A REPORT

ON

REPRESENTING UNSTRUCTURED DATA USING NOSQL

BY

CHIRAG PABBARAJU

2014A7PS393G

B.E (Hons) Computer Science Engineering

Prepared in partial fulfillment of the

Practice School –I Course Nos.

BITS C221/BITS C231/BITS C241

At

INDIRA GANDHI CENTRE FOR ATOMIC RESEARCH

KALPAKKAM

A Practice School-I Station of

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

K.K. BIRLA GOA CAMPUS

(JULY 2016)

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN), K.K. BIRLA GOA CAMPUS

Practice School Division

Station: INDIRA GANDHI CENTER FOR ATOMIC RESEARCH

Centre: KALPAKKAM

Duration: 23 MAY 2016 - 16 JULY 2016 **Date of Start:** 23 MAY 2016

Date of Submission: 12 JULY 2016

Title of the Project: REPRESENTING UNSTRUCTURED DATA USING NOSQL

ID No: 2014A7PS393G

Name: CHIRAG PABBARAJU

Discipline: COMPUTER SCIENCE ENGINEERING

Name of the Guide: E.SOUNDARARAJAN

Designation of the Guide: OFFICER-IN CHARGE, DIGITAL TECHNOLOGY SECTION, SIRD, RMG, IGCAR

Name of the PS Faculty: DR. MICAHEL ALPHONSE

Key Words: NOSQL, BIG DATA, UNSTRUCTURED DATA, MONGODB, DISTRIBUTED STORAGE

Project Areas: DATABASE MANAGEMENT SYSTEMS, BIG DATA HANDLING

ACKNOWLEDGEMENTS

I thank the management of BITS-Pilani and **Dr. A.K. BHADURI**, Director, IGCAR for giving me an opportunity to undergo my Practice School-1 program at such an esteemed Institution. It has been an honor and a privilege to be able to undergo training at such a prestigious institution.

I would like to express my gratitude to the PS program coordinator at IGCAR, **DR.M.SAIBABA**, Associate Director, Resource Management Group, IGCAR.

I am thankful to **Mr. E. SOUNDARARAJAN,** my PS guide for giving me an opportunity to study and do my project under him. The environment created by working under him was extremely conducive to the completion of my project.

I am grateful to **Dr. MICHAEL ALPHONSE** (Faculty, BITS Pilani Hyderabad Campus), for his continuous guidance and assistance in academic as well as non-academic related matters.

I would like to thank **Dr. KABALI** for his support and coordination of all the PS related activities.

I would like to thank all my **friends** for providing their invaluable support and help.

ABSTRACT

An ever-increasing quantity of varied forms of Big Data, characterized by 3 V's – Variety, Volume and Velocity, is being produced and consumed over the network day by day. 80-90% of all potentially usable business information, originates in unstructured form, that is a form not having a pre-defined data model or not being organized in a pre-defined manner, e.g., emails, images, audio files, video files, etc. Relational databases were not designed to cope in mass with such large volumes of unstructured data, nor could they deal with the scale and agility challenges that face modern applications. Consequently, NoSQL databases were introduced, that provide a mechanism for storage and retrieval of data modeled in means other than tabular relations used in relational databases. One of the widely used NoSQL databases, MongoDB, is a document-oriented database, which stores data as documents having a dynamic schema grouped into collections, in the form of key-value pairs. MongoDB has provisions of Replication and Sharding, which allow horizontal scaling of data into multiple database servers. SIRD at IGCAR has been generating a vast amount of unstructured data over the past years. The main objective of this paper is to describe the method of storing this large amount of unstructured data into a distributed database storage environment, like a multi-sharded MongoDB cluster, with data files with a common meta data attribute like its last date modified, stored together. The data is distributed amongst the participant servers using an efficient technique known as Consistent Hashing, which assigns a hash value within a hash ring to every database server. This enables correct retrieval of data to a large degree, from the respective server in which it is stored, even when server failure might occur, or servers get added. This methodology is powerful not only to enhance the data availability, but also to improve the database server's scalability, as well as efficient retrieval from the corresponding location of the required data from the distributed database environment.

Signature(s) of Student(s) Signature of PS Faculty

Date: Date:

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1. ORGANISATION OVERVIEW

Indira Gandhi Centre for Atomic Research [IGCAR], the second largest establishment of the Department of Atomic Energy next to Bhabha Atomic Research Centre, was set up at Kalpakkam, 80 Kms south of Chennai [MADRAS], in 1971 with the main objective of conducting broad based multidisciplinary programme of scientific research and advanced Engineering, directed towards the development of sodium cooled Fast Breeder Reactor [FBR] technology, in India. This is part of the second stage of Indian Atomic Energy Programme, which is aimed at preparing the country for utilization of the extensive Thorium reserves and providing means to meet the large demands of electrical energy in 21st century.

In meeting the objectives, a modest beginning was made by constructing a sodium cooled Fast Breeder Test Reactor [FBTR], with a nominal power of 40 MW, based on the French Reactor, RAPSODIE. The reactor attained its first criticality on 18th Oct, 1985 and has been in operation at its maximum attainable power level of 10.5 MW with a small core. It is the first of its kind in the world to use Plutonium Uranium mixed carbide as a driver fuel.

With the experience and expertise gained by the successful operation of FBTR, the Centre has embarked upon the design and construction of 500 MW, Prototype Fast Breeder Reactor [PFBR]. As a part of efforts for closing the fuel cycle, a Fast Reactor Fuel Reprocessing Plant is under construction. A 30 KW, U233 fuelled mini reactor [KAMINI] has been made operational for neutron radiography, neutron activation analysis etc.,

IGCAR utilizes its expertise and resources in enhancing its standing as a leading Centre of research in various branches of basic, applied and engineering sciences that have a bearing on Nuclear Technology like Structural Mechanics, Heat and MassTransfer, Material Science, Fabrication Processes, Non-Destructive Testing, Chemicalsensors, High temperature thermodynamics, Radiation Physics, Computer science etc.,

Apart from thrust areas related to nuclear technology, the Centre has credentials as a leader of research in various frontier and topical subjects like Quasi crystals, Oxide superconductors, Nano-structures, clusters, SQUID fabrication programs, exopolymers and experimental simulation of condensed matter using colloids etc.

IGCAR has extended its expertise and facilities to other vital sectors such as Defence, Space and other industries of India to develop techniques for reliable solutions to specialized problems. It has collaborations with educational and R & D institutes like Indian Institutes of Technology, Indian Institute of Science, BITS-Pilani, Regional Engineering Colleges, National Research Laboratories, Public Units and Institutes abroad.

1.2. SCIENTIFIC INFORMATION RESOURCE DIVISION (SIRD)

The Scientific Information Resource Division (SIRD) unit, (formerly Library and Information Services (LS&IS)) at Indira Gandhi Center for Atomic Research (IGCAR) was established in 1973. The SIRD at IGCAR is one of the most modern information resource centers in India

SIRD is in pursuance of its objectives of providing efficient and effective information services to its scientists and engineers working in the various units of DAE at Kalpakkam. The library provides usual services such as lending and reference. It holds in it a collection of more than 68,000 books, 32,000 back volumes of journals, 2,05,000 reports, 15,000 Standards and subscribes to 805 journals every year. SIRD is finding innovative methods to serve the needs of its users who are located over a large campus sprawling around 10 sq. kilometers. This is achieved by constantly enhancing and modernizing the library systems and services,

The SIRD not only focuses on making the content available but also on saving the user's time to acquire that information. The SIRD is developing various ways to display the information in a semantic way instead of a structural way to provide efficient searching. The library has hosted many services in its servers like Vaigai**,** Bodhi**,** and Amaravathi to make the information available

at any place inside the campus. These are run efficiently on the existing hardware by using virtualization technology.

A multimedia workstation consisting of scanner, audio-video embedding facilities and authoring tools, is used for preparing multimedia presentations for lectures and seminars as well as for Computer Based Tutorials. This is achieved by constantly enhancing and modernizing the digital library infrastructure and services. Excellent infrastructural facilities like advanced digital library servers, storage solutions, content creation facilities, enhanced e-resources, RFID based library management, are available in addition to conventional library services.

1.3. EXISTING SYSTEM AND NEED FOR THE NOSQL APPROACH

With the world at an advent of data explosion, the management and storage of the humongous quantity of data becomes a challenge. Traditional Relational databases, that store data records in tabular structures with a pre-defined data schema, have been powering a number of business applications over the years. However, RDBMS's proved to be incompetent in the storage of the huge chunks of varied data files bombarded onto the Internet. Relational Databases did not support storage of data having dynamic schema. It was also difficult to scale out the data storage across multiple database servers in RDBMS's, since that required complex join and merging operations. Storage in a local file system was also not a definitive solution, because it did not support efficient queries for specific data retrieval. Also, an index was difficult to maintain in a local file system.

To cope up with the Big Data storage demands, NoSQL database designs are being developed, that provide a mechanism for the management of data modeled in means other than tabular relations. NoSQL databases focus on horizontal scaling of data across multiple machines (cluster), which is a problem with relational databases. Various flavours of NoSQL databases use different data structures for the storage of data, like key-value pairs, documents, graphs, wide columns, etc. However, they essentially have the provision of dynamic schema for the storage of data files, that is, a strict pre-defined structure need not be necessary. NoSQL databases don't completely abide by the ACID properties (Atomicity, Consistency, Isolation, Durability), but are instead based on the paradigm of “eventual consistency” meaning that databases changes are

propagated to all the database nodes eventually. NoSQL databases also support redundant database servers storing replicated data in a cluster that ensure that data is not lost in the event of a failure at a single point.

2. TERMINOLOGIES AND CONCEPTS

2.1 UNSTRUCTURED DATA

Unstructured data is a generic label for describing data that is not contained in a database or some other type of data structure. Unstructured data can be textual or non-textual. Textual unstructured data is generated in media like email messages, PowerPoint presentations, Word documents, collaboration software and instant messages. Non-textual unstructured data is generated in media like JPEG images, MP3 audio files and Flash video files.

If left unmanaged, the sheer volume of unstructured data that is generated each year within an enterprise can be costly in terms of storage. Unmanaged data can also pose a liability if information cannot be located in the event of a compliance or lawsuit. The information contained in unstructured data is not always easy to locate. It requires that data in both electronic and hard copy documents and other media be scanned so a search application can parse out concepts based on words used in specific contexts. This is called semantic search. It is also referred to as enterprise search.

In customer-facing businesses, the information contained in unstructured data can be analyzed to improve customer and relationship marketing. As social media applications like Twitter and Facebook go mainstream, the growth of unstructured data is expected to far outpace the growth of structured data. According to the "IDC Enterprise Disk Storage Consumption Model" report released in fall 2009, while transactional data is projected to grow at a compound annual growth rate (CAGR) of 21.8%, it's far outpaced by a 61.7% CAGR prediction for unstructured data.

2.2. UNSTRUCTURED DATA VS STRUCTURED DATA

The term Structured Data came from the name for a common language used to access databases, called Structured Query Language, or SQL. Structured data refers to kinds of data with a high level of organization, such as information in a relational database. The database records are restricted to having a pre-defined format, in the sense that their attributes are already defined as the columns in the database table.

Unstructured Data (or unstructured information) refers to information that either does not have a pre-defined data model or is not organized in a pre-defined manner. Unstructured information is typically text-heavy, but may contain data such as dates, numbers, and facts as well. This results in irregularities and ambiguities that make it difficult to understand using traditional programs as compared to data stored in fielded form in databases or annotated (semantically tagged) in documents. Rich data types include things such as pictures, music, movies, and x-rays. Even basic office document types such as Word and PowerPoint are becoming increasingly rich media containers, where it's now easy for a user to embed much more than just text.

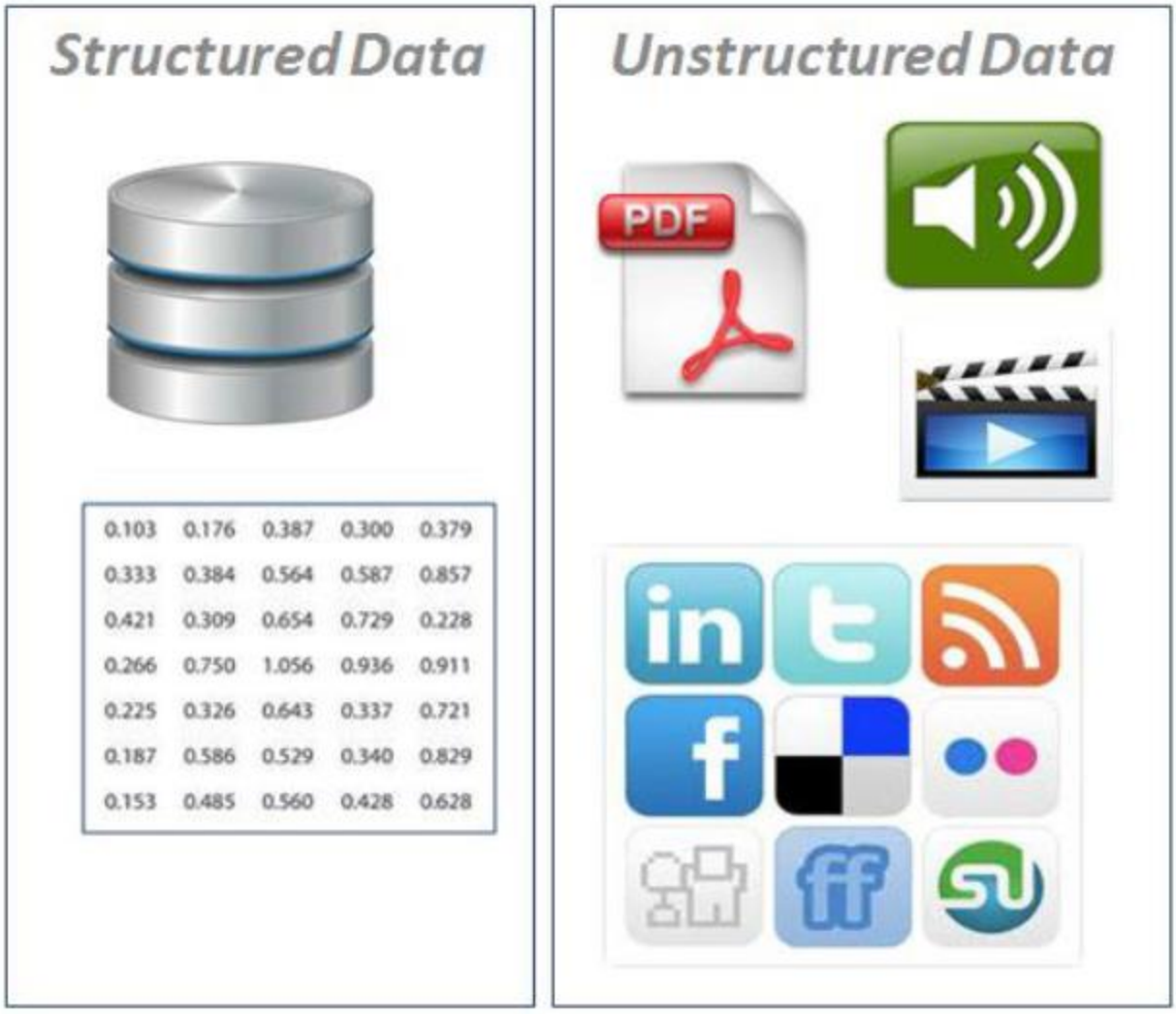

Figure 1. Structured Data vs Unstructured Data

Figure 2. Different Sources of Unstructured Data

2.3 SOURCES OF UNSTRUCTURED DATA IN BIG DATA ENVIRONMENT

Here are some examples of machine-generated unstructured data:

1) Satellite images:

This includes weather data or the data that the government captures in its satellite surveillance imagery. Just think about Google Earth, and you get the picture.

2) Scientific data:

This includes seismic imagery, atmospheric data, and high energy physics.

3) Photographs and video:

This includes security, surveillance, and traffic video.

4) Radar or sonar data:

This includes vehicular, meteorological, and oceanographic seismic profiles.

2.4 NOSQL

NoSQL encompasses a wide variety of different database technologies that were developed in response to a rise in the volume of data stored about users, objects and products, the frequency in which this data is accessed, and performance and processing needs. Relational databases, on the other hand, were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the cheap storage and processing power available today.

2.4.1 NOSQL DATA TYPES

1) Document databases pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.

2) Graph stores are used to store information about networks, such as social connections. Graph stores include Neo4J and Hyper Graph DB.

3) Key-value stores are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its value. Examples of key-value stores are Riak and Voldemort. Some key-value stores, such as Redis, allow each value to have a type, such as "integer", which adds functionality.

4) Wide-column stores such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows.

2.4.2 DYNAMIC SCHEMA

Relational databases require that the schema be defined before you can add data. For example, you might want to store data about your customers such as phone numbers, first and last name, address, city and state – a SQL database needs to know what you are storing in advance.

This fits poorly with agile development approaches, because each time you complete new features, the schema of your database often needs to change. So if it is required, that a few iterations into development, it is required to store customers' favorite items in addition to their addresses and phone numbers, it would be needed to add that column to the database, and then migrate the entire database to the new schema.

If the database is large, this is a very slow process that involves significant downtime. If you are frequently changing the data your application stores – because you are iterating rapidly – this downtime may also be frequent. There's also no way, using a relational database, to effectively address data that's completely unstructured or unknown in advance.

NoSQL databases are built to allow the insertion of data without a predefined schema. That makes it easy to make significant application changes in real-time, without worrying about service interruptions – which means development is faster, code integration is more reliable, and less database administrator time is needed.

2.4.3 AUTO SHARDING

Because of the way they are structured, relational databases usually scale vertically – a single server has to host the entire database to ensure reliability and continuous availability of data. This gets expensive quickly, places limits on scale, and creates a relatively small number of failure points for database infrastructure. The solution is to scale horizontally, by adding servers instead of concentrating more capacity in a single server.

"Sharding" a database across many server instances can be achieved with SQL databases, but usually is accomplished through SANs, and other complex arrangements for making hardware act as a single server. Because the database does not provide this ability natively, development teams take on the work of deploying multiple relational databases across a number of machines. Data is stored in each database instance autonomously. Application code is developed to distribute the data, distribute queries, and aggregate the results of data across all of the database instances. Additional code must be developed to handle resource failures, to perform joins across the different databases, for data rebalancing, replication, and other requirements. Furthermore, many benefits of the relational database, such as transactional integrity, are compromised or eliminated when employing manual sharding.

NoSQL databases, on the other hand, usually support auto-sharding, meaning that they natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool. Data and query load are automatically balanced across servers, and when a server goes down, it can be quickly and transparently replaced with no application disruption.

Cloud computing makes this significantly easier, with providers such as Amazon Web Services providing virtually unlimited capacity on demand, and taking care of all the necessary database administration tasks. Developers no longer need to construct complex, expensive platforms to support their applications, and can concentrate on writing application code. Commodity servers can provide the same processing and storage capabilities as a single high-end server for a fraction of the price.

2.5 THE CAP THEOREM

In order to store and process massive datasets, a commonly employed strategy is to partition the data and store the partitions across different server nodes. Additionally, these partitions can also be replicated in multiple servers so that the data is still available even in case of servers' failures. Many modern data stores, such as Cassandra and Big Table, use these and other strategies to implement highly-available and scalable solutions that can be leveraged in cloud environments. Nevertheless, these solutions and other replicated networked data stores have an important restriction, which was formalized by the CAP theorem: only two of three CAP properties (consistency, availability, and partition tolerance) can be satisfied by networked shared-data systems at the same time. Consistency, as interpreted in CAP, is equivalent to having a single up-to-date instance of the data.

Therefore, consistency in CAP has a somewhat dissimilar meaning to and represents only a subset of consistency as defined in ACID (Atomicity, Consistency, Isolation and Durability) transactions of RDBMSs, which usually refers to the capability of maintaining the database in a consistent state at all times. The Availability property means that the data should be available to serve a request at the moment it is needed. Finally, the Partition Tolerance property refers to the capacity of the networked shared-data system to tolerate network partitions. The simplest interpretation of the CAP theorem is to consider a distributed data store partitioned into two sets of participant nodes; if the data store denies all write requests in both partitions, it will remain consistent, but it is not available. On the other hand, if one (or both) of the partitions accepts write requests, the data store is available, but potentially inconsistent. Despite the relative simplicity of its result, the CAP theorem has had important implications and has originated a great variety of distributed data stores aiming to explore the trade-offs between the three properties. More specifically, the challenges of RDBMS in handling Big Data and the use of distributed systems techniques in the context of the CAP theorem led to the development of new classes of data stores called NoSQL and NewSQL.

3. MODELLING OF THE DATABASE STORAGE ENVIRONMENT

3.1 MONGODB

MongoDB is an open source document-oriented database and one of the leading NoSQL databases written in C++. The word 'Mongo' is coined form 'humongous', meaning that the database is capable of managing huge amounts of structured as well as unstructured data. MongoDB provides for the storage of data files having a changing schema, in the sense that the documents need not have the same set of fields, and the fields having the same names in different documents need not have the same data type. The document oriented structure also has the provision of embedded documents, removing the need for complex joins between related tables in traditional relational databases.

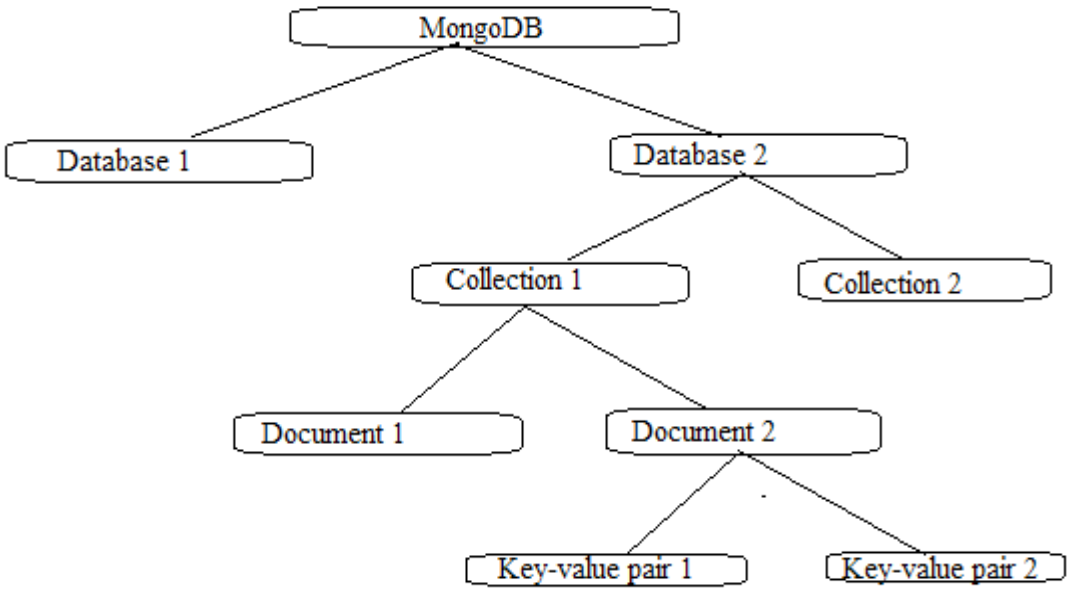

Figure 3. Structure of MongoDB

3.1.1 DATA MODEL

JavaScript Object Notation (JSON) is an open, human and machine-readable standard that facilitates data interchange, and along with XML is the main format for data interchange used on the modern web. JSON supports all the basic data types you'd expect: numbers, strings, and boolean values, as well as arrays and hashes. MongoDB use JSON documents in order to store records, just as tables and rows store records in a relational database. Here is an example of a JSON document:

```
{
    '_id' : 1,
    'name' : { 'first' : 'John', 'last' : 'Backus' },
    'contribs' : [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
    'awards' : [
        {
            'award' : 'W.W. McDowell Award',
            'year' : 1967,
            'by' : 'IEEE Computer Society'
        }, {
            'award' : 'Draper Prize',
            'year' : 1993,
            'by' : 'National Academy of Engineering'
        }
    ]
}
```

Figure 4. A Sample JSON Document

A JSON database returns query results that can be easily parsed, with little or no transformation, directly by JavaScript and most popular programming languages – reducing the amount of logic you need to build into your application layer.

MongoDB represents JSON documents in binary-encoded format called BSON behind the scenes. BSON extends the JSON model to provide additional data types and to be efficient for encoding and decoding within different languages.

The key advantage of using BSON is efficiency, as it is a binary format. Documents are contained in “collections”, they can be seen as an equivalent to relational database tables. Collections can contain any kind of document, no relationship is enforced, and still documents within a collection usually have the same structure as it provides a logical way to organize data. As data within collections is usually contiguous on disk, if collections are smaller better performance is achieved. Each document is identified by a unique ID (“–id” field), which can be given by the user upon document creating or automatically generated by the database. An index is automatically created on the ID field although other indexes can be manually created in order to speed up common queries. Relationships can be modeled in two different ways embedding documents or referencing documents. Embedding documents means that a document might contain other data fields related to the document, i.e. a document modeling a blog post would also contain the post’s comments. This option might lead to de-normalization of the database, as the same data might be embedded in different documents. Referencing documents can be seen as the relational database equivalent of using a foreign-key. Instead of embedding the whole data, the document might instead store the ID of the foreign document so that it can fetch it efficiently.

For the insertion of files like images, audio files, etc., that exceed the standard document size of 16Mb, MongoDB provides an implementation known as GridFS, that divides a file into chunks (of maximum size 255k), and stores each chunk of data in a separate document. These chunks themselves are stored in MongoDB collections.

3.1.2 QUERY MODEL

Many traditional SQL queries have a similar counterpart on Mongo DB’s query Language. Queries are expressed as JSON objects and are sent to Mongo DB by the database driver (typically using the “find” method). More complex queries can be expressed using a Map Reduce operation, and it may be useful for batch processing of data and aggregation operations. The user specifies the map and reduces functions in JavaScript and they are executed on the server side. The results of the operation are stored in a temporary collection which is automatically removed after the client gets the results. It is also possible for the results to be stored in a permanent collection, so that they are always available.

3.1.3 REPLICATION AND SHARDING

As the size of the data to be stored increases beyond bounds, vertical scaling, that is, the piling up of the data on the same database server is extremely inefficient. On one hand, it is extremely expensive to increase the memory of a single machine. On the other hand, storage of the data is extremely risk-prone, since failure at a single point can lead to the loss of the entire data stored.

Sharding or horizontal scaling, is MongoDB's approach to distributing the data set across multiple machines, or shards. Sharding reduces the number of operations each shard handles, and as a result, the cluster increases capacity and throughput horizontally. MongoDB exhibits sharding through sharded clusters. A sharded cluster (Figure 1) consists of config servers, query routers and the shards that store the data. Shards are the data stores, that are the individual database servers in the cluster. Query routers, or specifically mongos instances, are the interface between client applications and the various shards. The mongos instance directs the client's request to the concerned shard, and returns the results to the client. Config servers hold the cluster's metadata, that is, they store the mapping details of the data to the various shards. The query router uses this metadata to get routed to the shard the client application is trying to access.

To ensure further availability and prevent loss of data due to individual server crashes, MongoDB replicates the data in each shard into multiple redundant database servers that constitute a replica set. A replica set follows a master slave configuration, and consists of a primary server as well as multiple secondary servers. Read and write operations are directed to the primary server, and the changes are then replicated in all the secondary servers in the replica set. In case the primary server happens to fail, the data is still available in the secondary servers. The secondary servers elect a new primary server amongst themselves, and the structure of the replica set is mainatained.

Thus, through Replication and Sharding, MongoDB provides for high availability, as well as high scalability, which results in efficient maintenance and processing of queries.

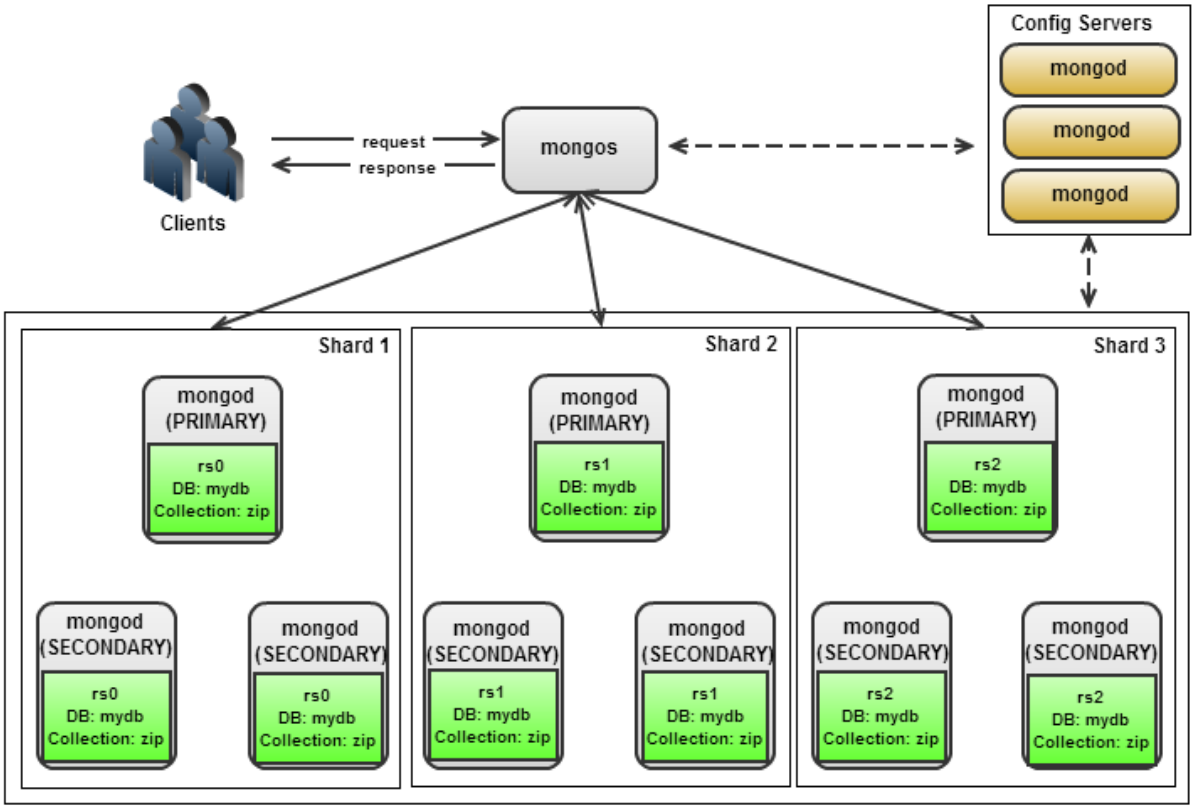

Figure 5. Sharding and Replication in MongoDB

3.2 CONSISTENT HASHING ALGORITHM

Conventional Hashing techniques assign buckets to objects using a formula similar to Hashcode(object) modulo N), where N is the number of buckets available. In the event of server failure, or when additional database servers are added, the parameter 'n' changes. Thus, every previously inserted object will now be pointed to an altogether different shard by the hash formula mentioned above. If we now wish to retrieve a data file, the Hashing function mentioned above will redirect us to a database server into which the object might not have been originally inserted, leading us to believe that the file is missing. Thus, all files have to be reallocated once again, even when a single server is added/deleted.

Consistent Hashing largely solves this problem. It is a methodology to efficiently distribute data files across multiple shards, ensuring minimum reallocation in the event of server addition/removal. A Hash Ring (Figure 2) is maintained, into which all database servers as well as the Data files are hashed to. Every server and data file gets hashed to a unique position on the

ring. For example, a ring consisting of the values -2^{32} to 2^{32} can be constructed. A good hashing function like the MD5 hashing function can assign every data file as well as server to a point on this ring. The shard which a data file gets allocated is the server which is closest to the position of the data file on the hash ring in the clockwise direction. Supposing a server is now added, only the data files in the sector of the ring that were getting hashed to the server near the new server's hash on the ring have to be considered for reallocation. Thus, a maximum of a fraction of 1/N data files have to be reallocated, when the (N+1)th server is added. Similar is the case with removal of servers.

The data load of each server can be balanced with the introduction of virtual servers on the ring. Every physical server can have replicated representatives hashed to different locations on the hash ring. With a fairly large number of replicas for each physical server, the load of storing the huge database can be evenly distributed. Widely used databases like Memcached, Apache's Dynamo, Cassandra, etc. have all adopted the technique of consistent hashing in some form to ensure scalability and availability.

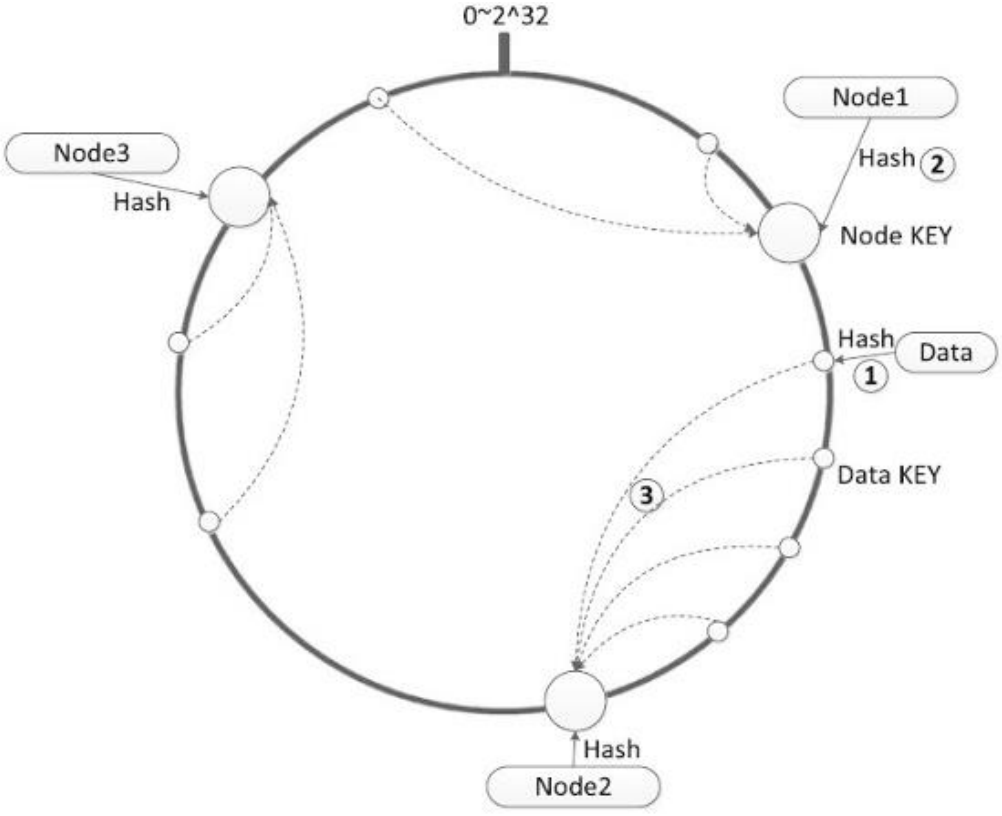

Figure 6. The Consistent Hashing ring

4. SOFTWARE AND HARD WARE REQUIREMENTS

Hardware Requirements:

Processor: Intel(R) Core™ i5 CPU 680 @ 3.60GHz

Memory: 4GB

System Type: 64 bit operating system

Software Requirements:

ECLIPSE IDE

MONGODB

5. CASE STUDY AT SIRD, IGCAR AND RESULTS

Scientific Information and Resource Division (SIRD) is responsible for providing efficient and effective library and information retrieval services at IGCAR, Kalpakkam. SIRD has been generating huge volumes of data in the past years. The data consists of a collection of various data files, including photos, videos, e-journals, ppt files, audio files, word files, applications, excel datasheets, etc. The data has been dumped into various local storage servers, and is in an extremelyunstructured state. The retrieval of data files from this huge database requires the File Explorer to sift through the entire set of files and is inefficient.

This unstructured data can be stored in a multi sharded MongoDB cluster, with the config server storing the metadata about the data files stored in each server. The data files could be distributed amongst the servers using the technique of consistent hashing described above, supporting addition and removal of servers with minimum reallocation of files. The files could be hashed on their date of modifications. Thus, files modified on the same date will get hashed to the same shard. The extraction of files modified on a certain date, from the huge set of files having various dates of modifications simply requires querying the mongos instance. The input date would get hashed to the shard at which the files with the date of modification having the input date

had been hashed to. The search would then be localized to an extremely small domain, as compared to having to sift through the entire data set on a whole.

Sample data sets ranging from 50GB to 600GB, having all varieties data files, were inserted into the clustered Mongo DB, being hashed on their dates of modification. Time for retrieval of data files with an input date, both in the MongoDB, as well as the normal system file explorer were recorded for each data set size. The trends were plotted on a graph (Figure 4):

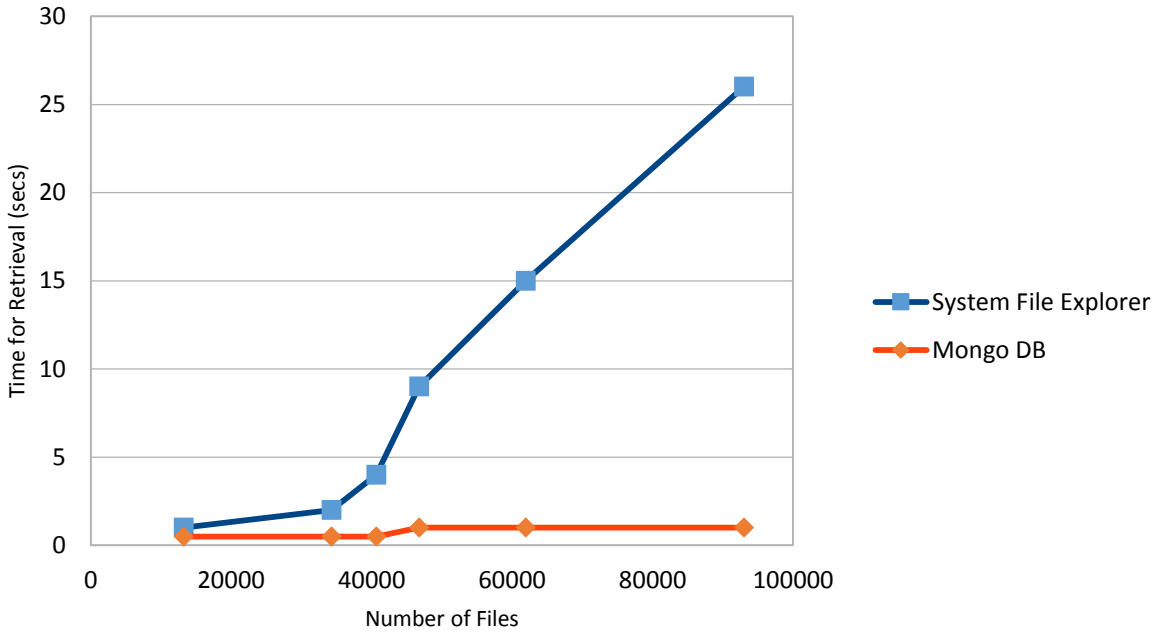

Figure 7. Plot of time to retrieve data files vs Number of Files

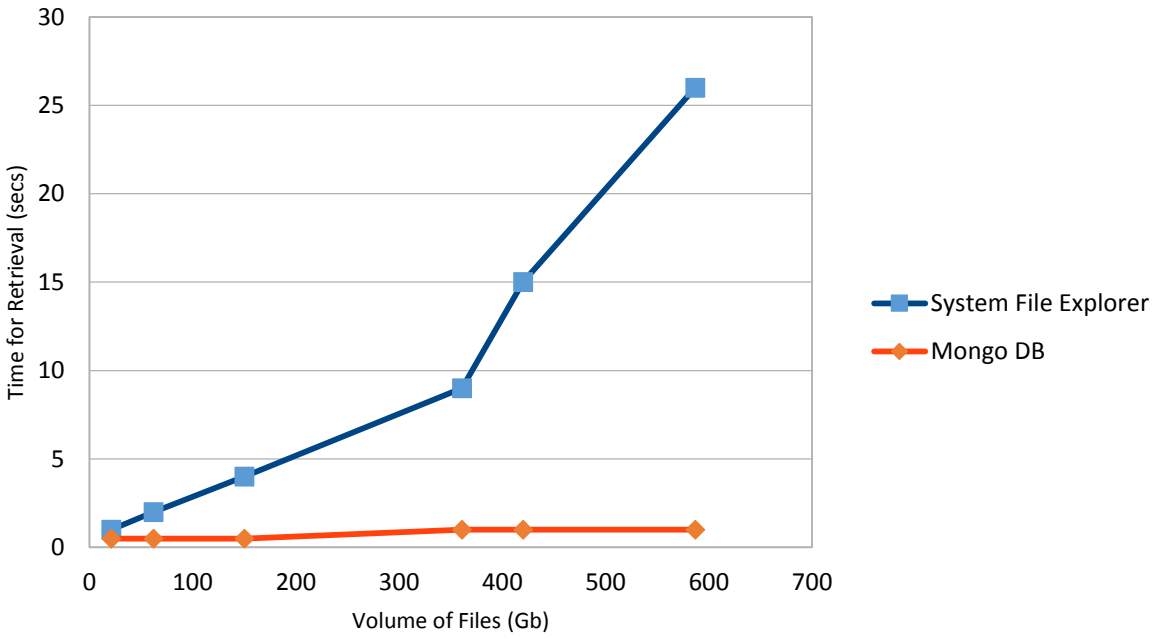

Figure 8. Plot of time to retrieve data files vs Volume of Files

As the number of files stored in the File System increases, the retrieval of a particular file takes considerable time. The time taken to retrieve a file approximately increases linearly with the number of files and correspondingly, with the volume of data.

On the other hand, in the distributed Mongo DB database, the files are hashed on their dates and are correspondingly stored in their respective shards. Thus, for an input date query, the search is localized to only that node, to which the input date hashes, and thus the domain to search for is drastically reduced. This is reflected by the approximately constant time for retrieval in the MongoDB distributed database system.

In this way, a distributed Mongo DB storage environment is ensuring high availability due to redundant replicated database servers, high scalability because of Sharding, centrally regulated by a config server, as well as efficient retrieval because of the consistent hashing technique, which also ensures prevention of data loss/reallocation in the event of addition/removal of participant database servers.

6. CONCLUSION

In this research work, a large dataset of various types of unstructured data has been successfully inserted into a distributed database environment like a multi-sharded MongoDB cluster, providing horizontal scalability of the database. Consistent Hashing has been used to distribute the dataset into multiple shards providing even load balancing, as well as minimum reallocation of data in the case of a changing number of database servers. Since the files have been hashed on an attribute in their meta data (here: Date of Modification), retrieval of the files based on an input query on that attribute becomes very efficient in terms of time, since the search gets localized to one shard from a large number of shards, storing smaller chunks of the huge database. Thus, features like scalability, availability, and efficiency in terms of retrieval have been greatly exhibited in the clustered MongoDB environment, with consistent hashing.

7. SCOPE FOR FUTURE WORK

In the future, this implementation can be incorporated into the cloud environment to handle humongous quantities of data like tens to hundreds of terabytes. This provision is necessary, since application users of Facebook, Whatsapp, Instagram etc are bombarding huge quantities of unstructured data onto the web. Handling and storage of such huge datasets in the cloud environment is the future challenge.

REFERENCES

1. MongoDB Documentation- https://docs.mongodb.com/manual/

2. MongoDB Tutorials - http://www.tutorialspoint.com/mongodb/

3. Consistent Hashing - http://www.tom-e-white.com/2007/11/consistent-hashing.html

4. JAVA – The Complete Reference, by Herbert Schildt

5. MongoDB Java API - https://api.mongodb.com/java/current/

6. Wikipedia - https://en.wikipedia.org/wiki/NoSQL

7. Clarence J. M. Tauro. "A Comparative Analysis of Different NoSQL Databases on Data Model,Query Model and Replication Model". ACM SIGOPS Operating Systems Review archive Volume 44 Issue 2, April 2010 Pages 35-40

8. Banker K (2011) "MongoDB in action". Manning Press, USA

9. Jiang, Wenbin et al. "A Novel clustered MongoDB-based storage system for unstructured data with high availability". Springer-Verlag. Wien 2013

10. Abadi D (2012) Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. Computer 45(2):37–42

APPENDIX

A. SOURCE CODE FOR CONSISTENT HASHING

```
package mongowork;

import java.io.UnsupportedEncodingException;

import java.security.MessageDigest;

import java.util.*;


public class ConsistentHash {


        private int replicas;

        private SortedMap<Integer,String> ring = new TreeMap<Integer,String>();


        public ConsistentHash(List<String> shards, int replicas){

                this.replicas=replicas;

                for (String shard:shards)

                        addToRing(shard);

        }

        //add servers to points on the hash ring

        private void addToRing(String shard){

                for (int i=0;i<replicas;i++){
```

```
                ring.put(getMD5(shard).hashCode()+(i*1135262919),shard);

        }
}

//returns shard assigned to a key
public String getShard (String s){
        int hash = getMD5(s).hashCode();
        if (!ring.containsKey(hash)){
                SortedMap<Integer,String> tmap = ring.tailMap(hash);
                hash = tmap.isEmpty() ? ring.firstKey() : tmap.firstKey();
        }
        return ring.get(hash);
}

//MD5 Hash to generate a good hash
private static String getMD5(String input) {
        byte[] source;
                try {
                //Get byte according by specified coding.
                source = input.getBytes("UTF-8");
```

```
} catch (UnsupportedEncodingException e) {
source = input.getBytes();
}
String result = null;
char hexDigits[] = {'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};
try {
MessageDigest md = MessageDigest.getInstance("MD5");
md.update(source);
//The result should be one 128 integer
byte temp[] = md.digest();
char str[] = new char[16 * 2];
int k = 0;
for (int i = 0; i < 16; i++) {
byte byte0 = temp[i];
str[k++] = hexDigits[byte0 >>> 4 & 0xf];
str[k++] = hexDigits[byte0 & 0xf];
}
result = new String(str);
} catch (Exception e) {
e.printStackTrace();
```

```
                }
                return result;
            }
}
```

B: SOURCE CODE FOR INSERTION AND RETRIEVAL

```
package mongowork;

import java.net.UnknownHostException;
import java.util.*;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.UnsupportedEncodingException;
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.text.SimpleDateFormat;
import java.lang.*;

import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
```

```
import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.gridfs.GridFS;
import com.mongodb.gridfs.GridFSDBFile;
import com.mongodb.gridfs.GridFSInputFile;


public class test1 {

	public static void main (String[] args) throws IOException{
		final long startTime = System.nanoTime();
		List<String> shards= new ArrayList<String>();
		shards.add("PC1shard1");
		shards.add("PC1shard2");
		shards.add("PC1shard3");
		shards.add("PC1shard4");
		shards.add("PC1shard5");
		shards.add("PC1shard6");

		shards.add("PC2shard1");
		shards.add("PC2shard2");
		shards.add("PC2shard3");
		shards.add("PC2shard4");
		shards.add("PC2shard5");
		shards.add("PC2shard6");
```

```
        ConsistentHash c1= new ConsistentHash(shards,5);

        //System.out.println(c1.getShard("01/07/2018"));
/*      System.out.println(c1.getShard("31/08/2015"));
        System.out.println(c1.getShard("03/07/2005"));
        System.out.println(c1.getShard("01/09/2015"));
        System.out.println(c1.getShard("12/10/2014"));
        System.out.println(c1.getShard("11/06/2013"));  */

        File folder = new File("C:/input/");
        File[] fileList1 = folder.listFiles();
        //System.out.println(f.getName());
        //for (File f: fileList1)
        //System.out.println(f.getName());
        PrintStream holdout=System.out;

        File folder33 = new File("//10.1.6.39/photo42/2014/06 JUNE");
        File[] fileList33 = folder33.listFiles();

        File folder34 = new File("//10.1.6.39/photo42/2014/05 MAY");
        File[] fileList34 = folder34.listFiles();

        File folder35 = new File("//10.1.6.39/photo42/2013/05.MAY");
        File[] fileList35 = folder35.listFiles();

        File folder36 = new File("//10.1.6.39/photo42/2014/08 AUGUST");
        File[] fileList36 = folder36.listFiles();
```

```
File folder37 = new File("//10.1.6.39/photo42/2014/09 SEPTEMBER");
File[] fileList37 = folder37.listFiles();

//can give input date to this function to retrieve all the corresponding files
retrieveByDate ("08/27/2013",c1);

//insert begins
//insertion into the MongoDB servers based on consistent hashes
MongoClient mongo11 = new MongoClient("Lenovo-PC",27021);
MongoClient mongo12 = new MongoClient("Lenovo-PC",27022);
MongoClient mongo13 = new MongoClient("Lenovo-PC",27023);
MongoClient mongo14 = new MongoClient("Lenovo-PC",27024);
MongoClient mongo15 = new MongoClient("Lenovo-PC",27025);
MongoClient mongo16 = new MongoClient("Lenovo-PC",27026);
DB db11 =  mongo11.getDB("chogba");
DB db12 =  mongo12.getDB("chogba");
DB db13 =  mongo13.getDB("chogba");
DB db14 =  mongo14.getDB("chogba");
DB db15 =  mongo15.getDB("chogba");
DB db16 =  mongo16.getDB("chogba");


MongoClient mongo21 = new MongoClient("Lenovo-PC11",27021);
MongoClient mongo22 = new MongoClient("Lenovo-PC11",27022);
MongoClient mongo23 = new MongoClient("Lenovo-PC11",27023);
MongoClient mongo24 = new MongoClient("Lenovo-PC11",27024);
```

```
MongoClient mongo25 = new MongoClient("Lenovo-PC11",27025);
MongoClient mongo26 = new MongoClient("Lenovo-PC11",27026);
DB db21 =  mongo21.getDB("chogba");
DB db22 =  mongo22.getDB("chogba");
DB db23 =  mongo23.getDB("chogba");
DB db24 =  mongo24.getDB("chogba");
DB db25 =  mongo25.getDB("chogba");
DB db26 =  mongo26.getDB("chogba");
SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy HH:mm:ss");

String s= new String();

File[] fileList;

int count =1;
//parsing through all files and inserting them in correct shard – total of 12 shards in 2 host machines
for (File f1: fileList37){
if (f1.isDirectory()){
//System.out.println(f1.getName());
fileList = f1.listFiles();
for (File f: fileList){
if (f.isDirectory())continue;

s=c1.getShard(sdf.format(f.lastModified()).toString().substring(0,10));

if (s.equals("PC1shard1")){
insertIntoMongoDB(db11, f.getName(), f.getAbsolutePath(), "PC1shard1", sdf);
```

```
                }

if (s.equals("PC1shard2")){
        insertIntoMongoDB(db12, f.getName(), f.getAbsolutePath(), "PC1shard2", sdf);

                }

if (s.equals("PC1shard3")){
        insertIntoMongoDB(db13, f.getName(), f.getAbsolutePath(), "PC1shard3", sdf);

                }

if (s.equals("PC1shard4")){
        insertIntoMongoDB(db14, f.getName(), f.getAbsolutePath(), "PC1shard4", sdf);

                }

if (s.equals("PC1shard5")){
        insertIntoMongoDB(db15, f.getName(), f.getAbsolutePath(), "PC1shard5", sdf);

                }

if (s.equals("PC1shard6")){
        insertIntoMongoDB(db16, f.getName(), f.getAbsolutePath(), "PC1shard6", sdf);

                }
```

```
if (s.equals("PC2shard1")){
        insertIntoMongoDB(db21, f.getName(), f.getAbsolutePath(), "PC2shard1", sdf);

                }

if (s.equals("PC2shard2")){
        insertIntoMongoDB(db22, f.getName(), f.getAbsolutePath(), "PC2shard2", sdf);

                }

if (s.equals("PC2shard3")){
        insertIntoMongoDB(db23, f.getName(), f.getAbsolutePath(), "PC2shard3", sdf);

                }

if (s.equals("PC2shard4")){
        insertIntoMongoDB(db24, f.getName(), f.getAbsolutePath(), "PC2shard4", sdf);

                }

if (s.equals("PC2shard5")){
        insertIntoMongoDB(db25, f.getName(), f.getAbsolutePath(), "PC2shard5", sdf);

                }

if (s.equals("PC2shard6")){
```

```
                insertIntoMongoDB(db26, f.getName(), f.getAbsolutePath(), "PC2shard6", sdf);

                        }
                System.out.println(count++);

                }

                }
        }
        //insert ends
        System.setOut(holdout);
                System.out.println("DONE");

                final long duration = System.nanoTime() - startTime;

                System.out.println(duration);

        }
//the insertion function – establishes connection with appropriate shard, makes the GridFS object and inserts it
private static void insertIntoMongoDB (DB db, String fname, String pathname, String extension, SimpleDateFormat sdf) throws IOException{

    //MongoClient mongo = new MongoClient("localhost",shardport);
```

```
        //DB db =  mongo.getDB("chogba");
        //SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy HH:mm:ss");



        File file1 =  new File (pathname);
        GridFS gfsinp = new GridFS(db,extension);
        GridFSInputFile gfsfile = gfsinp.createFile(file1);
        gfsfile.setFilename(fname);
        gfsfile.put("date", sdf.format(file1.lastModified()).toString().substring(0,10));
        gfsfile.save();


        }




//computes hash of input date, connects with the respective shard, and outputs all files in a text file
private static void retrieveByDate(String str,ConsistentHash c1) throws FileNotFoundException{

        List<GridFSDBFile> flist1 = new ArrayList<GridFSDBFile>();
        //List<GridFSDBFile> flist2 = new ArrayList<GridFSDBFile>();
```

```
String s =  c1.getShard(str);
//System.out.println(s);

if (s.charAt(2)=='1'){
        MongoClient mongo = new MongoClient("Lenovo-PC",(27020+s.charAt(8)-48));
        DB db =  mongo.getDB("chogba");

        BasicDBObject bd1 = new BasicDBObject();
        bd1.put("date", str);
        GridFS gfsimg = new GridFS(db,s);

        flist1 = gfsimg.find(bd1);
        //flist2.addAll(flist1);
}

else{
        MongoClient mongo = new MongoClient("Lenovo-PC11",(27020+s.charAt(8)-
48));
        DB db =  mongo.getDB("chogba");
```

```
            BasicDBObject bd1 = new BasicDBObject();
            bd1.put("date", str);
            GridFS gfsimg = new GridFS(db,s);

            flist1 = gfsimg.find(bd1);
            //flist2.addAll(flist1);
        }

        System.setOut(new PrintStream(new FileOutputStream("C:/chogba/output.txt")));
        for (int i=0;i<flist1.size();i++)
            System.out.println(flist1.get(i));
        }
}
```