# HBase and Hypertable for large scale distributed storage systems

A Performance evaluation for Open Source BigTable Implementations

Ankur Khetrapal, Vinay Ganesh
Dept. of Computer Science, Purdue University
*{akhetrap, ganeshv}@cs.purdue.edu*

## Abstract

BigTable is a distributed storage system developed at Google for managing structured data and has the capability to scale to a very large size: petabytes of data across thousands of commodity servers. As now, there exist two open-source implementations that closely emulate most of the components of Google's BigTable i.e. HBase and Hypertable. HBase is written in Java and provides BigTable like capabilities on top of Hadoop. Hypertable is developed in C++ and is compatible with multiple distributed file systems. Both HBase and Hypertable require a distributed file system like Google File System (GFS) and the comparison therefore also takes into account the architectural differences in the available implementations of GFS like systems. This paper provides a view of the capabilities of each of these implementations of BigTable, and should help those trying to understand their technical similarities, differences, and capabilities.

## Introduction

Implementing distributed, reliable, storage-intensive file systems or database systems is fairly complex. These systems face several challenges like data placement algorithms, cache management policies for quick retrieval of data, provide a high degree of fault-tolerance because of deployment over thousands of nodes, scalability and security to some extent.

The key motivation behind systems like BigTable is the ability to store structured data without first defining a schema provides developers with greater flexibility when building applications, and eliminates the need to re-factor an entire database as those applications evolve. BigTable allows you to organize massive amounts of data by some primary key and efficiently query the data.

The HBase project is for those whose cannot afford Oracle license fees or whose MySQL install is starting to buckle because tables have a few blob columns and the row count is heading north of a couple of million rows. HBase is for storing huge amounts of structured or semi-structured data.

## Related Work

Google's BigTable was not the first solution towards the problem of managing structured data in a distributed environment. The problem has been widely researched and there exist a number of generic and specific solutions in the industry as well as academia. Microsoft's Boxwood Project, developed in C# and C, provides components with overlapping functionality with Google's Chubby Lock Service, GFS and BigTable. However, Boxwood is a research project and there are no performance comparisons available for any large deployments of the Boxwood Project.

Mnesia is a distributed Database management system and provides and extremely high degree of fault tolerance. Mnesia provides a large number of features such as distributed storage, table fragmentation, no impedance mismatch, no GC overhead, hot updates, live backups, and multiple disc/memory storage

options. Mnesia is developed in Erlang and layers on top of CouchDB to provide BigTable like features.

Dynamo is a distributed storage system by Amazon however; it focuses on writes as compared to BigTable that focuses on reads and assumes writes to be almost negligible. SimpleDB is another service from Amazon that offers BigTable like functionalities. However, Bigtable values are an uninterpreted array of bytes and SimpleDB stores only strings; SSDS has string, number, datetime, binary and boolean datatypes.

## HBase

### Introduction

HBase is an Apache open source project whose goal is to provide Big Table like storage. Data is logically organized into tables, rows and columns. Columns may have multiple versions for the same row key. The data model is similar to that of Big Table. There are a few differences in HBase from Big Table. Currently with HBase, only 1 row at a time can be locked. The next version will allow multi row locking. SSTable is called HStore in HBase and each HStore has 1 or more MapFiles which are stored in HDFS. Currently these MapFiles cant be mapped to memory. HBase identifies a row range by table name and start key where as in Big Table it uses the table name and the end key.

### Requirements

HBase requires java 1.5.x and Hadoop 0.17.x. ssh must be installed and sshd must be running to use Hadoop's scripts to manage remote Hadoop daemons. The clocks on cluster members should be in basic alignments. Some skew is tolerable but wild skew can generate odd behaviors. All the table data is stored in the underlying HDFS.

### Architecture Overview (Implementation)

There are three major components of the HBase architecture:

1. **HBaseMaster.** The HBaseMaster is responsible for assigning regions to HRegionServers. The first region to be assigned is the *ROOT region* which locates all the META regions to be assigned. The HBaseMaster also monitors the health of each HRegionServer, and if it detects a HRegionServer is no longer reachable, it will split the HRegionServer's write-ahead log so that there is now one write-ahead log for each region that the HRegionServer was serving. After it has accomplished this, it will reassign the regions that were being served by the unreachable HRegionServer. In addition, the HBaseMaster is also responsible for handling table administrative functions such as on/off-lining of tables, changes to the table schema (adding and removing column families), etc.

2. **HRegionServer.** The HRegionServer is responsible for handling client read and write requests. It communicates with the HBaseMaster to get a list of regions to serve and to tell the master that it is alive. Region assignments and other instructions from the master "piggy back" on the heart beat messages.

3. **HBase client.** The HBase client is responsible for finding HRegionServers that are serving the particular row range of interest. On instantiation, the HBase client communicates with the HBaseMaster to find the location of the ROOT region. This is the only communication between the client and the master.

## Evaluation

### Observations

HBase has a new Shell which allows you to do all the admin tasks which include create, update, insert, etc. commands. The row counter is very slow. When updates were made to the table, say for example when rows of the table were deleted; the size of the table in the HDFS used to increase. This is mostly because of the fact that Major compactions occur with less periodicity. So the changes do not reflect as expected immediately.

**System Configuration**

The machine used for the single node evaluation of HBase had an Intel Core2 Duo – 2 GHz processor with 3 GB memory and 200 GB of secondary storage was available. Scripts for cause random/sequential read/write were implemented to evaluate the performance of HBase. We also used the performance evaluation scripts that were already made available with HBase the tests. Performance was monitored on the standalone setup only.

All the evaluations were done using one HRegionServer. HBase performed well and as expected for most of the tests performed. In some instances it scaled poorly and overall performance is still several orders of magnitude worse than BigTable.

**Performance of the Scanner**

HBase provides a cursor like Scanner interface to the contents of the table. When one doesn't know the row you are looking for we can use this. We can configure the number of rows per fetch in the hbase-default.xml file. This corresponds to the number of rows that will be fetched when calling next on the scanner if it is not served from the memory. The performance for the Scanner was thus tested for different values of rows per fetch. The following results were obtained

| **Rows per fetch** | **Rate of row fetch** |
|---|---|
| 1 | 1600 rows/second |
| 10 | 9000 rows/second |
| 20 | 18000 rows/second |

Thus it is seen that the performance of the scanner improves significantly by configuring the number of rows per fetch to a larger number. This can be attributed to the fact that by increasing the number of rows per fetch, we are reducing the number of RPC calls made significantly – hence better rates observed. Higher caching values will enable faster scanners but will eat up more memory and some calls of next may take longer and longer times when the cache is empty.

**Scaling the column families**

(Note :- This test was carried out by Kareem Dana at Duke University over a year ago. The same is performed on a newer version of HBase now by us.)

A table having a specified number of column families was created and wrote 1000 bytes of data into each column family. After creating the table and adding data into it, random reads were performed across the different column families. Then we tried to carry out sequential updates to the data in these column families. The following results were observed.

| Number of column families | 100 | 300 | 500 | 550 |
|---|---|---|---|---|
| Reads/Sec | 170 | 165 | 170 | Timeout |
| (Sequential)Writes/sec | 250 | 250 | 260 | - |
| (Random) Writes/sec | 240 | 250 | 235 | - |

On trying to create over 500 column families, sometimes it was able to create upto 600 column families but most often it used to timeout or hang. The read and write performance was found not to depend on the number of column families.

**Reads/Writes**

The same table that was used for the previous test was used. The client code was modified to write 1GB of data into 1 million rows, each row having a single column whose value is randomly-generated 1000 bytes of data. Both random and sequential read operations and write operations were performed. The performance evaluation script that was available with HBase was used to do the required tests and the following results were observed.

| **Operation** | **Rate** |
|---|---|
| Sequential reads | 310 Reads/sec |
| Sequential writes | 1600 Writes/sec |
| Random Reads | 290 Reads/sec |
| Random writes | 1550 Writes/sec |

When compared with the results put up in the HBase site it is evident that the numbers have not improved much over new releases. Reads a significantly slower than writes as reads from memory has not been implemented yet which essentially means that reads pay the price of accessing the disk repeatedly.

**Pitfalls**

HBase is still under development. Currently, here are only 3 committers working on it. As a result the development is not rapid and there are some essential features that are still under development. MapFiles in HBase cannot be mapped to memory. When the HBase master dies, the entire cluster shuts down. This is because they an external lock management system like Chubby has not been implemented yet. HBase master is the single point to access all HRegionServers and thus translates to a single point of failure. Performance really depends heavily on the number of RPC calls made. So a general thumb rule would be to configure parameters such that it shall minimize the number of RPC calls.

## Hypertable

**Introduction**

Hypertable is an open source, high performance, scalable database, modeled after Google's Bigtable. It stores data in a table, sorted by a primary key. There is no typing for data in the cells, all data is stored as uninterpreted byte strings as in BigTable. Scaling is achieved by breaking tables in contiguous ranges and splitting them up to different physical machines. Data is stored as <key,value> pairs. All revisions of the data are stored in Hypertable, so timestamps are an important part of the keys. A typical key for a single cell is <row> <column-family> <column-qualifier> <timestamp>.

**Requirements**

Hypertable is designed to run on top of a "third party" distributed filesystem that provides a broker interface, such as Hadoop DFS or CloudStore (earlier known as KFS, developed in C++). However, the system can also be run on top of a normal local filesystem. All table data is stored in the underlying distributed filesystem.

**Architecture Overview (Implementation)**

Hypertable consists of the following components interacting with each other as described in Fig. 1.

1. **Hyperspace.** Hyperspace is the equivalent of Chubby lock service for Hypertable. It provides a file system for storing small amounts of metadata and acts a lock manager. In the current implementation of Hypertable, it is implemented as a single server.

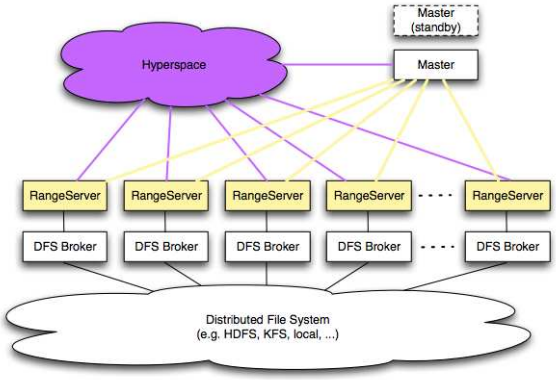


Figure 1: Processes in Hypertable and how they relate to each other.

2. **RangeServers.** When the size of the table increases beyond a certain threshold, it is split into multiple tables, each of which is stored at a Range Server. The ranges for the new data are assigned by the Master. This is analogous to ChunkServers in BigTable terminology.

3. **Master.** The master handles all meta operations such as creating and deleting tables. The master is also responsible for range server allotment for table splits. As per the current implementation, there is only a single master process.

4. **DFSBroker.** Hypertable achieves independence from a distributed filesystem by using a DFSBroker. The DFSBroker converts standardized filesystem protocol messages into

the system calls that are unique to the specific filesystem.

Hypertext Query Language (HQL) is used as the query language with Hypertable. HQL closely follows SQL type syntax including primitives like SELECT, INSERT, DELETE.

## Evaluation

### Experimental Setup for Hypertable

The Elastic Compute Cloud (EC2) infrastructure service from Amazon was used as a testbed for the performance evaluation. Amazon EC2 provides the following instance configurations. For brevity purposes, we only describe the instances used in the evaluation.

1. **Small Instance**: 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB of instance storage, 32-bit platform

2. **Large Instance**: 7.5 GB of memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of instance storage, 64-bit platform

3. **High-CPU Medium Instance**: 1.7 GB of memory, 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each), 350 GB of instance storage, 32-bit platform

EC2 Compute Unit (ECU) – One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

### RightScale

RightScale is a third-party web tool for managing the deployments over Amazon EC2. It provides an easy interface for adding/deleting servers to the deployments and managing remote access to those servers via a simple to use web-based ssh interface. However it becomes a major hurdle due to the lack of support available about its usage and basic tools. RightScale's wiki fails to mention some of the important aspects of managing a large deployment over EC2 including bundling a running instance and managing credentials for sub-accounts. RightScale provides pre-built/configured images for easy deployment of basic systems like Hadoop however due to lack to support about setting it up and providing proper credentials, setting up a Hadoop cluster from scratch turned out to be an easier task than using RightScale.

Being a third-party tool, RightScale does not seem to offer any specific advantage over the native Amazon interface or ElasticFox.

### Hypertable Benchmark Implementation

We set up a Hypertable cluster with N RangeServers to measure the performance for *random reads* and *random writes* into a test table. Rows are by default sorted by the primary key in Hypertable. A random write corresponds to creating rows in no specific order where the final location of each row is decided by the master node on the fly. The data used for evaluation of Hypertable was random data created on the fly by using a random() function and creating a fixed length random key of 12 bytes.

*Sequential reads* and *sequential write* performance are measured by reading/writing data from rows in a fixed order. Throughput for writes is measured in terms of records inserted per sec and cells scanned per sec for reads.

### Testbed Configuration

In the experimental setup, the master was running as a Small Instance while the RangeServers were running on High-CPU Medium Instance with each RangeServer running on a single node. The test were also performed with the master node running on a Large instance, however, as in case of Bigtable, the master was not found to be a performance bottleneck and hence similar results were obtained.

For the purpose of this evaluation, Hypertable was running over HDFS however since it supports a broker interface that can be used with any GFS-like

distributed file system, we also plan to evaluate the performance over CloudStore, earlier known as Kosmos File System (KFS), which is developed in C++. In the current setting, HDFS was configured with 3-way replication.

As in BigTable, clients control whether or not the tablets held by RangeServers are compressed or not. For basic evaluation of the system, compression was turned off in order to compare with the numbers provided for BigTable.

**Variable Factors**

The following factors are critical when measuring the performance of Hypertable for random reads and writes

.

1. Blocksize: This is the size of the value for a corresponding key to be written into the table.

2. RangeServers: This denotes the resources available for the system and acts as a measure of scalability of Hypertable.

**Fault Tolerance**

Hypertable is still under development and therefore there are some critical features that are missing from the current release. As per the documentation, currently Hyperspace and Master are implemented as a single server leading to a single point of failure.

**Performance**

As proceeded in the BigTable paper, we begin the performance evaluation of Hypertable with only 1 RangeServer. The fault tolerance of Hypertable was evaluated using a single RangeServer. It was found that Hypertable does not tolerate the failure of RangeServers gracefully. If a RangeServer crashes or becomes unavailable to the master, the system is not able to recover and the data at the range is lost as per the system.

The following table contains the results obtained with a single RangerServer compared to the results from the BigTable paper. The performance numbers provided in this section correspond to only the successful runs of random reads and writes. In the current evaluation, clients write approximately 1 GB data in the RangeServer.

| **Experiment** | **Hypertable** | **BigTable** |
|---|---|---|
| Random reads | 431 | 1208 |
| Random Writes | 1903 | 8850 |
| Sequential Reads | 621 | 4425 |
| Sequential Writes | 1563 | 8547 |

Figure 2. Number of 1000 byte values read/written per second in a cluster with only one RangeServer.

Comparing with BigTable, the initial numbers seem way behind. Each random read involves a transfer of 64KB block over the network out of which only 1000 byes are used, hence leading to a lower throughput for random reads as compared to random writes. The RangeServer executes approximately 431 reads per second which translates to approximately 27MB/s of data read from the HDFS as compared for 75 MB/s for BigTable and GFS.

Sequential reads and sequential writes were expected to be similar since the bottleneck for writes is writing to the commit log and not the RangeServers themselves. This is consistent across BigTable, HBase and Hypertable.

Fig. 3. shows the variation of throughput (records inserted per second) for different block sizes for inserting a fixed amount of data. For the purpose of these measurements, 1000 byte records were inserted randomly into the table amounting to a total of 1GB on a cluster with a master and one RangeServer.

Increase in aggregate throughput is observed as the system is scaled by adding multiple RangeServers but the increase does not seem as drastic as described for BigTable. As in case of BigTable, the increase in throughput is far from linear. For example, the performance of random writes increases by a factor of 1.6 approximately as the number of RangeServers increases by a factor of 3.2

The performance increase is not linear as current version of Hypertable does not perform any load

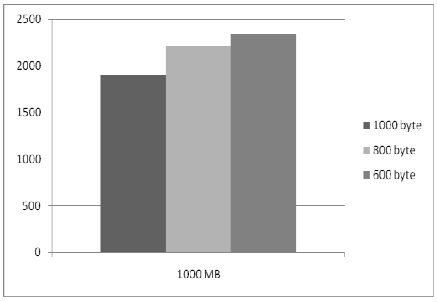


Fig. 3. Variation of throughput (records inserted/sec) with blocksize with random writes.

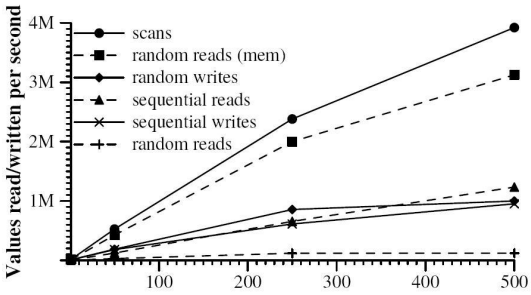


Fig. 4. Results from BigTable

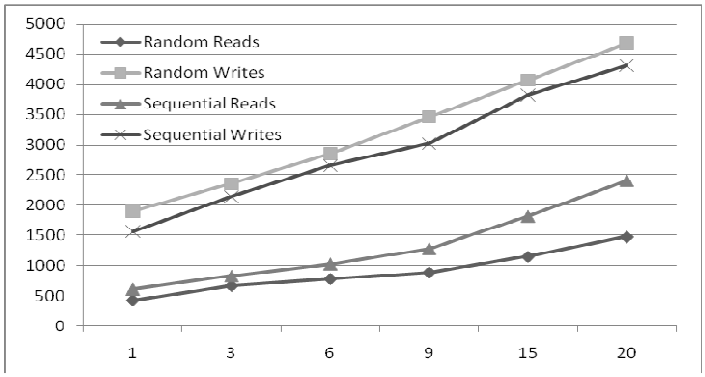


Figure 5. Total number of 1000-byte values read/written per second with increase in number of RangeServers.

balancing amongst the RangeServers. As for BigTable, the random reads benchmark shows thee worst scaling with an aggregate increase in throughput only by a factor of 3 for a 20 fold increase in the RangeServers.

## Experience

### System Reliability

The current release of Hypertable (0.9.12) seems to be relatively unstable with frequent failures of master node leading to a complete loss of data stored in the system. The failures were particularly observed when writing large amounts of data into the system. The frequency of the system reaching an unresponsive state was comparatively higher when the writes were of greater than a few GB. Hypertable appears to be relatively stable to random reads and failures were not frequent when reading large chunks of data. HBase, on the other hand, seemed to be much more reliable than Hypertable when run on a single node in terms of dealing with large chunks of data.

While writing large chunks of data, some of the failures were reported as “Hadoop I/O error” signaling either the limitations of HDFS under stress or incompatibilities between Hypertable and HDFS.

### Hypertable Query Language (HQL)

The query language for describing the loose schema of the tables used in Hypertable is Hypertable Query Language. HQL closely resembles SQL and is easy to use.

## Other Minor Contributions

**Log4cpp:** It is a library used to provide logging support for systems developed in C++ corresponding to Log4j for Java. The last release was in 2002 and is incompatible with g++ 4.3.x and hence minor fixes were required.

## Future Work

In order to do a complete evaluation of Hypertable, a performance analysis over CloudStore is planned. A combination for CloudStore and Hypertable when compared against HBase and Hadoop, would make up a new chapter in the age old C++ vs. Java battle for large scale distributed storage systems.

Another important aspect is to scale up comparatively to the extent described by Google. Amazon EC2 does provide the resources to scale up to a much higher extent than described in the report, however failures of master node in Hypertable limits repeating the experiment in the same setup. We have coordinated with the Hypertable development group and we plan to scale the system up further once the bug is resolved.

Scaling up HBase is another aspect that was planned for the project. We plan to scale HBase up to similar set up and study the performances under a consistent setup.