# HBase on top of HDFS

Kevin Böckler

**Abstract.** Cloud-Computing relies more and more on large-scale database systems which are able to store huge amounts of data in a fast and reliable fashion. While Google has developed its *BigTable* solution, with the Hadoop filesystem Apache reproduces this concept and builds a distributed file and database system out of single, largely spread components. This paper reviews Apache's implementation and focuses on main concepts of the underlying filesystem and the communication between single entities. At the end a small proof of concept is shown by setting up a single-machine database cluster.

## 1 Overview

This paper is about describing mechanisms of the Hadoop file and database system which comprises of two parts: Chapter 2 shows the basic idea of distributed file systems. This is a concept Apache uses for implementing their underlying filesystem shown in Chapter 3. Upon this layer, Hadoop comes up with the second part: HBase - a database implementation - presented in Chapter 4. The resulting features and use cases can be seen in Chapter 5. A small demo setup targets the easy-to-use Java-API[1] and will deliver small facts in a brief overview in Chapter 6.

## 2 Distributed File Systems

Storage in CC[2] always comes with huge amounts of data which are uploaded and downloaded from distinct regions by many users. From the view of a client, each file should be located as near as possible to the reader to accomplish lower latencies and higher data throughput. For these requirements not a lone server can serve all the network requests, instead several servers should be spread among the field to deal with Big Data. The most important factors of storing in a DFS[3] will be given in the next sections. Right after the implementation of the Hadoop's version of a DFS is presented.

### 2.1 Scalability

When many clients are using a Cloud Service, the resulting high server loads cannot be handled by a single centralized server. Hundreds or thousands of users access a resource at the same time while adding data with ideally no time-costs when doing database tasks. The overall performance of a DFS should degrade only very moderately when writing or reading huge amounts of data.

### 2.2 Transparency

Clients of a DFS don't want to know anything about storage locations of the server holding the actual data. Instead addresses with logical file names should be used which are resolved to a connection to the host whenever a request is made. Hence the client does not directly know, where the file comes from, whether the data is split up among multiple servers or the data will be read from the original file or a replicated one. Often addresses look like UNIX pathnames such as `/a/b/c`. These addresses are resolved by the so called *pathname traversal* which follows the root node until the last child node has been found.

One differentiates between the terms *Location Transparency* and *Location Independeny*, the former describes the presented concept of hiding a real location of data while the latter means that the file name stays the same if the linked resource is moved - due to server crashes or file replication. More information about transparency are provided by Levy and Silberschatz [1].

### 2.3 Fault Tolerance

Writing data to a DFS leads to two major requirements: first a file has to be *available* so that whenever a request is made the client can read or probably write to this file. Next to availability there is *Robustness* which implies a file survives a hardware failure - in that case the file can be recovered by other machines, e.g. by using replications. File replications play a big role in DFS as so called replica increase the robustness immensely by leading to higher synchronization costs: each file write now has to be reflected to all replica and probably cached versions of all clients. Caches will be covered in section 2.5.

Also mention that using *stateless* connections will have less data loss, because the server gets information on every incoming request. Whereas *stateful* connections allow more efficient communication between the server and the client.

### 2.4 Semantics

Semantics describe the access-pattern of resources in the given DFS. It covers synchronization when it comes to

1. Application programming interface
2. Cloud Computing
3. Distributed File System

write access. The following ones are commonly used and listed by Levy and Silberschatz [1]:

**UNIX Semantics.** Every single write to a file is synchronized and reflected to all replica.

**Session Semantics.** On a opened session (Open File) the most recent data is fetched and an update and reflection is only made when the session is closed (Close and Flush File).

**Immutable Semantics.** Once a file has been written it can not be modified (immutable) - it is declared as being shared and therefore is read-only.

**Transaction-like Semantics.** The same as Session Semantics but also locks the file while a session is opened due to concurrency issues.

### 2.5 Remote-Access

Accessing data results in either doing a Remote Service Request or having the data in the cache. When resources do not change too frequently in the cloud, a client is supposed to cache data because reads will be more efficient. Caching data trades off lower access times against issues with reliability and autonomy since caches are not validated by the DFS. Hence the client deals with validation including the frequency doing those checks.

In real applications, clients often cache their own writes until a certain amount of flushable data has been reached. The collected data is then sent to the server to reduce a high number of low-payload-calls. Often this is a better approach then immediately send every chunk to the server since servers also tend to use big file block sizes (of several MB[4]).

## 3 HDFS[5] - AN IMPLEMENTATION OF A DFS

By implementing a DFS not all of the criteria can be met. There always occur tradeoffs when improving one property to the costs of another one (e.g. speed against throughput). This section explains the architecture and workflow of a the HDFS explained by Borthakur [2] and will afterwards give a summary of which requirements are satisfied.

### 3.1 Architecture of HDFS

The domain of HDFS is split into so called clusters which can spread all over the network and allow a **scalable** infrastructure. Figure 1 shows such as cluster. Every cluster is comprised of one *NameNode* and multiple *DataNodes*. The former has the task to hold meta data of the underlying stored files, i.e. file names and their distribution of blocks among the DataNodes. The latter execute the task to write the actual data of files to the disk. Therefore a file is typically split into file blocks of 64 MB. Each of these can end up stored in a different DataNode. When a client writes a file to a HDFS cluster, it asks the NameNode for where to put the blocks of this file which will answer with addresses to the concerning DataNodes. Hence real throughput of data never leads through the NameNode which will lead to a scalable distribution of file traffic. This distributed file write operation is highlighted blue in Figure 1. The client can call operations on the NameNode like opening, closing and renaming a file. The actual write and read process is done within a connection to a DataNode.

NameNodes hold meta data in two structures: one is the locally persisted file `FsImage` which holds all the information such as the host namespace or modification timestamps. Also block addresses of Datanodes or indices to files are stored in this structure. The other one is called *EditLog*. It represents the same data as the `FsImage` but hold those information in the main memory. Each client operation, which changes meta data, ends up as a record in the EditLog. The NameNode will persist all records of the EditLog to the `FsImage` at a so called *Checkpoint*. These processes occur on node startup - after the Safemode shown in Section 2.3 has finished - or at given intervals.

### 3.2 File semantics and replication

Files in HDFS are designed to have **Immutable** semantics which were described in Section 2.4. Therefore once a file is written the HDFS never again has to deal with synchronizing updates to the stored files. Once a file is written and closed it is handled as being shared. This access pattern is called *write-once-read-many* (WORM).

A file is started to being replicated while creating/writing it. One result is a faster access time so clients from all over the place can read this file, because a replica of the file will be located to a DataNode next to the client. The other important reason for replication is fault tolerance: if a DataNode crashed or gets unavailable in the network a file should better be available in multiple

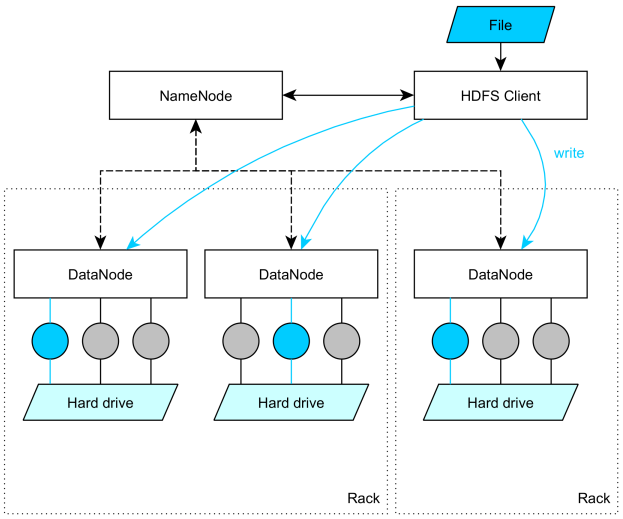


Figure 1: HDFS Architecture: One Cluster

4. MegaByte
5. Hadoop Distributed File System

places. NameNodes are responsible for managing replications - see more in Section 3.3.

The replication process while writing is shown in Figure 2: a file is stored in block 3,5,7 when the figure shows the storage of block 3. Normally a client writes chunks of data (4 KB[6]) to the first DataNode. This DataNode will persist the data and simultaneously push this chunk to the next DataNode. That process is called *Pipelining*. By default HDFS clusters use a *replication-factor* of 3. The strategy is to store the first replica in the local rack and the second one in an external rack. A rack is comprised of DataNodes within one region and hence the nodes have a similar network context (same latencies) or may be unavailable all at once (if a shared DNS[7] node fail).

To allow greater scalability the storage capacities of DataNodes can be rebalanced. If a DataNode reaches a storage percentage beneath a certain threshold, the NameNode can ensure other DataNodes to move their data to the less used one so all DataNodes end up in having a more equal file amount compared to each other.

### 3.3 Fault Tolerance

In Section 2.3 the term Robustness was named. It means making a file recoverable even if the hardware fails. In both cases, only a network issue or a hardware failure of a DataNode server, files are not accessible from a client. File replication solved this problem by making sure, enough redundancy exists among the scalable network. NameNodes therefore have some managing tasks: they watch over their underlying DataNodes and send orders to replicate or re-replicate files (in case one replica has been lost). Hence DataNodes have to make sure, that they are still alive. This is done by a so called *Heartbeat* message periodically sent by the DataNode to the NameNode. If a DataNode gets unavailable, the NameNode won't receive any more Heartbeat messages and as a result will issue other DataNodes to recover and re-replicate its lost data.

6. KiloByte
7. Domain Name Space

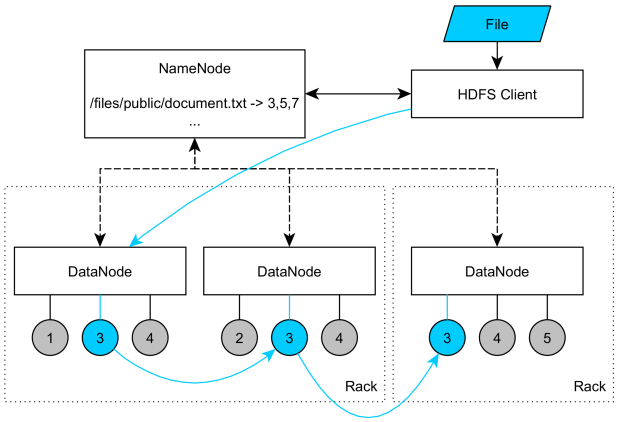


Figure 2: Replication of a file block

When a NameNode starts up, at that process it will begin working in *Safemode*-state. In that state, a NameNode requests Blockreports from its DataNodes to get an overview of all the hosted file blocks and their replication-factor. The NameNode remains in the Safemode-state until a certain part of all file blocks are checked to have a minimum amount of replicas alive. If replicated blocks are missing, the NameNode will issue new replication procedures.

### 3.4 HDFS Summary

The HDFS architecture of clusters and their managing NameNodes actually allow a great scalability among widely spread clients. They can always connect to cluster which is located near a client and data is supposed to be replicated all along the global network.

For HDFS-Clients the file system is transparent to some point, meaning file names are represented UNIX-like without giving information about its physical storage location. Although if a client really wants to read or write data, it has to connect to one or more distinct DataNodes to do I/O[8]-Operations.

Replication of data leads to tremendous fault tolerance and robustness of stored files. With the help of NameNodes, failovers are done very transparently and fast. Availability and accessibility of data is provided by the central address management of a NameNode.

In contradiction to these advantages of HDFS making this file system useful for application in CC-services, there is the single-point-of-failure when it comes to NameNodes. Reliability of these nodes is not truly given because of no overlying observer or network which maintains the NameNode of the working cluster.

## 4 HBase

DATABASES are always used when storing, changing and reading structured data comes into place. In CC data storage has to be very fast accessible and reliable. In the past relational SQL databases were used for companies to store their data. Nowadays however the importance for NoSQL databases arises. This chapter will give a brief overview about the differences of NoSQL databases compared to relational ones. HBase as a NoSQL database will be introduced right after. Goals, applications and the architectural design of the Hadoop database solution on top of HDFS will be presented.

### 4.1 NoSQL database systems

The term NoSQL does not stand for one distinct name but means something like *Not Only SQL* or *Not Relational* - compare Cattell [3]. He also states out some key features which differentiates NoSQL and SQL databases.

First NoSQL systems can scale horizontally on many (distributed) servers. Therefore data can be replicated

8. Input / Output

throughout this scaled server landscape. Also requests and calls are more simple in a NoSQL system allowing easier and more efficient access to database objects. This is because SQL requires statements which NoSQL does not. Efficiency in storing and indexing data is higher in NoSQL databases since indexes and main memory can be distributively used. While SQL databases have got a static database scheme where changing it is difficult at some point of time, NoSQL systems provide a dynamically changeable format so new attributes can easily be added throughout use time. Another important feature of NoSQL systems is the weaker concurrency model which is required to be used. This will be explained in the next section.

#### 4.1.1 Concurrency Model of database systems

There are two models which describe concurrency access on shared data in general. *ACID*[9] provides transactional-like properties which make sure, that data is always available and up-to-date. Every change is immediately reflected to all copies (replications and caches). In a distributed system this concurrency model is very hard to achieve.

In contrast to ACID there exists BASE[10] which is a weaker model and therefore suitable for NoSQL systems. It demands only an *eventually consistent* reflection of data, so changes are not immediately written to all files so it might take some time until each client reads the most recent version.

The term *Concurrency Control* is related to the way a database system deals with concurrent accesses on the same resource. They are ACID, *MVCC*[11] - clients can create multiple writes on the same file at once - or *Locks* - a file is exclusively opened to write by one client. HBase uses the last concurrency control model.

### 4.2 Reasons for HBase

HBase implements an *Extensible Record Store* which is a NoSQL-database system. Besides Extensible Record Stores and Relational Stores (SQL systems) there are some more types of stores which are presented by Cattell [3]. This kind of store represents a database with each row having a primary key. Usually when reading rows this is done by requesting a so called *range* rather than selecting a distinct row by an index function (such as hashing).

**Efficiency.** Because of this Store implementation, HBase features efficient range queries and sorting techniques using B-Tree indexing. Updates to HBase are done efficiently by storing them as durable change records to a log file - see more in Section 4.3. To receive efficient read access HBase uses *MapReduce* to process data. Konishetty et al [4] give some more details on Hadoop and MapReduce. Moreover a client can read multiple rows per single fetch-request which results in faster per-row access times. Khetrapal et al [5] did a case study about reads and writes times - also by observing differences between sequential and random access of rows.

**Storage capacities.** HBase is developed to store BLOB[12]s in each single column. Database tables also tend to have millions of rows. Thus using HBase allows to manage huge amounts of data while it is only suggested to use HBase when having these demands of huge data storage.

**Scaling.** While HBase is a NoSQL system it allows great horizontal scaling and an extensible data model. Accordingly fields of a database row are packed together in so called *Column Families* which are stored distributively and independently. Hence it is easy to add new column families at run time.

**Atomic operations.** As stated in the section before HBase uses the *Locks* concurrency control so each row operation is locked for writing by one client exclusively. This allows atomic access to a single row and thus provides strong consistency.

**Transparency.** Another huge advantage is that the whole process of partitioning and distributing data is done transparently so the user of HBase will not know anything about it.

**Fault-Tolerance.** Underlying HDFS files, which are stored by HBase, are treated fault-tolerant (compare Section 3.3). In general that means a crash of a DataNode is detectable and files are transparently replicated and restored to achieve robustness. HBase additionally allows configurations for writing data very fast to the hard drive rather than keeping them in fault-prone memory - depending on the client's needs.

**API Support.** HBase provides a Java-API, Thrift-API, REST-API and drivers for JDBC[13]/ODBC[14].

### 4.3 Architecture of HBase

The Hadoop database system consists of two parts: one is the actual HBase layer and the other one represents the Hadoop filesystem (HDFS). The latter was described in the chapter before. HBase wraps the underlying filesystem and adds new operations and tasks which will appear to the HBase client as database table operations. For this purpose HBase consists of three components: the *HBaseMaster*, *HRegionServer* and *HRegion*. Their tasks and synergy will be described in this chapter.

#### 4.3.1 HBaseMaster

**Connection to HBase.** A client who wants to use the HBase database system - in future terms called HClient - first receives a connection to a HRegionServer (explained in Section 4.3.3) from the HBaseMaster. This server knows about all connected HRegionServers and

9. Atomicity, Consistency, Isolation and Durability
10. Basically Available, Soft state, Eventually consistent
11. Multi version concurrency control
12. Binary large object
13. Java Database Connectivity
14. Open Database Connectivity

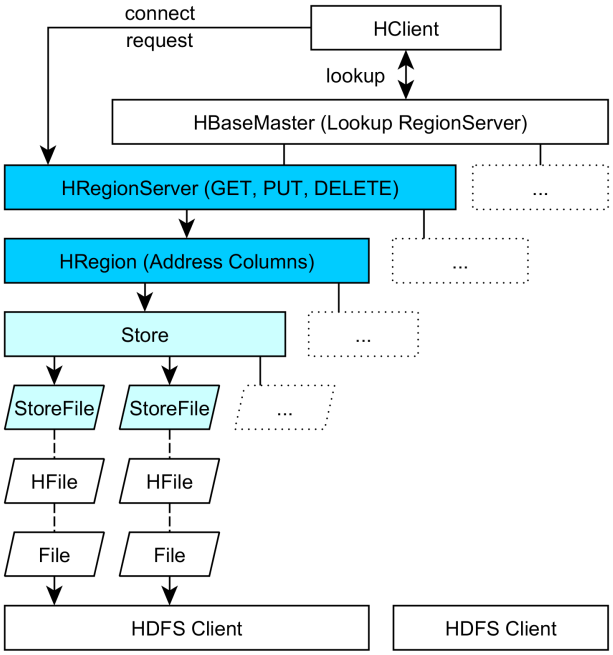


Figure 3: Architecture of HBase based on George [6]

can be queried for available connections for the HClient. Additionally, to find a HBaseMaster server, the HClient first looks up the address of the server in an overlay coordination network called *ZooKeeper*. Hunt et al [7] give an explanation of this service which aims to connect distributed systems.

**Heartbeat.** The HBaseMaster needs to overwatch all connected HRegionServers and uses Heartbeat messages (comparable to Heartbeats of HDFS) to check if some servers went offline. Therefore if a HResionServer crashes, the HBaseMaster has to reassign the HRegions which were connected to the former server.

**Load Balance.** During runtime the HBaseMaster can equalize the server load of HRegionServers. If some servers contain a lot higher amount of HRegions and thus they manage more data, those HRegions can be reassigned to other HRegionServers.

**Schema changes.** The HBaseMaster also holds meta data such as changes to the database schema. If tables or specific columns are modified, the HBaseMaster deals with altering the underlying database schema due to the HRegionServers which hold portions of a database table (see Section 4.3.3).

### 4.3.2 HRegionServer

**Read and write operations.** Once a HClient is connected to a HRegionServer it can execute R/W[15]-tasks such as `GET`, `PUT` or `DELETE` data. HRegionServers do not store the payload data on their own filesystem: those R/W-operations are forwarded into concrete HRegions. This means a HRegionServer can be associated to one or more

15. Read/Write

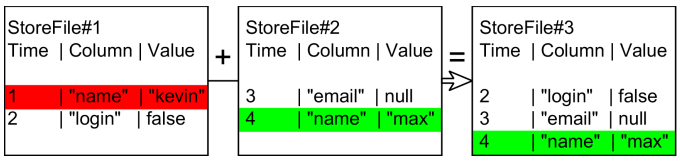


Figure 4: Example of a StoreFile Compaction

underlying HRegions which will deal with the actual file handling (see Section 4.3.3).

**Handling HRegions.** A HRegionServer manages underlying HRegions. Because there can be multiple regions connected to one HRegionServer, unequal load balances can occur (described in Section 4.3.1). Therefore the HBaseMaster issues split operations on HRegions. This results in the HRegionServer receives this issue and splits up a single HRegion into two, distributing their associated data files. Besides those splits it is also possible to merge HRegions together which is called *compaction of regions*.

### 4.3.3 HRegion

Each HRegion holds a file called *HLog*. This log file is stored in a HDFS-file and represents the WAL[16] of HBase. R/W-operations are directly appended to this log file and will be flushed eventually to the underlying filesystem. This allows a more efficient storage of data, because only larger portions of R/W-operations are written to a file.

Next to the WAL each HRegion comprises zero or more so called *Store*s. The file operations of the WAL are finally flushed to these Stores.

### 4.3.4 Store

Note, that *Store == Column Family*. So for each group of columns (or attributes) in a database table there will be one Store representing only this subset of columns. In general, a CF[17] is comprised of attributes which logically belong together (e.g. a persons surname and firstname).

**Storage.** When the overlying HRegion flushes data to its Stores, each Store receives a portion of row-based R/W-operations belonging to that CF. The changes to the according rows are applied and saved in the *MemStore*. This is the representation of the Store and its data in the main memory of the Store for more efficient reads and writes. Eventually the MemStore will be written to the embedded *StoreFile*s. Each Store holds several StoreFiles which are explained in a while.

**Compaction of StoreFiles.** Because the underlying filesystem is HDFS and its files are immutable, changes to the Store are always written by records defining what concrete part of a CF has been changed. Each record receives a timestamp so the most recent change to a database field is returned by a read-operation. Those records and hence the representation of this Store is

16. Write-ahead-Log
17. Column Family

written in many StoreFiles (because records can only be created in new StoreFiles). To achieve faster lookups - which involves less records to search for the most recent timestamp and such - StoreFiles are periodically compacted. As illustrated in Figure 4, a compaction process can merge several StoreFiles to receive only a few bigger files out of many small ones (*minor compaction*) or one single big StoreFile (*major compaction*). The obsolete StoreFiles will then be deleted by removing the HFiles in the underlyingHDFS.

#### 4.3.5 StoreFile and HFile

Each StoreFile exactly wraps one *HFile*. When data is send from the Store to the StoreFile, then the data is written to the HFile, that is directly mapped to a HDFS-file. This is where data leave the HBase layer and moves into the Hadoop filesystem.

HFiles are replicated in HDFS. When one file is changed (throughout a put request), this change has to be reflected among all replica. While this takes some time, foreign read requests might either fetch the latest updated version or the prior one. If a HClient wants to make sure that a request always gets the latest change, HBase can be configured to so called *STRONG consistency*. Otherwise, requests will deliver best effort with *TIMELINE consistency*. More details about this can be found in the Apache reference guide [8].

## 5 Application

The last chapters presented HBase on top of HDFS. The following sections will give a clue about how this distributed database system can be used or how it is adapted to fit certain real-world applications. First the motivation to use HBase is given by looking at Facebook. After that, possible improvements of the HBase implementation is presented in general. The last section will then cover the HClient - API illustrating the major methods to use.

### 5.1 Motivation using HBase

HBase features some important properties when it comes to storing data in the cloud. The following advantages are presented by Borthakur et al [9]. They are needed for Facebook in order to allow an applicable implementation of their Messaging-service.

- High random and sequential write throughput
- Efficient random and sequential reads
- Elasticity (add storage without overhead)
- Efficient and low-latency Consistency
- High availability and recovery
- Fault isolation (failure only affect a few people)
- Atomic operations (read, write, modify)

Also Facebook had to make sure that once HBase is in production, from this single moment on over 500 million people will be using the database system which then has to be reliable and totally fault-tolerant.

Besides the Facebook Messaging-service HBase is also used in Facebook's Insights-service, which determines and stores statistics about page visits, clicks and so on, and Metrics System, which captures hardware loads of Facebook's servers.

### 5.2 Improving HBase

As stated before, Facebook had to adjust some implementation designs of Apache to realize the usage in such a big context as its Messaging-service is.

Facebook decided to store data in smaller HBase clusters to increase the fault isolation. Because in the Hadoop system there are single-point-of-failures, HDFS-NameNodes are wrapped by so called *AvatarNodes* wich actually means a NameNode is replicated one time, so if a NameNode fails, the copied NameNode can instantly take the former place and the crashed one can restart.

In the HBase layer, every cluster has a single-point-of-failure as there is only one HBaseMaster. To deal with it, Facebook connects the master to the overlying ZooKeeper network and will upload the HBaseMaster state to a ZooKeeper-node so that on crash the state of the cluster can be recovered. There are many more slight changes which are given by Borthakur et al [9] which often slightly increases some access times.

One thing to mention is that all HDFS-connections to DataNodes are handled by Java's RPC[18] interface. This might be a little slow. For this case, Islam et al [10] showed an implementation of using RDMA[19] over JNI[20] and a network called *InfiniBand*. This results in an average of 20% lower latencies and higher throughput.

### 5.3 HClient - API

In General a client application just wants to use the whole Hadoop filesystem in a transparent fashion. Thus, once a cluster has been set up, a client will issue some basic database-table operations like `get`, `put`, `delete`. HBase delivers a very simple API to execute such requests from the client's point of view. In Java, the main classes to use can be instantiated very easily:

**`Configuration`.** Sets up some basic information like the IP[21]-address of the ZooKeeper-service.

**`HTable`.** In combination with the `Configuration` this class connects to a specified database-table on which one of the following commands can be issued.

**`GET`.** This request can fetch data specified by a row-identifier in a random-access pattern.

**`SCAN`.** Similar to `GET`, but receives rows by range-queries with a sequential-read.

**`PUT`.** Writes single values to a column within a column-family. Therefore the family has to exist, columns can be added in runtime.

18. Remote Procedure Call
19. Remote Direct Memory Access
20. Java Native Interface
21. Internet Protocol

| CF1 : logindata | CF2 : contactdata |
|---|---|
| username | firstname |
| password | lastname |
| | email |
| | town |

Table 1: User relation

**DELETE.** Removes values the same way like putting ones.

The use of these few classes fulfills the major tasks when just doing some I/O-operations with HBase. These classes will also be used in a small Java-client presented in Chapter 6. For a full documentation of the Java-API, see Apache's Snapshot of their version 2.0.0 [11].

## 6 Demo

This chapter shows a small demo setup where a proof of concept of HBase with HDFS shall be given. Only some main features are considered and the following sections describe the installed services and executed tasks. At the end, a summarizing result will be given.

### 6.1 Setup

In this test the Hadoop system is set up in the so called *pseudo-distributed* mode which means all needed services are loaded on their own and act as being distributed but indeed are all on the same local host machine. Hence the following proof of concept runs on one test server[22].

While Hadoop delivers an API for client accesses (i.e. the HClient), for the purpose of this demo some more operations are needed such as putting many rows of data at once. To do so, the HClient is wrapped in a self-implemented Java-Project[23] which deals with fetching and reading multiple sets of data via the HClient interface and also takes some times in milliseconds.

For the needs of the test the demo uses a `User` relation consisting of two column families shown in Table 1.

### 6.2 Executed Tasks

This small demo will determine some scalability of the database system with respect to read operations. Furthermore in this small context the influence of major compactions should be tested concerning reads of data. The following tasks are executed and the needed time of reading data has been measured:

1) Fetch 1000 users out of 1000 users
2) Fetch 1000 users out of 10000 users
3) Fetch 1000 users out of 100000 users
4) Fetch 1000 users out of 1000000 users
5) Fetch 1000* users out of 1000000 users - * users experienced additionally five Updates in the WAL
6) Fetch 1000* users out of 1000000 users - * users' Updates have been compacted by major compaction

22. System Specs: VirtualBox-VM with: CPU: 4 Cores up to 3,4 Ghz, RAM: 2 GB, OS: Xubuntu64

23. https://github.com/fraxxor/hbase_java_client

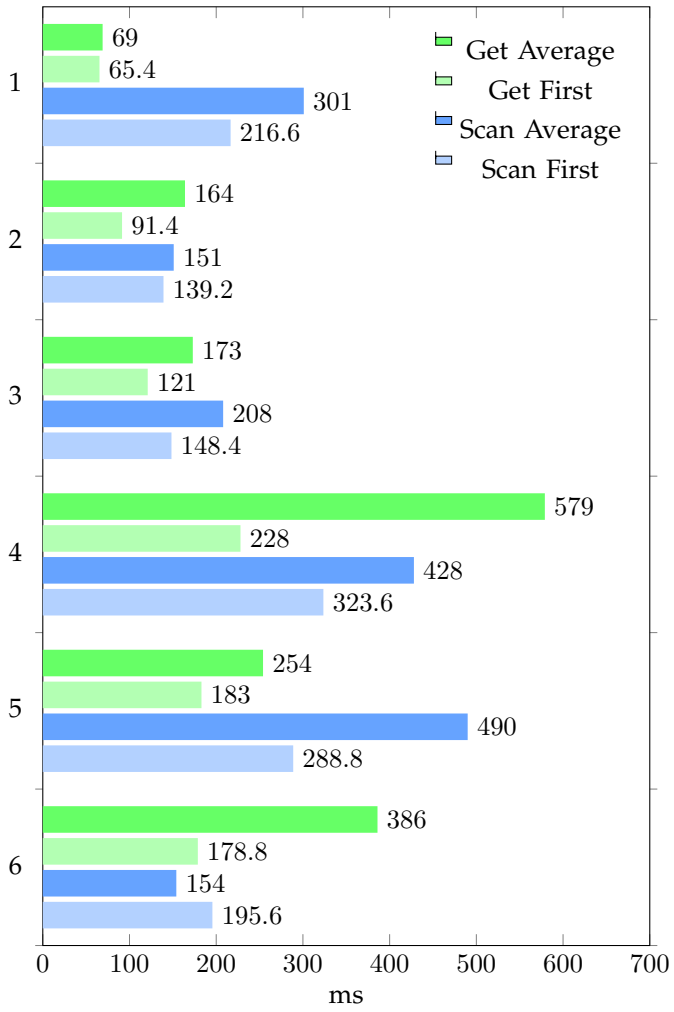


Figure 5: Results of the Performance Test

Each fetch process is separated into reading by random access (GET) and by sequential read (SCAN). The needed time is taken from the first fetch process after putting the data into the database and also an average value is calculated through five fetch requests.

### 6.3 Results & Summary

Figure 5 shows the results of the small performance test by comparing the measured times in milliseconds, taken by the different tasks.

Even though the tests are not conclusive with respect to real distributed systems, some trends can be seen in the diagram. From Task 1 to 4 one can check that reading requests seem to scale logarithmically compared to put data. When increasing the database rows by factors like 1000 only small amounts of time have been added to the read-responses. Due to some noise it can now be answered whether random or sequential reads are faster - they seem to perform quite equal compared to each other. The question, if major compactions after doing

some PUT-operations (Task 5 & 6) result in better reading performance cannot be confirmed - the measured times are lying closely together.

To put in all in a nutshell the proof of concept has been made. The database system is scalable when considering reads. Still HBase is optimized for real distributed production use and the mostly intended tasks will be writing large amount of data very frequently. So measuring the reads of a pseudo-distributed server in this small demo is only one minor approach of testing.

## Acknowledgements

Special Thanks go to the developers of this LaTeX-Template, published on http://www.ieee.org.

## References

[1] E. Levy and A. Silberschatz, "Distributed File Systems: Concepts and Examples," vol. 22, no. 4, pp. 321—-374, 1990. [Online]. Available: http://doi.acm.org/10.1145/98163.98169

[2] D. Borthakur, "HDFS architecture guide," pp. 1–14, 2008. [Online]. Available: http://pristinespringsangus.com/hadoop/docs/hdfs_design.pdf

[3] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1978919

[4] V. K. Konishetty, A. K. K, K. Voruganti, and G. V. P. Rao, "Implementation and Evaluation of Scalable Data Structure over HBase," in *ICACCI '12 Proceedings of the International Conference on Advances in Computing, Communications and Informatics*. New York, NY, USA: ACM, 2012, pp. 1010–1018.

[5] A. Khetrapal and V. Ganesh, "HBase and Hypertable for Large Scale Distributed Storage Systems. A Performance evaluation for Open Source BigTable Implementations," 2008. [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:HBase+and+Hypertable+for+large+scale+distributed+storage+systems+A+Performance+evaluation+for+Open+Source+BigTable+Implementations#1

[6] L. George, "HBase Architecture 101 - Storage ," 2009, accessed: 2014-12-07. [Online]. Available: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

[7] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems." *USENIX Annual Technical ...*, vol. 8, p. 9, 2010. [Online]. Available: https://www.usenix.org/event/usenix10/tech/full_papers/Hunt.pdf

[8] Apache, "The Apache HBase™ Reference Guide," 2014, accessed: 2014-12-07. [Online]. Available: http://hbase.apache.org/book/book.html

[9] D. Borthakur, S. Rash, R. Schmidt, A. Aiyer, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, and A. Menon, "Apache hadoop goes realtime at Facebook," *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*, p. 1071, 2011. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1989323.1989438

[10] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High performance RDMA-based design of HDFS over InfiniBand," *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, Nov. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6468497

[11] The Apache Software Foundation, "HBase 2.0.0-SNAPSHOT API," 2014, accessed: 2014-12-12. [Online]. Available: https://hbase.apache.org/apidocs/index.html

**Kevin Böckler** Scientist and Enthusiast for web development. For some years he creates projects with PHP, Java, RoR, JS, and HTML/CSS. Additionally he focuses on Clean Code concepts.

## Abbreviations

| | |
|---|---|
| **ACID** | Atomicity, Consistency, Isolation and Durability |
| **API** | Application programming interface |
| **BASE** | Basically Available, Soft state, Eventually consistent |
| **BLOB** | Binary large object |
| **CC** | Cloud Computing |
| **CF** | Column Family |
| **DFS** | Distributed File System |
| **DNS** | Domain Name Space |
| **HDFS** | Hadoop Distributed File System |
| **I/O** | Input / Output |
| **IP** | Internet Protocol |
| **JDBC** | Java Database Connectivity |
| **JNI** | Java Native Interface |
| **KB** | KiloByte |
| **MB** | MegaByte |
| **MVCC** | Multi version concurrency control |
| **ODBC** | Open Database Connectivity |
| **RDMA** | Remote Direct Memory Access |
| **RPC** | Remote Procedure Call |
| **R/W** | Read/Write |
| **WAL** | Write-ahead-Log |