UPPSALA
UNIVERSITET

IT 16 027
Examensarbete 30 hp
Juni 2016

A Performance Comparison of SQL and NoSQL Databases for Large Scale Analysis of Persistent Logs

ABDULLAH HAMED AL HINAI

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

A Performance Comparison of SQL and NoSQL Databases for Large Scale Analysis of Persistent Logs

ABDULLAH HAMED AL HINAI

Recently, non-relational database systems known as NoSQL have emerged as alternative platforms to store, load and analyze Big Data. Most NoSQL systems, such as MongoDB, Redis, HBase, and Cassandra sacrifice consistency for scalability which means that users may not be able to retrieve the latest changes in the data but can execute faster queries. Alternatively, these systems provide what is known as eventual data consistency. Similarly, relational database systems allow relaxed levels of consistency to obtain performance improvements. In this master thesis project previous performance and scalability benchmarking experiments are reproduced and extended to two new popular state-of-the-art NoSQL database systems: Cassandra and Redis. Additionally, a relational database system not used in previous research was tested in this project, in addition to a new release of an already-tested open source relational system. The purpose of these experiments is to extend the previous evaluation to two relational systems and two non-relational database systems regardless of their data model by measuring the time needed to load and query persistent logs under different indexing and consistency settings. The results of this research show that there is no specific type of system consistently outperforming the others but the best option can vary depending on the features of the data, the type of query and the specific system.

Handledare: Khalid Mahmood
Ämnesgranskare: Matteo Magnani
Examinator: Edith Ngai
IT 16 027
Tryckt av: Reprocentralen ITC

Acknowledgments

First of all, I would like to extend my heartfelt thanks to my beloved wife Amani Al Khaifi, who tolerated me without growl to spend countless weekdays, weekends and evenings, over the last year in this research project, while she was taking care of the fruits of our love, our kids, and at the same time working hard in her PhD research. My warm thanks are also extended to my lovely daughter and son, Shayma and Hamed, for their patience about my preoccupation during the last period, even during the official vacations and events. Love, appreciation and thanks to my parents who have been patient with me when I have been far away from them and my home country Oman, busy in my master study.

I would also thank Uppsala University represented by the Department of Information Technology, which gave me the opportunity to conduct my master degree, my reviewer and the examiner at this department.

Finally, many thanks to my employer Sultan Qaboos University (SQU), Oman, which provided me with a study leave to complete my higher education and gain study experience in Sweden.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AOF	Append Only File
CAP	Consistency Availability Partition-tolerance
CSV	Comma Separated Values
CCM	Cassandra Cluster Manage
CA	Cassandra Single Node
CA-SH	Cassandra with Sharding
CA-SH-R3-W	Cassandra with Sharding, Replica 3, Weak Consistency
CA-SH-R3-S	Cassandra with Sharding, Replica 3, Strong Consistency
CQL	Cassandra Query Language
DBMS	Database Management System
DB-C	DataBase Commercial
DB-O	DataBase Open Source
NoSQL	Not-only SQL
SQL	Structured Query Language
RAM	Random Access Memory
RDBMS	Relational Database Management System
RDB	Redis DataBase File
SSTable	Sorted String Table
UDBL	Uppsala DataBase Laboratory

1- Introduction

Relational Database Management Systems (RDBMSs) can be used to efficiently store and query large amounts of data. However, the performance of an RDBMS can be negatively affected by the requirement of *full transactional consistency*, where a set of properties known as ACID (for: atomicity, consistency, isolation and durability) are guaranteed by the system. In contrast to RDBMSs, non-relational data stores (NoSQL) are often designed to allow only what is known as *eventual consistency* to further improve scalability and performance.

A comparison between the performance of RDBMSs and NoSQL database systems was conducted in [2], where *relaxed consistency* was used in the relational systems to decrease overhead. This overhead was previously found to be evenly divided among four components of a RDBMS: logging, locking, latching, and buffer management [3]. However, whether the performance for persisting and querying logs could be enhanced by utilizing a weaker consistency model had not been explored before.

RDBMSs have been used for more than four decades and developers are accustomed to their keywords, simple query languages and indexing strategies. However, many state-of-the-art NoSQL datastores were designed to use similar keywords and query languages in order to achieve the popularity of RDBMS. These systems are also known as SQL-like query language datastores. In addition to their SQL-like syntax, they typically support new indexing strategies, as in the case of Cassandra [4].

The purpose of the current project was to define a benchmark for the performance evaluation of basic queries over historical data, in particular log databases. Moreover, it aimed at exploring the impact of various indexing schemes and query execution strategies among different consistency levels with two relational storages (SQL) and two non-relational (NoSQL) databases for scalable loading and analysis of persistent logs. The experiments included several database engines: two major RDBMS systems, Cassandra [4], and Redis [5]. This comparison was an extension of a research that was conducted within the Uppsala Database Laboratory (UDBL) [1].

The following factors that can influence the performance of bulk loading, querying, and analyzing persistent logs were investigated in this project:

1. Indexing strategies (i.e. primary and secondary index utilization) .
2. Relaxing consistency.
3. Data parallelization over multiple nodes (manual or auto-sharding).

The expexted result of the project was a benchmark including basic queries for accessing and analyzing persisted data. The properties of these queries are: basic selection, range search and

aggregation. This thesis also presents and discusses the results of the benchmark applied to the aforementioned systems.

2- Background

This section introduces the importance of Database Management Systems for computer applications and the scenario of a persisting logs application from a real industrial world. Moreover, it provides an overview about databases transactions, indexing strategies, consisting levels and sharding (partitioning). Finally, the differences about relational (SQL) and non-relational (NoSQL) datastores which naturally leads into two popular state-of-the-art NoSQL databases Cassandra and Redis and two major relational DBMS will be investigated and compared.

2.1 Motivation for Database systems

A database is defined as organized collections of data. Some of people use the term database to refer to datastore systems, while others refer it as collection of datasets of the information that is stored into the datastores system [6]. These datastores systems which take care about the whole datasets and their organization, retrieval, storage and update are called Database Management Systems (DBMS). These systems interact with different users' applications, computer systems and other DBMSs to facilitate huge data storage and management. DBMSs are now classified into two different categories: Relational DBMS which uses Structured Query Languages (SQL) and it is known as SQL Database, and Non-Relational DBMS which is known as NoSQL and stands for Not Only Structured Query Languages [7]. Regardless of the differences between the two approaches, global applications agree on their importance. A benchmark of performance among these two states-of-the-arts DBMS was explored in this project.

2.2 Application scenario

As noted earlier in this report, the current research was an extension of a research conducted within UDBL research group in Uppsala University [1] which investigated and compared only one state-of-the-art NoSQL database, MongoDB, with two relational databases. The present project was extended to cover two more state-of-the-art NoSQL datastores, new commercial relational database and the latest version of open source relational DBMS used before. Therefore, in this project, same application and datasets that were used in the previous research were reused.

The application consisted of real world persisting logs within industrial fields in which human manpower was replaced by machines that are continuously controlled and monitored by computers. Therefore, each machine has many sensors which read values of different factors such as pressures, temperatures and other important values. All these values have to be stored within fast and efficient datastores to ensure proper performance of huge values loading in both retrieving and writing logs. Database schema of this application has one collection, i.e. table, called Measures. Measures consist of machines factor ‘m’, sensor ’s’, beginning timestamp of reading the value ‘bt’, ending timestamp of reading the value ‘et’, and the measured value ’mv’. On this table (collection) there is a composite key consists of three main columns (machine, sensor, and beginning timestamp of reading the value). With this format, now the measures (m, s, bt, et, mv) are ready to receive persisting data logs from large scale data measurements from machines' sensors.

This big data logs will be bulk-loaded into two different state-of-the-art databases, Cassandra [4] and Redis [5], as well as two popular relational DBMS. After completing the bulk loading process, there will be different execution of fundamental queries for accessing and analyzing persisted logs. The properties of these queries are basic selection, range search and aggregation.

2.3 Transactions

Transaction is a primary key of a database to be a datastore. A transaction is made of multiple related tasks to perform concurrent database tasks, either performing all of them, or none of them. The tasks of single transaction have to be executed in order and after all are completed, the change is reflected and committed to the database permanently. Rolling back a transaction if required, has to undo all tasks related to same transaction. This feature is called ‘Atomicity’ and it is a property of ACID. This feature guarantees that no state in the database transaction is left partially uncompleted or in an inconsistant state. By guarantee of transaction atomicity, both developer and client can make all changes in one transaction without caring about consistency maintenance. Consistency is the third property of the DBMS transactions, this feature ensures that any data written in the database, must be valid and passed all the rules of commit and keeps the database always consistent while the clients are reading.

Multiple clients can write\read simultaneously in parallel from same database, each running different transactions, however, each transaction has to be as if it is the only one in the system and no transaction has to affect any other transactions. For instance, if two transactions are running in parallel reading\writing same value, change the value from first transaction before the

second transaction committed change in the same value, the behavior of one or both of these transactions can be affected. This feature which works as locking is called the 'Isolation' and it is the third property of ACID. Although, some applications allow only one transaction at a time, this delivers very poor performance. The final property of the transaction is that the database has to ensure durability of the transactions i.e. holds all latest updates even if the system fails or restarts after the commit and before reflected to the disk. However, this property could be dispensed with crash-recoverability on the applications [8].

2.4 Indexing

Indexing is data structure that can be used to accelerate the process of retrieving data within datastore table. All relational and some of non-relational DBMSs have indexing strategies and supporting multiple indexing (Secondary Indexing) for different columns within the same table. Therefore, the intelligent system of the DBMSs such as Query Optimizer can select which index to perform query with, to get fast retrieval, or do full scan for the table if the linear search by index is inefficient for large databases [1]. Indexing strategy prevents the overall search of huge Big Data in most of the time. Indexing in DBMS is similar to what we can see in all books as table of contents and it can be defined in different attributes based on where the performance is needed. Therefore, indexing can be classified as:

Primary Index: It is defined in the key, where the key field is the primary key of the table generally. It is unique and none duplicated.

Secondary Index: It can be defined on both candidate and non-candidate keys of the table or the relation. Its value can be duplicated.

Clustering Index: The table records are physically ordered based on the clustering key, therefore, there is only one clustering index has to be defined within a table.

2.5 Sharding

Sharding or partitioning, is simply physically breaking the large databases to multiple smaller partitions, and distribute the data among these parts, whilst putting them back together on the querying or analyzing purposes. Sharding is used for scaling the systems up to provide performance advantages in both bulk loading and analyzing by reducing datasets in single database. Replication of the data coming up with partitioning idea, where copy of each part of database is maintained in other part or node, therefore, it does not only provide a performance, it also provides availability, reliability and ensures fault tolerance.

2.6 Relational and Non-Relational Databases

Relational databases have been used for more than four decades now [9] for various types of real applications. On the other hand, the non-relational datastores which are known as NoSQL (Not Only SQL) have started just recently. In 2009, the term NoSQL began to call for non-relational, distributed, horizontally scalable design and open-source databases. Nowadays, this term covers many databases models such as key-value, wide-column, graph and documents datastores. The non-relational datastores were brought to the surface to solve the problems claimed by some users and researchers about state-of-the-art relational datastores performance among Big Data and new era of web applications. The main difference between relational and non-relational databases can be summarized in three points:

2.6.1 Scaling

The relational DBMS are designed to scale up vertically for more datasets size, i.e. it requires bigger capacity of hardware to be more powerful as the size of data increases, while NoSQL datastores are designed to scale out horizontally, i.e. add new partitions automatically to scale as data size increases. This is one of the features of NoSQL datastores that is often used to compare them to RDBMS.

2.6.2 Type of collections

The relational DBMS deals with structured tables to store the data, these tables have normal relations to each other to be in normalized state, while non-relational datastores deal with semi or unstructured data collections that are not related to each other instead each collection standsalone. Some claim that this design in NoSQL datastores for their data collections may affect the performance positively [10] and that is because keeping normalized data may have negative effects on the performance in structure model comparing to de-normalized model of NoSQL datastores

2.6.3 Consistency

Consistency is a state of both ACID and CAP theorems [11], in the context of the databases, data cannot be written to the disk unless it passes the rules of valid data and that change in the data has to be seen from all nodes simultaneously. In case of any transaction or change creates inconsistent data, the whole transaction has to be rolled back. Consistency is important in most

cases especially in financial transactions therefore, it is usually easier to define consistency within database level instead of application level.

RDBMSs have natural atomic transactions which means that if there are many actions such as insert, delete and update have to be done within one transaction, atomic consistency must ensure that either all of these actions are processed or none of them. In other words, RDBMS is based on using ACID (Atomicity, Consistency, Isolation, Durability) properties during the operations of the management [12] and these properties give the reliability for the transactions. In contrast, NoSQL datastores systems that lack these properties are lighter, perform better and provide eventual consistency from BASE (Basically Available, Soft State, Eventual Consistency) properties [13], in which the time between actions of one transaction could vary in the execution time, but all writes or changes will be reflected as the time goes on. However, RDBMS has an optional lower consistency level similar to eventual consistency in non-relational databases called weak or relaxed consistency [1]. The variety of consistency levels between these state-of-the-arts might be important for some applications and not necessary for others. These variations were investigated in the current work.

2.7 Competing DBMS

In this project, four datastores were studied. They are as following:

2.7.1 RDBMS

Two relational DBMSs were involved in this performance evaluation; DB-C which is a major commercial relational vendor and the popular open source relational database, known as DB-O. The state-of-the art DB-C varies from the one that was used in previous benchmark project [1] and a newer version of DB-O was included in this evaluation. The results of both DB-O and DB-C were compared with NoSQL datastores experimented in this project. In addition, since the previous project [1] investigated both consistency levels (relaxed and strong) in same application scenario and concluded that there is no difference or impact on the performance level between both consistencies for both DB-C and DB-O and to keep the results of this investigation comparable with the previous experimental studies, only the relaxed consistency was used in this investigation and comparison.

2.7.2 Cassandra

"Apache Cassandra is defined as a distributed storage system for managing huge volume of structured data spread out across many commodity servers, while ensuring highly available service with no single point of failure."[14] "Cassandra's data model is a partitioned row store with tunable consistency. Rows are organized into tables and the first component of a table's primary key is the partition key. Within a partition, rows are clustered by the remaining columns of the key. Other columns may be indexed separately from the primary key."[15] Cassandra has SQL-like interface called CQL (Cassandra Query Language) which provides most keywords of SQL but it does not support joins or sub-queries. This state-of-art database supports build-in secondary indexing in any column of column-family key-space.

2.7.2.1 Indexing strategies and Keys in Cassandra

Indexing in Cassandra requires to be matched with queries that are planned to be run in future, that means planning for queries must be done prior to the indexing structure. An example of indexing is the application scenario of this project explained in the box below;

```
CREATE TABLE measuresA
    (
        m  int,
        s  int,
        bt  timestamp,
        et  timestamp,
        mv  Double,
        PRIMARY KEY (m,s,bt)
    );
```

Since we have composite key in machine identifier 'm', sensor identifier 's' and begin time 'bt'; in Cassandra the first part of "Primary key" is the "Partition key" [15][14] in this case the partition key is the machine identifier 'm'. Sensor identifier 's' and begin time 'bt' are called the clustering columns within machine identifier 'm'.

```
cqlsh:bench> select * from measuresa;

 m    | s | bt   | et   | mv
------+---+------+------+----
  50  | 1 |  810 |  812 |  0
  50  | 2 |  810 |  812 |  0
  50  | 3 |  810 |  812 |  0
  50  | 4 |  810 |  812 |  0
  50  | 5 |  810 |  812 |  0
  10  | 1 |  810 |  812 |  0
  10  | 2 |  810 |  812 |  0
  10  | 3 |  810 |  812 |  0
  10  | 4 |  810 |  812 |  0
  10  | 5 |  810 |  812 |  0
```

Fig1. Snapshot of selection queries results shows machine identifiers 10 and 50

As an example of the above table, measuresA, I assumed that there are 10 different machine identifiers ‘m’ (10, 20, 30, 40, 50, 60, 70, 80, 90,100) each identifier has five different sensors (1, 2, 3, 4, 5) and each sensor has begin time ‘bt’, end time ‘et’ and measured value ‘mv’.

```
-------------------
RowKey: 50   Partion key vlaue
=> (name=1:810.0:, value=, timestamp=1430135237217999)
=> (name=1:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=1:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
=> (name=2:810.0:, value=, timestamp=1430135237217999)
=> (name=2:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=2:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
=> (name=3:810.0:, value=, timestamp=1430135237217999)
=> (name=3:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=3:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
=> (name=4:810.0:, value=, timestamp=1430135237217999)
=> (name=4:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=4:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
=> (name=5:810.0:, value=, timestamp=1430135237217999)
=> (name=5:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=5:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
-------------------
RowKey: 10   Partion key vlaue
=> (name=1:810.0:, value=, timestamp=1430135237217999)
=> (name=1:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=1:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
=> (name=2:810.0:, value=, timestamp=1430135237217999)
=> (name=2:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=2:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
=> (name=3:810.0:, value=, timestamp=1430135237217999)
=> (name=3:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=3:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
=> (name=4:810.0:, value=, timestamp=1430135237217999)
=> (name=4:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=4:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
=> (name=5:810.0:, value=, timestamp=1430135237217999)
=> (name=5:810.0:et, value=4089600000000000, timestamp=1430135237217999)
=> (name=5:810.0:mv, value=0000000000000000, timestamp=1430135237217999)
-------------------
```


Fig 2.Snapshot of how the data is structured within the table

Listing the results of selected all data query based on how data are ordered internally is shown in figure 1, while figure 2 shows how the data is structured within the table in a database is dependent on the partition key, where the partition key in this example is the machine identifier. It is clear that under each partition key, all attributes belong to that key are listed. In figure 2, the red box represents one row of data and the first line in the red box includes the clustered values which are in our example the sensor 's' and begin time 'bt' followed by rows that indicate values of end time 'et' and 'mv' thatare not part of clustering columns. Note that both of the 'et' and 'mv' values are identified by their clustered columns.

As mentioned above, planning the structure of the table keys depends on the queries that are willing to be performed because if we want to retrieve data from the above structure with a query such as:
[select * from measuresA where mv = 0;],
this will not be possible in Cassandra, since 'mv' is not part of primary key and it is not indexed, so a secondary index on measured value 'mv' must be applied to perform the above query. Moreover, Cassandra does not allow the query with any part of primary key if it is not following the sequence of primary key [16] because it may affect the performance of searching/reading data from the database.

```
cqlsh:bench> select * from measuresa where s = 1;
InvalidRequest: code=2200 [Invalid query] message="Cannot execute this query as
it might involve data filtering and thus may have unpredictable performance. If
you want to execute this query despite the performance unpredictability, use ALL
OW FILTERING"
```

Fig 3. Error when skipping the partition key and retrieve by any clustering columns within primary key.

To solve the problem of the query in figure 3, Cassandra provides "ALLOW FILTERING" option which enables (some) queries that need filtering with skipping the partitioning key as well as sequence of primary key elements [16]. However, when executing a query with beginning time 'bt' only in where clause as in figure 4, Cassandra will reject it even with "ALLOW FILTERING" option.

```
cqlsh:bench> select * from measuresa where  bt = 810  allow FILTERING;
InvalidRequest: code=2200 [Invalid query] message="PRIMARY KEY column "bt" cannot be
 restricted (preceding column "s" is either not restricted or by a non-EQ relation)"
```

Fig 4. Error result with part of primary key and "ALLOW FILTERING"option

The option that was available to execute the above query was to make a secondary index on beginning time 'bt' column within primary key without using "ALLOW FILTERING" option as indicated in figure 5.

```
cqlsh:bench> create index bt_index on measuresa(bt);
cqlsh:bench> select * from measuresa where  bt = 810;

 m    | s | bt   | et   | mv
------+---+------+------+----
  50  | 1 | 810  | 812  |  0
  50  | 2 | 810  | 812  |  0
  50  | 3 | 810  | 812  |  0
  50  | 4 | 810  | 812  |  0
  50  | 5 | 810  | 812  |  0
```

Fig 5. indexing part of primary key

Unfortunately, Cassandra does not support running the above mentioned query with greater than or less than operators. As described in [17], "Cassandra supports greater-than and less-than comparisons but for a given partition key, the conditions on the clustering column are restricted to the filters that allow Cassandra to select a contiguous ordering of rows".

```
cqlsh:bench> select * from measuresa where  bt > 810 allow FILTERING;
InvalidRequest: code=2200 [Invalid query] message="PRIMARY KEY column "bt" cannot be
restricted (preceding column "s" is either not restricted or by a non-EQ relation)"
```

Fig 6. Error to perform Basic Analytic Query

To allow Cassandra to select data that have greater-than or less-than at specific beginning time 'bt', the sensor component of the primary key must be included in the filter using an equality condition and the 'ALLOW FILTERING' option, i.e. it must be specified from the beginning for which sensor exactly the data is retrieved. So the query should be as below:

[select * from measuresa where s = 1 and bt > 810 ALLOW FILTERING;]

As mentioned previously one of our plans was executing different queries, basic selection, range search and aggregation with differentindexing strategies. However, most of these queries depend on measured values ‘mv’ and the only way to perform all our queries was to redefine the primary key of our table measuresA as following:

[PRIMARY KEY ((m,s,bt), mv);]

The partition key of this primary key consists of; machine identifier ‘m’, sensor identifier ’s’ and begin time ‘bt’ all together. The measured value ‘mv’ here represents the clustering column and has no direct impact on the composite key of the table since it was included for querying purposes only.

Figure 7 clearly shows that all data was partitioned based on the composite key (m,s,bt) and clustered within each partition by the measured value ‘mv’. After redefining the primary key it became possible to execute all queries we planned to include in our mini benchmark. Hence, the primary key in Cassandra for our model consisted of composite partition key which was used internally to separate the rows of Cassandra data structure. It also consisted of clustering key which was used to organize non primary keys (columns) within each partitioned row. Therefore, the concept of the Cassandra primary key for our application is different from SQL primary key concept but still we have super composite partition key since ‘bt’ value is a timestamp value. It is important here to note that the partition key and clustering columns might need to be altered in order to run different queries.

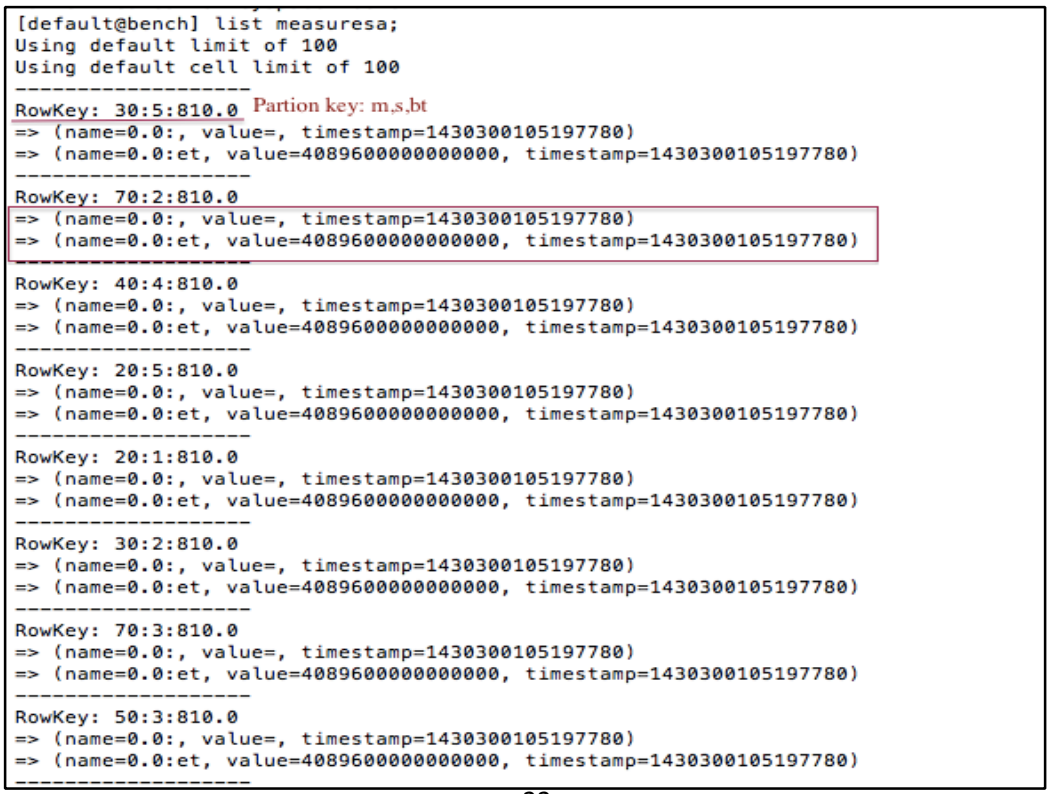

Fig 7. New data structure depends on new Primary Key

2.7.2.2 Consistency levels in Cassandra

Consistency in Cassandra has multiple levels and each level concerns with different strategies and replica numbers. Changing these levels of consistency affects read and write performance which is written in Cassandra user guide as a warning. Replica number determines how many copies of same data exist in different nodes for availability purpose and it ensures fault tolerance and avoids failure. Replica could have two or more copies of every row in a column family across participating nodes. In case of more than one replica, when a client connects to any node of the database cluster (peer connection) that node in this case is called a coordinator. If the write request for example is written in

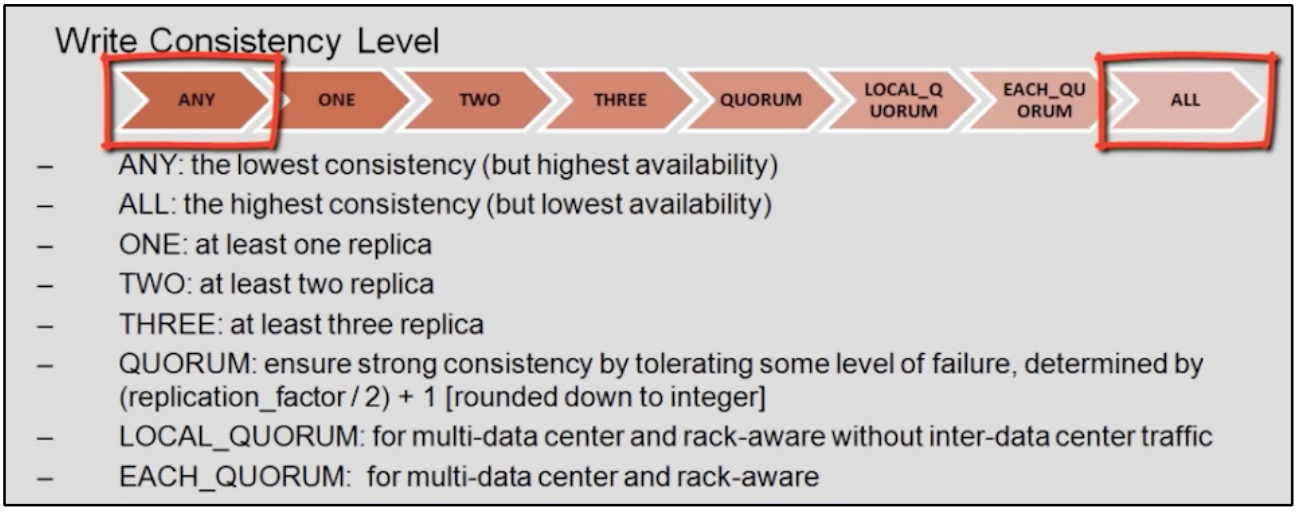

Fig 8. Write consistency levels in Cassandra [18]

different node than the coordinator, then the latter sends sub-request to the node intended.
Although only three levels of consistency are mentioned in Cassandra user guide (One, Quorum, ALL), figure 8 shows different writing consistencies of Cassandra, this snapshot was takenfrom a tutorial in YouTube for Cassandra Administration video series by Packt Publishing [18]. The weakest one is level ‘ANY’ whereas ‘ALL’ is the strongest consistency. These levels are:

- ANY: the weakest, write or update must succeed in any available node\replica.
- ALL: the strongest consistency, write or update must succeed in all nodes\replica.
- ONE: write\update must succeed in at least one node responsible for that row (primary or replica).
- TWO: write \update effects at least two replica
- THREE: write\update effect at least three replica
- QUROM: write\update effects majority of the replica, i.e. must succeed in a minimum number of replica nodes determined by ((replication_factor /2)+ 1).

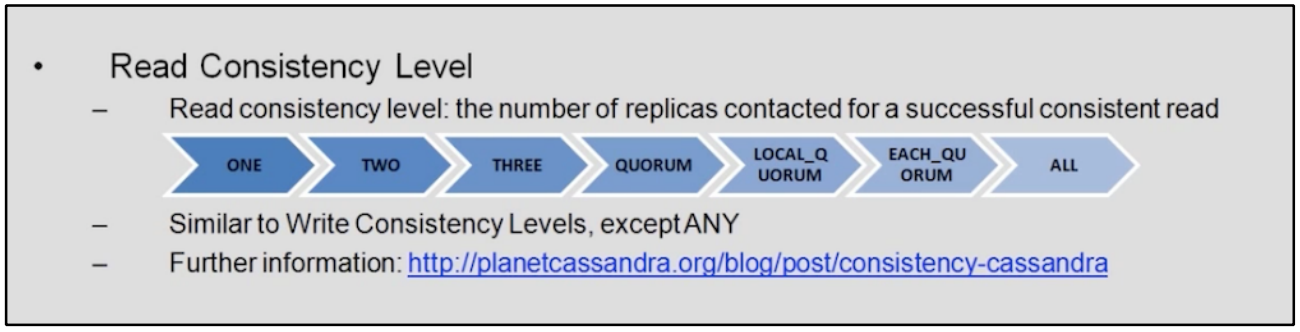

Fig 9. Read consistency levels in Cassandra [18]

The read consistency levels in Cassandra are displayed in figure 9. Read consistency levels are similar to writing consistency levels except the 'ANY' level which is not included but it is equal to ONE. An example in the Cassandra user guide shows that as the consistency level is changed from ONE to QUORUM to ALL, the performance of reading deteriorates from 2585 to 2998 to 5219 microseconds, respectively [17]. The read consistency levels can be explained as below [20]:

- ONE: the weakest, reads from the nearest node holding the data.
- TWO: reads from at least two closest nodes holding the data.
- THREE: reads from at least three nearest replication nodes holding the data.
- QUORUM: reads the result from a quorum of nodes with the most recent data timestamp.
- Local_Quorum: reads the result from a quorum of nodes with the most recent data timestamp in the same data-center as the coordinator node.
- Each_Quorum: reads the result from a quorum of nodes with the most recent data timestamp in all data-centers.
- ALL: this is the strongest and it reads the results from all replicas.

All these consistencies have been tested while doing the experiments in single node of Cassandra but unfortunately these consistency levels have no effects on single node\replica. Note that, the default consistency was set at level ONE for both writing and reading.

2.7.2.3 Reading and Writing paths in Cassandra

The process of data insertion in Cassandra is explained in figure 10 which was taken from Cassandra

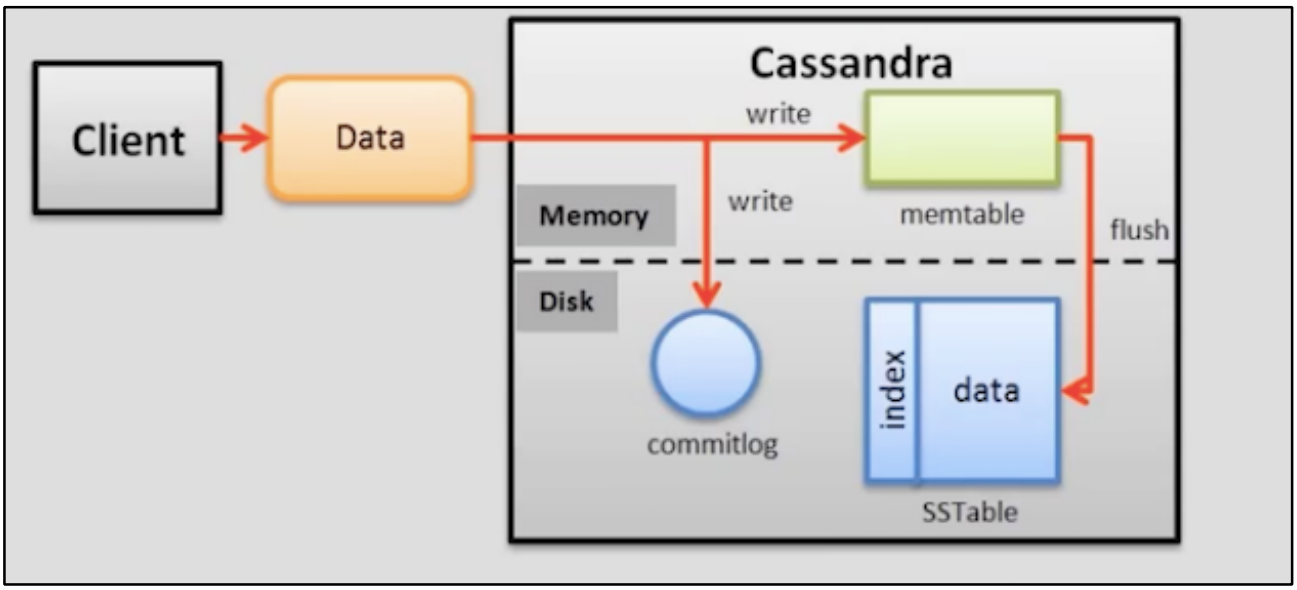

Fig 10. Write path in Cassandra [18]

Administration video series by Packt Publishing on YouTube [18]. Initially, when data is inserted to Cassandra, it is first written to a commitlog file which ensures durability and safety of the data and at the same time data is written to in-memory table known by Memtable. The latter then buffers the writes and eventually when it is full or reaches certain size, it flushes the data to a disk structure table called SSTable (Sorted String table). In more details, when Memtable exceeds the size, it will be replaced by new Memtable and the old Memtable will be marked as pending for flush which later it will be flushed by another thread [18][19]. SSTable is data file containing row data fragments and only allows appending data. In each flush operation, data is written on a disk as new SSTable in background. After transferring the data from Memtable to new SSTable, the SSTable will eventually be compressed. As a result of the above processes, the column for requested row could be fragmented over several SSTable and unflushed Memtable [18][19]. To reduce fragmentation and save the space of the disk, SSTable files are merged into a new SSTable occasionally.

The SSTables and their associated files for our experiments are presented in figure 11. The name of the SSTable started with name of Key-space (database name) followed by table name then the created order number. Note that the file which contains the data is the file which ends by Data.db.

The process of normal writing path in Cassandra is more expensive in term of time if the user wants to make bulk loading for Big Data. Bulk loading is useful for performance test, migrating historical data and when changing the topology of the clusters. Cassandra provides a tool called SSTable loader for bulk-loading process [21].

Fig 11. SSTables

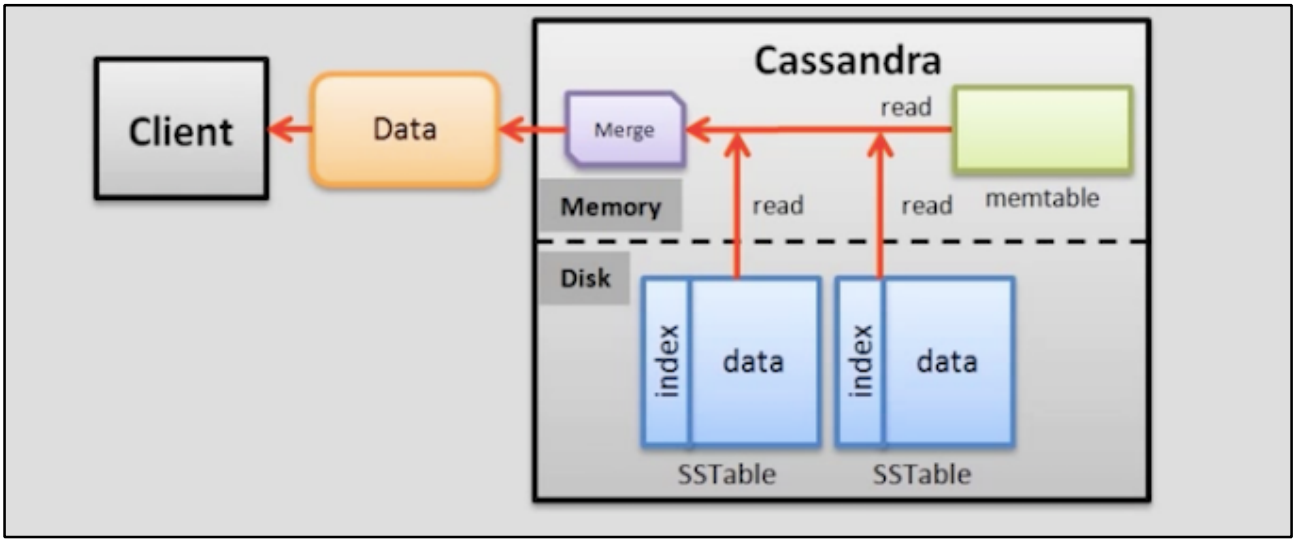

Fig 12. Read path in Cassandra [18]

When there is any read request from a client, the node which is connecting and serving the client called a coordinator. Any node can be a coordinator as well as a replica and this is known as peer connection strategy. If the coordinator is not holding the needed data, it will contact and forward to number of nodes as specified by the consistency level. The fastest replica received by the coordinator will be checked by in-memory comparison. Simultaneously, the coordinator checks all the remaining replicas in the background and updates the replicas which have inconsistent data, these operations are called Read and Repair [18]. As shown in figure 12, in each node, when a read request is received, rows from all related SSTable and unflushed Memtables are merged. [18] [19].

2.7.2.4 Migrating Data to Cassandra

Migrating data from external file or other databases to Cassandra has many options and developer can choose depending on the existing data and how fast the data must be migrated. The options below are available for migrating data to Cassandra [22]:

- COPY command – Cassandra Query Language Shell CQLSH provides a copy command to bulk load raw data from external data file such as CSV file but this is not recommended for massive files since it follows the normal path of writes.

- SSTable loader – this tool is provided by Cassandra for massive data and fast insertion process It basically transfers an existing SSTables which contain the data directly to SSTables of the Cassandra tables.

• Sqoop – this utility is used in Hadoop to migrate data from RDBMS into a Hadoop cluster. Cassandra Enterprise provider DataStax provides pipelining data from RDBMS table directly into a Cassandra column-family, so this is basically used for migrating from existing database to Cassandra database.

• ETL tools – there are many ETL tools that support Cassandra both as a source and target datastore i.e. migrating from Cassandra database into another Cassandra database. Some of these tools are free to use (e.g. Talend, Pentaho, Jaspersoft).

In our benchmark, Copy command and SSTable loader wereexperimented. A selective part from comparison of both options on bulk loading the data and analyzing the logs indicated that SSTable loader is the fastest option that can be used for benchmarking purposes (Appendix F). In addition, analyzing the data which was bulk loaded by SSTable loader is faster since all needed data is located in SSTables of the Cassandra column-family. In contrast, the data which was loaded by copy option is dispersed across SSTables and Memtables.

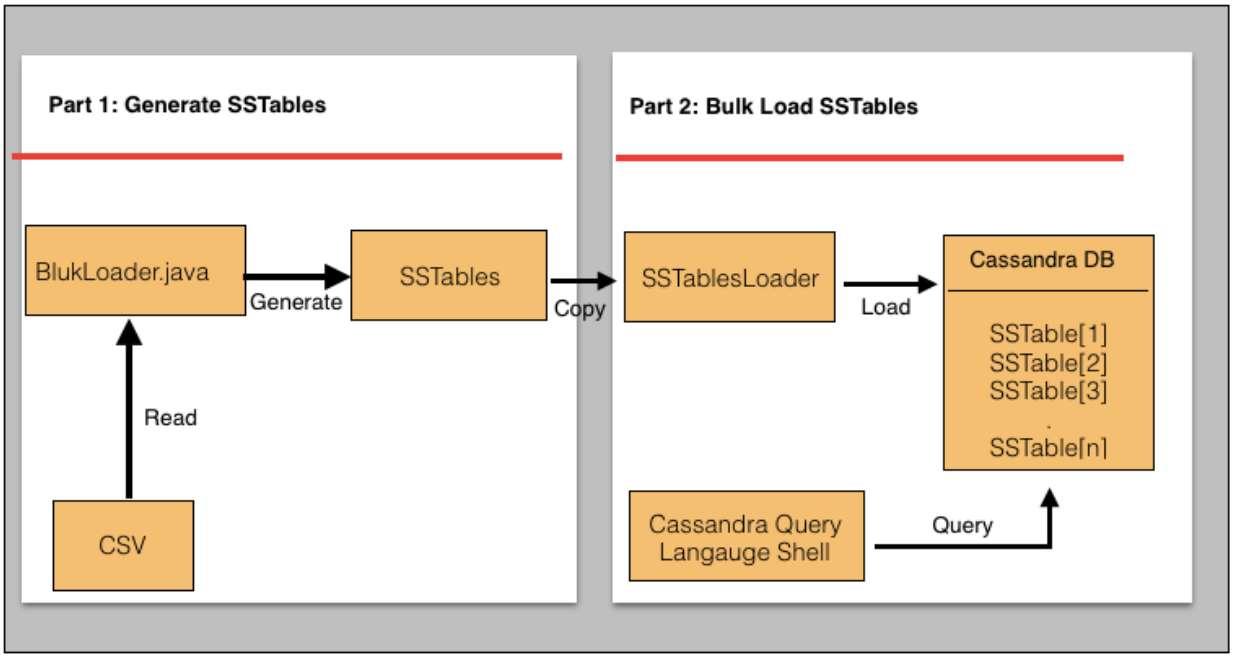

Fig 13. Bulk loading Process [18]

When using SSTable loader for bulk loading, there are two main steps must be performed ,figure 13. In the first part, the programmer has to implement java class which reads the data to be uploaded from CSV file, at the same time this class generates SSTables similar to the tables that are generated

with normal process of writing data to Cassandra clusters. This allows omitting and skipping insertion to Comitlog files and Memtables. In the second part, the SSTable loader tool that is provided by Cassandra will be used to copy the generated SSTables of part one. The copying process also includes loading these SSTables to Cassandra Key-space (schema). Bulk loading process using SSTable loader tool is very fast comparing with normal data insertion via Cassandra clients.
The API convertor code of part1 of figure 13 is available in Appendix A, it includes a java class that was re-coded to fit our application scenario data and tables. The example can be found in [23] which was also explained in Cassandra provider Datastax in [21]. Understanding SSTables, SSTable loader utility and re-coding bulk loader java class require a lot of time due to leakage in the guide recourses and dependencies.

2.7.2.5 Partitioning in Cassandra (sharding multiple nodes)

Cassandra carries out partitions across all nodes specified within a cluster database. All the data are divided into ‘’The Ring’’ and each node in the ring is responsible for one or more key ranges overall database. The user has the control over the partitions in term of order and distribution i.e. random or ordered partitions and over how many nodes on the ring should be replicated. The two basic data partitions in Cassandra are:

Random partitioning: This is the default and recommended, since it divides the data equally across all nodes as possible using MD5 hash function.

Ordered partitioning: this is not recommended, since it stores and partitions the data in sorted order specified by user across the nodes therefore, some sorted partitions may contain more data and other less.

To ensure fault tolerance and no failures for any node, there is an option to replicate each partition data across participated nodes or even data centers. The number of replicas can be specified as needed and in case of many replicas when a client connects to any node of the cluster that node in this case is called the coordinator. If write\read request has to be in different node or the coordinator is not responsible about that part, then the coordinator will forward the request to the number of nodes hold that part as specified by the consistency level. The fastest replica returns the data will be checked by in-memory comparison and at the same time the coordinator checks all the remaining replicas and updates the replica which has inconsistent data, this operation is called read and repair.

2.7.3 Redis

Redis is an open source, non-relational, in-memory data structure store, used as message broker database and cache [5] [24]. It also provides build-in replication and automatic sharding -partitioning- in Redis cluster. Data in Redis can be represented in five different data structure based on key-value concept. Also, retrieving the data from database is based on using different commands for each data type that has been used for data representation. Since Redis is used in caching, it supports sophisticated Least Recently Used (LRU) eviction of keys algorithm depending on user's needs.

The data in Redis can be structured based on the following key-value structure [25]:

- **Strings:** strings is the simplest data structure of Redis, it saves the value associated with the key, as a string.

- **Hashes:** Hashes represent the data as multiple fields-values for a particular Redis key. Both fields and values are saved as strings.

- **Lists:** List of elements for a particular key, these elements are sorted based on insertion order and saved as strings. They are similar to Arrays of strings data.

- **Sets:** sets of unordered strings elements for particular key. These elements can be used for intersections, union, different sets operation with other sets.

- **Sorted Sets:** sets of ordered elements for a particular key, each element associated with floating value is called the score which is used for ordering the elements accordingly.

- **Bitmaps:** strings values are handled as an array of bits for a particular key.

- **Hyperloglogs:** this data structure is used for probabilistic counting of sets. Elements of this data structure are encoded as strings.

Based on the above data structures and after long investigations, it was agreed to use two different data structures in parallel (hashed and sorted set) for this benchmarking scenario in order to accommodate the datasets and to execute the queries of the experiments as real world application. Hashes data structure was used for key lookup purposes whilst sorted set is the only data structure in Redis which provides the range search queries.

2.7.3.1 Indexing in Redis

Redis is key-value datastore in which data is indexed and addressed based on the key name through which any value could be obtained. Therefore, the concept of this is similar to primary key indexing in SQL datastores. For this reason, we can fairly say that Redis has indexing feature by nature. However, since it has multiple data structures for representing the data, the developer can choose among these data structures based on the data values required for secondary indexing purposes [26]. Redis users can use Sorted Sets, Sets and Lists data structures for indexing and they can run these as a standalone accommodation for their data. Alternatively, the above data structures can be associated in a complicated way with other data structures such as Hashes and this ends up to same as using two different data structures. The easiest and simplest Redis data structure to be used as secondary indexing or i.e. for a key that needs to be sorted for better performance on the retrieval, is the Sorted Sets. Sorted Sets can represent the data of normal RDBMS column as key-score-elements where the score is a numerical floating number and the elements are set of associated value belong to that score. Sorted Set data type saves elements based on the score values in an ascending order. This data structure is only one which the user can search for information based on giving range of score values (minimum, maximum) [26]. For instance, in our scenario application, measured value (MV) is represented as following:

[ZADD mv 2.5 1:8:16.3:17.4], where;

- ZADD: is the insertion command for Sorted Set.
- mv: the key or the Sorted Set name which is equivalent to columns' name of 'mv' in RDBMS.
- 2.5: is the score value for particular elements associated to the score.
- 1:8:16.3:17.4 : are the elements for that score, which are in our case machine number '1', sensors number '8', beginning time '16.3, and ending time '17.4'

Note that unlimited score-elements values can be added for these datasets under 'mv' key.

2.7.3.2 Consistency in Redis

Redis as a single node is always consistent and considered as the strongest consistency level and it acknowledges the writes to the clients. However, Redis cluster which runs multiple nodes and replicas (master-slaves) lacks strong consistency behaviors [27]. This means, the slaves may lose the acknowledged writes by the master due to the asynchronous replication feature of Redis cluster for any reason. However, this slave node can take over the master node and converts into the master in case of main master gets crashed or cannot reply because of network problem. Redis developer can force the database to flush to the disk before it replies to the client programmatically but this could impact the performance which is the main concern of Redis. Redis provides WAIT [28] command which is used for waiting a period of time for writing acknowledgment from slaves. Unfortunately even with this feature there is no guarantee for strong consistencies because even if the slaves do not reply during that period Redis will continue handling the requests from clients and reflecting changes into the master. Therefore, Redis cluster (partitions) is considered always the weakest in the consistency while a single node of Redis is always consistent [28].

2.7.3.3 Redis Persistence

Redis is very simpler in-memory datastore and its reading and writing paths are not complicated. Redis keeps all the data in-memory (RAM) while it is running. In addition, Redis saves the data into two different disk files, one is called Redis Database File (RDB) and the other is known as Append Only File (AOF). RDB file saves the representation of datasets and it is used by Redis for loading the data after the restart of the memory and it is used also for backups. In contrasts, AOF file saves the log of writes operations command in the same format used by clients, therefore it is used mostly for durability. However, these persistent options can be disabled if the purpose of Redis is not for storing but only needs the data in-memory while the server is running such as in caching systems. The data can be saved in RDB based on the system configuration file. Since the user can control the duration of saving operation that is performed to the disks files, it can be tuned to be every number of writes (e.g. 50 writes) or every number of seconds (e.g. 20 seconds) [29]. In addition, the user can manually run saving command. During each saving operation, a new RDB file is created to copy-on-write data from existing RDB file to new RDB file with the latest unsaved data and then the old file will be replaced. In contrast, AOF file appending and synchronization are done after each SET or insertion commands [30]. Clients always receive the requested data from the stored data in Redis in-memory files which guarantees fast retrieval operations for such system.

2.7.3.4 Partitioning in Redis

Partitioning (sharding) in Redis is the processes of dividing the keys and associated values among symmetrical running Redis instances, therefore, each instance will accommodate subsets of total keys-values. In addition, each node or instance has a file which contains all the cluster nodes information and the keys each node have with respect to each node has a special ID, IP and port number in the cluster. All nodes have auto discovering features for corresponding nodes within same cluster for unreachable nodes detection as well as they can perform a slave node to be as a master by election if required. Replicating the keys among different nodes is supported in Redis clustering for availability. Master-slaves are used in term of data replication between the main node and its replica in asynchronous consistency. In case of any master becomes unreachable for some reason, an automated operation will be performed by cluster nodes to replace that master node by its replica then the replica (the slave) takes over to be the master while the old master becomes the slave after resuming the work. Moreover, if there is a master without any slave, it will get automated slave from a master which has more than one slave. During the clustering operation of creating the number of nodes with master and slaves, there is an internal operation performed to divide number of known slots among nodes to accommodate the corresponding keys. This operation is called Key Distribution Mode and it distributes the key space of the database to 16384 slots. Each node of the cluster is responsible about an equal range of slots which are in turn equally distributed among all nodes, note that each slot can allocate multiple keys. Redis clusters nodes use the information of slots distribution to redirect the clients to the right nodes for writing\reading key-value. Redis cluster topology is a full mesh where each node can communicate to any other nodes using TCP connections. Therefore, clients can connect to any node in the cluster even the slaves. The connected node will analyze the requested query and if the slot of the query is available within this node, it will reply by the data needed otherwise it will redirect the client to the node that accommodates the key slot. There is no forwarding methodology of the request from node to another and waiting for the reply, it just completes redirecting from connected node to the right node. Although this automated operation is supported in Redis Command shell (Redis-cli), in case of external API clients, this operation has to be programmatically handled by the developer by using HASH_SLOT = CRC16 (key) mod 16384 function [31]. From the previous explanation, it is obvious that each Redis node works as a standalone server and it cares about the data that has been saved or to be saved in its slots only. Theoretically, this makes no difference in the performance if the machine or server hardware resources such as the RAM are bigger in size which can accommodate all data.

2.7.3.5 Redis Mass insertion

The normal insertion to Redis using Redis-cli shell commands is not that efficient for huge amount of data because normal insertion is time consuming when it comes to waiting for the reply of each insertion commands. Therefore, as most database systems provide utilities for bulk loading massive data, Redis system provides the ability for the users to upload millions of key-values in short time as fast as possible without waiting replies of each command. The bulk loading in Redis called 'Mass Insertion' [32] in which the clients have to create a txt file containing Redis insertion commands such as SET & ZADD with the associated data to be inserted in row format. Below is an example of part of such file:

```
HMSET measuresa:10:1:0810  m 10 s 1 bt 0810 et 0812 mv 0
ZADD  mv   0  10:1:0810:0812

HMSET measuresa:10:2:0810  m 10 s 2 bt 0810 et 0812 mv 1
ZADD  mv  1  10:2:0810:0812
```

To insert million records of raw data, the client has to generate a file contains one million rows of Redis commands with their data. Once the file is ready, the remaining thing is to feed the database by the data by magic commands of Redis Mass Insertion called pipe mode shown below:

```
cat data.txt | redis-cli --pipe
```

This command allows the client to upload huge data faster than normal insertion since it feeds that server with data without waiting for the reply of each command. At the end of transferring all data, a message of information appears for the client with the number of records that have been transferred and an error messages if any.

This Redis technique for Mass Insertion still has some drawbacks:

- It does not provide the time consumed by the server to upload the data, therefore, execution time was manually added in redis-cli for this project for testing this feature.
- In case the data covers the maximum of utilized memory of the machine (RAM), it deadlocks and stops without giving any notice or message about the problem, while the client is waiting assuming that data is still transferring.

- The main problem of this feature is that it supports the bulk loading only for the connected node but not for multiple Redis instances or cluster nodes. Therefore, an Application Programming Interface (API) was developed (Appendix A) for both single mode server and partitioning mode while running multiple Redis nodes in parallel with respect to HASH_SLOT function which is responsible for distributing the data among nodes slots as explained in section (2.7.3.4).

3- Related Work

In [1], both types of state-of-the-art database were benchmarked for persistent data from real industrial world application and the investigation was conducted for various configurations of indexing strategies. The results of this benchmark showed that SQL databases have more advantages of performance compared with NoSQL databases by having the query optimizer. However, this benchmark investigated only one NoSQL database. In the current project, the investigation was extended to cover more state-of-the-art NoSQL datastores. Additionally, a different state-of-art relational database from commercial vendor that has been used in the previous research was applied in this project. Moreover, new release of the relational database from open source vendor was used here.

It was mentioned in [33] that indexing and multiple indexes per table in SQL data-stores can affect the performance of different types of querying and give good decision for query optimizer to choose proper indexing among multiple indexes. However, experiments with different indexing strategies and experiments within bulk loading condition were not performed before. Analyzing Big Data logs from real application for such research needs to be tested within persistent data logs environments which is the main focus of this project.

Indexing strategies have been experimented in different researches and showed that they affect querying performance of both database types although in [34] this reality was acknowledged, the research experiments did not use any indexing strategies. The reason is that the NoSQL database used for the comparison does not support automatic generation of query plans. Our research conducted all comparison experiments with one or multiple indexing and compared them with no indexing for the same experiments. Moreover, the YCSB [35] benchmark and TCP-H DSS benchmark [36] were run over SQL server [37] without examining them for relaxed and strong (ACID) consistencies. In the present research we considered different consistency levels among all experiments. In addition, data parallelization over multiple nodes was investigated.

4- Methodology

4.1 Data Set

Our evaluation data set was generated in real-world industrial application where many machines monitored by administration for the purpose of productive quality. Therefore, each machine has multiple sensors reading their values from different perspectives and these values are loaded and stored in datastore. After the loading, multiple queries execute to verify and analyze the persistent logs. The size of the data set was divided in different sizes linearly: one, two, four and six gigabytes. Each data-set size includes 19,530,000 , 37,800,000 , 74,550,00 and 111,180,000 records respectively as shown in Figure 4.1. These records were injected into datastores which were then evaluated by this benchmark.

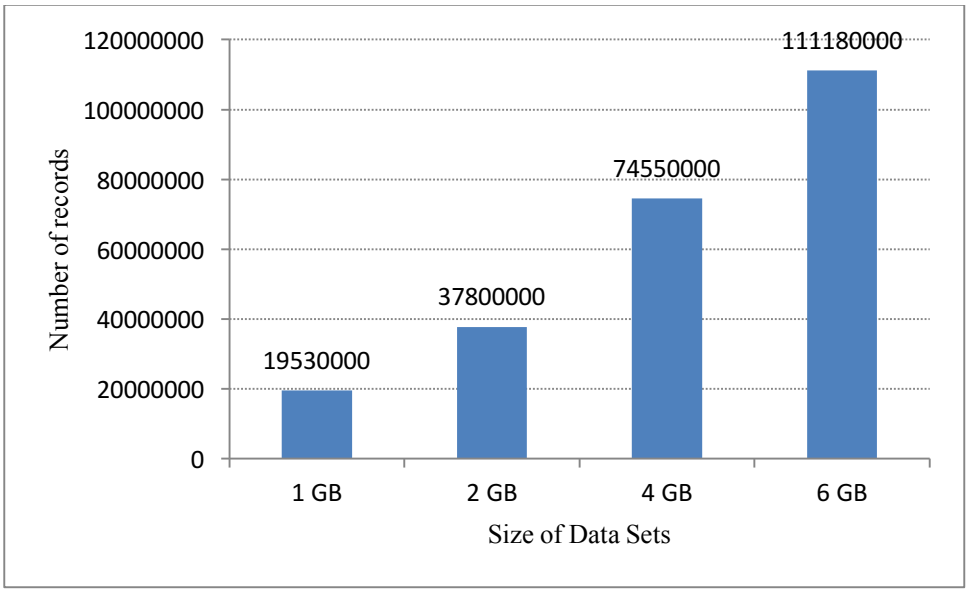

Fig 4.1. The Volume of data for experiments

4.2 Queries

The project was expected to define a benchmark that includes fundamental queries for accessing and analyzing persisted streams. Before executing the queries, datasets have to be bulk loaded into the datastores linearly for different indexing strategies. The properties of the queries were basic selection, range search, aggregation and other advanced queries to discover the efficiency of query processing and index utilization of the DBMSs. This benchmark was limited to these queries since it is known that non-relational NoSQL datastores do not support complicated numerical operators,

joins and multiple sup-queries. Most of these queries were already conducted in the previous project held in UDBL research group [1].

4.2.1 Basic selection, Q1

First query is key lookup query, precisely, finding a record for a given machine 'm', sensor 's' and begin time 'bt'. The query for all data-stores is specified as following:

SQL (RDBMs)	CQL (Cassandra)	REDIS-CLI (Redis)
SELECT * FROM measures WHERE m = ? AND s = ? AND bt= ?;	SELECT * FROM measures WHERE m = ? AND s = ? AND bt= ?;	HGETALL measures:m(?):s(?):bt(?)

Table 4.2.1 Lookup Query Q1

4.2.2 Range search, Q2

A query from a client (external client) was called and executed and then the client was asked for dataset within specific range search of measured values. The computational method of the query's result will be at the client side. Multiple queries (seven) were performed for such testing with different range search and selectivity results. Such query is presented as following;

SQL (RDBMs)	CQL (Cassandra)	REDIS-CLI (Redis)
clientCount(SELECT * FROM measures WHERE mv > ? AND mv< ?)	clientCount(SELECT * FROM measures WHERE mv> ? AND mv< ? ALLOW FILTERING;)	clientCount(ZRANGEBYSCORE mv min_value max_value)

Table 4.2.2 Range Search Query Q2

4.2.3 Aggregation, Q3

This query has same range values of Q2, aggregation with range search, but the computational method of query's results is within the server side. Also multiple queries (seven) were performed for such testing with different range search and selectivity results. Such query is displayed as below:

SQL (RDBMs)	CQL (Cassandra)	REDIS-CLI (Redis)
SELECT count(*) FROM measures WHERE mv > ? AND mv< ?	SELECT count(*) FROM measures WHERE mv> ? AND mv< ? ALLOW FILTERING;	ZCOUNT mv min_value max_value

Table 4.2.3 Aggregation Query Q3

4.3 Benchmarks Environment Experiments setup

The experiments of benchmarks were performed in a machine running Intel® Core™ i5-4670S CPU @ 3.10GHz x 4, with Ubuntu 14.04 LTS 64-bit operating system. The machine has 16GB of physical memory (RAM) and 500GB of disk space.

Acronym	Name , Consistency Level, Distribution	Properties
DB-C	DB-C & Weak consistency, Non-Distributed System	No logging, Read Committed
DB-O	DB-O & Weak consistency, Non-Distributed System	No logging, Read Committed
CA	Cassandra with single node, Non-Distributed System	Read & write consistency one, Durable
CA-SH	Cassandra with cluster of 4 nodes, Distributed System	Read & write consistency one, Durable
CA-SH-R3-W	Cassandra with cluster of 4 nodes, 3 replica, Weak consistency, Distributed System	Read & write consistency one, Non Durable
CA-SH-R3-S	Cassandra with cluster of 4 nodes, 3 replica, Strong consistency, Distributed System	Read & write consistency ALL, Durable
Redis	Redis with single node, Strong Consistency, Non-Distributed System	Read & write Strong consistency
Redis-SH	Redis with 6 nodes shards, no replica, Strong Consistency, Distributed System	Read & write Strong consistency

Table 4.3. Consistency configurations for the experiments

4.3.1 Relational DBMS Configuration

For both state-of-art RDBMs, the query result cache was turned off and the transaction isolation level was set to Read Committed which is second weakest level at isolation ladder. In addition, the query logging was disabled. In the open source database system, the new default storage engine was used which allows transactional and isolation features. The buffer size was utilized carefully depending on the resources of the machine environment.

4.3.2 Cassandra Configurations

For all Cassandra systems, both heap_newsize and max_heap_size variables were carefully set to 800MB and 8GB respectively. In addition, all variables related to the reading timeout such as ‘read_request_timeout_in_ms’, ‘range_request_timeout_in_ms’ and ‘request_timeout_in_ms’ were increased to fit the long period of reading from the clients and most of them had maximum default as

10 seconds only. Since Cassandra Query Language (CQL) terminal was used which is not printing the execution time after querying, the terminal was edited to add this feature. Moreover, the client_timeout variable which also had 10 seconds maximum time to read was also increased. The default and recommended random partition strategies were used. The querying cache was stopped by setting 'row_cache_size_in_mb' to zero in Cassandra configuration file. As recommended in Apache Cassandra provider page, swapping was disabled for performance matter which I found that it is better than with swapping. In this investigation, the latest release 2.1.9 of Cassandra was used for both Shards and non-shards.

For Sharding systems of Cassandra (CA-SH, CA-SH-R3-W, CA-SH-R3-S), Cassandra Cluster Manager (CCM) tool was used and it was provided by Apache Cassandra provider (DataStax). CCM creates multi-node Cassandra clusters on the local machine, therefore 4 nodes were created to run Sharding experiments. As mentioned above, all configurations such as heap_newsize and max_heap_size were configured carefully and equally between nodes to fit the size of datasets results. The configured values per node for both heap_newsize and max_heap_size were 300MB and 3GB respectively.

4.3.3 Redis Configurations

Redis systems' configuration file was updated to fit the resources of the environments and the clients output buffers limitation was set to zero to force disconnecting the clients if there is no reading to the data or idle. Since turning on append-only-file variable which was used only for durability guarantee and had no effects on the performance as investigated in this study (data not shown), it was turned off to save more memory and CPU usage. As Redis in this project was used as datastore not as cached systems, the eviction policy was set to no eviction. In Redis, the max-memory variable was not set for certain size therefore, it was automated to use how much RAM is available. In contrast, in sharding experiments, the memory was equally distributed among all nodes and the system. The default reply-timeout was increased since we expected long time than defaults. Automated saving variable which is responsible to write the data to the disk was switched off and manual saving command was used after each loading. The latest release 3.0.5 of Redis for both shards and non-shards was used in this investigation.

For Redis sharding or as called in this thesis a Redis-SH, official Redis clustering tutorial which is available in the system provider website was used to create a cluster of 6 shards. The benchmark shards database for 1GB, 2GB, and 4GB of dataset requires approximately 9.11 GB, 15.45 GB and 34 GB respectively of memory size (RAM). Only clustering without any replications was

investigated due to the limitation of memory in the benchmark environments. Moreover, for memory limitation reason, for the 4GB of dataset, the data structure which is related to Q2 & Q3 only was loaded as will be discussed later (Appendix C).

4.3.4 Bulk-loading Experiments

For each consistency configurations DBMSs shown in Table 4.4, the bulk-loading time was measured for 1, 2, 4 and 6 GB of raw data with different indexing levels whenever it was compatible with the systems. The data which had to be loaded, it was saved in CSV file format where each row of raw data represented machine identifier 'm', sensor identifier 's' , begin time 'bt', end time 'et', and measured value 'mv'. To be fair in this evaluation, the experiments were started with empty databases each time.

4.3.4.1: Experiments with No-Indexing

First experiment of bulk-loading was conducted without any indexing strategies in RDBMS using fast batch loader utility of each system which accepts high-volume data loads from CSV file into a single table. In Cassandra as explained in section (2.3.2), there is no strategy to keep the table which holds the data without indexing. Therefore, these experiments were not performed for CA, CA-SH, CA-SH-R3-W and CA-SH-R3-S. In Redis there is only one setting for all bulk loading experiments since it is indexed congenitally by key attributes as mentioned in section (2.3.3). Moreover, two different scenarios of bulk loading in Redis were investigated here; the Mass Insertion utility [32], provided by system provider, and the API that was developed by me (Appendix A). The API reads the data directly from CSV file and bulk loads it to the datastore while the Mass Insertion utility uses the Redis command protocol format and reads only from text file format. The latter option led to developing a special data convertor API which reads from CSV file and appends Redis command format then it exports the data into txt file format (Appendix A). The expectation for both of RDBMS was to have faster performance since the distribution of the data among the table has no constraints. For in-memory systems, Redis and Redis-SH, it was expected to be fast at the beginning until the loaded data volume reaches certain size at which it goes slower due to the fact that the data is ordered based on the key order and the regarded size of the memory which is already loaded by data. Note that the auto-saving feature was disabled for all systems during the loading.

4.3.4.2: Experiments with Primary indexing (Sensor Key Index)

At this stage, the primary index was built in composite key (m,s,bt) before loading the data into the systems. All Cassandra systems (CA, CA-SH, CA-SH-R3-W, CA-SH-R3-S) were involved in this investigation since primary key creation is essential to create column-family within these systems. However, since bulk loading is required for range and aggregation queries investigation, the measured value was included in this primary key and considered as clustering key of our case scenario as explained in section (2.8.2). Therefore, the primary key of Cassandra column-family was set to be ((m,s,bt),mv). Because the Redis and Redis-SH systems are key-values systems, the structure of the keys was formatted to include our composite key, therefore, same Redis results from the last bulk loading (with no index) were used for comparison. It was expected to see a reduction in the performance of both state-of-art relational databases due to data and indexing distribution constrains among the table. Bulk loading for both non-sharding systems (CA,Redis) was expected to be much faster than their Sharded forms (Redis-SH, CA-SH,CA-SH-R3-W, CA-SH-R3-S). In addition, CA-SH-R3-S was expected to be the slowest among all systems because of its strongest consistency level.

4.3.4.3: Experiment with Primary and Secondary indexing

This experiment included an additional secondary indexing on measured value column with primary key index that was used in the last experiment. Building both indexing had to be done before data loading to emulate the reality of our case scenario. Each system of our experiment configuration supports both primary and secondary indexes except for Redis and Redis-SH because they lack the existence of indexing in traditional manner. However, Redis is capable of sorting the values according to the key name format of hashing data structure which we already used in our investigation. In order to create secondary indexes, another data structure of Redis was used known as Sorted Set on measured value (mv). Both Redis data structure formats were bulk loaded simultaneously to also accommodate our experiments queries later. It was predicted from this experiment that both relational database and state-of-art Cassandra systems performance to be adversely affected. While Redis and Redis-SH results were thought to be same as the last experiments since no change was added.

4.3.5 Basic Selection Query Experiments

The scalability of key lookup basic selection query Q1 was measured for each consistency configuration DBMSs, shown in Table 4.4. The investigation covered all data sizes 1, 2, 4 and 6 GB

of raw data with different indexing levels if they are compatible with the systems. In this part of the study, the command line utilities of each system was used and for each query run the average time of three executions was measured.

4.3.5.1: Experiment Q1 with No-Indexing

This part of our investigation evaluated the systems with Q1 in which no indexing was built. Unfortunately, all Cassandra configured systems could not compete in this experiment since the primary indexing is the main requirement to build the table in these systems. Both state-of-art DB-C and DB-O were predicted to be slower than in-memory Redis and Redis-SH due to the full table scanning for the key in both RDBMSs.

4.3.5.2: Experiment Q1 with Primary indexing (Sensor Key Index)

Building the primary indexing on the composite key was the main feature of this experiment in which all investigated systems and their tunable levels were involved including Cassandra. For CA, CA-SH, CA-SH-R3-W and CA-SH-R3-S, the primary key also included clustering key of measured value in order to generalize our column family to suite both Q2 and Q3. It must be noted here that whether to include or omit the measured value had no impact on the results of bulk loading and Q1. Compared with full table scanning in last experiment, the performance of both DB-C and DB-O was expected to be more efficient after building the composite key index due to the fast retrieval of the key which is based on the indexing. We also expected Cassandra to be fast due to the quick reach to the partition including the key which is based on the composite primary key. Redis and Redis-SH were expected to be super fast because of preparedness of the key-value in the memory compared with other systems which have to hit the disk to fetch the data.

4.3.5.3: Experiment Q1 with Primary and Secondary indexing

In this experiment, an extra secondary key on the measured value was built with the existing primary composite key. Also, this experiment investigated how the systems perform when there is an extra indexing in same table which actually requires speeding up other reading queries such as our range search and aggregation queries. The expectation of this experiment included a non-observable effect on the performance due to that our key lookup query was based on the primary key index here.

4.3.6 Range Search Query Experiments

In order to know how the system performs and scales when there is a third party client reading from the database, the analytic range search query Q2 was executed for this investigation. For each system, simple client was developed to read seven different selective range queries using most popular and recommended drivers for each system. Note that, all selective data row results were sent to the client for counting purpose. All the developed application program interfaces (API) are available in appendix (E). This experiment was tested for each consistency configurations DBMSs shown in Table 4.4 and it covered all data sizes 1, 2, 4 and 6 GB of raw data. As in key lookup query Q1, the average time of three executions was measured.

4.3.6.1: Experiment Q2 with No-Indexing

This section deals with testing the performance of the systems in which no indexing was built. As in Q1 with no-indexing investigations, Cassandra systems were not part of this experiment. It was assumed that there will be a delay in retrieving the data due to the network level traffic between client and database systems. Moreover, for both RDBMSs, the performance was expected to be slower than in-memory systems due to full table scanning for measured values within the selective range compared with preparedness of the same data in sorted sets of Redis in-memory.

4.3.6.2: Experiment Q2 with Primary indexing (Sensor Key Index)

Building a primary index based on the composite key may not have a positive effect on the performance for analytic queries but it may affect the performance of some systems which save both data and indexes in the same table-space as in DB-O. In addition, it also allows the column-family systems to start participation in Q2. No performance advantage was expected for both RDBMSs since the Q2 is based on measured value indexing instead of the sensor key index. For Cassandra, it was proposed to observe slowness compared to the others since there is no indication to which partitions range to search for the selectivity of measured values, therefore it has to pass through all partitions ring. In contrast to Cassandra, Redis and Redis-SH were speculated to perform faster than others due to the readiness of the results sorted in-memory.

4.3.6.3: Experiment Q2 with Primary and Secondary indexing

In order to be fair in our investigation for performance of Q2, a secondary index was built on the measured value column with the existence of the primary key to meet the reality. The systems

performance in RDBMSs was expected to improve and this improvement might be observed in Cassandra systems too. Redis and Redis-SH results from last experiment were used here since they do not utilize any secondary key and their data structure 'Sorted Set' was used as secondary indexing.

4.3.7 Aggregation Query Experiments

In these experiments, the systems were requested to count the total number of sensors that have measured values between two selective values and return single number of the total. The experiments in this part were based on using the command line utilities of each system with supported build in functions for aggregation to remove the overhead of transferring the data to a third party client. Also, these experiments used same seven different selective range queries, holding same values as in Q2. The measurements of these experiments covered all consistency configurations DBMSs shown in Table 4.4 with all data sizes from 1 to 6 GB of raw data. As in both Q1 and Q2, the average time of three executions was measured.

4.3.7.1: Experiment Q3 with No-Indexing

The first Q3 experiment was carried out without any indexing utilities as in section (4.4.3.1) and only DB-C, DB-O, Redis and Redis-SH were investigated here excluding Cassandra and its derivatives. Both DB-C and DB-O were assumed to be slow for full table scanning to find the results whereas Redis and Redis-SH were expected to be unmatched super fast.

4.3.7.2: Experiment Q3 with Primary indexing (Sensor Key Index)

In theory, indexing columns which is not of primary concern for the query to be run has no effect on the performance of the query. Hence, this experiment either approves or disapproves the theory with our investigated systems. As in the experiments setup of section (4.4.3.2), Cassandra started to compete for a position in this comparison experiment of Q3. Over all, the expectation from this experiment was to be similar to that of Q2 with primary indexing with a difference in the execution time since transferring the massive datasets results to the clients was eliminated.

4.3.7.3: Experiment Q3 with Primary and Secondary indexing

Similar to Q2 experiment (section 4.4.3.3), Q3 was experimented with an extra secondary index on measured value column with the primary indexing to speed up the fetching of the selective sensors

data from the database systems. All systems were expected to significantly improve in comparison to the results of the last experiments, however, Redis and Redis-SH were thought to possibly be similar since no change was done but they should be still strong competitors.

5- Evaluation and Benchmark

5.1 Bulk-loading Experiment Results

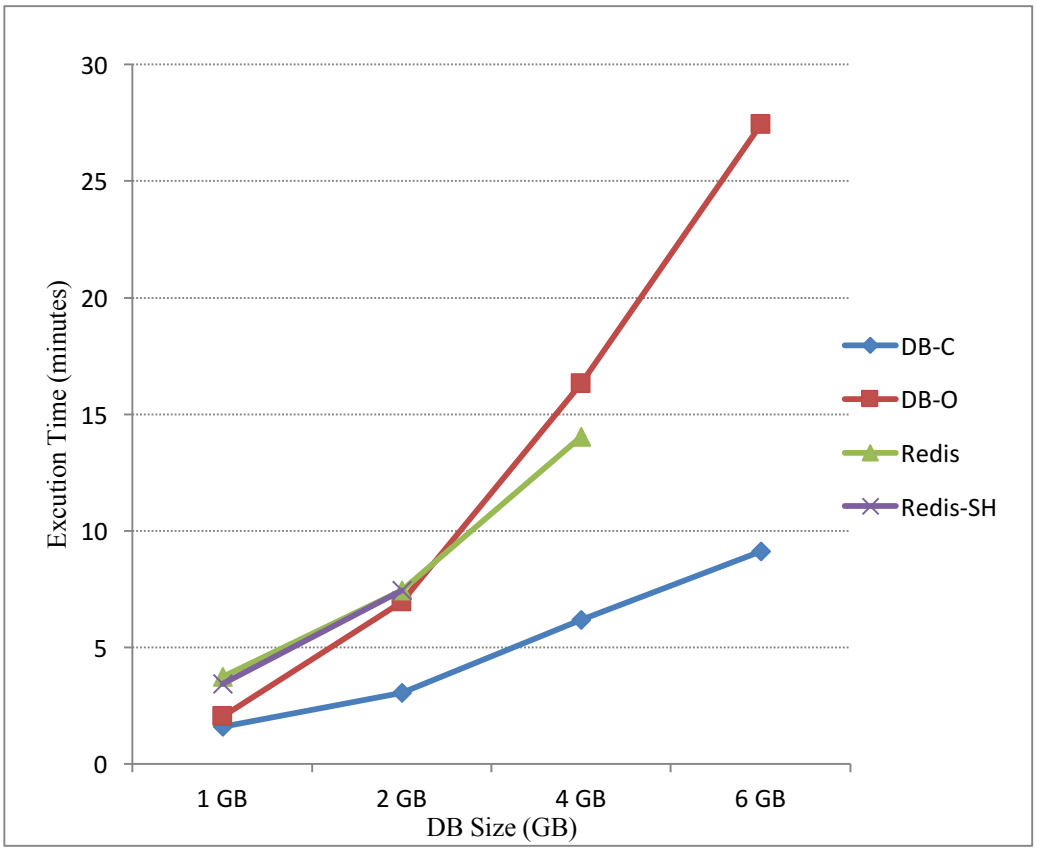

Fig.5.1. Performance of bulk loading without indexing

Fig.5.1. shows the results of bulk loading without any indexing for DB-C, DB-O, Redis and Redis-SH among four datasets sizes. Over all, DB-C performed better than others in this experiment, while DB-O was observed to be faster than both Redis and Redis-SH between 1GB and 2GB before it started to scale slow in 4GB and 6GB. For both types of Redis, the performance was identical, that is because Redis-SH does not distribute the data for shards in parallel.

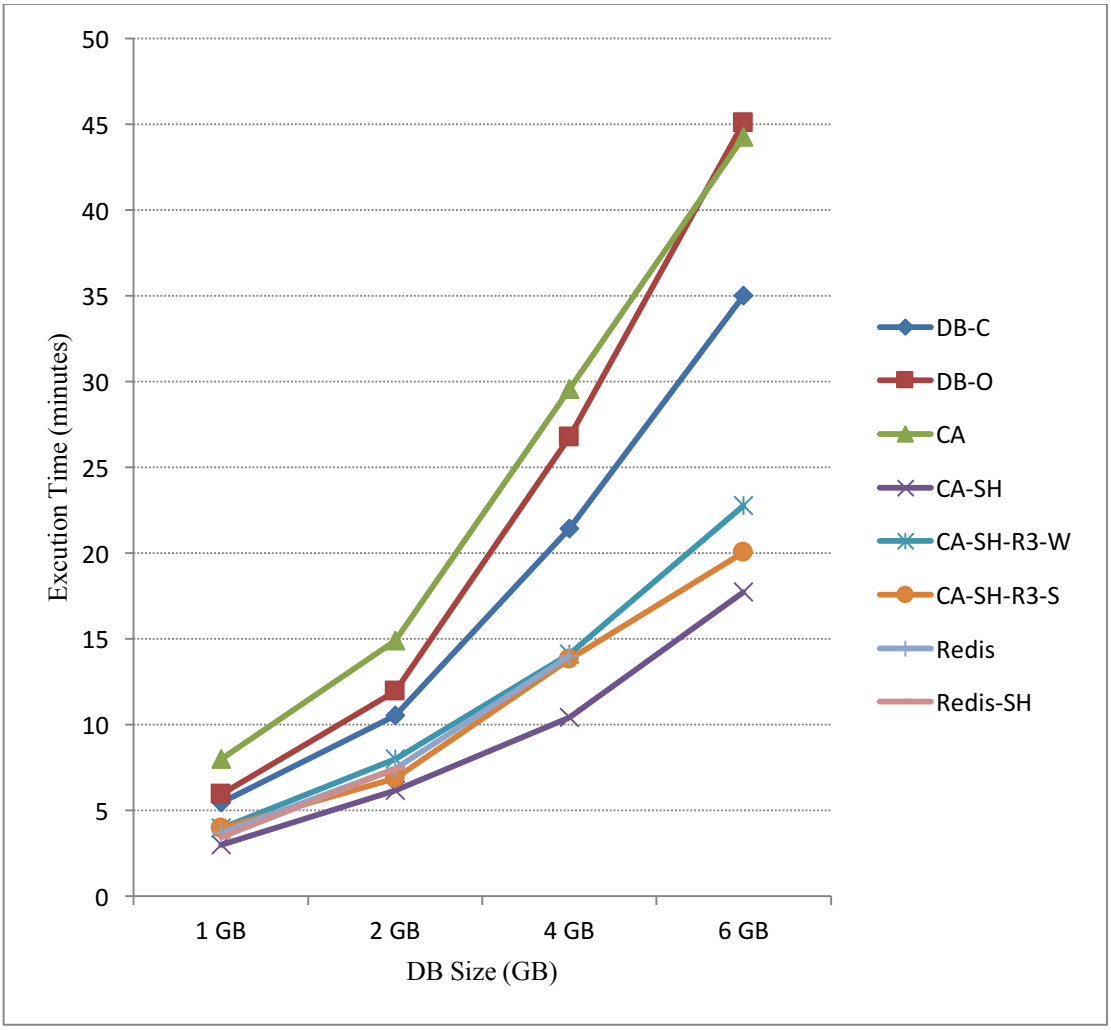

Fig.5.2. Performance of bulk loading with sensor key index

Fig.5.2. illustrates the performance of bulk loading with sensor key index with all of the DBs participated in this experiment. All datastores belong to Cassandra sharding (CA-SH, CA-SH-R3-W and CA-SH-R3-S) were observed to be much faster than others due to the concurrent data distribution among the shards or nodes while CA lacks this feature which causing it to be the slowest in this experiment. Redis & Redis-SH were identical same as in previous experiment, also they scaled faster than both RDBMS and CA. It is clear that both DB-C and DB-O performed worse when there was no indexing which could be attributed to the dependency of data distribution on the indexing strategies.

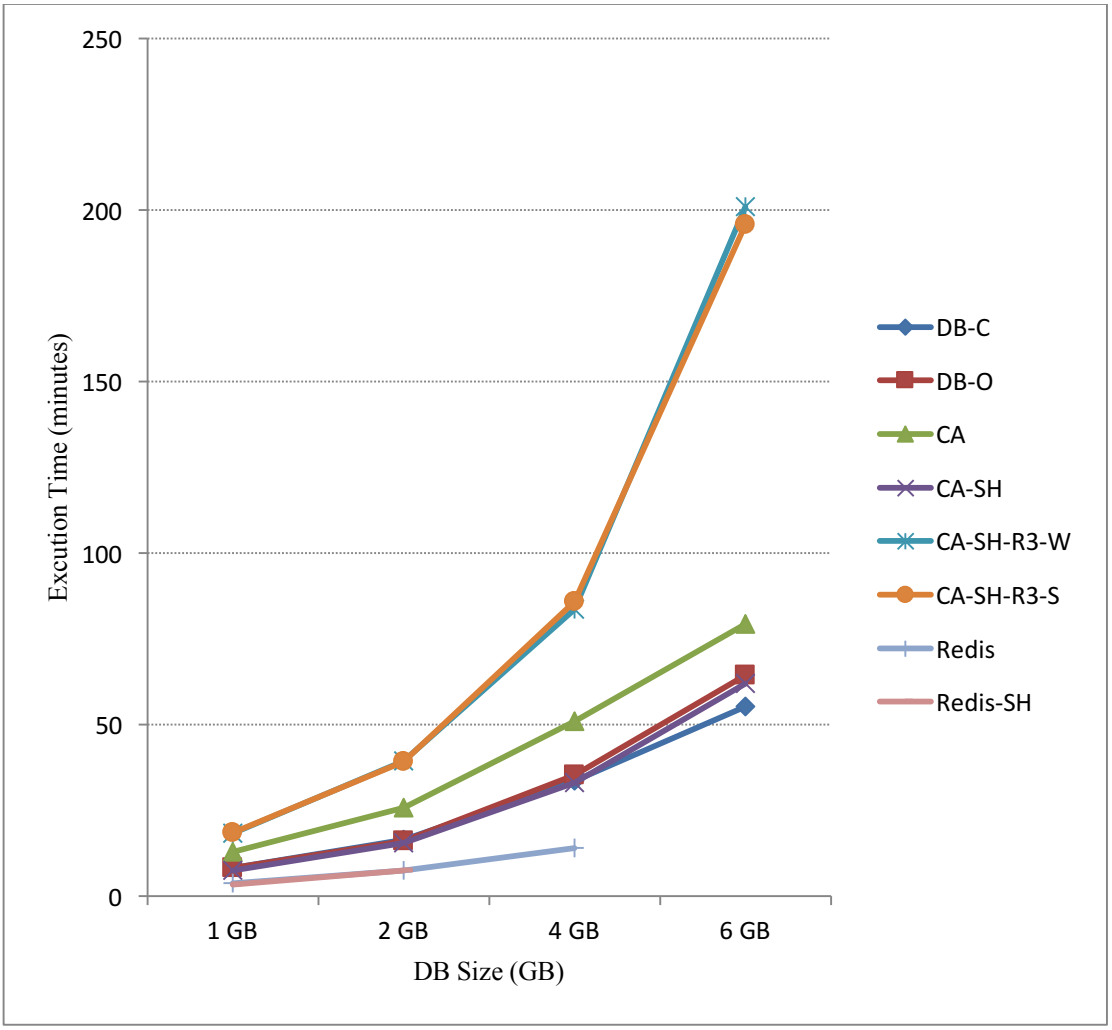

Fig.5.3. Performance of bulk loading with sensor key and measured value indexes

Fig.5.3. clarifies that all DBs offer scalable loading performance. Redis and Redis-SH were in the lead up to 4GB and 2GB respectively. DB-O proceeded as fast as DB-C up to 74,550,000 (4GB) million of raw rows however it performed slower at 111,180,000 of data rows than DB-O. Both CA-SH-R3-W and CA-SH-R3-S scaled significantly worse than others because both of them after loading the data to all nodes and replicas, they rebuilt the indexing among the data which took up to 3 hours for loading 6GB of data. Note that both CA-SH-R3-W and CA-SH-R3-S had identical performance because they use SSTable loader feature which directly loads SSTables to the correct replica nodes. This feature does not go through the normal coordinated write process, so consistency levels have no effects on performance.

A summary of all bulk loading experiments is presented in table 5.1. in which the performance of each system was evaluated with respect to the performance of other systems within this experiment. The evaluation is presented column wise.

System\Indexing Strategy	No Index (Fig.5.1.)	Sensor key Index (Fig.5.2.)	Sensor Key & MV indexes (Fig.5.3.)
DB-C	Very Good	Good	Good
DB-O	Good	Bad	Good
CA	Not apply	Bad	Bad
CA-SH	Not apply	Very Good	Good
CA-SH-R3-W	Not apply	Very Good	Very Bad
CA-SH-R3-S	Not apply	Very Good	Very Bad
Redis	Good	Very Good	Very Good
Redis-SH	Good	Very Good	Very Good

Table 5.1: Summary of the bulk loading experiments**

** The evaluation in this table is based on the overall numerical results of the experiments (see Appendices F, G, H, I).

5.2 Basic Selection Experiment Results

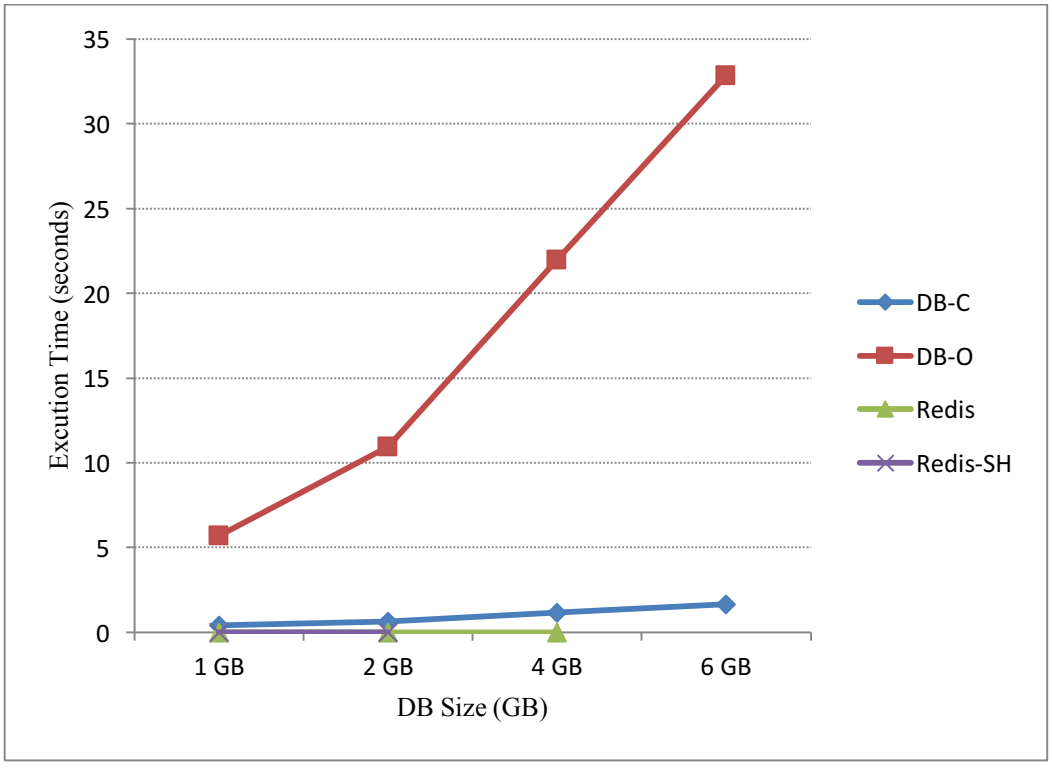

Fig.5.4. Performance of Q1 without indexing

The first experiment of basic selection or key lookup was conducted for four systems DB-C, DB-O, Redis and Redis-SH, while it was not applicable to perform this experiment for other databases. Both in-memory systems (Redis & Redis-SH) performed super fast since their data are ready in the RAM and ordered by key-value. Although DB-C has no indexing, its results were opposite to the expectation since full scanning among massive data rows were fast compared with DB-O which reached to more than 32 times of the DB-C at 6GB and 21 times of the Redis at 4GB.

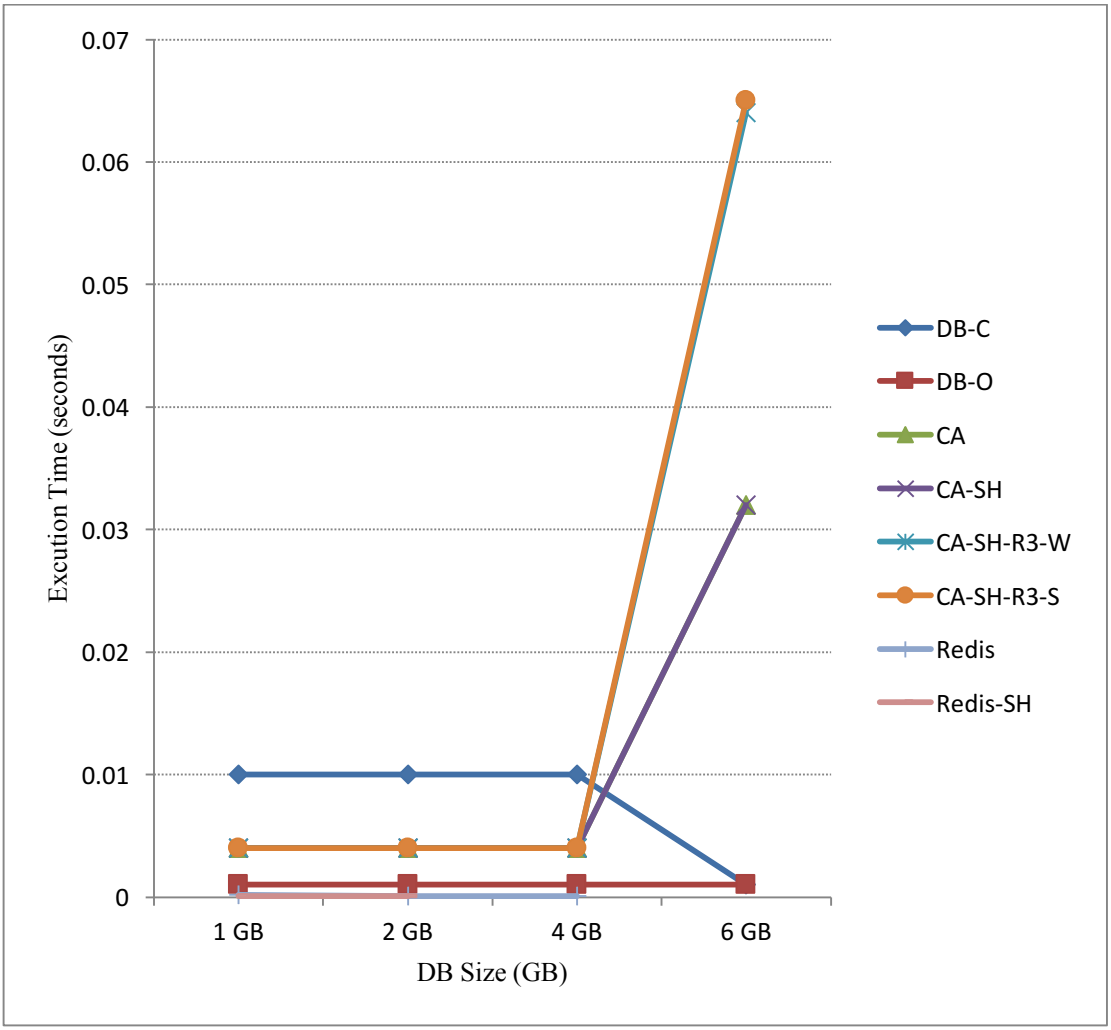

Fig.5.5. Performance of Q1 with sensor key index

In the second experiment of key lookup, indexing was built on the composite primary key. As expected, both Redis and Redis-SH were the fastest and they had same results since Redis sharding redirects the clients to the node which has the key-value store. However in Cassandra, the coordinator forwards the request to the node which has the data and waits for its reply then it replys

to the clients. RDBMS had better results in most datasets sizes than previous experiment, this was because the fast fetching used the index on the composite key instead of the table full scanning. In addition, it is clear that DB-O performed better than DB-C for 1GB, 2GB and 4GB that could be due to the optimization table feature which was used after bulk loading immediately. This feature reorganizes the data on the table with associated index data, reduces table space on disk and improves I/O efficiency when accessing the table. Also, the above chart indicates that most systems that belong to Cassandra (CA, CA-SH, CA-SH-R3-W, CA-SH-R3-S) had identical results for all data sizes except at 6GB where both CA-SH-R3-W and CA-SH-R3-S jumped to higher execution time by only 0.060 of seconds (figure 5.5).

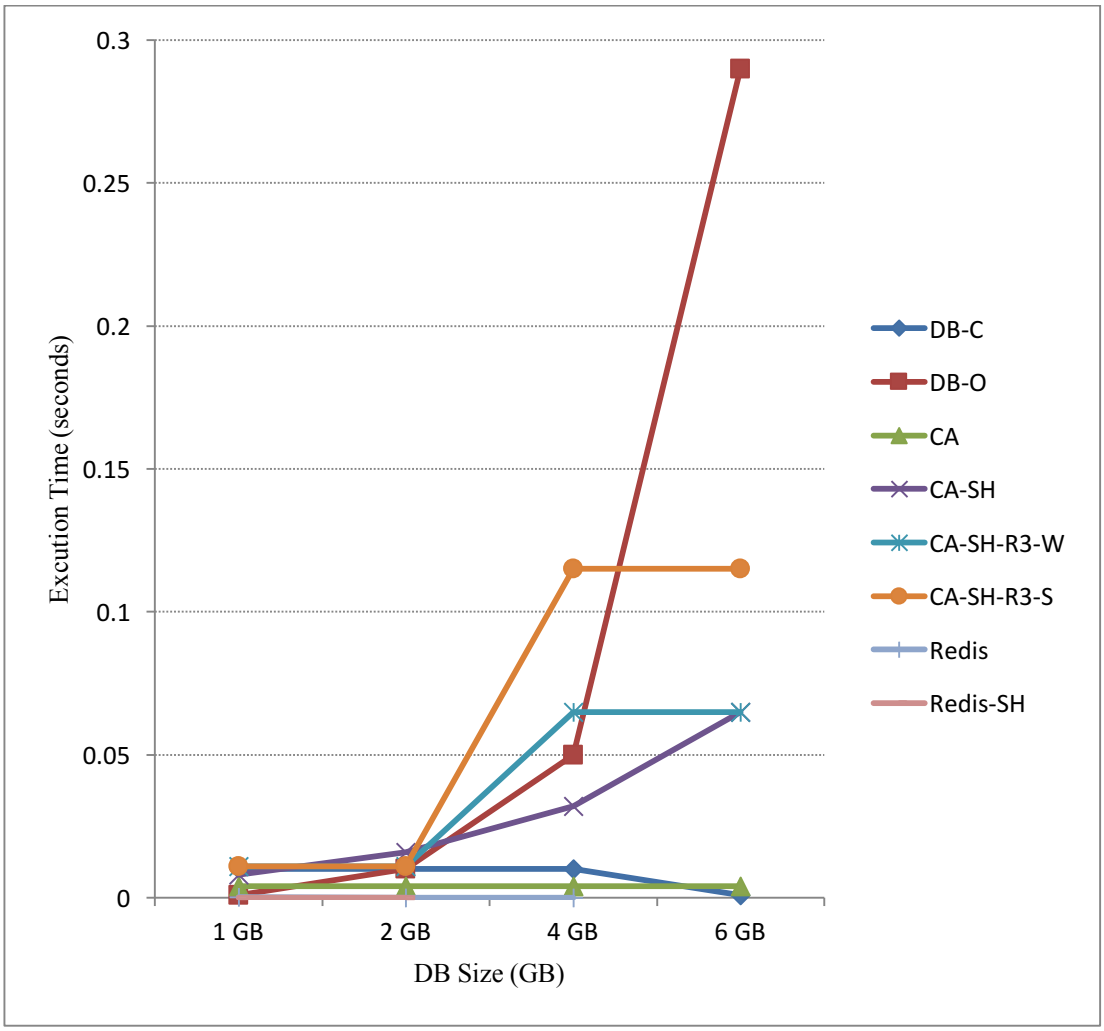

Fig.5.6. Performance of Q1 with sensor key and measured value indexes

The third experiment of key lookup included building an extra indexing on the 'mv' column. As a result of this experiment, DB-C, CA, Redis and Redis-SH were not affected compared with previous

experiments that was because the column in which the index was built on, was not used in this query. Due to the previously mentioned reason, Redis and Redis-SH had indexing congenitally on all experiments. Other systems were negatively affected in this experiment. In Cassandra, creating indexing on MV column did not affect the steps of data fetching since the data was partitioned by (m,s,bt), and Cassandra depends on the Q1 that directly touches the node with the query values (m,s,bt). This information was obtained by looking at the execution plan of all systems by enabling 'tracing on' feature. By looking at the table features using 'describe table_name' command on CQL, the data in the table was ordered internally in each cluster by 'mv' ascending without even creating secondary indexing on it. The only speculation for this delay of the performance in these experiments that creating a secondary index on 'mv' column caused an increase in the size of the table which in turn led to increase the size of the database and therefore hitting the performance by absorbing the server resources.

A summary of all Basic Selection experiment is presented in table 5.2 in which the performance of each system was evaluated with respect to the performance of other systems within this experiment. The evaluation is presented column wise.

System\Indexing Strategy	No Index (Fig.5.4.)	Sensor key Index (Fig.5.5.)	Sensor Key & MV indexes (Fig.5.6.)
DB-C	Good	Good	Very Good
DB-O	Very Bad	Very Good	Good
CA	Not apply	Good	Very Good
CA-SH	Not apply	Good	Good
CA-SH-R3-W	Not apply	Good	Good
CA-SH-R3-S	Not apply	Good	Good
Redis	Very Good	Very Good	Very Good
Redis-SH	Very Good	Very Good	Very Good

Table 5.2: Summary of the Basic Selection experiments**

** The evaluation in this table is based on the overall numerical results of the experiments (see Appendices F, G, H, I).

5.3 Range Search Experiment Results

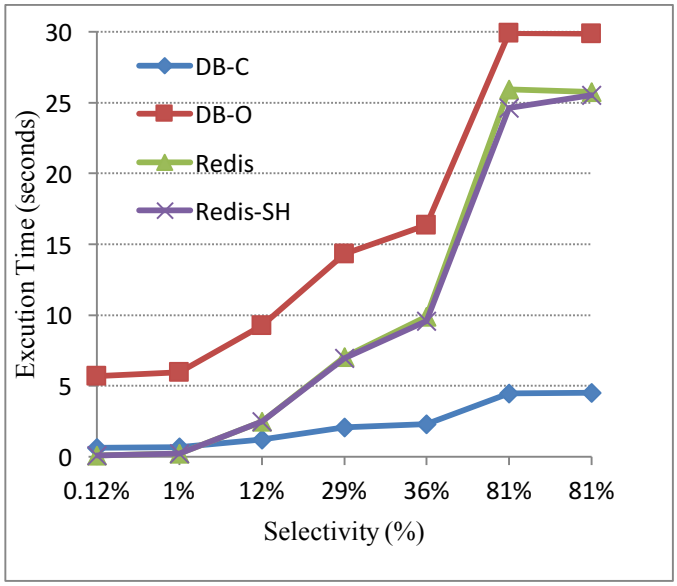

Fig.5.7. Performance of Q2 without indexing for 1 GB

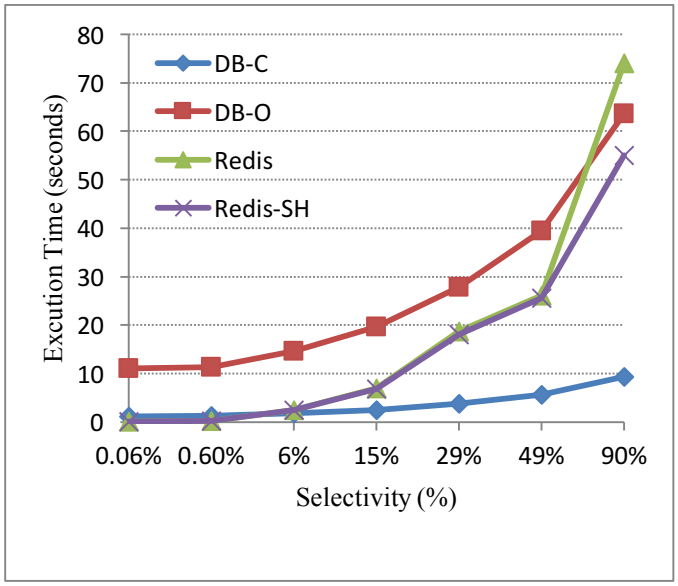

Fig.5.8. Performance of Q2 without indexing for 2 GB

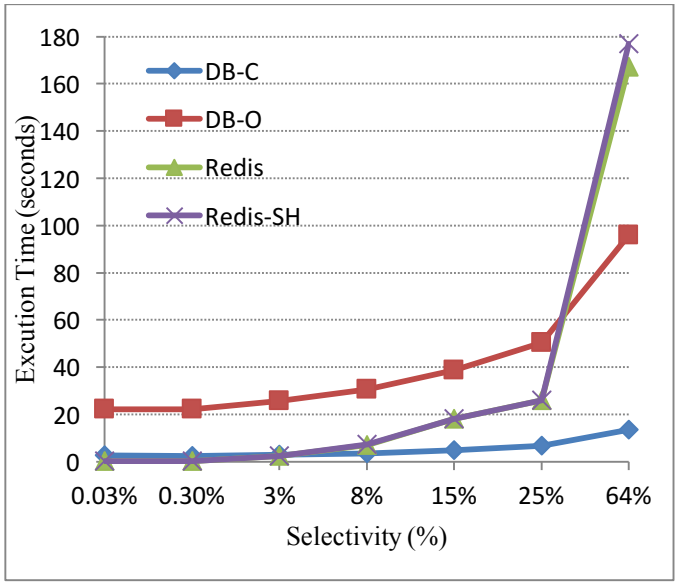

Fig.5.9. Performance of Q2 without indexing for 4 GB

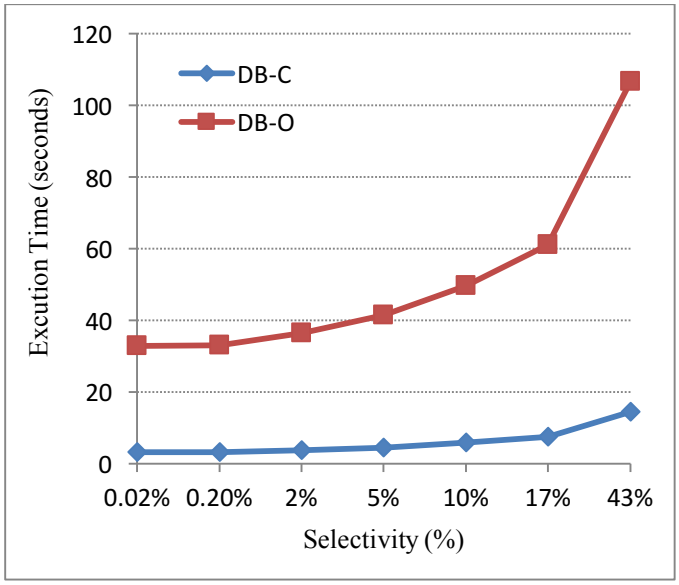

Fig.5.10. Performance of Q2 without indexing for 6 GB

First experiments of Range Search query (Q2) were conducted without indexing for 1GB, 2GB, 4GB and 6GB of datasets. As shown in the above figures (Fig.5.7, Fig.5.8, Fig.5.9, Fig.5.10), although there were no indexes, DB-C was faster in all experiments compared with other systems while the DB-O was the worst in all experiments. However, results of Redis and Redis-SH were not expected because in spite of the fact that Sorted Set structure used for Q2 was stored sorted in-memory. retrieving the data on large range of selectivity was worst than DB-O but they performed fast at lower range of selectivity. Since the sizes of memory in these experiments were 7 GB, 13.5 GB and 12.65 GB (only part 2) for 1GB, 2GB and 4GB datasets respectively, this delay could be due to full usage of the device memory and so there was no space left for querying, transferring, swapping and buffering the datasets results. In addition, this delay could be due to the driver used in the API since Q2 was executed from API JAVA client. The driver used in this experiment was JEDIS which is highly recommended by Redis.

A summary of all Q2 without indexing experiment is presented in table 5.3 in which the performance of each system was evaluated with respect to the performance of other systems within this experiment

System\Data Sets	1GB (Fig.5.7.)	2GB (Fig.5.8.)	4GB (Fig.5.9.)	6GB (Fig.5.10.)
DB-C	Very Good	Very Good	Very Good	Very Good
DB-O	Very Bad	Very Bad	Very Bad	Very Bad
CA	Not apply	Not apply	Not apply	Not apply
CA-SH	Not apply	Not apply	Not apply	Not apply
CA-SH-R3-W	Not apply	Not apply	Not apply	Not apply
CA-SH-R3-S	Not apply	Not apply	Not apply	Not apply
Redis	Good\Bad*	Good\Bad*	Good\Bad*	Not apply
Redis-SH	Good\Bad*	Good\Bad*	Good\Bad*	Not apply

Table 5.3: Summary of the Q2 without indexing**

*Good\Bad: it is good where the selectivity range is small, and becomes bad when the range of the selectivity increases.

** The evaluation in this table is based on the overall numerical results of the experiments (see Appendices F, G, H, I).

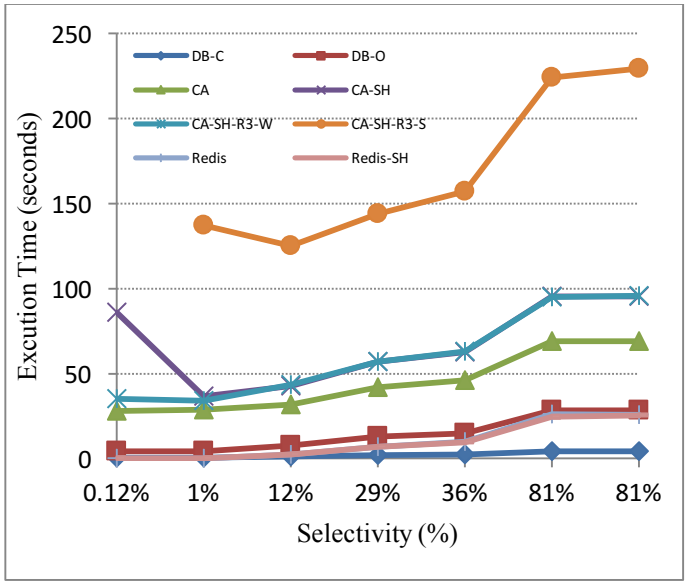

Fig.5.11. Performance of Q2 with sensor key index for 1GB

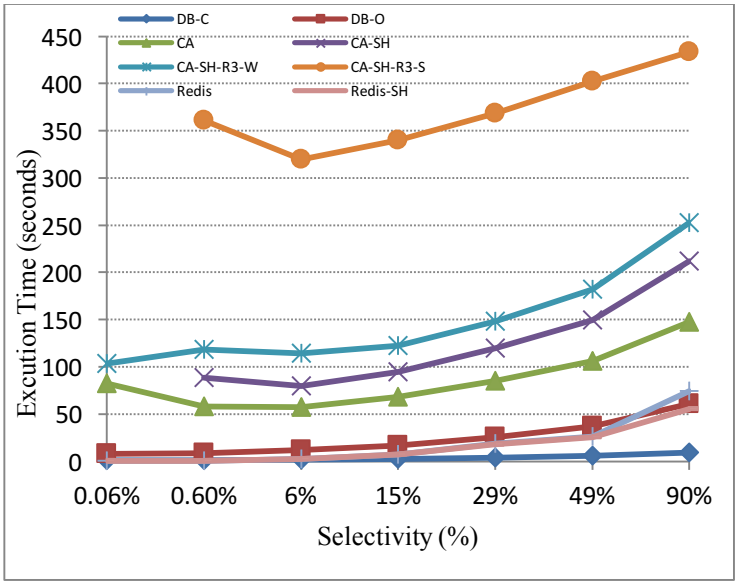

Fig.5.12. Performance of Q2 with sensor key index for 2GB

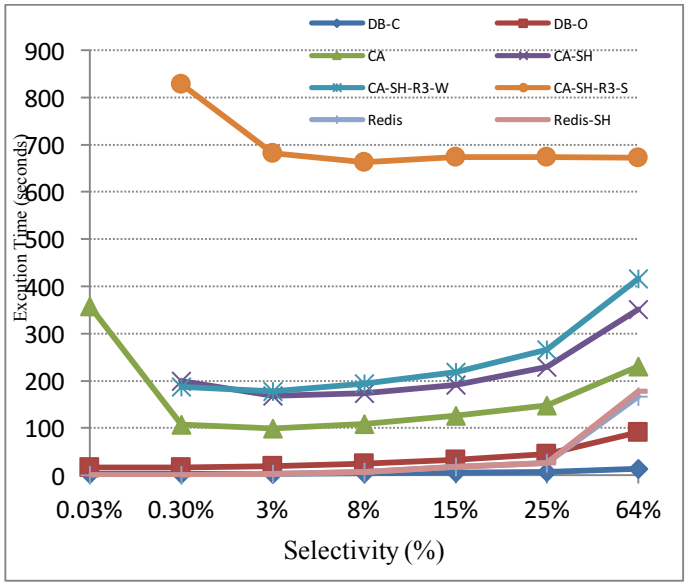

Fig.5.13. Performance of Q2 with sensor key index for 4GB

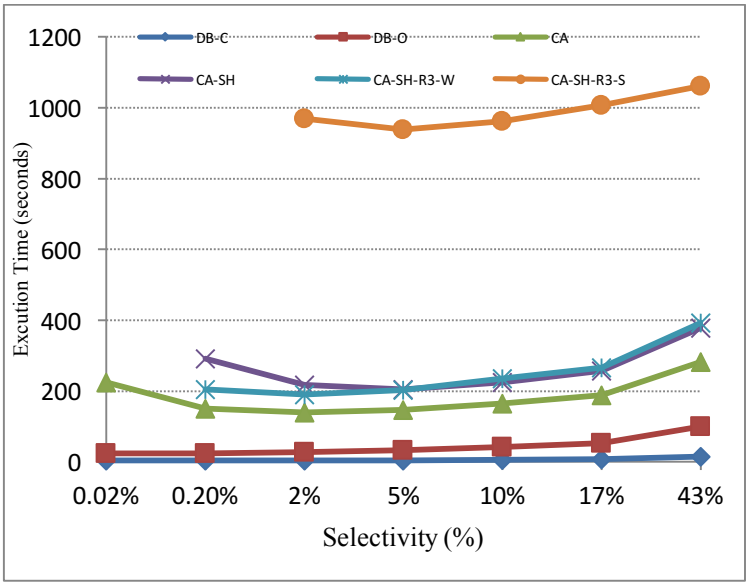

Fig.5.14. Performance of Q2 with sensor key index for 6GB

Figures 5.11, 5.12, 5.13, 5.14 show the performance of range query (Q2) from client side with primary sensor key index. The selectivities varied in each size of data as it is shown in the graphs. It is obvious from all above graphs that DB-C beaten all systems, but contrary to expectations, the results of DB-C were not affected by building the primary index when compared to no indexing experiment. Row prefetching that this system characterized by could be the reason and this feature may overwhelm the building of primary index on the column 'mv' which Q2 mainly depends on. In Contrast, the ratio of performance in DB-O increased slightly compared with no indexing at all. As in last experiments, both Redis & Redis-SH started good at low range of selectivity, while they

performed very bad when the selectivity became higher. All systems belong to Cassandra in this experiments performed very bad, for example, at 4GB of dataset size and at 64% of selectivity, their results in seconds were 230, 350, 416, 672 for CA, CA-SH, CA-SH-R3-W, CA-SH-R3-S respectively, compared to the results of DB-C and DB-O at same point 13, 90 consecutively. In addition, most of the Cassandra systems had problems at very low selectivity in most of the data sizes due to their prolonged spinning inside the systems searching for the results in each node, cluster and partition and using all the memory and the CPU until the system crashed and shutdown automatically and since Cassandra systems are based on Java, this problem might occur due to garbage collection. The summary of these experiments is summarized in Table 5.4 in which the performance of each system was evaluated with respect to the performance of other systems within this experiment.

System\Data Sets	1GB (Fig.5.11.)	2GB (Fig.5.12.)	4GB (Fig.5.13.)	6GB (Fig.5.14.)
DB-C	Very Good	Very Good	Very Good	Very Good
DB-O	Good\Bad*	Good\Bad*	Good\Bad*	Bad
CA	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH-R3-W	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH-R3-S	Very Bad	Very Bad	Very Bad	Very Bad
Redis	Good\Bad*	Good\Bad*	Good\Bad*	Not apply
Redis-SH	Good\Bad*	Good\Bad*	Good\Bad*	Not apply

Table 5.4: Summary of the Q2 with sensor key index**

*Good\Bad: it is good where the selectivity range is small, and becomes bad when the range of the selectivity increases.
** The evaluation in this table is based on the overall numerical results of the experiments (see Appendices F, G, H, I).

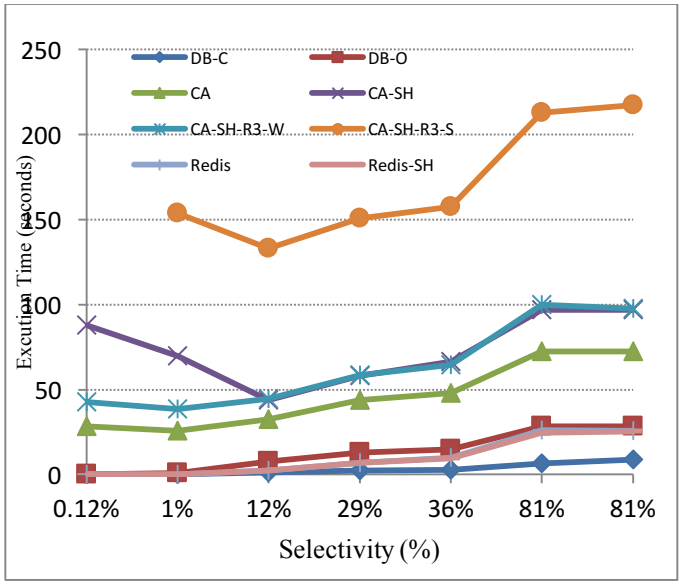

Fig.5.15. Performance of Q2 with sensor key measured value for 1GB

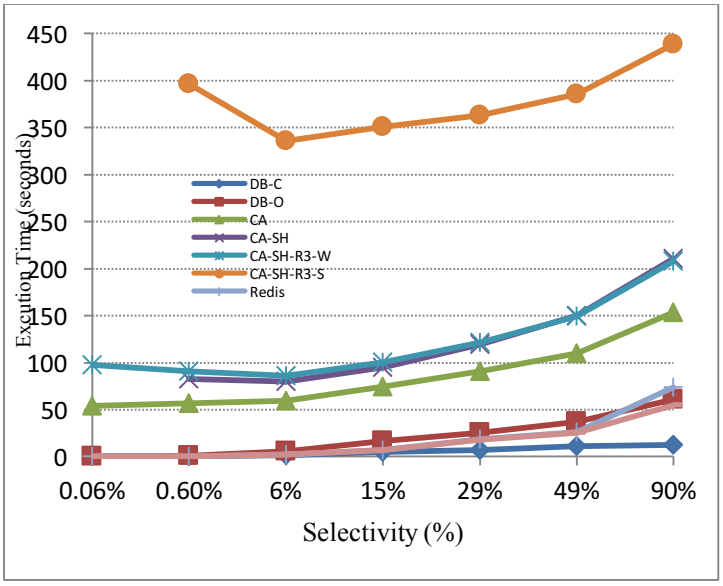

Fig.5.16. Performance of Q2 with sensor key and measured and value for 2GB

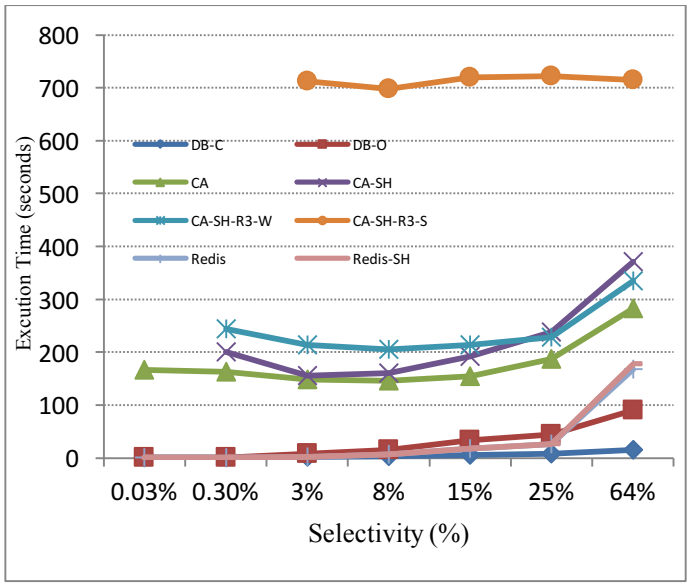

Fig.5.17. Performance of Q2 with sensor key and measured value for 4GB

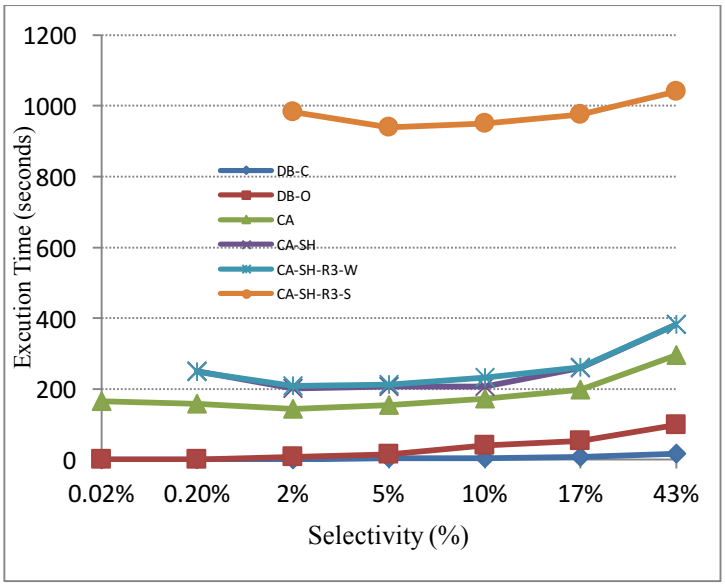

Fig.5.18. Performance of Q2 with sensor key and measured value for 6GB

The overall summary of the Q2 with sensor key and measured value indexes, illustrated in Table 5, indicate that the results of this evaluation are similar to that of the experiments conducted previously without adding 'mv' indexes. The DB-C evaluation as usual was very good in all experiments related to this part. There was significant performance in DB-C by adding the 'mv' index which Q2 depends on. This significant improvement in the performance was seen at the first four selectivities queries then it started to weaken. This was in comparison with the same points in previous experiments where there was no secondary indexing on that column. For

instance, at 4GB of data with only sensor key index, the selectivity results of DB-C were 2.144, 2.21, 2.73, 3.502, 4.831, 6.584, 13.606, while after adding a secondary index on 'mv' at the same point, the results were as following 0.246, 0.301, 1.646, 2.77, 5.225, 8.144, 14.93. Note that the sets number of results at each of the last points are 23736, 213624, 2326128, 5577960, 10894824, 18388477, 47631229 rows. The reason for this variation in the performance among the selectivities could be explained by the large volume of information in the results sets that must be transferred between the database system and the API. Redis, Redis-SH and DB-O had similar evaluation, they performed fast at low selectivity and then their performance went down as the selectivity increased. This evaluation showed that CA-SH-R3-S performance was the worst, since it has 3 replicas from which the results must be collected and compared before replying to the client. As a result, its performance reached up to more than 1000 seconds at 6GB of datasets when the selectivity was 43%. Creating an index on a clustering column in Cassandra as in our scenario case of our experiments had very bad influence on the performance of aggregation queries since it had to check all partitions in all clusters searching for values of 'mv' within the selectivity range. CA system was preferable compared to CA-SH and CA-SH-R3-W which they showed identical results .

System\Data Sets	1GB (Fig.5.15.)	2GB (Fig.5.16.)	4GB (Fig.5.17.)	6GB (Fig.5.18.)
DB-C	Very Good	Very Good	Very Good	Very Good
DB-O	Good\Bad*	Good\Bad*	Good\Bad*	Good\Bad*
CA	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH-R3-W	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH-R3-S	Very Bad	Very Bad	Very Bad	Very Bad
Redis	Good\Bad*	Good\Bad*	Good\Bad*	Not apply
Redis-SH	Good\Bad*	Good\Bad*	Good\Bad*	Not apply

Table 5.5: Summary of the Q2 with sensor key and measured value indexes**

*Good\Bad: it is good where the selectivity range is small, and becomes bad when the range of the selectivity increases.
** The evaluation in this table is based on the overall numerical results of the experiments (see Appendices F, G, H, I).

5.4 Aggregation Query Experiment Results

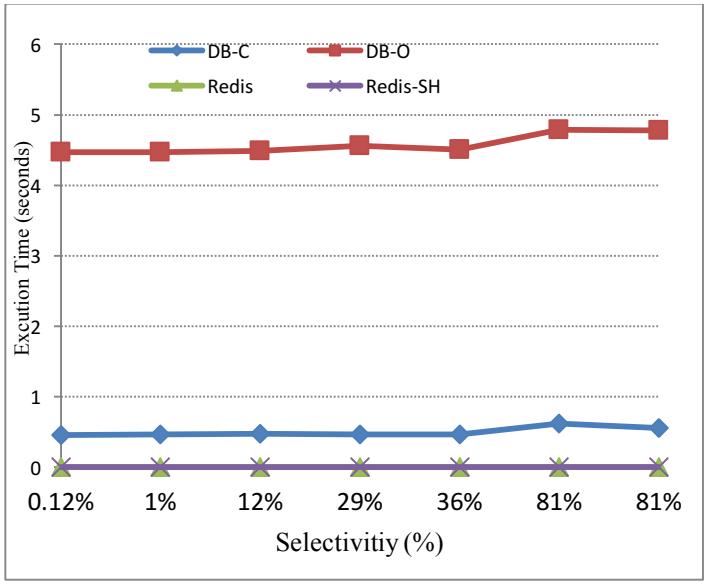

Fig.5.19. Performance of Q3 without indexing for 1 GB

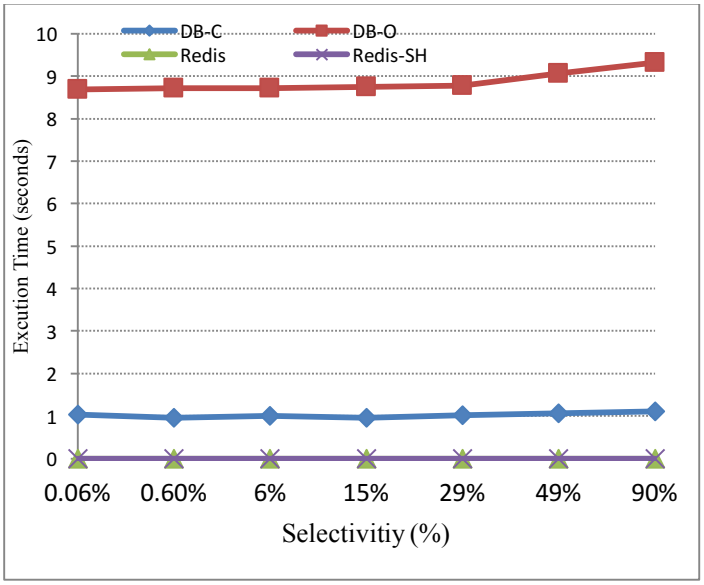

Fig.5.20. Performance of Q3 without indexing for 2 GB

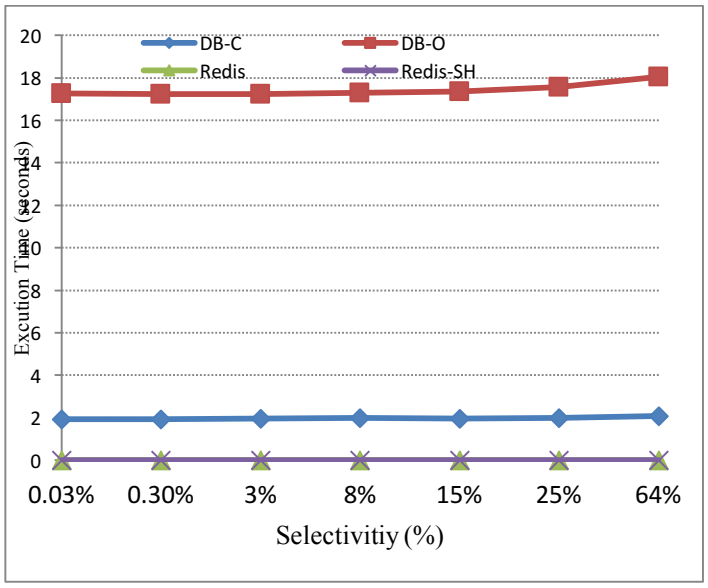

Fig.5.21. Performance of Q3 without indexing for 4 GB

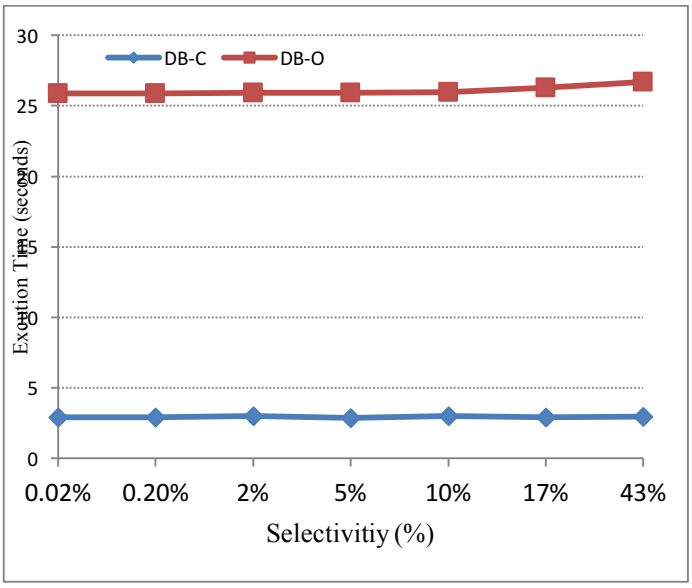

Fig.5.22. Performance of Q3 without indexing for 6 GB

The performance results of the aggregation query (Q3) without indexing for the investigated systems are illustrated in Figures 5.19, 5.20, 5.21, 5.22. Same selectivity ranges of Q2 were used in these experiments excluding API client, i.e. the reading was directly done from the systems shells. Cassandra systems were excluded from these experiments since they have to be indexed all the time by basic indexing strategy such as primary key which is used as partitioning key

later. Here we found that both Redis and Redis-SH performed super fast compared to previous experiments of Q2 which could be explained by the elimination of data transfer between the system and the API client. The execution time of all selectivities in 1GB, 2GB and 4GB datasets was negligible (0.1 millisecond). Their results as usual are identical since Redis is known to redirect the shell client user to the node or shards where the key-value is stored. The high speed of data fetching could also be explained by the structure of storing the key-value in Sorted Set which as this name implies its sorting depends on the 'mv' value. The performance of DB-C was in the second position in these experiments and it was much faster when compared to Q2 at the same level especially at high range of selectivity. Although there was no API client in this investigation, the DB-O still scaled worse than other systems but better than itself in Q2.

System\Data Sets	1GB (Fig.5.19.)	2GB (Fig.5.20.)	4GB (Fig.5.21.)	6GB (Fig.5.22.)
DB-C	Good	Good	Good	Good
DB-O	Bad	Bad	Bad	Bad
CA	Not apply	Not apply	Not apply	Not apply
CA-SH	Not apply	Not apply	Not apply	Not apply
CA-SH-R3-W	Not apply	Not apply	Not apply	Not apply
CA-SH-R3-S	Not apply	Not apply	Not apply	Not apply
Redis	Very Good	Very Good	Very Good	Not apply
Redis-SH	Very Good	Very Good	Very Good	Not apply

Table 5.6: Summary of the Q3 without indexing**

** The evaluation in this table is based on the overall numerical results of the experiments (see Appendices F, G, H, I).

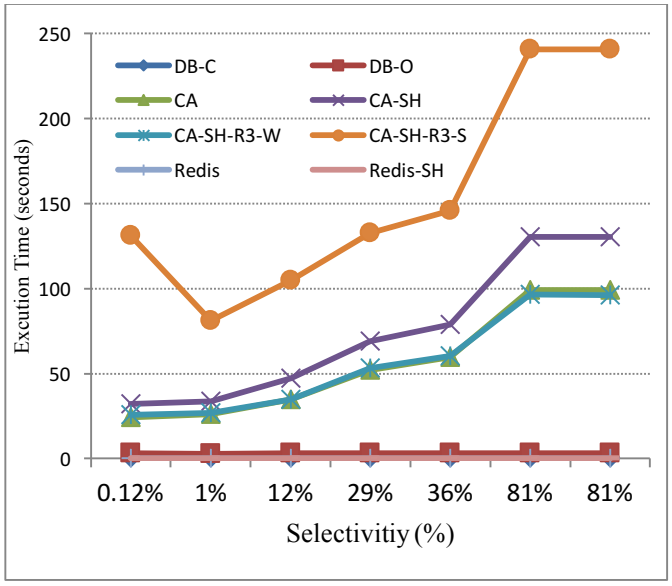

Fig.5.23. Performance of Q3 with sensor key index for 1GB

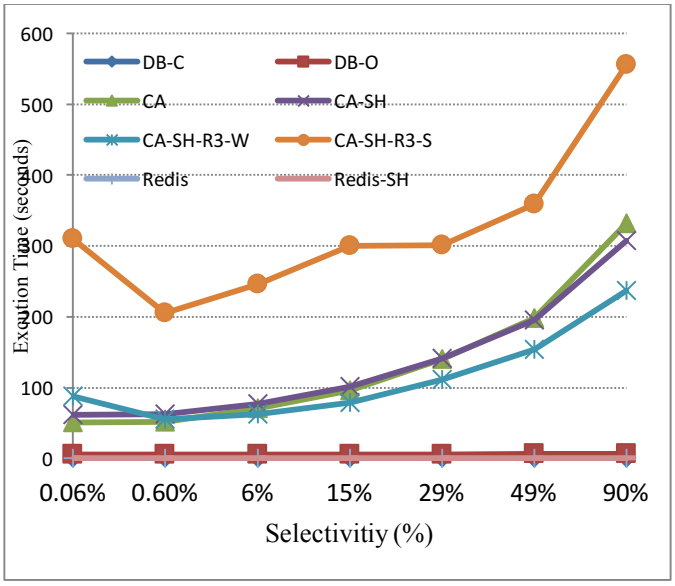

Fig.5.24. Performance of Q3 with sensor key index for 2GB

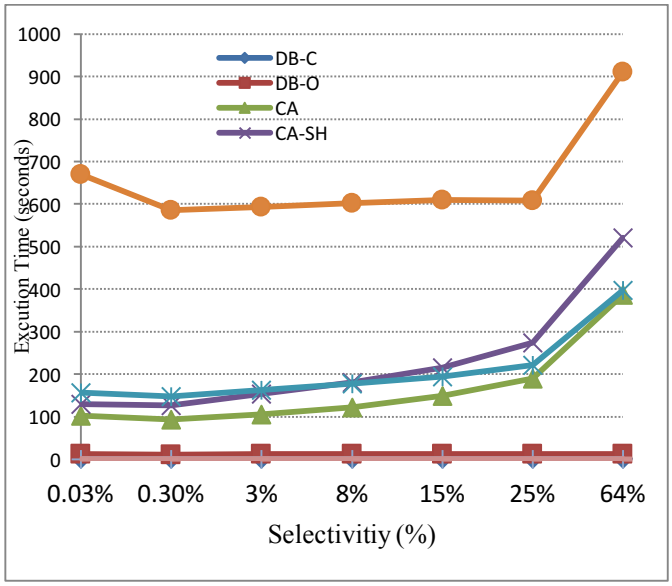

Fig.5.25. Performance of Q3 with sensor key index for 4GB

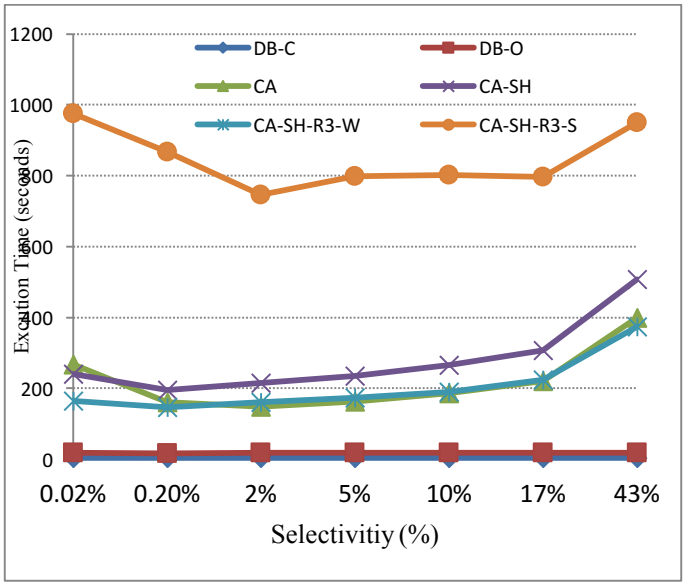

Fig.5.26. Performance of Q3 with sensor key index for 6GB

The results of adding primary indexing to the systems before executing Q3 are presented in figures 5.23, 5.24, 5.25 and 5.26. The Redis and Redis-SH were also the leaders in the performance among other systems. The results of adding primary indexing or not to DB-O were found to be identical which indicates that this indexing had completely no effect on DB-O performance, yet its performance stayed higher than DB-O and Cassandra systems at all data sizes of these experiments, therefore it deserved to be the second best. Although building primary index was not expected to be advantageous for Q3 performance of the systems, DB-O

showed the opposite and performed better than without it. The reason could be due to that the data was sorted and clustered in the table based on primary indexing which facilitates data fetching compared without primary composite index. Despite of this improvement, its performance stayed way below the performance of Redis, Redis-SH and DB-O. All figures indicate that Cassandra systems (CA, CA-SH, CA-SH-R3-W, CA-SH-R3-S) had problem to scale with our application scenario in both Q2 and Q3 which made their performance to be the worst compared to Redis, Redis-SH, DB-C and DB-O. Apparently, this issue is expected from the Cassandra developers since it is reported in their user guide that running queries such as (Q2,Q3) with 'ALLOW FILTERING' option in Cassandra Query Language (CQL) shell has performance issues because it has to touch all clusters and partitions of all rings in order to search for the results. Among Cassandra systems, CA, CA-SH, CA-SH-R3-W, CA-SH-R3-S are respectively ordered from high to low based on their performance.

System\Data Sets	1GB (Fig.5.23.)	2GB (Fig.5.24.)	4GB (Fig.5.25.)	6GB (Fig.5.26.)
DB-C	Good	Good	Good	Good
DB-O	Bad	Bad	Bad	Bad
CA	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH-R3-W	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH-R3-S	Very Bad	Very Bad	Very Bad	Very Bad
Redis	Very Good	Very Good	Very Good	Not apply
Redis-SH	Very Good	Very Good	Very Good	Not apply

Table 5.7: Summary of the Q3 with sensor key index**

** The evaluation in this table is based on the overall numerical results of the experiments (see Appendices F, G, H, I).

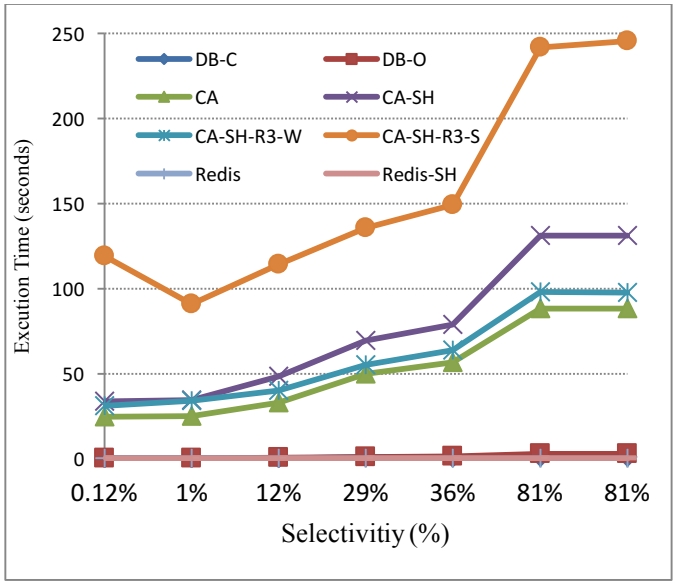

Fig.5.27. Performance of Q3 with sensor key and measured value indexes for 1GB

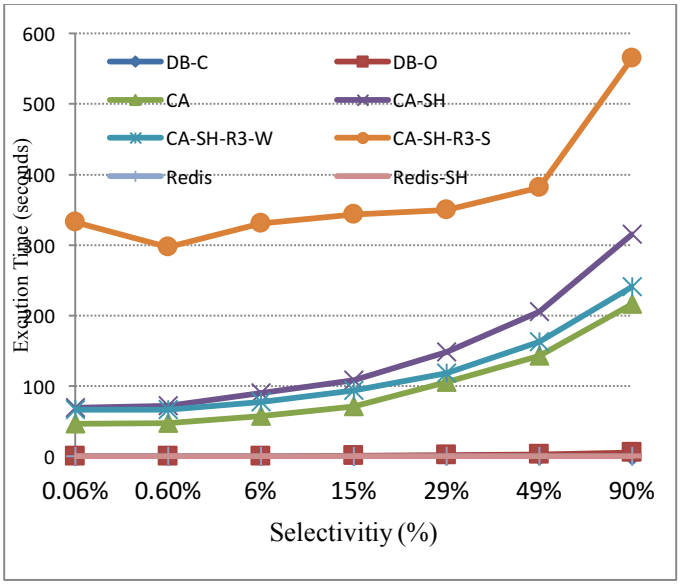

Fig.5.28. Performance of Q3 with sensor key and measured value indexes for 2GB

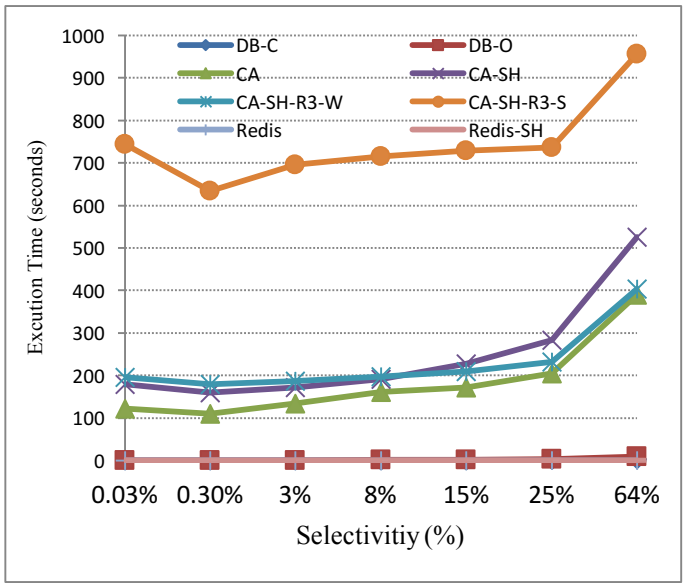

Fig.5.29. Performance of Q3 with sensor key and measured value indexes for 4GB

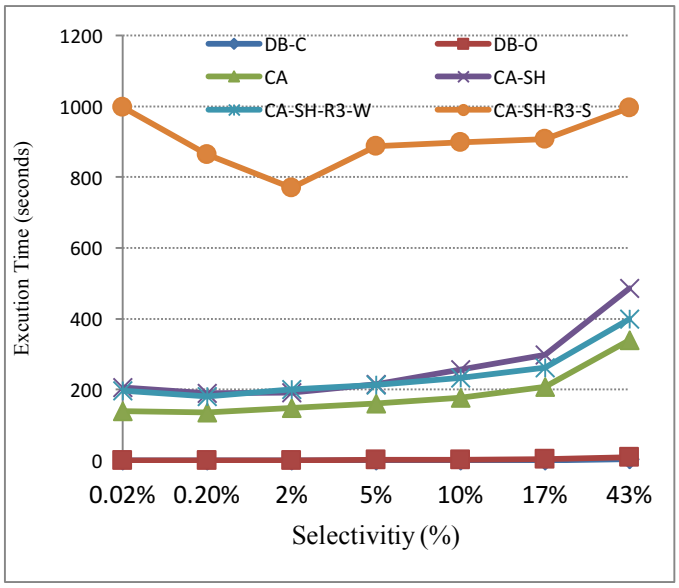

Fig.5.30. Performance of Q3 with sensor key and measured value indexes for 6GB

Last experiment investigated Q3 in all systems with sensor key and measured value, the results of this experiment are illustrated in figures 5.27, 5.28, 5.29 and 5.30. All figures show that Redis and Redis-SH had performance incomparable with all systems, for this reason, their scores remained top the list. DB-C and DB-O scaled much butter compared to their performance when there was no secondary indexing on 'mv', still the DB-C performed around 4 times faster than DB-O. Adding a secondary index to all Cassandra systems had no significant effect on the performance and the results seemed to be almost similar when compared with previous

experiments without measured value indexing. Therefore, secondary indexing cannot be used to improve the performance instead it is more useful for making querying by some columns possible. Because these queries end up by hitting all partitions and clusters searching for the results, this made it very slow for our case problem to run Q2 and Q3.

System\Data Sets	1GB (Fig.5.27.)	2GB (Fig.5.28.)	4GB (Fig.5.29.)	6GB (Fig.5.30.)
DB-C	Very Good	Very Good	Very Good	Very Good
DB-O	Good	Good	Good	Good
CA	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH-R3-W	Very Bad	Very Bad	Very Bad	Very Bad
CA-SH-R3-S	Very Bad	Very Bad	Very Bad	Very Bad
Redis	Very Good	Very Good	Very Good	Not apply
Redis-SH	Very Good	Very Good	Very Good	Not apply

Table 5.8: Summary of the Q3 with sensor key and measured value**

** The evaluation in this table is based on the overall numerical results of the experiments (see Appendices F, G, H, I).

6- Analyses and Discussion

This section discusses and summarizes the results based on answering the research questions included in the project proposal.

1- How suitable the database systems for loading and analyzing of large-scale persistent logs?

For RDBMS bulk loading, as illustrated in figures 1, 2 and 3, in the absence of indexing, both DB-C and DB-O performed better among other systems while they performed less in the presence of indexing before the loading although their performance stayed higher than CA, CA-SH, CA-SH-R3-W and CA-SH-R3-S when both sensor key and measured value indexes were existing. The slowness of bulk loading of RDBMS with indexing was due to the distributions of data based on indexing at the database tables, 33 and 35 minutes for bulk loading 74,550,000 records of row data in DB-C and DB-O respectively with both primary and secondary indexing.

Redis and Redis-SH had same performance in bulk loading since the system has no features to distribute the data concurrently to all shards of Redis. Redis has no primary or secondary indexing strategies since its key-value feature is sorted by nature. Both Redis systems performed slower compared with non-indexed DB-C & DB-O but then they started to beat DB-C & DB-O when primary and secondary indexing were built. In contrast, both Redis systems performances were at the same level of most Cassandra systems except CA which has single node and hence it lacks the concurrent bulk loading feature. In the last experiments of bulk loading where both sensor and measured value indexes system were existing, the Redis systems had the best loading time and they took around 14 minutes for bulk loading 74,550,000 records of row data. However, they required around 6 times of memory (RAM) than row data sizes since they keep all the datasets information into the RAM for fast retrieving performance.

For Cassandra systems bulk-loading, the effective way to load large-scale data into Cassandra was to use bulk loader tool that was useful for only transferring historical data from CSV file or similar. Nevertheless, in our real application, the bulk-loading or large-scale insertion is directly done from the sensors of the factories' machines which must pass by all normal insertion process (Comitlog, Memtables, SSTables) and this process requires huge time in comparison with using bulk-loader tool. Moreover, SSTable loader tool is not that efficient when the tables in which data is inserted has one or more secondary indexes. As experimented above, bulk loading in

Cassandra with secondary indexing strategies is time consuming compared with only primary indexing. Overall results of Cassandra in these experiments indicate that CA is the least efficient among the Cassandra systems with shards or clusters because Cassandra with Cassandra Cluster Manger (CCM) clustering performed 4 times better compared with no shards, this was due to the concurrent bulk loading into 4 nodes at once, while in CA the data were queued for bulk loading in single node. CA-SH was much faster than all systems including Redis due to the concurrent bulk loading reason just mentioned in addition to the absence of replication. Unfortunately, all Cassandra systems failed to be compared with other systems when they had a secondary indexing. It was observed that after data bulk loading using SSTable loader, Cassandra systems spent more time to rebuild and distribute the data based on the secondary indexes. In order to analyze persistent logs in Cassandra systems, developers have to plan before creating the tables or column-family in which queries of analyzation are run since the distribution of partition keys and primary keys are important with querying and analyzing.

All systems found to be suitable and fast for look up queries that match the records of primary key elements which in turn are the key in the key-value systems used in this comparison. However, when there was no primary index in DB-O, its performance tended to decline at this stage.

Overall results of selective range query (Q2) and selective aggregation query (Q3) showed that both Redis systems performed much better than others due to their capabilities of fetching and retrieving the data from memory instead of disk as most systems do. Moreover, they keep the data sorted in Sorted Set structure in memory, a feature that makes it effective without competition. In this part of the investigation, DB-C had the second best performance compared with others. Although it was shown in [1] that the relational DBMS from commercial vendor having sophisticated query optimizer, the current investigation did not encounter the aforementioned query optimizer in DB-C. Obviously, all results of DB-C were linear even with large selectivity of mass sizes of databases. DB-O performance was bad for Q2 & Q3 when there was no primary and secondary indexing but it scaled better after indexing DB-O exceeded the expectations and defeated all Cassandra systems because of the issues regarding Cassandra structures for our application scenario and also because of the table optimization feature that was used in DB-O which had advantageous impact on the performance. Therefore, DB-O scored the third in these benchmarking experiments.

Although in [39] benchmark found that Cassandra is fast in reading than writing, this study shows that reading data from various Cassandra systems setups is time consuming based on our application scenario for Q2 and Q3. The reason behind that is due to the complicated distribution structure of the data within the column-family, i.e. the data in Cassandra systems has to be divided and stored in different rings or partitions based on the partition keys then it is clustered within each partition locally based on the cluster keys. As a result, searching for a column value within a cluster without giving any values for the partitioning keys as in Q2 and Q3 has to hit all rings and partitions searching for the selectivity range values. Compared to systems belong to Cassandra (CA, CA-SH, CA-SH-R3-W, CA-SH-R3-S), it was found that CA performed better than others in Q2 and Q3 since it is running in single node and it does not have to forward the request to all nodes and wait for their responses. In addition, CA-SH and CA-SH-R3-W had mostly identical results since both of them forward the requests to the nodes and read one reply per each node or cluster, while in CA-SH-R3-S, the system has to wait for all replies from all three replicas and compares the data results for strong consistency, this makes CA-SH-R3-S at the bottom of the performance list.

2. What is the impact of different indexing strategies?

Redis uses zero user indexing, it is a key-value store system, keys are the main part, and values are nothing, Redis does not allow the user to query object's values or store data based on values indexing. Therefore, the programmer has to define the keys according to the querying needs. However, Redis gives the programmer the opportunity to define the values which have to be queried as range values to be under one key name structure called Sorted Set, these values are ordered by default to be faster during retrieval.

Both state-of-art RDBMS investigated in this project support primary and secondary indexing strategies. The experiments of our investigation found that adding primary index and measured value secondary index, combined or scattered, negatively influenced the bulk loading in both systems. However, they scaled much better for basic selection or key lookup query Q1 after the addition of the sensor key composite index and they scaled even better when adding the secondary index on measured value column for Q2 and Q3. It was expected that adding the primary index on the composite key (m,s,bt) should not affect Q2 & Q3 which depends on 'mv' values but apparently DB-O was found to be affected by this indexing mainly due to data organisation within the table which accidentally affects fetching the mv data.

Cassandra provides both primary indexing which is a must to create the table or as it is called, column family. Unlike most of NoSQL databases, Cassandra provides a secondary indexing for their columns but based on the current investigations, the secondary indexing in Cassandra was given the name 'semi secondary index' since its existence is only for convenience not for performance. All benchmark queries Q1, Q2 and Q3 were distorted by building the secondary indexing including the bulk loading. Creation of primary key index in Cassandra should be carefully performed since the user querying and data storing structure depend on it.

3. How sophisticated is the query optimizer for the investigated databases in choosing the appropriate execution plan for scalable query execution?

It was claimed in [1] that the commercial RDBMS has query optimizer which is advantageous to its performance by enabling it to shift to full scanning when the selectivity retrieval depends on the indexing is more expensive than full table scan. In all investigated systems, the presence of query optimizer was almost non-existent and it is clearly seen in all graphs and figures that our systems used only indexing execution plan for scalability. Therefore, the query optimizer in previous DB-C [1] is better than the query optimizer of the current DB-C.

4. What is the impact of relaxing consistency in loading and analyzing persistent logs?

It was evident in [1] that relaxing consistency does not improve the performance for both RDBMS, accordingly, only one level of consistency was applied for comparison in this project. Redis is simple and its single instance is always consistent and strong, however, in Redis shards or clusters where the replication of the data is distributed among slaves replicas, the consistency becomes very weak so it does not guarantee strong consistency. This is explained by the asynchronous replication that Redis uses [27] and it is possible in certain conditions that one of Redis clusters loses writes that were acknowledged by the system to the client but not yet replicated on the slaves and that slave which loses the writes can be automatically a master if the main master gets crashed or failed to respond. Note that WAIT command does not make Redis cluster strongly consistent [28] and it does not prevent a slave that was not able to receive the write to be a master, this problem was encountered when investigating Redis with replicas. Due to memory limitation that I had in our machine environment of the benchmarking, it was not possible to perform the clustering with replica investigation since that required around 14 GB of

RAM to make only one replica of 1GB of data and the memory size which was occupied by 1 GB without replica was 7 GB of RAM in single mode server and around 9 GB of RAM in cluster mode (shards). That means at least 6X of RAM is needed to make the replica with shards for our case data. The only investigated consistency here was the strong consistency and the expectation was to find no impact on the performance of reading and writing with replication (master-slaves) mode since Redis redirects the clients to the node where the needed key-value is stored.

Cassandra has different consistency levels as explained in details in Cassandra section in this report. Both weakest and strongest consistencies investigated in this study had no influence on loading and analyzing of our datasets when there was single node used as in CA, or even in multiple nodes with only one replica as in CA-SH. The impact of consistencies appeared within multiple nodes and replicas as in CA-SH-R3-W and CA-SH-R3-S. The weakest read consistency resulted in significant fast performance comparing with the strongest consistency in same system setup in analyzing persistent logs as clearly seen from experiments Q2 and Q3. This observation was in response to the long waiting time for all dataset results from all 3 replicas and comparing them to the case of strong consistency. On the other hand, in the weakest reading consistency, the system coordinator only waits for any or one fastest response. Since the fastest SSTable tool was used for bulk loading in all our investigated systems, both consistency levels had no impact on bulk loading the logs, that's because SSTable loader directly loads data to the correct nodes and replicas from the backend and does not load through the normal coordinated write process. However, based on theories of Cassandra behavior, bulk loading persistent logs in normal writes process should have a positive effect on the performance in the weakest consistency when compared with the strongest writing consistency.

5. Does data parallelization provide significant performance advances for scalable loading and query execution?

Both investigated state-of-art Redis and Cassandra provide parallelization of data among horizontal clustering or sharding. As explained in results section, both Redis and its shards mode Redis-SH have identical results in both data loading and analyzing mainly due to the simple design of the system in shards mode. Redis-SH redirects the clients to the node or the server where the requested key-value is located, in this situation, the Redis-SH nodes work as standalone servers and they directly reply to the clients. This is in contrast to the parallelization in most of datastores systems which are designed to be as server-client where the server is the coordinator. In addition, for the bulk loading, we found that the tool provided by Redis

community and used for bulk loading data injections had no capabilities to work with Redis in shards or partitions mode in which the data has to be distributed among different nodes. To overcome this problem, a simple program was implemented to distribute the data among the clusters of Redis. This program proved to be faster than PIPE feature provided by Redis-cli in single node. In Summary of Redis, the data parallelization did not provide any performance advantage for both loading and retrieving the data.

In contrast, Cassandra clustering or parallelization, has smart parallel distribution and rebalancing the data among the system cluster nodes, this feature gives Cassandra high level advantage compared to its counterparts in loading scalability, but unfortunately, reading the data from multiple participated nodes of the parallelization is time consuming, this conclusion was drawn based on our results. The delay in data retrieval was speculated to be due to the complicated structure of the data distributions in the columns-family. In addition, when a client connects to any node of the database cluster (peer connection), that node in this case is called the coordinator, if the read request for example has to be read in different node than the coordinator, then the coordinator sends sub-request to the node intended and waits for a reply with the results. Then, the coordinator will send the results to the client. The same thing is applied in the writing scenario in normal process. This complicated bureaucrate design, make the Cassandra systems fail in our both range and aggregation queries.

7- Conclusion and Future work

The results of this research show that there is not a specific type of system consistently outperforming the others, but the best option can vary depending on the features of the data, the type of query and the specific system.

Both in-memory systems Redis and Redis-SH performed well compared to all the other systems when loading and analyzing persistent logs. DB-C also showed a similar good performance. Although the open source RDBMS DB-O also obtained acceptable results it could not compete with the systems mentioned above. All Cassandra systems had a comparably good performance in bulk loading raw data using the Cassandra bulk loader tool (SSTable loader) which loads the data to a live cluster of nodes and transfers the relevant part of the data to each node and replica in parallel, but their performance degraded when secondary indexing were created before the bulk loading. All Cassandra systems performed poorly in aggregation and range queries but they were competitive in key lookup queries retrieving a particular record.

Primary and secondary index utilization in both RDBMSs resulted in good performances for looking up and matching a specific key and for retrieving a selective range of data as in our second and third queries. However, creating these indexes before the bulk loading had a negative impact on the performance. Both Redis and Redis-SH have no built-in primary or secondary indexes, however the key structure can be used as a primary key access as in our basic selection query where our key was the composite key. In addition, Redis data structures can be used to work as secondary indexes as in our second and third queries. In these cases the key structure identifies how the data are sorted. Cassandra systems provide both primary indexes and secondary indexes but unfortunately a secondary index of Cassandra does not provide full secondary indexing as in RDBMSs We refer to this as a ‘semi-secondary index’. The primary index in Cassandra is a fundamental data structure and the main factor determining the efficiency of data retrieval. Unlike primary keys, secondary indexes in Cassandra were found to be negatively affecting the performance of both loading and querying tasks.

Although the level of reading and writing consistencies had no effect on CA and CA-SH, it significantly influenced both CA-SH-R3-W and CA-SH-R3-S where replicas were used. However, it had no impact on the loading of persistent logs when using a fast data migration tool such as the SSTable loader utility. In contrast, Redis is always consistent since it redirects the clients’ requests to the key-value node. In case of replication, Redis is less consistent but this had

no effect on the performance due to the asynchronous replication behavior characterizing this system.

Data parallelization over multiple shards or nodes had a significant positive effect on the performance of bulk loading the persistent logs in Cassandra systems, due to the parallel distribution behavior of data among nodes of the cluster. However, it negatively affected data analysis and filtering. On the other hand, Redis parallelization had no effects neither on loading nor on analyzing the persistent logs.

As a finale note, these benchmarking experiments have been conducted using mid-range hardware. It would be interesting to repeat the same experiments on high-end machines, with not less than 64 GB of memory for non-sharding experiments, and 32 GB of nodes for sharding experiments. Moreover, client applications could be run from standalone machines other than the server running the database systems, to simulate a more realistic scenario.

Bibliography

1. Mahmood, K., Risch, T., & Zhu, M. (2014). Utilizing a NoSQL Data Store for Scalable Log Analysis. Proceedings of the 19th International Database Engineering & Applications Symposium on - IDEAS '15.
2. Stonebraker, M. (2010). SQL databases v. NoSQL databases. Communications of the ACM Commun. ACM, 53(4), 10.
3. Harizopoulos, S., Abadi, D. J., Madden, S., & Stonebraker, M. (2008). OLTP through the looking glass, and what we found there. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data - SIGMOD '08.
4. Cassandra. (n.d.). Retrieved March 15, 2015, from http://cassandra.apache.org/
5. Redis. (n.d.). Retrieved March 15, 2015, from http://redis.io/
6. Beynon-Davies, P. (2003). Database systems.
7. List Of NoSQL Databases [currently 225]. (n.d.). Retrieved May 15, 2015, from http://nosql-database.org/
8. J. G. (n.d.). The Transaction Concept: Virtues and Limitations. Appeared in Proceedings of Seventh International Conference on Very Large Databases, Sept. 1981. Published by Tandem Computers Incorporated. Retrieved March 15, 2015, from http://research.microsoft.com/en-us/um/people/gray/papers/thetransactionconcept.pdf
9. Codd, E. F. (1970). A relational model of data for large shared data banks. Communications of the ACM Commun. ACM, 13(6), 377-387.
10. Hammes, Dayne, Hiram Medero, and Harrison Mitchell. "Comparison of NoSQL and SQL Databases in the Cloud." Comparison of NoSQL and SQL Databases in the Cloud (2014): Paper 12. AIS Electronic Library (AISeL). SAIS 2014 Proceedings., 2014. Web. 27 Apr. 2016. <http://aisel.aisnet.org/sais2014/12>
11. Brewer, E. A. (2000). Towards robust distributed systems (abstract). Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing - PODC '00, 7-19.
12. Gray, J., & Reuter, A. (1993). Transaction processing: Concepts and techniques. San Mateo, CA: Kaufmann.
13. Pritchett, D. (2008). Base: An Acid Alternative. Queue, 6(3), 48-55. Retrieved March 15, 2015, from http://queue.acm.org/detail.cfm?id=1394128

14. What is Apache Cassandra? (2015). Retrieved March 15, 2015, from http://planetcassandra.org/what-is-apache-cassandra/
15. Cassandra. (2013, June 25). Retrieved March 15, 2015, from http://en.wikipedia.org/wiki/Apache_Cassandra, Most information has been taken from DataStax Enterprise website
16. Cassandra Query Language (CQL) v3.3.1. (n.d.). Retrieved February 15, 2015, from https://cassandra.apache.org/doc/cql3/CQL.html#selectStmt
17. CQL for Cassandra 2.x Documentation. (2016, February 12). Retrieved February 12, 2016, from http://docs.datastax.com/en/cql/3.1/pdf/cql31.pdf
18. Cassandra Administration Tutorial: Write and Read Paths | packtpub.com. (2014, September 19). Retrieved February 16, 2015, from https://www.youtube.com/watch?v=d9NvnMcTVdQ , Part of 'Cassandra Administration' video series, by the packtpub.com channel
19. Roth, G. (2012, October 14). Cassandra by example - the path of read and write requests. Retrieved February 16, 2015, from http://www.slideshare.net/grro/cassandra-by-example-the-path-of-read-and-write-requests
20. Understanding Data Consistency in Apache Cassandra. (2011, November 11). Retrieved February 17, 2015, from http://www.slideshare.net/DataStax/understanding-data-consistency-in-apache-cassandra
21. Using the Cassandra Bulk Loader, Updated. (2014, September 26). Retrieved February 17, 2016, from http://www.datastax.com/dev/blog/using-the-cassandra-bulk-loader-updated
22. A Brief Introduction to Apache Cassandra | DataStax Academy: Free Cassandra Tutorials and Training. (2015, January 2). Retrieved February 17, 2015, from https://academy.datastax.com/demos/brief-introduction-apache-cassandra, This tutorial may updated over time
23. Yukim/cassandra-bulkload-example. (2015, March 5). Retrieved April 17, 2015, from https://github.com/yukim/cassandra-bulkload-example/, This may updated over time
24. Carlson, J. L. (2013). Redis in action [1]. Printed in the United States of America in Manning Publications Co, 2013, ISBN 9781617290855. Retrieved May 25, 2015, from www.finebook.ir/download/book/46/14400/redis-in-action.pdf
25. An introduction to Redis data types and abstractions. (n.d.). Retrieved May 17, 2015, from http://redis.io/topics/data-types-intro This source may change over time, This website is open source software, sponoserd by RedisLabs

26. Secondary indexing with Redis. (n.d.). Retrieved June 17, 2015, from http://redis.io/topics/indexes, This source may change over time, This website is open source software, sponoserd by RedisLabs
27. Redis cluster tutorial. (n.d.). Retrieved June 17, 2015, from http://redis.io/topics/cluster-tutorial, This source may change over time, This website is open source software, sponoserd by RedisLabs
28. WAIT numslaves timeout. (n.d.). Retrieved June 17, 2015, from http://redis.io/commands/wait, This source may change over time, This website is open source software, sponoserd by RedisLabs
29. Redis Persistence. (n.d.). Retrieved June 17, 2015, from http://redis.io/topics/persistence, This source may change over time, This website is open source software, sponoserd by RedisLabs
30. Redis persistence in practice. (2014, January 23). Retrieved June 17, 2015, from http://www.slideshare.net/eugef/redis-persistence-in-practice-1, This source may change over time, This website is open source software, sponoserd by RedisLabs
31. Redis Cluster Specification. (n.d.). Retrieved July 17, 2015, from http://redis.io/topics/cluster-spec, This source may change over time, This website is open source software, sponoserd by RedisLabs
32. Redis Mass Insertion. (n.d.). Retrieved July 17, 2015, from http://redis.io/topics/mass-insert, This source may change over time, This website is open source software, sponoserd by RedisLabs
33. Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., Dewitt, D. J., Madden, S., & Stonebraker, M. (2009). A comparison of approaches to large-scale data analysis. Proceedings of the 35th SIGMOD International Conference on Management of Data - SIGMOD '09, 165-178.
34. Floratou, A., Teletia, N., Dewitt, D. J., Patel, J. M., & Zhang, D. (2012). Can the elephants handle the NoSQL onslaught? Proc. VLDB Endow. Proceedings of the VLDB Endowment, 5(12), 1712-1723.
35. Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking cloud serving systems with YCSB. Proceedings of the 1st ACM Symposium on Cloud Computing - SoCC '10, 143-155.
36. TPC - Benchmarks. (n.d.). Retrieved February 17, 2015, from http://www.tpc.org/information/benchmarks.asp

37. SQL Server. (n.d.). Retrieved February 17, 2016, from https://www.microsoft.com/en-us/server-cloud/products/sql-server/

38. MongoDB. (n.d.). Retrieved July 17, 2015, from http://www.mongodb.org/

39. Kuhlenkamp, J., Klems, M., & Röss, O. (2014). Benchmarking scalability and elasticity of distributed database systems. Proc. VLDB Endow. Proceedings of the VLDB Endowment, 7(12), 1219-1230. Retrieved January 17, 2016, from http://www.vldb.org/pvldb/vol7/p1219-klems.pdf

Appendix A

Application Program Interfaces (API)

```
package transfer;

import java.io.BufferedReader;
import java.io.File;
import java.io.*;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigDecimal;
import java.net.HttpURLConnection;
import java.net.URL;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.List;

import org.supercsv.io.CsvListReader;
import org.supercsv.prefs.CsvPreference;


/**
 * Usage: java transfer.transfer
 */
public class transfer
{
   public static final String CSV_URL = "./measuresA3_000.csv";
   public static final String fileName = "./Redis_measuresA3_000.txt";

   public static void main(String[] args)
   {



     try (
         BufferedReader reader = new BufferedReader(new FileReader(CSV_URL));
         CsvListReader csvReader = new CsvListReader(reader, CsvPreference.STANDARD_PREFERENCE)
        )
         {
         FileWriter writer = new FileWriter(fileName, true);
         BufferedWriter bufferedWriter = new BufferedWriter(writer);
         List<String> line;
         while ((line = csvReader.read()) != null)
            {

            bufferedWriter.write(new String("hmset
                measuresa:"+line.get(0)+":"+line.get(1)+":"+line.get(2)+" m"                                                                    +line.get(0)+" s
"+line.get(1)+" bt "+line.get(2)+" et"                                                                +line.get(3)+" mv "+line.get(4) ));

                    bufferedWriter.newLine();
            bufferedWriter.write(new String("zadd averages "+ line.get(4)+" "+
             line.get(0)+":"+line.get(1)+":"+line.get(2)+":"+line.get(3)));
            bufferedWriter.newLine();
            }

        bufferedWriter.close();
         }
         catch (IOException e)
         {
            e.printStackTrace();
         }

   }
}
```

Data Convertor for Redis Format

```
import redis.clients.jedis.Jedis;
import redis.clients.jedis.*;
import java.util.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.*;
import java.io.IO Exception;
import java.io.InputStreamReader;
import java.math.BigDecimal;
import java.net.HttpURLConnection;
import java.net.URL;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.List;

public class MassInserion_Redis {

  public static void main(String[] args) {
  //Connecting to Redis server on localhost
    Jedis jedis = new Jedis("localhost",6379, 300000000);
    System.out.println("Connection to server sucessfully");
    double startTime = System.currentTimeMillis();
    String CSV_URL = "/usr/bin/measuresA6GB.csv";
    BufferedReader br = null;
    String line = "";
    String cvsSplitBy = ",";
   int lineNumber = 0;

   try {

        String key = null;
        String key1 = "mv";
        Map<String, String> map = new HashMap<>();
        Map<Double, String> scoreMembers = new HashMap<Double, String>();
         Pipeline p = jedis.pipelined();
         br = new BufferedReader(new FileReader(CSV_URL));
           while ((line = br.readLine()) != null)
           {
              String[] row = line.split(cvsSplitBy);
              ++lineNumber;

              key = "measuresa:"+row[0]+":"+row[1]+":"+row[2];
              map.put("m", row[0]);
              map.put("s", row[1]);
              map.put("bt", row[2]);
              map.put("et", row[3]);
              map.put("mv", row[4]);
              p.hmset(key, map);

              Double score = new Double(row[4]);
              p.zadd(key1,score,row[0]+":"+row[1]+":"+row[2]+":"+row[3]);


           if (lineNumber % 10000 == 0){ p.sync(); }

           }

         jedis.save();

        }
        catch (FileNotFoundException e)
            {
                e.printStackTrace();
            }
        catch (IOException e)
            {
                e.printStackTrace();
               }
        finally {
              if (br != null) {
                            try {
                                br.close();
                            } catch (IOException e) {
                                        e.printStackTrace();
                            }
              }
        }
    jedis.save();
    jedis.close();
    System.out.println("Abdullah: Number of rows has been inserted" + lineNumber);
    System.out.println("Done");

    double endTime = System.currentTimeMillis();
    System.out.println("That took " + (endTime - startTime) + " milliseconds");
    System.out.println("That took "+ ((endTime - startTime)/1000.0000) +" seconds ");
          }
}
```

Mass Insertion API For Redis

```
import redis.clients.jedis.Jedis;
import redis.clients.jedis.*;
import redis.clients.util.JedisClusterCRC16;
import java.util.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.*;
import java.io.IO Exception;
import java.io.InputStreamReader;
import java.math.BigDecimal;
import java.net.HttpURLConnection;
import java.net.URL;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.List;


public class MassInserion_Redis_Cluster {



  public static void main(String[] args) {
  //Connecting to Redis server on localhost
    Jedis jedis = new Jedis("localhost",7000, 300000000);
    Jedis jedis1 = new Jedis("localhost",7001, 300000000);
    Jedis jedis2 = new Jedis("localhost",7002, 300000000);
    Jedis jedis3 = new Jedis("localhost",7003, 300000000);
    Jedis jedis4 = new Jedis("localhost",7004, 300000000);
    Jedis jedis5 = new Jedis("localhost",7005, 300000000);
    System.out.println("Connection to servers sucessfully");
    double startTime = System.currentTimeMillis();

    String CSV_URL = "/usr/bin/measuresA4GB.csv";
    BufferedReader br = null;
            String line = "";
            String cvsSplitBy = ",";
    int lineNumber = 0;

        try {

        int slot ;
        String key = null;
        String key1 = "mv";
        Map<String, String> map = new HashMap<>();
        Map<Double, String> scoreMembers = new HashMap<Double, String>();
        Pipeline p = jedis.pipelined();
        Pipeline p1 = jedis1.pipelined();
        Pipeline p2 = jedis2.pipelined();
        Pipeline p3 = jedis3.pipelined();
        Pipeline p4 = jedis4.pipelined();
        Pipeline p5 = jedis5.pipelined();

        br = new BufferedReader(new FileReader(CSV_URL));
           while ((line = br.readLine()) != null)
           {
              String[] row = line.split(cvsSplitBy);
              ++lineNumber;

              key = "measuresa:"+row[0]+":"+row[1]+":"+row[2];
              map.put("m", row[0]);
              map.put("s", row[1]);
              map.put("bt", row[2]);
              map.put("et", row[3]);
              map.put("mv", row[4]);
              p.hmset(key, map);

              slot = (JedisClusterCRC16.getSlot(key))% 16384;
              //System.out.println("This key"+ key + " slot is " + slot );

              if( slot >= 0 &&  slot <= 2730 )
                 {p.hmset(key, map);}

              else if(slot >= 2731 &&  slot <= 5460)
                    { p1.hmset(key, map); }

              else if(slot >= 5461 &&  slot <= 8191)
                    { p2.hmset(key, map); }

              else if(slot >= 8192 &&  slot <= 10922)
                    { p3.hmset(key, map); }

              else if(slot >= 10923 &&  slot <= 13652 )
                     { p4.hmset(key, map); }

              else { p5.hmset(key, map);}
```

Mass Insertion API For Redis Cluster

```
//mv key is always in slots between 8192 && 10922 in cluster of 6 nodes
         Double score = new Double(row[4]);
         p3.zadd(key1,score,row[0]+":"+row[1]+":"+row[2]+":"+row[3]);

         if (lineNumber % 10000 == 0){ p.sync(); p1.sync(); p2.sync(); p3.sync(); p4.sync(); p5.sync();}

          }
         //p.sync();
         jedis.save();
         jedis1.save();
         jedis2.save();
         jedis3.save();
         jedis4.save();
         jedis5.save();
        }
        catch (FileNotFoundException e)
            {
              e.printStackTrace();
          }
        catch (IOException e)
            {
              e.printStackTrace();
             }
        finally {
            if (br != null) {
                        try {
                            br.close();
                        } catch (IOException e) {
                                    e.printStackTrace();
                        }
            }
        }

    jedis.save();
    jedis1.save();
    jedis2.save();
    jedis3.save();
    jedis4.save();
    jedis5.save();

    jedis.close();
    jedis1.close();
    jedis2.close();
    jedis3.close();
    jedis4.close();
    jedis5.close();

    System.out.println("Abdullah: Number of rows has been inserted" + lineNumber);
    System.out.println("Done");


       double endTime = System.currentTimeMillis();
           System.out.println("That took " + (endTime - startTime) + " milliseconds");
       System.out.println("That took "+ ((endTime - startTime)/1000.0000) +" seconds ");

        }
}
```

Mass Insertion API For Redis Cluster Continue

```
/*
* reusing the idea of bulkloading exmaple
* from http://www.datastax.com/dev/blog/using-the-cassandra-bulk-loader-updated
* the code has been completely re-coded to fit our scenario
*
*/
package bulkload;

import java.io.BufferedReader;
import java.io.File;
import java.io.*;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigDecimal;
import java.net.HttpURLConnection;
import java.net.URL;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.List;
import org.supercsv.io.CsvListReader;
import org.supercsv.prefs.CsvPreference;
import org.apache.cassandra.config.Config;
import org.apache.cassandra.dht.Murmur3Partitioner;
import org.apache.cassandra.exceptions.InvalidRequestException;
import org.apache.cassandra.io.sstable.CQLSSTableWriter;

public class BulkLoad
{
  public static final String CSV_URL = "../measuresA1GB.csv";

  /** Default output directory */
  public static final String DEFAULT_OUTPUT_DIR = "./data";

  /** Keyspace name */
  public static final String KEYSPACE = "quote";
  /** Table name */
  public static final String TABLE = "measuresA";

  /**
   * Schema for bulk loading table.
   * It is important not to forget adding keyspace name before table name,
   * otherwise CQLSSTableWriter throws exception.
   */
  public static final String SCHEMA = String.format("CREATE TABLE %s.%s (" +
                                      "m int, " +
                                      "s int, " +
                                      "bt Double, " +
                                      "et Double, " +
                                      "mv Double, " +
                                      "PRIMARY KEY ((m,s,bt),mv) " +
                                      ")", KEYSPACE, TABLE);

  /**
   * INSERT statement to bulk load.
   * It is like prepared statement. You fill in place holder for each data.
   */
  public static final String INSERT_STMT = String.format("INSERT INTO %s.%s (" +
                                           "m, s, bt, et, mv " +
                                           ") VALUES (" +
                                           "?, ?, ?, ?, ? " +
                                           ")", KEYSPACE, TABLE);

  public static void main(String[] args)
  {

    // Create output directory that has keyspace and table name in the path
    File outputDir = new File(DEFAULT_OUTPUT_DIR + File.separator + KEYSPACE + File.separator + TABLE);
    if (!outputDir.exists() && !outputDir.mkdirs())
    {
      throw new RuntimeException("Cannot create output directory: " + outputDir);
    }

    // Prepare SSTable writer
    CQLSSTableWriter.Builder builder = CQLSSTableWriter.builder();
    // set output directory
    builder.inDirectory(outputDir)
        // set target schema
        .forTable(SCHEMA)
        // set CQL statement to put data
        .using(INSERT_STMT)
        // set partitioner if needed
        // default is Murmur3Partitioner so set if you use different one.
        .withPartitioner(new Murmur3Partitioner());
    CQLSSTableWriter writer = builder.build();
```

Creates SSTables from CSV for Cassandra Bulk-loader

```
        try (
            BufferedReader reader = new BufferedReader(new FileReader(CSV_URL));
            CsvListReader csvReader = new CsvListReader(reader, CsvPreference.STANDARD_PREFERENCE)
        )
        {


            // Write to SSTable while reading data
            List<String> line;
            int lineNumber = 0;
            while ((line = csvReader.read()) != null)
            {

                ++lineNumber;
                writer.addRow(
                        new Integer(line.get(0)),
                        new Integer(line.get(1)),
                        new Double(line.get(2)),
                        new Double(line.get(3)),
                        new Double(line.get(4))
                        );
            }
        }
        catch (InvalidRequestException | IOException e)
        {
            e.printStackTrace();
        }

    try
    {
        writer.close();
    }
    catch (IOException ignore) {}
  }
}
```

Creates SSTables from CSV for Cassandra Bulk-loader Continue

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import java.sql.*;


public class GettingStarted_DB_C {


   public static void main(String[] args) throws SQLException {

    Connection conn = DriverManager.getConnection(dbURL, properties);

    //the
            //URL of database server
                String dbURL = "jdbc:oracle:thin:@abd-HP-EliteOne-800-G1-AIO:1521:orcl";
                //String dbURL = "jdbc:oracle:oci:@:orcl";
                    Properties properties = new Properties();
                    properties.put("user", "sys as sysdba");
                properties.put("password", "oracle");
                //this reduces round trips to the database by fetching multible rows of data each                    time data is fetched
                //the extra data is stored in client-side buffer for later access by the client
        //th        //prefetching feature is really faster since it let the data ready before the query
        Str       properties.put("defaultRowPrefetch", "10000");
        Pre


            //creating PreparedStatement object to execute query
        int count = 0;
        double startTime = System.currentTimeMillis();
                PreparedStatement preStatement = conn.prepareStatement(sql);
                ResultSet results = preStatement.executeQuery();
        System.out.println("i got the results \n");


   while(results.next()) {
     ++count;
   }
   double endTime = System.currentTimeMillis();
   System.out.println("That took " + (endTime - startTime) + " milliseconds");
   System.out.println("That took "+ ((endTime - startTime)/1000.0000) +" seconds ");



       if (count == 0) {
          System.out.println("No records found");
       }
       else System.out.println("number of records found : " + count);

     conn.close();

   }


}
```

DB-C API For Q2

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import java.sql.*;

public class GettingStarted_DB-O {


   public static void main(String[] args) throws SQLException {


      Conn//URL of database server
           String dbURL = "idbc:mysql://localhost:3306/quote";
       //thProperties properties = new Properties();
       //St properties.putt("user", "root");
       //St
       //Sp properties.putt("password", "root");
       //String sql="SELECT * FROM measuresA WHERE mv>15.53 AND mv<15.81";
            //String sql="SELECT * FROM measuresA WHERE mv>15.75 AND mv<16.2 ";
       //String sql="SELECT * FROM measuresA WHERE mv>0 AND mv<16.16";
       String sql="SELECT * FROM measuresA WHERE mv>0 AND mv<16.72";
       String sql1 ="reset query cache";
       String sql2 = "flush query cache";

       PreparedStatement preStatement1 = conn.prepareStatement(sql1);
       preStatement1.executeQuery();
       preStatement1.close();
       PreparedStatement preStatement2 = conn.prepareStatement(sql2);
       preStatement2.executeQuery();
       preStatement2.close();

           //creating PreparedStatement object to execute query
           int count = 0;
           double startTime = System.currentTimeMillis();
           PreparedStatement preStatement = conn.prepareStatement(sql,
                                                ResultSet.TYPE_FORWARD_ONLY,
                                                      ResultSet.CONCUR_READ_ONLY);
                       preStatement.setFetchSize(Integer.MIN_VALUE);
                       ResultSet results = preStatement.executeQuery();
           System.out.println("i got the results \n");


   while(results.next()) {
      ++count;
   }
   double endTime = System.currentTimeMillis();
   System.out.println("That took " + (endTime - startTime) + " milliseconds");
   System.out.println("That took "+ ((endTime - startTime)/1000.0000) +" seconds ");



      if (count == 0) {
         System.out.println("No records found");
      }
      else System.out.println("number of records found : " + count);


    results.close();
    preStatement.close();
    conn.close();

   }

}
```

DB-O API For Q2

```
import com.datastax.driver.core.*;



public class GettingStarted {



   public static void main(String[] args) {


      Cluster cluster;
      Session session;

   // Connect to the cluster and keyspace "bench"
   cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
   cluster.getConfiguration().getSocketOptions().setConnectTimeoutMillis(1000000);
   cluster.getConfiguration().getSocketOptions().setReadTimeoutMillis(100000000);
   session = cluster.connect("quote");
   System.out.println("Connection to server sucessfully");

   String [] Query = new String [] {"SELECT  * FROM measuresA WHERE mv>2.5 AND mv<3 ALLOW FILTERING;",
                                    "SELECT  * FROM measuresA WHERE mv>2 AND mv<7 ALLOW FILTERING;",
                                    "SELECT  * FROM measuresA WHERE mv>15.6 AND mv<15.7 ALLOW FILTERING;",
                                    "SELECT  * FROM measuresA WHERE mv>15.53 AND mv<15.81 ALLOW FILTERING;",
                                    "SELECT  * FROM measuresA WHERE mv>15.75 AND mv<16.2 ALLOW FILTERING;",
                                    "SELECT  * FROM measuresA WHERE mv>0 AND mv<16.16 ALLOW FILTERING;",
                                    "SELECT  * FROM measuresA WHERE mv>0 AND mv<16.72 ALLOW FILTERING;"};


   for ( int i = 0; i < 7 ; i++) {
   int count = 0;
   double startTime = System.currentTimeMillis();
   ResultSet results = session.execute(Query[i]);
   System.out.println("\n i got the results \n");


   for (Row row : results) {
      ++count;
   //System.out.format("%s %s\n", row.getString("s"), row.getString("mv"));
    //System.out.format("%d %f\n", row.getInt("s"), row.getDouble("mv"));
   }
   double endTime = System.currentTimeMillis();
   System.out.println("Q"+i+" took " + (endTime - startTime) + " milliseconds");
   System.out.println("Q"+i+" took "+ ((endTime - startTime)/1000.0000) +" seconds ");


      // Clean up the connection by closing it
      //cluster.close();

      if (count == 0) {
         System.out.println("No records found");
      }
      else System.out.println("number of records found : " + count);
      System.out.println("\n*****************************************\n");

   } //end of for-loop one

         cluster.close();
         session.close();

   }



}
```

CA,CA-SH API For Q2

```
import com.datastax.driver.core.*;



public class GettingStarted_weak {



   public static void main(String[] args) {


      Cluster cluster;
      Session session;

   cluster = Cluster.builder().addContactPoint("127.0.0.1").withQueryOptions(new
         QueryOptions().setConsistencyLevel(ConsistencyLevel.ONE)).build();

   cluster.getConfiguration().getSocketOptions().setConnectTimeoutMillis(1000000);
   cluster.getConfiguration().getSocketOptions().setReadTimeoutMillis(100000000);
   session = cluster.connect("quote");
   System.out.println("Connection to server sucessfully");

   String [] Query = new String [] {"SELECT  * FROM measuresA WHERE mv>2.5 AND mv<3 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>2 AND mv<7 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>15.6 AND mv<15.7 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>15.53 AND mv<15.81 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>15.75 AND mv<16.2 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>0 AND mv<16.16 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>0 AND mv<16.72 ALLOW FILTERING;"};


   for ( int i = 0; i < 7 ; i++) {
   int count = 0;
   double startTime = System.currentTimeMillis();
   ResultSet results = session.execute(Query[i]);
   System.out.println("\n i got the results \n");


   for (Row row : results) {
      ++count;
   }
   double endTime = System.currentTimeMillis();
   System.out.println("Q"+i+" took " + (endTime - startTime) + " milliseconds");
   System.out.println("Q"+i+" took "+ ((endTime - startTime)/1000.0000) +" seconds ");


      if (count == 0) {
         System.out.println("No records found");
      }
      else System.out.println("number of records found : " + count);
      System.out.println("\n*********************************************\n");

   } //end of for-loop one

    // Clean up the connection by closing it
         cluster.close();
         session.close();

   }



}
```

CA-SH-R3-W API For Q2

```
import com.datastax.driver.core.*;



public class GettingStarted_strong {



   public static void main(String[] args) {


      Cluster cluster;
      Session session;

   cluster = Cluster.builder().addContactPoint("127.0.0.1").withQueryOptions(new
         QueryOptions().setConsistencyLevel(ConsistencyLevel.ALL)).build();
   cluster.getConfiguration().getSocketOptions().setConnectTimeoutMillis(1000000);
   cluster.getConfiguration().getSocketOptions().setReadTimeoutMillis(100000000);
   session = cluster.connect("quote");
   System.out.println("Connection to server sucessfully");

   String [] Query = new String [] {"SELECT  * FROM measuresA WHERE mv>2.5 AND mv<3 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>2 AND mv<7 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>15.6 AND mv<15.7 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>15.53 AND mv<15.81 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>15.75 AND mv<16.2 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>0 AND mv<16.16 ALLOW FILTERING;",
                                                "SELECT  * FROM measuresA WHERE mv>0 AND mv<16.72 ALLOW FILTERING;"};


   for ( int i = 0; i < 7 ; i++) {
   int count = 0;
   double startTime = System.currentTimeMillis();
   ResultSet results = session.execute(Query[i]);
   System.out.println("\n i got the results \n");


   for (Row row : results) {
      ++count;
     }

   double endTime = System.currentTimeMillis();
   System.out.println("Q "+i+" took " + (endTime - startTime) + " milliseconds");
   System.out.println("Q "+i+" took "+ ((endTime - startTime)/1000.0000) +" seconds ");




      if (count == 0) {
         System.out.println("No records found");
      }
      else System.out.println("number of records found : " + count);
      System.out.println("\n*****************************************\n");

   } //end of for-loop one


    // Clean up the connection by closing it
          cluster.close();
          session.close();

   }

}
```

CA-SH-R3-S API For Q2

```
import redis.clients.jedis.Jedis;
import java.util.*;

public class GettingStarted_Redis {
  public static void main(String[] args) {
    //Connecting to Redis server on localhost
    Jedis jedis = new Jedis("localhost",6379, 300000000);
    System.out.println("Connection to server sucessfully");
    int count = 0;
    double startTime = System.currentTimeMillis();
    //change the value of the selectivity here
    Set<String> set = jedis.zrangeByScore("mv", "(0" ,"(16.72");

   for (String s : set) {
       ++count;
      }
  double endTime = System.currentTimeMillis();
  System.out.println("That took " + (endTime - startTime) + " milliseconds");
  System.out.println("That took "+ ((endTime - startTime)/1000.0000) +" seconds ");



    if (count == 0) {
       System.out.println("No records found");
    }
    else System.out.println("number of records found : " + count);
}
}
```

Redis API For Q2

```
import redis.clients.jedis.Jedis;
import java.util.*;
public class GettingStarted_Redis_cluster {
  public static void main(String[] args) {
    //Connecting to Redis server on localhost node where the mv key store
    Jedis jedis = new Jedis("localhost",7003, 300000000);
    System.out.println("Connection to server sucessfully");
    int count = 0;
    double startTime = System.currentTimeMillis();
    //change the value of the selectivity here
    Set<String> set = jedis.zrangeByScore("mv", "(0" ,"(16.72");

   for (String s : set) {
       ++count;
      }
  double endTime = System.currentTimeMillis();
  System.out.println("That took " + (endTime - startTime) + " milliseconds");
  System.out.println("That took "+ ((endTime - startTime)/1000.0000) +" seconds ");



    if (count == 0) {
       System.out.println("No records found");
    }
    else System.out.println("number of records found : " + count);
}
}
```

Redis-SH API For Q2

Appendix B

Performance of Cassandra SSTable Loader V.S Copy Command

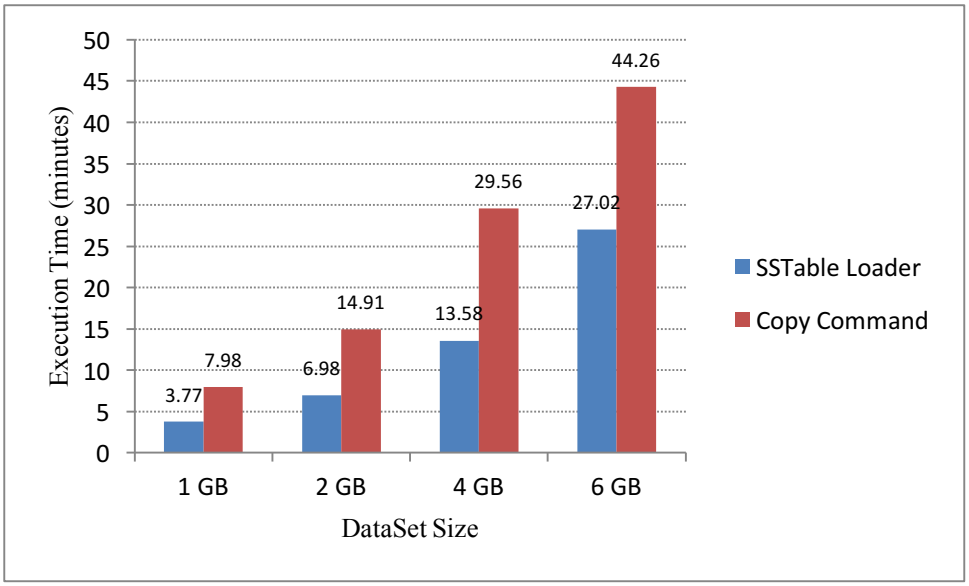

Fig1: SSTable loader VS Copy Command for Cassandra in single node CA

DataSet Size	SSTable Loader	Copy Command
1 GB	3.77	7.98
2 GB	6.98	14.91
4 GB	13.58	29.56
6 GB	27.02	44.26

Table 1: SSTable loader VS Copy Command for Cassandra in single node CA

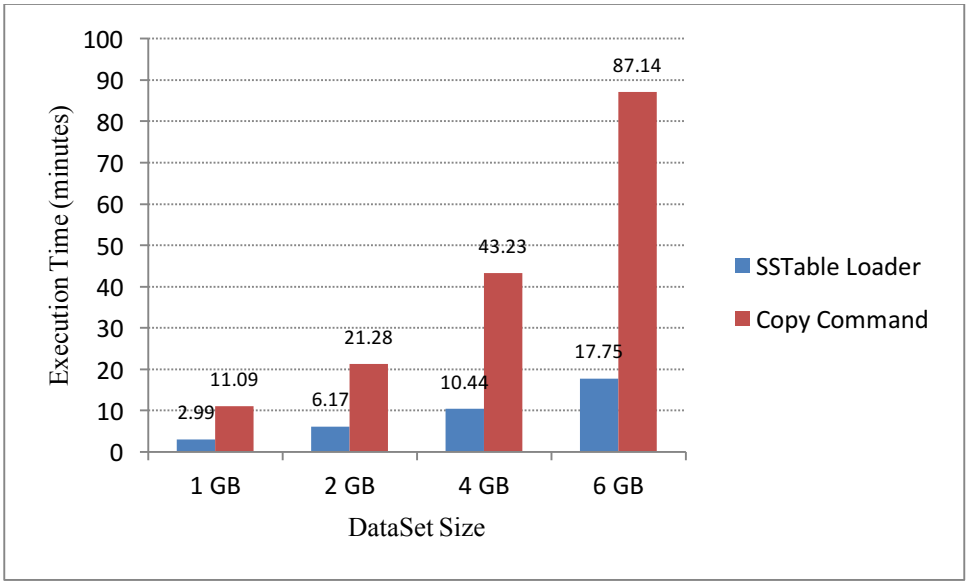

Fig2: SSTable loader VS Copy Command for Cassandra in partition mode CA-SH

DataSet Size	SSTable Loader	Copy Command
1 GB	2.99	11.09
2 GB	6.17	21.28
4 GB	10.44	43.23
6 GB	17.75	87.14

Table 2: SSTable loader VS Copy Command for Cassandra in partition mode CA-SH

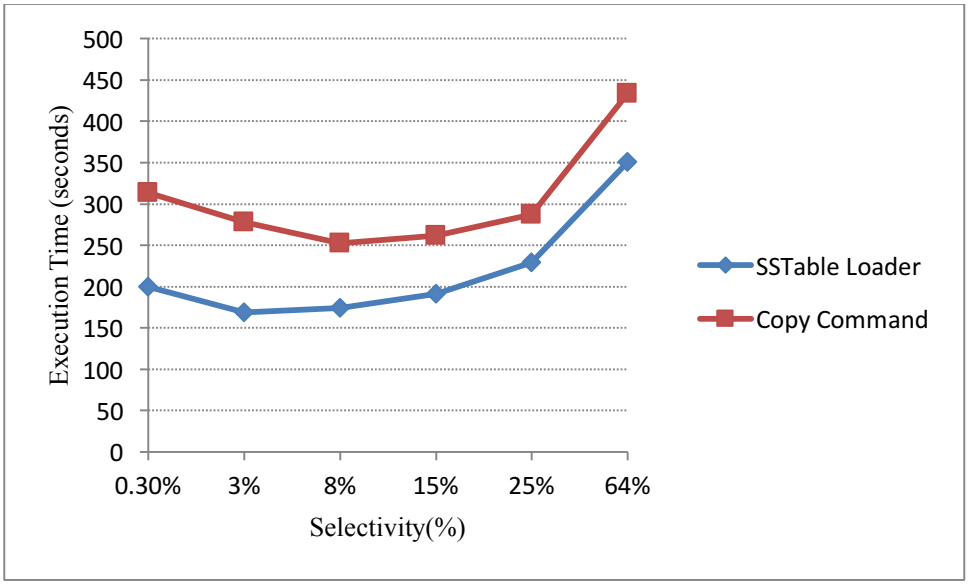

Fig3: Executing Q2 of CA-SH in SSTable Loader Data VS Copy Command for 4 GB data

Selectivity	0.3%	3%	8%	15%	25%	64%
SSTable Loader	199.6	168.6	174.1	191.0	229.4	350.7
Copy Command	313.9	277.8	252.7	261.6	287.0	433.9

Table 3: Executing Q2 of CA-SH in SSTable Loader Data VS Copy Command for 4 GB data

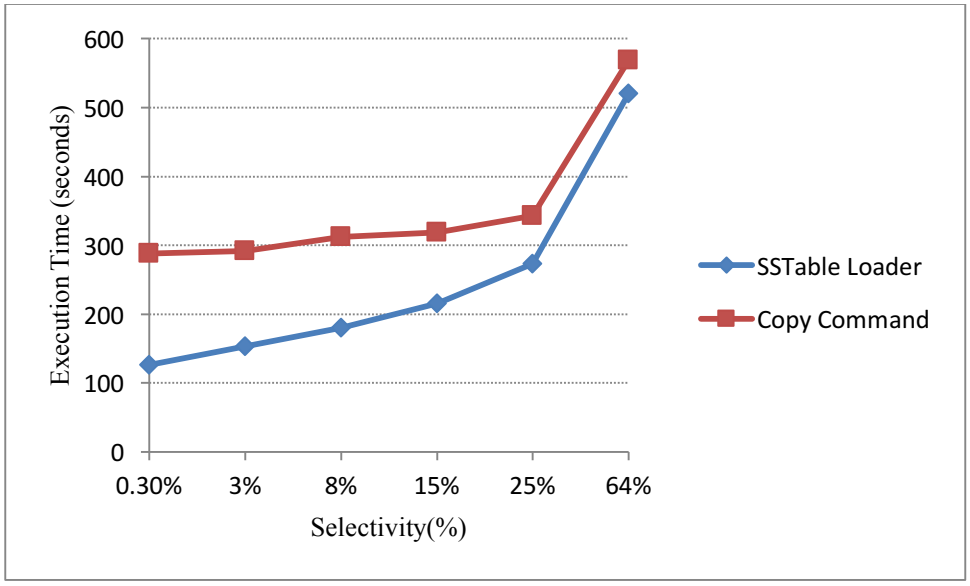

Fig4: Executing Q3 of CA-SH in SSTable Loader Data VS Copy Command for 4 GB data

Selectivity	0.3%	3%	8%	15%	25%	64%
SSTable Loader	126.7	153.4	180.0	215.2	273.5	520.7
Copy Command	288.4	292.2	311.9	319.1	343.3	568.4

Table 4: Executing Q3 of CA-SH in SSTable Loader Data VS Copy Command for 4 GB data

Appendix C

Redis bulk loading Issues

As we can see from bulk loading experiments section, that both Redis & Redis-SH has not bulk loaded for 6GB of datasets, and for Redis-SH only the data structure part which is related for Q2&Q3 has been uploaded only. That because the machine which is running the experimental servers has been limited memory to 16GB of RAM, and as we can see from following data, that Redis need more RAM to accommodate the data. Therefore, tries had been conducted to bulk loaded each data structure separately, and comparison has been conducted to see if there and performance different between uploaded the both data structure (Hash & Sorted Sets) or only one of them, for only 1GB &2GB of datasets. The results has been found that no different at all in the queries execution time when both data structure uploaded, or one of them.

DataSet size	Both Data (RAM Size)	Both data structure (DISK Size)	Data for Q1 (RAM)	Data for Q1 (DISK Size)	Data for Q2, Q3 (RAM Size)	Data for Q2, Q3 (DISK Size)
1 GB	6.95 GB	3.2 GB	3.66 GB	2,2 GB	3.30 GB	1 GB
2 GB	13.52 GB	6.3 GB	7.12 GB	4.3 GB	6.40 GB	2 GB
4 GB	Not Fit	Not Fit	14.10 GB	8.6 GB	12.65 GB	4 GB
6 GB	Not Fit	Not Fit	Not Fit	Not Fit	Not Fit	Not Fit

Table 1: RAM and Disk size which been allocated for Rides single node

Size of memries used	Both Data (RAM)	both data (DISK)	Data for Q1 (RAM)	Data for Q1 (DISK)	Data for Q2, Q3 (RAM)	Data for Q2, Q3 (DISK)
1 GB	9.114 GB	3.253 GB	Not Calculated	Not Calculated	Not Calculated	Not Calculated
2 GB	Not Fit	Not Fit	11.12 GB	4.328 GB	6.405 GB	2GB
4 GB	Not Fit	Not Fit	Not Fit	Not Fit	12.655 GB	4GB
6 GB	Not Fit	Not Fit	Not Fit	Not Fit	Not Fit	Not Fit

Table 2: RAM and Disk sizes which has been allocated for cluster of nodes

Appendix D

What has been added for Cassandra and Redis command lines.

Unfortunately Cassandra Query Language shell (CQLsh) has not supported printing execution time for the queries, therefore, CQLsh code which is written in python has been analyzed, and printing executing time feature has been added as following:

```
# ABD added this to start calculating time in cqlsh code
timestart = time.time()
self.perform_statement(statement) #this already there
# ABD added this to get the time has been taken by the query in cqlsh code
timeend = time.time()
# ABD added this to printing time in cqlsh code
print "ABD its took %s." % (describe_interval(timeend - timestart))
```

Redis has Redis-cli command shell to performs the commands, this shell unfortunately printing the execution time only for slow queries which took long time to get the reply, moreover, the piping (mass insertion) also not printing the time of bulk loading, while we have most of our query were super fast, and our bulk loading time is valuable, therefore, as what has been done in Cassandra, Redis-cli shell code in C has been analyzed, and feature of printing the execution time has been added.

```
// in Bulk import (pipe) mode function in redis-cli
 long long start = ustime(); //ABD added this to calculate the time in top of the function

redisReaderFree(reader); //already there
   printf("errors: %lld, replies: %lld\n", errors, replies); //already there
   //ABD added this to calculate and print the time after the databse reply
  printf("\n--------ABD: The Request accomplished in: %.5f seconds--------\n",(float)(ustime()-
start)/1000000);
   if (errors) //already there
      exit(1); //already there
   else        //already there
      exit(0);  //already there
```

```
/*to print execution time do the following
got to redis.conf file and change slowlog-log-slower-than value to zero
his may not working
for that reason we will add our own time calculator in redis-cli.c
when ever i added the needed, i comment it with 'ABD'
in particular, i add this feature in issueCommandRepeat() and pipeMode() functions
after adding the timing features, run ‘make all’ under src file */

static int issueCommandRepeat(int argc, char **argv, long repeat) {
  long long start = ustime(); //ABD added this to calculate the time
   while (1) {
      config.cluster_reissue_command = 0;
      if (cliSendCommand(argc,argv,repeat) != REDIS_OK) {
         cliConnect(1);

         /* If we still cannot send the command print error.
          * We'll try to reconnect the next time. */
         if (cliSendCommand(argc,argv,repeat) != REDIS_OK) {
            cliPrintContextError();
            return REDIS_ERR;
         }
       }
       /* Issue the command again if we got redirected in cluster mode */
       if (config.cluster_mode && config.cluster_reissue_command) {
         cliConnect(1);
       } else {
          break;
      }
   }
   //ABD added this to calculate and print the time
   printf("\n--------ABD: The Request accomplished in: %.5f seconds--------\n",(float)(ustime()-start)/1000000);
   return REDIS_OK;  // already there
}
```

//this code already exist

Appendix E

Databases size after bulk loading

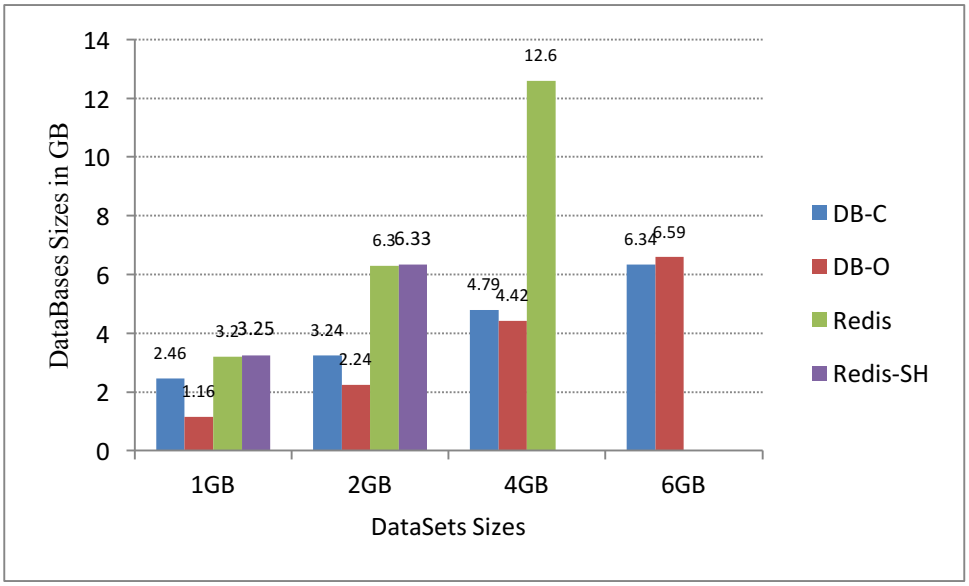

Fig 1: Size of Data Bases with no index in GB

DB \ DataSets	1GB	2GB	4GB	6GB
DB-C	2.46	3.24	4.79	6.34
DB-O	1.16	2.24	4.42	6.59
CA	Not apply	Not apply	Not apply	Not apply
CA-SH	Not apply	Not apply	Not apply	Not apply
CA-SH-R3-W	Not apply	Not apply	Not apply	Not apply
CA-SH-R3-S	Not apply	Not apply	Not apply	Not apply
Redis	3.2*	6.3*	12.6*	Not apply
Redis-SH	3.25*	6.33*	Not apply**	Not apply

Table 1: Size of Data Bases with no index in GB

*For Redis, & Redis-SH, only size of RDB file has been taken, since AOF file has been disabled.
** For Redis-SH at 4GB, only database size for data structure of Q2 & Q3 has been taken, since the limitation of RAM couldn't help to upload all data or only data for Q1.

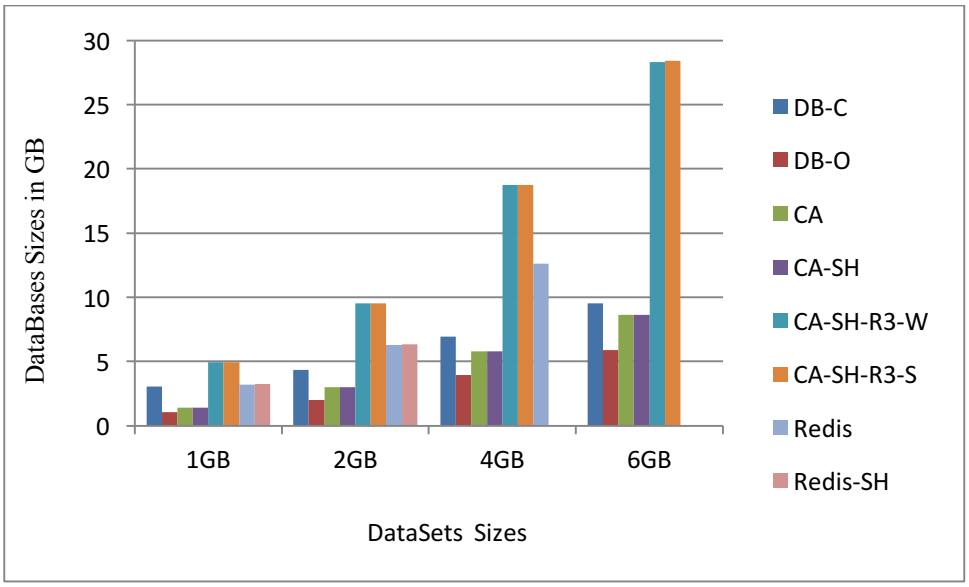

Fig 2: Size of Databases with sensor key index in GB

DB \ DataSets	1GB	2GB	4GB	6GB
DB-C	3.03	4.34	6.92	9.54
DB-O	1.04	2.01	3.96	5.90
CA	1.42	3.00	5.80	8.64
CA-SH	1.42	3.00	5.80	8.64
CA-SH-R3-W	4.92	9.51	18.72	28.30
CA-SH-R3-S	4.92	9.52	18.72	28.40
Redis	3.2*	6.3*	12.6*	Not apply
Redis-SH	3.25*	6.33*	Not apply**	Not apply

Table 2: Size of Data Bases with sensor key index in GB

*For Redis, & Redis-SH, only size of RDB file has been taken, since AOF file has been disabled.
** For Redis-SH at 4GB, only database size for data structure of Q2 & Q3 has been taken, since the limitation of RAM couldn't help to upload all data or only data for Q1.

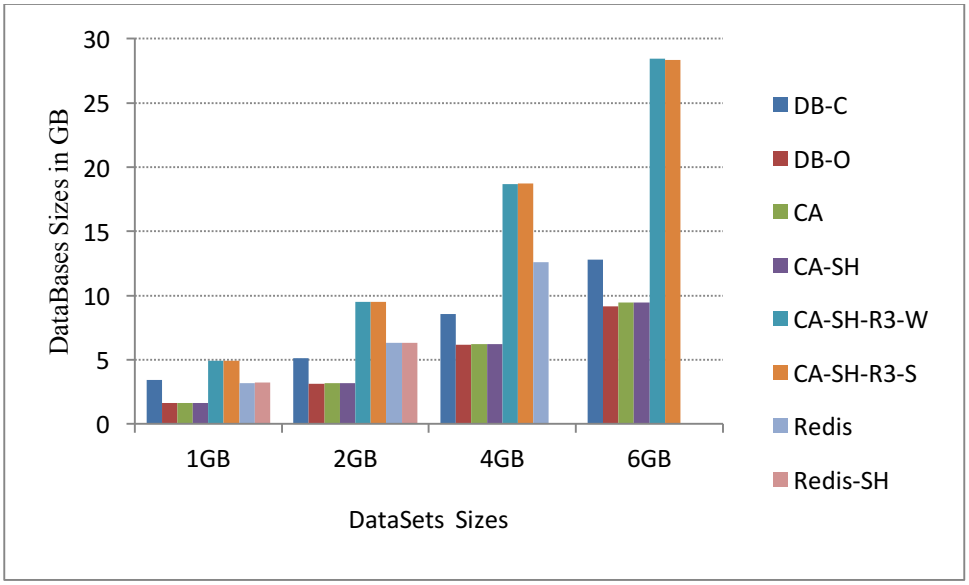

Fig 3: Size of Data Bases with sensor and measured value key indexes in GB

DB \ Datasets	1GB	2GB	4GB	6GB
DB-C	3.44	5.15	8.57	12.78
DB-O	1.62	3.12	6.16	9.18
CA	1.64	3.17	6.23	9.46
CA-SH	1.64	3.16	6.24	9.44
CA-SH-R3-W	4.91	9.50	18.68	28.44
CA-SH-R3-S	4.92	9.52	18.72	28.36
Redis	3.2*	6.3*	12.6*	Not apply
Redis-SH	3.25*	6.33*	Not apply**	Not apply

Table 3: Size of Data Bases with sensor and measured value key indexes in GB

*For Redis, & Redis-SH, only size of RDB file has been taken, since AOF file has been disabled.
** For Redis-SH at 4GB, only database size for data structure of Q2 & Q3 has been taken, since the limitation of RAM couldn't help to upload all data or only data for Q1.

Appendix F

The Data results of bulk loading

DB \ Data Set	1 GB	2 GB	4 GB	6 GB
DB-C	1.6070	3.0540	6.1710	9.1170
DB-O	2.0480	6.9350	16.3160	27.4210
Redis	3.7250	7.4400	14.0190	Not apply
Redis-SH	3.4330	7.4400	Not apply	Not apply

Table 1: Performance of bulk loading without indexing (in minutes)

DB \ Data Set	1 GB	2 GB	4 GB	6 GB
DB-C	5.4520	10.5490	21.4140	35.0155
DB-O	5.9280	11.9470	26.7740	45.0790
CA	7.9819	14.9114	29.5633	44.2590
CA-SH	2.9860	6.1710	10.4400	17.7460
CA-SH-R3-W	3.9530	7.9820	14.1240	22.7710
CA-SH-R3-S	3.9510	6.8980	13.8200	20.0500
Redis	3.7250	7.4400	14.0190	Not apply
Redis-SH	3.4330	7.4400	Not apply	Not apply

Table 2: Performance of bulk loading with sensor key index (in minutes)

DB \ Data Set	1 GB	2 GB	4 GB	6 GB
DB-C	8.3540	16.4305	33.6380	55.2780
DB-O	8.2440	16.2040	35.2370	64.4010
CA	12.9110	25.7750	50.9570	79.4430
CA-SH	7.4630	15.5600	33.1300	62.0000
CA-SH-R3-W	18.3030	39.3500	83.6930	200.9190
CA-SH-R3-S	18.6040	39.2770	85.8780	195.8210
Redis	3.7250	7.4400	14.0190	Not apply
Redis-SH	3.4330	7.4400	Not apply	Not apply

Table 3: Performance of bulk loading with sensor and mv indexes (in minutes)

Appendix G

The Data results of Q1

DB \ Data Set	1 GB	2 GB	4 GB	6 GB
DB-C	0.3900	0.6200	1.1500	1.6400
DB-O	5.6800	10.9300	21.9800	32.8400
Redis	0.0002	0.0001	0.0001	Not apply
Redis-SH	0.0001	0.0001	Not apply	Not apply

Table 4: Performance of Q1 without indexing (in seconds)

DB \ Data Set	1 GB	2 GB	4 GB	6 GB
DB-C	0.0100	0.0100	0.0100	0.0010
DB-O	0.0010	0.0010	0.0010	0.0010
CA	0.0040	0.0040	0.0040	0.0320
CA-SH	0.0040	0.0040	0.0040	0.0320
CA-SH-R3-W	0.0040	0.0040	0.0040	0.0640
CA-SH-R3-S	0.0040	0.0040	0.0040	0.0650
Redis	0.0002	0.0001	0.0001	Not apply
Redis-SH	0.0001	0.0001	Not apply	Not apply

Table 5: Performance of Q1 with sensor key index (in seconds)

DB \ Data Set	1 GB	2 GB	4 GB	6 GB
DB-C	0.0100	0.0100	0.0100	0.0010
DB-O	0.0010	0.0100	0.0500	0.2900
CA	0.0040	0.0040	0.0040	0.0040
CA-SH	0.0080	0.0160	0.0320	0.0650
CA-SH-R3-W	0.0110	0.0110	0.0650	0.0650
CA-SH-R3-S	0.0110	0.0110	0.1150	0.1150
Redis	0.0002	0.0001	0.0001	Not apply
Redis-SH	0.0001	0.0001	Not apply	Not apply

Table 6: Performance of Q1 with sensor and mv indexes (in seconds)

Appendix H

The Data results of Q2

DB \ Selectivity	0.12%	1%	12%	29%	36%	81%	81%
DB-C	0.6340	0.6880	1.2180	2.0660	2.3130	4.4690	4.4950
DB-O	5.6900	5.9750	9.2670	14.3260	16.3310	29.9040	29.8650
Redis	0.0910	0.2140	2.4790	7.0450	9.9070	25.9500	25.7520
Redis-SH	0.0910	0.2100	2.4640	6.9650	9.5570	24.6250	25.5320

Table 7: Performance of Q2 without indexing for 1GB (in seconds)

DB \ Selectivity	0.06%	0.6%	6%	15%	29%	49%	90%
DB-C	1.1790	1.2520	1.7730	2.5140	3.7480	5.6680	9.3810
DB-O	11.0700	11.3040	14.6280	19.6790	27.7780	39.4090	63.6290
Redis	0.0910	0.2060	2.4510	6.9480	18.7720	26.1310	74.0550
Redis-SH	0.0900	0.2050	2.4350	6.8420	18.0050	25.5290	55.0390

Table 8: Performance of Q2 without indexing for 2GB (in seconds)

DB \ Selectivity	0.03%	0.3%	3%	8%	15%	25%	64%
DB-C	2.6130	2.3460	2.7680	3.5160	4.8150	6.6060	13.5640
DB-O	22.1330	22.2340	25.5480	30.6030	38.7230	50.2530	95.8240
Redis	0.1720	0.2090	2.4480	7.0520	17.9880	25.9140	167.1580
Redis-SH	0.0900	0.2030	2.4330	7.0970	18.1170	25.8540	177.0000

Table 9: Performance of Q2 without indexing for 4GB (in seconds)

DB \ Selectivity	0.02%	0.2%	2%	5%	10%	17%	43%
DB-C	3.2070	3.2170	3.6960	4.5260	5.8550	7.5890	14.4160
DB-O	32.8400	33.1100	36.4370	41.4840	49.6080	61.1920	106.7480

Table 8: Performance of Q2 without indexing for 6GB (in seconds)

DB \ Selectivity	0.12%	1%	12%	29%	36%	81%	81%
DB-C	0.6660	0.7170	1.2320	2.0200	2.3080	4.4600	4.4840
DB-O	4.1650	4.4390	7.8700	12.9610	14.9430	28.4310	28.4380
CA	27.8790	28.7100	31.6340	41.8030	46.2230	69.0430	69.2020
CA-SH	85.8840	36.6760	42.5620	57.0620	62.7550	95.5160	95.3710
CA-SH-R3-W	35.0020	34.1310	43.3640	56.9700	63.0380	95.1740	95.8260
CA-SH-R3-S	can't execute	137.0490	125.1490	143.9760	157.0770	224.0990	229.3250
Redis	0.0910	0.2140	2.4790	7.0450	9.9070	25.9500	25.7520
Redis-SH	0.0910	0.2100	2.4640	6.9650	9.5570	24.6250	25.5320

Table 9: Performance of Q2 with sensor key index for 1GB (in seconds)

DB \ Selectivity	0.06%	0.6%	6%	15%	29%	49%	90%
DB-C	1.1540	1.2110	1.7200	2.5330	3.7410	5.6270	9.3070
DB-O	8.0720	8.3530	11.8310	16.9330	25.2120	36.7580	60.8480
CA	82.5910	58.0830	57.2060	67.8590	85.2710	106.0150	147.6080
CA-SH	can't execute	88.3630	79.4550	94.9080	119.8450	149.7350	211.8140
CA-SH-R3-W	103.6940	118.0200	114.1610	122.4840	147.9340	181.9040	252.4630
CA-SH-R3-S	can't execute	360.9440	319.6660	339.8220	368.1460	402.1230	433.7090
Redis	0.0910	0.2060	2.4510	6.9480	18.7720	26.1310	74.0550
Redis-SH	0.0900	0.2050	2.4350	6.8420	18.0050	25.5290	55.0390

Table 10: Performance of Q2 with sensor key index for 2GB (in seconds)

DB \ Selectivity	0.03%	0.3%	3%	8%	15%	25%	64%
DB-C	2.1440	2.2100	2.7300	3.5020	4.8310	6.5840	13.6060
DB-O	15.8900	16.1860	19.6770	24.8060	33.1850	44.6950	90.7470
CA	357.3070	106.8410	98.4370	108.7320	126.3000	148.1910	230.2620
CA-SH	can't execute	199.0600	168.6360	174.0530	191.0060	229.3460	350.6940
CA-SH-R3-W	can't execute	186.8540	177.0380	193.7480	218.1830	265.8490	416.7350
CA-SH-R3-S	can't execute	828.0920	681.6840	662.3640	674.4030	673.8080	672.1730
Redis	0.1720	0.2090	2.4480	7.0520	17.9880	25.9140	167.1580
Redis-SH	0.0900	0.2030	2.4330	7.0970	18.1170	25.8540	177.0000

Table 11: Performance of Q2 with sensor key index for 4GB (in seconds)

DB \ Selectivity	0.02%	0.2%	2%	5%	10%	17%	43%
DB-C	3.3560	3.2880	3.7460	4.5270	5.8290	7.5410	14.4270
DB-O	23.9110	24.1860	27.6820	32.8170	41.1300	52.6610	99.1380
CA	225.3310	150.0620	139.7870	147.1570	165.4100	188.2950	282.9770
CA-SH	can't execute	290.7060	217.8340	205.4770	224.8960	257.0640	378.6450
CA-SH-R3-W	can't execute	205.0390	189.7490	202.9540	234.8180	265.3950	392.6210
CA-SH-R3-S	can't execute	can't execute	969.3620	938.2440	961.0210	1007.9220	1061.4500

Table 12: Performance of Q2 with sensor key index for 6GB (in seconds)

DB \ Selectivity	0.12%	1%	12%	29%	36%	81%	81%
DB-C	0.1720	0.2620	1.1500	2.4610	2.9700	6.6750	8.8700
DB-O	0.0970	0.8280	7.8690	12.9730	14.9490	28.4260	28.4320
CA	28.2890	25.8160	32.5460	43.7200	48.1340	72.5970	72.5830
CA-SH	87.8190	69.7860	43.8940	58.1270	66.2370	96.8350	96.7740
CA-SH-R3-W	42.5660	38.6120	44.6190	58.6820	64.4000	99.8060	97.7530
CA-SH-R3-S	can't execute	153.6190	133.1840	150.7130	157.5570	212.9130	217.4490
Redis	0.0910	0.2140	2.4790	7.0450	9.9070	25.9500	25.7520
Redis-SH	0.0910	0.2100	2.4640	6.9650	9.5570	24.6250	25.5320

Table 13: Performance of Q2 with sensor key & mv indexes for 1GB (in seconds)

DB \ Selectivity	0.06%	0.6%	6%	15%	29%	49%	90%
DB-C	0.1910	0.3060	1.6490	5.3390	7.0770	11.0760	12.9000
DB-O	0.0990	0.8400	6.0230	16.9370	25.2300	36.7500	60.9520
CA	53.8920	56.5150	59.7220	74.5480	90.9200	109.9470	153.7920
CA-SH	can't execute	82.8990	79.7480	94.6970	119.7740	150.1120	210.4940
CA-SH-R3-W	97.3580	90.8170	85.9000	100.2500	121.5960	149.3130	207.8580
CA-SH-R3-S	can't execute	396.6600	335.7060	350.9900	362.8870	385.8100	438.8090
Redis	0.0910	0.2060	2.4510	6.9480	18.7720	26.1310	74.0550
Redis-SH	0.0900	0.2050	2.4350	6.8420	18.0050	25.5290	55.0390

Table 14: Performance of Q2 with sensor key & mv indexes for 2GB (in seconds)

DB \ Selectivity	0.03%	0.3%	3%	8%	15%	25%	64%
DB-C	0.2460	0.3010	1.6460	2.7700	5.2250	8.1440	14.9300
DB-O	0.1000	0.8810	8.3540	14.7680	33.1570	44.6750	90.7470
CA	166.8890	163.1100	148.0080	145.8070	154.9140	187.2970	283.2370
CA-SH	can't execute	200.8940	155.6770	160.5630	192.2180	238.4090	370.8450
CA-SH-R3-W	can't execute	243.6860	213.9820	205.5920	213.1550	227.6710	334.3380
CA-SH-R3-S	can't execute	can't execute	712.6060	697.2240	719.5190	722.5350	714.3500
Redis	0.1720	0.2090	2.4480	7.0520	17.9880	25.9140	167.1580
Redis-SH	0.0900	0.2030	2.4330	7.0970	18.1170	25.8540	177.0000

Table 15: Performance of Q2 with sensor key & mv indexes for 4GB (in seconds)

DB \ Selectivity	0.02%	0.2%	2%	5%	10%	17%	43%
DB-C	0.3310	0.3060	1.6200	3.9270	4.2740	8.2270	18.0110
DB-O	0.1030	0.8910	8.2380	15.6200	41.0750	52.6540	99.0620
CA	165.5860	157.7780	143.2410	154.6730	172.4320	198.0770	296.1510
CA-SH	can't execute	249.8260	200.7870	207.7510	207.7510	259.8020	382.5970
CA-SH-R3-W	can't execute	248.7380	208.0960	211.5480	231.8440	261.9230	381.8550
CA-SH-R3-S	can't execute	can't execute	982.6970	938.8820	949.4180	975.4540	1040.0790

Table 16: Performance of Q2 with sensor key & mv indexes for 6GB (in seconds)

Appendix I

The Data results of Q3

DB \ Selectivity	0.12%	1%	12%	29%	36%	81%	81%
DB-C	0.4600	0.4700	0.4800	0.4700	0.4700	0.6200	0.5600
DB-O	4.4700	4.4700	4.4900	4.5600	4.5100	4.7900	4.7800
Redis	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
Redis-SH	0.0002	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

Table 17: Performance of Q3 without indexing for 1GB (in seconds)

DB \ Selectivity	0.06%	0.6%	6%	15%	29%	49%	90%
DB-C	1.0300	0.9600	1.0000	0.9600	1.0200	1.0600	1.1100
DB-O	8.6900	8.7100	8.7100	8.7400	8.7800	9.0600	9.3200
Redis	0.0001	0.0001	0.0001	0.0001	0.0001	0.0002	0.0001
Redis-SH	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

Table 18: Performance of Q3 without indexing for 2GB (in seconds)

DB \ Selectivity	0.03%	0.3%	3%	8%	15%	25%	64%
DB-C	1.9400	1.9300	1.9500	1.9800	1.9700	1.9900	2.0900
DB-O	17.2700	17.2400	17.2400	17.2900	17.3400	17.5700	18.0500
Redis	0.0002	0.0002	0.0001	0.0002	0.0002	0.0002	0.0002
Redis-SH	0.0001	0.0001	0.0002	0.0001	0.0001	0.0001	0.0001

Table 19: Performance of Q3 without indexing for 4GB (in seconds)

DB \ Selectivity	0.02%	0.2%	2%	5%	10%	17%	43%
DB-C	2.9000	2.9100	3.0000	2.8800	3.0200	2.9000	2.9700
DB-O	25.9000	25.8600	25.9100	25.9200	25.9500	26.2700	26.6800

Table 20: Performance of Q3 without indexing for 6GB (in seconds)

DB \ Selectivity	0.12%	1%	12%	29%	36%	81%	81%
DB-C	0.5600	0.4400	0.4700	0.4700	0.4400	0.5100	0.5200
DB-O	3.0600	2.9600	3.0200	3.1000	3.0500	3.2800	3.2800
CA	24.3550	25.9570	34.9190	52.2440	59.6130	99.1660	99.2160
CA-SH	32.0140	33.8120	47.1750	69.1530	78.6640	130.2750	130.1730
CA-SH-R3-W	25.6550	26.9050	34.8150	53.1890	60.3450	96.3460	96.1450
CA-SH-R3-S	131.1420	80.9210	104.6620	132.6980	145.6740	240.5700	240.4620
Redis	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
Redis-SH	0.0002	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

Table 21: Performance of Q3 with sensor key index for 1GB (in seconds)

DB \ Selectivity	0.06%	0.6%	6%	15%	29%	49%	90%
DB-C	1.0400	0.9300	0.9500	0.9500	0.9500	1.0100	1.0200
DB-O	5.9100	5.7500	5.8800	5.9500	6.0800	6.3700	6.4600
CA	50.9990	51.7490	70.4300	95.9680	140.4380	198.2370	331.7330
CA-SH	61.8660	62.5140	76.9130	101.8450	141.5450	195.8600	307.7470
CA-SH-R3-W	87.5760	55.7500	62.4550	78.5280	111.7710	154.2790	237.0380
CA-SH-R3-S	310.3230	205.4150	245.6400	300.0000	301.1300	359.1860	555.3840
Redis	0.0001	0.0001	0.0001	0.0001	0.0001	0.0002	0.0001
Redis-SH	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

Table 22: Performance of Q3 with sensor key index for 2GB (in seconds)

DB \ Selectivity	0.03%	0.3%	3%	8%	15%	25%	64%
DB-C	1.9600	1.8900	1.9200	1.9200	1.9600	1.9600	2.0300
DB-O	11.5900	11.3100	11.4500	11.5400	11.6700	12.0200	12.7300
CA	102.6830	92.7660	105.5310	121.6070	149.6530	190.3640	387.3080
CA-SH	128.9800	126.7110	153.3680	180.0000	215.2130	273.5230	520.6690
CA-SH-R3-W	157.0940	147.8620	162.9160	177.7570	193.4970	220.9580	396.9170
CA-SH-R3-S	670.5090	586.0750	592.9040	602.5280	609.1710	608.8110	911.3570
Redis	0.0002	0.0002	0.0001	0.0002	0.0002	0.0002	0.0002
Redis-SH	0.0001	0.0001	0.0002	0.0001	0.0001	0.0001	0.0001

Table 23: Performance of Q3 with sensor key index for 4GB (in seconds)

DB \ Selectivity	0.02%	0.2%	2%	5%	10%	17%	43%
DB-C	2.9700	2.8600	2.8700	2.8900	2.9500	2.9100	2.9800
DB-O	17.2900	17.2000	17.2800	17.3300	17.5700	17.9200	18.8300
CA	268.1550	159.9810	148.7010	163.0220	186.6630	220.2090	399.1210
CA-SH	240.1000	195.5790	215.5040	235.1460	266.1810	307.0630	506.7790
CA-SH-R3-W	164.4260	145.4950	161.0000	173.2430	189.3150	224.5510	374.8690
CA-SH-R3-S	976.0140	867.1130	745.8430	799.1670	802.7120	797.0340	949.8190

Table 24: Performance of Q3 with sensor key index for 6GB (in seconds)

DB \ Selectivity	0.12%	1%	12%	29%	36%	81%	81%
DB-C	0.0010	0.0200	0.1300	0.8400	0.6400	0.7800	0.7700
DB-O	0.0200	0.0600	0.4300	1.0100	1.2500	2.8500	2.8600
CA	24.6540	25.1050	32.9170	49.8910	56.5050	88.3500	88.3500
CA-SH	33.6680	34.3130	48.4760	69.5520	78.7140	131.0250	131.1240
CA-SH-R3-W	31.1160	34.1660	40.0720	55.0200	63.8750	97.8990	97.7960
CA-SH-R3-S	119.0540	90.7370	114.2220	135.7780	149.1790	241.8750	245.7320
Redis	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
Redis-SH	0.0002	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

Table 25: Performance of Q3 with sensor key & mv indexes for 1GB (in seconds)

DB \ Selectivity	0.06%	0.6%	6%	15%	29%	49%	90%
DB-C	0.0300	0.0400	0.1300	0.2600	1.6800	1.4800	1.5500
DB-O	0.0200	0.0600	0.4300	1.0000	1.9700	3.3100	6.1300
CA	46.8410	47.2910	57.9050	70.8770	105.2770	142.9890	216.4960
CA-SH	68.9240	72.4190	90.0310	108.5540	148.7030	205.7880	315.5590
CA-SH-R3-W	66.3270	66.8670	77.4300	93.5390	118.1280	163.2170	241.1730
CA-SH-R3-S	332.8790	296.8560	330.6830	343.7720	349.6180	381.0220	564.6540
Redis	0.0001	0.0001	0.0001	0.0001	0.0001	0.0002	0.0001
Redis-SH	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001

Table 26: Performance of Q3 with sensor key & mv indexes for 2GB (in seconds)

DB \ Selectivity	0.03%	0.3%	3%	8%	15%	25%	64%
DB-C	0.0010	0.0100	0.0900	0.2200	0.4300	0.7200	3.0700
DB-O	0.0100	0.0400	0.4200	1.0000	1.9800	3.3300	8.6300
CA	122.4730	109.9960	133.4270	160.4320	172.2390	204.5790	390.1830
CA-SH	179.6280	160.2690	171.2660	190.8980	227.4180	282.8740	525.4670
CA-SH-R3-W	196.4210	179.2320	186.2600	197.2790	209.0670	232.4090	403.1260
CA-SH-R3-S	743.8200	633.9360	695.5170	714.7110	728.5270	736.0670	956.3630
Redis	0.0002	0.0002	0.0001	0.0002	0.0002	0.0002	0.0002
Redis-SH	0.0001	0.0001	0.0002	0.0001	0.0001	0.0001	0.0001

Table 27: Performance of Q3 with sensor key & mv indexes for 4GB (in seconds)

DB \ Selectivity	0.02%	0.2%	2%	5%	10%	17%	43%
DB-C	0.0100	0.0300	0.1000	0.2100	0.4200	0.6900	4.3900
DB-O	0.0100	0.0700	0.4300	1.0000	1.9600	3.3200	8.5700
CA	139.4430	136.1380	148.5490	160.1640	176.1890	207.3730	340.1740
CA-SH	205.4530	189.9000	191.2180	215.6880	256.1200	297.3320	485.9990
CA-SH-R3-W	197.6940	179.7770	199.7600	213.7160	232.9090	261.7440	400.0810
CA-SH-R3-S	996.7570	863.4350	769.5130	887.8400	897.8280	906.6330	996.6770

Table 28: Performance of Q3 with sensor key & mv indexes for 6GB (in seconds)