Rethinking HBase: Design and Implementation of an Elastic Key-Value Store over Log-Structured Local Volumes

Giorgos Saloustros
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
Heraklion, GR-70013 Greece
gesalous@ics.forth.gr

Kostas Magoutis
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
Heraklion, GR-70013 Greece
magoutis@ics.forth.gr

***Abstract*—HBase is a prominent NoSQL system used widely in the domain of big data storage and analysis. It is structured as two layers: a lower-level distributed file system (HDFS) supporting the higher-level layer responsible for data distribution, indexing, and elasticity. Layered systems have in many occasions proven to suffer from overheads due to the isolation between layers; HBase is increasingly seen as an instance of this. To overcome this problem we designed, implemented, and evaluated HBase-BDB, an alternative to HBase that replaces the HDFS store with a thinner layer of a log-structured B+ tree key value store (Berkeley DB) operating over local volumes. We show that HBase-BDB overcomes HBase's performance bottlenecks (while retaining compatibility with HBase applications) without losing on elasticity features. We evaluate the performance of HBase and HBase-BDB using the Yahoo! Cloud Serving Benchmark (YCSB) and online transaction processing (OLTP) workloads on a commercial public Cloud provider. We find that HBase-BDB outperforms a tuned HBase configuration by up to 85% under a write-intensive workload due to HBase-BDB's reduced background-write activity. HBase-BDB's novel elasticity mechanisms operating over local volumes are shown to be as performant as HBase's equivalent features when stress-tested under TPC-C workloads.**

***Keywords*-NoSQL, key-value stores, HBase, elasticity**

I. Introduction

In the Big Data era, continuously produced information in the fields of social networks, biology, physics, and the Internet of Things results into massive volumes of data that must be stored and processed, primarily for knowledge mining and decision making. Storing and processing such data sets effectively requires novel distributed systems, many of which have been introduced and in everyday use in recent years. In designing such systems, layered architectural approaches [1] are often chosen to reduce complexity, reuse existing code, and achieve lower time-to-market. However, layering is often responsible for performance penalties due to the lack of integration inherent in such designs [2].

HBase [3], a NoSQL data store for semi-structured data deriving in many ways from Google's BigTable [4], has achieved "reference point" status in the space of Big Data technologies. The HBase design follows a layered architecture, in the spirit of previous systems [5] [6], stacked in two layers. In the bottom layer, HBase uses HDFS [7] as a storage back-end. HDFS exposes to its client applications a shared namespace and implements scalability and fault tolerance mechanisms at the file layer. Having solved those issues in its storage back-end, HBase focuses in the logic and elasticity features of the database.

While HBase has been deployed and used widely in recent years, its power-users have been hinting at performance issues under specific workloads, as highlighted in a recent study of a large-scale deployment at Facebook [8]. In this study, Facebook analyzed HBase performance serving its popular Facebook Messages (FM) [9] application. Their traces from a large-scale cluster shed light to interesting results: FM, just like other popular Internet applications such as e-mail, SMS, and Chat, diverge from HBase's initial design goals of high sustained bandwidth (rather than low latency [10]). While HBase is by design a good fit for streaming workloads with a lot of sequential I/O such as MapReduce, it is not optimal for a wide range of workloads that are dominated by random reads (often including a small share of short (order of KBs) range-queries). In order to achieve high sustained bandwidth, HBase (which we will also refer to as HBase-HDFS[1]) performs aggressive storage reorganization. Performing key value reorganizations on top of an append-only distributed file system results in high network load and significant write amplification, impacting read performance. HBase additionally performs write-ahead logging for durability which further amplifies writes.

In this paper, we propose an alternative architecture to HBase, named ***HBase-BDB***, to overcome the aforementioned problems. We show that the replacement of HDFS with a thinner layer of a local key-value store implemented over local volumes benefits performance without requiring a major re-engineering effort. Since there are several local key-value store engines with different properties available, we decided to leverage one of them (BerkeleyDB (BDB) [11] Java Edition[2]) that fits well our design goals. BDB-JE is a robust, efficient, widely deployed integrated database engine.

[1]We use the term HBase-HDFS to refer to a standard, off-the-shelf software distribution of HBase.

[2]http://www.oracle.com/technetwork/products/berkeleydb/overview/index-093405.html

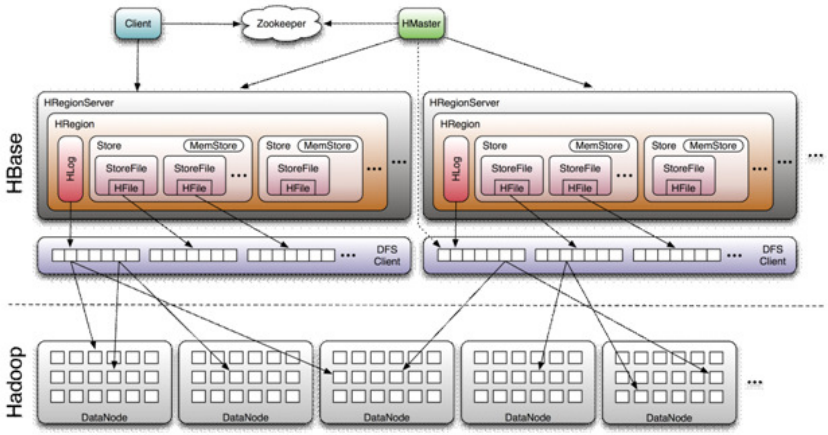


Figure 1: HBase architecture [3]. See Figure 2 for a high-level comparison with the architecture of HBase-BDB

It implements a B+ tree index, known to perform well for random read workloads and provide good support for range queries. The entire database is implemented as a log [12] avoiding the need for a separate write-ahead (commit) log. Since BDB-JE is available in a replicated high-availability edition we inherit those properties in HBase-BDB. Removing HDFS from the picture takes away several convenient mechanisms that underlie HBase's elasticity architecture. To make up for this loss we design and implement new efficient elasticity mechanisms suitable for HBase-BDB. Overall, our key contributions in this paper are

- Design and implementation of a distributed key-value store architecture maintaining HBase's front-end and replacing HDFS with log-structured B+-trees over direct-attached file systems, improving performance and eliminating overheads due to HBase layering
- Novel, efficient elasticity mechanisms for splitting and moving data regions over the direct-attached filesystems

The rest of the paper is organized as follows: in Section II we present a brief overview of HBase, performance challenges, and work that relates to ours. In Section III we describe the architecture of HBase-BDB focusing on the mapping of the HBase data model to the B+-tree indexed storage manager and the elasticity mechanisms whose efficiency is key to adapting to growing volumes of data. In Section IV we present our extensive evaluation focusing on both performance and elasticity using Yahoo! YCSB and OLTP (TPC-C) benchmarks and in Section V we conclude.

II. Background

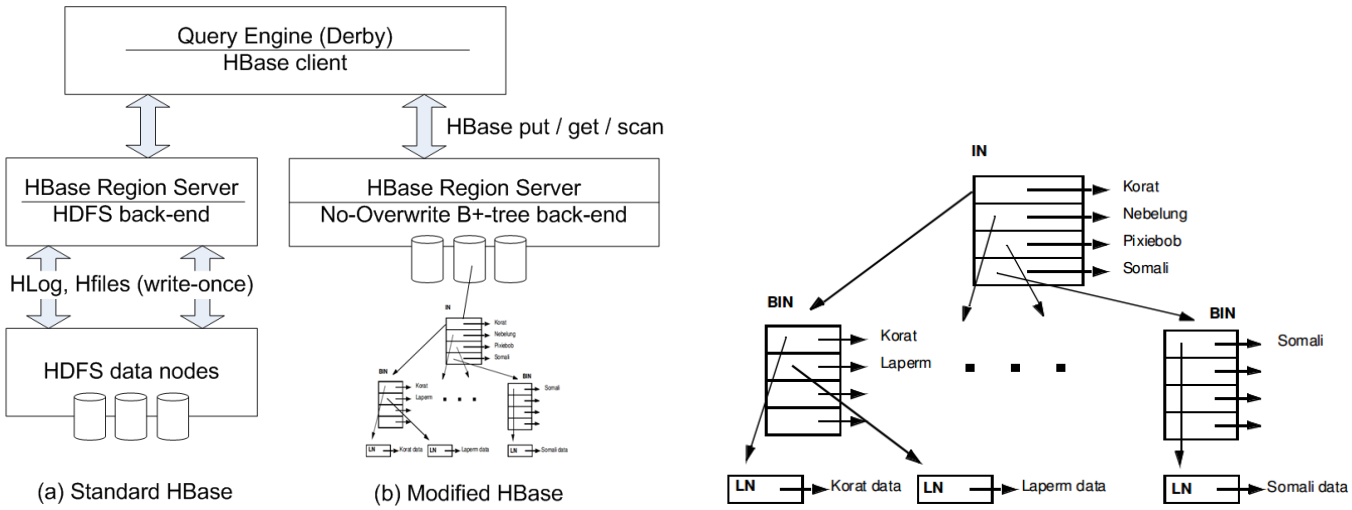
Apache HBase [3] is a column-oriented database management system modeled after Google's BigTable [4] and running over the Hadoop Distributed File System (HDFS) [7]. It is a distributed store whose primary objective is the hosting of very large data sets on clusters of commodity hardware. HBase supports a lightweight schema for semi-structured data. It consists of a set of associative arrays, named tables. Its data model, depicted in Figure 3, comprises

- Row: Each table comprises of a set of rows. Each row is identified through a unique row key.
- Column family: The data in a row are grouped by column family. Column families also impact the physical arrangement of data stored in HBase.
- Column qualifier: Data within a column family is addressed via its column qualifier. Column qualifiers are added dynamically in a row and different rows can have different sets of column qualifiers.

The basic quantum of information in HBase is a *cell* addressed by row key, column family and column qualifier. Each cell is associated with a timestamp. In this way HBase is able to keep different versions of HBase cells.

Figure 1 ([3]) shows the overall architecture of HBase. HBase consists of a master node named HMaster, responsible for keeping the catalogue of the database and also managing Region servers. Region servers are responsible for horizontal partitions of each table called *regions*. Each region contains a row-key range of a table. Clients initially contact HMaster, which directs them to the appropriate Region server(s) currently hosting the targeted region(s). HBase uses the Apache Zookeeper [13] coordination service for maintaining various configuration properties of the system.

HBase uses a log structured storage organization with the LSM-trees [14] indexing scheme at its core, a good fit for the HDFS append-only file system [7] it operates on. Log-structured storage organization goes back to the log-structured file system (LFS) [12] and POSTGRES [15]. It is a popular design choice in a variety of systems to this day [16][17][18][19] for achieving high performance and availability. Records inserted in an LSM-Tree are first pushed into a memory buffer; when that buffer exceeds a certain size, it is sorted and flushed to a disk segment in a log fashion, named *HFile*. Read or range queries examine the



(a) Distributed key-value store with B+ tree backend

(b) Berkeley DB B+ tree structure

Figure 2: (a): HBase vs. HBase-BDB; (b): BerkeleyDB Java Edition index structure [11]

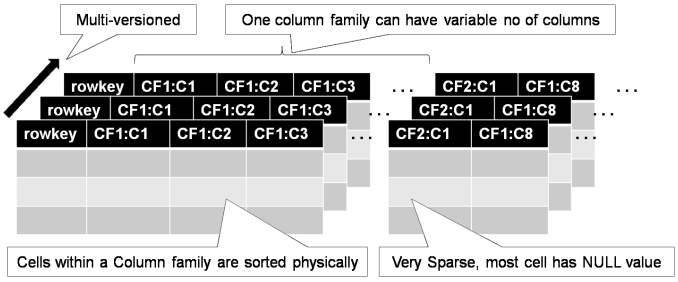


Figure 3: HBase data model (unchanged by HBase-BDB)

memory buffer and then the set of HFiles of the region. For improving indexing performance, the number of HFiles for a region can be reduced by periodic merging of HFiles into fewer and larger HFiles through a process called *compaction* (similar to a merge sort). As the data set grows, HBase scales by splitting regions dynamically and eventually moving regions between region servers through move operations. A move in HBase transfers management responsibility. The actual data transfer takes place through future compaction operations. Since HDFS always creates a local replica at the node initiating the write to a new file, the process maintains data locality. For durability, HBase keeps a write-ahead-log (WAL) in each region server. A WAL is an HDFS file to which all mutations are appended.

The compaction operations necessary to maintain acceptable read performance and the need to have a separate WAL in HBase lead to performance issues observed in large-scale HBase installations. A key cause is the fact that a lot of the HBase-HDFS machinery operates at the file level: Compaction for example, is a file-level operation, leaving database state intact. Replication is also done at file level. As a consequence each compaction operation triggers replication, typically involving three hosts, for the newly created files. This results in large amounts of network traffic and especially write activity reaching the devices [8]. Since HDFS is an append-only file system, reorganizing and merging files requires whole-file copies and is not amenable to known optimizations such as VTree [20]. Previous studies [20] [21] [22] have pointed out to the overheads of compaction in LSM trees. bLSM [22] proposes a scheduler that bounds write latency by appropriate scheduling of compactions. It does not however address the network load and write amplification issues of HDFS. VT-tree [20] proposes a heuristics-based approach that reduces merge operations but cannot be applied on top of an append-only file system. LogBase [21] focuses on the impact of the write-ahead-log and data approach followed in HBase and suggests a B+-tree based design on top of HDFS. Our approach relates to LogBase in that we also propose an alternative layering of HBase. Not being reliant on HDFS however, means that we had to design new elasticity mechanisms for our architecture.

III. Design and implementation

HBase-BDB is the result of re-engineering HBase to replace its LSM-Tree implementation over an HDFS back-end with the use of a collection of Berkeley Database (BDB) Java Edition (JE) [11] storage managers over local file systems. This design leverages the log structured [12] B+ tree implementation at the core of BDB-JE. Since data at each node are stored and organized as a full log there is no need for a distinct WAL, eliminating one cause of write amplification in HDFS. Additionally, the aggressive storage reorganization needed in HBase-HDFS to improve random read performance is not needed in HBase-BDB. BDB-JE still needs to reorganize its storage layout to reclaim space; however this is a far less aggressive operation compared to HBase-HDFS. Furthermore, HBase-BDB maintains elasticity properties (which HBase-HDFS achieves through the use of HDFS) by re-implementing two key operations (split, move) with minimum service disruption. The use of local file systems allows any node to reorganize its data without affecting any other node or any of its replicas. In the following sections we describe more details on the design of HBase-BDB.

Figure 2b depicts the structure of the BDB B+ tree, which consists of Internal Nodes (IN), Bottom Internal Nodes (BIN), and Leaf Nodes (LN) which hold the (key, offset-to-disk for value) pair. A single instance of BDB can manage multiple databases (a BDB database maps to an HBase region in HBase-BDB) writing everything to a logical (per database) log, which is the only on-disk structure. A log is implemented as a number of physical files of configurable size. Below we present the schema mapping of HBase in HBase-BDB and its basic operations.

A. Schema mapping and basic operations

HBase-BDB currently supports the full range of the HBase client API (read, insert, update, scan) on the HBase schema. Below we present the mapping mechanism.

Mapping each cell to a key/value pair would increase significantly the key space, introducing significant overhead. For this reason we map each row to a key/value pair. In more detail the key comprises the row key concatenated with the row timestamp, so multi-version control is still supported in HBase-BDB. For example if the row with key Key1 is created at timestamp T1 and later modified at timestamp T2, two different tuples (with keys Key1T1 and Key1T2) are stored in BDB. To store rows with the same key prefix in reverse chronological order (useful in scan operations), the actual timestamp stored is (MaxLongValue – timestamp). Data payload of the value pair consists of column qualifiers with their corresponding data. HBase-BDB maintains a separate database for each column family, just as standard HBase does. This means that in the general case each region, which can have multiple column families, maps to a set of BDB databases.

In the current version of HBase-BDB partial updates to a row (e.g., changing a single column qualifier) need to be performed as full row updates (read, modify, write) due to BDB API restrictions. Since targeted workloads have a low write percentage [8], we believe that this is a manageable cost for row sizes of practical interest. However, if even this is not the case appropriate modifications to BDB internals can optimize the update operation. A get operation over a time range retrieves all values stored for the requested key in that range. Scans specify a range of keys in addition to a time range. A delete erases all versions of the specified key.

B. HBase-BDB write and read path

For efficiency BDB uses a log buffer for writing to disk (default size 3MB), issuing periodically sync to disk operations. BDB writes all put requests into a *log buffer*. When that buffer fills up a writeToFile() operation is issued which copies its content to the OS buffer cache. Put operations issued while a writeToFile takes place do not block; instead they are written to a *write queue* which is flushed to the log buffer after completion of the writeToFile. A *checkpointer thread* periodically issues sync-to-disk operations. The checkpointer thread wakes up after a configurable number of bytes are written to the buffer cache or a configurable number of seconds have passed since its last sync operation. Sync-to-disk operations are also issued after the writing of the header in a physical file or after the close operation in a physical file.The write path of a put(key, value) operation consists of the following steps (for simplicity we assume transactions are disabled):

1) Find the appropriate BIN for key.
2) Take ownership of the latch (physical lock) for this BIN
3) Create the LN, Serialize key,value pair
4) Lock the LogBuffer
5) Append to the LogBuffer the modifications (LN and modified BINs and INs)
6) If Log Buffer is full writeToFile()
7) Unlock LogBuffer
8) Update BIN
9) Release latch

On receiving a put request, an HRegion server first deserializes the request into an application buffer. We modified the deserialization method on the server side (the HBase client remains unaffected) to produce an appropriate byte format for storing it directly to BDB, in order to avoid unnecessary byte-manipulation overhead. Next, each row key is concatenated with a timestamp to be able to store multiple versions of a row. As an example of ordering a valid key sequence is: $[user1\ |3]$, $[user1\ |2]$, $[user1\ |1]$, $[user2\ |3]$, $[user2\ |2]$, $[user2\ |1]$, $[user3\ |3]$, $[user3\ |2]$, $[user3\ |1]$. The put operation writes to the log buffer of BDB and then the region server responds to the HBase client.

Read and scan operations use BDB cursor objects for retrieving rows. BDB cursors provide extensive search support for fetching rows based on a criterion in addition to exact matches. This feature is very useful for fetching different versions of a row. For each read/scan request the row key is concatenated with the request timestamp and the corresponding rows are fetched.

C. Elasticity mechanisms

Distributed key-value data stores must be designed with elasticity in mind to rapidly adapt to growing (or shrinking) datasets. Key operations for elasticity are the splitting of data regions and their movement between servers. HBase takes advantage of properties of the underlying storage layer (HDFS) to achieve rapid region splits and moves as follows: A region in HBase is stored as a group of sorted HDFS files. At split time HBase performs a *virtual* split by creating two daughter regions corresponding to the two halves of the region (but point to the same HDFS files) and defers data movement to the next compaction operation. At that time it performs the actual data movement operation separating the data files of the newly created regions. Re-assignment of regions to region servers (a move) is again performed *virtually* through HDFS file close and open operations: The region server responsible for the transferred region receives a close-region command from the HMaster and the destination region server receives an open-region command. Future compaction operations will ensure locality of the data.

The HBase master maintains the mapping between tables and their regions. The master has no knowledge of the physical location of region HFiles since they are managed by HDFS. Since HBase-BDB replaces HDFS, the shared named space abstraction used for handling regions (e.g., keeping track of the last host that managed any region) is implemented by the *RegionMap* structure, maintained by a Zookeeper instance. Before opening a region, region servers consult this structure to locate and fetch region files.

D. Split operation

In HBase-BDB, we use HBase's strategy for splitting a region based on size. When a region, which maps to BDB database, exceeds a certain size and is about to split, the mid-point (Middle-row-key) in the region's key space [Start-row-key, End-row-key] is determined. Subsequently the region is broken into a daughter region A, which contain keys [Start-row-key, Middle-row-key], and another daughter region B that holds the key space [Middle-row-key, End-row-key]. For calculating the "middle-row-key" we initiate a BDB cursor which points to the start of the region. Then, through the B+ tree index we efficiently skip half the number of rows in the region and thus get to the middle-row-key of the region. Then we create a new copy of the entire region on disk by closing the database and copying all of its underlying log files. The original copy is renamed to "region A" and

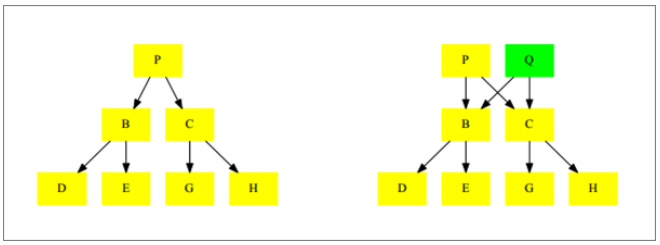


Figure 4: BTRFS copy-on-write operation copying file P to Q [23]

the new copy to "region B". The copy operation on files of size of hundreds of MB to a few GBs, can be an expensive operation in conventional file systems (such as Linux ext4).

To address this problem we leverage the copy-on-write features available in the Linux BTRFS (B-tree) file system [23] to avoid eager in-place updates, replacing them with deferred writes only when the need arises (i.e., when a write actually takes place, if ever). BTRFS has the ability to implement a file copy as a pure metadata operation, just by cloning the root as shown in Figure 4), without eager data movement on the disk device. An actual data copy is issued if and when a data block needs to be updated. In addition to avoiding physical copy of BDB log files to other locations on disk, this methodology reduces disk space usage.

The two daughter regions produced by a copy operation contain several keys that are outside the key ranges and for which they are no longer responsible. Removing those keys is the responsibility of a garbage collector thread, called GBCollector, that is executed periodically, and whose sole task is to delete out-of-range rows. Delete operations issued by GBCollector at the BDB level translate into tomb stoning keys (that is marking them for deletion, without actually deleting them) in BDBs underlying log files. Removing the physical file space of those keys is the responsibility of the Berkeley DB cleaner process. When a physical's log file valid data drops under a configurable threshold, its data are appended to the end of the log and space is reclaimed by deleting the file. The efficiency of the BDB cleaner is a key performance parameter: Fewer and/or smaller log files backing regions means lower cost in moving (transferring) those log files over the network, reducing the cost of future region movement operations.

The above procedure describes a failure-free scenario. To handle failures during a split operation we record information about ongoing such operations in a Zookeeper structure (directory) called *split_transactions*. A split transaction starts by recording the names of the involved regions (parent, daughterA, dayghterB) as 3 records (*znodes* in ZooKeeper terminology) under /split_transactions. On successful completion these records are erased. In the case of failure, a recovery operation examines this structure; if information about an interrupted transaction is found, a recovery action

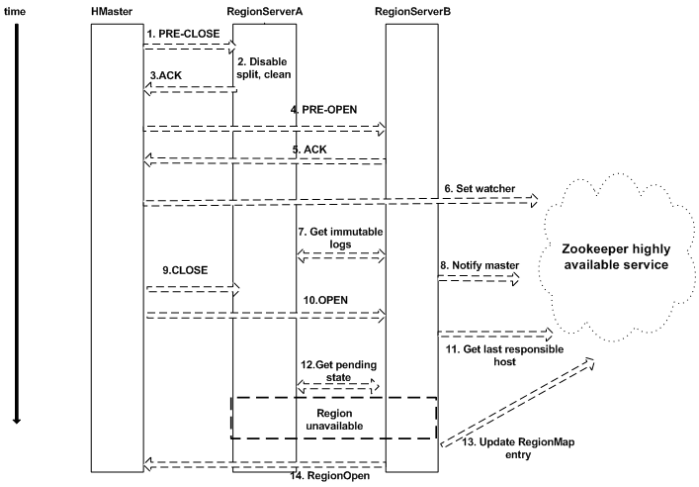


Figure 5: Move operation protocol shown in steps

(undo or redo) can bring the regions to a consistent state.

E. Move operation

In HBase-BDB a region move operation is implemented via network data transfer between two region servers, as shown in Figure 5. A brute force implementation could result to long unavailability of the region during data transfer. To minimize the window of unavailability we introduce a mechanism that *prefetches* region data at the target server prior to closing it at the source server. The mechanism is safe by exploiting the immutability of BDB logs: log segments that are closed (and thus only to be read in the future) can be transferred without blocking. The protocol proceeds as follows (Figure 5): When the master decides to move a region it sends a PRE-CLOSE command to the region server currently handling the region (*RegionServerA*). On receiving the command, *RegionServerA* disables split and cleaning operations for the specific region while continuing to serve read and write operations for it. Split and clean operations are disabled for ensuring log immutability in case of log reorganization. On successful reply from *RegionServerA*, the master sends a PRE-OPEN command to the target region server (*RegionServerB*). On successful reply from RegionServerB it sets a *watcher* (request for notification upon a change) on a Zookeeper instance on a node called `/MOVE/RegionName` that contains RegionServerA hostname. *RegionServerB* then contacts RegionServerA to ask for a list of the region's physical log files. The log file names follow the format *(LogIDNumber).jdb*, where *LogIDNumber* is an increasing counter of the physical log files kept internally by BDB. RegionServer B then fetches all log files except the current working file (the one with the maximum *LogIDNumber*). After fetching successfully all the log files it notifies the Master through the watcher previously set in Zookeeper. When the Master receives the notification it issues a CLOSE region command to *RegionServerA* which will sync and close the region. Then the master sends an OPEN command to *RegionServerB*. RegionServerB consults *RegionMap* structure. Continuing, it will get any log files that were not already part of the prefetch stage. Since the majority of operations are reads we expect this time to be minimal. To optimize network throughput and leverage core parallelism we use a number of concurrent connections with a dedicated thread per connection. Additionally we reduce CPU overhead during the actual log transfer by exploiting the `sendFile()` system call via the `transferTo()` method of the Java nio package[3] at *RegionServerA*. Fault tolerance during a move is handled similarly to a split. As a move does not modify state (other than creating data files at the destination), it is possible to easily abort or complete a move transaction (we currently opt for the latter).

Finally, support for replication is built into HBase-BDB via use of BDB-JE's high-availability option and HBase-BDB makes straightforward use of it. HBase-BDB manages replicas by extending RegionMap to maintain replica groups of each region. Full management of replica groups by HBase-BDB is still ongoing work. In this paper we concentrate on non-replicated configurations as a full exposition would require far more space than available in this paper.

IV. Evaluation

A. Platform tuning

Appropriate configuration of HBase is known to be key to achieving good performance [4]. In particular, long stop-and-halt garbage collection (GC) periods in a default configuration typically arise from (1) delaying to start GCs, and; (2) memory fragmentation caused mainly by memstore flushes. To address (1) we start a GC when occupied heap exceeds 60% of available space which is typically higher than the default; we also use the Parallel New collector for the young generation objects and the Concurrent Mark-Sweep collector for the old generation objects in the heap. We address (2) by allocating contiguous regions of memory for memstores (setting the *hbase.hregion.memstore.mslab.enabled* property), thus reducing memory fragmentation. We thus ensure that best practices in HBase configuration are followed to avoid interference with our evaluation.

B. HBase vs. HBase-BDB: YCSB benchmarks

To determine the performance tradeoffs between standard HBase (referred as HBase-HDFS) and HBase-BDB we compare the two systems on a two-machine client server setup using the YCSB [24] benchmark. The machines used in these experiments have 4 core Intel Xeon processors at 2 GHz with 6GB of memory and support 8 hardware threads. The server machine hosts a single region server,

[3]http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html

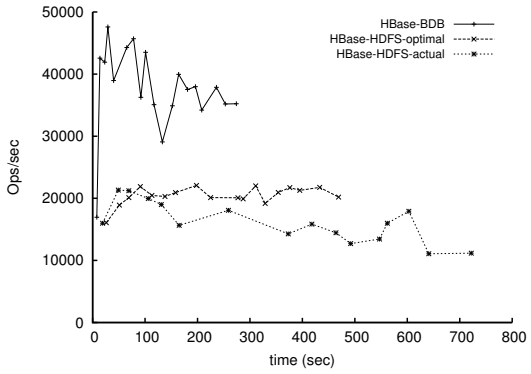
[4]http://blog.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-1/

an HBase master, a single HDFS namenode (HBase-HDFS only), a single HDFS datanode (HBase-HDFS only) and a Zookeeper node. The server machine dedicates a Western Digital WD5001AALS disk rotating at 7200 RPM (including a 32 MB cache) for storage. The file system used is ext4. Note a difference from the setup used in elasticity experiments, which has an additional disk per server with the Linux BTRFS file system. The client machine is dedicated to running the YCSB benchmark. The versions of Hadoop and HBase used are 1.2.1 and 0.94.13 respectively. To ensure HBase-HDFS and HBase-BDB are at an equal footing we set both the HBase block cache and BDB cache to 1GB. The setting is small (1/15th of the dataset) to ensure I/O intensive workloads. We use variations of HBase-HDFS setups with Bloom Filters enabled and disabled to value their impact on read performance. YCSB supports different types of operations (write, read, or scan) on a single table with rows of size 1KB. YCSB populates a uniformly balanced database with 8 pre-created regions with 10 million rows resulting in a 15GB dataset in all cases. Read requests follow also uniformly random distribution. Finally scan operations requests are also uniformly distributed and fetch on average 64KB of data.

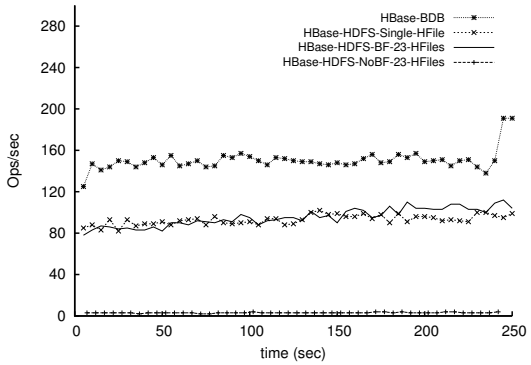
Write workload. For HBase-BDB we use the EVICT_LN cache policy instead of the default LRU, which gives priority to retaining index blocks in the cache over data blocks. We set the BDB log segment size to 32MB so that a sync operation is issued per 32MB. We have tuned HBase-HDFS memstore size to create HFiles[5] of over 70MB in size and set HDFS block size to 256 MB. This was done so that a newly created HFile fits within an HDFS block. Finally, we have disabled splits since they are going to be studied in the next experiments. In the following graphs we depict the operations per sec observed during the population of the database. Since the database size is the same for all setups time for completion of each setup varies.

We consider the default version of HBase-HDFS that performs compactions with the default settings (minimum and maximum thresholds 3 and 5 HFiles respectively) and throttles writes when the number of HFiles reaches 7 per region. The latter limit is set to ensure that HBase eventually matches the speed of writing processes with its ability to compact HFiles so that read and scan latency remains bounded. We call this version HBase-HDFS-actual and represents a typical real world setup of HBase. We also consider a version of HBase-HDFS that does not perform compactions (HBase-HDFS-write-optimal). While this setup is optimal for writes, this version will in time result into unboundedly high read latencies as it is shown in later read experiments. However, it provides an upper bound of what write throughput would be possible if compactions had no resource impact. In figure 6a we depict the write operations/sec during a database population. In all of the three setups CPU load is close to saturation at the server node. As we can see from 6a HBase-BDB performs about 80% (average 36000 ops/sec) better than HBase-HDFS-write-optimal (20000 ops/sec). HBase-BDB operates at nearly disk speed due to its log structure. On the other hand, HBase-HDFS optimal and actual write at least twice to HDFS (once to a write-ahead log (WAL) and then to HFiles) resulting in

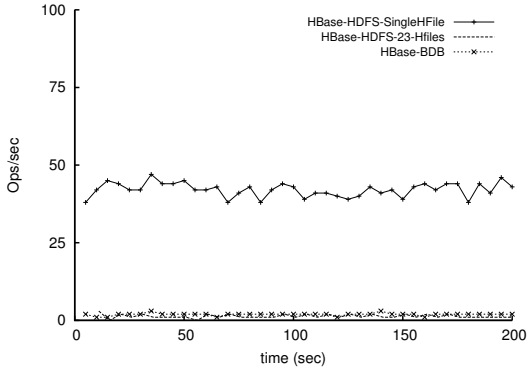
[5]HFiles are HDFS files resulting from HBase compactions.



(a) Random write



(b) Random read



(c) Scan

Figure 6: YCSB throughput for random write, read and scan workloads

extra overhead in the write path. Moreover, the additional work caused by compactions drops the overall performance of HBase-HDFS-actual 30% compared to optimal. We thus conclude that the need to be proactive on compaction for maintaining acceptable levels of read performance and the use of a WAL can be a limiting factor compared to a log-structured B+-tree where cleaning is also eventually needed but does not stand on the critical path.

Read workload. In this experiment we run a random read workload on HBase-BDB and three versions of HBase-HDFS: HBase-HDFS-SingleHFile, in which the entire dataset has been compacted into a single HFile; HBase-HDFS-BF-23HFiles, in which we have approximately 23 HFiles per region; and HBase-HDFS-NoBF-23HFiles, which differs from the latter in that row key Bloom Filters are disabled. HBase-HDFS block size is set to minimal value of 32KB as suggested[6] to best match YCSB key/value size. In Figure 6b we observe that HBase-BDB has about 30% better throughput than HBase-HDFS-SingleHFile, both limited by disk's random read I/O. Bloom filters for HBase-HDFS have a high hit ratio which leads to an almost identical performance between HBase-HDFS-SingleHFile and HBase-HDFS-BF-23-HFiles. Disabling bloom filters lead to a dramatic drop in read performance as we can see from HBase-HDFS-NoBF-23HFiles. As we conclude, HBase-BDB random read operation performs better than HBase-HDFS. It is benefit by its B-tree dense indexing which results to more efficient IO with the device.

Scan workload. In this experiment we study a scan workload with sizes drawn from a uniform distribution fetching on average 64KB of data per scan). Figure 6c depicts performance of the HBase-BDB, HBase-HDFS-SingleHFile, and HBase-HDFS-23-HFiles configurations. HBase-HDFS-SingleHFile scan performance is an order of magnitude better than the two other setups. This is because the storage layout of HBase-HDFS is particularly optimized for scans: in each scan, HBase-HDFS position the cursor to the correct position in the HFile and reads consecutive data blocks. As the number of HFiles increases (which will be a common case in an actual deployment), scan performance drops. This is because contrary to reads, scans do not benefit by the use of Bloom filters: fetching the next N rows starting from row r requires finding the starting position in all HFiles and memstore and bring to memory the N consecutive blocks from all HFiles that are needed by the query. HBase-BDB exhibits comparable performance to HBase-HDFS-23-HFiles as it generally produces a non-linear I/O pattern to disk for fetching data blocks for each row due to the non-uniform storage of consecutive row keys. In conclusion, scan performance is better for HBase-HDFS than HBase-BDB only when HFiles number is limited to small number. Performance is comparable in all other cases. However this comes to the cost of frequent compactions that stand in the critical path. Also scans consist a small portion of the read workloads [8]. HBase-BDB scan performance could also benefit by improved cleaning.

[6]http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/io/hfile/HFile.html

C. TPC-C experiments on a Cloud platform

Our experimental setup consists of 6 Ubuntu-12.04 virtual machines (VMs) provided by Flexiant and managed by Flexiant Cloud Orchestrator (FCO). All VMs have 4 virtual cores, 4GB of memory and an additional disk with a BTRFS filesystem for storing data. Each VM runs Linux Ubuntu-12.04 with kernel 2.5.0-24-generic-pae. One VM hosts the HBase Master, Zookeeper and Namenode. Derby [25] and the TPC-C benchmark are hosted in separate VMs. Each region server uses 1GB of memory as a shared cache across all underlying region instances. The experimental setup is depicted in Figure 7.

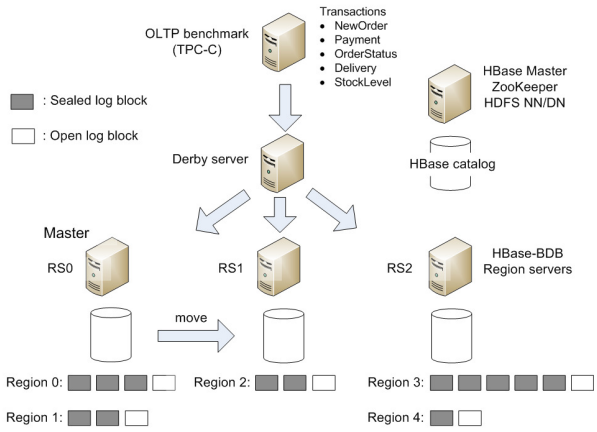


Figure 7: Experimental setup

TPC-C database consists of 11 hbase tables as it is shown in descending order of accesses in table I. Derby maps each sql tuple into an HBase row. The transaction mix consists of 45% NewOrder, 43% Payment, 4% of StockLevel, 4% OrderStatus, and 4% Delivery transactions generated by a total of 100 threads. The TPC-C workload consists mostly of read operations: 88% of the workload consists of transactions with SELECT and INSERT SQL operations based on primary key, mapping to get and put HBase operations (80%-20% respectively); the remaining 12% of the transactions include range queries, which map to HBase scan operations of average length 15.

Split operation In the first experiment we study the characteristics of region split operations during the TPC-C population process, with a 6GB dataset load balanced across three region servers. During population no move operations take place. HBase load balancer is disabled so no move operations take place. The split operation includes the following stages:

HBase Table	Read ops	Write ops
ORDER_LINE	98%	2%
STOCK	95%	5%
NEW_ORDER	94%	6%
ITEM	100%	0%
CUSTOMER	81%	19%
DISTRICT	61%	39%
OORDER	74%	26%
WAREHOUSE	76%	24%
IDX_CUSTOMER_NAME	100%	0%
HISTORY	100%	0%
IDX_OORDER	9%	91%

Table I: TPC-C workload characterization

1) Disable write/read operations for region and wait outstanding operations to finish (Wait time)
2) Close and sync underlying database of region to be splitted (Close/Sync time)
3) Perform a copy of database log files in a different location on the same device creating two Daughter Regions, A and B (CopyRegion time)
4) Open and rebuild state for Daughter Region A (OpenDaughterA time)
5) Open and rebuild state for Daughter Region B (OpenDaughterB time)

A split operation is triggered when the number of valid keys in a region exceeds 131,072. In the population phase we observe a total of 37 split operations, separated in two groups according to the relative size of their physical logs: 28 splits of 142MB (SMALL) and 9 splits of 244MB (LARGE).Figure 8 depicts the total split time and the breakdown for each stage (including wait time for outstanding operations to finish) in the case of SMALL and LARGE regions. We observe that locally copying a region's physical log files (CopyRegion) is an efficient operation and nearly independent of region size due to BTRFS's indirect copy feature, taking time comparable to the time spent for opening and closing a database.

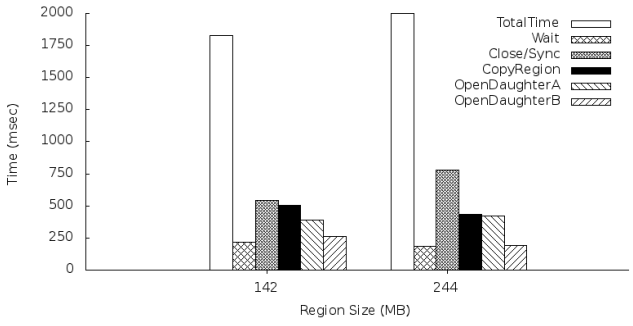
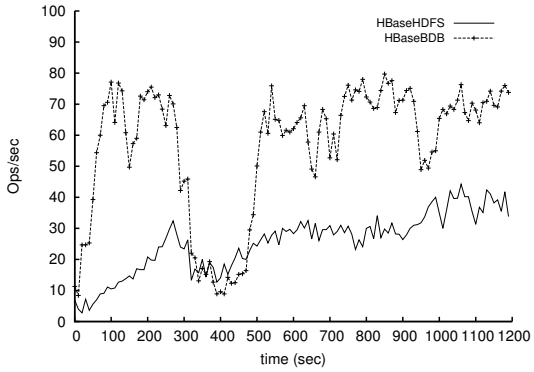
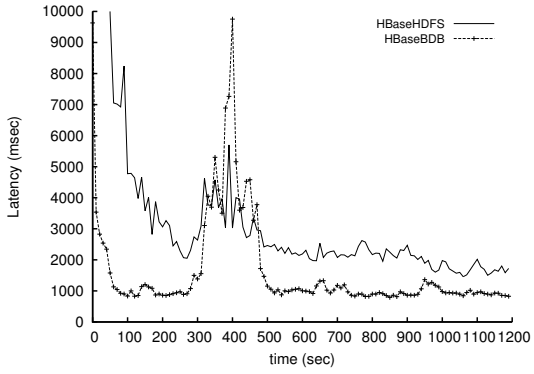


Figure 8: Time duration breakdown of split operation stages

Move operation In this experiment we study TPC-C performance during intensive elasticity actions occurring during data redistribution in a growing cluster. In our initial setup (same for both HBase-BDB and HBase-HDFS) we have two region servers with an equal share of data regions. Each region in the case of HBase-HDFS has previously undergone a major compaction down to a single HFile and is collocated with the corresponding datanode server of HDFS. We have disabled the default load balancer and implemented a custom region moving policy in both cases. At 300 sec a new region server is added to the cluster and we move 1/3 of the total number of regions (initially hosted in the two servers) to the newly added server, initiating a new move every 4 sec. Data movement in the case of HBase-HDFS takes place after a move operation by means of a major compaction, causing region HFiles to move to the newly added server. Figures 9a and 9b depict TPC-C performance over a 20 min period.



(a) TPC-C throughput



(b) TPC-C response time

Figure 9: TPC-C average throughput and response time

In Figures 9a and 9b we can identify three phases: a stable phase in which no move operations take place (0-300 sec), an elasticity phase (300-500 sec) where data is moved to the newly added server, and finally another stable phase after 500 sec. HBase-BDB performance during the stable phase is limited by the CPU of the Derby server; during the elastic phase the bottleneck moves to the CPU of the region servers (which are then engaged in network transfers). HBase-HDFS performance is always limited by the region server CPUs,

mainly due to the large number of random reads on small (200-300 byte) rows. Both systems are impacted during the elasticity phase (300-500 sec). HBase-HDFS (being the more CPU bound system at the region servers) is seen to benefit more from the addition of a third server.

V. Conclusions

In this paper we presented HBase-BDB, a distributed key-value store that shares HBase's data model and data distribution mechanisms but departs from it in the use of a log-structured B+-tree indexed storage back-end over locally-attached files systems. With the use of a log structured key value store combined with novel elasticity mechanisms HBase-BDB is able to improve over HBase in random read and write YCSB workloads by 30% and 85% respectively. HBase-BDB lags behind HBase only in random scans; these however are only a small share of overall operations in popular workloads such as e-mail, SMS, Chat, etc. Support for elasticity in HBase-BDB is shown to be effective in TPC-C experiments yielding similar availability and performance impact to what is achievable with HBase-HDFS.

VI. Acknowledgments

We thankfully acknowledge the support of the CoherentPaaS (FP7-611068) and PaaSage (FP7-317715) EU projects.

References

[1] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.

[2] M. Welsh and D. Culler, "Virtualization considered harmful: OS design directions for well-conditioned services," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001, pp. 139–144.

[3] Apache HBase, http://www.hbase.org.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[5] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *ACM SIGPLAN Notices*, vol. 31, no. 9, 1996, pp. 84–92.

[6] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A scalable distributed file system," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997, pp. 224–237.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[8] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of HDFS under HBase: a Facebook messages case study," in *Proceedings of USENIX Conference on File and Storage Technologies*, 2014, pp. 199–212.

[9] A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya, "Storage infrastructure behind Facebook messages: Using HBase at scale," *IEEE Data Eng. Bull.*, vol. 35, no. 2, pp. 4–13, 2012.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5, 2003, pp. 29–43.

[11] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *USENIX Annual Technical Conference, FREENIX Track*, 1999, pp. 183–191.

[12] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[14] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[15] M. Stonebraker, *The design of the Postgres storage system*. Morgan Kaufmann Publishers, 1987.

[16] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011, pp. 29–41.

[17] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis, "Corfu: A shared log design for flash clusters." in *Proceedings of USENIX Conference on Networked Systems Design and Implementation*, 2012, pp. 1–14.

[18] M. Vrable, S. Savage, and G. M. Voelker, "BlueSky: a cloud-backed file system for the enterprise," in *FAST*, 2012, p. 19.

[19] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 1–14.

[20] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with VT-trees." in *Proceedings of USENIX Conference on File and Storage Technologies*, 2013, pp. 17–30.

[21] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "LogBase: a scalable log-structured database system in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1004–1015, 2012.

[22] R. Sears and R. Ramakrishnan, "bLSM: a general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 217–228.

[23] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Transactions on Storage*, vol. 9, no. 3, p. 9, 2013.

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[25] R. Vilaça, F. Cruz, J. Pereira, and R. Oliveira, "An effective scalable SQL engine for NoSQL databases," in *Distributed Applications and Interoperable Systems*. Springer, 2013, pp. 155–168.