

Spring 5-22-2017

Reducing Query Latency for Information Retrieval

Swapnil Satish Kamble
San Jose State University

Follow this and additional works at: http://scholarworks.sjsu.edu/etd_projects



Part of the [Databases and Information Systems Commons](#)

Recommended Citation

Kamble, Swapnil Satish, "Reducing Query Latency for Information Retrieval" (2017). *Master's Projects*. 514.
http://scholarworks.sjsu.edu/etd_projects/514

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Reducing Query Latency for Information Retrieval

A Project Report

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

Swapnil Satish Kamble

May 2017

The Designated Project Committee Approves the Project Titled

Reducing Query Latency for Information Retrieval

By

Swapnil Satish Kamble

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2017

Dr. Robert Chun	Department of Computer Science
-----------------	--------------------------------

Mr. James Casaletto	Department of Computer Science
---------------------	--------------------------------

Dr. Thomas Austin	Department of Computer Science
-------------------	--------------------------------

ABSTRACT

As the world is moving towards Big Data, NoSQL (Not only SQL) databases are gaining much more popularity. Among the other advantages of NoSQL databases, one of their key advantage is that they facilitate faster retrieval for huge volumes of data, as compared to traditional relational databases. This project deals with one such popular NoSQL database, Apache HBase. It performs quite efficiently in cases of retrieving information using the rowkey (similar to a primary key in a SQL database). But, in cases where one needs to get information based on non-rowkey columns, the response latency is higher than what we observe in the previous case. This project discusses an approach which aims towards decreasing this latency. It also compares the performance of the existing approach and the proposed approach for various scenarios.

ACKNOWLEDGEMENTS

I am very thankful to my advisor Dr. Robert Chun and committee members Mr. James Casaletto and Dr. Thomas Austin for constantly supporting me throughout the Master's project and helping me focus on the right path to complete the project. Their insights on various topics has helped me learn many new things while implementing my project. It would not have been possible to complete this project without their guidance. Last but not the least, I would like to thank all my family members and friends for always supporting me and believing in me.

TABLE OF CONTENTS

1. Introduction.....	9
1.1. Importance.....	9
1.2. Problem Description.....	9
2. Background.....	11
2.1. Apache HBase.....	11
2.2. Elasticsearch.....	13
3. Related Work.....	16
4. Dataset	22
5. Approach.....	23
5.1. Overview.....	23
5.2. Technical Details.....	28
6. Experiments.....	41
6.1. Hardware Configuration.....	41
6.2. Experiment 1.....	41
6.3. Experiment 2.....	43
7. Results.....	45
8. Conclusion.....	49
9. Future Work.....	50
10. References.....	52
11. Appendix.....	55

11.1.	Code for secondary indexing from HBase to Elasticsearch.....	55
11.2.	Code for Data Retrieval from HBase.....	57

LIST OF TABLES

1. Sample Data.....9

2. Sample entry in Index Table.....25

3. Data Distribution for Experiment 1.....42

4. Data Distribution for Experiment 2.....44

5. Consolidated Results.....45

LIST OF FIGURES

1. HBase Architecture.....	12
2. Elasticsearch Architecture.....	15
3. Snapshot of Data.....	22
4. Data Ingestion.....	24
5. Data Retrieval (Traditional HBase approach).....	26
6. Data Retrieval (Our approach).....	27
7. Using ImportTSV tool.....	29
8. Command for secondary indexing of data.....	30
9. Connecting to HBase.....	31
10. Getting data from HBase.....	32
11. Connecting to Elasticsearch.....	33
12. A secondary index entry in Elasticsearch.....	33
13. Index HBase data to Elasticsearch.....	35
14. Command for retrieval of data.....	37
15. Regular search in Elasticsearch.....	37
16. Searching using Scroll API in Elasticsearch.....	39
17. Individual Gets from HBase for rowkeys retrieved from Elasticsearch.....	40
18. Comparison of Results (First few distributions).....	46
19. Comparison of Results (Remaining distributions).....	47

1. INTRODUCTION

1.1. Importance

For the last several years, there has been a considerable challenge in the field of data storage, web applications and services handling a huge amount of data, which require low latency solutions. NoSQL systems have the capability of scaling to the needs of such applications. HBase is one of the semi-structured distributed Key/Value stores, which is an example of NoSQL. Many Internet companies like Twitter and Facebook have used this as it is an open source solution which is highly scalable. Unfortunately, this database system still has many problems which need to be solved. This project aims at solving one such shortcoming in HBase, the details of which we will see in the following section.

1.2. Problem Description

Consider the following HBase table containing students' data:

ID	details: fname	details: lname	details: sex
1	Swapnil	Kamble	Male
2	Swathi	Nambiar	Female
3	Ansen	Mathew	Male
4	Shivika	Sodhi	Female
5	Akshay	Mangudkar	Male

TABLE I
SAMPLE DATA

The above table contains "ID" as the rowkey and a column family named "details". The "details" column family has columns "fname" (first name), "lname" (last name) and

“sex”. Apache HBase has a primary key (rowkey) column that stores unique values in sorted order. A query that contains filters on the rowkey can leverage the sortedness of the data by doing skip-scan and provide good performance. In the current version, the rowkey is the only field that is indexed, which fits the common pattern of queries based on the rowkey. Thus, in the above example, if we want to fetch data from the table based on the value of “ID”, the retrieval would be pretty fast.

But, there can be use cases where the queries can have filters on non-rowkey columns. For example, in the above table there can be a query asking for all records where sex = Male. Here, “sex” is a non-rowkey column. For these kind of queries, HBase needs to do a full table scan. For the above example, it would not make much of a difference. But if one has millions/billions of records in the table, this can cause a huge overhead and thus result in a high latency.

In this project, we are going to solve this problem and propose a solution which can avoid a full table scan and reduce the response latency so that the user does not have to wait for a very long time in getting the response. We aim to accomplish this by using an alternative approach, the details of which will be discussed in further parts of the report.

2. BACKGROUND

The solution we are going to propose would involve a combination of various ecosystems and concepts, which we need to understand before proceeding ahead. This section would give a brief overview of each of the components which would be a part of our architecture:

2.1. Apache HBase

Apache HBase is an open source NoSQL database that enables users to access their huge datasets in realtime. HBase is highly scalable and is suitable for large datasets having rows in the range of millions to billions. It can also combine data sources having a variety of different structures and schemas. It is integrated with Hadoop and works easily along with other data access ecosystems through YARN (Yet Another Resource Negotiator – a cluster management tool). It is used by enterprises for scenarios requiring real-time analysis for end user applications.

HBase is widely used in the development of projects. It is a column-oriented key - value store and gains its popularity because of its integration with Hadoop and Hadoop Distributed File System. It is suitable for scenarios that require faster read and write operations on large datasets demanding high throughput as well as low input and output latency.

HBase is capable of handling increases in load by the addition of more server nodes. It provides optimal performance in cases where consistency is critical by giving leverage to developers having SQL expertise through a modern distributed system.

The data model of HBase stores semi-structured data, which can have different data types, column size and field size. Due to such a layout, data partitioning and distribution can be done easily across the cluster. It has the following logical components:

- HBase Tables – It is a logical collection of rows stored in individual partitions called Regions.
- HBase Row - A record holding data in a table.
- RowKey -The identifier of every entry in an HBase table, much like a primary key in relational databases.
- Columns -Any number of fields can be stored for each rowkey.
- Column Family - Column families refers to the grouping of data present in rows.

Let us have a look at the HBase architecture as seen in [9]:

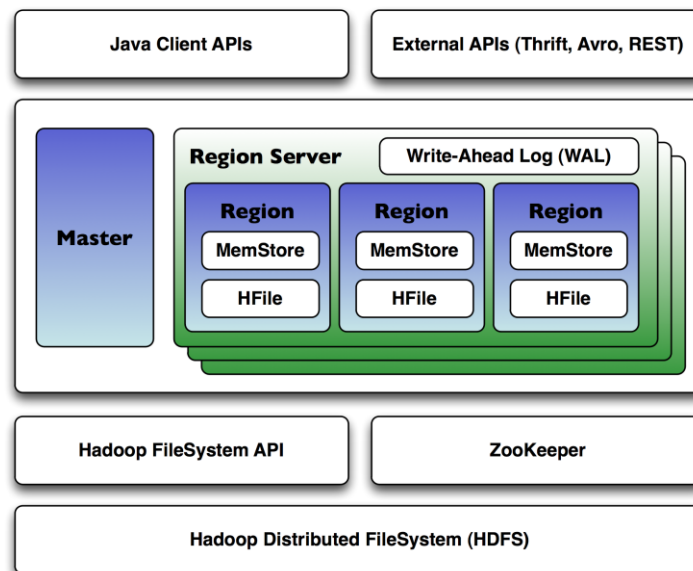


Fig. 1. HBase Architecture [9].

It consists mainly of 4 components:

- HMaster: It is used for monitoring all Region Servers in the cluster and serves as an interface for all metadata changes.
- HRegionserver: It serves and manages regions/data which are present in a distributed cluster. The region servers run on Data Nodes present in the Hadoop cluster.
- HRegions: HRegions are the basic building elements of HBase cluster that consists of the distribution of tables and are comprised of Column families.
- Zookeeper: Zookeeper is a centralized monitoring server which maintains configuration information and provides synchronization in a distributed environment.

2.2. Elasticsearch

Elasticsearch is a highly scalable open-source engine used for full-text search and analytics. It enables users to store huge volumes of data quickly and fire real time queries on them with a very low response time. It is mostly used in the backend for applications having complex search features and requirements.

It is a search engine, similar to a database differing in the way the data is stored. It has a similar structure when compared to MYSQL. For example:

- Elasticsearch – Database
- Index – Database
- Type – Table
- Document – Row

- Field – Column

Elasticsearch works on the concept of an inverted index. In this data structure, everything is indexed. This enables one to search for a specific word in all documents in a much faster way. It is similar to an index at the end of a book. It is schema less. It uses mappings, which makes the working to be much easier. It can also predict data-types automatically. If it doesn't do so accurately, you can provide a mapping while creating an index.

Elasticsearch is built on top of Lucene. Lucene has proven to be the best of its kind in open source search software. Lucene contains the implementation of everything related to searching and indexing text. Elasticsearch builds an infrastructure around Lucene. Though Lucene is a great tool, it can be painful to use it directly as it does not provide any mechanism for working in a cluster. Elasticsearch provides an easier and more useful API (Application Program Interface – a set of protocol and tools for building software applications). It takes care of the infrastructure and operational tools required to scale across multiple nodes in a cluster.

The following diagram shows the architecture of an Elasticsearch cluster as seen in [10]:

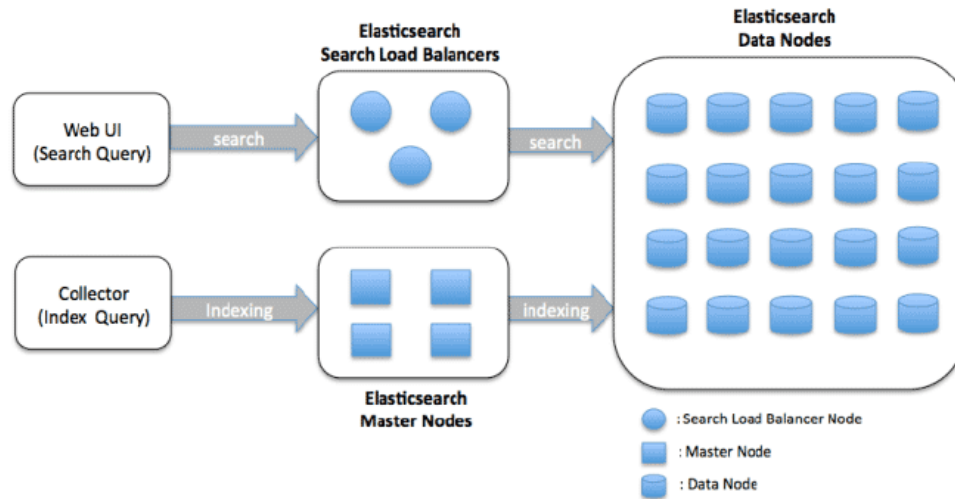


Fig. 2. Elasticsearch Architecture [10].

The functions of these three types of nodes are described below:

- Data node

This node serves the purpose of storing data. On receiving a request from a client, it creates an index or searches data from shards.

- Master node

It is responsible for maintaining a cluster, and requests indexing or search to data nodes.

- Search balancer node

On receiving a search request, it requests data, gathers it and delivers the result.

3. RELATED WORK

In this section, we will have a look at what are the existing approaches and what efforts have been taken by others to solve this problem.

First, let us see what provision does HBase provide for the kind of query we talked about in section 1.2 i.e. the use case where the query has a filter on non-rowkey columns. HBase has various kinds of filters and ValueFilter is the one suitable for this case. This is the syntax it has:

- `scan <table_name>, {COLUMNS => '<column_family:column_name',
FILTER => "ValueFilter (='binaryprefix:column_value')" }`

So, if we have a table named 'students' and we want to find all records having sex = Female, our query would look something like this:

- `scan 'students', {COLUMNS => 'details:sex',
FILTER => "ValueFilter (='binaryprefix:Female')" }`

But this would lead to a full table scan and that is what we are trying to avoid.

Apart from this, HBase also gives a provision of using a composite rowkey i.e. a combination of some columns as the rowkey. We can overcome this limitation by keeping the column (from which we intend to retrieve data frequently) as a part of the rowkey. But again there can be hundreds and thousands of columns in our data. We do not want to keep adding columns to the rowkey and make it too long. In HBase, values are always freighted with their rowkey, column details and timestamp. Due to this, the indices kept on HFiles to facilitate random access can eat up a lot of RAM. The patterns we select for our rowkey, column details can be repeated several billion times in our data based on the number of

records. Thus, we should be careful while choosing the rowkey. A very long rowkey can prove to be inefficient.

Let us now see what efforts others have taken to solve this problem:

- In [1], they propose using secondary indexing in HBase using coprocessors (similar to a trigger in a relational database). By creating secondary indexes in another table, they can change how data would be retrieved, thus avoiding a full table scan. Let us have a look at their approach: They plan to use one more table for indexing on the primary table. The coprocessor would bind to a family and will be used for defining a secondary index for that family (or any specific column in it). For each put operation, these are the steps they take:

1. Apply the put operation to the main table.
2. Put the jobs for secondary table edits in a shared job queue
3. The shared job queue would then pick up these jobs and execute them

But, there can be some problems with this approach. The secondary table could be offline because of another Region Server failure, so we may have long-waiting secondary updates. In such cases, we can not guarantee the secondary index updates, if an old HLog file was already deleted. We would have to keep track of the pending updates and prevent removal of logs to handle this. Other than that, this article just focuses on how to do the secondary indexing. It does not provide for a way to retrieve data after this has been done.

- In [2], they have a similar approach to what we saw in [1], but it addresses some of the problems and limitations we saw in approach [1]. They take care of Region

Collocation i.e. keeping the primary table and the index table on the same Region Server. This will not bring in the problem we had in the previous approach, i.e. if the index table was present on some other Region Server and that Region Server fails, there would be long-waiting secondary index updates. Region Collocation would also avoid RPC calls and thus it would not badly affect the write throughput. They have handled the put operation in actual table and the one in the index table in such a way which will help in maintaining the consistency between the actual table data and the index table data. WAL (Write Ahead Log – ensures durable writes for HBase) data corresponding to both actual put and index data put is synced together. In this approach, they have also talked about how they would handle the data retrieval part to leverage the secondary indexing. They plan to handle the index table lookup at the server side using co processors.

- In [3], the method they talk about the following steps: Establish a global index on a distributed memory and an Hbase table; identify a query for non-primary attributes, locate a corresponding index node corresponding to the global index, and send a request to the index node having a result. This guarantees that nodes not having the result set are not inquired. This solves the problem of performance waste. The query performance of the non-primary key attributes on HBase is improved.
 1. They have two ways of handling the non-primary key index: centralized index and distributed index. Centralized index is centrally managed. They expand the traditional single node in the index structure of the data management system without understanding the true distribution of data.

2. HBase-indexer is a centralized program. HBase sends updated data to the index server, analyses the data and generates the corresponding index data. The index server pushes data to the index on SolrCloud service periodically. Solr service can be accessed to locate content on HBase. This indexing mechanism periodically updates the index.
 3. Distributed program does not maintain the overall index. It is specific to each compute node. There is no dependence between computing nodes, which allows for concurrent execution retrieval requests. When a retrieval request is made, tasks will be distributed concurrently on all compute nodes. The final result would be returned on all nodes and data sets.
 4. Retrieval tasks assigned to each node are independently executed. Thus, parallel computing resources are utilized.
- In [4], the tools they are using are Apache Pheonix and Apache Calcite. Apache Phoenix is an open source, relational database engine supporting OLTP(On-line Transaction Processing) for Hadoop using Apache HBase as its backing store. Apache Calcite is an open source framework for building databases and data management systems. It includes a SQL parser, an API for building expressions in relational algebra, and a query planning engine. A secondary index in Phoenix is a projection of part or all of the columns of the original HBase table, and is usually indexed (and sorted) on a different key other than the primary key of the original table.
 1. Consider a table T1: {rowkey, b, c, d}.
 2. Suppose the user creates 2 indexes:

- Index 1 is created on 2 columns {b, c}. When Phoenix creates the corresponding table in HBase, it creates a table with 3-part primary key: {b, c, rowkey}. The reason for appending the rowkey to the list is that the combination of {b, c} may not be unique. Adding the rowkey makes the tuple a primary key. The table is sorted on this primary key.
 - Index 2 is created on 1 column: {d}. Corresponding HBase table has a 2-part primary key: {d, rowkey}.
3. These indices are registered as pre-populated materialized views in Calcite. The key column's collation trait is leveraged during logical planning.
 4. Consider the query: `SELECT * FROM T1 WHERE b = 10 AND c > 20 AND d < 15`
 5. This is converted by Phoenix to:
 - `SELECT * FROM T1 WHERE rowkey IN (SELECT rowkey FROM Index1 WHERE b = 10 AND c > 20) AND rowkey IN (SELECT rowkey FROM Index2 WHERE d < 15)`
 6. But there might be some disadvantages for this approach:
 - It will add a new semi-join for every index that is used. This adds to the search space during the logical planning phase. Suppose the outer query had joins with other tables: T1, T2, T3 and suppose each of them had 2 indices that were used by the query. Since all of these indices are available during the logical planning phase, Calcite will treat potentially 6 semi-joins together and try to find the optimal join ordering for these.

- A second consideration is that if the SQL query contains only the conditions 'c > 20 AND d < 15' then Phoenix would not be able to use Index 1 since the column 'c' is not a prefix of the sortkey: {b, c, rowkey}.

4. DATASET

In this section, we will have a look at the dataset we are going to use for this project. We will be using a dataset of City of Chicago employees, which is obtained in the form of a CSV (Comma Separated Values) file from [13]. Here are some details about the dataset:

No. of records = 2,027,276

It consists of the following columns:

Id	:	Employee ID
Lname	:	Employee's Last name
Fname	:	Employee's First name
Position Title	:	Employee's Job Position Title
Department	:	Employee's Job Department
Employee Annual Salary	:	Annual salary of the employee
Sex	:	Employee's Gender

The following image shows a snapshot of the data we have:

Id	Lname	Fname	Position Title	Department	Employee Annual Salary	Sex
32678	ZURAWSKI	MARY E	POLICE OFFICER	POLICE	\$95106.00	MALE
32679	ZUREK	FRANCIS	ELECTRICAL MECHANIC	OEMC	\$95888.00	MALE
32680	ZUREK	MARY H	SENIOR PUBLIC INFORMATION OFFICER	FINANCE	\$85008.00	FEMALE
32681	ZURITA	ADRIEL	POLICE OFFICER	POLICE	\$71790.00	MALE
32682	ZVANJA	TINA M	LEGAL SECRETARY	LAW	\$75420.00	FEMALE
32683	ZWARYCZ MANN	IRENE A	CROSSING GUARD	OEMC	\$18023.20	MALE
32684	ZWARYCZ	THOMAS J	POOL MOTOR TRUCK DRIVER	WATER MGMNT	\$74048.00	MALE

Fig. 3. Snapshot of Data

5. APPROACH

In this section, we will have a look at how we plan to implement our solution and the technical details behind it.

5.1. Overview

The solution we propose consists of the following steps:

1. Data Ingestion:
 - a. Importing data from a CSV file into a HBase table
 - b. Secondary Indexing of HBase table in ElasticSearch
2. Data Retrieval:

It consists of the following steps:

- a. User queries the Java API
- b. The Java API searches for the required value in ElasticSearch documents and retrieves relevant rowkeys
- c. Now, the Java API gets all those records for returned rowkeys from HBase table
- d. Finally, the Java API returns the results to the user

Let us see every step in detail:

1. Data Ingestion:

The following figure shows the steps involved in Data Ingestion:

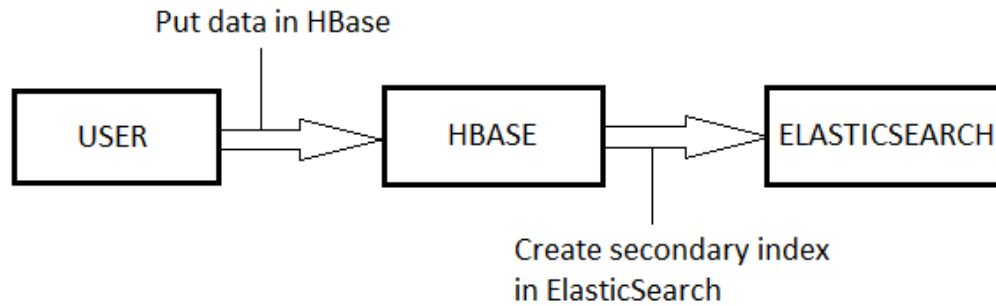


Fig. 4. Data Ingestion.

a. Importing data from a CSV file into a HBase table

Before indexing data into HBase, first we need to import data from the CSV file into HBase.

b. Secondary Indexing of HBase table in ElasticSearch

We now have our data in HBase. The second step in the data ingestion process is indexing data from HBase to Elasticsearch. This is a one-time additional step which the user must take before he can query the data through our approach in contrast to the traditional HBase approach, where in the user can directly query HBase once he has the data stored in it. In this step, we need to create a secondary index on the data we have in HBase. But first, let us see what a secondary index is:

In simple words, a secondary index provides users with an efficient way for accessing data in a database by using some information other than the regular (primary) key. Based on what data we have in our main table, we can make corresponding entries to the index table. What gets added to index table is the indexed column's value and that respective record's rowkey.

Secondary indexes can be of two types: covering and non-covering indexes. A covering index contains all columns that are referenced in the query. A non-covering index contains only a subset of the columns. In our case, we will be dealing with a non-covering index. We will be focusing on just one column for this project.

For example, consider the employee dataset which is described in section 4. If we do secondary indexing based on one column, say the column “sex” and we have a record with ID = 32680 (Rowkey is the ID), where the sex = Female. An entry in the index table corresponding to this record will look like this:

Column Value	Rowkey
Female	32680

TABLE II
SAMPLE ENTRY IN INDEX TABLE

As shown above, entries will be made in the index table for all the records which are present in the main table in a similar fashion. The above representation just gives us an idea of how secondary indexing works. Our index table will be stored in Elasticsearch and the actual format of each entry in Elasticsearch will be discussed in the following section.

2. Data Retrieval:

Once the data ingestion is done, now the important part of data retrieval comes into picture. This is the part we are trying to optimize which would finally result in a low response latency. Before we have a look at what approach we are taking, let us see how the flow would look like for the traditional HBase approach:

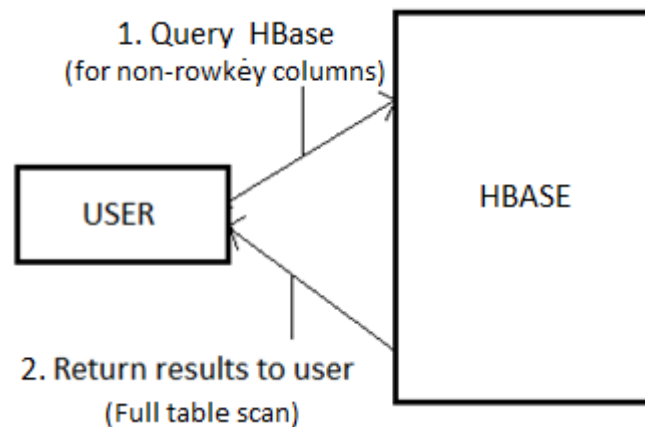


Fig. 5. Data Retrieval (Traditional HBase approach)

As seen in the above figure, a user who is not using our approach queries HBase directly for retrieval based on some non-rowkey column. HBase then performs a full table scan in order to retrieve the relevant rows for the user's query.

Now, let us have a look at how we plan to modify the data retrieval process:

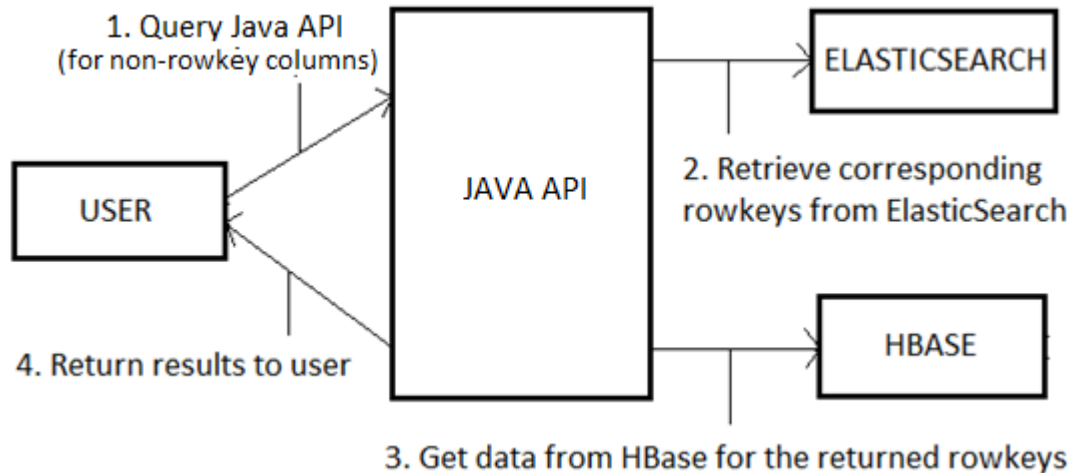


Fig. 6. Data Retrieval (Our approach)

Let us see every step shown in the above diagram in further detail:

a) Query Java API

Our approach would be bundled in a Java program and this is the API we provide to the user for his queries. In this first step, the user submits his query to our Java API. Say he submits a query of retrieving all employee records where sex = 'Female'.

b) Retrieve corresponding rowkeys from Elasticsearch

Now we have the data stored in HBase and indexed in ElasticSearch. This is the first step in data retrieval, wherein we query ElasticSearch first and get a list of all rowkeys pertaining to the indexed value "Female". In the example we are discussing, this step would give us a list of rowkeys: [32680,32682] i.e. the IDs of the records where sex = Female. This step would be fast as ElasticSearch is a powerful and fast search engine, as we have already discussed in section 2.3.

c) Get data from HBase for the returned rowkeys

Now that we have obtained all the rowkeys corresponding to the query, we will query HBase and retrieve only those records which have the rowkey present in our list of rowkeys. As in the above example, we would then query HBase and get records for the corresponding rowkeys: [32680,32682]. This would result in scanning the HBase table only for two keys, instead of a full table scan. In this example, we avoided scanning all 7 records in the HBase table. Here, it does not make much of a difference as the total number of records is very small. Think of a table having rows in the order of millions/billions. That is where, this approach would prove to be effective. By avoiding a full table scan, the latency of the response would be reduced and the user would be able to view the results in a shorter time and that is what we exactly aim to accomplish by this project.

d) Return results to the user

Once we have retrieved the results from HBase, we will print it out for the user and this is the final step in our solution.

5.2. Technical Details

In the previous section, we saw a logical plan, which demonstrates the strategy for our solution. In this section, we will describe all the steps that need to be taken for realizing our solution in practice:

1. Data Ingestion:

a. Importing data from a CSV file into a HBase table:

Before indexing data in Elasticsearch, we need to put data into the HBase table. We have our data file in a CSV format. HBase provides us with a tool called ImportTSV (Import Tab Separated Values), which can be used for importing data from a CSV file into a HBase table. The following screenshot shows how to use this tool:

```
[swapnilk@enode common_jars]$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv \  
> -Dimporttsv.columns=HBASE_ROW_KEY,det:fname,det:lname,det:pos,det:dept,det:sal,det:sex \  
> '-Dimporttsv.separator=,' \  
> empdata \  
> hdfs://nnode.kwartile.com:8020/user/swapnilk/mproject/data/empdata.csv
```

Fig. 7. Using ImportTSV tool

Let us understand the above command:

- The first line “hbase org.apache.hadoop.hbase.mapreduce.ImportTsv” specifies the tool we are using i.e. ImportTsv
- The second line:
“Dimporttsv.columns=HBASE_ROW_KEY,det:fname,det:lname,det:pos,det:dept,det:sal,det:sex” indicates the rowkey, column families and their column names of the HBase tables. Our first column “ID” is the rowkey and hence “HBASE_ROW_KEY” comes first, which tells the tool to use this column as a rowkey. Rest of the arguments specify “column family:column name” pairs corresponding to each of the columns.
- The third line “-Dimporttsv.separator=,” explicitly specifies the separator to be ‘,’ as the default separator for ImportTSV is a tab character ‘\t’.
- The fourth line “empdata” indicates that the name of our HBase table is empdata.

- The last line:

“hdfs://nnode.kwartile.com:8020/user/swapnilk/mproject/data/empdata.csv” specifies the path of our input file, which is supposed to be present in HDFS (Hadoop Distributed File System).

- Note: For the above import statement to work correctly, we need to have the HBase table created beforehand.

b. Secondary Indexing of HBase table in ElasticSearch

Now that we have our data in the HBase table, we can now proceed towards making a secondary index entry in Elasticsearch for each of our records in HBase. We have accomplished this part through a Java program. Both HBase and Elasticsearch provide a Java API, through which we can communicate with them and perform the required operations. These are the steps which are required to be performed for completing this part of the data ingestion process:

- We will be passing all the required arguments to our java program through the command line as follows:

```
[swapnilk@enode secIndexES]$ mvn exec:java \
> -Dexec.args = "<table_name>,<column_family>,<column_name>," \
> <cluster_name>,<host_name>,<index_name>,<index_type>"
```

Fig. 8. Command for secondary indexing of Data

In the above screenshot, the first line indicates the command for executing our java program.

The second line consists of command line arguments pertaining to the HBase table we want to use for performing secondary indexing. It consists of the following details:

- <table_name> : HBase table name
- <column_family> : The column family for index column
- <column_name> : The index column name

The third line consists of command line arguments pertaining to Elasticsearch, where we wish to index our column. It consists of the following details:

- <cluster_name> : Elasticsearch cluster name
 - <host_name> : Elasticsearch host name
 - <index_name> : Elasticsearch index
 - <index_type> : Index type
- Once our program gets all the required arguments from the user, it needs to make a connection to HBase now. The following screenshot shows how this connection will be made:

```
// Configuring HBase
Configuration config = SecIndexingHbaseToES.getHHConfig();

// Connecting to HBase
Connection conn = ConnectionFactory.createConnection(config);
Table table = conn.getTable(TableName.valueOf(params[0]));
```

Fig. 9. Connecting to HBase

As seen in the above screenshot, we create a configuration object for HBase and then use that for making a connection to HBase. In the last

line, we instantiate our table object, where `params[0]` corresponds to the table name, which we get as input from the user.

- After we are done connecting to HBase, we now need to retrieve all records from the table so that we can create a secondary index for each record in Elasticsearch. The following screenshot shows how this will be done:

```
// Instantiating the Scan class
Scan scan = new Scan();

// Scanning the required columns
scan.addColumn(Bytes.toBytes(params[1]), Bytes.toBytes(params[2]));

// Getting the scan result
ResultScanner scanner = table.getScanner(scan);
```

Fig. 10. Getting data from HBase

As seen in the above screenshot, we create a scan object in the first line. As we just want values from the columns we wish to index, we add those column details to the object, so that it scans just for that column. The column details are comprised of `params[1]` (column family) and `params[2]` (column name). These values are also obtained from the user. After adding the column details, we perform the table scan operation and retrieve all our column values in the scanner object.

- We have now retrieved all column values from the HBase table. Before starting the secondary indexing process, we need to make a connection to Elasticsearch. The following screenshot shows how we can connect to Elasticsearch using the Java API:

```
//Configuring settings for Elasticsearch
Settings settings = Settings.builder()
    .put("cluster.name", params[3]).build();

//Connecting to Elasticsearch
TransportClient client = new PreBuiltTransportClient(settings)
    .addTransportAddress
    (new InetSocketTransportAddress(InetAddress.getByName(params[4]), 9300));
```

Fig. 11. Connecting to Elasticsearch

As seen in the above screenshot, we first create a settings object where we supply the cluster name `params[3]`, the value of which we have obtained from the user. Using these settings, we create a `TransportClient` object for communicating with Elasticsearch. During this, we also specify the hostname `params[4]` (obtained from user) and the port no. 9300.

- Now, we are connected to Elasticsearch. At this point in time, we can go ahead with the secondary indexing process. First, let us have a look at how an actual secondary index entry would look like in Elasticsearch. The following image gives us the structure of a secondary index entry:

```
[swapnilk@enode ~]$ curl -XGET 64.71.190.35:9200/newempdata/sex/0?pretty
{
  "_index" : "newempdata",
  "_type" : "sex",
  "_id" : "0",
  "_version" : 1,
  "found" : true,
  "_source" : {
    "colvalue" : "MALE",
    "rowkey" : "0"
  }
}
```

Fig. 12. A secondary index entry in Elasticsearch

In the above screenshot, we see how a secondary index entry would look like in Elasticsearch. It stores files in a json format in the form of key:value pairs. As discussed in Section 2.2, we can make sense of the above screenshot in a SQL analogy:

- Index (Database) : newempdata
- Type (Table) : sex (corresponds to index column)
- Document (Row) : All values for the `_source` key
- Field (Column) : A single key:value pair in `_source`

"colvalue" : "MALE" is the key : value pair where we'll be storing the column value from the table. "rowkey" : "0" is the key : value pair where we'll be storing its corresponding rowkey. Entries of such format will be made in Elasticsearch for all the records in the HBase.

- Now that we have understood the structure of each secondary index entry in Elasticsearch, let us see how do we do this for every record in HBase. We have already retrieved all records from HBase, as shown in the previous steps. The following screenshot shows how to index each record in Elasticsearch:

```
// Reading values from scan result and indexing it into Elasticsearch
for (Result result = scanner.next(); result != null; result = scanner.next())
{
    XContentBuilder EsEntry = XContentFactory.jsonBuilder()
        .startObject()
        .field("colvalue",
            new String
                (result.getValue(params[1].getBytes(),params[2].getBytes())))
        .field("rowkey", new String(result.getRow()))
        .endObject();
    bulkProcessor.add(new IndexRequest(params[5],params[6],Integer.toString(counter))
        .source(EsEntry));
    //client.prepareIndex(params[5], params[6],Integer.toString(counter))
    //    .setSource(EsEntry).execute().actionGet();

    counter++;
}
```

Fig. 13. Index HBase data to Elasticsearch

Let us understand what is going on in the above screenshot. In the for loop, we are scanning through each record retrieved from HBase. We make a JSON (JavaScript Object Notation) object `EsEntry` using `XContentBuilder` library provided by Elasticsearch. Here we are building the key:value pairs for “colvalue” and “rowkey”. We are getting these values from the HBase table. `params[1]` and `params[2]` correspond to column family and column name respectively. The `getRow()` function gets the rowkey from HBase for the corresponding record.

Once this JSON object has been created we are adding an `IndexRequest` to the `BulkProcessor`. `params[5]` and `params[6]` correspond to index name and index type respectively. We set the source for the `IndexRequest` as `EsEntry`, which is the JSON object we just created.

Notice the commented part in the image. Even this statement can insert an entry into Elasticsearch for our record. But if we use this for indexing millions of documents, the program would run forever. The first time we were doing secondary indexing, we were using this method and it took around 6 hours for indexing around 500,000 records. Then, later we found the BulkProcessor API, which we are currently using.

Using the BulkProcessor API, we can keep adding IndexRequests to it as we process each record. Depending on the settings we provide to it, it will automatically send a bulk indexing request to Elasticsearch after a certain number of IndexRequests are added to it. In our case, we have set this value to 50,000. Due to the BulkProcessor API, we can index around 2,000,000 records in just 67 seconds.

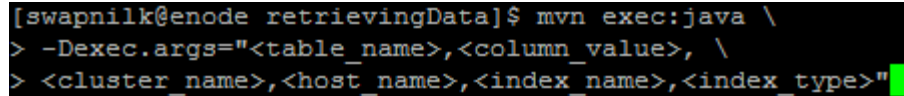
- We have now completed the Data Ingestion step and a secondary index has been created in Elasticsearch for all our records in HBase.

2. Data Retrieval:

Now that we are done with the Data Ingestion part and we have our secondary indexes ready, we can go ahead with the data retrieval. These are the technical details for each step in Data Retrieval:

a. Query Java API

The user needs to submit his query to our program, which is again bundled in the form of a Java program. The following screenshot shows how the user would submit his request to our Java program:



```
[swapnilk@enode retrievingData]$ mvn exec:java \
> -Dexec.args="<table_name>,<column_value>, \
> <cluster_name>,<host_name>,<index_name>,<index_type>"
```

Fig. 14. Command for retrieval of data

In the above screenshot, the first line indicates the command for executing our Java program. The second line takes arguments such as the HBase table name (<table_name>) and the column value (<column_value>), which is the column value we want to search for. According to the example we have been discussing, it can be for eg. "Female". The last line corresponds to all the arguments for Elasticsearch i.e. cluster name, host name, index name and index type.

b. Search in ElasticSearch documents and retrieve relevant rowkeys

Now that the user has submitted his query, the Java program will go ahead to fetch the relevant rowkeys from Elasticsearch, which correspond to the column value entered by the user. The connection to Elasticsearch will be made in the same way as it was shown during the Data Ingestion part. The following screenshot shows how we search for the rowkeys in Elasticsearch:

```
//Regular search
SearchResponse response = client.prepareSearch(params[4])
    .setTypes(params[5])
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setFetchSource(new String[]{"rowkey"}, null)
    .setQuery(QueryBuilders.queryStringQuery(params[1]).defaultField("colvalue"))
    .setSize(100)
    .execute()
    .actionGet();
SearchHit[] results = response.getHits().getHits();
```

Fig. 15. Regular search in Elasticsearch

As seen in the above screenshot, we can get a response from Elasticsearch by passing in all the required parameters. The different parameters are described below:

- `params[4]` indicates the index which needs to be searched.
- `params[5]` indicates the index type which has to be searched.
- The `setFetchSource` method specifies the field in Elasticsearch which we want to retrieve. This is field “rowkey in our case”.
- The `setQuery` method indicates the query, which has our parameter `params[1]`, which corresponds to the column value.
- The `setSize` method sets the maximum no. of returned results limit to 100.
- The `execute` and `actionGet` method execute our search and get all the results.
- We then store our results in the results variable.

We have now retrieved the rowkeys for our column value. But there is one problem in the method mentioned above. The maximum value our `setSize` method takes is 10000. This means that if there are more than 10000 records which we seek to retrieve from Elasticsearch, we can't do it using the method mentioned above.

To overcome this problem, we can use the Scroll API provided by Elasticsearch. The following screenshot shows how we can accomplish this:

```

//Using the scroll API for searching Elasticsearch
SearchResponse scrollResp = client.prepareSearch(params[4])
    .addSort(FieldSortBuilder.DOC_FIELD_NAME, SortOrder.ASC)
    .setScroll(new TimeValue(60000))
    .setTypes(params[5])
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setFetchSource(new String[]{"rowkey"}, null)
    .setQuery(QueryBuilders.queryStringQuery(params[1])
        .defaultField("colvalue"))
    .setSize(10000)
    .get();
int counter = 0;
do{
    for (SearchHit hit : scrollResp.getHits().getHits()) {
        // For each rowkey obtained from Elasticsearch,
        // get the corresponding record from HBase
    }
    scrollResp = client.prepareSearchScroll(scrollResp.getScrollId())
        .setScroll(new TimeValue(60000)).execute().actionGet();
}while(scrollResp.getHits().getHits().length != 0);

```

Fig. 16. Searching using Scroll API in Elasticsearch

In the above image, we have used the Scroll API for retrieving results from Elasticsearch. The scroll API helps us retrieve large numbers of results (or even all results) from Elasticsearch index, similar to the way you would use a cursor on a traditional database. As visible in the above image, even though we have given 10000 as the argument for the setSize method, we can still retrieve all remaining records by scrolling through it. By scrolling we mean that getting each batch of 10000 records in the do while loop until we get no more records from Elasticsearch. Thus, the Scroll API helps us retrieve each rowkey corresponding to our column value.

- c. Get data from HBase for the returned rowkeys

We have now retrieved all rowkeys corresponding to our desired results.

Now we can get our data from HBase by doing individual gets for our rowkeys.

The following screenshot shows how this can be done:

```
for (SearchHit hit : scrollResp.getHits().getHits()) {
    // For each rowkey obtained from Elasticsearch,
    // get the corresponding record from HBase
    Map<String, Object> result = hit.getSource();

    // Instantiating Get class
    Get g = new Get(Bytes.toBytes((String) result.get("rowkey")));

    // Reading the data
    Result res = table.get(g);
    // Reading values from Result class object
    byte [] value = res.getValue(Bytes.toBytes("det"), Bytes.toBytes("lname"));

    counter++;
}
```

Fig. 17. Individual Gets from HBase for rowkeys retrieved from Elasticsearch

In the above screenshot, we send a get request to HBase for each rowkey that we obtained from Elasticsearch. By using the rowkey, we are getting the corresponding last name of the employee from the HBase table. This get request works pretty fast as it directly gets a record from HBase based on the rowkey. And HBase is designed in a way that it works quite efficiently for retrieval with rowkeys. Through these individual gets, we retrieve all the records for the corresponding rowkeys.

d. Return results to the user

Now that we have all our results, we can print it out for the user.

6. EXPERIMENTS

6.1. Hardware Configuration

Before we have a look at what experiments we are going to perform, it is important to know what hardware configuration are we doing the experiments on. We are using a cluster of 4 machines connected in a network. Each machine has a configuration as shown below:

- Processor : Intel(R) Xeon(R)
- CPU Cores : 4
- Operating System : CentOS Linux 7
- Memory : 94 GB

We are running both HBase and Elasticsearch on this cluster of 4 machines.

6.2. Experiment 1

In the first experiment, we are indexing the “sex” column in our employee dataset.

Indexing will be done our Java program by issuing the following command.

```
mvn exec:java -Dexec.args="empdata2m, det, sex, myapplication, 64.71.190.34, empdata2m, sex"
```

After indexing is done, we will be retrieving all employee records where sex = “Female”. This retrieval of data would be done in two ways: HBase’s traditional approach and our approach.

We will repeat this experiment for different number of female records each time and compare the retrieval time for these two approaches in every case. The following table shows the distribution of data that we will be using for this experiment:

Total No. of Records = 2,027,276
No. of Female Records to be retrieved
186
349
592
820
1014
1216

TABLE III
DATA DISTRIBUTION FOR EXPERIMENT 1

For each of these distributions, we will be retrieving data in the following two ways:

- HBase's traditional way:

```
scan 'empdata2m',{ COLUMNS => 'det:sex', FILTER => "ValueFilter(=, 'binaryprefix:FEMALE' )" }
```

By the upper query, we intend to perform a scan on the table named 'empdata2m' by applying a filter on the column 'sex' belonging to 'det' column family for retrieving only those records where sex = 'FEMALE'.

- Our approach:

```
mvn exec:java -Dexec.args="empdata2m,FEMALE,myapplication,64.71.190.34,empdata2m,sex"
```

Through the upper query, we intend to do the same thing by running our Java program, which implements our approach.

6.3. Experiment 2

In the previous experiment, we are mainly focusing on retrieving data for a category which has very less number of records as compared to the total number of records. We should also try and compare results for other distributions of data, in which the category we want to retrieve results for, has records in a higher proportion as compared to the total number of records.

To try these different distributions, we chose to index the data by the 'Department' column for this experiment, as this column has a varied distribution of data. We will try to retrieve data through HBase's traditional approach and our approach, as done in section 6.1. We will then compare these results.

The following command is used for indexing data from the 'Department' column:

```
mvn exec:java -Dexec.args="empdata2m,dept,myapplication,64.71.190.34,empdata2m,dept"
```

Now that indexing is done, retrieval experiments will be performed. The following table explains the distribution of data that we will be using for this experiment:

Total No. of Records = 2,027,276	
DEPARTMENT	No. of Records to be retrieved
POLICE	810898
FIRE	298592
STREETS & SAN	136152
OEMC	120900
WATER MGMNT	116498
AVIATION	100006
TRANSPORTN	68448
GENERAL SERVICES	60264
PUBLIC LIBRARY	57846
FAMILY & SUPPORT	38564
HEALTH	32116
LAW	25110
BUILDINGS	16430
PROCUREMENT	5332
IPRA	4526
INSPECTOR GEN	3906
ADMIN HEARNG	2356
TREASURER	1488
LICENSE APPL COMM	62

TABLE IV
DATA DISTRIBUTION FOR EXPERIMENT 2

These two commands are used for the two approaches:

- Hbase's traditional way:

```
scan 'empdata2m',{ COLUMNS => 'det:dept', FILTER => "ValueFilter(=, 'binaryprefix:POLICE' )" }
```

- Our approach:

```
mvn exec:java -Dexec.args="empdata2m,POLICE,myapplication,64.71.190.34,empdata2mdept,dept"
```

7. RESULTS

We performed both the experiments mentioned in the previous section and got interesting results for the time taken for retrieving different number of records from the HBase table having a total of approximately 2,000,000 records. Let us have a consolidated view at our results from both these experiments. This will help us understand how our approach performs as compared to the traditional approach when handling different distributions of data. The following table shows the overall results for both the experiments:

No. of Records being retrieved	Retrieval Time (s)	
	Our approach	Hbase
62	1.85	5.27
186	1.94	5.83
349	2.16	5.78
592	2.48	6.09
820	2.57	4.97
1014	2.75	6.06
1216	2.88	5.9
1488	3.11	5.48
2356	3.69	5.63
3906	5.05	5.64
4526	5.29	5.83
5332	5.65	5.93
16430	13.28	6.81
25110	19.26	8.1
32116	23.71	8.66
38564	27.74	8.75
57846	40.1	10.95
60264	42.78	11.31
68448	46.99	11.65
100006	64.63	19.89
116498	77.09	23.45
120900	82.82	24.5
136152	90.79	24.95
298592	196.18	61.24
810898	512.13	158.51

TABLE V
CONSOLIDATED RESULTS

As we can see in the above tables, we get interesting results depending upon the different distributions of data.

Let us have a visual view of the results to better understand them. The following graph has 'No. of records being retrieved' on the X axis and 'Time (in seconds)' on the Y axis for the range of records to be retrieved from 0-16,430:

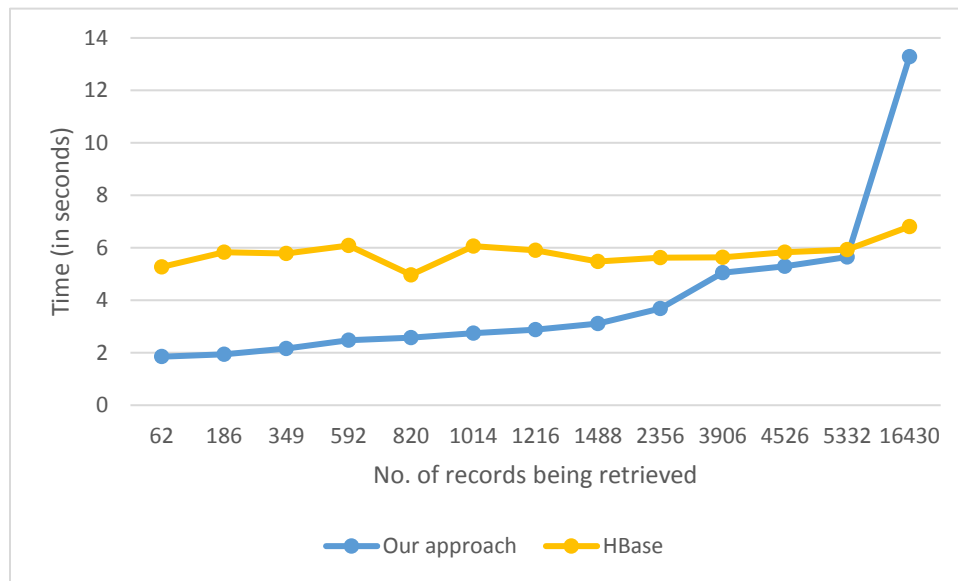


Fig. 18. Comparison of results (First few distributions)

As we can see in the above graph, our approach performs better than the traditional approach of HBase when the no. of records to be retrieved is around 2000. For the range of $2000 < \text{no. of records} < 5000$, the performance of our approach is almost the same as that of HBase. But after that we see that our approach takes almost 14 seconds for retrieving around 16,000 records as compared to HBase which takes only 4 seconds.

The following graph has 'No. of records being retrieved' on the X axis and 'Time (in seconds)' on the Y axis for the remaining range of records:

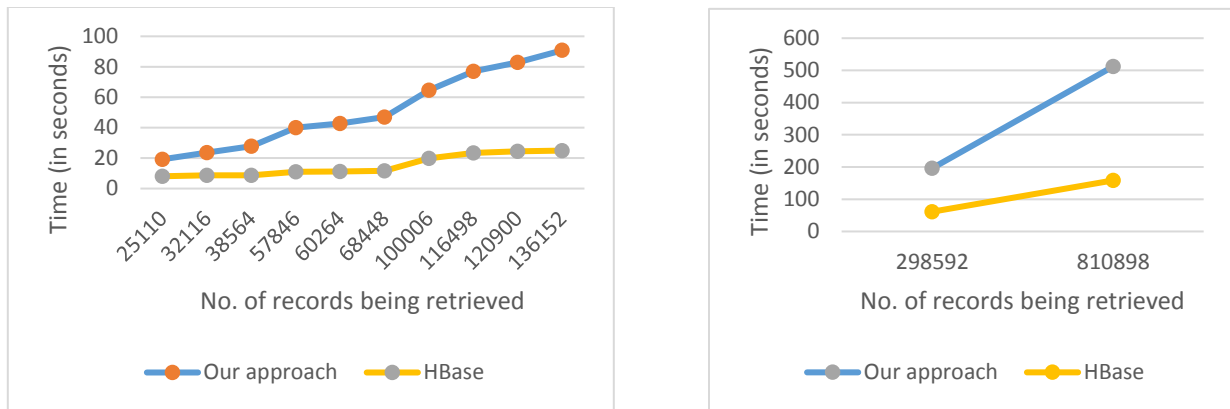


Fig. 19. Comparison of results (Remaining distributions)

We plotted the above two graphs separately in order to get a better picture in terms of the time taken for retrieving different no. of records. We see that in both of the graphs, the performance of our approach worsens as the no of records to be retrieved goes on increasing.

From our results, we see that our approach is only appropriate and useful if the retrieval is based on a column which has a large number of categories, which means that there would be many categories having a less number of records per category as compared to the total number of records in the table. If the number of records to be retrieved keep on increasing, the traditional approach itself would give a considerably better performance than our approach, as seen in the results. Our approach takes even more time as compared to the traditional approach when the number of records to be retrieved goes on increasing.

Such kind of a behavior could be possible because of the following reason. In our approach for data retrieval, we must query Elasticsearch first to get each rowkey corresponding to that value and then we must fire multiple individual get queries to

HBase to get the results. This extra work is fine in cases where the records to be retrieved is less. But when the number of records to be retrieved is high, it seems that this becomes an overhead and thus our approach's performance keeps on worsening as the number of records to be retrieved increase. In those cases, a direct scan performs much better.

8. CONCLUSION

We got interesting results for the experiments we carried out. If the number of records to be retrieved is less as compared to the total no. of records (for our case approximately 2000 / 2,000,000 meaning 1 in every 2000 records), then our approach is suitable. Our approach would be more effective when the total number of records is relatively high and the number of records to be retrieved is pretty less i.e. in cases where the column has a large number of different categories. But as the number of records to be retrieved approaches the total number of records, we see that the retrieval time for our approach worsens. Thus, before using this as a solution in any product or for any use case, the distribution of the data to be retrieved as compared to the total number of records should be considered before the approach can be decided. Apart from this, if a user wishes to use our approach, he should have Elasticsearch installed on his system and he would have to carry out the one-time additional step of secondary indexing data present in HBase to Elasticsearch.

9. FUTURE WORK

Firstly, the approach we have implemented requires secondary indexing of the data in a HBase table. We do that for an existing HBase table. What if the table keeps growing incrementally? To handle this, we would have to implement a mechanism such that in cases like these, whenever an update is made to the HBase table, it automatically creates a secondary index entry in Elasticsearch. This can be achieved through the use of a coprocessor, which acts like a trigger in a database. A postPut coprocessor could be implemented for this purpose. In simple words, it would make an entry in ElasticSearch after each put operation in HBase. Through this feature, the user can keep using this functionality even if the HBase table is being updated periodically.

Secondly, our approach only works for a single column. It creates a non-covering index which basically means that it does not cover all the columns we have. In the real world scenario, we would be having queries which might be involving data retrieval based on multiple non-rowkey columns. We can extend this approach in the future to handle such cases.

Thirdly, our approach is a standalone solution. We can integrate this solution as a black box to the user in some product which works on top of HBase. Apache Drill is one such example. It is a schema-free SQL engine having the capability of working on top of a NoSQL database (HBase in our case).

Lastly, if we were to integrate this into some product, we would have to make some additional considerations. Consider Apache Drill, where we use HBase as a storage plugin and fire this kind of a SQL query for retrieving data based on a non-rowkey column. We saw that

our approach is sensitive to the distribution of data. To handle this, Drill should in the backend, have some metadata about how the data is distributed. Based on that, it should decide whether to take this path of execution or go with the traditional approach instead.

10. REFERENCES

- [1] "HBase Secondary Indexing,2011. [Online]. Available: <https://wiki.apache.org/hadoop/Hbase/SecondaryIndexing>
- [2] "Secondary Indexing Design Updated",JIRA Issues. [Online]. Available: [https://issues.apache.org/jira/secure/attachment/12621909/SecondaryIndex%20Design Updated 2.pdf](https://issues.apache.org/jira/secure/attachment/12621909/SecondaryIndex%20Design%20Updated%202.pdf)
- [3] "HBase non-primary key index building and inquiring method and system", 2015. [Online]. Available: <https://www.google.com/patents/CN104850572A?cl=en>
- [4] "Pheonix Secondary Indexing". [Online]. Available: https://phoenix.apache.org/secondary_indexing.html
- [5] B. R. Chang, H. F. Tsai, Y. A. Wang and C. Y. Chen, "Realization of secondary indexing to NoSQL database with intelligent adaptation," *2015 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, Tainan, 2015, pp. 449-452.
- [6] H. Chen, K. Lu, M. Sun, C. Li, H. Zhuang and X. Zhou, "Enumeration System on HBase for Low-Latency," *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Shenzhen, 2015, pp. 1185-1188.
- [7] DRILL-3929: Support the ability to query database tables using external indices. [Online]. Available: <https://issues.apache.org/jira/browse/DRILL-3929>
- [8] A. Jacobs, "The pathologies of big data," *Communications of the ACM—A Blind Person's Interaction with Technology*, vol. 52, no. 8, pp. 36–44, 2009.

- [9] Scott Leberknight 2013, Handling Big Data with HBase Part 3: Architecture Overview, DZone, accessed 24 April, 2017. <<https://dzone.com/articles/handling-big-data-hbase-part-3>>
- [10] Rama Krishna Panguluri 2015, Deep Dive on Elastic Search, SlideShare, accessed 26 April, 2017. <<https://www.slideshare.net/ABCTalks/elastic-search-overview>>
- [11] J. Pokorny, "NoSQL databases: a step to database scalability in web environment," International Journal of Web Information Systems, vol. 9, no. 1, pp. 69–82, 2013.
- [12] P. Zhou, J. Lei, and W. Ye, "Large-scale data sets clustering based on MapReduce and Hadoop," Journal of Computational Information Systems, vol. 7, no. 16, pp. 5956–5963, 2011.
- [13] Dataset. [Online]. Available: <https://catalog.data.gov/dataset/current-employee-names-salaries-and-position-titles-840f7>
- [14] C. Boja, A. Pocovnicu, and L. Batagan, "Distributed parallel architecture for big data," Informatica Economica, vol. 16, no. 2, pp. 116–127, 2012.
- [15] "HBase Filtering". [Online]. Available: http://www.cloudera.com/documentation/enterprise/5-7-x/topics/admin_hbase_filtering.html#xd_583c10bfdbd326ba-7dae4aa6-147c30d0933--7c9e
- [16] M. Hausenblas and J. Nadeau, "Apache drill: interactive ad-hoc analysis at scale," Big Data, vol. 1, no. 2, pp. 100–104, 2013.
- [17] Bao Rong Chang, Hsiu-Fen Tsai, Chia-Yen Chen, Chien-Feng Huang, and Hung-Ta Hsu, "Implementation of Secondary Index on Cloud Computing NoSQL Database in Big Data Environment," *Scientific Programming*, vol. 2015, Article ID 560714, 10 pages, 2015.

[18] “How to Index MapR-DB Data into Elasticsearch”. [Online]. Available:

<https://www.mapr.com/blog/how-index-mapr-db-data-elasticsearch>

[19] “Secondary Indexing for MapR-DB using Elasticsearch”,2016. [Online]. Available:

<https://www.mapr.com/blog/secondary-indexing-mapr-db-using-elasticsearch>

[20] “The how to of HBase CoProcessors”. [Online]. Available:

<http://www.3pillarglobal.com/insights/hbase-coprocessors>

[21] DRILL-3637: ElasticSearch storage plugin. [Online]. Available:

<https://issues.apache.org/jira/browse/DRILL-3637>

11. APPENDIX

11.1. Code for Secondary Indexing from HBase to ElasticSearch

```
package secIndexES;

import java.io.IOException;

public class SecIndexingHbaseToES {

    public static org.apache.hadoop.conf.Configuration getHHConfig() {
        Configuration conf = HBaseConfiguration.create();
        InputStream confResourceAsStream = conf.getConfResourceAsStream("hbase-site.xml");
        int available = 0;
        try {
            available = confResourceAsStream.available();
        } catch (Exception e) {

            System.out.println("Didn't reach configuration");
        } finally {
            IOUtils.closeQuietly(confResourceAsStream);
        }
        if (available == 0 ) {
            System.out.println("Reaching configuration");
            conf = new Configuration();
            conf.addResource("core-site.xml");
            conf.addResource("hbase-site.xml");
            conf.addResource("hdfs-site.xml");
        }
        return conf;
    }

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

        String[] params = new String[7];
        //Checking no of arguments
        if(args.length < 1){
            System.out.println("Usage:\n");
            System.out.println("Parameters 1-3 correspond to HBase");
            System.out.println("Parameters 4-7 correspond to Elasticsearch");
            System.out.println("mvn exec:java -Dexec.args=\"<table_name>,<column_family>,<column_name>,\"
                + "<cluster_name>,<host_name>,<index_name>,<index_type>\"");
            System.exit(0);
        }
        else{
            params = args[0].split(",");
            // Instantiating Configuration class
            System.out.println("Reached before config");

            // Configuring HBase
            Configuration config = SecIndexingHbaseToES.getHHConfig();

            // Connecting to HBase
            Connection conn = ConnectionFactory.createConnection(config);
            Table table = conn.getTable(TableName.valueOf(params[0]));

            //Configuring settings for ElasticSearch
            Settings settings = Settings.builder()
                .put("cluster.name", params[3]).build();
```



```

//Connecting to Elasticsearch
TransportClient client = new PreBuiltTransportClient(settings)
    .addTransportAddress
        (new InetSocketTransportAddress(InetAddress.getByName(params[4]), 9300));

System.out.println("Reached before connection");

//Create BulkProcessor

BulkProcessor bulkProcessor = BulkProcessor.builder(
    client,
    new BulkProcessor.Listener() {
        @Override
        public void beforeBulk(long executionId, BulkRequest request) {
            System.out.println("Going to execute new bulk composed of "
                + request.numberOfActions() + " actions");
        }
        @Override
        public void afterBulk(long executionId,
            BulkRequest request,
            BulkResponse response) {
            System.out.println("Executed bulk composed of "
                + request.numberOfActions() + " actions");
        }
        @Override
        public void afterBulk(long executionId,
            BulkRequest request,
            Throwable failure) {
            System.out.println("Error executing bulk" + failure);
        }
    })
    .setBulkActions(50000)
    .setConcurrentRequests(1)
    .build();

// Instantiating the Scan class
Scan scan = new Scan();

// Scanning the required columns
scan.addColumn(Bytes.toBytes(params[1]), Bytes.toBytes(params[2]));

// Getting the scan result
ResultScanner scanner = table.getScanner(scan);

int counter = 0;
// Reading values from scan result and indexing it into Elasticsearch
for (Result result = scanner.next(); result != null; result = scanner.next())
{
    XContentBuilder EsEntry = XContentFactory.jsonBuilder()
        .startObject()
        .field("colvalue",
            new String
                (result.getValue(params[1].getBytes(), params[2].getBytes())))
        .field("rowkey", new String(result.getRow()))
        .endObject();
    bulkProcessor.add(new IndexRequest(params[5], params[6], Integer.toString(counter))
        .source(EsEntry));
}

```

```

        counter++;
        if(counter%50000 == 0)
            System.out.println(counter);
    }

    System.out.println("No of records read = "+counter);
    //closing the scanner
    scanner.close();
    bulkProcessor.close();
}
}
}

```

11.2. Code for Data Retrieval from HBase

```

package retrievingData;

import java.io.IOException;

public class RetrieveDataFromES {

    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub
        String[] params = new String[6];
        //Checking no of arguments
        if(args.length < 1){
            System.out.println("Usage:\n");
            System.out.println("Parameters 1-2 correspond to HBase");
            System.out.println("Parameters 3-6 correspond to Elasticsearch");
            System.out.println("mvn exec:java -Dexec.args=\"<table_name>,<column_value>,"
                + "<cluster_name>,<host_name>,<index_name>,<index_type>\"");
            System.exit(0);
        }
        else{
            String parameters = new String();
            for(int i = 0;i < args.length;i++){
                if(i != args.length - 1)
                    parameters = parameters + args[i] + " ";
                else
                    parameters = parameters + args[i];
            }
            params = parameters.split(",");
            long startTime = System.currentTimeMillis();
            Settings settings = Settings.builder()
                .put("cluster.name", params[2]).build();

```

```

TransportClient client = new PreBuiltTransportClient(settings)
    .addTransportAddress(new InetSocketTransportAddress
        (InetAddress.getByName(params[3]), 9300));

//Configuring Hbase
Configuration config = HBaseConfiguration.create();

//Connecting to Hbase
Connection conn = ConnectionFactory.createConnection(config);
Table table = conn.getTable(TableName.valueOf(params[0]));
/*
//Regular search
SearchResponse response = client.prepareSearch(params[4])
    .setTypes(params[5])
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setFetchSource(new String[]{"rowkey"}, null)
    .setQuery(QueryBuilders.queryStringQuery(params[1]))
    .defaultField("colvalue")
    .setSize(100)
    .execute()
    .actionGet();
SearchHit[] results = response.getHits().getHits();
*/

//Using the scroll API for searching Elasticsearch
SearchResponse scrollResp = client.prepareSearch(params[4])
    .addSort(FieldSortBuilder.DOC_FIELD_NAME, SortOrder.ASC)
    .setScroll(new TimeValue(60000))
    .setTypes(params[5])
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setFetchSource(new String[]{"rowkey"}, null)
    .setQuery(QueryBuilders.queryStringQuery(params[1]))
    .defaultField("colvalue")
    .setSize(10000)
    .get();

int counter = 0;
do{
    for (SearchHit hit : scrollResp.getHits().getHits()) {
        // For each rowkey obtained from Elasticsearch,
        // get the corresponding record from HBase

        Map<String, Object> result = hit.getSource();

        // Instantiating Get class
        Get g = new Get(Bytes.toBytes((String) result.get("rowkey")));

        // Reading the data
        Result res = table.get(g);
        // Reading values from Result class object
        byte [] value = res.getValue(Bytes.toBytes("det"), Bytes.toBytes("lname"));

        counter++;
    }
    scrollResp = client.prepareSearchScroll(scrollResp.getScrollId())
        .setScroll(new TimeValue(60000)).execute().actionGet();
}while(scrollResp.getHits().getHits().length != 0);

```

```
        System.out.println("No of records = " + counter);
        client.close();
        long endTime = System.currentTimeMillis();
        long totalTime = endTime - startTime;
        System.out.println("Total time taken = " + (float)totalTime/1000);
    }
}
```