Implementation and Evaluation of Scalable Data Structure over HBase

Vamshi Krishna Konishetty[†], Arun Kumar K[†], Kaladhar Voruganti[‡], G V Prabhakara Rao[†]

[†]Sri Sathya Sai Institute of Higher Learning, Prasanthi Nilayam, India
{vamshi2105,arunk786}@gmail.com, gvprabha@sssihl.edu.in

[‡]NetApp Inc., SanJose, USA
Kaladhar.Voruganti@netapp.com

ABSTRACT

With the emergence of commodity hardware architectures and distributed open source software, users are performing analytics on more types of data. Web 2.0 applications like social networking sites have to deal with a lot of meta-data which in some cases can't fit into main memory. Currently, it is the responsibility of the application programmers to manually map these in-memory data structures into persistent storage systems like a database or file system. Ideally, the application programmers would like the underlying programming language/middle ware software to seamlessly manage the scalable data structures.

It is increasingly becoming hard to use the traditional database or storage controller systems to store this meta-data because of cost and scale reasons. Thus, new NoSQL database architectures are emerging that are built on commodity hardware architectures and they can scale to large sizes in an incremental manner. Thus, there is an opportunity for the builders of NoSQL systems to provide scalable in-memory data structures. However, currently, these types of data structure interfaces are not available in the popular Hadoop NoSQL infrastructure. In this paper, we show how to implement the Set data structure and its operations in a scalable manner on top of Hadoop HBase. We then propose and implement optimizations for three Set operations. We also discuss the limitations of implementing this data structure in the Hadoop ecosystem. We evaluate our algorithms and optimizations on a real Hadoop cluster. Our primary conclusion is that the Hadoop ecosystem provides an excellent framework to implement scalable data structures.

Categories and Subject Descriptors

E.1 [**Data**]: DATA STRUCTURES—*Distributed data structures*; H.2.4 [**Information Systems**]: DATABASE MANAGEMENT—*Distributed databases*

ICACCI'12, August 3-5, 2012, Chennai, T Nadu, India.

General Terms

Experimentation, performance, optimization

Keywords

Hadoop, HBase, MapReduce, Distributed data strcture, Nosql

1. INTRODUCTION

With the emergence of commodity hardware and open source distributed software packages (e.g. Hadoop) there has been an increase in the building of large scale distributed applications that are leveraging these building blocks. For e.g. DropBox [1] file sharing application, LinkedIn [2] and facebook [3] social networking applications, real-time communication applications like Twitter [4] are emerging. Most of these applications have a lot of meta-data, and ideally would like to store the meta-data in memory in data structures like hash tables, lists, key-value stores, ordered sets etc. The scale of these data structures is so large that in many cases, the entire data structure cannot be stored in memory, and thus, the application builder has to manually map these data structures on to an underlying database (e.g. relational) or a file-system.

In many cases, the application builders are preferring scalable commodity hardware based architectures (NoSQL) that usually leverage open source software rather than use traditional, proprietary, monolithic database and storage servers to store this meta-data because these commodity based architectures are more cost-effective, and also scale better as the size of the data increases. However, there still exists an impedance mismatch in that the application programmers have to map their in-memory data structures to the persistent data abstractions (Key-Value data model) that are provided by these commodity NoSQL [15] architectures. Currently, systems like MemCache [5] and Redis [6] are being built that are trying to satisfy the above mentioned requirements. However, MemCache only provides Key-Value store interface, and Redis is not very much suitable for important data.

In this paper, we build scalable data structures on top of a commodity, open source NoSQL architecture.To achieve this, we analyze the differences between traditional database architectures and the emerging NOSQL database architectures, and provide a survey of some common NoSQL systems. We selected HBase as the desired NoSQL database

and present a detailed study of HBase, MapReduce and HDFS Hadoop [14] components that together provide the NoSQL infrastructure to implemented the Set data structure and its associated operations Union, Intersection and Difference on top of HBase (which is part of the Hadoop eco-system). Three optimizations were proposed and then, subsequently, analyzed the performance of our optimizations by implementing them in HBase on top of a real Hadoop cluster.

The paper is organized as follows. In section 2, we describe the existing systems which provide similar scalable datastructure implementation. Section 3 explains the HBase, MapReduce and HDFS Hadoop components that together provide the NoSQL infrastructure. Section 4 illustrates the design and implementation of scalable data structure and algorithms for operations on it, including optimizations proposed. Finally, section 5 summarizes experiments, the contributions of this paper and describes plans for further extensions.

2. RELATED WORK

We have selected HBase as the NoSQL database to build the scalable data structures because it is the most popular open source NoSQL system. For instance, very recently Hadoop and Hbase were used as core technology in Facebook for messaging facility[10]. However, there are other NoSQL systems available and we will now briefly describe them. There are a couple of NoSQL systems (Redis and Hazelcast) that also provide scalable data structure interfaces. We will describe these two NoSQL systems in this section

2.1 Redis

Redis[6] is an open source NoSql type database in which data is stored by employing an advanced key/value store mechanism. It is sponsored by VMware and is fully implemented in C language. The major difference between Redis and other storage systems is that each key can not only have a string as a value, but even a set or sorted set or list or hash can be the values. On the other hand, it supports high level atomic server side operations on the data structures such as set, hash, list and sorted set.

The outstanding performance by Redis is achieved with the in-memory data store, whose persistence is attained in two different ways. One is snapshotting, and is a semi-persistent durability mode where the dataset is asynchronously transferred from memory to disk from time to time. The second and safer alternative is an append-only file (a journal) that is written as operations modifying the dataset in memory are processed. Redis is able to rewrite the append-only file in the background in order to avoid an indefinite growth of the journal. Some other features include a simple check-and-set mechanism, pub/sub and configuration settings to make Redis behave like a cache. Though it employs master/slave relation among nodes within cluster, their roles can change dynamically.

2.2 Hazelcast

Hazelcast[8] is an open source based highly scalable data distribution platform for Java. It can perform thousands of operations per second while accommodating the facility to recover safely during failure situations. It supports memcache, a distributed caching system, and the size of Hazelcast cluster can be dynamically increased it ensures that data is spread uniformly across all nodes automatically. Various distributed data structures like map, queue, topic, list etc are implemented very efficiently that can be directly used by distributed applications.

As data is uniformly partitioned on all nodes in the cluster, each node holds (1/n * total-data) + backups , n being the number of nodes in the cluster. If a node goes down, the replica with the same data gets re-distributed to remaining live nodes, as a result of which no data gets lost. Since nodes are not classified as master/slave, there is no single point of failure, moreover, all the nodes have got equal importance and responsibilities.

3. ARCHITECTURE

The complete work was carried on Hbase infrastructure that makes use of Hadoop MapReduce and HDFS components.

3.1 HBase Infrastructure

HBase[9] is an open source NoSql database developed under Apache Software Foundation. It is built as a database layer on the top of Hadoop Distributed File System. As it being modeled after Google Bigtable, most of the functionalities and implementations are similar to Bigtable[7]. Usually in applications where there is a requirement of hosting very large tables which are sparse, distributed on thousands of machines, and have billions of rows and millions of columns, HBase is the perfect solution to deal with such data-intensive applications. The data model employed in Hbase is similar to Bigtable. Flexibility of adding columns on the fly unlike relational databases is provided. There exist schema design options such as Bloom Filters, Table-configured region sizes, compression, and block sizes that all affect the performance.

There exist two Catalog Tables in HBase architecture named -ROOT- and .META. that have got utmost importance. The .META. table keeps a list of all regions in the system where as -ROOT- keeps track of where the .META. table is. The information about the location of all regions and region servers is maintained with the help of these two tables. The table is physically divided into individual partitions called *Regions*. Each region is exactly served by a single *Region server* and which, in turn, serve the clients with the requested data. A Region server is a physical machine within the HBase cluster that hosts multiple number of regions on it. The regions are internally stored on the HDFS. Though the HDFS provides reliable and fault tolerant storage, it does not provide fast lookups to the records present within the files. Thus, HBase keeps data in indexed StoreFiles on HDFS and this is primarily responsible for high-speed record lookups.

Each Region holds a Store that hosts a MemStore and zero or more StoreFiles (HFiles). A single Store corresponds to a column family for a table for a given region. MemStore holds in-memory modifications to the Store. StoreFiles are those where the actual data resides. Each Region server maintains a special HLog file WAL (Write Ahead Log) which ensures that HBase writes are durable. HMaster is the implementation of Master node, which usually runs on the same node where HDFS Name node runs in the cluster. HMaster has some crucial responsibilities such as Region Server failover,

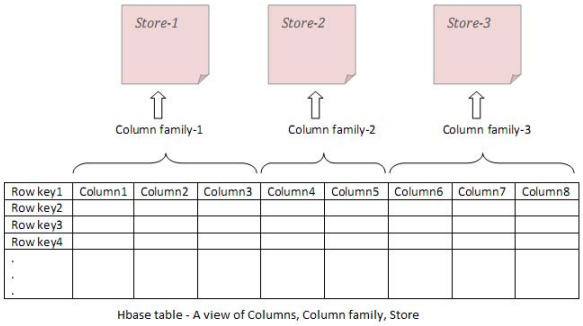

Figure 1: HBase table column, column family storing

Region splitting, Periodic Load balance, Regular checking and cleaning up of .META. table. The initial interaction of client takes place by querying the -ROOT- and .META. to obtain the desired data location. This interaction does not involve Master, rather zookeeper provides the location of -ROOT- to the client. Even Mapreduce is supported by Hbase infrastructure, where Hbase tables can act as either source or sink to the Mapreduce jobs.

3.2 MapReduce Framework

The MapReduce[11] is a powerful programming model introduced by Google in the year 2004. The main motive of this model is to process huge datasets that spread across multiple number of machines of a cluster. The interesting fact about MapReduce is that, the computation is moved to a place where data resides rather than moving data across the network to node where application is running and causing a lot data transfer.

In MapReduce, the input data-sets are split into chunks which are processed in parallel by number of tasks in a fault tolerant manner. The programs coded in this style are parallelized automatically and executed by the MapReduce framework. The burden of input data partitioning, program's execution scheduling across the nodes of the cluster, handling node failures and inter-node communication etc are taken care by the MapReduce framework.

MapReduce framework has one master referred to as *Job tracker* (one per entire cluster) and slaves known as *Task Trackers*that reside on per each node in the cluster[12]. Both Jobtracker and Tasktrackers run as MapReduce daemons over Hadoop distributed file system. In the typical Hadoop cluster, JobTracker and namenode present on the same node while TaskTracker and datanode present on the same nodes. Figure 2 clearly explains how HDFS organized among nodes of the cluster and the way Mapreduce daemons running on HDFS. Hadoop tasks typically fail due to network and hardware failures. The HDFS datanode corresponds to data storage, MapReduce task tracker corresponds to computation and both of these typically stay together on the same node of the cluster.

Usually the MapReduce job's input/output locations are specified by applications along with the map/reduce functions that implement the required interfaces. The map function processes the input key/value pairs, generating intermediate key/value pairs which in turn are fed to the reduce

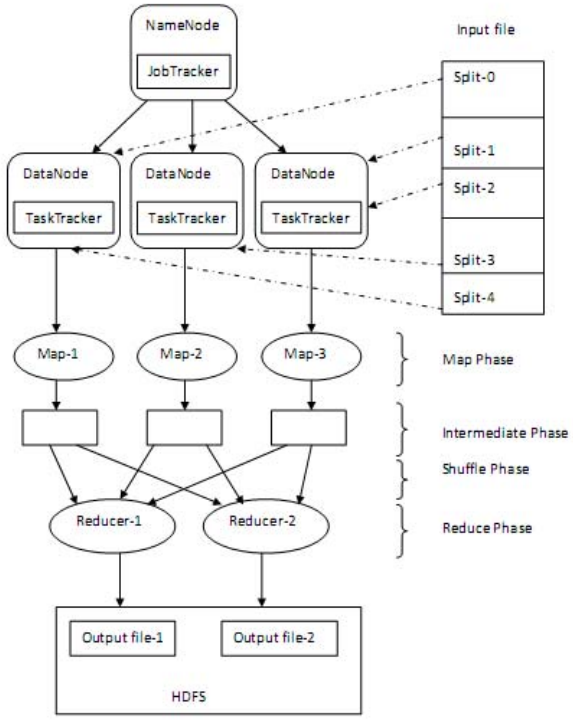

Figure 2: A view of Mapreduce daemons running on HDFS

task to combine and process. Each MapReduce job can be configured according to the needs of the application.

4. ALGORITHM

This section focuses on the design details of the Set data structure and the corresponding Union, Intersection and Difference set operators.

Since data can only be stored HBase in a table, we design the Set data structure over HBase by mapping it to a HBase table. The Set is mapped to an HBase table where as all the elements of the set are mapped to the rows within the HBase table, and thus, provide the abstraction of set to the user or application programmer. The user will not know how the data is stored internally within HBase and he/she will not be aware of the physical location from where data is retrieved.

As the set data structure is mapped to HBase table which gets distributed across nodes, the set becomes a distributed data structure. Moreover, since the set data structure is implemented over the HBase table, it can scale up to thousands of physical servers, and thus, becomes a very scalable data structure. The basic nature of set i.e 'all the set elements are unique' is assured by maintaining at most one version for each data cell in HBase table.

Various operations that can be performed on set include

- Set Union
- Set Intersection
- Set Difference

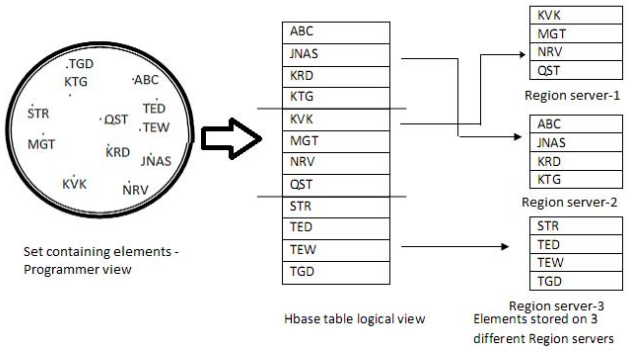

Figure 3: Set abstraction over HBase table

4.1 Set Union

Set union results in a new set, which contains all elements from all the sets on which the operation is applied. Here we design the union of two sets, where the elements are assumed to be normal strings. That is, all the elements correspond to the rows within the table, where the row keys are the elements themselves.

Algorithm 1: Set union

```
1  s1,s2 are scanners attached to Tables T1 and T2.
2  r1, r2 are the row keys of the rows fetched by s1 and
   s2.
3  for r1=s1.next (), r2=s2.next (); (r1 is not null)
   OR (r2 is not null) do
4      if r1 is not null then
5          put r1 in to Resultant table.
6          Fetch next row from table T1 into r1.
7      end
8      if r2 is not null then
9          put r2 in to Resultant table.
10         Fetch next row from table T2 into r2.
11     end
12 end
```

In the programmer's perspective Set union is implemented as copying two tables into other table ensuring that all the rows in the resultant table are unique to satisfy the basic condition on set elements. Here we implement the set union operation in a serial fashion i.e the rows from each table are fetched one after other and are directly inserted in to the resultant table. But according to the fact that all the rows within the HBase table are sorted, each time a row is inserted in to the resultant table from different table, a sort operation is initiated to retain the sort order on the rows of table.

. On the other hand, foreach row to fetch from (or insert to) a table, which is distributed and all its regions are physically located in distinct servers, an RPC (Remote Procedure Call) has to be invoked by the client.

4.2 Set Intersection

The intersection operation on multiple sets will give the elements common to all the sets on which it is applied. Here we check the intersection of two sets where the elements are assumed to be normal strings. Set intersection as seen by the programmer is implemented by finding all those rows which are common to the given multiple tables and put them in a single resultant table. There are two ways in which set intersection algorithm is implemented and both are serial implementations.

Algorithm 2: Set Intersection-1

```
1  s1,s2 are scanners attached to Tables T1 and T2.
2  r1, r2 are the row keys of the rows fetched by s1
   and s2.
3  for r1=s1.next (), r2=s2.next (); (r1 is not null)
   AND (r2 is not null) do
4      if r1 is equal to r2 then
5          put r1 in to Resultant table.
6          Fetch next row from table T1 into r1.
7          Fetch next row from table T1 into r2.
8      end
9      else if r1 is less then r2 then
10         Fetch next row from table T1 into r1.
11     end
12     else
13         Fetch next row from table T2 into r2.
14     end
15 end
```

In the first implementation, the each row from two different tables are fetched and checked for the condition to see if they are equal. If they satisfy, the corresponding row is inserted into resultant table. Otherwise, the next rows are fetched from appropriate table and the process of checking for intersection condition is continued till the rows in smaller table are exhausted. This is because the maximum possible size of a set intersection on two sets is the size of the smaller table.

Algorithm 3: Set Intersection-2

```
1  T1,T2 are Tables.
2  r:row key of Table.
3  foreach r:row key in T1 do
4      if r exists in T2 then
5          put r in the resultant table.
6      end
7  end
```

The second implementation of set intersection is carried by fetching a single row from one table, then the intersection condition is checked between this row and all the rows present in another table. If any row is found to satisfy the condition, that row is inserted into resultant table. In a case when no such row is found, then same procedure is carried for next row from the first table. In this case the table chosen for scanning each row should be smaller than the table in which the row's existence is checked. This is because of the fact that, irrespective of the size of the table, a row can be checked if it is present in table. But the speed of scanning is based on the size of table. Hence for scanning we use small table and for lookup for each row, we use the bigger table.

4.3 Set difference

The Set Difference operation outputs those elements present in the set left to the set difference operator and not present in the set that falls on right of the operator.

In this algorithm, the scanning of only one table takes place, while the other table acts as a lookup table.

Algorithm 4: Set Difference

```
1 T1,T2 are Tables.
2 r:row key of Table.
3 foreach r:row key in T1 do
4 |   if r does not exists in T2 then
5 |   |   put r in the resultant table.
6 |   end
7 end
```

The Set Difference operation in the programmer's view will be, finding the rows from first table but that does not exist in the second table, and putting them in the resultant table. This operation is implemented similar to the way how the second algorithm of intersection operator was implemented as discussed above. But the difference comes in the condition being checked that is, for each row in one table, its existence is checked on the other table. If for any row a corresponding row in second table has been found, then that row is not inserted into the resultant table. In the other case, the same procedure is repeated for the next row fetched from first table. Processing of tables is chosen in such a way that, table being scanned for each row should be smaller than table in which lookup of rows take place.

4.4 Optimizations

We use three optimizations to improve the performance of the algorithms that were presented so far. These optimizations are:

- Caching the Rows
- Batch Processing
- Using MapReduce

Caching the Rows

In all the serial implementations, when a row is to be fetched, a request is sent to a remote location where the region server serving this particular row resides. Usually in the applications or the situations where an entire tables (containing potentially millions or even billions of rows) data has to be scanned this kind of row fetching will be a very tedious task because of repeated requests to fetch the rows from the nodes that are physically apart. Furthermore, since all the data is stored persistently within the disk on nodes, for every request a disk access is required, thus adding the IO latency to the execution time.

Therefore, to be free from all such factors that will adversely affect the performance, we facilitate caching of the rows on the client machine itself. The caching here is the number of rows that can be fetched in a single request from a remote server and stored on the client machine. So, for specified number of subsequent rows, explicit requests need not be sent as they are already available at the client. Increasing the number of rows being cached will increase the performance but at the cost of more RAM utilization. Huge amount of main memory will be consumed and the performance of other running applications can, in turn, become low. Hence a careful decision has to be taken in order to obtain the optimal performance with a proper balance between speed of execution and memory requirement.

The main application of caching is found where often mass scanning of very large tables takes place.

Batching the Number of put operations

The applications where a lot of operations have to be performed that require RPC a lot of network bandwidth is utilized. Many times network links would be accessed causing high traffic on the channels. Additionally the long standing persistent connection introduces network overhead leading to performance degradation. Therefore without compromising on the performance as well as avoiding network overhead, we have to come up with a method that enhances the performance by reducing the latency to a greater extent.

For satisfying the above mentioned criteria for better performance, we have come up with yet another optimization. The performance is optimized by leverage the inbuilt facility of HBase i.e batch processing. In all the implementations of various algorithms so far discussed, the important and widely used operation that requires RPC is the put operation. Hence whenever there is a need for performing more number of put operations, we group the put operations into a batch and then all those put operations are executed via a single RPC. So, we have to consciously choose the number of put operations that could be batched together to provide optimal execution time.

MapReduce

MapReduce is a special approach that deals with parallel processing of huge data, and when it is started, many parallel running tasks execute and do the required processing on data. Multiple MapReduce jobs can be run in parallel and this leads to multiple tasks running in parallel for each job, thus contributing towards more parallelism. Here we show the Mapreduce implementation of all set operations one per each. In all the MapReduce implementations, we preferred *capacity scheduler* rather than Hadoop default scheduler, because, default scheduler submitted jobs in serial manner, due to which parallelism was not observed.

Algorithm 5: Set union-1

```
Map (K: key, V: value)
Insert this key to the Result table
```

Two MapReduce jobs, will simultaneously scan two different tables and insert the keys from those tables into a single resultant table. These jobs are Map-only jobs and they do not have reducer. The duplicate keys in the resultant table are dealt with the number of versions maintained in table. Since we fixed the maximum number of versions to be maintained as 1, only the latest value will be stored. The two jobs will have the same map method as shown in Set union-1.

It is the mapreduce implementation of 'Algorithm 3: Set Intersection-2', which is a serial implementation. But, because of the parallelism introduced by mapreduce, the performance of same algorithm increases to a greater extent.

Algorithm 6: Set Intersection-3

```
1 T1,T2 are Tables.
2 r:row key of Table.
3 For Intersection of T1 and T2.
4 T1 is input for Map
5 Map (K: key, V: value)
6 if K is present in T2 then
7 |  Insert the K into resultant table.
8 end
```

Algorithm 7: SetDifference-2:Phase-1

```
1 T1, T2 are tables
2 For T1-T2 (i.e T1 Set Difference T2)
3 Map (K: key, V: value)
4 Append Table name to the K with a
  delimiter in between.
5 Store it in a file-i
```

The N number of Map-only jobs will run with the map method shown in Phase-1. The job-i will store its key in the file-i.

Algorithm 8: SetDifference-2:Phase-2

```
1 Map (K: key, v: value)
2 Split the value based on the delimiter
3 Store the tokens in v1 and v2
4 Now pass the pair v1 and v2 to map
  output file
5 Reduce (K: key, L:List of values)
6 foreach v in L do
7 |  Increment the counter
8 end
9 if (counter = 1) AND (v and T1 are
  same) then
10 |  Insert the key in to resultant table.
11 end
```

After N jobs finish their execution, one job starts running by taking file-i, ∀ i=1 to N , as input.This job has map and reduce methods as shown in Phase-2.

4.5 Probable Potential Optimizations

We have not implemented the optimizations being proposed in this section. Looking at various aspects of HBase and MapReduce architecture, we propose new optimizations that could potentially improve the performance:

1. In this scheme, a new variant of storage mechanism is proposed. According to the Hbase architecture[13], the blocks of a single column family would be distributed among different data nodes of HDFS and this adds extra latency to performance because multiple network accesses are performed as illustrated in above section. we can optimize this by reducing multiple requests and multiple network accesses. This would be possible by making all the blocks of same column family to be stored on a single HDFS data node rather than on multiple data nodes.

   Hence for any request of data that belongs to the same column family, with just a single request to a single HDFS Data node, we can retrieve the desired data. Though we have not implemented it yet practically, we anticipate that if this new scheme is implemented, the performance would really improve.

2. Yet another potential idea that can improve the performance is, to increase the number of reduce tasks automatically on the nodes where the size of data to be processed is greater than the data on other nodes. Usually when a table is stored in the HBase cluster, it is divided into regions that spread across multiple nodes. If we assume that each table is completely stored on a single Region server, i.e no span of tables across the nodes, we can consider a scenario, where rows from multiple tables are selected based on a criteria and inserted to other table. The insertion is performed by Reduce tasks that run on all nodes. At this point, our idea requires the *Selectivity factor*[1] of *select* operation on each table to be known, and the region server hosting that table also must be known before hand. This enables us to increase the number of reduce tasks on that particular region server, where selectivity factor is higher. This, in turn, triggers increased parallelism on such nodes, and the overall execution time decreases.

4.6 Limitations

We found some limitations with respect to optimizations as well as HBase, MapReduce frameworks.

- When the caching is increased beyond certain value, it has resulted poor performance as the memory consumed rose to a level where other applications could not get the sufficient memory and thus their execution time increased.

- An increase in the number of put operations in a batch beyond certain value, lead to performance degradation. The reason is due to the RPC overhead.

- In HBase every single column file is divided into blocks that are stored on a different HDFS data node. So when there is request for data belonging to the same column family, the data blocks have to be fetched from multiple HDFS data nodes and this results in multiple requests being sent to multiple nodes in the cluster. This leads to heavy network traffic overhead. Therefore, this type of *mapping the blocks of HBase column family to different HDFS nodes* [13] added extra latency to actual execution time.

- Lack of Multi table input feature in the combined HBase and Mapreduce environment.

[1]The number of rows within the table that satisfy a specified criteria.

5. EXPERIMENTS AND RESULTS

In this section, we evaluate the different implementations and optimizations of the Set data structure operators Union, Intersection and Difference.

5.1 Cluster Configuration

To conduct various experiments to evaluate the performance of algorithms, we have set up a 3 node Hadoop cluster where all nodes are running on Ubuntu-10.04 LTS 32-bit operating system. Out of the three nodes, two nodes are of similar configuration each having Pentium (R) Dual-Core CPU 2.20 GHz (1024 KB cache), 4 GB RAM, 320 GB disk space. The third node is a laptop with Pentium (R) Dual-Core CPU 2.00 GHz (1024 KB cache), 3 GB RAM, 250 GB disk space. All these nodes are connected with 1 Gbps Local Area Ethernet Network. The Hadoop software environment for the experiments being conducted consists of Hadoop-0.20.2, HBase-0.90.4 with embedded zookeeper, Eclipse Helios IDE for development, java 1.6.0_26.

5.2 Dataset And Workloads

For our experiments we have selected Set elements to be a string. We have created a dataset of size 1.5 GB, which contains almost 16.5 million random strings. Each string is a combination of alphanumeric characters and each element of set has a size of 10 Bytes. The workload for our experiments includes searching in the very large table, sequential fetching of rows from tables, random insertion of multiple rows into tables which cause lot of RPC overhead and rigorous sort operation (which is an implicit workload initiated by every put operation on HBase table).

The performance of various algorithms is evaluated based on the time of execution obtained. There are three set operations implemented on two very large sets each having millions of elements. All the experiments were conducted three times and the average of those are selected as the final results. We have baseline implementations as well as optimizations applied on all the operations of set. In each of the three operations, we show different implementations covering all types of implementations.

5.3 Set Intersection

Out of three different variants of set intersection implementations, we show the performance with respect to basic (serial) implementation as well as two optimizations. The performance of set intersection is evaluated for different values of two configurable parameters. The parameters to which the set intersection executuion time is sentive are cache and batch.

No. of put operations per 1 RPC	Number of rows cached	Time of execution in sec.
1	1	10555
1	10	5441
1	100	4576
1	1000	4564
1	10000	4509
1	100000	1941
1	1000000	4499

Table 1: Algorithm 2: Set Intersection-1 implementation for varied caching

In the graph shown in Figure 4, performance of set intersection for different number of rows cached is shown. X-axis represents the Number of rows cached (in powers of 10) and Y-axis represents execution time.

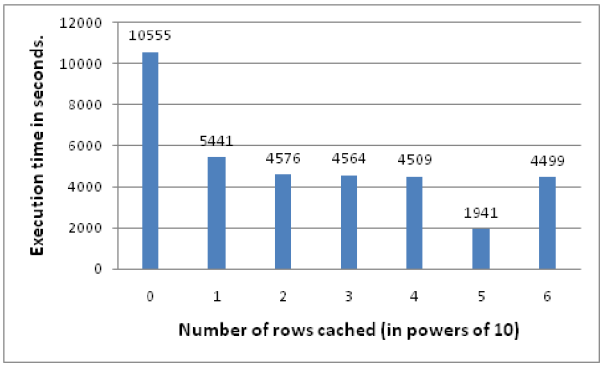

Figure 4: Set intersection for different cache values

In graph shown in Figure 4, we observe a drastic fall in execution from first case to the second case because the cache size is increased by 10 times. The reason for this is addressed by an operating system issue i.e caching at the client machine. The default block size in the HBase is 64 KB, which is the basic unit of data that can be accessed or transferred. The first bar corresponds to caching of only one row. So for every row fetched in the table, a data block of 64 KB has to be transferred from remote location, but only one row is read from that entire 64 KB block and for subsequent row fetch, one more 64 KB block is read. In this process except one row, the rest of the data in the block is not useful and is unnecessarily transferred and hence it took extra time for execution. In second case, for every row fetch request, 10 rows are cached, i.e within the obtained 64 KB block which is transferred from remote location, 10 rows are useful and read. Hence here for every 10 rows a 64 KB block is transferred unlike in the first case. The decrease in time from first case to second is 51.5%.

The difference in times is not significant from 2nd case to 5th case, this is because in all these cases, as the specified number of rows being cached can fit in to a single 64 KB block obtained, for every row fetch request a single block of 64 KB is transferred. On the other hand, again from 5th case to 6th case there is 43% decrease in time. This is due to the row fetch call, specifying large number of rows to be cached. This large number of rows will not be fit in a single 64 KB block. Hence for a single row fetch request, in order to cache the specified number of rows, nearly fifteen 64 KB blocks are transferred and for the next very large number of rows, no block transfer is required. Hence very less number of block transfers occurs giving the optimal performance. And for the next cases, as the caching size increased, more amount of memory is eaten up resulting in more number of misses within the cache and this results in an increase in execution time that eventually results in thrashing behavior. For example in a case where caching was 10000000, the program terminated because of above mentioned reason.

In the graph shown in Figure 5, set intersection performance is compared for different values of RPC batch size. On X-axis the Number of put operations batched per an

Rows per 1 RPC (batch)	Rows cached	Execution time in sec.
1	1	87240
100	1	52065
1	100000	38688
100	100000	5732

Table 2: Algorithm 1: Set Union Serial Implementation

RPC (in powers of 10) is labeled and on Y-axis the time of running in seconds is labeled.

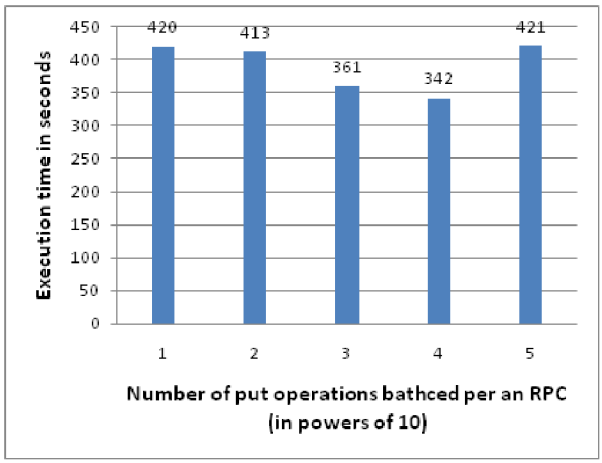

Figure 5: Set intersection for different batch values

The first case in the graph shown in Figure 5, has the highest time for execution (among first four cases), because, for every put operation, one RPC (Remote Procedure Call) is invoked and put is called for every row in the table. From the first to fourth cases, the time taken kept on decreasing, because, for every single RPC, multiple number of put operations are taking place. It is possible with a special method available within HBase, that provides batch processing capability to reduce the number of RPC calls. So, multiple put operations are grouped into a batch and then that batch of operations are performed in a single RPC. Hence the number of RPCs is reduced for each case. Therefore, this results in a decrease in the execution time. But, contrary to this usual behavior, in the last case the time taken increased from the previous case. The cause for this is addressed from the view of network issues. In this case, within a single RPC connection, very high number of put operations are being performed. So, the time taken for performing this large number of operations in single RPC is more, because of various network factors such as the existence of persistent network connection for long time makes a fraction of whole bandwidth not available for other network communications for quite some time.

5.4 Set Union

Among the two implementations of set union, we show the performance serial implementation with two optimizations applied at a time. That is the behavior of set union for varied combination of batch and catch.

In this graph various combinations of RPC batching and cache size values are labeled on X-axis and the execution time in seconds is labeled on the Y-axis. The batch corresponds to the number of Put operations that take place per a single RPC and cache size corresponds to the number of rows fetched in each access to server.

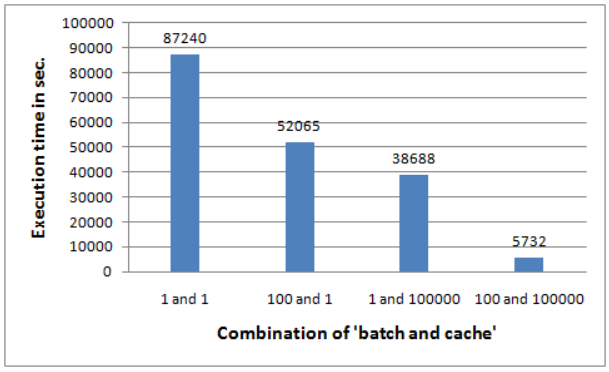

Figure 6: Set union serial implementation for different batch and cache values

In the bar graph shown in Figure 6, the first bar represents the basic implementation of set union operator, i.e every row has to be fetched from the remote location, and this incurs a lot of network access latency, and batch value of 1 corresponds to case where for every RPC only one row is inserted into table. So for a very large table if for every row an RPC is invoked this causes very high RPC overhead. The specific cache and batch values are chosen because they gave minimum execution time for most of the algorithms. The last and the smallest histogram bar shows the least execution time for optimal combination of cache and batch. The drastic difference in the times of execution between this and first bar is caused by the fact that, in this case 100000 rows are fetched in a single access to remote table. This, in turn, reduces the remaining 99999 remote accesses to fetch the subsequent rows, and, added to that, within a single RPC invocation 100 rows are being inserted unlike in the basic implementation, thus reducing the rest of 99 RPCs and this, in turn, saves a lot of time.

5.5 Set Difference

The performance of three types of implementations for set difference are evaluated. The two of which are serial and mapreduce implementations of a same algorithm (SetD1) and third is the separate mapreduce implementation (SetD2).

Implementation type	rows cached	put operations per 1 RPC	Execution time in sec.
Set Difference-3	100000	100	6541
Set Difference	100000	100	5875
Set Difference-2	100000	100	1541

Table 3: Different Implementations for Set difference

In the graph shown in Figure 7, three types of set difference implementations are compared based on time of execu-

tion. X-axis shows the type of implementation and Y-axis shows the execution time.

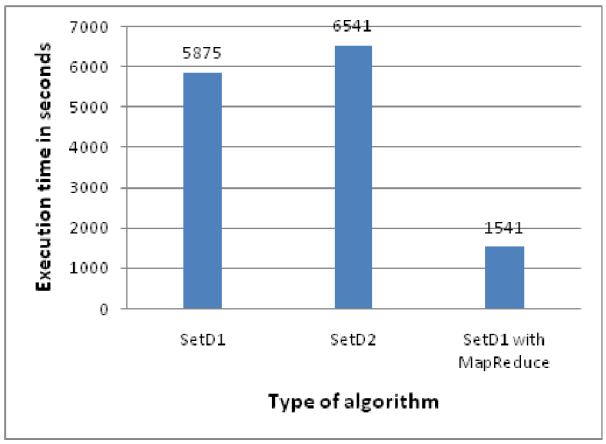

Figure 7: Comparison of different Set difference implementations

The setD2 (Set Difference2) takes the longest time because of the reading from file/table and writing to file/table are involved, causing lot of IO latency. In SetD1 (Algorithm 4: Set Difference), for every row in one table, it is checked for its existence in other table. So, this is a serial implementation where number of search operations is equal to number of rows in the table being scanned. Third case (SetD1 with Mapreduce) is MapReduce version of the SetD1, and hence because of parallelism provided by multiple simultaneously running map tasks, the time taken gets drastically reduced.

5.6 Special observation

In the graph shown in Figure 7, the execution speeds are very much different for serial and MapReduce versions of the implementations, where the algorithm and parameters i.e catch, batch are fixed in both implementations. In addition to the variation in execution speeds of the two approaches, the difference in the resource utilization between two implementations is very significant.

6. CONCLUSION AND FUTUREWORK

Working with HBase and HDFS was difficult because of lack of proper documentation and support. Numerous hours were spent in trying to properly configure the system and support from Hadoop community was sporadic at best.

In this paper we present an overview of NoSQL databases, and then we provide the technical details of some common NoSQL systems. We subsequently implemented the Set data structure on top of HBase. We also implemented some optimizations to the set operators. We evaluated our optimizations on a real Hadoop cluster, and we also discuss the disadvantages of the current Hadoop infrastructure for implementing these scalable data structures.

In this paper, we show that it is possible to implement scalable data structure like Set on top of the Hadoop HBase infrastructure. The advantages of using HBase are 1) it allows one to leverage the inherent parallelism that is available in the MapReduce framework and 2) one can seamlessly build a scalable data structure since HBase region server architecture can scale across multiple nodes. However, it is important to note that this scale is obtained at the cost of not having a strong consistency model. It is also important to note that HDFS was initially built to support the MapReduce framework, and thus, it was not designed to support low latency requirements. Thus, HBase nodes need to use large caches in order to try and cache a large part of the table working set (data structure working set).

In future, we plan to implement other data structures like lists, hash tables, ordered sets etc on top of HBase. We would also like to see whether it is possible to use an existing file system like ZFS to potentially get better performance than HDFS. We would also like to evaluate whether it is possible to build these scalable data structure with strong consistency semantics. Finally, we would like to evaluate the performance of these scalable data structure on top of HBase with Redis and other NoSQL systems.

7. ACKNOWLEDGMENTS

We dedicate this work to the Founder Chancellor of our university, Bhagawan Sri Sathya Sai Baba.

8. REFERENCES

[1] http://www.dropbox.com/
[2] http://in.linkedin.com/
[3] http://www.facebook.com/
[4] http://twitter.com/
[5] http://memcached.org/
[6] http://redis.io/
[7] Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Robert E. Gruber Fay Chang, Jeffrey Dean, "Bigtable: A distributed storage system for structured data," .
[8] http://www.hazelcast.com/index.jsp
[9] http://hbase.apache.org/
[10] Karthik Ranganathan Samuel Rash Joydeep Sen Sarma Nicolas Spiegelberg Dmytro Molkov Rodrigo Schmidt Jonathan Gray Hairong Kuang Aravind Menon Amitanand Aiyer Dhruba Borthakur, Kannan Muthukkaruppan, "Apache hadoop goes realtime at facebook," .
[11] Sanjay Ghemawat Jeffrey Dean, "Mapreduce: Simplified data processing on large clusters," .
[12] http://hadoop.apache.org/common/docs/current/mapred_tutorial.html
[13] http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html
[14] http://hadoop.apache.org/
[15] http://nosql-database.org/