Fall 2015

Entity and Relational Queries over Big Data Storage

Nachappa Achakalera Ponnappa
San Jose State University

Entity and Relational Queries over Big Data Storage

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Nachappa Achakalera Ponnappa

December 2015

The Designated Project Committee Approves the Project Titled

Entity and Relational Queries over Big Data Storage

by

Nachappa Achakalera Ponnappa

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2015

Dr. Duc Thanh Tran	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science
Dr. John Cruchon Dupeyrat	Data Scientist, Salesify, Inc.

ABSTRACT

Entity and Relational Queries over Big Data Storage

by Nachappa Achakalera Ponnappa

Big data storage involves using NoSQL technologies to handle and process huge volumes of data. NoSQL databases are non-relational, schema-free where data is stored as key-value pairs. The aim of the thesis is to implement Entity and Relational queries on top of Big Data storage.

In order to achieve this, we use NoSQL technologies like MongoDB and HBase. We implement various methodologies and solutions on top of MongoDB and HBase to map data across different tables and implement entity and relational queries to retrieve entities from huge volumes of data. We also measure the performance of both the technologies and optimize them to increase the retrieval speed.

ACKNOWLEDGMENTS

I am very thankful to my advisor Dr. Duc Thanh Tran for his continuous guidance and support throughout the project and believing in me. I would also like to thank the committee members Prof. Thomas Austin and Dr. John Cruchon Dupeyrat for guiding and monitoring the progress of the project and their valuable time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

1 INTRODUCTION

The main aim of this thesis is to implement entity and relational queries on top of the Big Data processing layer and using the Big Data programming model supported by that layer. This involves using various NoSQL technologies to store and process Big Data. NoSQL databases are typically key-value stores that are non-relational, distributed, horizontally scalable, and schema-free.

The main challenge in Big Data is gathering and processing huge volumes of data during which existing data and storage models need to be considered to enhance the importance of implementation issues. These issues include performance decrease by join operations and pressures on storage space as data tends to grow and exceed the capacity of hardware storage.

A schema needs to be defined even with big unstructured or semi-structured data because handling data relationships can be more complex. Data relationship logic cannot be hidden in a program, as it is not a good way to manage the complexity of data. Since big data uses the ‘structure layer’ approach, the data schema can only be known after the data is created.

Entity Queries can be defined as entity-lookup based on *identifiers, values, property-value pairs*.

Example: look up entities with “ID123”, “Shawn” or “Name: Shawn”

Relational Queries can be defined as entity-lookup based on *related entities, relationship-entity pairs*

Example: look up entities related to “ID123” or “AdvisorOf: ID123”

The proposed methodology is to define the schema for existing data after it has been collected and stored. The schema can be used at runtime to retrieve the data while processing. Also, the schema function should be as isolated and atomic as possible.

To achieve this, the scope of the data the schema applies to, and the versions of data the schema function can work for, should be explored. This methodology will be distinguished from traditional fixed schema being defined before the data is collected.

Since NoSQL technologies like MongoDB (document-based) and HBase (Big Table) are used to achieve proposed methodology, Figure 1 explains it graphically.

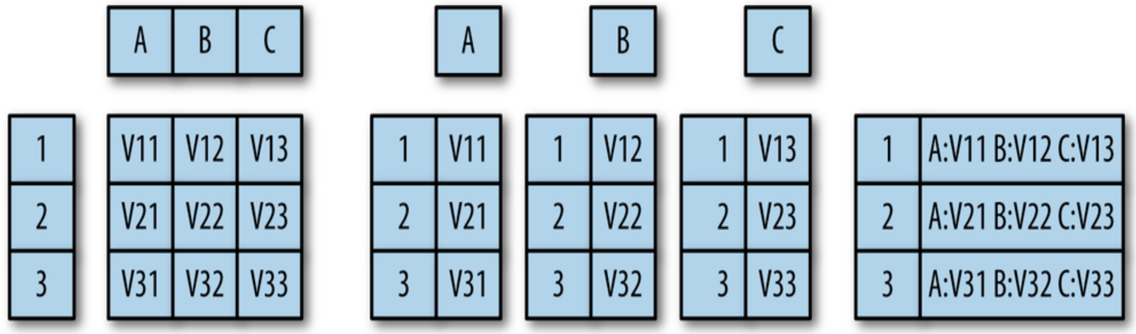


Figure 1. Comparing data storage systems

In the key-value store, the record is stored by its key while the user determines the relationship between the stored data and any schema associated with it. A columnar database decomposes rows into their individual fields and then stores, one field per file, in individual column files. In a Relational Database Management System (RDBMS), each row is an unique and individual entity. The schema defines the contents of each row, and the rows are stored sequentially.

When the user does not have any idea on the structure of the data, a key-value store is a better choice and own low-level queries can be implemented on top of it (e.g. processing of images and anything not easily expressed in SQL). However, if the data possesses some structure such as the ability to be represented in columns, or extensive and repeated references to the same data, then a relation or columnar model may be preferred.

Columnar databases are preferable when the data is easily divided into individual log records that don't need to cross-reference each other.

It is also well suited when the contents are relatively small. Columnar databases can be used to optimize queries by selecting and processing only a subset of columns from each record.

The columnar approach will not provide a performance boost when the schema has a limited number of columns (for instance, an image database containing small date column, a small ID column, and a large image column).

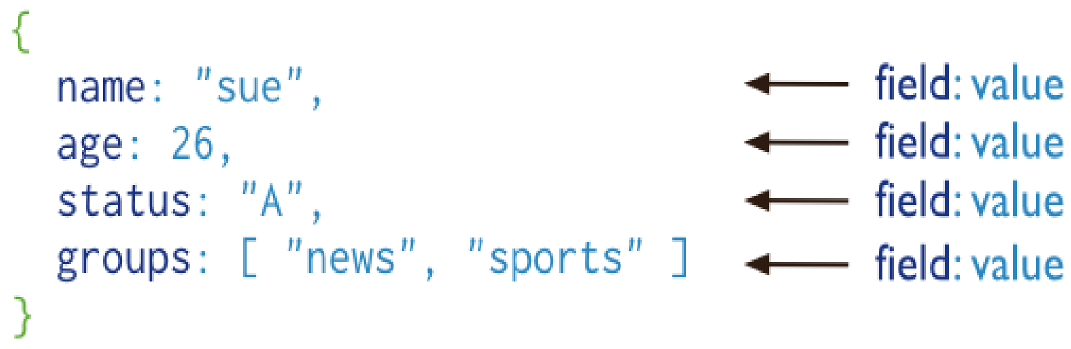
CHAPTER 2

2 BACKGROUND

This section introduces NoSQL technologies like MongoDB and HBase and helps the reader to get familiarized with various concepts revolving around them.

2.1 MongoDB

MongoDB, document database is a NoSQL database where documents are stored in the value part of the key-value store. Here the documents are indexed using a BTree and queried using a JavaScript query engine. Figure 2 illustrates an example of a document with records stored as key-value pairs. By default, _id field will be used as a primary key in each document. Each document can have a different structure and exist within the same collection.



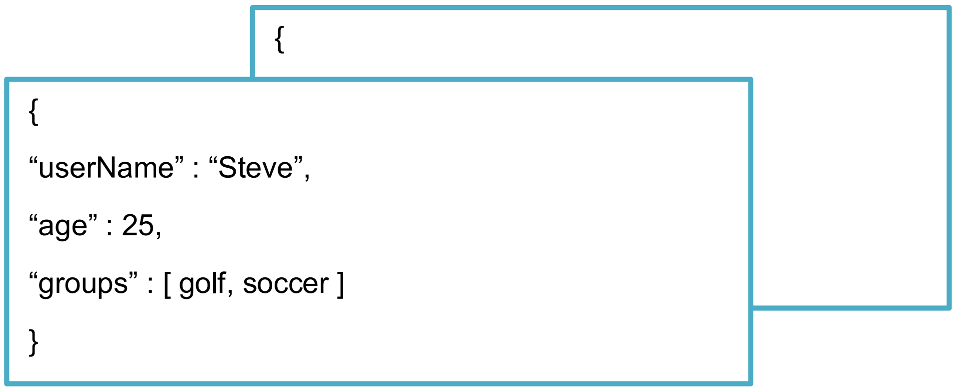
```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

Figure 2. Document structure

Compared to a relational database, a document-oriented database treats document as a row, a collection corresponds to a table and database to a schema.

2.1.1 Collection

A collection is a set of documents. It is equivalent to a table in a relational database containing a set of records. Figure 3 demonstrates a collection which has two documents. The documents in the same collection need not have the same set of fields or follow a fixed structure. Also, the fields in a document may hold different types of data.



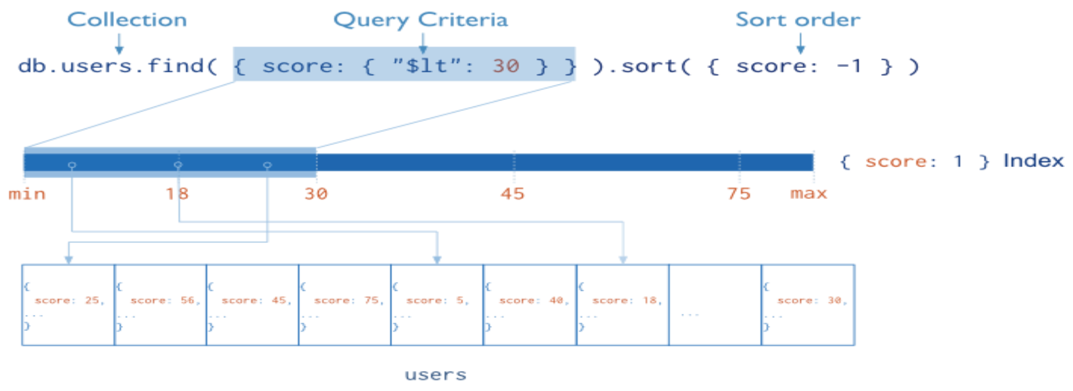
```
{
"userName" : "Steve",
"age" : 25,
"groups" : [ golf, soccer ]
}
```

Figure 3. Collections

2.1.2 Documents: Indexing

Indexing a document will result in making a query efficient and retrieving the entities queried, in a faster manner. An index can be used to restrict the number of documents to be inspected. Without an index being defined, a query would trigger a scan for every document in a collection thereby increasing the time to retrieve data.

By default, MongoDB will create an index on the _id field. A BTree is used to create an index and stores the data in fields ordered by values. Indexes in MongoDB are very similar to indexes in other database systems. In MongoDB, an index is defined at the collection level, and it is supported on any field of the documents in a MongoDB collection. As an example, let's consider Figure 4.



```
db.users.find( { score: { "$lt": 30 } } ).sort( { score: -1 } )
```

Figure 4. Indexing a collection

In the above example, ‘users’ is the collection, which is being queried, and an index is created on the score field. The above query restricts the score to less than 30 and sorts the field in descending order since -1 is specified as the argument.

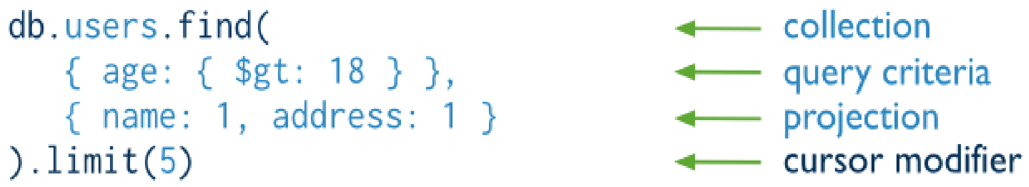
Any number of fields can be indexed depending on the query and the columns to be retrieved. An index can be created on a single field, multiple fields or array of fields.

2.1.3 Queries

A MongoDB query is used to specify a criteria or condition that is used to identify and retrieve the documents, bases on the specified criteria. A query may include any number of projections to specify the fields to be returned. A query can also impose sort orders, skips, and limit to restrict the documents being displayed. A set of operators may be included in a query to define how the find() method selects documents from a collection.

2.1.4 Query Interface

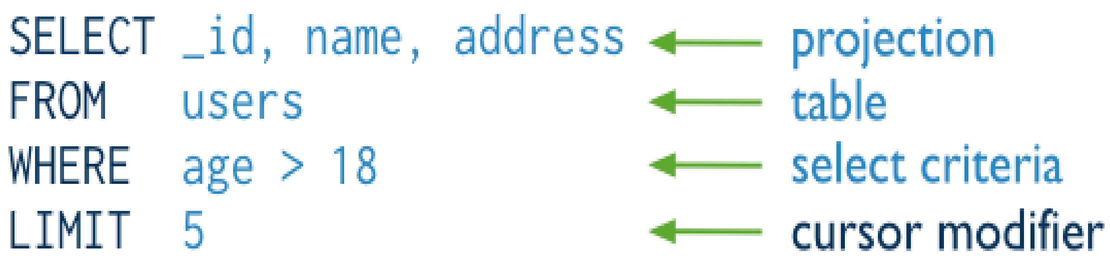
An example of a document-oriented database with terms relating to a relational database is as below:



```
db.users.find(
   { age: { $gt: 18 } },
   { name: 1, address: 1 }
).limit(5)
```

Figure 5. Query with conventions

The same query in SQL:



```
SELECT _id, name, address
FROM   users
WHERE  age > 18
LIMIT  5
```

Figure 6. Relational database conventions

2.1.5 Comparing relational databases with NoSQL document databases

In a relational database system, a schema must be defined before any record in added into a database. The schema is a structure defined in a formal language supported by the database and also provides a mapping for the tables along with their relationship to different tables existing in that database. Within each table, a constraint should be defined in terms of rows and columns, which also include the type of data to be stored in each column.

In contrast, a document-oriented database contains records, which are stored in the form of documents. Documents can be complex depending on the kind of data to be stored.

It also allows us to store nested data, which contains additional information about the record. It is also possible to use one or more document to represent a specific type of entity. The following figure demonstrates the use of document-based objects:

ID	Name	Address	Review
2355	Star Bucks	San Jose	Good Coffee
4128	McDonalds	Atlanta	Crunchy chips
3908	Amstel	Washington	Quality products

Table 1. Business Objects data in tabular form

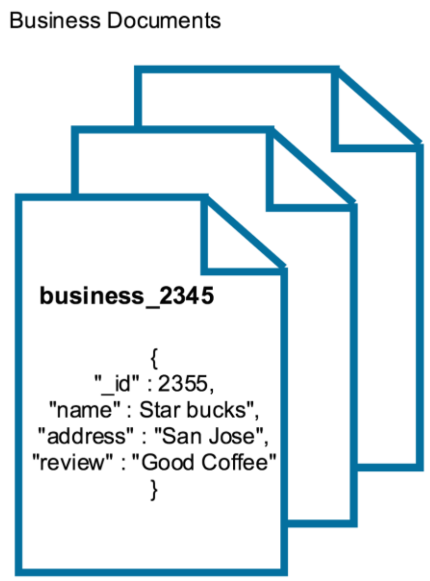


Figure 7. Business Documents

In this example, we have a table (Table 1) that stores information about certain businesses and their respective attributes: business_id, business_name, business_address, reviews and so on. From the above illustration, we can see that the relational model sticks to a particular schema with a specified number of fields that represent data for a specific purpose and data type. Figure.7 represents a document-based model where an individual document is maintained for each business. With this model, we can have store any number of fields in the document without having to follow a fixed schema.

In a document-oriented model, data entities are stored as documents and each document enables us to store, and access/modify the data (update, delete). Instead of storing names and data types for the columns, the data is defined in the document and a value is provided as the description. If we wish to add more columns/attributes to the relational model, we need to modify the database schema to incorporate new columns. In a document-based model, we would simply add additional key-value pairs into the documents that are represented as new fields.

Data is typically shared across multiple tables in a relational model. In such a scenario, there is less duplicated data. We would have repeated information about the businesses and the reviews (for each business) in case we did not separate business and review information stored into different set of tables. The problem with such an approach is that, when the information is changed across the tables, we need to lock the tables simultaneously to ensure that the information changes across the table consistently. In addition, since a relational model follows a fixed structure, it makes it hard to change the schema while distributing data across multiple servers.

Let's consider two different document structures in the document-oriented model; one for business and one for reviews. Instead of dividing the entities into tables and rows, we would turn them into documents. By maintaining a reference in the business document to a review document, we create a relationship between the two entities.

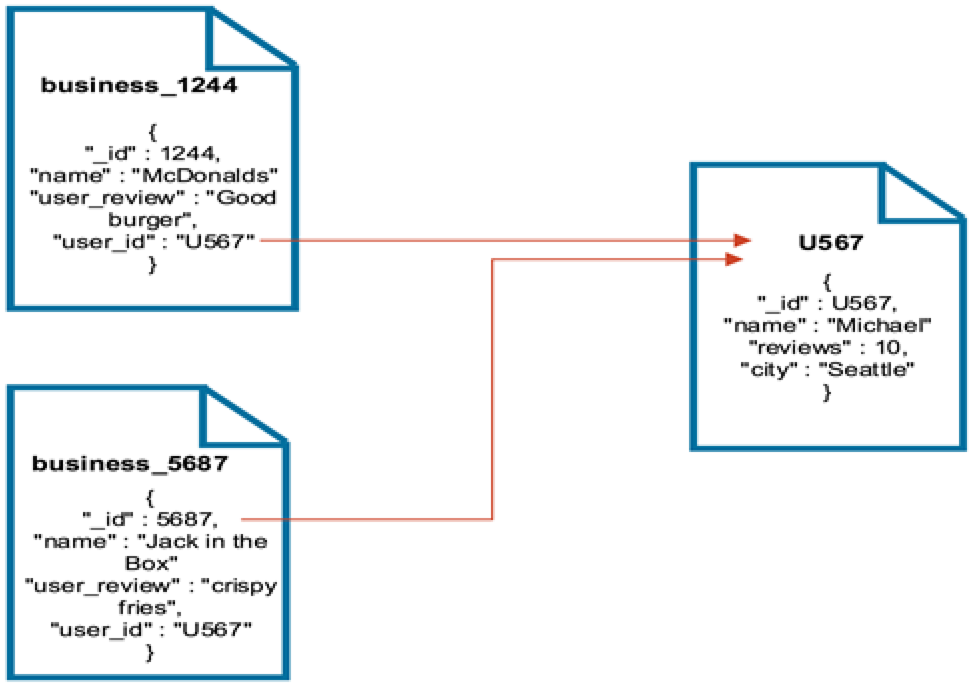


Figure 8. Business information with user reviews

In the above example, we have two businesses with reviews from the same user. We have represented each business as a separate document and add the user reference information in the user field. There are many advantages in following the document-oriented approach when compared to the traditional RDBMS model. Firstly, updating the schema is just updating the documents in a document-oriented model, which can be done with no system downtime. Secondly, the information can be distributed across multiple servers with great ease. It is also easier to move or replicate entire objects to a different server since all the data is contained within the documents.

2.1.6 Modeling documents for retrieval

When the user has knowledge about the relationship between documents, it is up to the user to determine how the document should be modeled and structured. A document can have an entity that is related to many other entities from a different collection. In other words, a document can have references to another document with a one-to-many relationship, which is often known as a has-many relationship. Let's consider an example of one-to-many relationship where a single business can have many reviews associated with it, i.e. business has-many reviews and conceptually it would appear as follows:

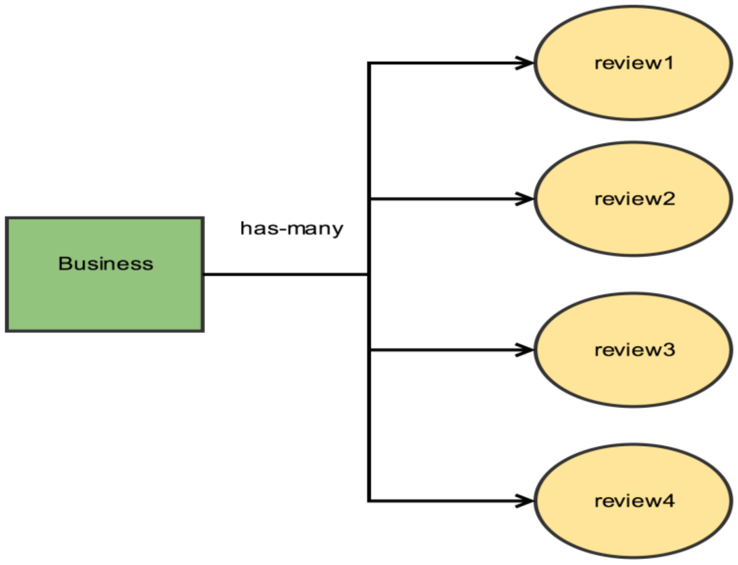


Figure 9. Business – Review mapping

In the business document, we reference the review by storing review_ids in an array. The business document having many reviews can be structured as below:

```
{
“Business_id”: “business_a”
“reviews” : [“review_1”,”review_2”,”review_3,..”],
…
}
```

Since we are working with a flexible, document-centric design, the user can store all the references to the object in the opposite way as described earlier. The review object can also store references to the business object which is known as having a many-to-one relationship, also called as belongs-to relationship. In the business document, we have business_id as the unique qualifier which can be used to refer from review documents. Each of the review documents can be represented by the following JSON document:

```
{
“review_id”: “review_1”,
“reviewed_on_business”: “business_1”,
“text”:“ABC business provides good service”,
“stars” : “5”,
…
}
```

With this alternative approach, information about the relationship between business and review objects can be provided in each review document where "reviewed_on_business" field would be used to link the business document.

Out of the two different techniques explained to model the document, it is up to the user to determine and choose the most appropriate one to the requirements on hand. When many updates from different processes are expected to occur in a document, it is optimal to choose belongs-to relationship model.

On the other hand, retrieving information from the documents is also a priority. The way the documents are related to each other and how references are provided between documents influence the way the data is retrieved. Since the business document maintains a has-many relationship and contains references for reviews, the user needs to find all the reviews associated with a business. Different business requirements may require different modeling techniques as explained. In our scenario, we use has-many relationship model since we need to retrieve all the reviews associated with a business and by indexing the field containing the array of reviews, we can retrieve data faster and achieve better performance.

2.2 Apache HBase

HBase is a member of column family in a NoSQL database, which runs as a distributed and scalable data store on top of Hadoop. This allows HBase to use Hadoop's MapReduce programming model and leverage the distributed processing paradigm of Hadoop Distributed File System (HDFS). HBase is a powerful database that blends real-time querying with the key-value store and performs batch processing via MapReduce. HBase has a different approach for modeling the data and defines a four-dimensional data model in which the following coordinates define each cell:

- **Row Key:** Each record has a unique row key. The row keys do not have a data type associated with it and is treated as a byte array. This is similar to a primary key in relational database model. As per the row key, records in HBase are stored in a sorted manner.
- **Column Family:** Data within a row is organized into column families. Each row has the same set of column families, but across rows, column qualifiers need not be associated with the same column families.
- **Column Qualifier:** Column qualifiers are column families which define the actual columns. We can treat a column qualifier as the column itself.
- **Version:** Each column can have different number of versions associated with it and the user can access the data for a specific version of a column qualifier.

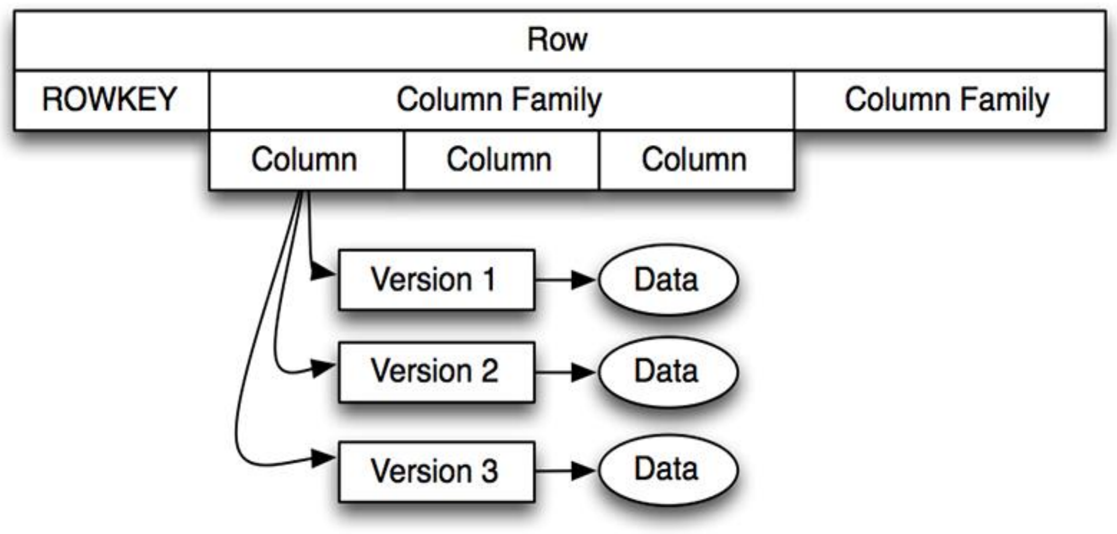


Figure 10. HBase Four-Dimensional Data Model

As shown in Figure 10, an individual row is composed of one or more column families and can be accessed through its row key. Each column family can have one or more column qualifiers (referred to as Column in the Figure 10.) and each column can have multiple versions. The user needs to know the row key, column family, column qualifier and the version in order to access a particular set of data.

While designing an HBase data model, it is essential to know the way data is going to be accessed. The user can access the data stored in HBase in the following ways:

- Using the table scan for a range of row keys.
- Using MapReduce while batch processing.

This dual approach of accessing the data makes HBase a powerful database.

HBase has master-slave architecture, composed of 3 types of servers namely - Region servers, Data node and Name node. Data for reads and writes are provided by the Region server. The clients communicate directly with the region servers while accessing the data. HBase Master takes care of region assignment and DDL operations such as creating/deleting tables.

The **data node(s)** stores the data which is maintained by the region server. All HBase data is stored in Hadoop Distributed File System (HDFS) files. Region servers are placed in close conjunction with the data nodes to access the data.
The meta-data information for all the physical blocks in the files is maintained by the **name node(s).**

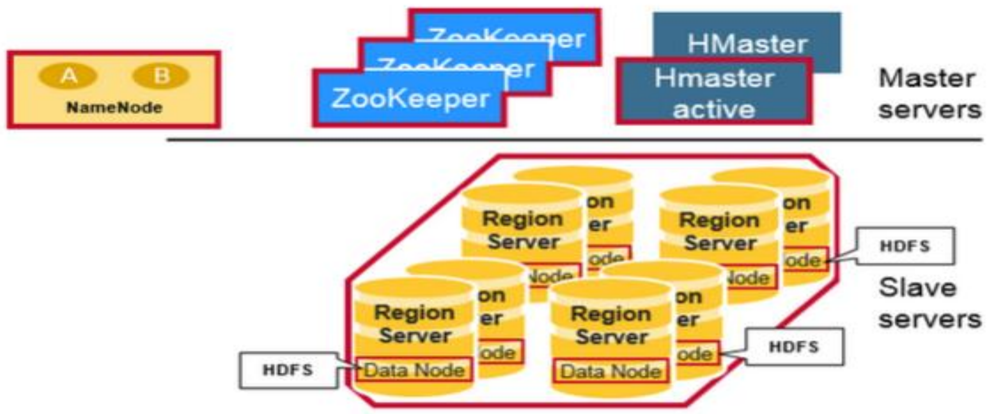


Figure 11. Master-Slave architecture in HBase

2.2.1 Regions

HBase tables are horizontally divided based on row key range into "Regions". All the rows of the data from the region's start key to end key are stored in a region. These regions are assigned to the nodes in the cluster, called Region-Servers which serve data for read and writes.

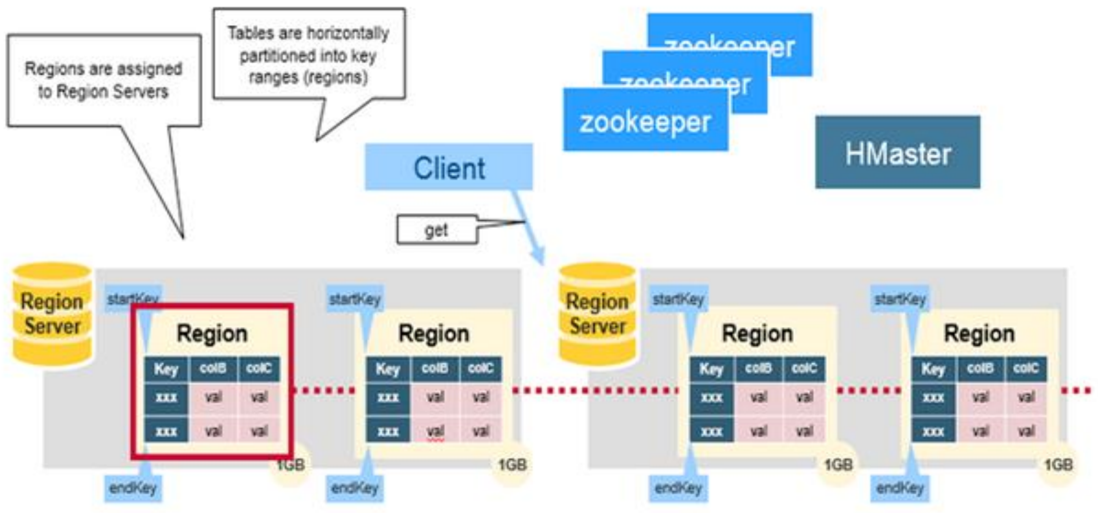

Regions are assigned to Region Servers
Tables are horizontally partitioned into key ranges (regions)
zookeeper
zookeeper
zookeeper
Client
HMaster
get
Region Server
startKey
Region
Key
colB
colC
xxx
val
endKey
1GB

Figure 12. Region Server with Zookeeper in HBase

2.2.2 HBase Master

The HBase Master is responsible for region assignment and DDL operations. An HBase master is responsible for:

- Assigning regions at the beginning, re-assigning regions during recovery operations and load balancing
- Keeping track and monitoring all the region servers in the cluster.
- Provides an interface for creating, updating and deleting tables.

2.2.3 Zookeeper

HBase uses Zookeeper as a distributed coordination service in order to keep track of the server state in the cluster. Zookeeper keeps tracks of which servers are alive and available and also notifies HMaster when a server fails.

2.2.4 META Table

The Meta table is an HBase catalog table which stores the location of the regions in the cluster. The location of the Meta table is stored in the Zookeeper cluster. When a client reads or writes to HBase, the following operations occurs:

- The Zookeeper cluster provides details of the region server that hosts the Meta table.
- The .META server is queried by the client to get the information of the region server corresponding to the row key to be accessed. The client caches this information along with the location of the Meta table.
- The row corresponding to the Region server is fetched.
- The client uses the information cached to retrieve the location of the META table and the previously read row keys. It will use this information for future queries and only when the region server has changed; it will re-query and update the cache.

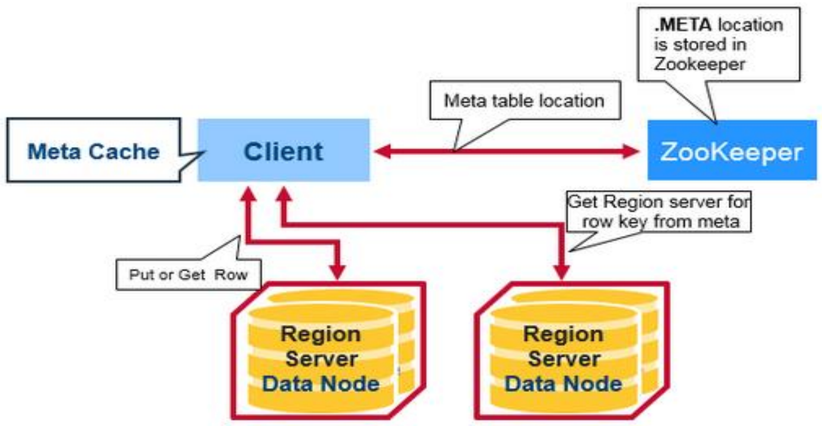


Figure 13. Meta table in HBase

The following are the advantages of using HBase:

- It provides a strong consistency model.
- It will scale automatically – when the data grows too large, the regions split and use HDFS to distribute and replicate data.
- Failure detection – When a node fails, the writes in progress will be automatically recovered and the changes will not be flushed. The region server that was handling the data will be reassigned where the node failed.
- Real-time queries – HBase provides real time, random access to the data to efficiently store and query data.

2.2.5 Apache Phoenix

Apache Phoenix is a relational database layer for Apache HBase. It maintains a query engine which transforms SQL queries into native HBase scans. Accessing HBase data with Phoenix can be substantially faster than direct HBase API as Phoenix parallelizes queries based on stats and pushes the processing into the region servers where data resides.

Phoenix table maps one to one with HBase table and there are 2 types of columns namely:

- Key-value columns - map to column qualifiers. They predefine the column qualifiers which appear in HBase table. It also lets you to create column qualifiers dynamically at real time.
- Row-key columns - they are made up of the primary key constraints.

Figure 14. depicts the working of Apache Phoenix in HBase architecture

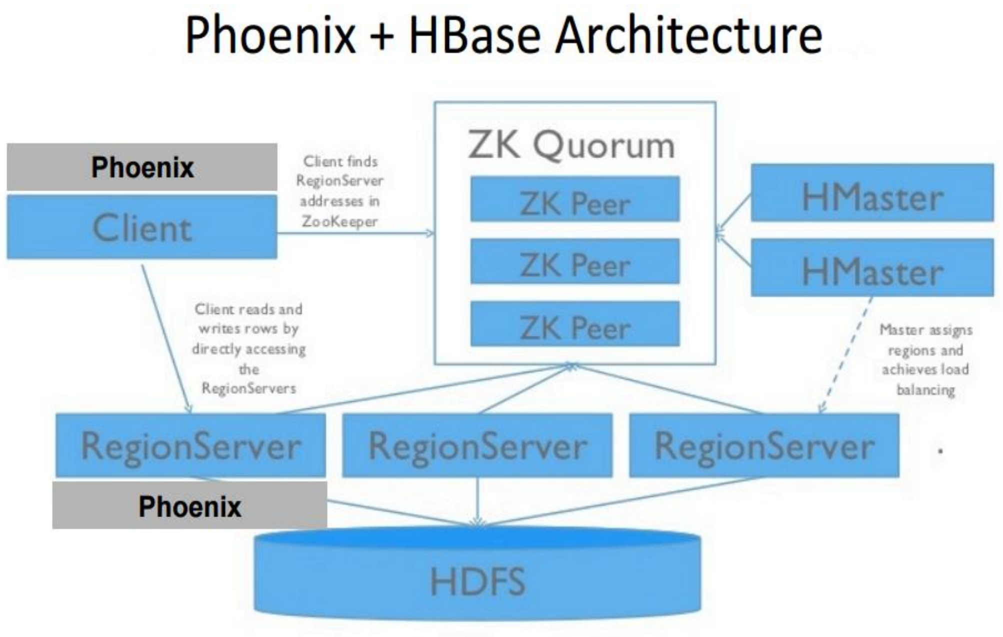

Phoenix + HBase Architecture
Phoenix
Client
Client finds RegionServer addresses in ZooKeeper
ZK Quorum
ZK Peer
ZK Peer
ZK Peer
HMaster
HMaster
Client reads and writes rows by directly accessing the RegionServers
Master assigns regions and achieves load balancing
RegionServer
RegionServer
RegionServer
Phoenix
HDFS

Figure 14. Phoenix and HBase architecture

2.3 Apache SOLR

Apache SOLR is a java based scalable solution built on top of Apache Lucene. SOLR is highly reliable, scalable and fault tolerant. It also provides distributed indexing, replication, load-balanced querying and automates failure and recovery. Some of the features in SOLR include:

- Supports multiple approaches to query, parsing, making it easy to find the data.
- Extensive filtering feature which allows applications to control what content is searched.
- Provides a flexible query interface allowing pluggable query parsers.
- Sort by any number of fields, and by complex functions of numeric fields.
- Near Real Time (NRT) search allows access to document addition and updates almost immediately.

2.4 Challenges with NoSQL databases

This section discusses few challenges faced by NoSQL databases. Few of the important ones are discussed as below:

- NoSQL technologies like MongoDB and HBase use unstructured data, and hence the concept of fixed schema does not apply here.
- Mapping data from different collections or tables is not possible as the JOIN operation is not supported by NoSQL technologies.
- The performance of the system can be hampered when a huge volume of data is being read/scanned to perform a query operation.
- Having nested/embedded document structure in MongoDB can make the document structure to be complex and hard to maintain.
- Mapping data across tables in HBase is hard as there is no join functionality supported by it. Custom MapReduce functionality needs to be implemented to achieve it.
- It is not always feasible to use MongoDB or HBase for certain problems since it depends on the business requirements and the type of data to be handled. Selection of the right technology involves understanding the requirements thoroughly.
- When DocumentDB is best suited? - When the user does not have any idea on the structure of the data, key-value store are a good choice and own low-level queries can be implemented on top of it.
- When BigTable is best suited? - When the data possesses some structure, such as the ability to be represented into columns, or has extensive and repeated references to the same data

CHAPTER 3

3 PROJECT SETUP

We installed MongoDB and HBase along with other essential technologies to measure the performance of each system. We imported a public key and creates a list file to setup MongoDB on a Linux environment (Ubuntu). Once all the necessary packages were installed, we checked the status of MongoDB server by executing: **sudo service mongod status**

The user can check the log files to verify if MongoDB server is up and running. The log files are located at /var/log/mongodb/mongod.log

We installed HBase as a single-node, standalone instance and modified the configuration files to specify the directory for Zookeeper. HBase can be started by executing the following command: **bin/start-hbase.sh**. We connect to the HBase shell to create tables, insert records into the table, and perform scan operations on the table using the command: **./bin/hbase shell**

In order to measure the performance of relational queries in HBase, we use Apache Phoenix – a relational database layer having a query engine to transform SQL queries into native HBase API calls. It has a metadata repository, which is type accessed to store data into HBase tables. In order to install Apache Phoenix on top of HBase we add the jar files; phoenix-[version]-client-minimal.jar and phoenix-core-[version].jar files to the class path of every HBase region server. The path will usually be at: **/hbase/hbase-<version>/lib** directory.

Since working in command line can be tedious, we used SQuirrel SQL Client, which is a database administration tool that allows the users to explore and interact with the HBase. It provides a GUI which has the look and feel of a relational database while working with HBase tables. We set up this by copying phoenix-[version]-client.jar to the lib directory where SQuirrel SQL client is downloaded. The user needs to add the phoenix JDBC driver by specifying the construct URL as jdbc:phoenix:localhost.

CHAPTER 4

4 EXPLAINING THE CODE

4.1 Dataset used

Yelp, the business review website makes an academic dataset available which is used for this thesis. The Yelp academic dataset contains details about various businesses, star ratings, reviews, users' information. The schema is as below:

Business Objects: Contain basic information about local businesses. The structure of the dataset is as below:

```
{
    'type': 'business',
    'business_id': (a unique identifier for this business),
    'name': (contains the full name of the business),
    'neighborhoods': (a list of neighborhood names, might be empty),
    'full_address': (contains the local address of a business),
    'city': (city),
    'state': (state),
    'latitude': (latitude),
    'longitude': (longitude),
    'stars': (contains the ratings for a business, rounded to half-
stars),
    'review_count': (review count),
    'categories': [(localized category names)]
 }
```

Review Objects: Contains information about the review text, star rating, along with corresponding user_id and business_id

```
{
    'type': 'review',
    'business_id': (identifies the reviewed business),
    'review_id': (a unique identifier for this review),
    'user_id': (identifies the user associated with a review),
    'stars': (contains the ratings provided by the user),
    'text': (contains the review(s) provided by the user),
    'date': (contains date, formatted like 'YYYY-MM-DD'),

}
```

User Objects: Contains aggregated information about the user who has reviewed a business.

```
{
    'type': 'user',
    'user_id': (a unique identifier for this user),
    'name': (contains the first name, last initial, like 'James P.'),
    'review_count': (contains the review count),
    'average_stars': (floating point average, like 4.31),
}
```

4.2 MongoDB implementation

4.2.1 Brute Force Method

Let’s consider business and review objects. We can find the relationship between these two objects by mapping the business_id key across both the collections and fetching the information about reviews for each business. The following code does the job:

Solution 1:

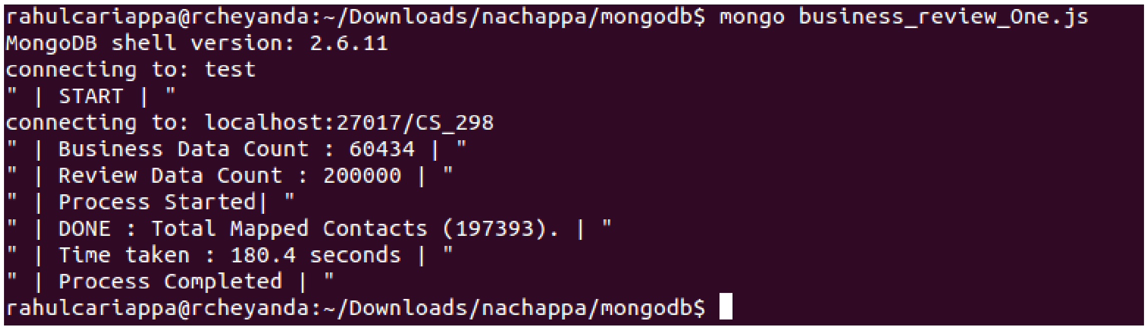
Code Snippet:

```
function mapCollections() {

	var bulkInsertOp = businessReviewData.initializeUnorderedBulkOp();
	reviewData.find().addOption(16).forEach(function(reviewDataDoc) {
			business_id = (reviewDataDoc.business_id).toString();
			businessData.find({"business_id":business_id}).addOption(16).forEach(fu
			nction(businessDataDoc) {
				bulkInsertOp.insert({
					"business_id" : business_id,
					"business_name" : businessDataDoc.name,
					"business_address" : businessDataDoc.full_address,
					"review_id" : reviewDataDoc.review_id,
					"review" : reviewDataDoc.text,
					"date" : reviewDataDoc.date
				});
			});
	});
	bulkInsertOp.execute();
} mapCollections();
```

In the above code, we see that business_id is used to map business and review objects together and store the selected fields into an auxiliary collection (business_review). The resulting collection will have review text mapped for each business. Once the data is stored in the auxiliary collection, it can be further indexed to retrieve entities in a faster manner.

Solution 1 can be optimized and the performance can be improved by using cursors. A cursor is a pointer to the result returned by the query. Instead of processing all the documents returned together, the cursor act as a pointer to each document returned by the query and hence the performance of the process improves.

Output:



```
rahulcariappa@rcheyanda:~/Downloads/nachappa/mongodb$ mongo business_review_One.js
MongoDB shell version: 2.6.11
connecting to: test
" | START | "
connecting to: localhost:27017/CS_298
" | Business Data Count : 60434 | "
" | Review Data Count : 200000 | "
" | Process Started| "
" | DONE : Total Mapped Contacts (197393). | "
" | Time taken : 180.4 seconds | "
" | Process Completed | "
rahulcariappa@rcheyanda:~/Downloads/nachappa/mongodb$
```

Figure 15. Business-Review mapping Output

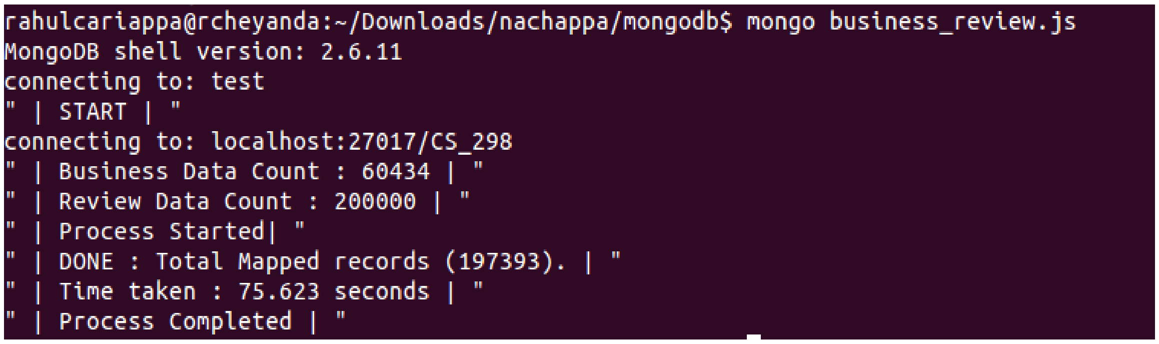
Solution 2: Optimization using Cursors.

Code Snippet:

```
function cursorAssoc() {

        var bulkInsertOp = businessReviewData.initializeUnorderedBulkOp();
        var consolidatedDataCur = null;
        reviewData.find().addOption(16).forEach(function(reviewDataDoc) {
                business_id = (reviewDataDoc.business_id).toString();

            var consolidatedData = null;
            consolidatedDataCur =
            businessData.find({"business_id":business_id});
            while(consolidatedDataCur.hasNext()){
                    matchCount++;
                    consolidatedData = consolidatedDataCur.next();
                    bulkInsertOp.insert({
                      "business_id" : business_id,
                      "business_name" : consolidatedData.name,
                      "business_address" : consolidatedData.full_address,
                      "review_id" : reviewDataDoc.review_id,
                      "review" : reviewDataDoc.text,
                      "date" : reviewDataDoc.date
                        });
                }
        });

        bulkInsertOp.execute();
        printjson(" | DONE : Total Mapped records (" + matchCount + "). | ");
        var end = new Date().getTime();
        var timenow = (end - start)/1000;
        printjson(" | Time taken : " + timenow + " seconds | ");

    printjson(" | Process Completed | ");
}

cursorAssoc();
```

In the above code snippet, we observe that the cursor defined as consolidatedDataCur reads the business objects collection and inserts only the mapped documents into the auxiliary collection. The performance of this solution is faster than the previous one.

Output:



```
rahulcariappa@rcheyanda:~/Downloads/nachappa/mongodb$ mongo business_review.js
MongoDB shell version: 2.6.11
connecting to: test
" | START | "
connecting to: localhost:27017/CS_298
" | Business Data Count : 60434 | "
" | Review Data Count : 200000 | "
" | Process Started| "
" | DONE : Total Mapped records (197393). | "
" | Time taken : 75.623 seconds | "
" | Process Completed | "
```

Figure 16. Business-Review mapping Cursor Output

Solution 3: Optimization using Hash Maps

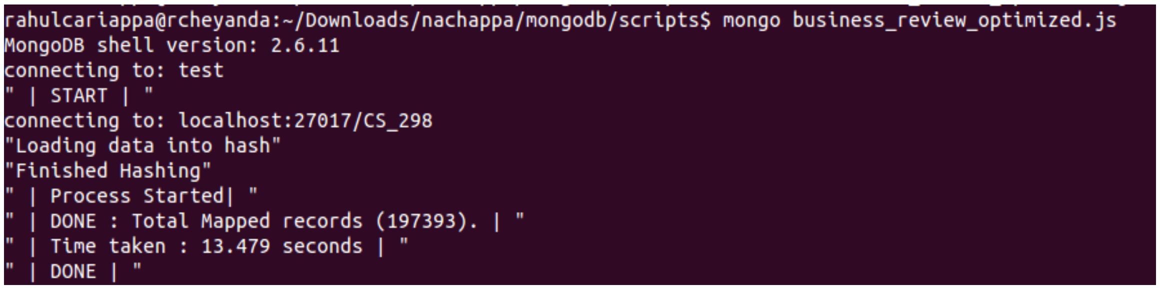
Solution 2 can further be optimized using Hash Maps to read the documents into memory and map them. In this solution, a hash of all the unique values will be created and stored into the memory which will further be used to map with different collections. Since business_id is the unique identifier for each business, we create a hash map for it storing all the business docs associated with it. We next check if the business_id of review objects is present in the hash map for business objects and proceed with mapping all the matching documents. This improves the performance to a great extent as everything is being read from memory and being inserted into the auxiliary collection.

The performance of such a design is O(n) when compared to O(n^2) of previous solutions. This implementation can be extended for user objects by creating a hash map for user_ids.

Code Snippet:

```
function generateHashMap(collectionName){
   var CS_298DB = db.getSiblingDB("CS_298");
   var businessData = CS_298DB.getCollection(collectionName);
   var hash = {};
   businessData.find().forEach(function(mydoc){ hash[mydoc.business_id] = mydoc;  })
   printjson("Finished");
   return hash;
}
function hashMapAssoc() {
        var businessData = generateHashMap("business_data");
        var businessReviewData =
        CS_298DB.getCollection("business_review_aux_HM");
        var bulkInsertOp = businessReviewData.initializeUnorderedBulkOp();
        reviewData.find().addOption(16).forEach(function(reviewDataDoc) {
        business_id = (reviewDataDoc.business_id).toString();
        if(business_id in businessData){
                bulkInsertOp.insert({
               "business_id" : business_id,
               "business_name" : businessData[business_id].name,
               "business_address" : businessData[business_id].full_address,
                "review_id" : reviewDataDoc.review_id,
                "review" : reviewDataDoc.text,
                "date" : reviewDataDoc.date
           });
   } });
      bulkInsertOp.execute();
 } hashMapAssoc();
```

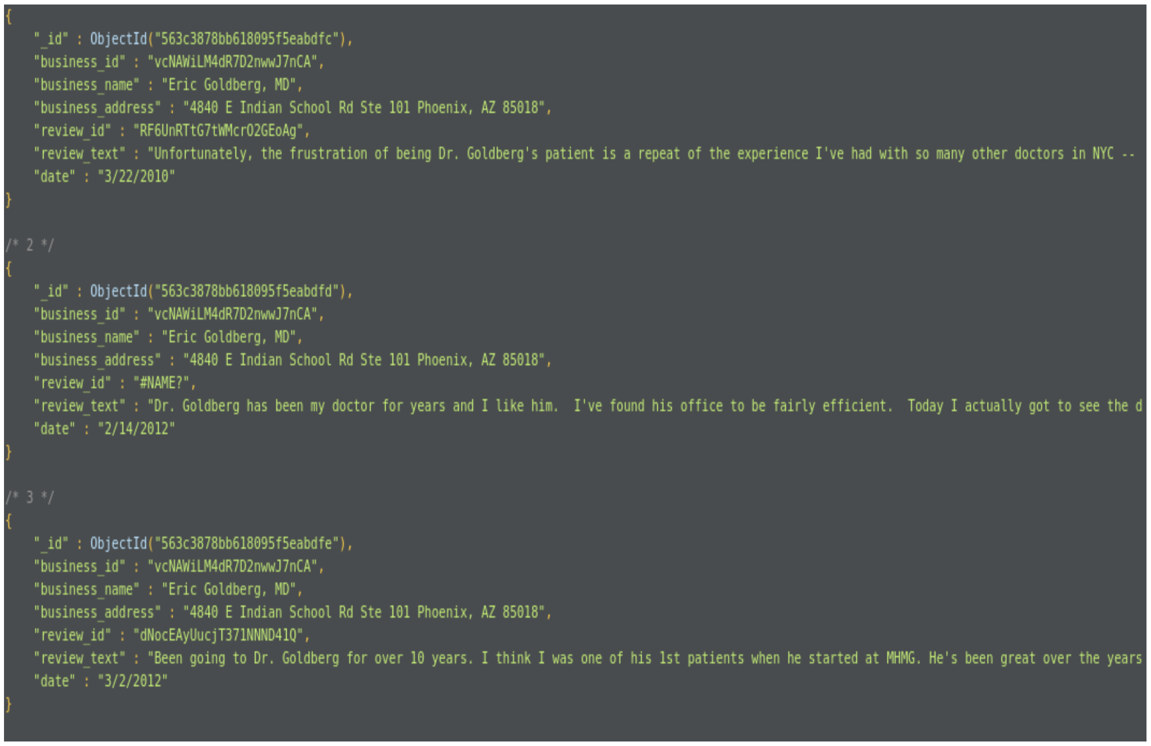
Output:



```
rahulcariappa@rcheyanda:~/Downloads/nachappa/mongodb/scripts$ mongo business_review_optimized.js
MongoDB shell version: 2.6.11
connecting to: test
" | START | "
connecting to: localhost:27017/CS_298
"Loading data into hash"
"Finished Hashing"
" | Process Started| "
" | DONE : Total Mapped records (197393). | "
" | Time taken : 13.479 seconds | "
" | DONE | "
```

Figure 17. Business-Review mapping Hash Map Output

Data in the auxiliary collection will be stored as:



```
{
    "_id" : ObjectId("563c3878bb618095f5eabdfc"),
    "business_id" : "vcNAWiLM4dR7D2nwwJ7nCA",
    "business_name" : "Eric Goldberg, MD",
    "business_address" : "4840 E Indian School Rd Ste 101 Phoenix, AZ 85018",
    "review_id" : "RF6UnRTtG7tWMcrO2GEoAg",
    "review_text" : "Unfortunately, the frustration of being Dr. Goldberg's patient is a repeat of the experience I've had with so many other doctors in NYC --
    "date" : "3/22/2010"
}

/* 2 */
{
    "_id" : ObjectId("563c3878bb618095f5eabdfd"),
    "business_id" : "vcNAWiLM4dR7D2nwwJ7nCA",
    "business_name" : "Eric Goldberg, MD",
    "business_address" : "4840 E Indian School Rd Ste 101 Phoenix, AZ 85018",
    "review_id" : "#NAME?",
    "review_text" : "Dr. Goldberg has been my doctor for years and I like him.  I've found his office to be fairly efficient.  Today I actually got to see the d
    "date" : "2/14/2012"
}

/* 3 */
{
    "_id" : ObjectId("563c3878bb618095f5eabdfe"),
    "business_id" : "vcNAWiLM4dR7D2nwwJ7nCA",
    "business_name" : "Eric Goldberg, MD",
    "business_address" : "4840 E Indian School Rd Ste 101 Phoenix, AZ 85018",
    "review_id" : "dNocEAyUucjT371NNND41Q",
    "review_text" : "Been going to Dr. Goldberg for over 10 years. I think I was one of his 1st patients when he started at MHMG. He's been great over the years
    "date" : "3/2/2012"
}
```

Figure 18. Business-Review Data Output

4.2.2 Method 2: Using MapReduce to establish a relationship between collections.

MongoDB supports MapReduce functionality where the map phase is applied to each input document emitting key-value pairs. The reduce phase collects and condenses the aggregated results present in different collections.

Since business_id is the unique identifier for each business, we will use the mapper to emit values for each of the business_id for both business and review objects.

Solution 3:

Code Snippet:

Mapper:

```
var mapBusiness = function() {
    emit(this.business_id, {business_id: this.business_id,name: this.name,
full_address:this.full_address, city:this.city, state: this.state, review_id: null, text: null});
};

var mapReview = function() {
    emit(this.business_id, {business_id: null, name: null, full_address:null, city:null, state:
null, review_id: this.review_id, text: this.text});
};
```

The reducer will combine the fields: business_id, name, full_address, city, state, review_id and text from both the collections and aggregate them into one single document.

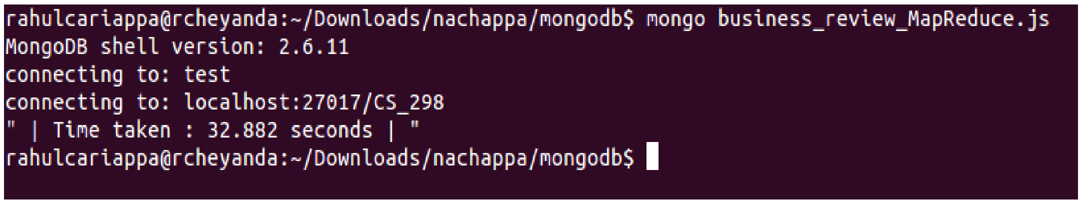
Reducer:

```
var reduceBusinessReview = function(key, values) {
   var outs={ business_id: null, name: null, full_address: null, city:null, state: null,
review_id: null, text: null}
        values.forEach(function(v){
        if(outs.business_id ==null){
            outs.business_id = v.business_id
        }
        if(outs.name ==null){
            outs.name = v.name
        }
        if(outs.full_address ==null){
            outs.full_address = v.full_address
        }
        if(outs.city ==null){
            outs.city = v.city
        }
        if(outs.state ==null){
            outs.state = v.state
        }
        if(outs.review_id ==null){
            outs.review_id = v.review_id
        }
        if(outs.text ==null){
            outs.text = v.text
        }
    });
    return outs;
};
```

In order to achieve the desired result of having aggregated fields, we run the reduce phase for review objects on the first function call and then for business objects on the second function call using the same resultant collection (mapReducedCollection):

```
db.Review_data_all.mapReduce(mapReview, reduceBusinessReview, {out: {reduce: 'mapReducedCollection'}})
db.business_data.mapReduce(mapBusiness, reduceBusinessReview, {out: {reduce: 'mapReducedCollection'}})
```

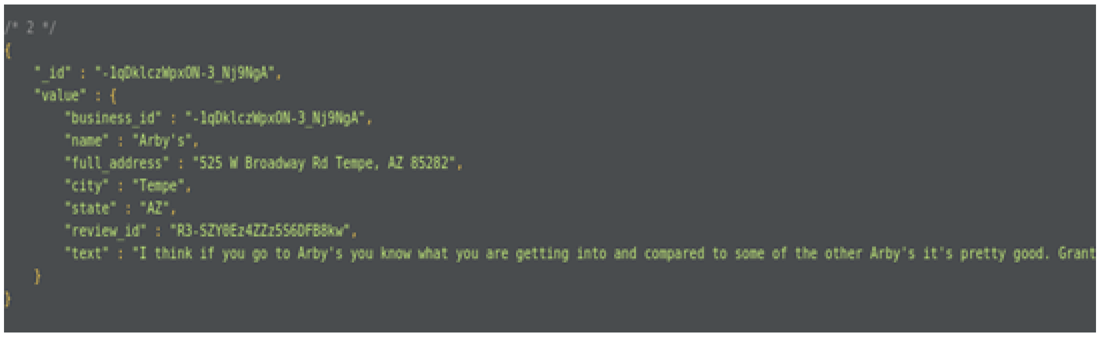
Output:



```
rahulcariappa@rcheyanda:~/Downloads/nachappa/mongodb$ mongo business_review_MapReduce.js
MongoDB shell version: 2.6.11
connecting to: test
connecting to: localhost:27017/CS_298
" | Time taken : 32.882 seconds | "
rahulcariappa@rcheyanda:~/Downloads/nachappa/mongodb$ 
```

Figure 19. Business-Review mapping MapReduce Output

Data will be stored in auxiliary collection as:



```
/* 2 */
{
    "_id" : "-1qDklczWpxON-3_Nj9NgA",
    "value" : {
        "business_id" : "-1qDklczWpxON-3_Nj9NgA",
        "name" : "Arby's",
        "full_address" : "525 W Broadway Rd Tempe, AZ 85282",
        "city" : "Tempe",
        "state" : "AZ",
        "review_id" : "R3-SZY0Ez4ZZz5S6DFB8kw",
        "text" : "I think if you go to Arby's you know what you are getting into and compared to some of the other Arby's it's pretty good. Grant
    }
}
```

Figure 20. Business-Review mapping MapReduce Data Output

4.2.3 Method 3: Data modeling using Many-to-Many relationship

In this proposed methodology, an auxiliary table to hold the keys from different parent collections will be created. The auxiliary table acts as a cache where each of the relationship between parent collections will be defined and can be used to look up to find the linking entities.

In this solution, the auxiliary table will store an array of IDs present in different collections which is being referenced by the parent collection. This will help the user to understand how many groups (entities) a collection would be linked to.

The implemented code for business and review object is as below:

Solution 4:

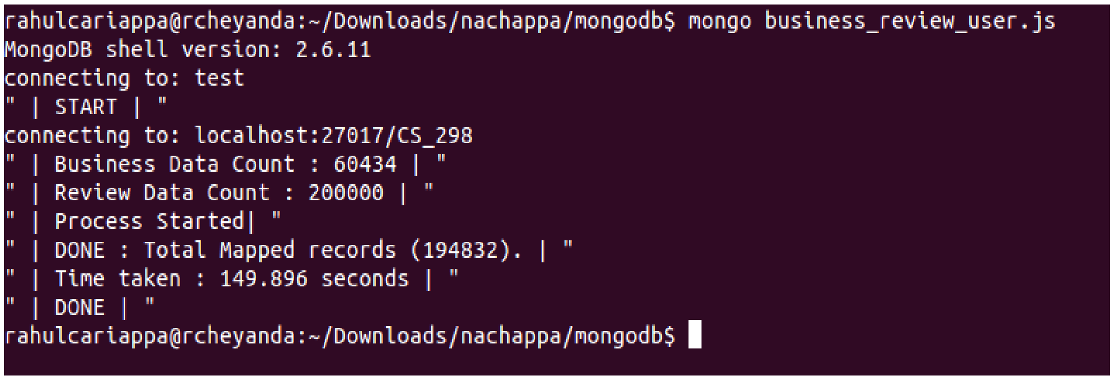
Code Snippet:

```
function propAssoc() {
    var bulkInsertOp = businessReviewData.initializeUnorderedBulkOp();
    var consolidatedDataCur = null;
    reviewData.find().addOption(16).forEach(function(reviewDataDoc) {
            business_id = (reviewDataDoc.business_id).toString();
            var consolidatedData = null;
            consolidatedDataCur = businessData.find({"business_id":business_id});
            while(consolidatedDataCur.hasNext()){
                consolidatedData = consolidatedDataCur.next();
                bulkInsertOp.find({"business_id":business_id}).upsert().update({
                  $set :{
                  "business_id" : business_id,
                  "business_name" : consolidatedData.name,
                  "business_address" : consolidatedData.full_address},
                  $addToSet: {"review_id" :reviewDataDoc.review_id}
```

```
                    });
            }
    });
        bulkInsertOp.execute();
}
propAssoc();
```

The above code creates an array field for all the review_ids associated with a single business. This helps the user to understand which business would have more reviews.

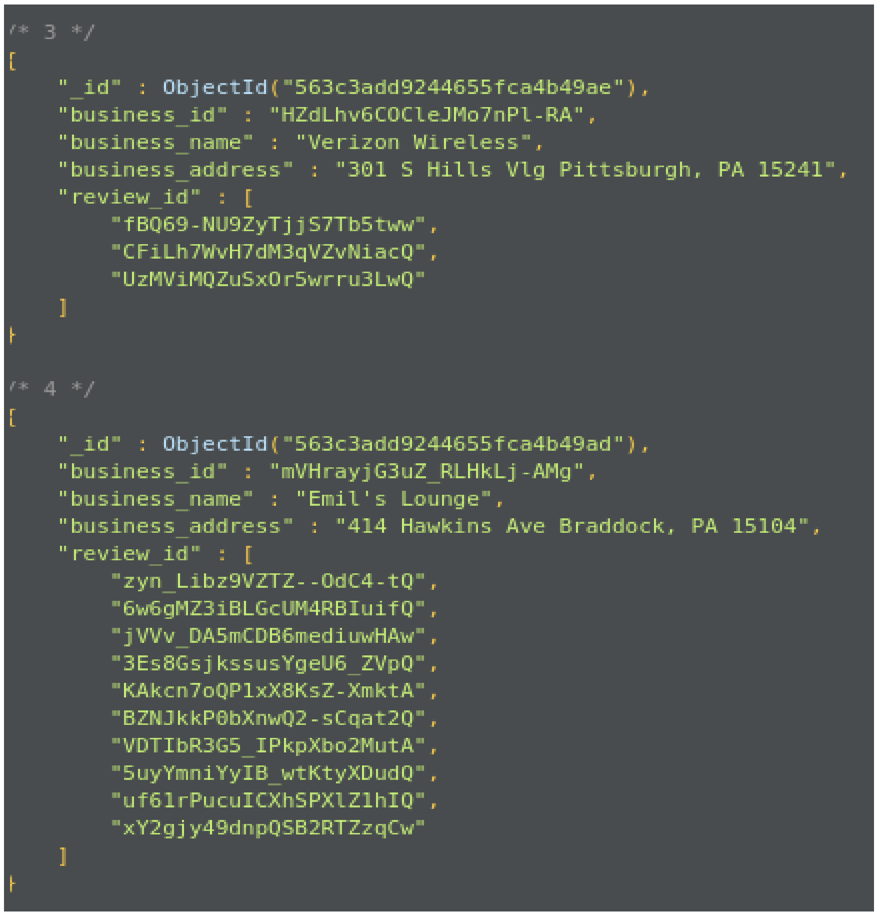
Output:



```
rahulcariappa@rcheyanda:~/Downloads/nachappa/mongodb$ mongo business_review_user.js
MongoDB shell version: 2.6.11
connecting to: test
" | START | "
connecting to: localhost:27017/CS_298
" | Business Data Count : 60434 | "
" | Review Data Count : 200000 | "
" | Process Started| "
" | DONE : Total Mapped records (194832). | "
" | Time taken : 149.896 seconds | "
" | DONE | "
rahulcariappa@rcheyanda:~/Downloads/nachappa/mongodb$
```

Figure 21. Business-Review mapping many-to-many relationship Output

Data stored in resultant collection is as below:



```
/* 3 */
{
    "_id" : ObjectId("563c3add9244655fca4b49ae"),
    "business_id" : "HZdLhv6COCleJMo7nPl-RA",
    "business_name" : "Verizon Wireless",
    "business_address" : "301 S Hills Vlg Pittsburgh, PA 15241",
    "review_id" : [
        "fBQ69-NU9ZyTjjS7Tb5tww",
        "CFiLh7WvH7dM3qVZvNiacQ",
        "UzMViMQZuSxOr5wrru3LwQ"
    ]
}

/* 4 */
{
    "_id" : ObjectId("563c3add9244655fca4b49ad"),
    "business_id" : "mVHrayjG3uZ_RLHkLj-AMg",
    "business_name" : "Emil's Lounge",
    "business_address" : "414 Hawkins Ave Braddock, PA 15104",
    "review_id" : [
        "zyn_Libz9VZTZ--OdC4-tQ",
        "6w6gMZ3iBLGcUM4RBIuifQ",
        "jVVv_DA5mCDB6mediuwHAw",
        "3Es8GsjkssusYgeU6_ZVpQ",
        "KAkcn7oQPlxX8KsZ-XmktA",
        "BZNJkkP0bXnwQ2-sCqat2Q",
        "VDTIbR3G5_IPkpXbo2MutA",
        "5uyYmniYyIB_wtKtyXDudQ",
        "uf61rPucuICXhSPXlZ1hIQ",
        "xY2gjy49dnpQSB2RTZzqCw"
    ]
}
```

Figure 22. Business-Review mapping many-to-many relationship Data Output

The same implementation can be extended to find a relationship between the business, reviews and the users who post reviews. It also tells us how many users reviewed a certain business.

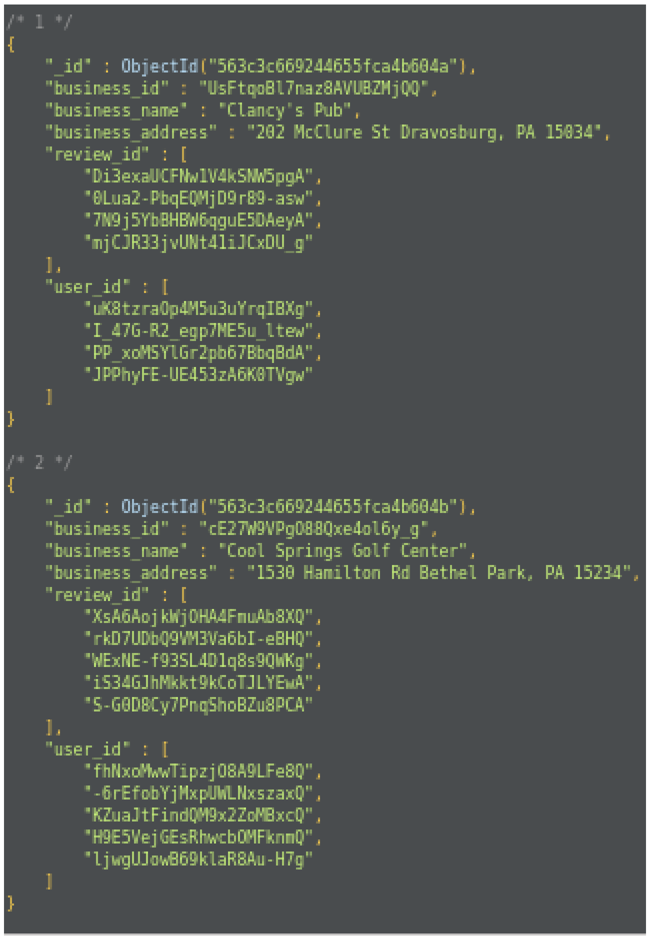
Solution 5:

Code Snippet:

```
function propAssoc() {
            var bulkInsertOp = businessReviewData.initializeUnorderedBulkOp();
            reviewData.find().addOption(16).forEach(function(reviewDataDoc) {
            business_id = (reviewDataDoc.business_id).toString();
            user_id = (reviewDataDoc.user_id).toString();
            var consolidatedData = null;
            consolidatedDataCur = businessData.find({"business_id":business_id});
             while(consolidatedDataCur.hasNext()){
                              consolidatedData = consolidatedDataCur.next();
                              var consolidatedData2 = null;
                              consolidatedDataCur2 = userData.find({"user_id":user_id});
                              while(consolidatedDataCur2.hasNext()){
                              consolidatedData2 = consolidatedDataCur2.next();

bulkInsertOp.find({"business_id":business_id}).upsert().update({
                                 $set :{
                                 "business_id" : business_id,
                                 "business_name" : consolidatedData.name,
                                 "business_address" : consolidatedData.full_address},
                                 $addToSet: {"review_id" : reviewDataDoc.review_id,
                                          "user_id" : consolidatedData2.user_id}
                                 });
                          }
                          }
            });
            bulkInsertOp.execute();
        } propAssoc();
```

Output:

Data with array fields for review and users will be represented as below**:**



```
/* 1 */
{
    "_id" : ObjectId("563c3c669244655fca4b604a"),
    "business_id" : "UsFtqoBl7naz8AVUBZMjQQ",
    "business_name" : "Clancy's Pub",
    "business_address" : "202 McClure St Dravosburg, PA 15034",
    "review_id" : [
        "Di3exaUCFNw1V4kSNW5pgA",
        "0Lua2-PbqEQMjD9r89-asw",
        "7N9j5YbBHBW6qguE5DAeyA",
        "mjCJR33jvUNt41iJCxDU_g"
    ],
    "user_id" : [
        "uK8tzraOp4M5u3uYrqIBXg",
        "I_47G-R2_egp7ME5u_ltew",
        "PP_xoMSYlGr2pb67BbqBdA",
        "JPPhyFE-UE453zA6K0TVgw"
    ]
}

/* 2 */
{
    "_id" : ObjectId("563c3c669244655fca4b604b"),
    "business_id" : "cE27W9VPgO88Qxe4ol6y_g",
    "business_name" : "Cool Springs Golf Center",
    "business_address" : "1530 Hamilton Rd Bethel Park, PA 15234",
    "review_id" : [
        "XsA6AojkWjOHA4FmuAb8XQ",
        "rkD7UDbQ9VM3Va6bI-eBHQ",
        "WExNE-f93SL4D1q8s9QWKg",
        "iS34GJhMkkt9kCoTJLYEwA",
        "S-G0D8Cy7PnqShoBZu8PCA"
    ],
    "user_id" : [
        "fhNxoMwwTipzjO8A9LFe8Q",
        "-6rEfobYjMxpUWLNxszaxQ",
        "KZuaJtFindQM9x2ZoMBxcQ",
        "H9E5VejGEsRhwcbOMFknmQ",
        "ljwgUJowB69klaR8Au-H7g"
    ]
}
```

Figure 23. Business-Review-User mapping many-to-many relationship Data Output

4.3 HBase Implementation

4.3.1 Loading data into HBase:

In order to load data into HBase tables, we chose to use Apache Phoenix on top of HBase to read and insert the input data into HBase tables. To achieve this, we first provided the structure for the table in a SQL script as below:

SQL Script:

```
CREATE TABLE IF NOT EXISTS BUSINESS_DATA (
     BUSINESS_TYPE CHAR (20) NOT NULL,
     BUSINESS_ID VARCHAR NOT NULL,
     BUSINES_NAME VARCHAR NOT NULL,
     NEIGHBORHOODS NOT NULL,
     FULL_ADDRESS VARCHAR,
     BUSINESS_CITY VARCHAR,
     BUSINESS_STATE VARCHAR,
     LATITUDE VARCHAR,
     LONGITUDE VARCHAR,
     STARS VARCHAR,
     REVIEW_COUNT INTEGER,
     OPEN_STATUS CHAR (10),
     CATEGORIES VARCHAR
     CONSTRAINT PK PRIMARY KEY (BUSINESS_ID)
);
```

The above script creates a table called BUSINESS_DATA with BUSINESS_ID as the primary key to the table.

We loaded the business data present in a file called BUSINESS_DATA.csv from the command line using **psql.py** script present in the Phoenix path: **/usr/local/phoenix-<version>/bin**

The script to load data is: **psql.py /path_to_input_file/BUSINESS_DATA.csv /path_to_sql/BUSINESS.sql**

We used the below scripts to define review and user objects as HBase tables.

SQL Script for Review Objects:

```
CREATE TABLE IF NOT EXISTS REVIEW_DATA (
     REVIEW_TYPE VARCHAR,
     USER_ID VARCHAR NOT NULL,
     BUSINES_ID VARCHAR NOT NULL,
     DATE DATE,
     REVIEW_ID VARCHAR NOT NULL,
     STARS INTEGER,
     REVIEW_TEXT VARCHAR
     CONSTRAINT PK PRIMARY KEY (REVIEW_ID)
);
```

SQL Script for User Objects:

```
CREATE TABLE IF NOT EXISTS USER_DATA (
     USER_TYPE CHAR(10),
     USER_ID VARCHAR NOT NULL,
     USER_NAME VARCHAR,
     YELPING_SINCE DATE,
     AVG_STARS VARCHAR,
     ELITE VARCHAR,
     FANS INTEGER,
     FRIENDS VARCHAR,
     REVIEW_COUNT INTEGER
     CONSTRAINT PK PRIMARY KEY (USER_ID)
);
```

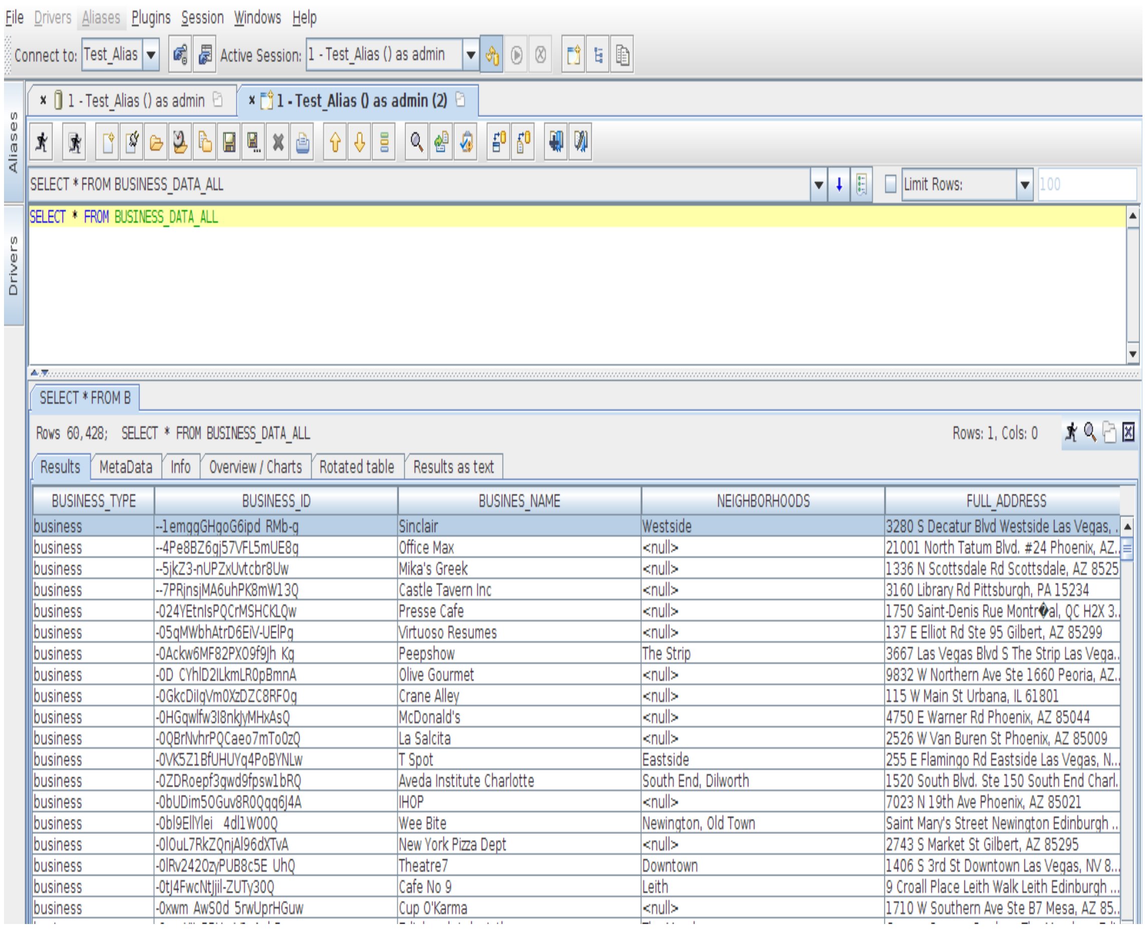
From the definition of tables above, we note that BUSINESS_ID, REVIEW_ID and USER_ID act as primary keys for Business, Review and User objects respectively.

We use SQL Squirrel Client installed on top of Apache Phoenix to query and view data. We can use the below query to check if the data is loaded correctly in our HBase tables.

Query:

Select * from BUSINESS_DATA_ALL

Business data: Record count - 60428



File Drivers Aliases Plugins Session Windows Help
Connect to: Test_Alias
Active Session: 1 - Test_Alias () as admin
Aliases
Drivers
1 - Test_Alias () as admin
1 - Test_Alias () as admin (2)
Limit Rows: 100
SELECT * FROM B
Results
MetaData
Info
Overview / Charts
Rotated table
Results as text

```
SELECT * FROM BUSINESS_DATA_ALL
```

Rows 60,428;   SELECT * FROM BUSINESS_DATA_ALL

Rows: 1, Cols: 0

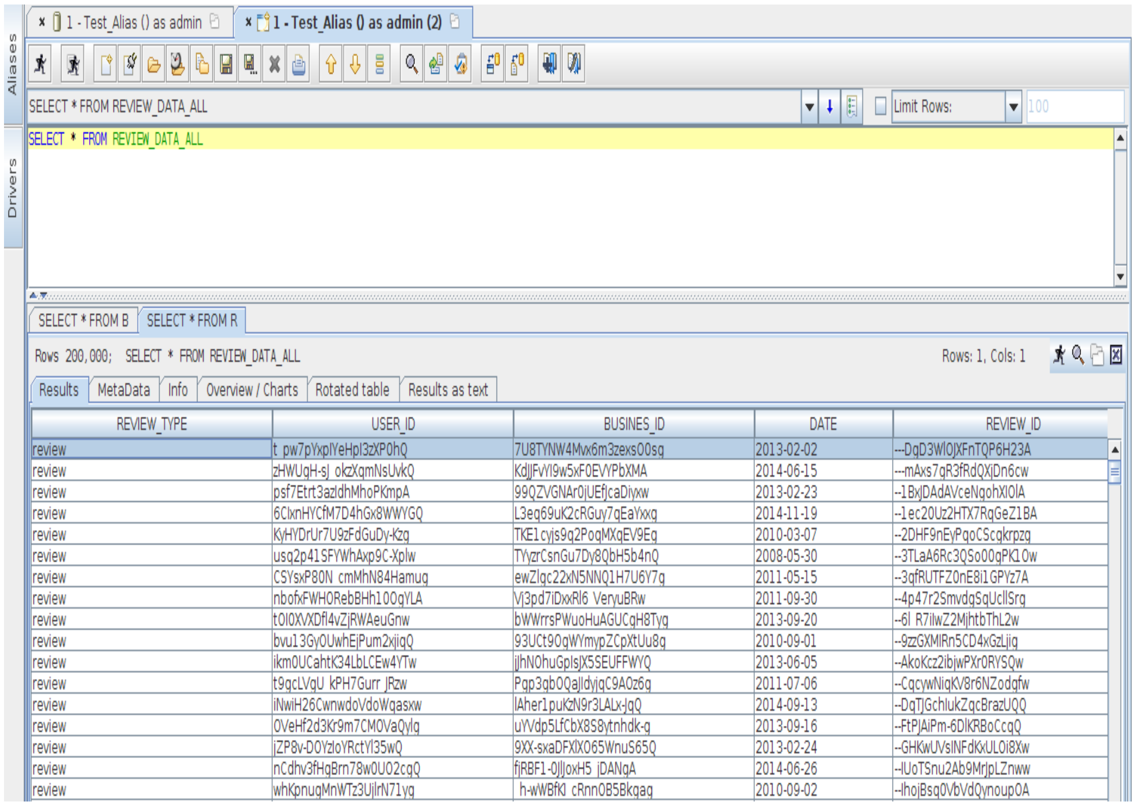
BUSINESS_TYPE	BUSINESS_ID	BUSINES_NAME	NEIGHBORHOODS	FULL_ADDRESS
business	--1emggGHgoG6ipd RMb-g	Sinclair	Westside	3280 S Decatur Blvd Westside Las Vegas, .
business	--4Pe8BZ6qj57VFL5mUE8g	Office Max	<null>	21001 North Tatum Blvd. #24 Phoenix, AZ..
business	--5jkZ3-nUPZxUvtcbr8Uw	Mika's Greek	<null>	1336 N Scottsdale Rd Scottsdale, AZ 8525
business	--7PRjnsjMA6uhPK8mW13Q	Castle Tavern Inc	<null>	3160 Library Rd Pittsburgh, PA 15234
business	-024YEtnIsPQCrMSHCKLQw	Presse Cafe	<null>	1750 Saint-Denis Rue Montr�al, QC H2X 3..
business	-05qMWbhAtrD6EiV-UElPg	Virtuoso Resumes	<null>	137 E Elliot Rd Ste 95 Gilbert, AZ 85299
business	-0Ackw6MF82PX09f9Jh Kg	Peepshow	The Strip	3667 Las Vegas Blvd S The Strip Las Vega..
business	-0D CYhlD2lLkmLR0pBmnA	Olive Gourmet	<null>	9832 W Northern Ave Ste 1660 Peoria, AZ..
business	-0GkcDiIgVm0XzDZC8RF0g	Crane Alley	<null>	115 W Main St Urbana, IL 61801
business	-0HGqwlfw3I8nkJyMHxAsQ	McDonald's	<null>	4750 E Warner Rd Phoenix, AZ 85044
business	-0QBrNvhrPQCaeo7mTo0zQ	La Salcita	<null>	2526 W Van Buren St Phoenix, AZ 85009
business	-0VK5Z1BfUHUYq4PoBYNLw	T Spot	Eastside	255 E Flamingo Rd Eastside Las Vegas, N...
business	-0ZDRoepf3qwd9fpsw1bRQ	Aveda Institute Charlotte	South End, Dilworth	1520 South Blvd. Ste 150 South End Charl.
business	-0bUDim5OGuv8R0Qqq6J4A	IHOP	<null>	7023 N 19th Ave Phoenix, AZ 85021
business	-0bl9EllYlei 4dl1W00Q	Wee Bite	Newington, Old Town	Saint Mary's Street Newington Edinburgh ..
business	-0lOuL7RkZQnjAl96dXTvA	New York Pizza Dept	<null>	2743 S Market St Gilbert, AZ 85295
business	-0lRv2420zyPUB8c5E UhQ	Theatre7	Downtown	1406 S 3rd St Downtown Las Vegas, NV 8...
business	-0tJ4FwcNtJjil-ZUTy30Q	Cafe No 9	Leith	9 Croall Place Leith Walk Leith Edinburgh ...
business	-0xwm AwS0d 5rwUprHGuw	Cup O'Karma	<null>	1710 W Southern Ave Ste B7 Mesa, AZ 85..

Figure 24. HBase Business Table Output

Query:

Select * from REVIEW_DATA_ALL

Review data: Record count – 200,000



```
SELECT * FROM REVIEW_DATA_ALL
```

Rows 200,000; SELECT * FROM REVIEW_DATA_ALL

Rows: 1, Cols: 1

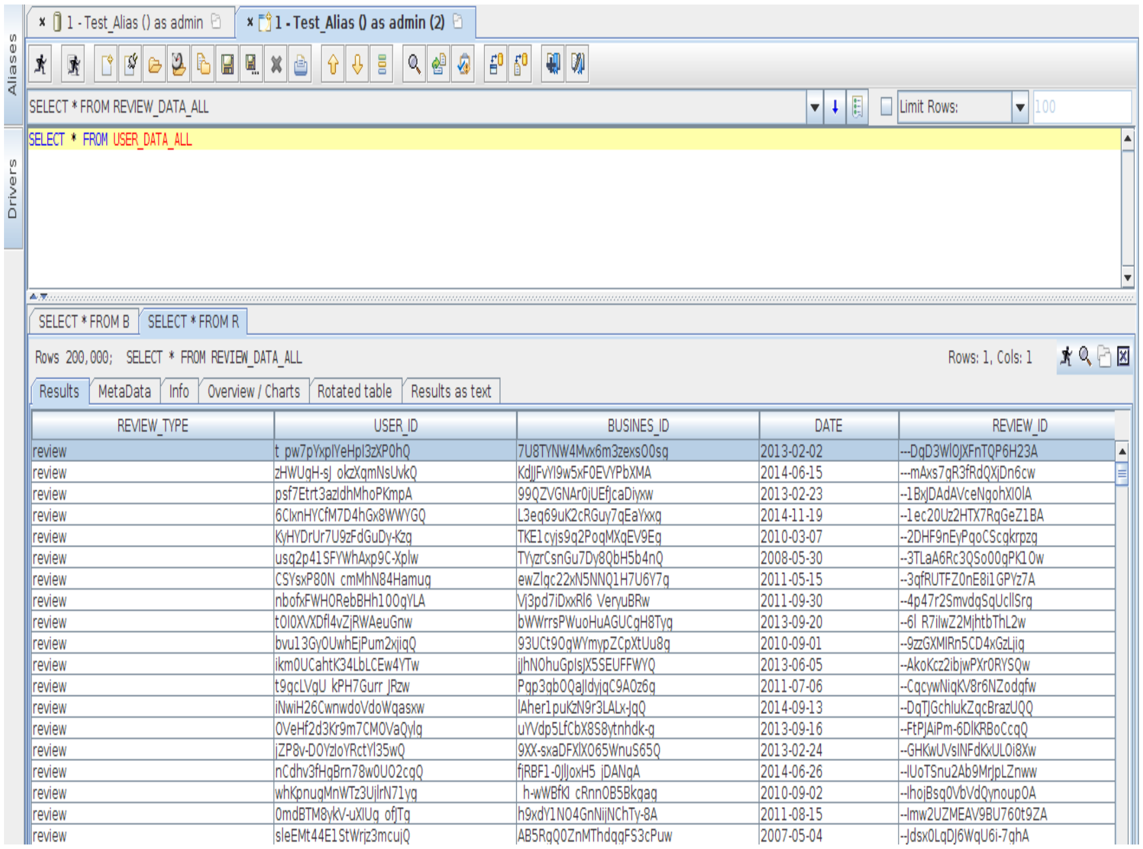
REVIEW_TYPE	USER_ID	BUSINES_ID	DATE	REVIEW_ID
review	t pw7pYxplYeHpl3zXP0hQ	7U8TYNW4Mvx6m3zexsO0sg	2013-02-02	---DgD3WlOJXFnTQP6H23A
review	zHWUgH-sJ okzXqmNsUvkQ	KdJJFvYI9w5xF0EVYPbXMA	2014-06-15	---mAxs7gR3fRdQXjDn6cw
review	psf7Etrt3azldhMhoPKmpA	99QZVGNAr0jUEfJcaDiyxw	2013-02-23	--1BxJDAdAVceNgohXlOlA
review	6ClxnHYCfM7D4hGx8WWYGQ	L3eq69uK2cRGuy7qEaYxxg	2014-11-19	--1ec20Uz2HTX7RqGeZ1BA
review	KyHYDrUr7U9zFdGuDy-Kzg	TKE1cyjs9q2PoqMXqEV9Eq	2010-03-07	--2DHF9nEyPqoCScgkrpzg
review	usq2p41SFYWhAxp9C-Xplw	TYyzrCsnGu7Dy8QbH5b4nQ	2008-05-30	--3TLaA6Rc3QSo00gPK1Ow
review	CSYsxP80N cmMhN84Hamug	ewZlgc22xN5NNQ1H7U6Y7g	2011-05-15	--3qfRUTFZ0nE8i1GPYz7A
review	nbofxFWHORebBHh10OgYLA	Vj3pd7iDxxRl6 VeryuBRw	2011-09-30	--4p47r2SmvdgSqUcllSrg
review	tOI0XVXDfl4vZjRWAeuGnw	bWWrrsPWuoHuAGUCgH8Tyg	2013-09-20	--6l R7ilwZ2MjhtbThL2w
review	bvu13GyOUwhEjPum2xjiqQ	93UCt9OgWYmypZCpXtUu8g	2010-09-01	--9zzGXMIRn5CD4xGzLjig
review	ikm0UCahtK34LbLCEw4YTw	jJhNOhuGplsJX5SEUFFWYQ	2013-06-05	--AkoKcz2ibjwPXr0RYSQw
review	t9gcLVgU kPH7Gurr JRzw	Pgp3gbOQaJldyjqC9A0z6g	2011-07-06	--CqcywNiqKV8r6NZodgfw
review	iNwiH26CwnwdoVdoWqasxw	lAher1puKzN9r3LALx-JqQ	2014-09-13	--DqTJGchlukZqcBrazUQQ
review	0VeHf2d3Kr9m7CM0VaQylg	uYVdp5LfCbX8S8ytnhdk-g	2013-09-16	--FtPJAiPm-6DlKRBoCcqQ
review	jZP8v-DOYzloYRctYl35wQ	9XX-sxaDFXlXO65WnuS65Q	2013-02-24	--GHKwUVsINFdKxULOi8Xw
review	nCdhv3fHgBrn78w0UO2cqQ	fjRBF1-0JlJoxH5 jDANgA	2014-06-26	--lUoTSnu2Ab9MrJpLZnww
review	whKpnugMnWTz3UjlrN71yg	h-wWBfKl cRnnOB5Bkgag	2010-09-02	--lhojBsq0VbVdQynoup0A

Figure 25. HBase Review Table Output

Query:

Select * from USER_DATA_ALL

User data: Record count – 366,715



1 - Test_Alias () as admin

1 - Test_Alias () as admin (2)

SELECT * FROM REVIEW_DATA_ALL

Limit Rows: 100

```
SELECT * FROM USER_DATA_ALL
```

SELECT * FROM B

SELECT * FROM R

Rows 200,000; SELECT * FROM REVIEW_DATA_ALL

Rows: 1, Cols: 1

Results | MetaData | Info | Overview / Charts | Rotated table | Results as text

REVIEW_TYPE	USER_ID	BUSINES_ID	DATE	REVIEW_ID
review	t pw7pYxplYeHpl3zXP0hQ	7U8TYNW4Mvx6m3zexs00sg	2013-02-02	---DqD3WlOJXFnTQP6H23A
review	zHWUgH-sJ okzXqmNsUvkQ	KdJJFvYI9w5xF0EVYPbXMA	2014-06-15	---mAxs7gR3fRdQXjDn6cw
review	psf7Etrt3azldhMhoPKmpA	99QZVGNAr0jUEfJcaDiyxw	2013-02-23	--1BxJDAdAVceNgohXIOlA
review	6ClxnHYCfM7D4hGx8WWYGQ	L3eq69uK2cRGuy7qEaYxxg	2014-11-19	--1ec20Uz2HTX7RqGeZ1BA
review	KyHYDrUr7U9zFdGuDy-Kzg	TKE1cyjs9q2PoqMXqEV9Eg	2010-03-07	--2DHF9nEyPqoCScgkrpzg
review	usq2p41SFYWhAxp9C-Xplw	TYyzrCsnGu7Dy8QbH5b4nQ	2008-05-30	--3TLaA6Rc3QSo00gPK10w
review	CSYsxP80N cmMhN84Hamug	ewZlgc22xN5NNQ1H7U6Y7g	2011-05-15	--3qfRUTFZ0nE8i1GPYz7A
review	nbofxFWH0RebBHh10OqYLA	Vj3pd7iDxxRl6 VeryuBRw	2011-09-30	--4p47r2SmvdgSqUcllSrg
review	tOI0XVXDfl4vZjRWAeuGnw	bWWrrsPWuoHuAGUCgH8Tyg	2013-09-20	--6l R7ilwZ2MjhtbThL2w
review	bvu13GyOUwhEjPum2xjiqQ	93UCt9OgWYmypZCpXtUu8g	2010-09-01	--9zzGXMIRn5CD4xGzLjig
review	ikm0UCahtK34LbLCEw4YTw	jJhN0huGpIsJX5SEUFFWYQ	2013-06-05	--AkoKcz2ibjwPXr0RYSQw
review	t9gcLVgU kPH7Gurr JRzw	Pgp3gbOQaJIdyjqC9A0z6g	2011-07-06	--CqcywNiqKV8r6NZodgfw
review	iNwiH26CwnwdoVdoWqasxw	lAher1puKzN9r3LALx-JqQ	2014-09-13	--DqTJGchlukZqcBrazUQQ
review	0VeHf2d3Kr9m7CM0VaQylg	uYVdp5LfCbX8S8ytnhdk-g	2013-09-16	--FtPJAiPm-6DlKRBoCcqQ
review	jZP8v-DOYzloYRctYl35wQ	9XX-sxaDFXlX065WnuS65Q	2013-02-24	--GHKwUVsINFdKxULOi8Xw
review	nCdhv3fHgBrn78w0UO2cqQ	fjRBF1-0JlJoxH5 jDANgA	2014-06-26	--IUoTSnu2Ab9MrJpLZnww
review	whKpnugMnWTz3UjlrN71yg	h-wWBfKl cRnn0B5Bkgag	2010-09-02	--IhojBsq0VbVdQynoup0A
review	0mdBTM8ykV-uXlUg ofJTg	h9xdY1NO4GnNijNChTy-8A	2011-08-15	--Imw2UZMEAV9BU760t9ZA
review	sleEMt44E1StWrjz3mcujQ	AB5RgQ0ZnMThdqgFS3cPuw	2007-05-04	--Jdsx0LqDJ6WqU6i-7ghA

Figure 26. HBase User Table Output

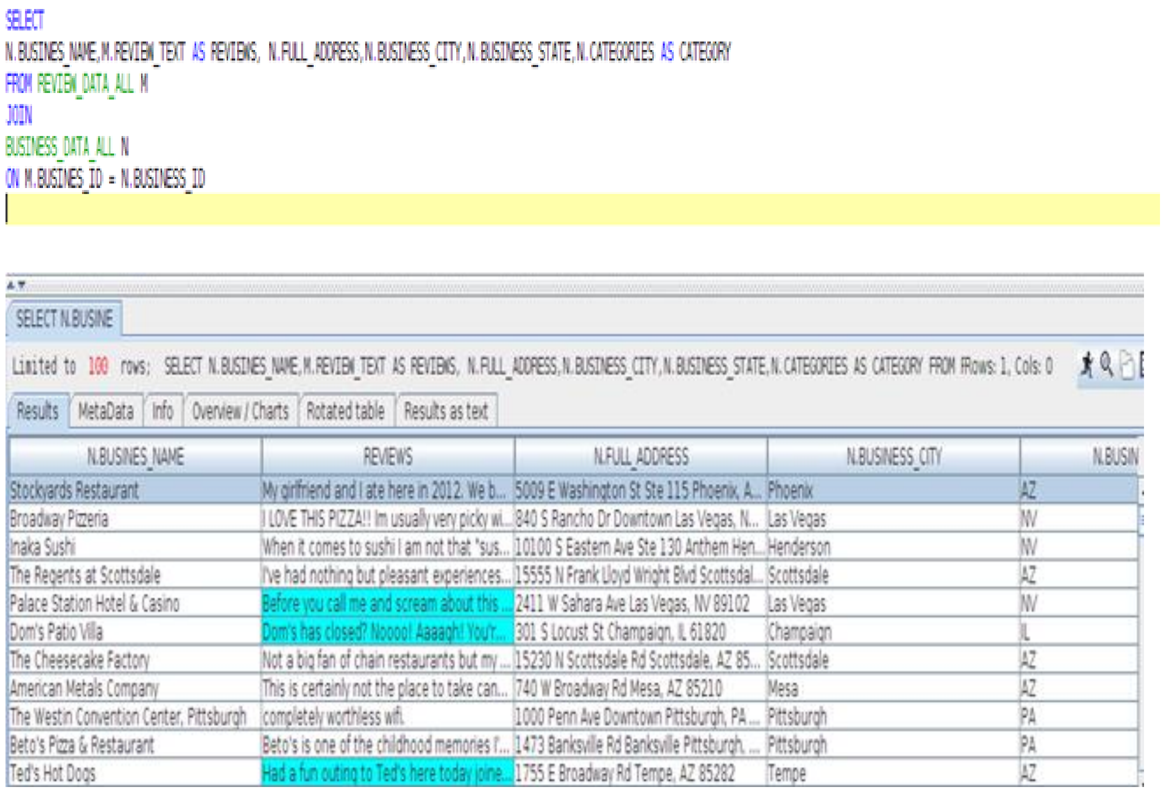
4.3.2 HBase Joins and Results:

Let’s consider business and review objects. We can find the relationship between these two objects by using the unique identifier: business_id and fetching information for reviews that each of the business has obtained. The join functionality supported by Apache Phoenix can be used to achieve this. The following code does the job:

Code Snippet:

Query:

```
SELECT N.BUSINES_NAME, M.REVIEW_TEXT AS REVIEWS, N.FULL_ADDRESS,
N.BUSINESS_CITY, N.BUSINESS_STATE, N.CATEGORIES AS CATEGORY
FROM REVIEW_DATA_ALL M
JOIN
BUSINESS_DATA_ALL N
ON M.BUSINES_ID = N.BUSINESS_ID
```



```
SELECT
N.BUSINES_NAME,M.REVIEW_TEXT AS REVIEWS, N.FULL_ADDRESS,N.BUSINESS_CITY,N.BUSINESS_STATE,N.CATEGORIES AS CATEGORY
FROM REVIEW_DATA_ALL M
JOIN
BUSINESS_DATA_ALL N
ON M.BUSINES_ID = N.BUSINESS_ID
```

SELECT N.BUSINE

Limited to 100 rows; SELECT N.BUSINES_NAME,M.REVIEW_TEXT AS REVIEWS, N.FULL_ADDRESS,N.BUSINESS_CITY,N.BUSINESS_STATE,N.CATEGORIES AS CATEGORY FROM #Rows: 1, Cols: 0

Results | MetaData | Info | Overview / Charts | Rotated table | Results as text

N.BUSINES_NAME	REVIEWS	N.FULL_ADDRESS	N.BUSINESS_CITY	N.BUSIN
Stockyards Restaurant	My girlfriend and I ate here in 2012. We b...	5009 E Washington St Ste 115 Phoenix, A...	Phoenix	AZ
Broadway Pizzeria	I LOVE THIS PIZZA!! Im usually very picky wi...	840 S Rancho Dr Downtown Las Vegas, N...	Las Vegas	NV
Inaka Sushi	When it comes to sushi I am not that "sus...	10100 S Eastern Ave Ste 130 Anthem Hen...	Henderson	NV
The Regents at Scottsdale	I've had nothing but pleasant experiences...	15555 N Frank Lloyd Wright Blvd Scottsdal...	Scottsdale	AZ
Palace Station Hotel & Casino	Before you call me and scream about this ...	2411 W Sahara Ave Las Vegas, NV 89102	Las Vegas	NV
Dom's Patio Villa	Dom's has closed? Noooo! Aaaagh! You'r...	301 S Locust St Champaign, IL 61820	Champaign	IL
The Cheesecake Factory	Not a big fan of chain restaurants but my ...	15230 N Scottsdale Rd Scottsdale, AZ 85...	Scottsdale	AZ
American Metals Company	This is certainly not the place to take can...	740 W Broadway Rd Mesa, AZ 85210	Mesa	AZ
The Westin Convention Center, Pittsburgh	completely worthless wifi.	1000 Penn Ave Downtown Pittsburgh, PA ...	Pittsburgh	PA
Beto's Pizza & Restaurant	Beto's is one of the childhood memories I'...	1473 Banksville Rd Banksville Pittsburgh, ...	Pittsburgh	PA
Ted's Hot Dogs	Had a fun outing to Ted's here today joine...	1755 E Broadway Rd Tempe, AZ 85282	Tempe	AZ

Figure 27. HBase Business-Review Join Output

Time taken for execution: 6.69 seconds

Record count – 197,393

The implementation can be extended to join three HBase tables. Business, Review, and User data can be joined using business_id and user_id to get data across all the three tables. The implementation of the same is as below:

Query:

SELECT N.BUSINESS_NAME, N.FULL_ADDRESS, N.BUSINESS_CITY, N.BUSINESS_STATE, N.CATEGORIES AS CATEGORY, M.REVIEW_TEXT AS REVIEWS, O.USER_NAME

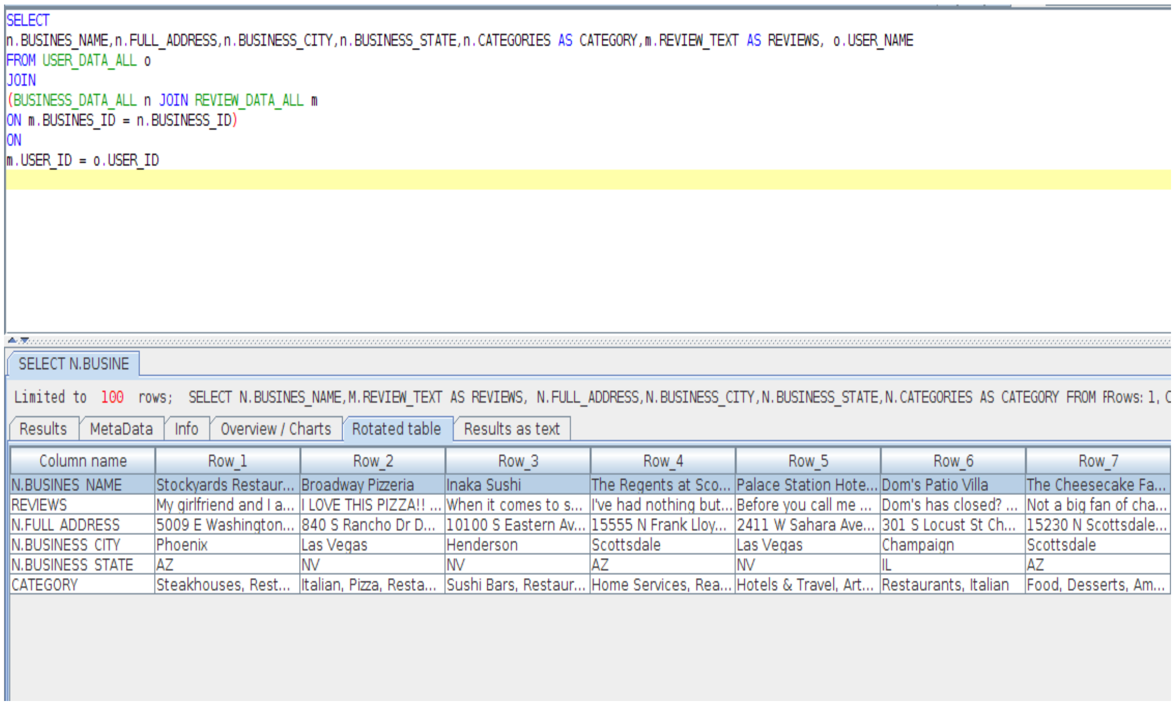
FROM USER_DATA_ALL O

JOIN

(BUSINESS_DATA_ALL N JOIN REVIEW_DATA_ALL M

ON M.BUSINES_ID = N.BUSINESS_ID)

ON M.USER_ID = O.USER_ID



```
SELECT
n.BUSINES_NAME,n.FULL_ADDRESS,n.BUSINESS_CITY,n.BUSINESS_STATE,n.CATEGORIES AS CATEGORY,m.REVIEW_TEXT AS REVIEWS, o.USER_NAME
FROM USER_DATA_ALL o
JOIN
(BUSINESS_DATA_ALL n JOIN REVIEW_DATA_ALL m
ON m.BUSINES_ID = n.BUSINESS_ID)
ON
m.USER_ID = o.USER_ID
```

```
Limited to 100 rows;  SELECT N.BUSINES_NAME,M.REVIEW_TEXT AS REVIEWS, N.FULL_ADDRESS,N.BUSINESS_CITY,N.BUSINESS_STATE,N.CATEGORIES AS CATEGORY FROM FRows: 1, C
```

Column name	Row_1	Row_2	Row_3	Row_4	Row_5	Row_6	Row_7
N.BUSINES_NAME	Stockyards Restaur...	Broadway Pizzeria	Inaka Sushi	The Regents at Sco...	Palace Station Hote...	Dom's Patio Villa	The Cheesecake Fa...
REVIEWS	My girlfriend and I a...	I LOVE THIS PIZZA!! ...	When it comes to s...	I've had nothing but...	Before you call me ...	Dom's has closed? ...	Not a big fan of cha...
N.FULL_ADDRESS	5009 E Washington...	840 S Rancho Dr D...	10100 S Eastern Av...	15555 N Frank Lloy...	2411 W Sahara Ave...	301 S Locust St Ch...	15230 N Scottsdale...
N.BUSINESS_CITY	Phoenix	Las Vegas	Henderson	Scottsdale	Las Vegas	Champaign	Scottsdale
N.BUSINESS_STATE	AZ	NV	NV	AZ	NV	IL	AZ
CATEGORY	Steakhouses, Rest...	Italian, Pizza, Resta...	Sushi Bars, Restaur...	Home Services, Rea...	Hotels & Travel, Art...	Restaurants, Italian	Food, Desserts, Am...

Figure 28. HBase Business-Review-User Join Output

Time taken for execution: 13.43 seconds

Record count: 197,393

4.4 Apache SOLR Implementation

We use Apache SOLR to index our documents into the SOLR cluster which is setup locally and use the enterprise search server to retrieve the entities from the documents.
To perform join on our documents, we first index each document into the cluster. We create a schema to index each document using the following command:

```
sudo su - solr -c "/opt/solr/bin/solr create -c schema_name -n data_driven_schema_configs"
```

Once the schema is created, we can index the documents using the command:

```
bin/post –c schema_name docs/business_document.csv
```

We can check if the document is indexed by checking the localhost with port 8983. This is the port when Apache SOLR cluster is up and running.

After all the documents are indexed into the SOLR cluster, we can perform the JOIN operation on them using the condition:

```
!join+fromIndex=fromCollection+from=id+to=id_to_be_joined
```

Since Apache SOLR retrieved target entities at almost NRT (Near Real Time), it could be used effectively to retrieve entities from a single collection which match the specified JOIN criteria.

The inclusion of this solution will help us determine the performance of NoSQL technologies against advanced information retrieval techniques used in search engine.

CHAPTER 5

5 PERFORMANCE MEASURE

This section compares and analyzes the performance of all the methodologies and solutions discussed so far. The performance of MongoDB and HBase for different proposed solutions is consolidated in the below table:

Table 2. Performance Comparison

Solution	**Method Description**	**Without Indexing (Time - Record Count)**	**With Indexing (Time - Record Count)**
1 Business_review.js	This method joins business and review objects using business_id explained in solution 1.	42.80 sec - 500	0.519 sec - 500
		101.83 sec - 1000	0.919 sec - 1000
		166.21 sec - 1500	1.374 sec - 1500
			43.575 sec – 50000
			168.764 sec – 200000
2 Business_review.js	This method joins business and review objects using business_id with optimized solution using cursors	20. 76 sec - 500	0.283 sec - 500
		41.88 sec - 1000	0.435 sec - 1000
		64.73 - 1500	0.689 sec - 1500
			20.714 sec – 50000
			90.362 sec – 200000
	Using HashMap to join business and review objects		**3.21 sec - 50000**
			10.2 sec – 200000
2 Business_review_user.js	This method joins business, review and user objects using business_id and user_id.	119.6 sec - 500	0.426 sec - 500
		305.30 sec - 1000	0.637 sec - 1000

		418.95 sec - 1500	1.068 sec - 1500
			34.449 sec – 50000
			132.264 sec – 200000
	Using Hash Map to join business, review and user objects		**10.8 sec - 50000**
			21.2 sec – 200000
3 Business_review_MapReduce.js	This method uses MapReduce to map business and review objects	70.289 sec	55.194 sec
4 Business_review_many-to-many.js	This method maps business and review objects for many-many relationship data model.	22.98 sec - 500	0.34 sec - 500
		48.57 sec - 1000	0.548 sec - 1000
		59.952 sec - 1500	0.79sec - 1500
			55.834 sec – 50000
			149.896 – 200000
4 Business_review_user_many-to-many.js	This method extends the solution 4 to map business, review and user objects using many-many relationship data model.	118.82 sec - 500	0.502 sec - 500
		235.11 sec - 1000	0.762 sec - 1000
		356.06 sec - 1500	1.202 sec - 1500
			135.474 sec – 50000
			160.23 sec – 200000
HBase query1 Query for two tables	**This query joins Business and Review HBase tables using Apache Phoenix on top of HBase**		**6.69 sec - 20000**
HBase query2 Query for three tables	**This query joins Business, Review and User HBase tables using Apache Phoenix on top of HBase**		**13.43 - 20000**

We observe that the performance of Apache Phoenix on top of HBase is faster followed by the performance of HashMap solution for MongoDB. Plotting the values on the graph would help us analyze the performance better. The graph for performance measure is as below:

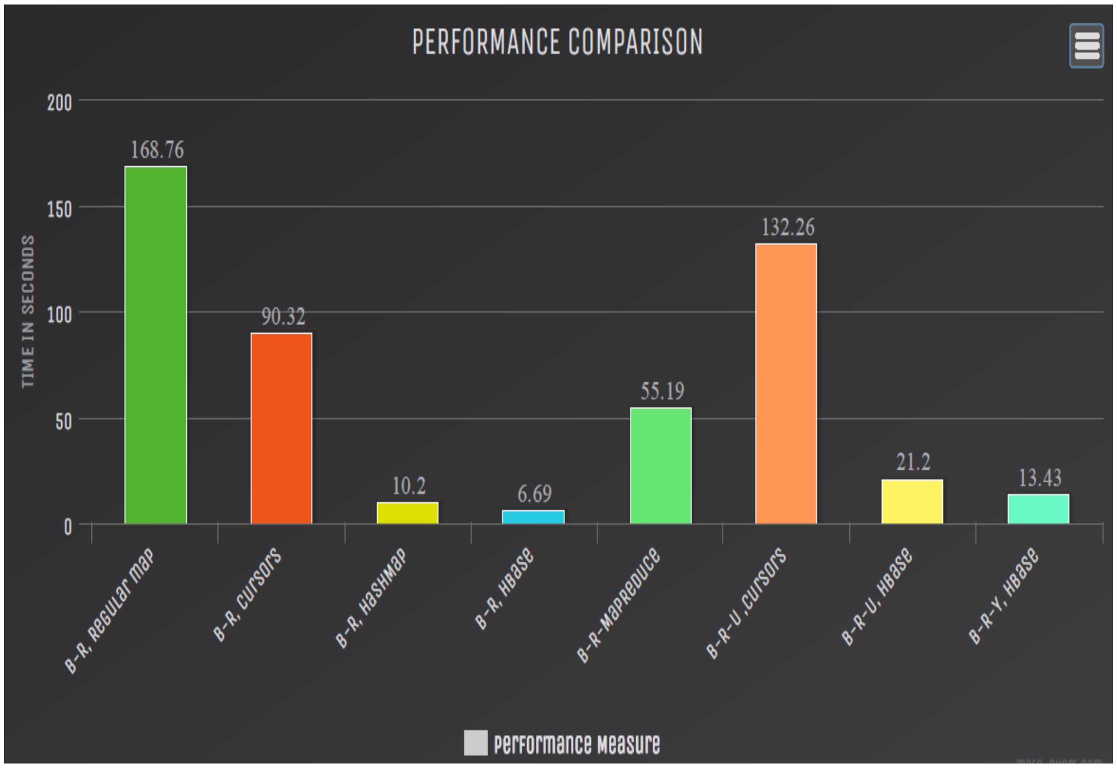


Figure 29. Perfomance Measure

In Figure 29, the terms on the X-axis indicate different implementations for join operation and on the Y-axis we have the running time (in seconds) for a method. The abbreviation B, R and U stands for Business, Review and User objects being associated with the implementation. We observe that B-R, Regular Map solutions take longest as we use an O(n^2) solution but as we optimize the solution using HashMap technique, the performance is vastly improved and the running time is 16x lesser than the regular map method. This is because of the in-memory computation performed by HashMap which uses O(n) time complexity to achieve the desired result.

We also observe that HBase join using Apache Phoenix provides a faster solution compared to HashMap solution for MongoDB.

The reason for this better performance is due to the fact that Apache Phoenix follows a Push-Down approach and parallelizes queries based on stats. Push Down is a technique where a part of the query is taken and pushed all the way down into the servers, so it actually executes on the server where the data resides. Also, Phoenix takes the queries and compiles it into a series of native HBase scans, executes and then orchestrates those scans and combines the results and returns it back to the result set.

CHAPTER 6

6 CONCLUSION

We performed several experiments starting from a small volume of data and using up to 500,000 records to be mapped to different entities. The solutions for both HBase and MongoDB were optimized to improve the performance of join operations.

From the experiments result, we see that the performance of Apache Phoenix is better as it uses the Push-down concept and scans the region servers for the data to be retrieved by the query. Also, it is noted that the performance of MongoDB can be enhanced by the use of HashMap where all the processing happens in-memory, reducing computational cost to a time complexity of O(n). The results obtained from these experiments are impressive as the solution is optimized to achieve join operation on huge volume of data. We can draw a conclusion that performing entity and relationship queries on NoSQL databases like MongoDB and HBase is efficient. Also lookup for a huge volume of data is executed faster which makes NoSQL an optimal choice for these operations. Furthermore, for future works we can extend the solution to different NoSQL technologies and measure the performance starting with few gigabytes of data. We can apply the optimization techniques developed in this thesis to other NoSQL technologies and measure their improvement.

LIST OF REFERENCES

[1] Pramod Sadalage, NoSQL Databases: An Overview, 2014
https://www.thoughtworks.com/insights/blog/nosql-databases-overview

[2] Jinbao Zhu, Data Modeling for Big Data, 2013
http://www.ca.com/us/~/media/files/articles/ca-technology-exchange/data-modeling-for-big-data-zhu-wang.aspx

[3] Dr. Fabio Fumarola, Document Oriented Databases, 2015
http://www.slideshare.net/fabiofumarola1/9-document-oriented-databases

[4] Comparing document-oriented and relational data, Couchbase Server Developer Guide, 2015
http://docs.couchbase.com/developer/dev-guide-3.0/compare-docs-vs-relational.html

[5] Carol McDonald, An In-Depth look at the HBase architecture, 2015
https://www.mapr.com/blog/in-depth-look-hbase-architecture

[6] Schemaless data modeling, Couchbase Server Developer Guide, 2015
http://docs.couchbase.com/developer/dev-guide-3.0/schemaless.html

[7] Steven Haines, Introduction to HBase, the NoSQL Database for Hadoop, 2014
http://www.informit.com/articles/article.aspx?p=2253412

[8] Sample storage documents, Couchbase Server Developer Guide, 2015
http://docs.couchbase.com/developer/dev-guide-3.0/sample-docs.html

[9] Archana Changale, Installing Apache HBase on Ubuntu for Standalone mode
https://archanaschangale.wordpress.com/2013/08/29/installing-apache-hbase-on-ubuntu-for-standalone-mode/#comments

[10] SQL Squirrel Client, 2015
http://squirrel-sql.sourceforge.net/

[11] Yelp Academic Dataset, 2015
https://www.yelp.com/academic_dataset

[12] Apache Phoenix Installation, Apache Phoenix, 2015
https://phoenix.apache.org/installation.html

[13] Using Apache Phoenix on HBase, MapR Documents, 2015
http://doc.mapr.com/display/MapR/Using+Apache+Phoenix+on+HBase

[14] Using JSON documents, Couchbase Server Developer Guide, 2015
http://docs.couchbase.com/developer/dev-guide-3.0/using-json-docs.html

[15] Modeling documents for retrieval, Couchbase Server Developer Guide, 2015
http://docs.couchbase.com/developer/dev-guide-3.0/model-docs-retrieval.html

[16] Install MongoDB on Ubuntu, MongoDB docs, 2015
https://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

[17] Apache HBase Reference Guide, Apache HBase, 2015
http://hbase.apache.org/book.html

[18] Safari, Data Storage for Analysis: Relational Databases, Big Data, and Other Options, 2015
https://www.safaribooksonline.com/library/view/network-security-through/9781449357894/ch04.html

[19] Luke P.Issac, SQL vs NoSQL Database differences explained, 2014
http://www.thegeekstuff.com/2014/01/sql-vs-nosql-db/

[20] Philip Shon, Apache HBase explained, 2014
https://www.credera.com/blog/technology-insights/java/apache-hbase-explained-5-minutes-less/