

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/289519291>

# Storing of Unstructured data into MongoDB using Consistent Hashing Algorithm

Thesis · December 2015

DOI: 10.13140/RG.2.1.3749.8961

CITATIONS

0

READS

3,250

1 author:



[Saran Raj](#)

Vyasaka technologies, Tamilnadu

8 PUBLICATIONS 0 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Storing of unstructured data into MongoDB using consistent hashing algorithm. [View project](#)

## **CHAPTER - 1**

### **INTRODUCTION**

#### **1.1 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's 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 Face book 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.

#### **1.2 STRUCTURED VS. UNSTRUCTURED**

##### **1.2.1 Structured Data**

The term SD came from the name for a common language used to access DBs, called Structured Query Language, or SQL.

The little snippet of SQL code here illustrates how an application could retrieve all rows from a table called Book where the Price is greater than 100 and request that the result be sorted in ascending order by title.

Most people haven't written DB code themselves, but they have used an office spreadsheet tool like Excel on Windows or Numbers on the Mac. These tools allow you to create simple DB tables.



Fig. 1.1 Structured Vs. Unstructured

fig 1.1 explains customer records with the typical information — name, address, and so forth. And that's it — you've created a DB table! While sophisticated database management systems (DBMS) allow you to do much more than you can do with a spreadsheet, the base concept is essentially the same: tables with rows and columns, often containing straight ASCII text as this example shows.

Customers: C:\...lition Data Table								
Customer ID	Company	Contact	Contact Title	Address	City	Region	Postal Code	
ALFKI	Alfreds Futt	Maria Anders	Sales Repre...	Obere Str. 57	Berlin		12209	
ANATR	Ana Trujillo	Ana Trujillo	Owner	Avda. de la ...	México D.F.		05021	
ANTON	Antonio Mor	Antonio Mor	Owner	Mataderos	México D.F.		05023	
AROUT	Around the	Thomas Har	Sales Repre...	120 Hanove	London		WA1 1DP	
BERGS	Berglunds s	Christina Be	Order Admi...	Berguvaväg	Luleå		S-951 22	
BLAUS	Blauer See	Hanna Moos	Sales Repre...	Fonsterstr. 57	Mannheim		68306	
BLOMP	Blondel pâr	Frédérique	Marketing M...	24, place K2	Strasbourg		67000	
BOLID	Bólido Comi	Martin Som	Owner	C/ Araguit, 67	Madrid		28023	
BONAP	Bon app'	Laurence L	Owner	12, rue des	Marseille		13006	
BOTTM	Bottom-Doll	Elizabeth Li	Accounting	23 Tsvass...	Tsvassen	BC	T2F 8M4	
BSBEV	B's Beverages	Victoria Ash	Sales Repre...	Fauntleroy	London		EC2 5NT	
CACTU	Cactus Comi	Patricio Sim	Sales Agent	Centro 333	Buenos Aires		1010	
CENTC	Centro com	Francisco C	Marketing M...	Siemas de	México D.F.		05022	
CHOPS	Chop-suey	Yang Wang	Owner	Hauptstr. 29	Bam		3012	
COMME	Comércio M	Pedro Afonso	Sales Assoc	Av. dos Lusí	São Paulo	SP	05432-043	
CONSH	Consolidate	Elizabeth Br	Sales Repre...	Berkley Ga	London		WX1 6LT	
DRACD	Drachenblut	Sven Ottlieb	Order Admi...	Walseweg 21	Aachen		52066	
DUMON	Du monde e	Janine Labr	Owner	67, rue des	Nantes		44000	
EASTC	Eastern Con	Ann Devon	Sales Agent	35 King Geo	London		WC3 6FW	
ERNSH	Ernst Handel	Roland Mein	Sales Mana	Kirchgasse 6	Graz		8010	
FAMIA	Família Anq	Aris Cruz	Marketing A	Rua Orós, 92	São Paulo	SP	05442-030	
FISSA	FISSA Fabri	Diego Roel	Accounting	C/ Moraltarz	Madrid		28034	

Fig. 1.2 Unstructured data in table

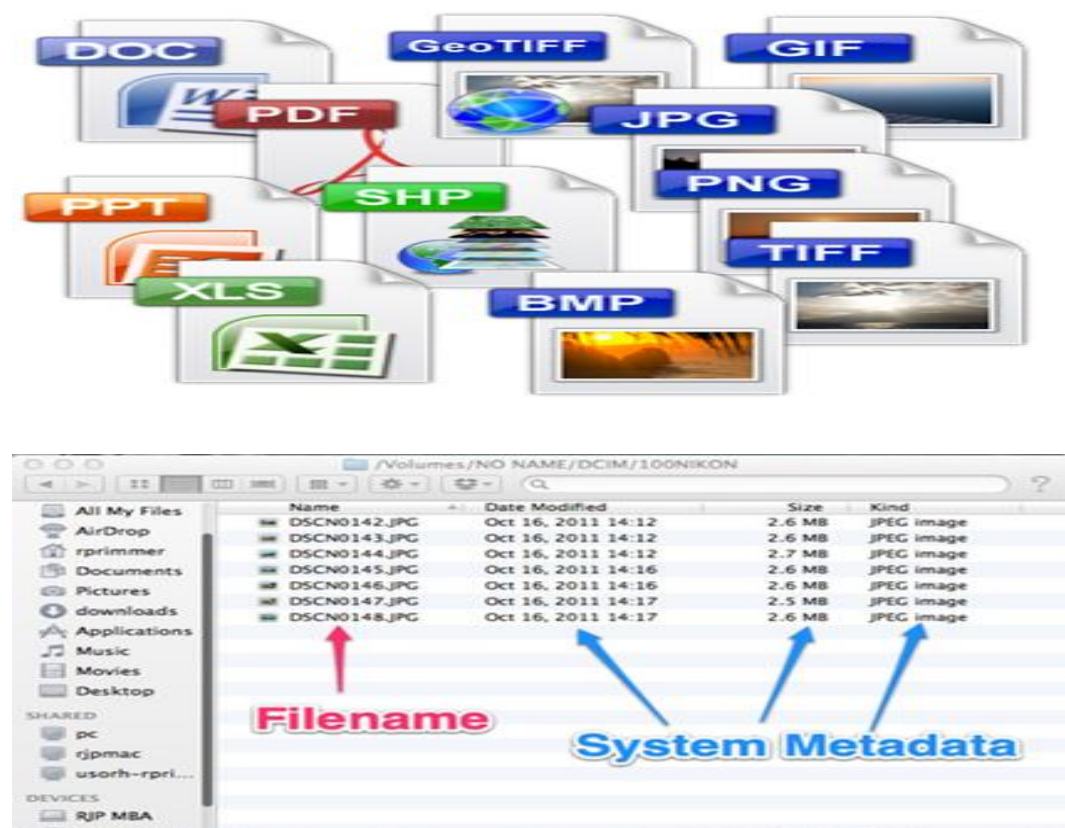
**Fig. 1.2** explains the unstructured data in table how it will be storing format of the unstructured data.

## 1.2.2 Unstructured Data

Earlier, that UD is simply the complement of SD; that is, it's everything other than a DB. That's a pretty broad class. To break that down further, UD can be divided into two subclasses: file and object.

## 1.2.3 Structure of Unstructured Data

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.



**Fig.1.3** Unstructured data types

Fig 1.3 this is an image of a file system on a disk that contains the files created by a digital camera. Each file is an image of type JPG and has associated system metadata (SMD). In this case, the SMD shown is:

- 1) The filename
- 2) The date the file was last modified
- 3) The file size and
- 4) The file type (JPEG image)

This view is quite familiar to computer users and is the principle mechanism for finding and using files on computer.

#### 1.2.4 Why is Unstructured Data Growing So Rapidly

A good question is why would one form of data, UD, grow so much more rapidly than the other? After all, both data types are required, as DBs perform essential functions.

One reason is that the actual user content of a DB is typically text, as shown in the earlier spreadsheet example. For about 40 years, files were likewise most often comprised of just text. But the world has changed. Now users want rich content, not just plain text.

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.



Fig. 1.4 Different kinds of unstructured data's

Fig 1.4 While rich data types provide a far superior user experience over text alone, they do so at the expense of storage space. Rich media types are not just slightly larger than basic text, they can be orders of magnitude larger.

To get a sense of both the difference in user experience and the different storage capacity usage of these two data types, consider this simple example.

### **1.3 Unstructured Data in a Big Data Environment**

Unstructured data is data that does not follow a specified format for big data. If 20 percent of the data available to enterprises is structured data, the other 80 percent is unstructured. Unstructured data is really most of the data that you will encounter. Until recently, however, the technology didn't really support doing much with it except storing it or analyzing it manually.

#### **1.3.1 Sources of unstructured big data**

Unstructured data is everywhere. In fact, most individuals and organizations conduct their lives around unstructured data. Just as with structured data, unstructured data is either machine generated or human generated.

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.

The following list shows a few examples of human-generated unstructured data:

## 1.4 MONGO DB

Mongo DB is an open source database that uses a document-oriented data model. Mongo DB is one of several database types to arise in the mid-2000s under the NoSQL banner. Instead of using tables and rows as in relational databases, Mongo DB is built on an architecture of collections and documents. Documents comprise sets of key-value pairs and are the basic unit of data in Mongo DB. Collections contain sets of documents and function as the equivalent of relational database tables.

Like other NoSQL databases, Mongo DB supports dynamic schema design, allowing the documents in a collection to have different fields and structures. The database uses a document storage and data interchange format called BSON, which provides a binary representation of JSON-like documents. Automatic sharding enables data in a collection to be distributed across multiple systems for horizontal scalability as data volumes increase.

Mongo DB was created by Dwight Merriman and Eliot Horowitz, who had encountered development and scalability issues with traditional relational database approaches while building Web applications at DoubleClick, an Internet advertising company that is now owned by Google Inc. According to Merriman, the name of the database was derived from the word *humongous* to represent the idea of supporting large amounts of data. Merriman and Horowitz helped form 10Gen Inc. in 2007 to commercialize Mongo DB and related software. The company was renamed Mongo DB Inc. in 2013.

The database was released to open source in 2009 and is available under the terms of the Free Software Foundation's GNU AGPL Version 3.0 commercial license. At the time of this writing, among other users, the insurance company MetLife is using Mongo DB for customer service applications, the website Craigslist is using it for archiving data, the CERN physics lab is using it for data aggregation and discovery and the New York Times newspaper is using Mongo DB to support a form-building application for photo submissions.

**1.5 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.

### 1.5.1 No SQL Database 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.

### 1.5.2 Dynamic Schemas

Relational databases require that schemas 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 you decide, a few iterations into development, that you'd like to store customers' favorite items in addition to their addresses and phone numbers, you'll need 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.

### **1.5.3 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.

## **CHAPTER 2**

### **LITERATURE REVIEW**

#### **2.1 NOSQL AND NEW SQL DATA STORES**

Katarina Grolinger et al.[2] implemented the advances in Web technology and the proliferation of mobile devices and sensors connected to the Internet have resulted in immense processing and storage requirements. Cloud computing has emerged as a paradigm that promises to meet these requirements. This work focuses on the storage aspect of cloud computing, specifically on data management in cloud environments. Traditional relational databases were designed in a different hardware and software era and are facing challenges in meeting the performance and scale requirements of Big Data. NoSQL and NewSQL data stores present themselves as alternatives that can handle huge volume of data. Because of the large number and diversity of existing NoSQL and NewSQL solutions, it is difficult to comprehend the domain and even more challenging to choose an appropriate solution for a specific task. Therefore, this paper reviews NoSQL and NewSQL solutions with the objective of: (1) providing a perspective in the field, (2) providing guidance to practitioners and researchers to choose the appropriate data store, and (3) identifying challenges and opportunities in the field. Specifically, the most prominent solutions are compared focusing on data models, querying, scaling, and security related capabilities. Features driving the ability to scale read requests and write requests, or scaling data storage are investigated, in particular partitioning, replication, consistency, and concurrency control. Furthermore, use cases and scenarios in which NoSQL and NewSQL data stores have been used are discussed and the suitability of various solutions for different sets of applications is examined. Consequently, this study has identified challenges in the field, including the immense diversity and inconsistency of terminologies, limited documentation, sparse comparison and benchmarking criteria, and nonexistence of standardized query languages.

The origin of the NoSQL term is attributed to Johan Oskar son, who used it in 2009 to name a conference about “open-source, distributed, non-relational databases”. Today, the term is used as an acronym for “Not only SQL”, which emphasizes that SQL-style querying, is not the crucial objective of these data stores. Therefore, the term is used as an umbrella classification that includes a large number of immensely diverse data stores that are not based on the relational model, including some solutions designed for very specific applications such

as graph storage. Even though there is no agreement on what exactly constitutes a NoSQL solution, the following set of characteristics is often attributed to them:

Simple and flexible non-relational data models. NoSQL data stores offer flexible schemas or are sometimes completely schema-free and are designed to handle a wide variety of data structures. Current solution data models can be divided into four categories: key-value stores, document stores, column-family stores, and graph databases. Ability to scale horizontally over many commodity servers. Some data stores provide data scaling, while others are more concerned with read and/or write scaling.

Provide high availability. Many NoSQL data stores are aimed towards highly distributed scenarios, and consider partition tolerance as unavoidable. Therefore, in order to provide high availability, these solutions choose to compromise consistency in favor of availability, resulting in AP (Available/ Partition-tolerant) data stores, while most RDBMs are CA (Consistent/Available).

Typically, they do not support ACID transactions as provided by RDBMS. NoSQL data stores are sometimes referred as BASE systems (Basically Available, Soft state, eventually consistent). In this acronym, Basically Available means that the data store is available all the time whenever it is accessed, even if parts of it are unavailable; Soft-state highlights that it does not need to be consistent always and can tolerate inconsistency for a certain time period; and Eventually consistent emphasizes that after a certain time period, the data store comes to a consistent state. However, some NoSQL data stores, such as Couch DB provide ACID compliance.

## **2.2 MONGODB**

### **2.2.1 Non-Relational Database**

Wilson A Higashin [3] implemented the Mongo DB (from "humongous") is an open-source document database and the leading NoSQL database. Written in C++. NoSQL refers to non-relational databases promises to handle large volumes of structured and unstructured data. Unlike relational databases they NoSQL databases do not require schemas to be defined beforehand. These fits perfectly fine with the agile technologies as each time a new feature is developed, the schema of the database is needed to be changed. The data-structure of non-relational database is not fixed. This allows the insertion of data without a predefined schema. This ultimately proves helpful in making significant changes in applications in real-time, with no interruption in service, leading to faster development, reliable code integration and lessens the amount of time by database administrators. NoSQL also referred as "Not only SQL" to

emphasize that these also support SQL like queries. Mongo DB is a document-oriented database released in 2009. It holds a set of collections; a collection holds a set of documents. A document further is a set of key-value pairs. These documents have a dynamic schema which means that documents in the same collection do not need to have the same set of fields or structure. The key feature for which Mongo DB is used is its flexibility which is understood as that the data is stored in JSON documents as shown in fig 1 below:

### **2.2.2 Key feature of Mongo DB: JSON**

These provides a rich data model that flawlessly maps to native programming language, and the dynamic schema makes it easier to unfold your data model as compared to a system with compelled schema RDBMS. Mongo DB is chosen for evaluation because it can manage small data as well as large data efficiently.

### **2.2.3 REDIS: KEY-VALUE STORE DATABASE**

Redis (Remote Dictionary Server) is an open source, advanced key-value store. The key feature of Redis is that it can store more complex data types and atomic operations like appending to a string; incrementing the value in a hash; pushing into a list; getting the member with highest ranking in a sorted set; computing set union, intersection, difference can be defined against these data types. It is in-memory but persistent on disk which means data does not disappear on system restart. Redis holds the whole dataset in memory. In-memory data set here means that data may be swapped out when not in use for long or not needed in future much. It is also termed as data structure server since keys may contain strings, lists, hashes, sets, and sorted sets.

In the author had explained the key-value implementation of database. He made an approach to persist the objects under race and failure conditions. As Redis is well famous for speed and scalability so it is widely used with cloud storage. As cloud web applications demand fast request and retrieval of data. Whereas relational database could have been proved a bottleneck for this requirement of speed. RDBMS had fixed schema and if they had to be used in future then there may be a lot of space wastage with lots of unused columns. Whereas KVS had been really simple while designing where values may be stored anytime using a unique key. Those values could be used later using that key. Also known as hash addressing. RDBMS have various serious advantages over KVS: such as they are well tested, also have good management tools, apart from this have programming patterns available. The

Key-values on the other hand are fairly newly introduced, each of them is having different approaches and these excel only in a very small segment. Author in has discussed.

## **2.3 Comparison between Document Databases-Mongo DB and Couch DB**

Clarence J. M. Tauro et al. [4] implemented and explained about the comparison between Mongo DB and Couch DB.

### **2.3.1 Mongo DB**

Mongo DB is an open-source document-oriented database written in C++ and is completely schema-free and manages JSON-style documents. It focuses on high-performance providing the developer with a set of features to easily model and query data.

#### **2.3.1.1 Data model**

Mongo DB stores data as BSON objects, which is a binary-encoded serialization of JSON-like documents. It supports all the data types that are part of JSON but also defines new data types, i.e. the Date data type and the Big Data type and also support all the data type that is the part of Jason. 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 is important to note that Mongo DB does not provide the ability to join documents, therefore when referencing documents, any necessary join has to be done on the client-side.

### **2.3.1.2 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.

### **2.3.1.3 Replication model**

Mongo DB provides Master-Slave replication and Replica sets, where data is asynchronously replicated between servers. In either case only one server is used for write operations at a given time, while read operations can be redirected to slave servers. Replica sets are an extension of the popular Master-Slave replication scheme in order to provide automatic failover and automatic recovery of member nodes. A replicate set is a group of servers where at any point in time there is only one master server, but the set is able to elect a new master if the current one goes down. Data is replicated among all the servers from the set. It illustrates the two different replication models. Since Mongo DB has only one single active master at any point in time, strong consistency can be achieved if all the read operations are done on the master. Since replication to the slave servers is done asynchronously and Mongo DB does not provide version concurrency control, reads from the slave servers employ eventual consistency semantics. The ability to read from slave servers is usually desired to achieve load balance, therefore the client is also able to enforce that a certain write has replicated to a certain number of slave servers. This feature helps dealing with important writes, where eventual consistency semantics might not be suitable, while at the same time providing the flexibility to read from slave servers.

## **2.3.2 Apache couch DB**

Is an open-source document-oriented database and is written in Erlang. It complies with the ACID properties, providing serializability.

### **2.3.2.1 Data model**

Data is stored as semi-structured documents in Couch DB. A document is a JSON file which is a collection of named key-value pairs. Values can be numbers, string, Booleans, lists

or dictionaries. Documents are not bound to follow any structure and can be schema-free. Each document Couch DB database is identified by a unique ID (the “-id” field). Couch DB is a simple container of a collection of documents and it does not establish any mandatory relationship between them.

#### **2.3.2.2 Query model**

Couch DB exposes a Restful HTTP API to perform basic CRUD operations on all stored items and it uses the HTTP methods POST, GET, PUT and DELETE to do so. More complex queries can be implemented in the form of views (as was seen before) and the result of these views can also be read using the REST API.

#### **2.3.2.3 Replication model**

Couch DB is a peer-based distributed database system it allows peers to update and change data and then bidirectional synchronizes the changes. Therefore, we can model either master-slave setups (where synchronizations are unidirectional) or master-master setups where changes can happen in either of the nodes and they must be synchronized in a bidirectional way. Each document is assigned a revision id and every time a document is updated, the old version is kept and the updated version is given a different revision id. Whenever a conflict is detected, the winning version is saved as the most recent version and the losing version is also saved in the document’s history. This is done consistently throughout all the nodes so that the exact same choices are made. The application can then choose to handle the conflict by itself (ignoring one version or merging the changes).

#### **2.3.2.4 Consistency model**

Couch DB provides eventual consistency. As multiple masters are allowed, changes need to be propagated to the remaining nodes, and the database does not lock on writes. Therefore, until the changes are propagated from node to node the database remains in an inconsistent state. Still, single master setups (with multiple slaves) are also supported, and in this case strong consistency can be achieved.

### **2.4 Partitioning**

Avinash Lakshman Facebook [5] explained how to do partitioning with the unstructured data. The most NoSQL and NewSQL data stores implement some sort of horizontal partitioning or sharding, which involves storing sets or rows/records into different



segments (or shards) which may be located on different servers. In contrast, vertical partitioning involves storing sets of columns into different segments and distributing them accordingly. The data model is a significant factor in defining strategies for data store partitioning. For example, vertical partitioning segments contain predefined groups of columns; therefore, data stores from the column-family category can provide vertical partitioning in addition to horizontal partitioning.

The two most common horizontal-partitioning strategies are range partitioning and consistent hashing. Range partitioning assigns data to partitions residing in different servers based on ranges of a partition key. A server is responsible for the storage and read/write handling of a specific range of keys. The advantage of this approach is the effective processing of range queries, because adjacent keys often reside in the same node. However, this approach can result in hot spots and load-balancing issues. For example, if the data are processed in the order of their Key values, the processing load will always be concentrated on a single server or a few servers. Another disadvantage is that the mapping of ranges to partitions and nodes must be maintained, usually by a routing server, so that the client can be directed to the correct server. Berkeley DB, Cassandra, HBase, and Mongo DB implement range partitioning as depicted in Table 2.

In consistent hashing, the dataset is represented as a circle or ring. The ring is divided into a number of ranges equal to the number of available nodes, and each node is mapped to a point on the ring. Figure 2 illustrates consistent hashing on an example with four nodes N1 to N4. To determine the node where an object should be placed, the system hashes the object's key and finds its location on the ring. In the example from Figure 2, object is located between nodes N4 and N1. Next, the ring is walked clockwise until the first node is encountered, and the object gets assigned to that node. Accordingly, object from Figure 2 gets assigned to node N1. Consequently, each node is responsible for the ring region between itself and its predecessor; for example, node N1 is responsible for data range 1, node N2 for data range 2, and so on. With consistent hashing, the location of an object can be calculated very fast, and there is no need for a mapping service as in range partitioning. This approach is also efficient in dynamic resizing: if nodes are added to or removed from the ring, only neighboring regions are reassigned to different nodes, and the majority of records remain unaffected [16]. However, consistent hashing negatively impacts range queries because neighboring keys are distributed across a number of different nodes. Voldemort, Riak, Cassandra, Dynamo DB, Couch DB, VoltDB, and Clustrix implement consistent hashing.

## 2.5 The CAP theorem

Abhinav Tiwari et al.[6] explains the CAP Theorem, in order to store and process massive datasets, a common 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 others strategies to implement high-available and scalable solutions that can be leveraged in cloud environments. Nevertheless, these solutions and others 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.

## 2.6 Querying

Lakshman A, Malik P et al.[7] had explained about the querying that similar to the selection of a data model, the querying capabilities of data stores play an important role when choosing among them for a particular scenario. Different data stores offer different APIs and

interfaces to interact with them. This is directly dependent upon the data model that a particular data store possesses. For example, a key value store cannot provide querying based on the contents of the values, because these values are opaque to the data store. On the other hand, a document store can do so because its data model provides the capability to index and query the document contents. Another important query-related feature of NoSQL and NewSQL data stores is their level of support for Map Reduce. Map Reduce, which was first developed by Google, is a programming model and an associated implementation for processing large datasets. It has now become a widely accepted approach for performing distributed data processing on a cluster of computers. Because one of the primary goals of NoSQL data stores is to scale over a large number of computers, Map Reduce has been adopted by most of them. Similarly, SQL-like querying has been a preferred choice because of its widespread use over the past few decades, and it has now also been adopted in the NoSQL world. Therefore, some of the prominent NoSQL data stores like Mongo DB offer a SQL-like query language or similar variants such as CQL offered by Cassandra and SparQL by Neo4j and Allegro Graph. As for the NewSQL category, the use of SQL as a query language is one of its defining characteristics, but the level of SQL support varies considerably. Clustrix and NuoDB are the most SQL-compliant of the solutions analyzed, having only minor incompatibilities with the standard.

On the other hand, Corbett et al. state that the Google Spanner query language “looks like SQL with some extensions to support protocol-buffer-value fields”, but they do not provide details about the language. Finally, VoltDB has a larger number of restrictions in place: it is not possible to use the having clause, tables cannot join with themselves, and all joined tables must be partitioned over the same value. It is also worth mentioning that the recommended way of interacting with VoltDB is through Stored Procedures. These procedures are written in Java, where programming logic and SQL statements are interspersed. On the other hand, a command-line interface (CLI) is usually the simplest and most common interface that a data store can provide for interaction with itself and is therefore offered by almost all NoSQL and NewSQL products. In addition, most of these products offer API support for multiple languages. Moreover, a REST-based API has been very popular in the world of Web-based applications because of its simplicity. Consequently, in the NoSQL world, a REST-based interface is provided by most solutions, either directly or indirectly through third-party APIs. Table 1 provides a detailed view of the different APIs support provided by the most prominent NoSQL and NewSQL solutions along with other querying capabilities offered.

## 2.7 Replication

Ford D, Labelle F, Popovici F [8]. explains about the replication. In addition to increasing read/write scalability, replication also improves system reliability, fault tolerance, and durability. Two main approaches to replication can be distinguished: master–slave and multi-master replication. In master–slave replication, shown in Figure 3.a, a single node is designated as a master and is the only node that processes write requests. Changes are propagated from the master to the slave nodes. Examples of data stores with master–slave replication are Redis, Berkeley DB, and HBase. In multi-master replication, illustrated in Figure 3b, multiple nodes can process write requests, which are then propagated to the remaining nodes? Whereas in master–slave replication the propagation direction is always from master to slaves, in multi-master replication, propagation happens in different directions. Couch DB and Couch base Server are examples of multi-master data stores. Three other data stores, Voldemort, Riak, and Cassandra, support master less replication, which is similar to multi-master replication as multiple nodes accept write requests, but as highlighted by the term master less, all nodes play the same role in the replication system. Note that all three of the data stores with master less replication use consistent hashing as a partitioning strategy. The strategy for placing replicas is closely related to node position on the partitioning ring, as shown Table 2. NewSQL replication schemes can be considered as multi-master or master less schemes because any node can receive update statements.

In VoltDB and Clustrix, a transaction/session manager receives the updates, which are forwarded to all replicas and executed in parallel. On the other hand, Google Spanner uses the Paxos state machine algorithm to guarantee that a sequence of commands will be executed in the same order in all the replica nodes. Note that Paxos is a distributed algorithm without central arbitration, which differs significantly from the other solutions. Finally, in NuoDB, the table rows are represented as in-memory distributed objects which communicate asynchronously to replicate their state changes. The choice of replication model impacts the ability of the data store to scale read and write requests. Master– slave replication is generally useful for scaling read requests because it allows the many slaves to accept read requests – examples are Berkeley DB and Mongo DB. However, some data stores such as HBase do not permit read requests on the slave nodes. In this case, replication is used solely for failover and disaster recovery. In addition, master–slave data stores do not scale write requests because the master is the only node that processes write requests. An interesting exception is the Neo4J database, which is able to handle write requests on the slave nodes also. In this case, write requests are synchronously propagated from slaves to master and therefore are slower

than write requests to master. Finally, multi-master and master less replication systems are usually capable of scaling read and write requests because all nodes can handle both requests. Another replication characteristic with a great impact on data stores throughput is how write operations are propagated among nodes. Synchronization of replicas can be synchronous or asynchronous. In synchronous or eager replication, changes are propagated to replicas before the success of the write operation is acknowledged to the client. This means that synchronous replication introduces latencies because the write operation is completed only after change propagation. This approach is rarely used in NoSQL because it can result in large delays in the case of temporary loss or degradation of the connection. In asynchronous or lazy replication, the success of a write operation is acknowledged before the change has been propagated to replica nodes. This enables replication over large distances, but it may result in nodes containing inconsistent copies of data. However, performance can be greatly improved over synchronous replication. As illustrated in Table 2, the majority of the data stores studied use asynchronous replication. Typically, NoSQL solutions use this approach to achieve the desired performance, Yet Couch DB uses it to achieve off-line operation. In Couch DB, multiple replicas can have their own copies of the same data, modify them, and then synchronize these changes at a later time.

## **CHAPTER 3**

### **PROPOSED SYSTEM**

#### **3.1. SCOPE OF THE PROJECT**

1. The previous works have been dealt with using the clustering method to store the data's.
2. In this project going to use Mongo DB to store the large amount of unstructured data.
3. Mongo DB is very consistent to store the unstructured data using a sharding technology.

Change is coming fast in enterprise storage, placing greater-than-ever stress on IT environments. It is not just the increasing flood of data pouring into data centers: an estimated 40 zeta bytes by 2020, representing 50-fold growth from 2010.<sup>1</sup> It is the fact that by 2015, 77% of that data will be unstructured, including audio, video, emails, digital images, social networking data and more. The looming question is: How should enterprises handle this unprecedented growth? The traditional method of managing file, block and object storage independently is expensive and complex, especially given the cascade of unstructured data pouring into enterprise data centers. That's why enterprises are beginning to replace their separate file and block solutions with unified storage systems that can manage structured and unstructured data on a common platform. But, not all unified storage systems are created equal. To make the transition to unified storage successful, and worth the investment and risk, the solution must be designed for the enterprise. It also needs to combine virtualized storage, intelligent file tiring, no disruptive migration, a unified management interface and a near 100% availability guarantee.

#### **3.2. FEASIBILITY STUDY**

##### **3.2.1 SCOPE 1**

###### **(1). Economical feasibility**

There is no need of any cost for implementing the technical flaw 1, if the system is available and the required software's are available. It is feasible to continue this work without incurring any cost. Software cost and the System cost is the only thing needed to do this work.

**(2). Technical feasibility**

Technically, it will require only Python and Mongo DB for implementing the technical flaw

1. Python coding is necessary for the project to implement the ideas and algorithms.

**(3). Operational feasibility**

Operational at any time after the implementation.

**(4). Schedule feasibility**

The implementation of technical flaw 1 identified can be completed within the given period of time.

**3.2.2 SCOPE 2**

**(1). Economical feasibility**

It is feasible in the economical wise to implement the technical flaw 2 with getting the data's from the source and implement the coding and store the data into the Mongo DB.

**(2). Technical feasibility**

Technically, it will require only Python and Mongo DB for implementing the technical flaw

2. Some of the parser is used to converting the data's into the text format.

**(3). Operational feasibility**

Operational at any time after the implementation.

**(4). Schedule feasibility**

The implementation of technical flaw 2 identified can be completed within the given period of time.

**3.2.3 SCOPE 3**

**(1). Economical feasibility**

It is feasible in the economical wise to implement the technical flaw 2 with getting the data's from the source and implement the coding and store the data into the Mongo DB.

## **(2). Technical feasibility**

Technically, it will require only Python and Mongo DB for implementing the technical flaw 2. Some of the parser is used to converting the data's into the text format.

## **(3). Operational feasibility**

Operational at any time after the implementation.

## **(4). Schedule feasibility**

The implementation of technical flaw 2 identified can be completed within the given period of time.

### **3.3 SPECIFIC OBJECTIVES OF THE PROPOSED SYSTEM**

The main objective of the proposed system focused to give the intended task of the user. It takes the large amount of unstructured data and it will be store into the Mongo DB.

1. To extract the unstructured data from the source.
2. To managing the queries to the categorization of the data.
3. For the data storage first create the shard cluster environment and after that store the large amount of unstructured data into the MongoDB database.

The relevant queries to be suggested are mainly focused to provide the suggestion of the particular query given by the user. The query suggestion gives the set of relevant queries to be given by the user. The queries and the needs will vary depending upon the user. The series of query patterns and compared the query given by the two users using the keyword query graph and Bayesian network.

The prediction of user behavior is another one objective which predicts the user behavior by using the URL clicked by the user and the reading of the user towards particular webpage. A user task can be seen as a set of meaningfully relevant search query trails within single session. A task may be simply represented by all such beginning search queries from the particular user query trails.



### 3.4 ARCHITECTURE DIAGRAM

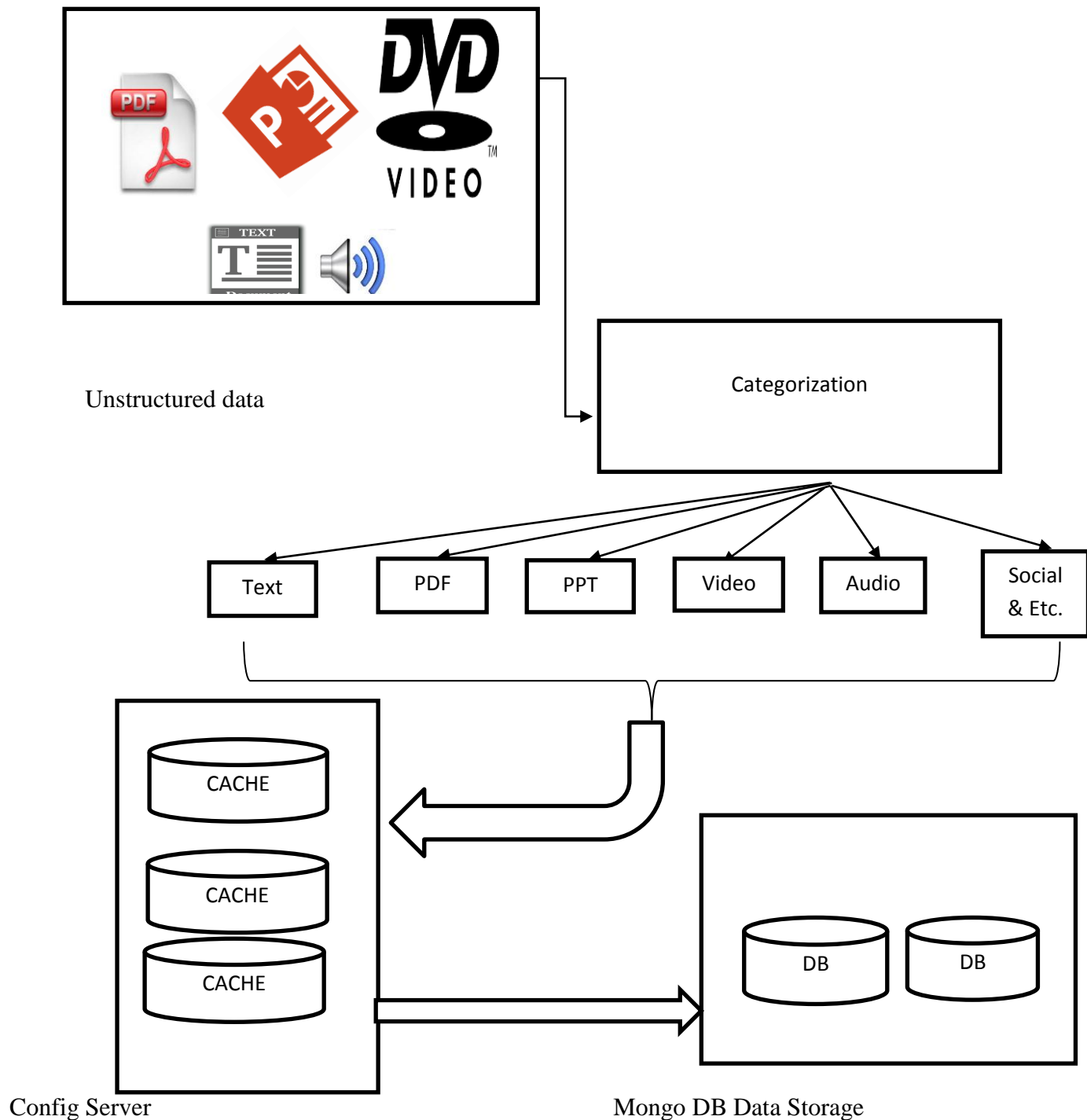


Fig.3.1 Architecture diagram

In fig 3.1 explains the task is very simple get the unstructured data from the source and it been analyzed by the system provider. And then the unstructured data it to been goes to the categorization part for categorize the data into the particular type of the data. All the data's are goes to the categorization part the all data's should be categorized by its specific

format of the data. In this step we are using the some of the parser to convert the data into its original format. Example apache PDFBOX is used to extract the pdf file and stores into the same format.

After the categorization method is completed all data's are collected into the temporary storage of the framework My Store. My Store is the temporary memory storage system it will be erased once the data that it moved from the My Store Framework. Whenever the framework it should containing the temporary memory files and it will be store the data for some times.

### **3.5 DATA FLOW DIAGRAM**

A Data Flow Diagram (DFD) is a graphical representation of the data flow through an intelligent information system and modeling the process aspects.

#### **3.5.1 DFD LEVEL 0**

The DFD level 0 shows the input, system and the output of this project. As this project using the unstructured data it to be categorizes and stored into the Mongo DB. In Figure 3.5, it explains the simple way of the project explanation. Get the source from the origin and send it into the categorization field after that the data will be stored into the Mongo DB database systems. In the categorization method we will using the many parser for converting the data's into the text format.

But, not all unified storage systems are created equal. To make the transition to unified storage successful, and worth the investment and risk, the solution must be designed for the enterprise. It also needs to combine virtualized storage, intelligent file tiring, no disruptive migration, a unified management interface and a near 100% availability guarantee.

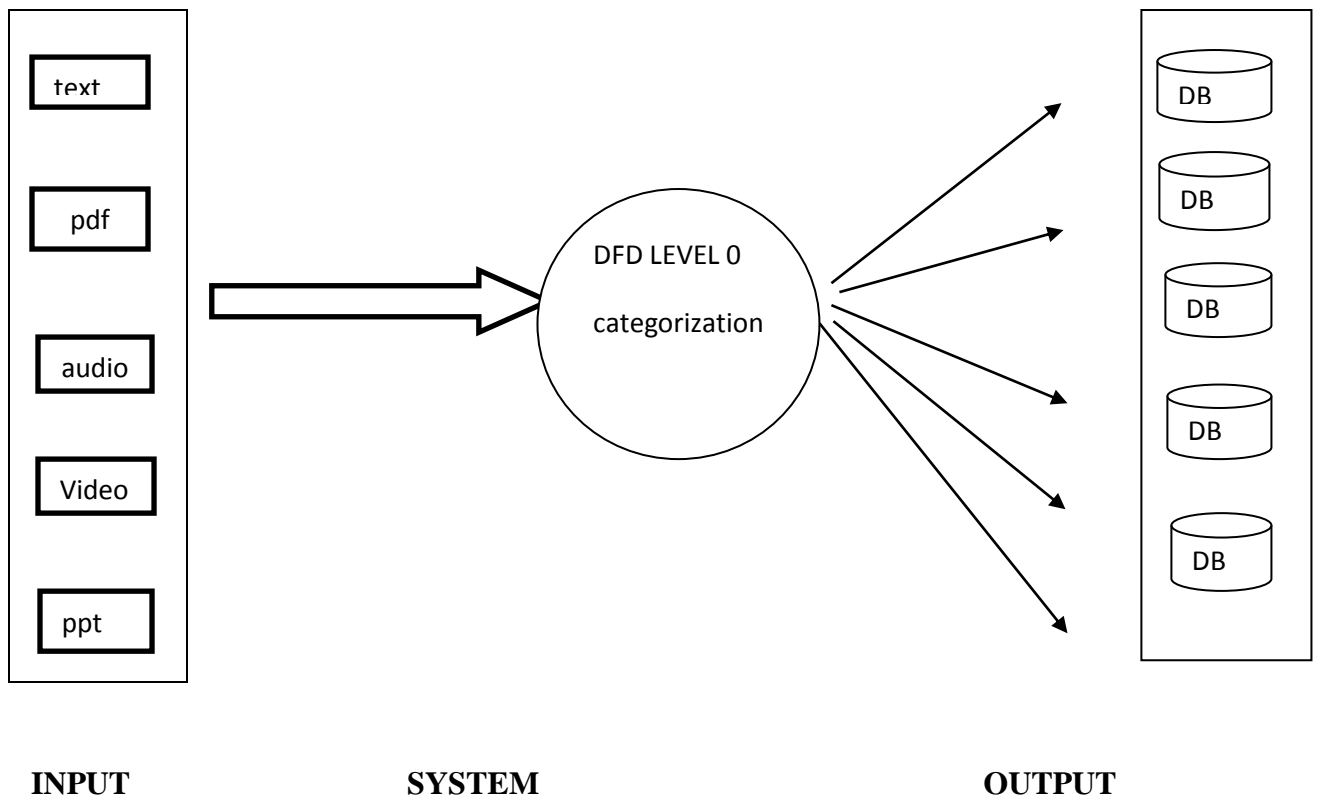


Fig. 3.2 DFD level 0

### 3.5.2 DFD Level 1:

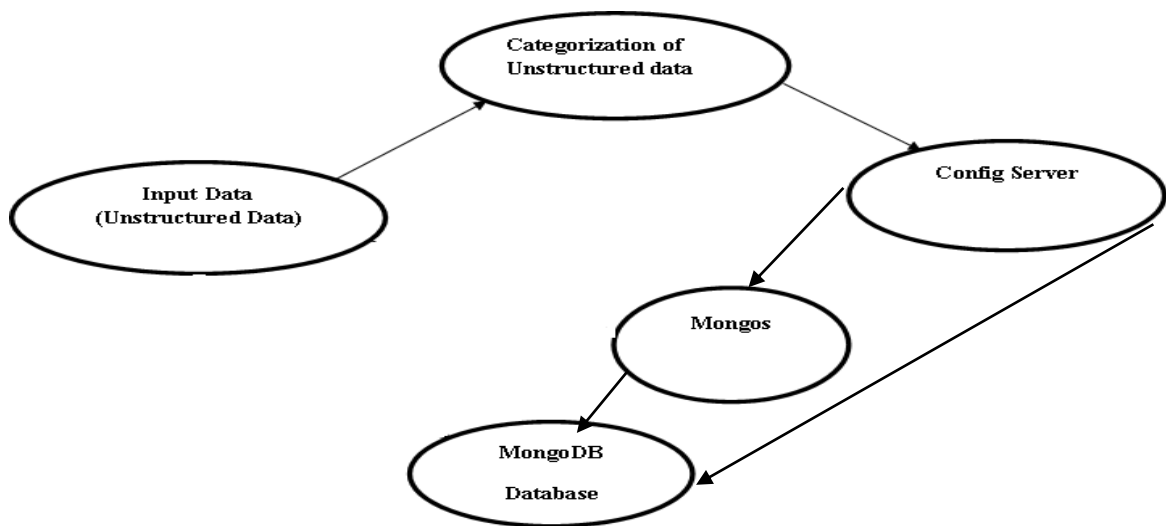


Fig. 3.3 DFD level 1

### 3.6 PROCESS DESCRIPTION

The proposed work consists of five processes that are config servers, config server availability, consistent hashing algorithm, solution for the consistent hashing algorithm, implementation in the process description.

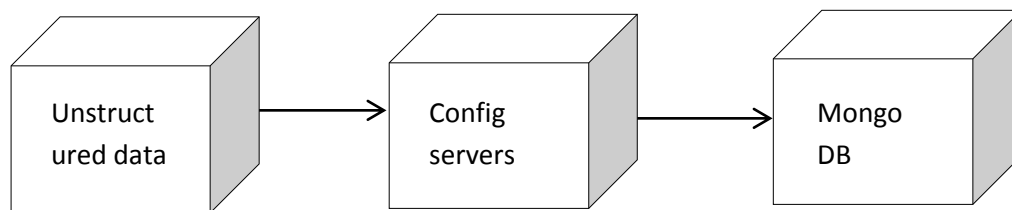


Fig. 3.4 Process description

#### 3.6.1 CONFIG SERVER

Config servers are special mongod instances that store the metadata for a sharded cluster. A production sharded cluster has exactly three config servers. All config servers must be available to deploy a sharded cluster or to make any changes to cluster metadata. Config servers do not run as replica sets. For testing purposes you may deploy a cluster with a single config server. But to ensure redundancy and safety in production, you should always use three.

#### 3.6.2 CONFIG SERVER AVAILABILITY

If one or two config servers become unavailable, the cluster's metadata becomes read only. You can still read and write data from the shards, but no chunk migrations or splits will occur until all three servers are available. If all three config servers are unavailable, you can still use the cluster if you do not restart the mongos instances until after the config servers are accessible again. If you restart the mongos instances before the config servers are available, the mongos will be unable to route reads and writes. Clusters become inoperable without the cluster metadata. To ensure that the config servers remain available and intact, backups of config servers are critical. The data on the config server is small compared to the data stored

in a cluster, and the config server has a relatively low activity load. These properties facilitate finding a window to back up the config servers.

If the name or address that a sharded cluster uses to connect to a config server changes, you must restart every mongod and mongos instance in the sharded cluster. Avoid downtime by using CNAMEs to identify config servers within the MongoDB deployment.

### 3.6.3 CONSISTENT HASHING ALGORITHM

Consistent hashing is a very simple solution to a common problem: how can you find a server in a distributed system to store or retrieve a value identified by a key, while at the same time being able to cope with server failures and network partitions? Simply finding a server for value is easy; just number your set of  $s$  servers from 0 to  $s - 1$ . When you want to store or retrieve a value, hash the values key modulo  $s$ , and that gives you the server. The problem comes when servers fail or become unreachable through a network partition. At that point, the servers no longer fill the hash space, so the only option is to invalidate the caches on all servers, renumber them, and start again. Given that, in a system with hundreds or thousands of servers, failures are commonplace, this solution is not feasible.

### 3.6.4 SOLUTION FOR THE CONSISTENT HASHING ALGORITHM

In consistent hashing, the servers, as well as the keys, are hashed, and it is by this hash that they are looked up. The hash space is large, and is treated as if it wraps around to form a circle - hence hash ring. The process of creating a hash for each server is equivalent to placing it at a point on the circumference of this circle. When a key needs to be looked up, it is hashed, which again corresponds to a point on the circle. In order to find its server, one then simply moves round the circle clockwise from this point until the next server is found. If no server is found from that point to end of the hash space, the first server is used - this is the "wrapping round" that makes the hash space circular.

The only remaining problem is that in practice hashing algorithms are likely to result in clusters of servers on the ring (or, to be more precise, some servers with a disproportionately large space before them), and this will result in greater load on the first server in the cluster and less on the remainder. This can be ameliorated by adding each server

to the ring a number of times in different places. This is achieved by having a replica count, which applies to all servers in the ring, and when adding a server, looping from 0 to the count - 1, and hashing a string made from both the server and the loop variable to produce the position. This has the effect of distributing the servers more evenly over the ring. Note that this has nothing to do with server replication; each of the replicas represents the same physical server, and replication of data between servers is an entirely unrelated issue.

### **3.7 HARDWARE AND SOFTWARE REQUIREMENTS**

#### **Hardware Requirements:**

System	: Quad core processor 2.80GHz.
Hard Disk	: 500 GB.
RAM	: 4 GB.

#### **Software Requirements:**

Operating system	: Windows 7
Coding Language	: JAVA/C++
Data Base	: Mongo DB

## CHAPTER 4

### SYSTEM IMPLEMENTATION

#### 4.1 INSERTING THE SIMPLE TEXT FILE INTO MONGODB

Inserting the simple text file into Mongo DB is a simple one you should write the code for that storing the text file it will automatically store into the Mongo DB. The storing data it be stored in the format of JSON. You don't need to convert the text file into the JSON format it automatically seems to be converted to the JSON format if you are using the Python Mongo DB Driver.

Simple example for storing the simple text file into the Mongo DB,

#### INSERTING THE TEXT FILE INTO MONGODB:

Using the simple java program to import and save the text files into the MongoDB using consistent hashing algorithm.

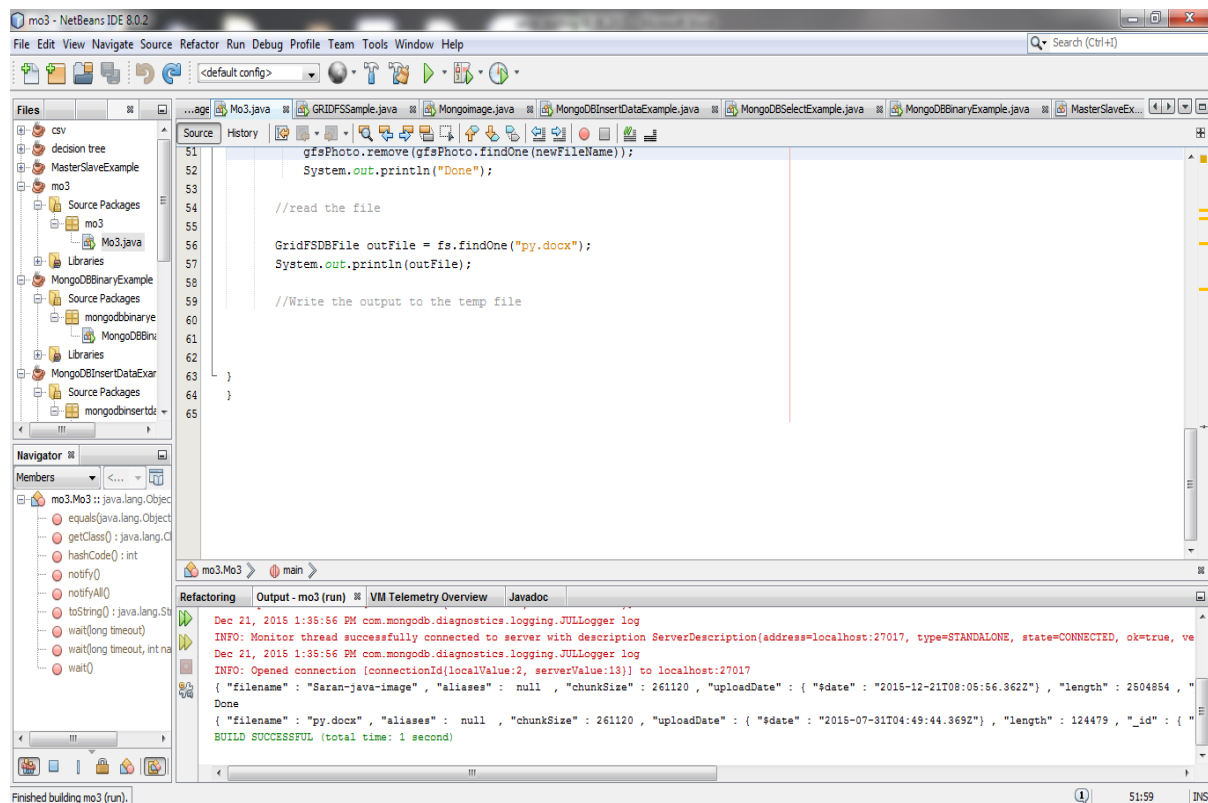


Fig. 4.1 inserting the text file into the MongoDB

## 4.2 INSERTING VIDEO FILE INTO MONGODB

In this work the program for insert and store the video files should wrote and successfully run in the IDE and the video files are stored successfully.

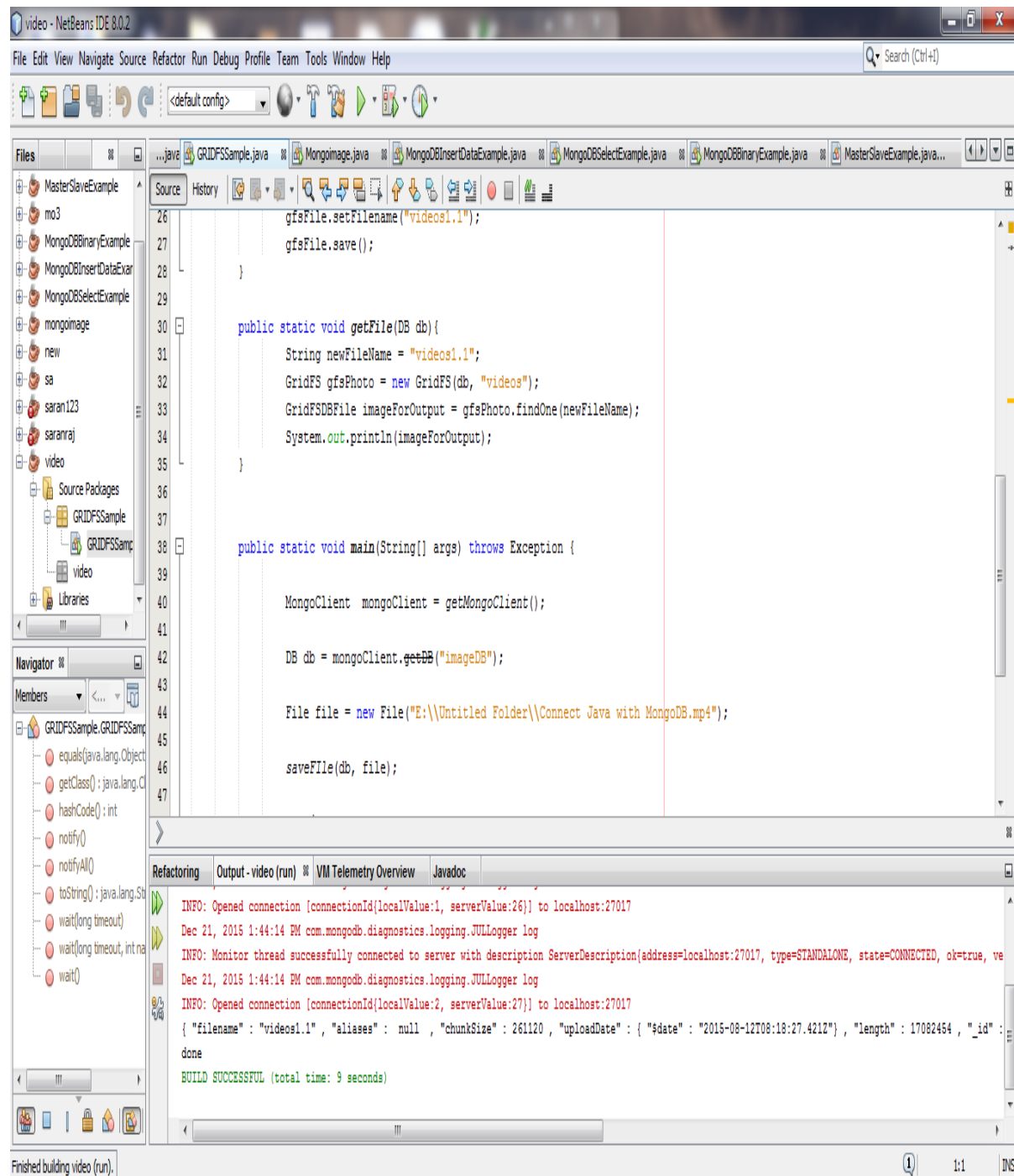


Fig. 4.2 Inserting the video files into the MongoDB



### 4.3 INSERTING IMAGE FILE INTO MONGODB

The image files are successfully stored using the java coding .

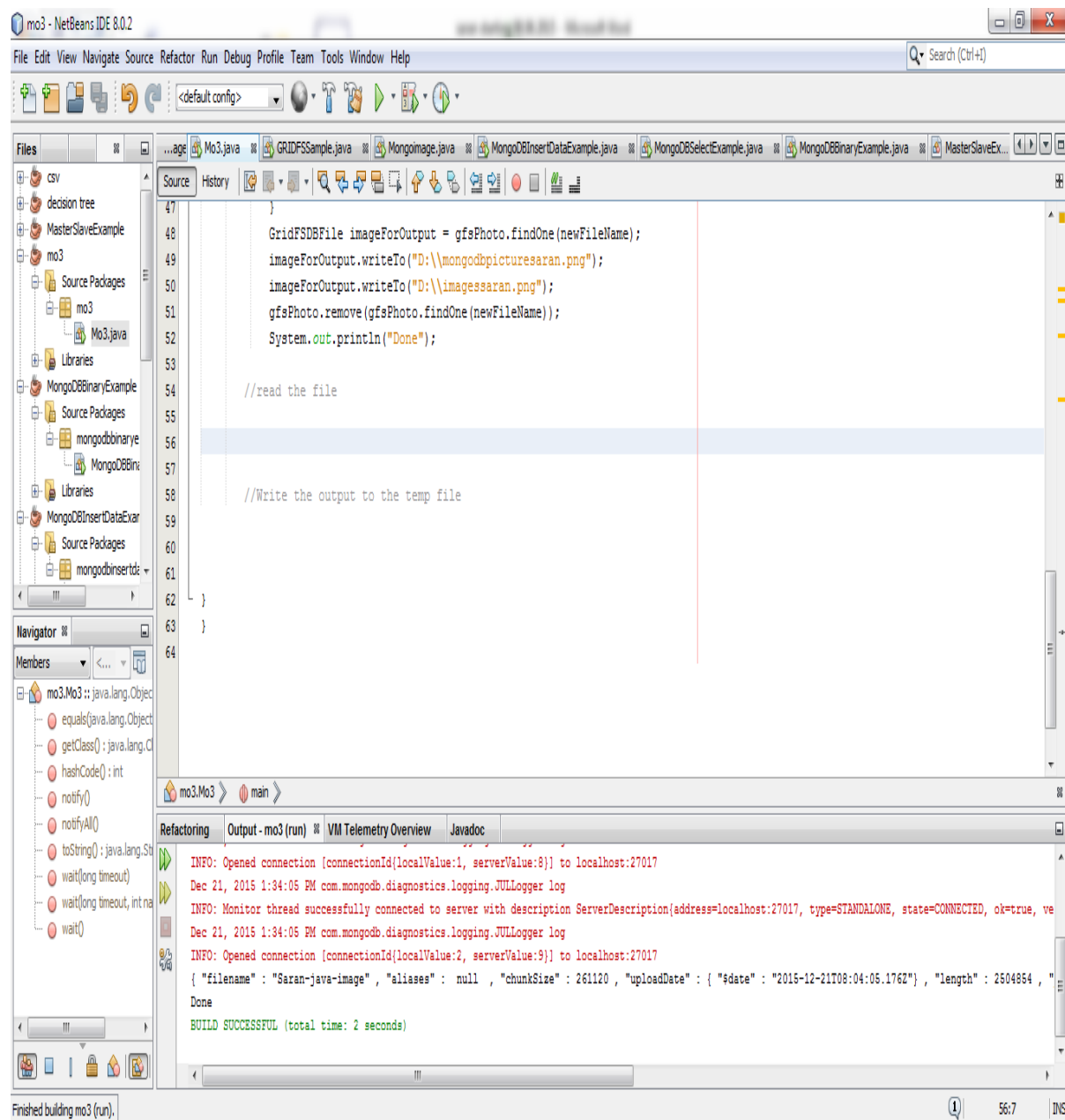


Fig. 4.3 Inserting image file into MongoDB

First we should create the database for the image database and we should mention the database name in the program.

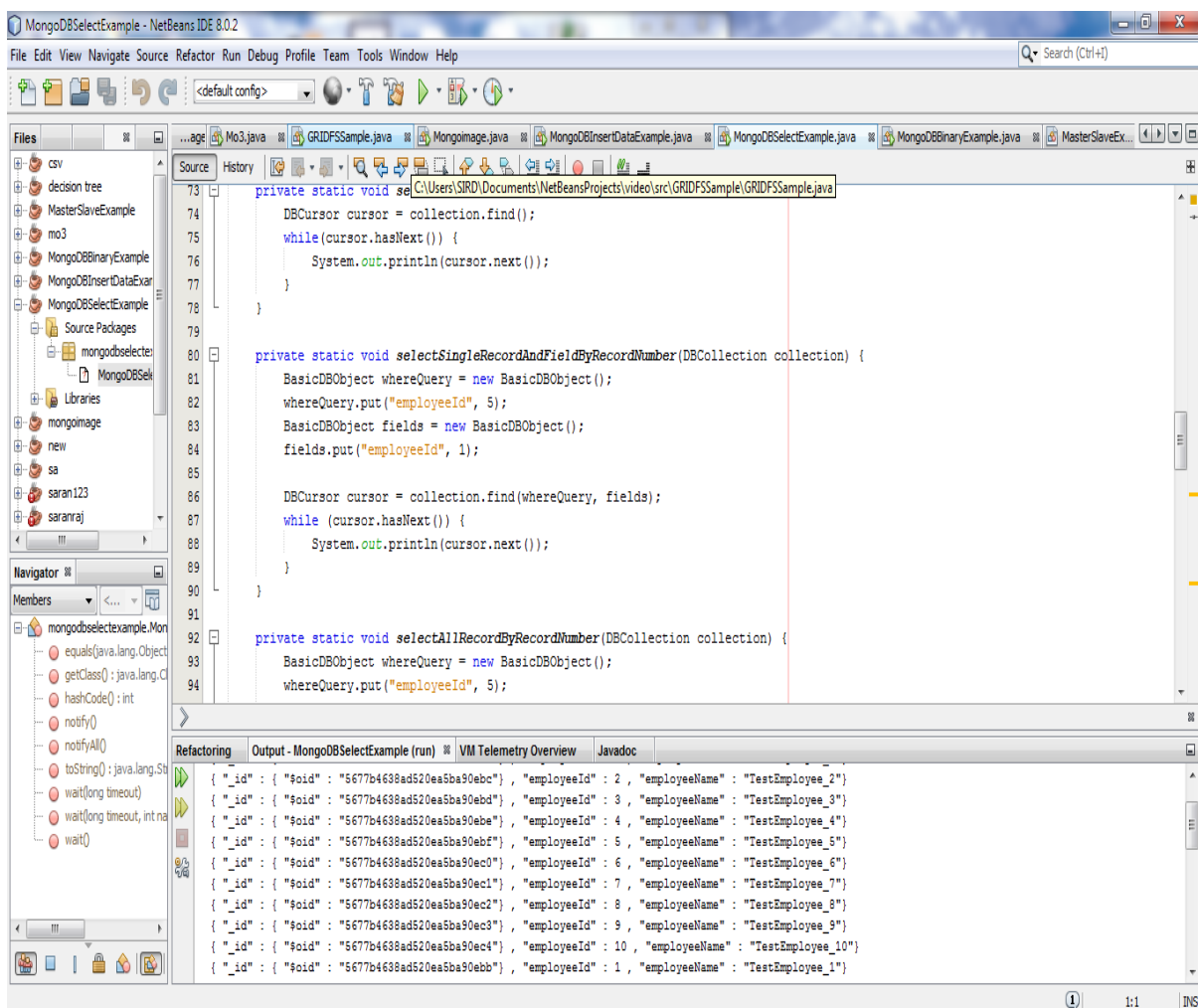
## CHAPTER 5

### RESULT AND ANALYSIS

#### 5.1 SAMPLE RESULTS

##### INSERTING LARGE AMOUNT OF UNSTRUCTURED DATA INTO MONGODB

The large amount of unstructured data available in Scientific Information Resource Division (SIRD) of Indira Gandhi Centre for Atomic Research are used in this program to storing into the MongoDB using the consistent hashing algorithm and it had been successfully store into the MongoDB.



The screenshot displays the NetBeans IDE interface. The left sidebar shows a project named 'mongodselectexample' with a 'Members' list containing various Java classes and methods. The main editor window shows the source code for 'MongoDBSelectExample.java'. The code includes three methods: 'select', 'selectSingleRecordAndFieldByRecordNumber', and 'selectAllRecordByRecordNumber'. The 'select' method uses a cursor to iterate through a collection and print each record. The 'selectSingleRecordAndFieldByRecordNumber' method uses a query to find a specific record and print its fields. The 'selectAllRecordByRecordNumber' method uses a query to find all records and print them. The bottom of the IDE shows the 'Output' window, which displays the results of the 'select' method, showing a list of records with their IDs, employee IDs, and employee names.

```
private static void select(DBCollection collection) {
    DBCursor cursor = collection.find();
    while(cursor.hasNext()) {
        System.out.println(cursor.next());
    }
}

private static void selectSingleRecordAndFieldByRecordNumber(DBCollection collection) {
    BasicDBObject whereQuery = new BasicDBObject();
    whereQuery.put("employeeId", 5);
    BasicDBObject fields = new BasicDBObject();
    fields.put("employeeId", 1);

    DBCursor cursor = collection.find(whereQuery, fields);
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
}

private static void selectAllRecordByRecordNumber(DBCollection collection) {
    BasicDBObject whereQuery = new BasicDBObject();
    whereQuery.put("employeeId", 5);
}
```

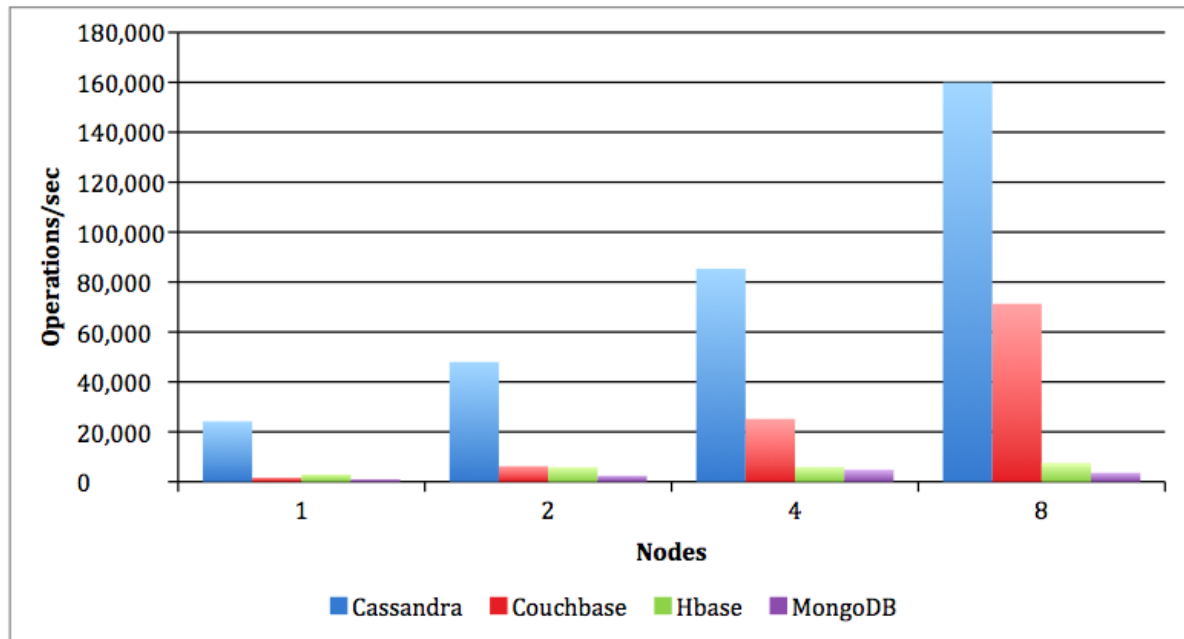
Output - MongoDBSelectExample (run)

```
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ebc" }, "employeeId" : 2, "employeeName" : "TestEmployee_2" }
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ebd" }, "employeeId" : 3, "employeeName" : "TestEmployee_3" }
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ebe" }, "employeeId" : 4, "employeeName" : "TestEmployee_4" }
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ebf" }, "employeeId" : 5, "employeeName" : "TestEmployee_5" }
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ec0" }, "employeeId" : 6, "employeeName" : "TestEmployee_6" }
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ec1" }, "employeeId" : 7, "employeeName" : "TestEmployee_7" }
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ec2" }, "employeeId" : 8, "employeeName" : "TestEmployee_8" }
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ec3" }, "employeeId" : 9, "employeeName" : "TestEmployee_9" }
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ec4" }, "employeeId" : 10, "employeeName" : "TestEmployee_10" }
{ "_id" : { "$oid" : "5677b4638ad520ea5ba90ebb" }, "employeeId" : 1, "employeeName" : "TestEmployee_1" }
```



## Insert-mostly Workload

Note that test runs in AWS against 16 and 32 nodes were not included due to a few technical issues that created inconsistent results.



Nodes	Cassandra	Couchbase	HBase	MongoDB
1	24,163.62	1,624.63	2,904.64	984.89
2	47,974.09	6,241.02	5,837.78	2,446.06
4	85,324.69	25,185.76	5,888.09	4,853.85
8	159,945.39	71,307.78	7,592.18	3,641.98

## **CHAPTER 6**

### **CONCLUSION AND FUTURE WORKS**

#### **5.1 CONCLUSION**

In this work, we successfully stored the large amount of unstructured data into the Mongo DB using the consistent hashing algorithm. In this work, the various types of unstructured data available in Scientific Information Resource Division (SIRD) of Indira Gandhi Centre for Atomic Research (IGCAR) are represented in MongoDB using the consistent hashing algorithm. This consistent hashing algorithm is mainly used to store the with low data loss and .Config servers dramatically improves the ability of handling failures and the availability, scalability of the system. We handled with the 100GB of data that is available in SIRD, IGCAR.

#### **5.2 FUTURED WORK**

In the future work of this system will create the cloud environment and store the large amount of data's like 10TB, 20TB etc. using the GridFS technology. GridFS is the MongoDB specification for storing and retrieving large files such as images, audio files, video files, etc. It is kind of a file system to store files but its data is stored within MongoDB collections.

## REFERENCES:

1. Gartner Inc. <http://www.gartner.com/>
2. Abadi J (2009) Data management in the Cloud: limitations and opportunities. *IEEE Data Eng Bull* 32(1):3–12
3. Lakshman A, Malik P (2009) Cassandra—a decentralized structured storage system. In: *Proceedings of the 3rd ACM SIGOPS international workshop on large scale distributed systems and middleware (LADIS'09)*. ACM, New York, pp 35–40
4. 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
5. Cassandra - A Decentralized Structured Storage System In: *Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles*. ACM, New York, pp 205–220
6. Abhinav Tiwari (2009) Data management in the Cloud: limitations and opportunities. *IEEE Data Eng Bull* 32(1):3–12. *ACM Trans Comput Syst* 26(2):1–26
7. Lakshman A, Malik P (2009) Cassandra: structured storage system on a p2p network. In: *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, New York, pp 5–5
8. Ford D, Labelle F, Popovici F (2010) Availability in globally distributed storage systems. In: *Proceedings of USENIX conference on operating system design and implementation (OSDI'10)*. USENIX, Berkeley, pp 1–7
9. Banker K (2011) *MongoDB in action*. Manning Press, USA
10. Pritchett D (2008) BASE: an acid alternative. *ACM Queue* 6(3):48–55
11. Jim G (1981) The transaction concept: virtues and limitations. In: *Proceedings of the 7th international conference on very large databases (VLDB)*. IEEE, New York, pp 144–154
12. Vogels W (2009) Eventually consistent. *Commun ACM* 52(1):40–44

## APPENDICES

### APPENDIX A

#### SOURCE CODE:

1. Image file storage coding:

```
package mo3;

import java.io.File;
import java.io.IOException;
import java.net.UnknownHostException;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.Mongo;
import com.mongodb.MongoException;
import com.mongodb.gridfs.GridFS;
import com.mongodb.gridfs.GridFSDBFile;
import com.mongodb.gridfs.GridFSInputFile;

public class Mo3 {

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

        // connect to MongoDB
        Mongo mongo = new Mongo("localhost", 27017);
        DB db = mongo.getDB("saran");
        GridFS fs = new GridFS(db);
        DBCollection collection = db.getCollection("dummyColl");
        String newFileName = "Saran-java-image";
        File imageFile = new File("D:\\mongodbpicture.png");
        File install = new File("D:\\MongoDB\\sara\\py.docx");
        GridFSInputFile inFile = fs.createFile(install);
        inFile.save();
        GridFS gfsPhoto = new GridFS(db, "photo");
        GridFSInputFile gfsFile = gfsPhoto.createFile(imageFile);
        gfsFile.setFilename(newFileName);
        gfsFile.save();
        DBCursor cursor = gfsPhoto.getFileList();
```

```

while (cursor.hasNext()){
    System.out.println(cursor.next());
}
GridFSDBFile imageForOutput = gfsPhoto.findOne(newFileName);
imageForOutput.writeTo("D:\\mongodbpicturesaran.png");
imageForOutput.writeTo("D:\\imagesaran.png");
gfsPhoto.remove(gfsPhoto.findOne(newFileName));
System.out.println("Done");
GridFSDBFile outFile = fs.findOne("py.docx");
System.out.println(outFile)
}
}

```

## 2. Video file storing

```

package GRIDFSSample;
import java.io.File;
import com.mongodb.DB;
import com.mongodb.MongoClient;
import com.mongodb.gridfs.GridFS;
import com.mongodb.gridfs.GridFSDBFile;
import com.mongodb.gridfs.GridFSInputFile;

public class GRIDFSSample {
    public static MongoClient getMongoClient() {
        MongoClient mongoClient = null;
        try {
            mongoClient = new MongoClient("localhost", 27017);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return mongoClient;
    }

    public static void saveFile(DB db, File file)throws Exception{

```



```

        GridFS gridfs = new GridFS(db, "videos");
        GridFSInputFile gfsFile = gridfs.createFile(file);
        gfsFile.setFilename("videos1.1");
        gfsFile.save();
    }

    public static void getFile(DB db){
        String newFileName = "videos1.1";
        GridFS gfsPhoto = new GridFS(db, "videos");
        GridFSDBFile imageForOutput = gfsPhoto.findOne(newFileName);
        System.out.println(imageForOutput);
    }

    public static void main(String[] args) throws Exception {
        MongoClient mongoClient = getMongoClient();
        DB db = mongoClient.getDB("imageDB");
        File file = new File("E:\\Untitled Folder\\Connect Java with MongoDB.mp4");
        saveFile(db, file);
        getFile(db);
        System.out.println("done");
    }
}

```

### 3. Document storage

```

package mongodbsselectexample;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.DBObject;
import com.mongodb.MongoClient;
import com.mongodb.WriteResult;
public class MongoDBSelectExample {

```

```

private static void setUpTestData(DBCollection collection){
    for (int i=1; i <= 10; i++) {
        collection.insert(new BasicDBObject().append("employeeId",
i).append("employeeName", "TestEmployee_"+i));
    }
}

public static void main(String[] args) throws UnknownHostException
{
    MongoClient mongo = new MongoClient("localhost", 27017);
    DB db = mongo.getDB("howtodoinjava");
    DBCollection collection = db.getCollection("users");
    WriteResult result = collection.remove(new BasicDBObject());
    System.out.println(result.toString());
    setUpTestData(collection);
    selectAllRecordsFromACollection(collection);
    selectFirstRecordInCollection(collection);
    selectSingleRecordAndFieldByRecordNumber(collection);
    selectAllRecordByRecordNumber(collection);
    in_Example(collection);
    lessThan_GreaterThan_Example(collection);
    negation_Example(collection);
    andLogicalComparison_Example(collection);
    regex_Example(collection);
}

private static void selectFirstRecordInCollection(DBCollection collection) {
    DBObject dbObject = collection.findOne();
    System.out.println(dbObject);
}

private static void selectAllRecordsFromACollection(DBCollection collection) {
    DBCursor cursor = collection.find();
    while(cursor.hasNext()) {
        System.out.println(cursor.next());
    }
}
}

```

```

private static void selectSingleRecordAndFieldByRecordNumber(DBCollection
collection) {
    BasicDBObject whereQuery = new BasicDBObject();
    whereQuery.put("employeeId", 5);
    BasicDBObject fields = new BasicDBObject();
    fields.put("employeeId", 1);
    DBCursor cursor = collection.find(whereQuery, fields);
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    }
}

```

```

private static void selectAllRecordByRecordNumber(DBCollection collection) {
    BasicDBObject whereQuery = new BasicDBObject();
    whereQuery.put("employeeId", 5);
    DBCursor cursor = collection.find(whereQuery);
    while(cursor.hasNext()) {
        System.out.println(cursor.next());
    }
}

```

```

private static void in_Example(DBCollection collection) {
    BasicDBObject inQuery = new BasicDBObject();
    List<Integer> list = new ArrayList<Integer>();
    list.add(2);
    list.add(4);
    list.add(5);
    inQuery.put("employeeId", new BasicDBObject("$in", list));
    DBCursor cursor = collection.find(inQuery);
    while(cursor.hasNext()) {
        System.out.println(cursor.next());
    }
}

```

```

private static void lessThan_GreaterThan_Example(
    DBCollection collection) {

```

```

BasicDBObject gtQuery = new BasicDBObject();
gtQuery.put("employeeId", new BasicDBObject("$gt", 2).append("$lt", 5));
DBCursor cursor = collection.find(gtQuery);
while(cursor.hasNext()) {
    System.out.println(cursor.next());
} }

private static void negation_Example(DBCollection collection) {
    BasicDBObject neQuery = new BasicDBObject();
    neQuery.put("employeeId", new BasicDBObject("$ne", 4));
    DBCursor cursor = collection.find(neQuery);
    while(cursor.hasNext()) {
        System.out.println(cursor.next());
    } }

private static void andLogicalComparison_Example(DBCollection collection) {
    BasicDBObject andQuery = new BasicDBObject();
    List<BasicDBObject> obj = new ArrayList<BasicDBObject>();
    obj.add(new BasicDBObject("employeeId", 2));
    obj.add(new BasicDBObject("employeeName", "TestEmployee_2"));
    andQuery.put("$and", obj);
    System.out.println(andQuery.toString());
    DBCursor cursor = collection.find(andQuery);
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    } }

private static void regex_Example(DBCollection collection) {
    BasicDBObject regexQuery = new BasicDBObject();
    regexQuery.put("employeeName",
        new BasicDBObject("$regex", "TestEmployee_[3]")
        .append("$options", "i"));
    System.out.println(regexQuery.toString());
    DBCursor cursor = collection.find(regexQuery);
    while (cursor.hasNext()) {
        System.out.println(cursor.next());
    } }

```

## APPENDIX B

### SCREENSHOTS

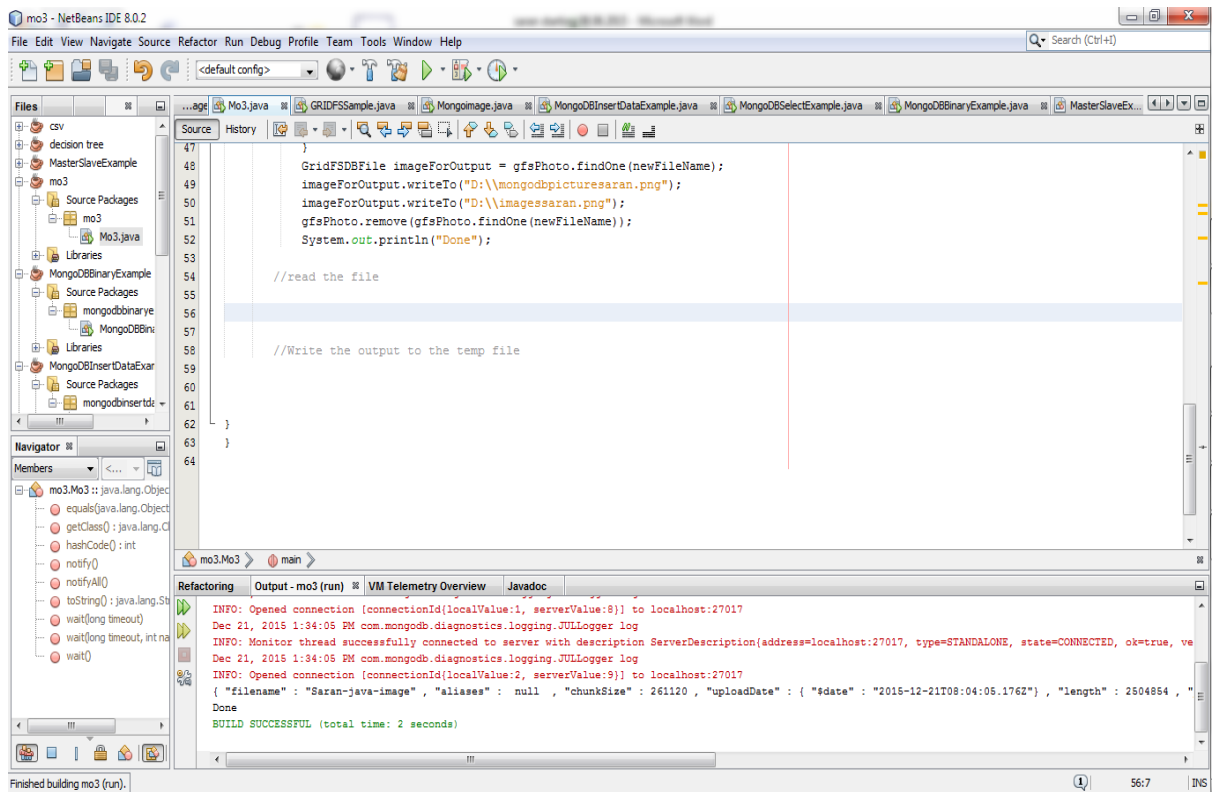


Fig. B.1 Screen shot for inserting a image file

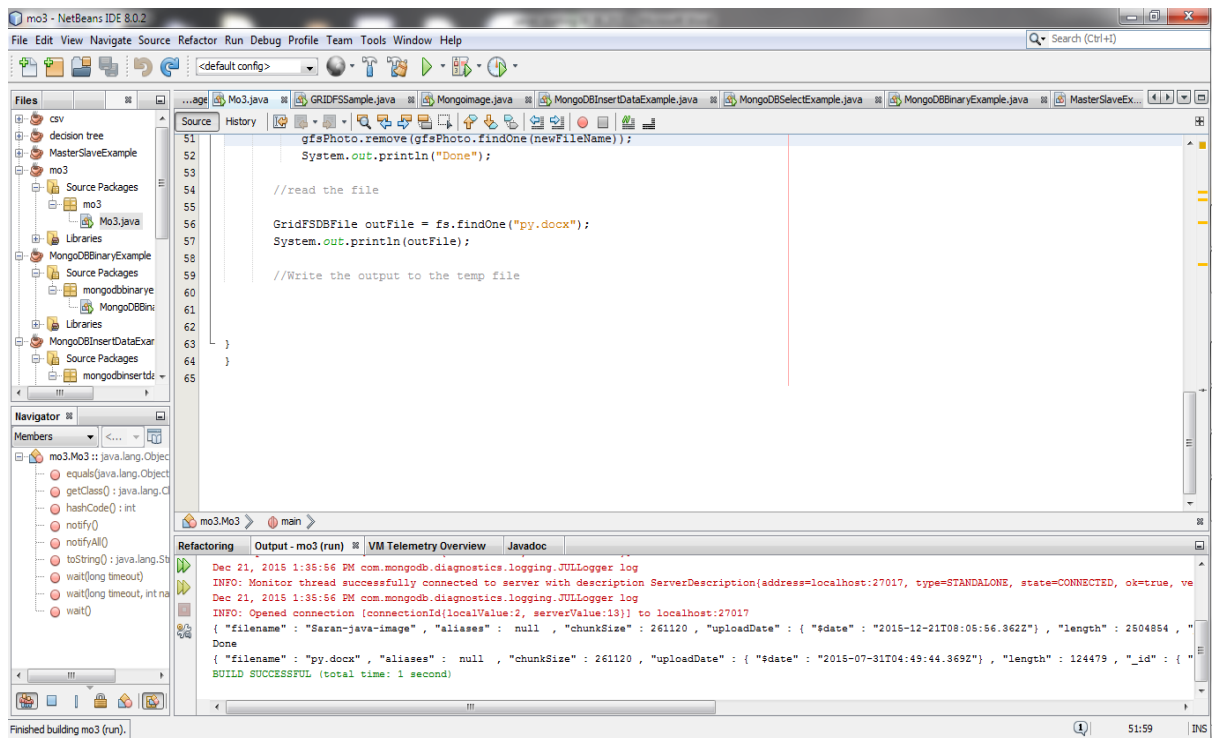
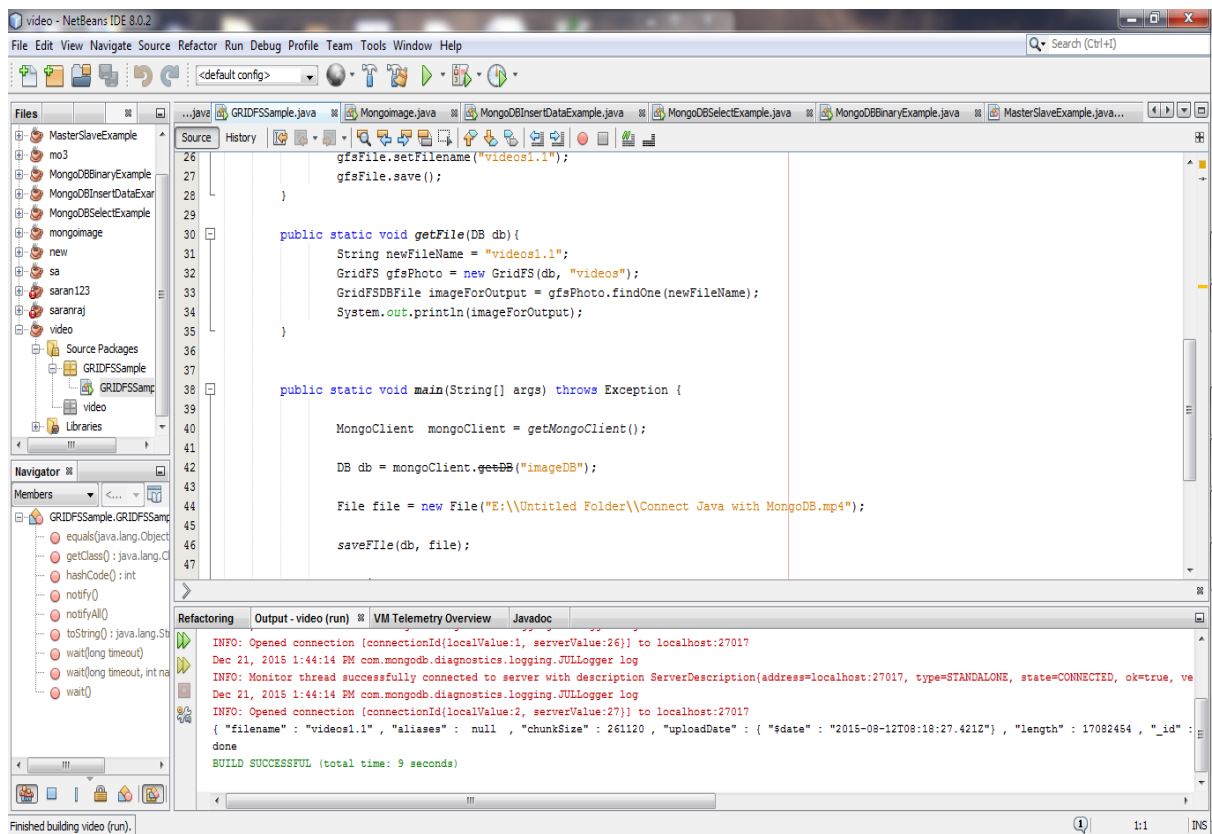
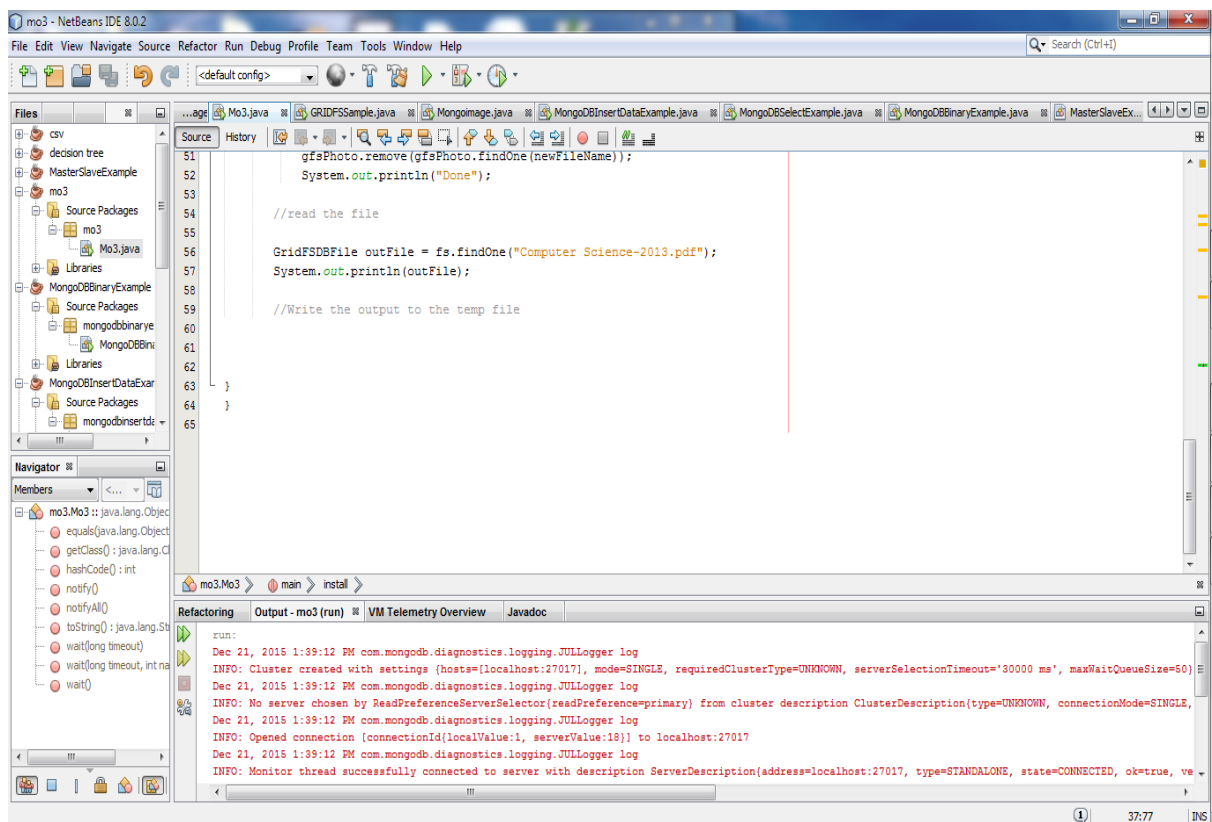


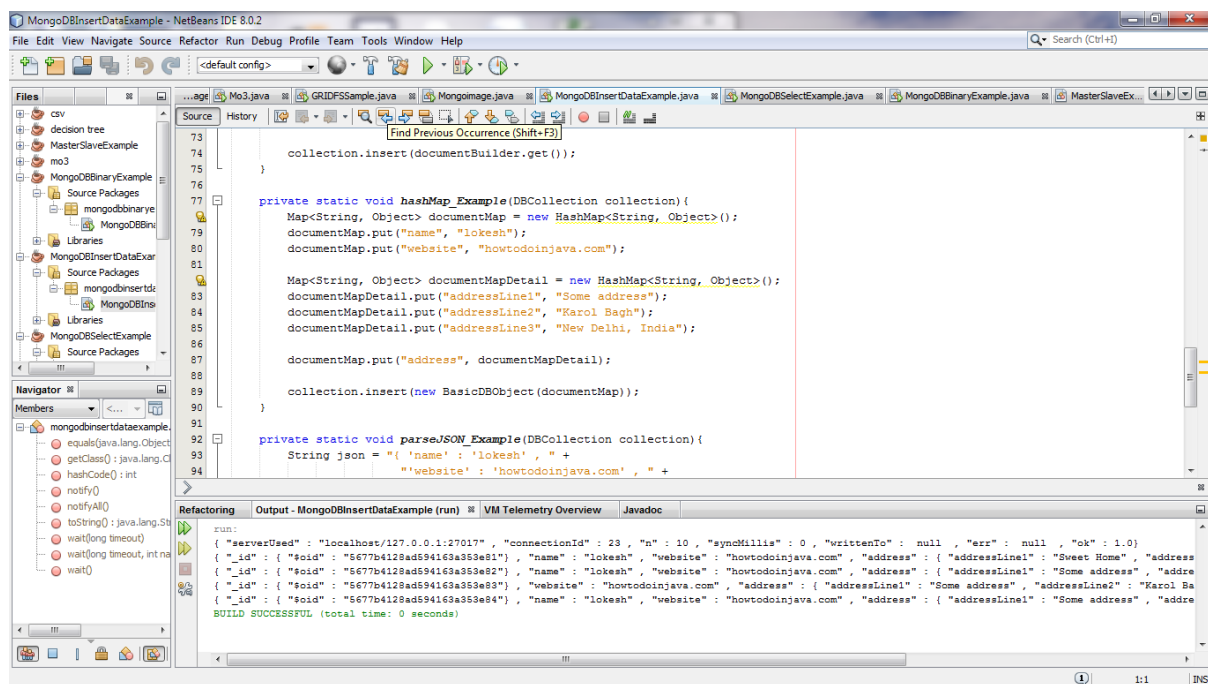
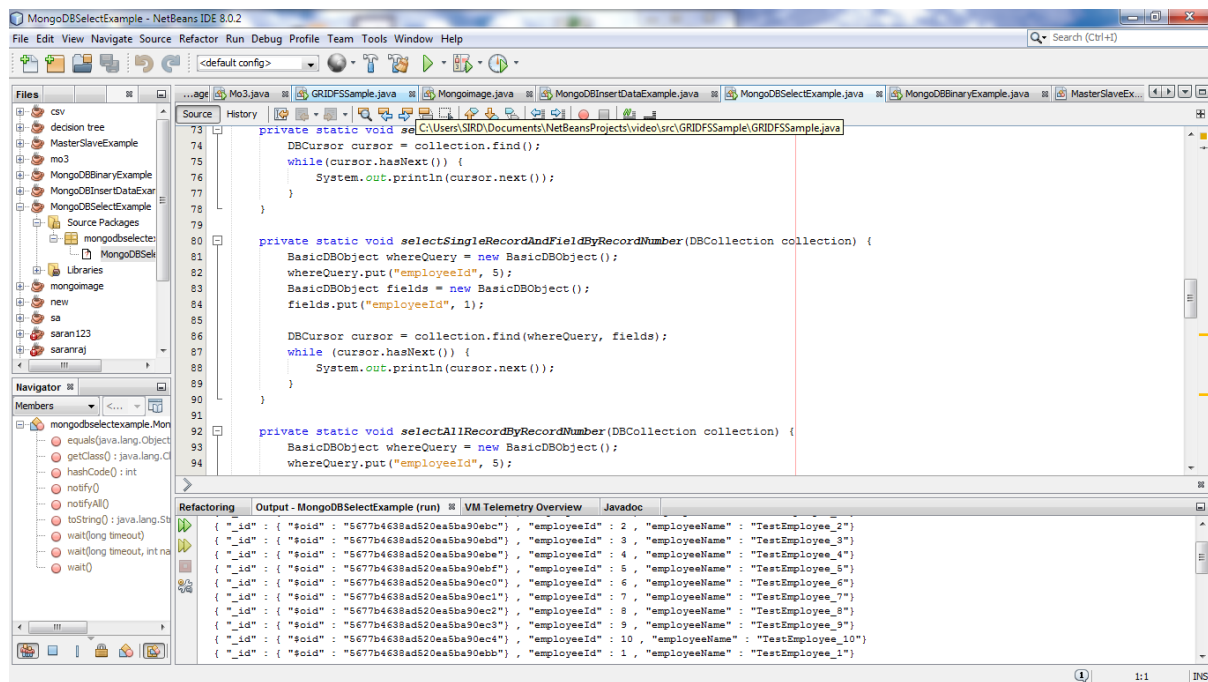
Fig. B.2 Screen shot for inserting a text file



B.3 Screen shot for inserting video file



B.4 Screen shot for inserting pdf file



B.5 Screen shot for inserting the large data's into database