand to process a huge number of small files are more and more universal in many areas such as education, astronomy, climatology, and bioinformatics, making it a compelling area to study. If the optimization through Accumulo can surpass the

disadvantages of HDFS in any case, or reduce significantly those downfalls, it would become an optimistic solution and will be usable broadly.

This thesis is designed as follows: Chapter 2 introduces the background of Accumulo; Chapter 3 describes the motivation of this study; Chapter 4 details the various ingest techniques in Accumulo; Chapter 5 demonstrates experimental results and analysis; Chapter 6 mentions some related works and finally, Chapter 7 will conclude the study.

# Chapter 2

# Background

This section introduces the characteristics of noSQL systems, then compares it with the traditional RDBMS systems, followed by an introduction of Accumulo–a key-value noSQL store, and lastly end up with the use of noSQL and Accumulo systems in real cases.

## 2.1 Introduction to noSQL Store

According to Cattell [15], the most important functionality of noSQL system is horizontal scaling and shared nothing and partitioning along with replicating across many commodity servers. Although there has not been a complete agreement upon the common functionality of a noSQL system, to be considered a noSQL one, it should have these typical characteristics:

1. As mentioned, it should be scalable in horizontal dimension, meaning that we easily supplement more servers to make the noSQL system more capable of storing and managing data.

2. In terms of parallel system design, it is less bound to Atomicity, Consistency, Isolation and Durability as compared with traditional Relational Database Management System (RDBMS).

3. It supplies a flexible and customizable manner of storing the data records.

4. It has distributed indexes to manage the data over a distributed environment.

## 2.2 Comparison of the traditional RDBMS and noSQL store

Traditionally and widely accepted, the RDBMS, which aims for maintaining the constant consistency at any time, relies on ACID, for Atomicity, Consistency, Isolation and Durability [16].

To compare this new noSQL system with traditional RDBMS (often referred to as database systems), it is essential to know the underpinning concept—CAP theorem—established by Eric Brewster [17], which is also a very popular tradeoff for a database system in every real case. According to him, CAP stands for Consistency, Availability and Partition–Tolerance, and a system can only achieve fully two out of three targets at the same time.

To compare with ACID and CAP, there is also a derivative term for noSQL systems which sacrifices some extent of Consistency to serve for Availability and Partitioning, which is known as BASE (Basically Available, Soft-state, Eventually consistent) [5]. Presumably, the first known company who broke the tradition of RDBMS transaction constraint and originated the idea of eventual consistency is Amazon. This functionality allows for the data fetched by a request to not always be up-to-date at a point in time; but ultimately, the consistent updates will be replicated across the data nodes [15].

Another interesting comparison between SQL and noSQL is provided by StoneBreaker [18], even though he only addressed the case of the online transaction processing (OLTP) system. The postulation is that every RDBMS cannot avoid the following overheads:

1. Logging: it is a well-known method in the concept of operating system and DBMS to protect the transaction-level consistency of the system, in which data will be written into a log before being committed and updated into the final storage.

2. Locking: another well-known method to handle concurrent access in which one record is available to one session at a time, avoiding any mismatch of simultaneous updates.

3. Latching: oftentimes, a short interval latches are used to update the data records in the system allowing for many threads.

4. Buffer management: which is the place storing recently-used versions of data in the unit of disk pages. Typically, it is one of the heaviest overhead of a RDBMS.

Given these four traditional cost, StoneBreaker reasons that many noSQL systems, which are disk-based, have buffer and are multithreaded, must incur in two out of four types above. However, the key conclusion from him is that, removing any of such overhead could boost the performance of databases by about 25%. This could be achieved by dealing with ACID transactions, multithreading and disk management, and more importantly, does not concern SQL techniques. SQL will thus retain its role in databases, or noSQL is not a replacement of RDBMS.

### 2.3 Classification of noSQL

The trend of Big Data, specifically noSQL system, has developed incredibly fast since its appearance. Therefore it is needed to consolidate and summarize what the main directions are in this field. Based on the uses of noSQL, apparently by design, there is no such thing as a one-size-fits-all system. Thus it is also a need to differentiate between the systems by having a good comparison. There have been many sources trying to categorize noSQL systems and here is one of the most reasonable classifications [5]:

1. **Key-Value store**: This is the simplest type of noSQL which contains an alpha–numeric key with the associated value and both can be stored in table. The value could be a text string or could be a more complex linked list or sets. Because of this design, the search will primarily be against the key and is limited to exact matches and thus most suited for managing the scalable profiles, product names, etc. One of its biggest advocate is Amazon Dynamo who uses it for managing the shopping cast. Other examples are BerkeleyDB (now owned by Oracle), Voldemolt, and Riak.

2. **Document store**: It is primarily used for storing and managing documents originated by IBM Lotus Notes. The document data is often called semi-structured data as there

is no separation between the data and schema. The document itself can be encoded in a format such as JSON, BSON, XML, YAML. Although it has the same key-value scheme as above, the value of this type is far more sophisticated because the column can be a semi-structured data that holds various name/value pairs. The search on this store can be attained against key or value. The two popular examples that are semi-structured and schema-free are widely known as MongoDB and CouchDB.

3. **Column store**: This family is derivative from Google BigTable which is used for many projects such as Google Web indexing, Google Earth and Google Finance [9]. These stores have a function similar to document stores since a column can hold many attributes per key. Many of such stores replicate not only Google BigTable but also Google File System (GFS) and the MapReduce programming framework. Their derivatives include Apache Hadoop, HBase/Accumulo, and MapReduce. Because MapReduce is designed for batch-style processing, these stores are best suited for this purpose.

4. **Graph databases**: So far the abovementioned types of noSQL systems share the common key-value scheme and also save and maintain it in table-like stores. In graph databases, they instead use a relational graph in an interconnected list of key-value pairs. By this function, its model is similar to Object Oriented Database's concepts such as node, node relationship (edge), and properties (key-value pairs). By this visual nature, the graph system is mostly suitable for traversing the data in applications, such as social networks or forensic investigations. But note that this database is, although good for traversing connections, not designed for query. There are emerging examples of this type, such as Neo4j, InfoGrid, and InfiniteGraph.

Going back to the story of noSQL vs. RDBMS, or ACID vs. BASE, one of the most important factors in distinguishing noSQL is the concurrency property on which noSQL can be classified as 4 types [15]:

1. Systems with traditional lock: only one user is permitted at one time to read or update the data

2. Systems with multi-version concurrency: allows for multiple users to read at the same time but not write, which leads to conflicts.

3. Systems without atomicity (no lock): runs the risk of inconsistent versions which is very undesirable.

4. System combining ACID and no atomicity to have the state in between of these two concurrency types.

Again, no matter how noSQL is classified, everyone should agree upon the ideas that every noSQL system can handle huge amounts of data and is highly scalable to the workload.

### 2.4 Use of noSQL

Concerning the use of noSQL stores, it is probably a must to first and foremost mention Google BigTable which is considered the pioneer leading to the burst of other similar systems. In a specific paper from Google [9], there are details about three applications that used BigTable up to 2006 which are Google Analytics, Google Earth and Personalized Search. First, in Google Analytics which provided various statistics helpful to webmasters about access on web pages. There are two main tables used which are raw click table (about 200TB) and summary table (about 20TB). In more detail, the summary table is stemmed from the raw click table by scheduled MapReduce jobs and the throughput limit of the system is that of Google FileSystem (GFS). Second is Google Earth, which was designed for users to interact with the world's surface as satellite images at different resolutions. The data for this system is about 70TB in disk; however, the serving system that stores the index was just about 500GB but was used for tens of thousands of queries per second per datacenter. As a result, it was hosted in hundreds of tablet servers and contained in-memory column families. Third, the interesting system, Personalized Search, was fundamentally user-centric system

from which users could retrieve their history of query, search, images or news to optionally personalize the search based on those patterns. In BigTable store, a user was mapped with a userID as a row, every action was stored in a table, and a column family was reserved for each type of action such as web queries. This application generated the user specific data by running MapReduce jobs over BigTable. Then the user could optionally add the column into their profile data. Nonetheless, the particular characteristics of this application is that it used the built-in replication system in the servers hosted, and it limited the quota for sharing profile between users as this action will yield a very big storage demand. Finally, not only did Google expose the uses of BigTable, but they also did advise some practical experiences based on their lessons during the time designing, implementing and maintaining their systems. The first lesson is natural. It is the error-prone property of a distributed system which requires much effort to redesign the protocols to support the fault-tolerance such as checksum for RPC. Another lesson which is originated from this is the necessity of implementing the monitoring system to track down all the components in a distributed environment as they are expanding over time to proactively deliver an intervention as the problem is likely to occur. Additionally, the most important lesson is, the more work and more time it takes to debug it, despite its better efficiency.

## 2.5 Introduction to Accumulo key-value Store

In Accumulo, a key-value pair is called an entry. A contiguous range of sorted keys is called a tablet which is located on a single tablet server, which is an instance in database cluster. While each tablet is assigned to one tablet server, however, each tablet server can contain many tablets, and this design allows for parallel clients to co-request for different queries on different tablets simultaneously. There is also an intuitive illustration on the structure of key-value as shown in the figure 2.1 below [8].

For store on disks, Accumulo splits the tablets into files which is called RFile and stores them in HDFS. There is also another active portion of the tablet, memtable, which will also

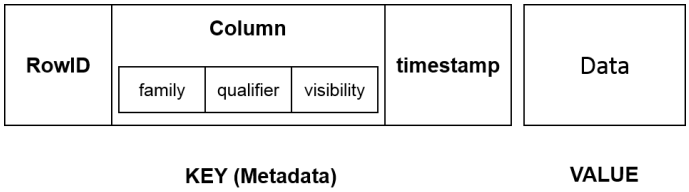



Figure 2.1: Accumulo key-value Pairs

be purged to store as RFile when the threshold is matched. By this way of controlling RFiles, there could be many RFiles on disks, degrading the performance of queries. The Accumulo solves this by periodically performing minor and major compactions to join the RFiles, in order to control the small number of them.

The following content will list out some remarkable functionalities of Accumulo, compared with the others on the same type [8]:

1. **Table Design and Configuration**: Besides server-side level programming mechanisms and cell-level access control, Accumulo does not completely depend on HDFS constraints but now allows the uses of the collection of HDFS URIs (host and path), enabling it to operate over multiple disjoint HDFS instances. Thus the scalability of Accumulo is much better than the predecessors, and in the case of federation, multiple namenodes can share a pool of datanodes.

2. **Integrity/Availability**: Originally, Accumulo has its master as a single point of failure but now it also has multiple masters which can be redundant to each other. If a master processing a table fails in the middle of process, instead of leaving the table inconsistent, the framework will work with lock manager (Zookeeper [20]) to resume the job until it finishes smoothly.

3. **Performance**: Data written is stored outside of Java managed memory into a C++ STL map of maps. The map is natively residing in the memory for faster response time. Upon running a scan, Accumulo also allows for many threads in pipeline manner and thus has a better running performance. Furthermore, scanned data is cached not only

in memory but also in another location for indexes and data (table-level configuration). Another per-table basis configuration is the RFile multi-level index tree which is used to store the last key value in a block. All of these functionalities can yield an impressive performance for Accumulo, as we can see below in benchmarks results in Evaluation section.

4. **Testing**: To write a unit test, people can employ Accumulo Mock implementation of its API, which is in-memory or in-process. Furthermore, to adopt the test to be more realistic, another cluster implementation, which is called Mini Accumulo Cluster, is very useful, even though it is slower. This Mini Cluster can adjoin with Accumulo Maven Plugin to better manage the test's lifecycle under the Maven controlling umbrella. Plus, there are some more pre-built tests such as functional tests, scale tests, and random walk tests which are equally useful for various purposes.

5. **Client API**: There are also a lot of beneficial APIs for users. For reading or writing batch jobs, batch scanners or batch writers can be used. Even more expediently in the case of data ingestion, Accumulo allows the use of Bulk Import for mass sorted key-value chunks to write to Accumulo at the same time, resulting in better performance, despite higher latency in scanning. Normally, Accumulo employs Map Reduce jobs to sort the data prior to this special import. Additionally, the API also provides the use of Thrift Proxy which can wrap the Java-based code with Apache Thrift [21], allowing Accumulo's sophisticated blocks of code which can be used in cross languages such as C++, Python, Ruby, etc.

6. **Internal Data Management**: Accumulo enables a locality group for faster scan which is a set of columns in a single file, and also can be reconfigured while online. Another useful function is that it allows loading Jars with Apache VFS. Finally, Accumulo also supports encryption at rest (RFile and WriteAheadLogs) and at motion (Thrift over SSL) which can increase the level of security.

Conclusively, Accumulo has the advantages of being created later than its pioneers, thus it has some more advanced features over its predecessors and should have its own distinct area of applications. Particularly, Accumulo, by its strong functionality serving for data ingestion and online data management, will be very flexible in the case of ingest or query for a huge amount of diverse data and in the case of dealing with this data.

### 2.6 Accumulo Ingest

Accumulo is equipped with 3 types of ingest:

1. Default ingest using batch writer: the writer acquires enough information about the Key (RowID, Column, Timestamp) and Value before writing into the Tablet sequentially, one row at a time.

2. Bulk Ingest: an improved mechanism which happens in two phases: (1) the whole directory containing the data will be loaded into HDFS; and (2) the import process will load data into table. Obviously prior to the load, the data model for the table should be pre-configured using Accumulo commands such as pre-split. The interesting thing about this is the second phase will be taken place as a MapReduce job with parallelized nature. This explains the fast speed of the operation. For the small files, however, the first operation will also be slow.

3. RFile ingest: One more ingest type which is not well-documented is the direct write to RFile (AccumuloOutputFormat) from HDFS storage without resorting to the use of Batch Writer.

Apart from these options, Accumulo has the following highlights to support ingest process:

**Write-Ahead Log**: Accumulo imitates the design of Google BigTable, in which a Write Ahead Log (WAL) is a log component residing in HDFS and is always available to every

tablet server in case of recovery after a crash. The second data placement is the In-Memory Table which is the main component of the Tablet Server.

**Compaction**: When a prior-set threshold is matched, the data in this memory table will be transformed into an RFile through a process which is called the minor compaction. RFile will be written to HDFS as the last phase of minor compaction.

Moreover, Accumulo will automatically perform the major compaction in which the RFiles will be concatenated to each other to form a single file with the deleted records removed. This will reclaim the fragmented storage inside HDFS and make the records continuous, leading to faster seek and query.

**Iterator Framework**: one big difference between Accumulo and Google BigTable is that Accumulo has the Iterator framework which is flexible for every programmer to insert or adjust the operations at every stage of processes. In detail, the processing phases of Accumulo is modeled in the traditional data structure, namely log-structured merge tree (LSM-tree). This enables Accumulo more flexible table store than Big Table.

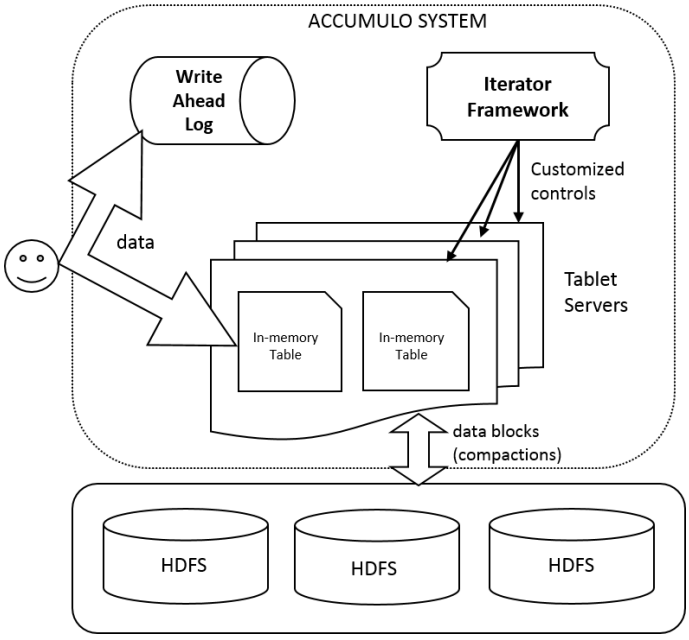



Figure 2.2: Accumulo Ingest Process

The load/ingest process of Accumulo is handled by the following processes (figure 2.2).

1. A client sets up the connection with the Master server which will redirect the connection to tablet servers. All the data reads and writes will be taken place between the client and tablet servers.

2. The client sends data to tablet servers, but data will be written to the Write Ahead Log first.

3. Physical data blocks will be written and managed in HDFS.

The abovementioned content details the background of noSQL system, as well as Accumulo, an emerging open-source, java-based key-value store which is highly regarded as imitating Google BigTable with high-level security and high performance. The following chapter will describe the motivation towards Accumulo and the specific area of study concerning this noSQL system.

# Chapter 3

# Motivation

## 3.1 Movitation of noSQL and Accumulo

There have been some works on optimizing the performance of Hadoop and its use for data analytics. One of which was SFMapReduce [22] in which the authors present a novel framework for both storage and processing levels. By redesigning and reorganizing the metadata and new interfaces for dealing with distributed data, SFMapReduce is able to boost the speed up to 14.5X compared with the original Hadoop without any customization.

Despite the achievement on performance of the Hadoop-based systems such as SFMapReduce framework on IO, the store of data directly in Hadoop is not an ideal choice for applications which require high-level security and interfaces compatible with other applications. Such applications are the flourishing Data Science workloads such as data mining, information retrieval, statistical analysis, machine learning and artificial intelligence, data visualization, etc. noSQL stores, which have been more and more popular, become the state-of-the-art data store for Big Data workloads for its superior in dealing with semi-structured and unstructured data. This document is not an exception to this inevitable shift, aiming to employ a centralized solution for Big Data Management.

Among noSQL, Accumulo key-value store, which has been incubated as an Apache open-source project since 2012, is one of the best key-value stores available for everyone. It has been designed to imitate Google BigTable [9] in Java-style manifestation (while BigTable is mainly C++).

Moreover, compared with its peers, Accumulo was deemed to yield the highest throughput, up to 115 million inserts/s [23]. Moreover, its more granular security feature (down to

ce ll-level) makes it more attractive for a more broad range of applications, especially those which require strict security.

## 3.2 Motivation of study on ingest/load process of Accumulo

As reasoned above, Accumulo and noSQL in general are ideal for dealing with high deluge of data in distributed manner. To adopt new framework as well as technology does not, however, come without obstacles. Among the bounty of such impediments, the issue with storage I/O is obviously the most important because another level of data store acting as the middleware layer of Hadoop HDFS and MapReduce computational framework, with several data management processes, will come with non-negligible overheads. Likewise, how to optimize the middle overhead of the data store itself becomes a key to success in employing it into a real high-performance, distributed data science application.

To investigate the performance of Accumulo compared with other types of workload, I perform a test on throughput of two systems: original Hadoop and Accumulo. Both systems are based on the same eight-node Hadoop clusters, with Accumulo having four tablet servers with one Master. The data are mixed with various size from 100KB up to 20MB, and are grouped into the directory size of 10GB, 20GB, 40GB and 60GB. The chart in Figure 3.1 below shows the average pattern on the performances of those systems.

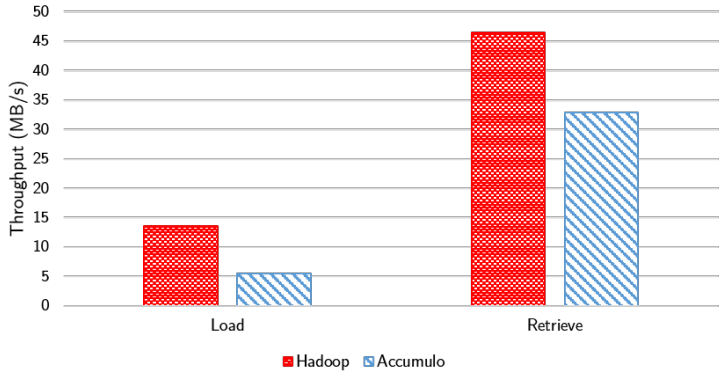



Figure 3.1: Load/Retrieval speeds comparison

Consequently, while the aim of this study is to investigate various strategies for dealing with diverse types of data, the focus will be the method of batch writer, which is the most popular, straightforward and eligible to any case of ingest. The following chapter will introduce different techniques to deal with Batch Writer ingest.

# Chapter 4

# Strategies of Various Ingest Performance

This section describes the basis of Accumulo ingest, followed by various methods for ingest using Batch Writer including the algorithms and characteristics of each one, then details the probable strategy to boost the ingest performance.

## 4.1 Architecture of BatchWriter Ingest

Considering that all the data to be ingested are located in one arbitrary node in an Accumulo cluster, the ingest process is broken down into three phases:

1. Read the data file from a node holding the data using a Batch Writer, a component of Accumulo.

2. Wrap it with Accumulo Key-Value data model which yields a mutation, and put it into a mutation queue to write.

3. Batch Writer writes mutations into tables from the mutation queue.

4. Data from Tablet Servers will then be physically written to HDFS with the number of replicas set by Hadoop configuration

Figure 4.1 below illustrates this architecture. In this figure, data file is a general file to be stored in Accumulo and it could be of any type. For the buffer, users can design any type of buffer available from the Java standard library, e.g. byte buffer. Buffering is a popular yet crucially important method of dealing with streams of data; by using this simple technique, the performance is usually sped up steadily. A mutation is a standard object defined by Accumulo, while a metadata is variant and is customized by users with such information as

file owner, creation time, group, etc. While any type of information is viable, the fields of metadata are strictly compliant to the data model defined by Accumulo key–value pair—this is to ensure the retrieval of data is always possible by any lateral.

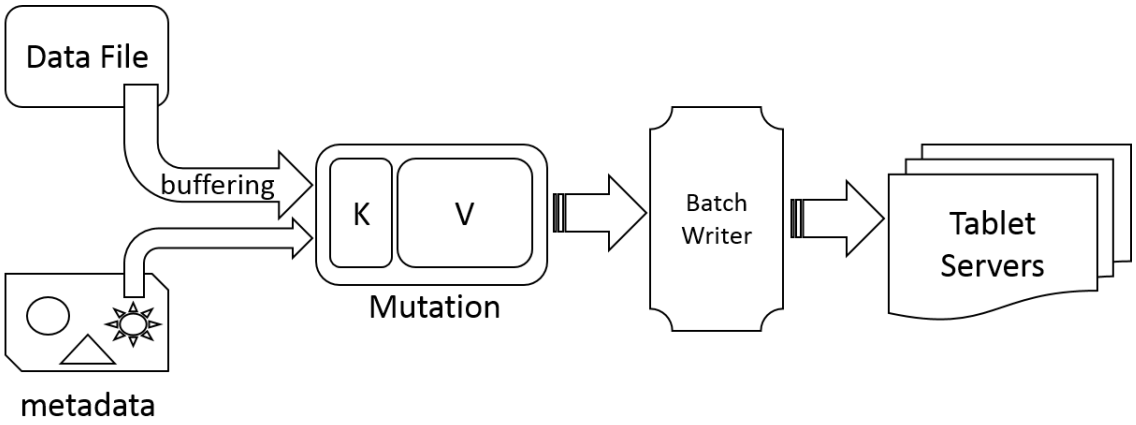



Figure 4.1: BatchWriter ingest

To explain this figure further, there are two very important components, specifically Batch Writer and Tablet Server that are designed to support concurrent operations. Batch writers are located in a cluster node triggering the ingest process while tablet servers are the very basic components of Accumulo on top of the Hadoop cluster. Consequently, they are the fundamental components to focus on in order to boost the ingest operation, of which methods are detailed in the next sections.

## 4.2 Design of various methods of Accumulo Batch Writer Ingest

### 4.2.1 Sequential Ingest

This is a straightforward approach where a batch writer will make an iteration through every file in the feeding directory and apply the processes above to each one on another; the second file will not start until the first one is complete. The illustration of this method is described in figure 4.2 below:

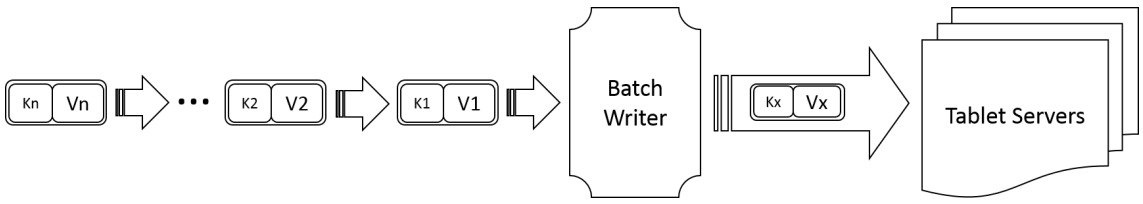



Figure 4.2: Sequential Ingest

The batch writer will take each file sequentially because there is only one thread is active as a whole. On one hand, this approach does not utilize each compute node's resources that become idle in the middle of blocking operations. On the other hand, it cannot take advantage of the concurrent property of batch writers because it only serves the files sequentially.

In short, the algorithm for this approach is as follow:

**Algorithm 1** Sequential Ingest Method

**Input**: BatchWriter configurations, Connector, Input Directory
**Output**: Ingest results
Initialization of BatchWriter using inputs configurations
$directoryLength \leftarrow$ number of files
**for** $i = 1 \rightarrow diretoryLength$ **do**
    **fis** : init a file output stream
    **if** fis == null **then**
        return
    **else**
        initialize a buffer equals to file size
        $buffer \leftarrow$ read continuously the binary content of a file
        initialization of metadata (creation time, owner, columns information, timestamp)
        $mutation \leftarrow$ write all data and metadata
        add mutation into BatchWriter
        close buffer

In this algorithm, the metadata which is freely designed by users makes the governance of the data in Accumulo store more convenient later. This is also important in many cases, like when users want to retrieve the data in a value specified by a logical formula or a range of a particular metadata. As a result, before ingest, users should already have a full picture of the application laying on top of this distributed store, and more importantly, of the possible use cases/queries in order to avoid the never-used metadata.

### 4.2.2 Asynchronous Ingest

If the abovementioned sequential method only employs one thread during processing of the ingest, this asynchronous method takes advantage of concurrent characteristics of Batch Writer to process multiple threads at a same time, as shown in Figure 4.3:

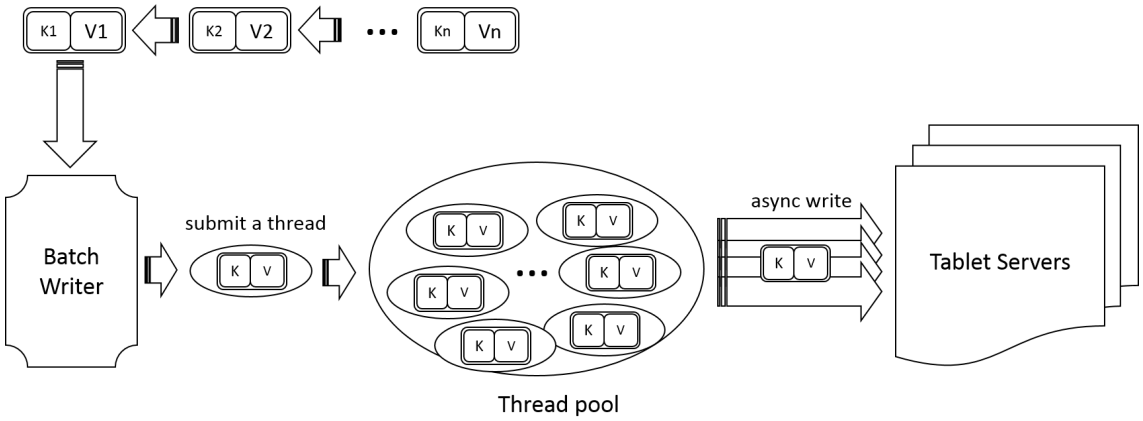



Figure 4.3: Asynchronous Ingest

This approach has advantages over the sequential method as it applies the asynchronous reads and writes of input files by using as many threads as possible to ingest them into tablet servers. Algorithm 2 describes this method in detail.

**Algorithm 2** Asynchronous Ingest Method

1: **Input**: BatchWriter configurations, Connector, Input Directory
2: **Output**: Ingest results
3: Initialization of BatchWriter using inputs configurations
4: Initialization of an Executor Service with a fixed thread pool
5: $directoryLength \leftarrow$ number of files
6: **for** $i = 1 \rightarrow diretoryLength$ **do**
7: initialize an asynchronous file channel for this file
8: initialize a buffer equals to file size
9: read files content from file channel into byte buffer
10: transfer byte buffer to a normal byte array of equal size
11: $mutation \leftarrow$ all data and metadata
12: submit a thread for this file into an executor service
13: close buffer

A key difference of the asynchronous method from the sequential one above is that we can submit multiple threads to a batch writer from users' input and the use of Executor Service which is a tool of Java to handle asynchronous threads. By employing this technique, the compute node's resources are utilized efficiently for the ingest since after reading one file, it does not wait for the corresponding result but rather immediately shifts to process the next one in the loop. To facilitate this method, a byte buffer that holds the data for each file before transferring it to the batch writer should be carefully handled since it is not thread-safe. Because there are now several files that depart to the Batch Writer at the same time, it will also utilize the performance of the Batch Writer efficaciously.

### 4.2.3 Parallel Ingest

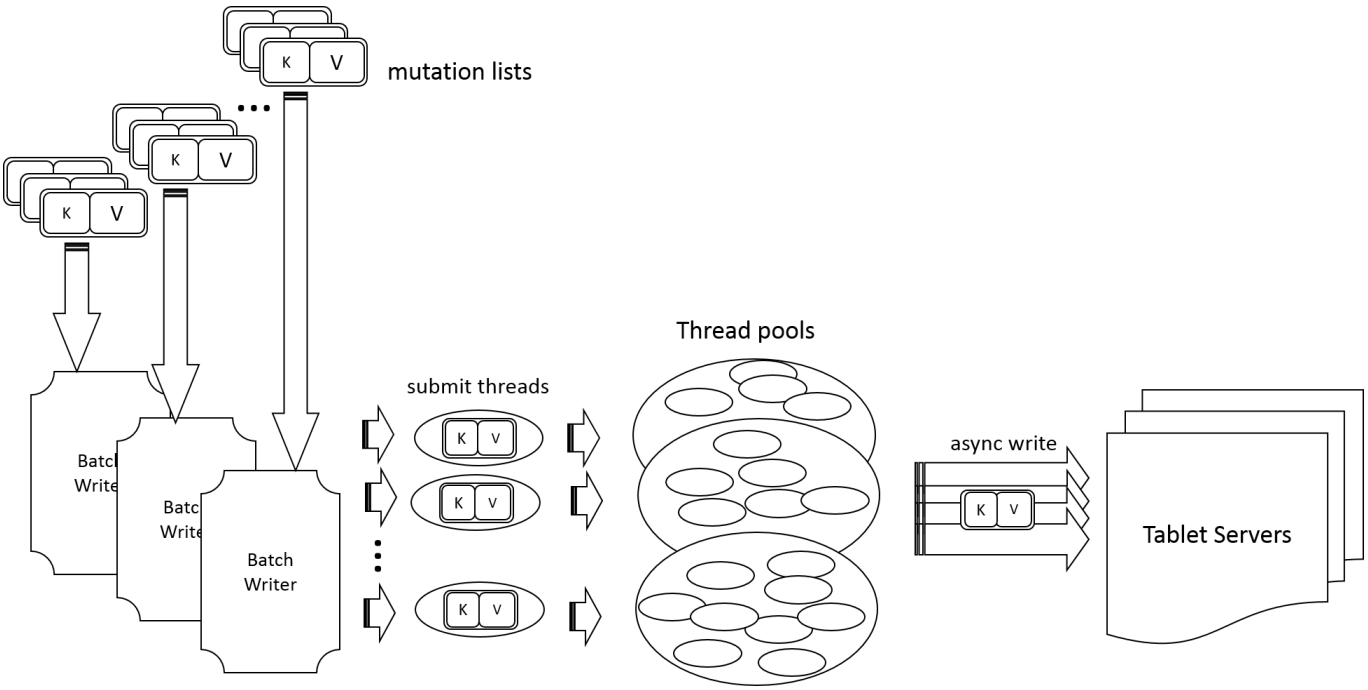



Figure 4.4: Parallel Ingest

The addressed asynchronous method above is a useful approach when dealing with many files while taking advantage of inherent concurrent characteristics of the batch writer. Nonetheless, considering the tablet is the terminal of the data ingested and also supports the concurrent process of Batch Writers, the asynchronous method is by no means optimal.

Therefore, an amended approach to the ones above is to spawn several batch writers at the same time, where each will take an equivalent portion of the data and will co-process the data in a parallel manner. Similarly, this parallel method can be considered a combination of the asynchronous method with multiple Batch Writers. This method is detailed as in Algorithm 3.

**Algorithm 3** Parallel Ingest Method

```
1: Input: BatchWriter configurations, Connector, Input Directory
2: Output: Ingest results
3: Initialization of BatchWriter using inputs configurations
4: Initialization of an Executor Service with a fixed thread pool
5: directoryLength ← number of files
6: runRounds ← number of rounds that all BathWriters will co-process
7: for i = 1 → numRounds do
8:     for j = 1 → numBatchWriters do
9:         for k = 1 → maxfileshandledbyoneBatchWriter do
10:            initialize an asynchronous file channel for this file
11:            initialize a buffer equals to file size
12:            read files content from file channel into byte buffer
13:            transfer byte buffer to a normal byte array of equal size
14:            mutation ← all data and metadata
15:            mutationList ← mutation
16:            submit a thread for this file into an executor service
17:            do an optional flush
18:            close buffer
19:        submit a thread for mutationList into an executor service
```

There are a number of considerations for this ingest strategy, however, that users should take into account carefully. First, the number of Batch Writers can be provided from users' input or set immutable. If the number of Batch Writers is too big, the cluster node handling them might not be able to tolerate the workload, whereas if it is too small then the performance of ingest is not optimized. Second, a batch writer is an object which is optimal for sending mutations to multiple Tablet Servers. Previously, a batch writer would add only one mutation each time. In this parallel ingest method, instead of passing mutations one at a time, we can put the mutations in a list first, then transfer the whole list to the Batch Writer. In this way, we gain some speed acceleration because of the sequence being used

(i.e. by eliminating the number of processes and hence the additional overhead of triggering each process for individual files). And finally, similar to the asynchronous method, the use of a byte buffer should be handled carefully as it is not thread-safe.

By employing this method, we are able to optimize the performance of the nodes ingesting into Accumulo. Nonetheless, the threshold varies by system. Therefore, the number of batch writers working simultaneously and the number of mutations in a list should be determined through a number of scrupulous empirical experiments, to find out the best parameters.

### 4.2.4 Factors Affecting Ingest

There are many factors that can influence the performance the ingest performances including those from within Accumulo, as well as those from external components.

The three ingest methods listed above each includes various techniques to deal with the batch writer, with parallel and distributed approaches to boost the speed of ingest in general. Nonetheless, there could be also a case when users want to simultaneously amass data from various ports into a centralized Accumulo repository—the data to be ingest is not located in one particular node.

In this particular scenario, using one Batch Writer at one node is exceedingly expensive in terms of network operations. Conversely, because tablet servers are designed to optimally handle asynchronous requests from clients, a more natural approach would be to employ each batch writer at every node and have them operated concurrently. The rest of the data will be digested appropriately by each tablet server.

Besides the concurrent operations, there is one more factor to consider, which is the network. Accumulo shares a common architecture with any database management system, in that there is always a special component resident in memory holding the indexing metadata for fast queries and processing. There is only one location for this metadata, which is the master node. For small files, there would obviously much more metadata than for larger files,

despite the fact that the total amount of storage space utilized may be the same. As a result, if the data is located at the master node, the ingest process should be very fast, because following the creation of the index metadata, there will be no further network operation involved. While in all other nodes other than master node, all of the big metadata from the nodes will be sent through the network to the master node simultaneously. Because the master node has to frequently handle many other processes and operations, its performance will be commensurately degraded based on the number of data locations.

For a similar reason, in the case of centralized data, it would be preferable that the data to be ingested is kept in the master node rather than the slave nodes, in effect to avoid unnecessary network operations.

Additionally, there are many internal configurations that also affect ingest to Accumulo, including compactions (minor and major), Write Ahead Log modes, file size, pre-split before ingest, number of splits, number of threads for batch writers, etc.

The following section will explain the evaluations of the three ingest methods mentioned and the effects of various factors on ingest performance.

# Chapter 5

# Evaluation

## 5.1 Experimental Environment

### 5.1.1 Cluster Setup

There are eleven nodes in the testing cluster. Each has a 2.67GHz 20-core Intel Xeon CPU, 130GB memory with sufficient hard disks. The machines are connected through 10 Gigabits Ethernet. All hardware boxes are homogeneous, including the operating system which is CentOS 6.5 x64.

### 5.1.2 Hadoop Setup

The system used for evaluation comprises of one master node with ten slave nodes, all running Apache Hadoop v2.6 on Java 7. The configurations of Hadoop are detailed in the table 5.1.

Table 5.1: Hadoop configurations

| **Parameter Name** | **Value** |
|---|---|
| yarn.nodemanager.resource.memory-mb | 16 GB |
| yarn.scheduler.maximum-allocation-mb | 6 GB |
| yarn.scheduler.minimum-allocation-mb | 1 GB |
| mapReduce.map.java.opts | 3 GB |
| mapReduce.reduce.java.opts | 4 GB |
| dfs.block.size | 128 MB |
| dfs.replication | 3 |

### 5.1.3 Accumulo Setup

The Accumulo system is built on top of Hadoop which has one master node and four slaves nodes (four tablet servers). For evaluation, Accumulo version 1.6.5 is used on top of Hadoop and Zookeeper. The parameters of Accumulo are detailed in Table 5.2 below.

Table 5.2: Accumulo configurations

| **Parameter Name** | **Value** |
|---|---|
| tserver.cache.data.size | 512 MB |
| tserver.cache.index.size | 1024 MB |
| tserver.mutation.queue.max | 4 M |
| tserver.sort.buffer.size | 1024 M |
| tserver.memory.maps.max | 4 GB |
| tserver.walog.max.size | 1 GB |

### 5.1.4 Data for Evaluation

Throughout the tests, we group files that have various sizes into a directory of 3GB, 6GB, 12GB and 24GB. The data is randomly generated using Linux tools (such as ***dd***) and if not clearly described, the data is—by default—mixed with many sizes from 15KB to 100MB each.

## 5.2 Performances of Various Ingest Methods

To begin the experiments, I explore the different performances of the three different ingest methods above: sequential, asynchronous and parallel ingest. To maintain the objectiveness of the results, no other configurations are altered except for the abovementioned parameters. The experimental result is demonstrated in Figure 5.1.

According to this chart, we can easily recognize the differencr between the parallel ingest (with four batch writers) compared with the other two, in which the parallel method outperforms the others of by least 50% faster. To explain, by using multiple batch writers, the parallel ingest has forced the Accumulo store to connect to different tablet servers and

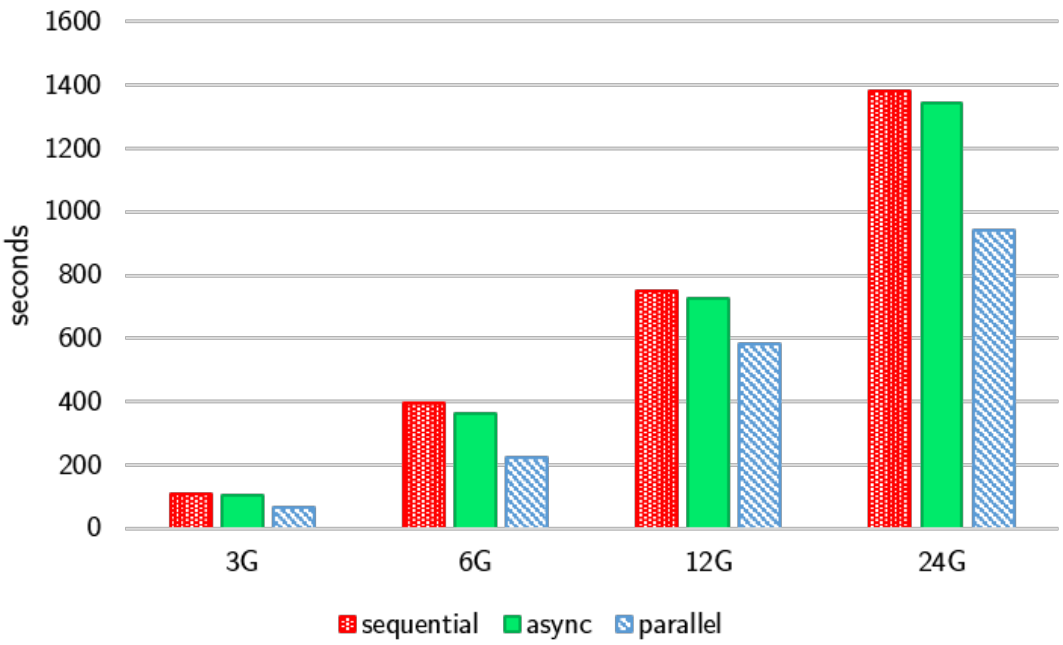



Figure 5.1: Performances of Different Ingest Strategies

use them efficiently. Conversely, because we retain the basic configurations, the sequential ingest is inefficient becuase it does not use all four tablet servers ub the process. Instead, because there is no fixed configurations, every mutation is sent to a random single tablet until a migration is triggered to balance the data consumed. To consider asynchronous ingest, particularly, by having a thread pool to handle multiple concurrent processes, this method supposes to outperform the sequential one. Nonetheless, due to having just one batch writer handling the mutations in sequential order, this ingest process is actually sequential in the case of no split is defined from the beginning.

To illustrate the resources consumption's difference, four indicators have been measured, including CPU, memory, network and IO for all of the cluster. This includes the Accumulo master, four tablet servers and six other Hadoop slave nodes. The results are represented in three following chart groups.

First, for sequential ingest, the utilization of resources is demonstrated in Figure 5.2. In this figure, although Accumulo master is certainly busy throughout the whole process, only

Tablet Server 4 has been involved and has the corresponding utilization statistics while the other three tablet servers are idle.

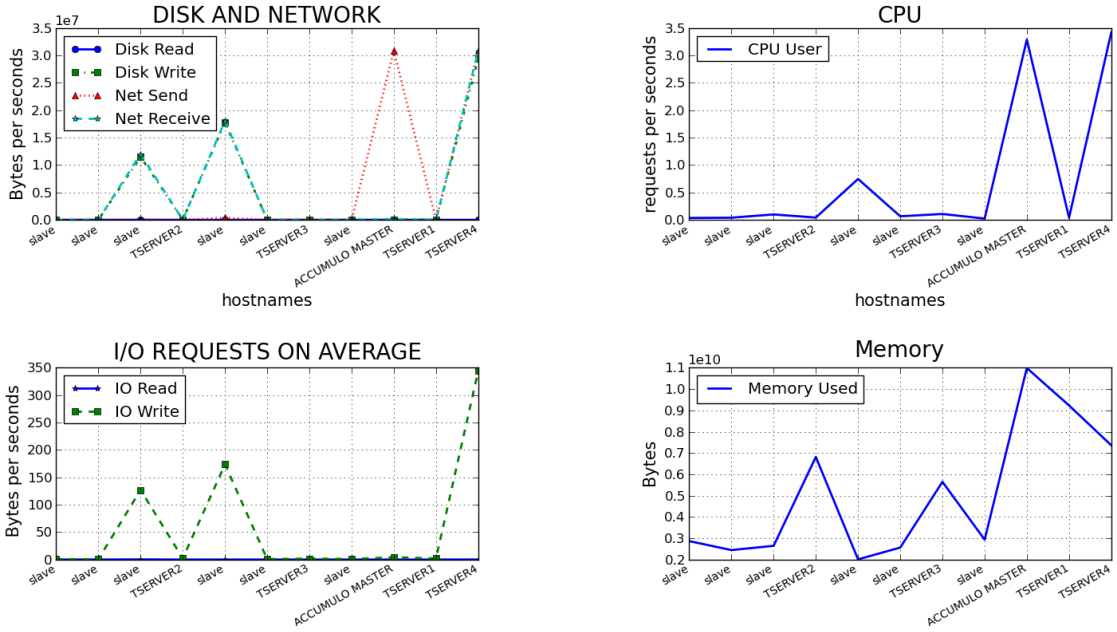



Figure 5.2: Resources Utilization of Sequential Ingest

Second, for asynchronous ingest, Tablet Server 2 has the highest utilization, followed by Tablet Server 1 and then 4, while Tablet Server 3 is idle, as shown in Figure 5.3. Moreover, the CPU requests are much higher for the involved tablet servers.

Finally, the parallel ingest method has pushed the four tablet servers to function, with the leading of Tablet Servers 1 and 2, followed by 4 and 3. Compared with the asynchronous method, this one has higher utilization on each tablet server involved (especially with network, disk, IO and memory) though having the similar idle state. We can see the utilization statistics for the parallel ingest strategy in Figure 5.4.

After all, one limitation for all the methods leading to idle tablet servers is that they have no knowledge about splits prior to the ingest; thus the ingest is skewed across tablet servers, regardless the method employed.

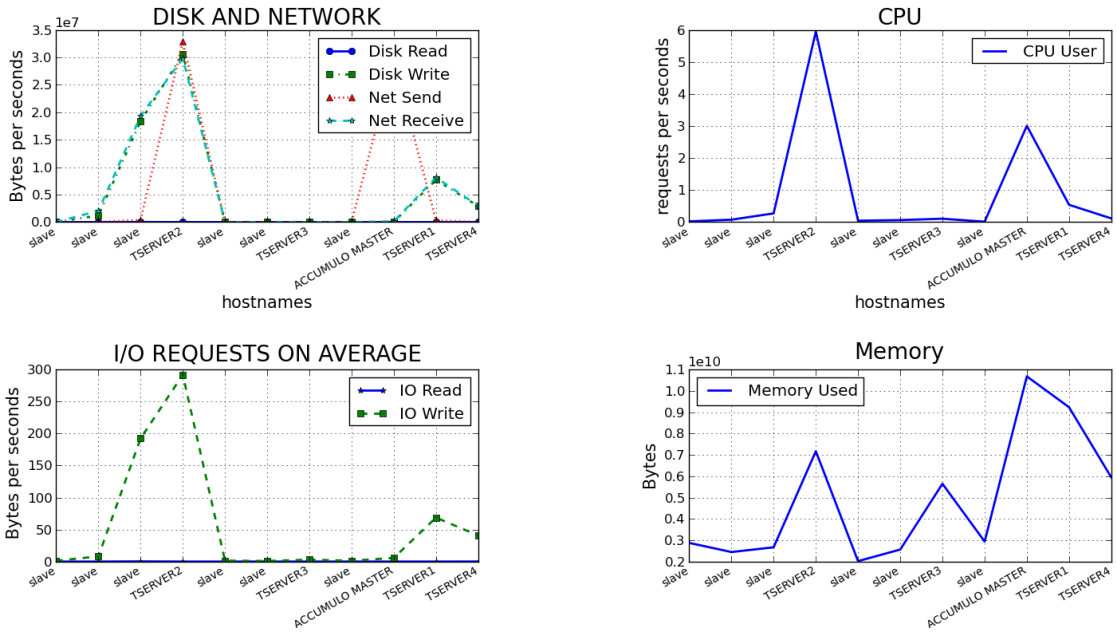



Figure 5.3: Resources Utilization of Asynchronous Ingest

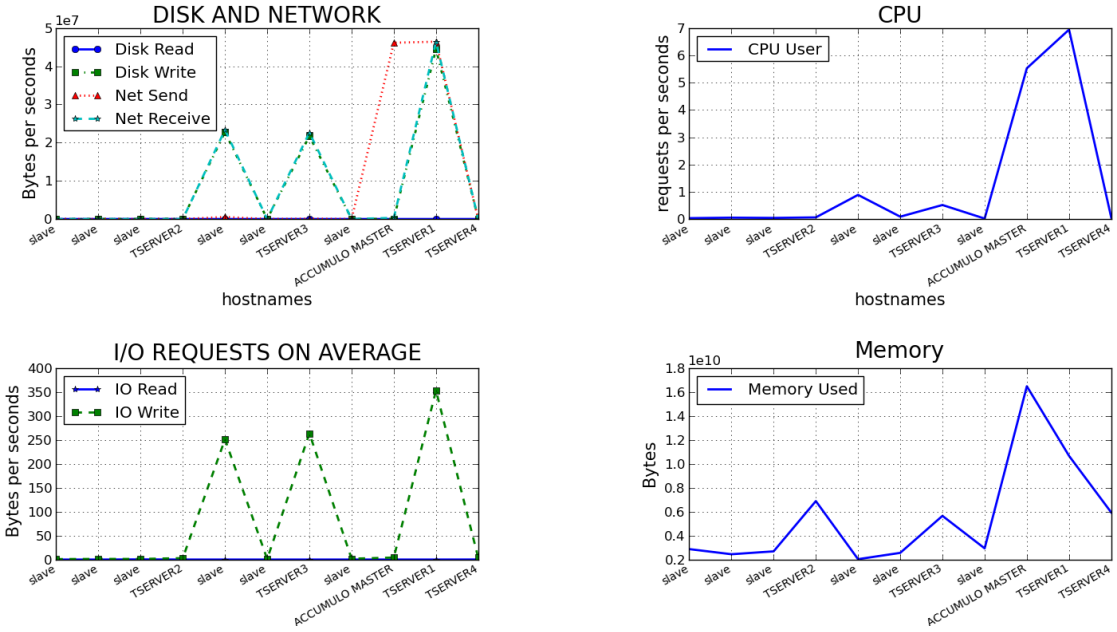


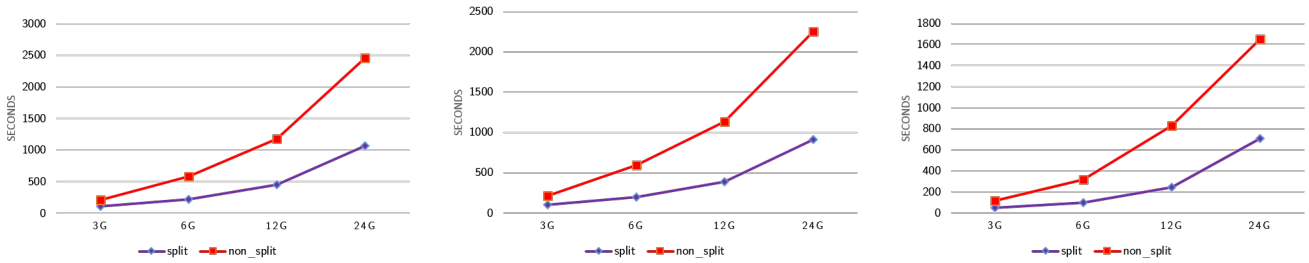

Figure 5.4: Resources Utilization of Parallel Ingest

## 5.3 Implication of Splits to Ingest

Since the model of the data to be ingested is controlled completely by users, particularly the metadata, the Accumulo master or all of the tablet servers have no prior knowledge about

how to balance the data while it is being ingested. As a result, pre-splitting the data at the very beginning is crucially important to the speed of ingest.

The effects of pre-split is shown in Figure 5.5




(a) Sequential Ingest (b) Asynchronous Ingest (c) Parallel Ingest

Figure 5.5: Implications of Pre-Splits

By default, the migration process will start to balance the data while ingest is being performed as soon as the ingested data reaches 1GB each time. As a result, if we don't change this configuration, the best way to ingest is to pre-split to the corresponding number of splits to avoid the delay of migration. For example, if we are to ingest 10GB, then we create a split file containing 10 splits and apply them to the respective table before the actual ingest process begins.

As we can see above, regardless of the ingest method used, pre-splitting boosts the ingest time significantly. And the improvement grows as the data size increases.

For resources utilization, Figure 5.6 shows the statistics of parallel ingest in the case of pre-splitting. In this figure, all the tablet servers are almost equally involved in the ingest process with similar consumption of hardware resources, in contrast to the case of no pre-splitting as seen in figure 5.4.

To conclude, pre-splitting is very important to the ingest process. Because of its huge significance, every ingest should apply this method properly in accordance with the data being ingested. This will achieve the best resources utilization and thus the best performance of the whole Accumulo system.

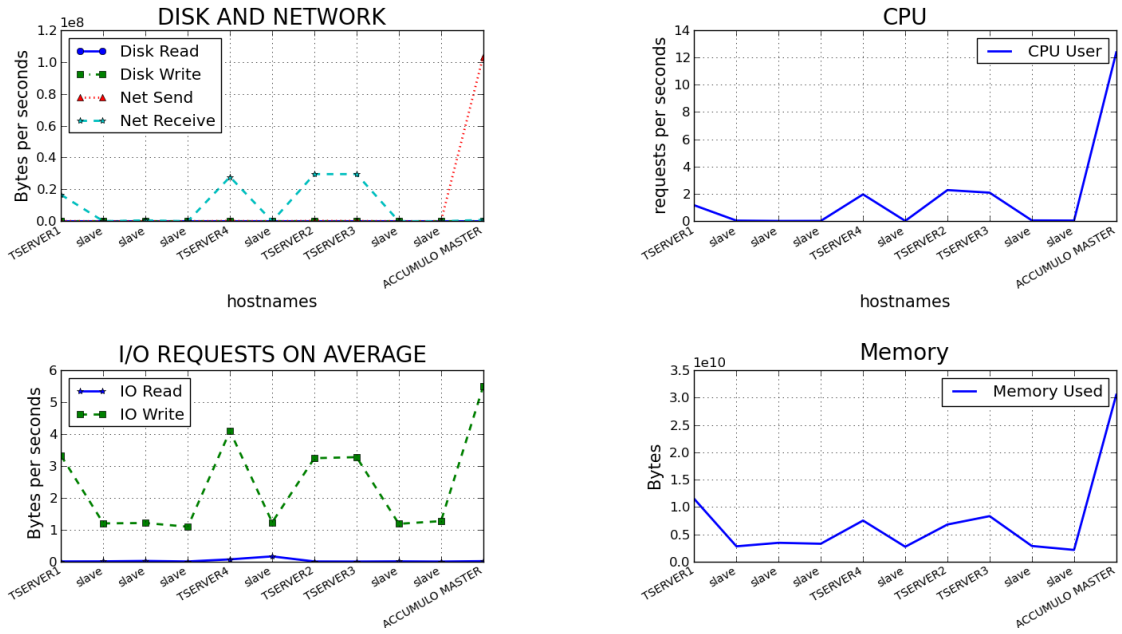



Figure 5.6: Resources Utilization of Parallel Ingest (pre-split case)

## 5.4 Implication of Write-Ahead Log Mode to Ingest

Write-Ahead Log (WAL) is a key mechanism in any database or operating system to maintain the recoverability of the whole system against any unexpected failure. This protection mechanism is thus essential, however, the tradeoff is that it introduces an obstacle to the high performance of system. Consequently, dealing with WAL properly will also boost up the speed of ingest.

Accumulo is equipped with three modes of WAL:

1. WAL sync mode: The best protection mode in which all the data blocks are committed successfully to HDFS storage at a physical level before the next key-value data.

2. WAL flush mode: To eliminate the latency stemmed from the commit to HDFS in the sync mode, this flush mode improves the speed by consecutively writing the data into the system without the need to wait for results returned by storing each row. While it expedites the operations, the system cannot be fully secured against any failure.

3. WAL disabled mode: If the flush mode still needs the late results of data being stored into HDFS, in this mode, WAL is completely deactivated, leaving the system unprotected under any failure, leading to complete data loss for any rows being processed at the time of failure. In return, the overhead of WAL is entirely removed.

Figures 5.7, 5.8 and 5.9 illustrate the results of the three WAL modes applied to each of the ingest strategies: sequential, asynchronous and parallel.

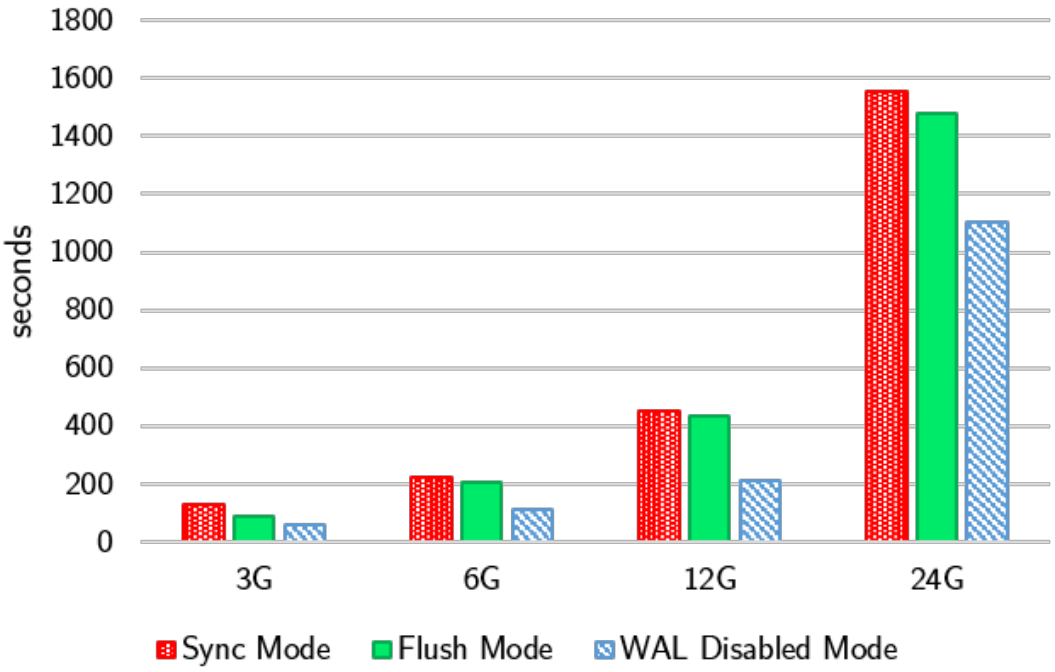



Figure 5.7: Different WAL Modes with Sequential Ingest

Conclusively, no matter which method is used for ingest, the WAL mode is a non-negligible factor largely affecting the speed. As a result, if speed is the first priority, users can choose flush mode, a compromise between the sync mode and the disabled mode, or even move further by employing WAL disabled mode to achieve the best performance.

## 5.5 Implication of Number of Batch Writer Threads on Ingest

Batch Writer in Accumulo has many configurations in which the number of threads has a direct influence on how it works in the ingest process. Conceptually, the more threads

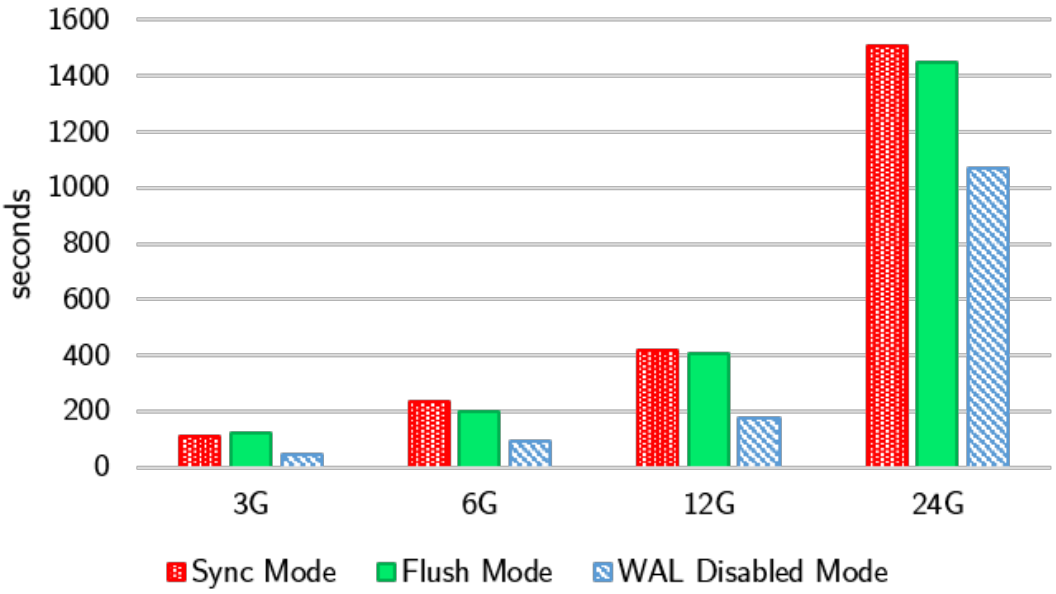



Figure 5.8: Different WAL Modes with Asynchronous Ingest

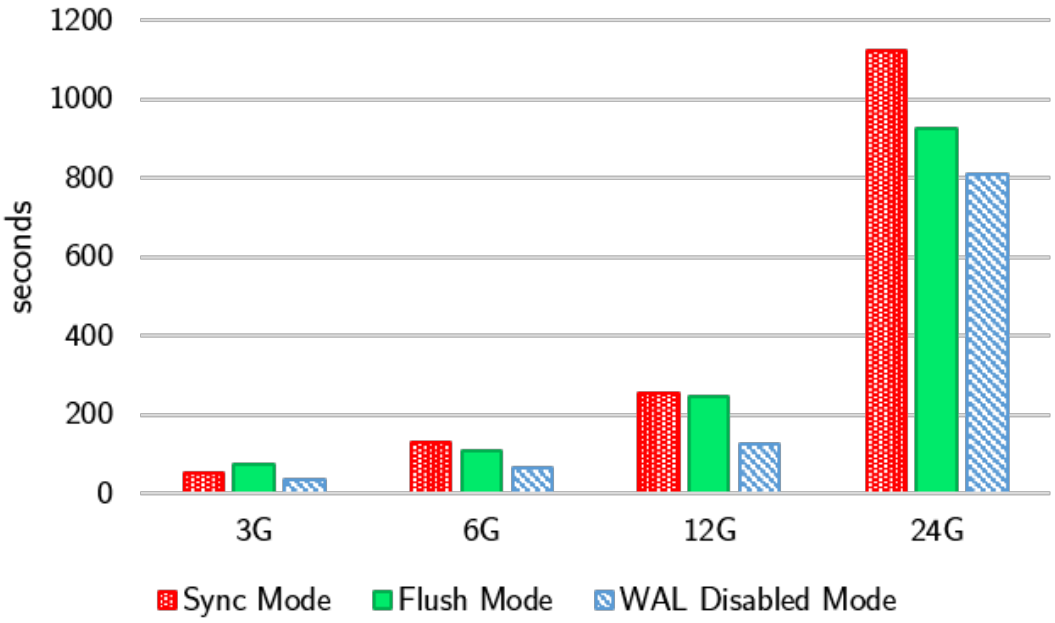
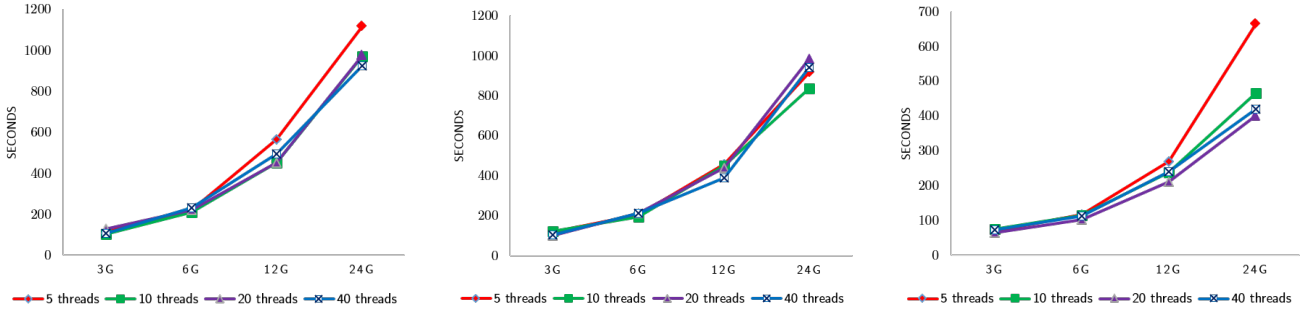



Figure 5.9: Different WAL Modes with Parallel Ingest

we have, the faster the ingest speed is. For evaluation, the bath writer will work with 5, 10, 20 and 40 threads with each of the three ingest strategies: sequential, asynchronous and parallel (with four batch writers). And the result is shown in Figure 5.10.




(a) Sequential Ingest (b) Asynchronous Ingest (c) Parallel Ingest

Figure 5.10: Implications of Number of Batch Writer Threads

According to this result, the number of threads has little implication on the performance of ingest. Except for the case of five threads in which the performance is slightly slower. For the other cases, there is no big difference on the ingest speed. The reason is that when we have too many threads, due to hardware limitation (speed of network, overhead of distributed communications and coherence for both Accumulo and HDFS, etc.), Accumulo is 'soaked' very early, and greater number of threads will not improve the efficiency. Moreover, the factor of parallelization does not come without an upper boundary. If we are to allot too many threads, Java heap will not be able to tolerate the workload; and an exception or failure will follow. In practice, consequently, we should handle the number of threads with great care.

## 5.6 Implication of File Size on Ingest

HDFS is deemed the most reliable open-source storage system for distributed environment and is very widely used. Nonetheless, HDFS is not designed to deal with small files because its default block size is 128MB. On the other side, Accumulo is a key-value store system that lies on top of Hadoop. Figure 5.11 demonstrates the result of the investigation of ingest with a file size parameter from 200KB to 102.4MB (size doubles at each step, the total size is 6GB) and with the three ingest strategies of sequential, asynchronous and parallel.

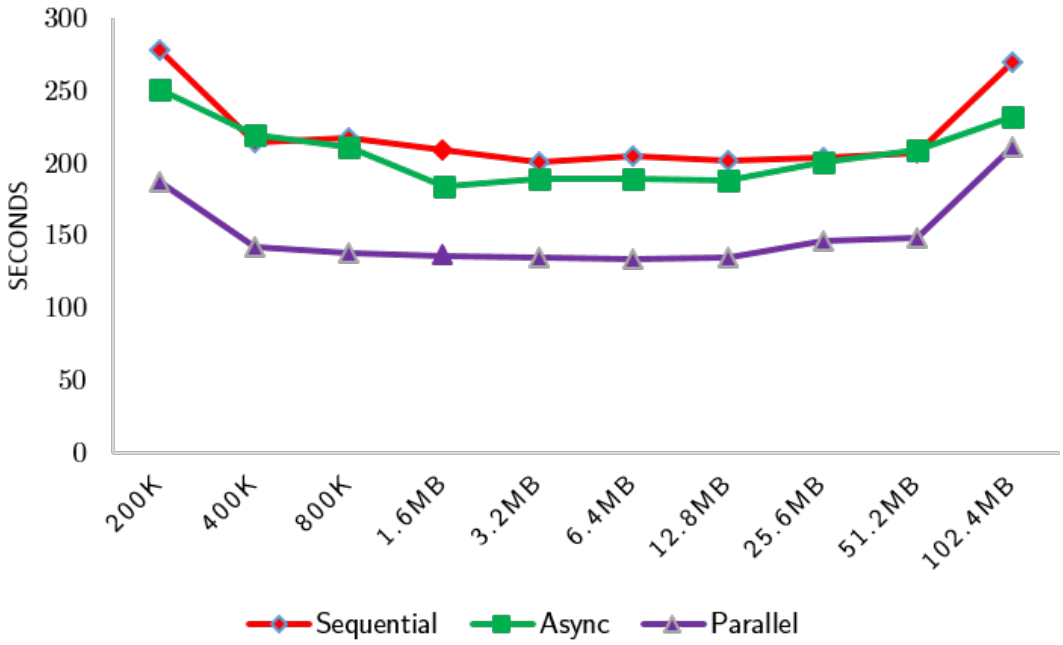



Figure 5.11: Performance of Ingest with Different File Sizes

According to this result, the ingest is faster and faster as each file size increases from 200KB to 1.6MB, and there is not much difference from other sizes (actually, there is a slight fluctuation) until the size of 102.4MB, where the ingest is relatively slower than other previous sizes. Regardless HDFS conceptually favors this size the most among the tested sizes (since HDFS blocksize is 128MB).

To reason, since we cannot ignore the fact that Hadoop is the underpinning layer, and it uses some buffering process, it is not easy to estimate the ingest performance with respect to file size. Moreover, Accumulo is a complicated system which has its own cache, index and flush scheme which are other obstacles for estimation. For example, for smaller file sizes, it is very fast to write mutations into buffers and then memory; however, with an equal total size, the number of these small files is very large, leading to longer metadata processing and also slower write to HDFS which favors big files. And as we hardly analyze this complexity for the most correct answer to the ingest performance on file size, we might have a premilinary explanation: for smaller size, the metadata processing is bigger, and HDFS unfavors those

sizes thus the ingest is slow; whereas, for very big sizes, the buffer processing is not so efficient which also degrades ingest speed.

### 5.7 Other Factors Affecting Ingest

Besides the parameters above, there are other ones that have influence on ingest. First, minor and major compactions can decrease the speed. Minor compaction is a flush operation of rows from memory to the respective tables, which will be eventually written into RFiles on Accumulo. In contrast, major compaction is carried out periodically in the background to chunk up the RFiles on disk within a tablet to facilitate the retrieval methods such as query process of continuous rows.

Based on the characteristics of minor compaction, there is an alterable configuration, namely ***tserver.memory.maps.max*** which has the default size of 1GB, to deal with this operation. In detail, if we do not want any minor compaction to happen during ingest, we increase the size of this configuration up to at least equal to the data ingested. If the data is too big compared with the available memory resource, just increase this configuration, e.g. to 5GB, so that the frequency of flush operation is decreased. Similarly, we would want major compaction to be least involved in ingest, and we can schedule it to take place later by the simple shell command ***compact -t tablename***.

There is also a bunch of other parameters that would be able to have influence on ingest which can be configured and altered in Accumulo configurations such as balancer, compress, tablet server's cache size, index size, memory map and many other configurations of HDFS and Accumulo.

# Chapter 6

# Related Work

There have not been many studies on Accumulo compared with its peers such as MongoDB, HBase, Cassandra, and others. Nonetheless, some of the announced studies have been successful in providing insights about Accumulo and how it performs compared with its peers.

Sen *et al* [19] gave us some initialized evaluations about Accumulo performance, which were Ingest test, Random Walker test, Batch Walker test, and Cell Security test. For ingest on pre-split table: by ingesting a table with some hundreds of billion to trillion entries, Accumulo performed rather well with the rate of 37M ingest/s in 300 nodes, 46M in 500 nodes, and up to 108M in 1000 nodes. In the other case of non-pre-split of 300, 500, and 100 cluster nodes, the ingest rate was then, on average, at 5.14M, 2.14M and 5.5M ingest/s, respectively.

In terms of benchmarks which are usually used to measure the performance of ingest or other operations, Cooper *et al* [24] created YCSB which is now widely used for many noSQL systems but originally not designed for Accumulo. Later, Patil *et al* [25] developed an extension which allowed for Accumulo benchmarks, from which the performance of Accumulo is better than HBase.

MIT Lincoln Laboratory is another reputed institution also specializing in Accumulo to serve their Big Data Framework, which is called Dynamic Distributed Dimensional Data Model (D4M) [26, 27]. The result speculated by this lab is impressive. Using just three and seven tablet servers, the ingest rate can be up to 800K and 1.5M per second. In another paper [28], they compare the ingestion rate with that of Cassandra and HBase, and proved that Accumulo has the far better performance: 4M entries/s compared with 35K

entries/s of Cassandra and 60K entries/s of HBase. These results show that Accumulo is a rational choice for not only security-aware applications but also heavy load ones. However, to my understanding, there has been no formal and complete study about various ingest on Accumulo. My study, as a result, supplements an additional piece of knowledge about Accumulo ingest performance.

# Chapter 7

## Conclusion and Future Work

noSQL is fast growing as it is replacing some traditional RDBMS in the era of Big Data. Nonetheless, since there are many types, along with many products, of noSQL systems, to pick the most appropriate solution is by no means an easy task, given the lightning fast pace of technological evolution and the need of skills for maintenance and development of these systems and surrounding applications.

Furthermore, noSQL was born after the well-established RDBMS, and thus it can hardly avoid many downfalls needing to be altered over time. This thesis has provided some fundamental background on these noSQL systems, focusing on a key-value store representative, namely Accumulo, which imitates one of the most famous system in the world–the Google BigTable.

Additionally, this thesis has identified the need for a fast interface for data ingest, which is a matter of essence because, in dealing with a deluge of data, the distributed systems themselves (HDFS or Accumulo) are scalable and elastic, the ingest should not become a bottleneck of the whole system. This problem is more and more crucial as, nowadays, the demands of online data analytics have emerged at an unprecedented pace.

Along with that motivation, this thesis has supplied many test cases with explanation on Accumulo behaviors in regards to various parameters affecting ingest, and the followed practical recommendations.

In the future, research needs to improve by amending the ingest methods and carrying out investigations on more parameters to make the analysis more rounded and reliable.

Bibliography


[1] Unstructured data, Wikipedia, http://en.wikipedia.org/wiki/Unstructured_data.

[2] Dean, Jeffrey and Ghemawat, Sanjay, "MapReduce: simplified data processing on large clusters," Communications of the ACM. vol. 51, no. 1, pp. 107-113, 2008.

[3] Apache Hadoop, https://hadoop.apache.org/.

[4] Data Science, Wikipedia, https://en.wikipedia.org/wiki/Data_science.

[5] A B M Moniruzzaman, Syed Akhter Hossain, "noSQL Database: New Era of Databases for Big Data Analytics-Classification, Characteristics and Comparison,"International Journal of Database Theory and Application, Vol. 6, No. 4, 2013.

[6] Apache Pig, http://pig.apache.org.

[7] Apache Hive, http://hive.apache.org.

[8] Apache Accumulo, http://accumulo.apache.org.

[9] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach,D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R.E. 2006, "Bigtable: a distributed storage system for structured data," In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (Seattle, WA, November 06 - 08, 2006). USENIX Association, Berkeley, CA, 15-15.

[10] Apache Cassandra, http://cassandra.apache.org.

[11] Apache HBase, http://hbase.apache.org.

[12] Weber, Zachary, and Vijay Gadepally. "Using 3D Printing to Visualize Social Media Big Data," IEEE HPEC, 2014.

[13] Jeremy Kepner, Vijay Gadepally, Peter Michaleas, Nabil Schear, Mayank Varia, Arkady Yerukhimovich, and Robert K. Cunningham. "Computing on masked data: a high performance method for improving big data veracity," CoRR, 2014.

[14] J. Kepner, D. Ricke, and D. Hutchinson. "Taming Biological Big Data with D4M," Lincoln Laboratory Journal, Vol 20, No 1, 2013.

[15] Cattell, R. "Scalable SQL and NoSQL Data Stores," SIGMOD Record, Vol 39, No 4, 2010.

[16] ACID, Wikipedia, http://en.wikipedia.org/wiki/AICD.

[17] Brewer,E, "CAP Twelve years later: How the rules have changed," IEEE computer, Feb. 2012.

[18] Stonebraker, M. "SQL databases v. NoSQL databases," Communications of the ACM, Vol 53, No 4, 2010.

[19] R. Sen, A. Farris, and P. Guerra, "Benchmarking apache accumulo bigdata distributed able store using its continuous test suite," in IEEE International Congress on Big Data, 2013, pp. 334341.

[20] Apache Zookeeper, http://zookeeper.apache.org.

[21] Apache Thrift, http://thrift.apache.org.

[22] Fang Zhou, Hai Pham, Jianhui Yue, Hao Zou and Weikuan Yu, "SFMapReduce: An Optimized MapReduce Framework for Small Files. IEEE International Conference on Network, Architecture and Storage (NAS)," August 2015, Boston, MA.

[23] Kerper et al, "Achieving 100,000,000 database inserts per second using Accumulo and D4M," arXiv preprint arXiv:1406.4923. 2014.

[24] Cooper, Brian F., et al, "Benchmarking cloud serving systems with YCSB," Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010.

[25] Patil, Swapnil, et al, "YCSB++: benchmarking and performance debugging advanced features in scalable table stores," Proceedings of the 2nd ACM Symposium on Cloud Computing. ACM, 2011.

[26] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Dynamic Distributed Dimensional Data Model (D4M) Database and Computation System," ICASSP 2012.

[27] C. Byun, W. Arcand, D. Bestor, B. Bergeron, M. Hubbell, J. Kepner, A. McCabe, P. Michaleas, J. Mullen, D. O'Gwynn, A. Prout, A. Reuther, A. Rosa and C. Yee, "Driving Big Data with Big Compute," IEEE HPEC, Sep 10 - 12, 2012.

[28] Kepner, Jeremy, et al, "D4M 2.0 schema: A general purpose high performance schema for the Accumulo database," High Performance Extreme Computing Conference (HPEC), 2013 IEEE. IEEE, 2013.