Lightweight Indexing for Log-Structured Key-Value Stores

Yuzhe Tang † Arun Iyengar ‡ Wei Tan ‡ Liana Fong ‡ Ling Liu § Balaji Palanisamy ♯

†*Syracuse University, Syracuse, NY, USA, Email:* ytang100@syr.edu

§*Georgia Institutue of Technology, Atlanta, GA, USA, Email:* lingliu@cc.gatech.edu

‡*IBM T.J.Watson Research Center, Yorktown, NY, USA, Email:* {aruni, wtan, llfong}@us.ibm.com

♯*University of Pittsburgh, Pittsburgh, PA, USA, Email:* bpalan@pitt.edu

Abstract

The recent shift towards write-intensive workload on big data (e.g., financial trading, social user-generated data streams) has pushed the proliferation of log-structured key-value stores, represented by Google's BigTable [1], Apache HBase [2] and Cassandra [3]. While providing key-based data access with a Put/Get *interface, these key-value stores do not support value-based access methods, which significantly limits their applicability in modern web and database applications. In this paper, we present* HINDEX*, a lightweight real-time indexing scheme on the log-structured key-value stores. To index intensively updated big data in real time,* HINDEX *aims at making the index maintenance as lightweight as possible. The key idea is to apply an append-only design for online index maintenance and to collect index garbage at carefully chosen time.* HINDEX *optimizes the performance of index garbage collection through tightly coupling its execution with a native routine process called compaction. The* HINDEX*'s system design is fault-tolerant and generic (to most key-value stores); we implemented a prototype of* HINDEX *based on HBase without internal code modification. Our experiments show that the* HINDEX *offers significant performance advantage for the write-intensive index maintenance.*

I. Introduction

In the age of cloud computing, various scalable systems emerge and prevail for big data storage and management. These scalable data stores, mostly called key-value stores, include Google's BigTable [1], Amazon's Dynamo [4], Facebook's Cassandra [5], [3], Apache HBase [2] among many others. They expose a simple Put/Get API which allows only key-based data accesses, in the sense that when writing/reading data in the key-value stores, user applications are required to specify a data key as the parameter. While the key-based Put/Get API supports basic workloads, it falls short when it comes to advanced web and database applications which require value-based data access. To gain wider application, it calls for value-based API support on the key-value stores.

On the other hand, many key-value stores deal with write-intensive big data. Typically, the workload against a key-value store is dominated by data writes (i.e. Put) rather than reads, and such workloads are prevalent in modern web applications. For instance, in Web 2.0, social users not only read news but also contribute their own thinking and write news themselves. It is also the case in other emerging domains, such as large system monitoring and online financial trading. To optimize the write performance, many key-value stores (e.g. HBase, Cassandra and BigTable) follow a log-structured merge design [6], in which the on-disk data layout is organized as several sorted files and writes are optimized by an append-only design. We call these Log-structured Key-Value Stores as LKVS (whose distinctive features are described in § II).

This work addresses the problem of *supporting a value-based API on the write-intensive data stored in* LKVS. For value-based access, a secondary index is essential. In common practice, the secondary index is implemented as a regular table in the underlying LKVS. In this situation, the index maintenance under a write-intensive workload is a challenge: On the one hand, the index maintenance needs to be lightweight in order for it to catch up with the high arrival rate of the incoming data writes; On the other hand, given mutable data where data updates overwrite previous data versions, the index maintenance needs to find and delete the obsolete versions (in order to keep the index fresh and up-to-date); such a task includes Get operations which are very expensive in LKVS systems (explained in § II).

In this paper, we propose HINDEX, a middleware system that supports the secondary index on top of an LKVS. To address the index-maintenance challenge, we propose a performance-aware approach. The core idea is to decompose an index-maintenance task to several sub-tasks, and only to execute the inexpensive ones synchronously while deferring the expensive ones. More specifically, given a data update, the index maintenance needs to perform two sub-tasks, that is, 1) to insert new data versions to the store and 2) to find and delete old versions. Sub-task 1) involves only Put operations, while sub-task 2), called an index repair, requires a Get operation to find the old version. The insight here is that LKVS is write-optimized in the sense of Put being fast and Get being slow, which makes sub-task 1) lightweight and the index-repair sub-task 2) heavyweight. HINDEX's strategy to schedule the index maintenance is to synchronously execute sub-task 1) while deferring the expensive index-repair sub-task.

A core design choice regarding the deferred index repair is when the execution should be deferred to. Our key observation is that a Get operation is much faster when it is executed after a compaction than before that. Here, a compaction is a native maintenance routine in LKVS; it cleans up obsolete data and reorganizes the on-disk data layout. To verify the observation, we conducted a performance study on HBase 0.94.2. A preview of the experiment results is shown in Figure 1; the Get can achieve more than $7\times$ speedup in latency when executed after a compaction comparing to that before a compaction. Based on this observation, we propose a novel design that *defers the index repair to the offline compaction process*. By coupling the index repair with the compaction it can save the index-repair overhead substantially.

While deferring the index repair to offline hours can improve the performance of itself, it may prolong the value-based read access, due to the need to check index-table inconsistency (caused by the online repair). We further propose to schedule the index repair operations online, by piggybacking them in the execution path of value-based reads. The online index repair adds small extra overhead (i.e. one local memory write) but can save huge by removing the need of maintaining another remote base-table Get. Note that unlike most existing online performance optimization schemes, our HINDEX does not need to profile or monitor the system resource utilization.

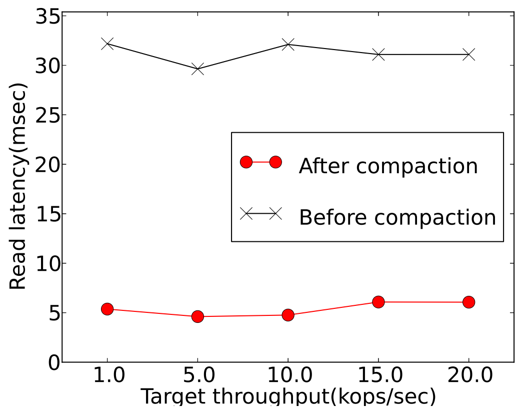


Fig. 1: Read latency before/after compaction

The contributions of this paper are summarized below.

- We coin the term LKVS that abstracts various modern industrial strength big-data storage systems (including HBase, Cassandra, BigTable, HyperTable, etc). We propose HINDEX to extend the LKVS's existing API by including a value-based access method.
- We make the index maintenance lightweight in HINDEX for write-intensive workloads. The core idea is to make it aware of performance; it defers expensive operations while executing inexpensive ones synchronously in an LKVS system. Specifically, we propose two lightweight scheduling strategies for expensive index-repair operations; an offline repair that is associated with the native compaction process and an online repair that is coupled with value-based read operation.
- We analyze the fault-tolerance of HINDEX in terms of both online operations and offline index-repair process. The fault tolerance in HINDEX is achieved without sacrificing performance efficiency.
- HINDEX is designed to be generic on any LKVS. While a generic implementation of HINDEX is realized in both HBase and Cassandra, we also demonstrate an HBase-specific implementation that optimizes performance without any internal code change in HBase (by using exposed system hooks). We open-source the HINDEX prototype on HBase. [1]

[1]https://github.com/tristartom/hindex

II. Background: LKVS

LKVS, represented by BigTable [1]/HBase [2] and Cassandra [3], has the following two common and distinctive features.[2] Note that specific LKVS systems may differ in other aspects (e.g. HBase shards data by range partitioning while Cassandra is based on consistent hashing).

- LKVS employs a key-value data model with a Put/Get API. In the data model, a data object is identified by a unique key k and consists of a series of attributes in the format of key-value pairs; a value v is associated with multiple overwriting versions, each with a unique timestamp ts. To update and retrieve an object, LKVS exposes a simple Put/Get API: Put(k,v,ts), Delete(k,ts) and Get$(k) \rightarrow \{\langle k,v,ts\rangle\}$. When calling these API functions, the presence of a key k is required, which makes them key-based access methods.
- LKVS uses the log-structured merge tree (or LSM) [6], [1] for local data persistence. In an LSM tree, the data writes are buffered in memory and then flushed to disk in a batched and append-only manner. With multiple flushes, it generates multiple on-disk files, and a read need essentially issue multiple random-access calls to those files. This behavior, making writes sequential disk access and reads multiple random access, is the reason behind the fast-Put and slow-Get characteristic of LKVS. An LKVS system typically exposes an administration interface called Compact, which merges multiple data files into one in the LSM tree and performs cleanup tasks for garbage collection. Details of Compact and LSM tree can be found in [6], [1].

III. The HINDEX Structure

In this section we first present the system and data model in HINDEX, then describe the materialization of HINDEX in the underlying LKVS, and at last formulate the problem of index maintenance.

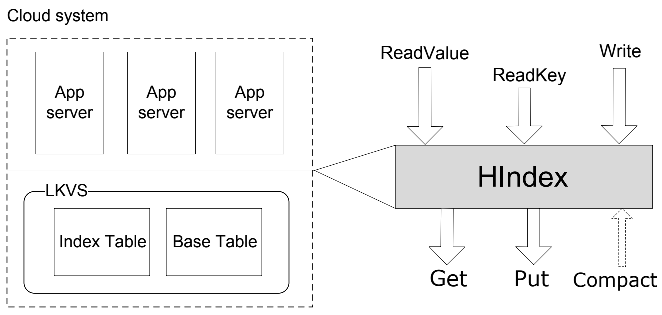


Fig. 2: HINDEX architecture

A. System and Data Model

In a cloud environment, the server system is typically organized into a multi-tier architecture, consisting of application and storage tiers. The application tier processes queries and prepares the data formatting for the writes, while the storage tier is

[2]Note that other than LKVS, there are key-value stores that are read optimized, such as PNUT [7].

responsible for persisting the data. We consider the use of LKVS in the storage tier, as shown in Figure 2. In this architecture, HINDEX is a middleware that resides between the application and storage tiers. To the application servers, it exposes both key-based and value-based API, as described below. The application servers are referred to as "client" (of HINDEX) in this paper. To the underlying LKVS, HINDEX translates the API invocations to the Put/Get operations.

- Write(k,v,ts): Given a row key k, it updates (or inserts) the value to be v with timestamp ts.
- ReadKey$(k,ts,m) \rightarrow \{\langle k,v',ts'\rangle\}_m$: Given a row key k, it returns the value versions before ts, that is, $ts' \leq ts$. HINDEX considers an *m-versioning* policy, which allows client applications to indicate the number of versions deemed as fresh (by m). The method would return the latest m versions of the requested key.
- ReadValue$(v_t,ts,m) \rightarrow \{\langle k',v,ts'\rangle\}$: Given queried value v_t, it retrieves all the row keys k' whose values v matches v_t. Here, v_t is an index token generated from value v; the tokenization process, denoted by $t(v) = \{v_t\}$, depends on different query predicates as discussed below. In addition, the retrieved results should adhere to the m-versioning policy; that is, the result version ts' must be among the latest m versions of its key k' as of time ts.

The first two methods are similar to the existing key-based Put/Get interface (with different internal implementations), while the last one is for value-based data access. In the API design, we expose timestamp ts for the client applications to specify the consistency requirement. In practice, generating a (globally) unique timestamp, if necessary, can be done by existing timestamp oracles [8], [9].

Query flexibility: Based on the new ReadValue API, the HINDEX can support various data types and query predicates. In addition to exact-match queries, for example, HINDEX can support string data and prefix search; in this case, the tokenization would be $t(v) = \{v_t\}$, such as any v_t is a prefix of a given string v. HINDEX can support numeric data with range queries; here $t(v) = v$ and it requires the underlying key-value stores to support range partitioning and scan operation (e.g. that is the case in BigTable and HBase). HINDEX also supports multiple indices and multi-dimension value-based search; it can be supported through issuing multiple ReadValue calls on different indices and intersecting the results. Without generality, we mainly focus on the exact-match query in the rest of the paper, that is, $v_t = v$.

B. Index Materialization

The index data is materialized in a regular data table inside the underlying LKVS. The index table is not directly managed by the client applications; instead, it is fully managed by our HINDEX middleware. In terms of structure, the index table is an inverted version of the base table; when the base table stores a (valid) key-value pair, say $\langle k,v\rangle$, the index table would store the reversed pair as $\langle v,k\rangle$. For different keys associated with the same value in the base table, HINDEX materializes them in the same row in the index table but as different versions; that is, $\langle v,k_1\rangle, \langle v,k_2\rangle$ are two versions in one cell in the index table. The versioning is disabled in the index table, and all obsolete index versions are required to be deleted explicitly.

In the case of skewed data distribution, it could occur that certain index rows for common values are huge. It may result in a long list of query results by a ReadValue call. In this situation, HINDEX provides a pagination mechanism to limit the number of key-value pairs in a ReadValue result and relies on applications to indicate such limit (and offset). Specifically, we allows an optional parameter p as in ReadValue$(v,ts,m,[p])$ which limits the number of ReadValue results up to p. Currently, we assume there is fixed ordering between values of the same key (e.g. based on the hash of the values) so that paginated results will not overlap.

C. Index Maintenance: Design Choices

In this work, we focus on the problem of maintaining the index table in LKVS. In the spectrum of the design space, the most write-optimized approach (baseline 1) is not to maintain the indices online (or maintain them offline), which can have no write amplification but at huge expenses of ReadValue latency; now it will need a full-table scan for processing a single ReadValue query. On the other end, the most read-optimized approach (baseline 2) would synchronously update the indices in place; that is, to keep every index entry up-to-date based on the latest data updates. While the no-maintenance approach works well in the case that there are no (online) ReadValue calls, the update-in-place approach would fit in read-intensive workloads. However, neither approach is suitable for our targeted workload which is write-intensive yet with considerable amount of ReadValue's.

To explore the design space, we formulate the design choices from two angles: 1) How to decompose an index-update task and express it in terms of Put/Get/Delete operations, 2) When to schedule the execution of those operations. For design aspect 1), there are choices, such as not to decompose but treat an index-update task as a whole, or to decompose it to two sub-tasks (i.e. index insert and repair). For design aspect 2), there are generally three scheduling choices; synchronously online, asynchronously online, and offline. Their designs range from being read-optimized to write-optimized.

Under this model, we can re-examine the baseline approaches and our HINDEX approach. Illustrated in Figure 3, we can see that HINDEX is optimized towards write-intensive workloads mixed with certain amount of (index) reads. This design choice is made based on the unique characteristic in IO performance of the underlying LKVS. Concretely, HINDEX approach is to 1) synchronously schedule index-insert sub-task (§ IV-A), 2) defer the execution of index repair sub-task with online option (§ IV-B) and offline option (§ V).

IV. Online HINDEX

This section describes the design of online HINDEX in terms of index maintenance, query evaluation and analysis of fault tolerance.

A. Put-Only Index Maintenance

Given a data update as a key-value pair $\langle k,v\rangle$, the index maintenance needs to include four tasks to keep both the index and base table up-to-date: 1) Deleting old versions associated with key k in the index table. This causes a Delete(v',k,ts')

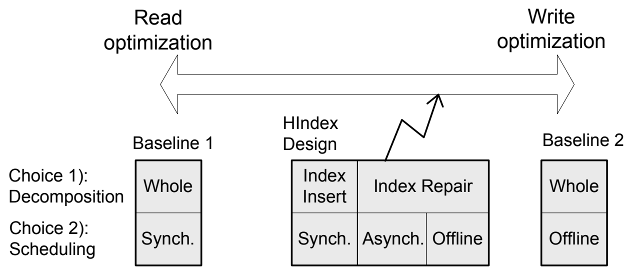


Fig. 3: Design choices of HINDEX maintenance

call, in which v' is the old version obsoleted by new version $\langle k,v\rangle$. Since old version v' is unknown from the original data update $\langle k,v\rangle$, it entails a Get operation to read the old version; 2) Reading the old version from the base table. This causes a call for Get$(k)\rightarrow\langle k,v'\rangle$; 3) Inserting new version to the base table, which causes Put(k,v,ts); 4) Inserting new version to the index, which causes Put(v,k,ts).

A straightforward way to execute the index maintenance process is to *synchronously* execute all the four operations, which is essentially what the traditional update-in-place indexing technique does (which is also widely used in many cloud databases [10], [11]). However, this strategy causes performance problems when applied to the LKVS case: Recall that a Get operation in LKVS is slow, and by attaching expensive Get to the data write path, it could increase the write overhead and slow down the system throughput, especially when the workload is dominated by data writes. To improve the online index maintenance efficiency, HINDEX employs a simple strategy to execute the Put-only [3] operations (i.e. operations 3) and 4)) synchronously and defer the execution of expensive index repair operations (i.e. operations 1) and 2)) to later time. Algorithm 1 illustrates the online Write algorithm. The two Put calls are annotated with the same timestamp ts. Here, we deliberately put the index update ahead of the base table update for the fault-tolerance concerns which will be discussed.

B. Processing Reads

The Put-only index maintenance may lead to obsolete index data (e.g. $\langle v',k\rangle$ is present in the index table after $\langle k,v\rangle$ is written). This requires a ReadValue query to always check whether an index entry is fresh. Given an index data $\langle v',k\rangle$, the freshness check is done by checking with the base table, from which multiple value versions of key k are co-located at the same place and version freshness can be easily known. Algorithm 2 illustrates the evaluation algorithm for ReadValue(v,ts,m): It first issues a Get call to the index table and reads the related index entries before timestamp ts. For each returned index entry, say ts', it needs to determine whether the entry is fresh under the m-versioning policy. To do so, the algorithm reads the base table by issuing a ReadKey query (which is a simple wrapper of a Get call to the base table), which returns all the latest m versions $\{ts''\}$ before timestamp ts. Depending on whether ts' show up in the list of $\{ts''\}$, the algorithm can then decide that it is fresh or obsolete. Only when the version is fresh, it is then

[3]Since Put is an append-only operation in LKVS, we may use the term "Put-only" and "append-only" interchangeably in this paper.

TABLE I: Algorithms for online writes and reads

Algorithm 1 Write(key k, value v, timestamp ts)

```
1: index.Put($v,k,ts$)
2: base.Put($k,v,ts$)
```

Algorithm 2 ReadValue(value v, timestamp ts, versioning m)

```
1: $\{\langle k,ts'\rangle\} \leftarrow$ index.Get($v,ts$)    $\triangleright$ $ts' \le ts$
2: for $\forall\langle k,ts'\rangle \in \{\langle k,ts'\rangle\}$ do
3:     $\{\langle k,v',ts''\rangle\} \leftarrow$ ReadKey($k$, $ts$, $m$)    $\triangleright$ $ts''$ is earlier than $ts$
4:     if $ts' \in \{\langle k,v',ts''\rangle\}$ then    $\triangleright$ $ts'$ is a fresh version regarding $ts$
5:         result_list.add($\{\langle k,v,ts'\rangle\}$)
6:     else
7:         if $ts' > \min\{\langle k,v',ts''\rangle\}$ then
8:             index.Delete($v,k,ts'$)    $\triangleright$ Cleanup dangling index data
9:         end if
10:    end if
11: end for
12: return result_list
```

added to the final result. If it is found that the index version ts' is not present in the base table, implying the occurrence of a failure, it issues a Delete call to remove the dangling index data.

C. Fault Tolerance

In a cloud environment where machines fail, it is possible that a Write can fail with only one Put completed. To deal with failure, our API has the following semantics.

- A Write is considered to succeed only when both Put operations complete. A read (either ReadKey or ReadValue) will return data that must be written successfully, and will not return any data unsuccessfully written.

Under this semantics, HINDEX can achieve consistent data reads and writes with efficiency. We consider the scenario where the machine issuing a Write call fails. It is a trivial case when the failure occurs either before or after the Write invocation; in this case, nothing inconsistent will be left in the LKVS and this can be guaranteed by the fault-tolerance and atomicity property of the underlying LSM tree. Thus, what is interesting is the case that failure happens between index.Put(v,k,t) and base.Put(k,v,t) in the write path. This case can lead to dangling index rows without corresponding data stored in the base table. This (inconsistent) situation can affect the reads under three circumstances: 1) ReadValue(v), 2) ReadValue(v') where v' is the version right before v and 3) ReadKey(k). For case 1), Algorithm 2 is able to discover the dangling index data (as in Line 7) and would correctly neglect such data to comply with our API semantics. It actually issues an index.Delete$(\langle v,k\rangle)$ to remove the dangling data from being considered by future ReadValue(v) invocations. For case 2), Algorithm 2 will not find any version that overwrites v' in the base table and would return $\langle k,v'\rangle$. For case 3), ReadKey returns $\langle k,v'\rangle$. In all cases, our API semantics holds. It is fairly easy to extend this analysis to multiple failed Write's.

V. Offline HIndex: Batched Index Repair

In HIndex, the index repair process eliminates the obsolete index entries and can keep the index fresh and up-to-date. This section describes the design and implementation of an offline and batched repair mechanism in HIndex.

A. Computation Model and Algorithm

To repair the index table, it is essential to find the obsolete data versions. A data version, say $\langle v', k, ts' \rangle$, is considered to be obsolete when either of the following two conditions is met.

1. There are at least m newer key-value versions of key k that exist in the system.
2. There is at least one newer Delete tombstone[4] of key k that exists in the system.

The process to find the obsolete versions, called index garbage collection, is realized by scanning the base table. Because the base table has the data sorted in the key order, which helps verify the above two conditions. Algorithm 3 illustrates the batched garbage collection algorithm on a data stream that is output from the table scan; the data stream is assumed to have key-value pairs ordered first by key and then by timestamp. The algorithm maintains a queue of size m and emits the version only when it is older than at least m versions of the same key k in the queue and it is repaired by previous repair processes (will be discussed in next paragraph). In the algorithm, it also considers the condition regarding a Delete tombstone; it will emit all the versions before the Delete tombstone marker. Note that our algorithm requires a small memory footprint (i.e. the queue of size m).

Our offline index repair process runs periodically (e.g. on a daily or weekly basis). To avoid duplicated work between multiple rounds of repairs, we require that each run of an index repair process is marked with a timestamp, so that the versions of interest to this run are those with timestamps falling in between the timestamp of this run and that of previous run (i.e. ts_{Last}). Any version that is older than ts_{Last} was repaired before by previous index-repair processes and is not considered in the current run.

B. Compaction-Aware System Design

To materialize the table scan in the presence of an offline compaction process, one can generally have three design options, that is, to run the index repair 1) before the compaction, 2) after the compaction or 3) coupled inside the compaction. In HIndex, we adopt the last two options (i.e. to couple the index repair either after or within the compaction). Recall that in an LKVS, the read performance is significantly improved after the compaction. The rationale of such design choice is that the table scan, being essentially a batch of reads, has its performance dependent on the compaction: Without the compaction, there would be a number of on-disk files and a key-ordered scan would essentially become a batch of random reads that make the disk heads swing between the multiple on-disk files.

Overall, the offline HIndex runs in three stages; as illustrated in Figure 4, it runs the offline compaction, garbage collection and index garbage deletion. After a Compact call is issued, the system runs the compaction routine, which then triggers the execution of index garbage collection and deletion (for the index repair). Specifically, the garbage collection emits the obsolete data versions to the next-stage garbage deletion process. The index garbage deletion issues a batch of deletion requests to the distributed index table. In the following, we describe our subsystems for each stage and discuss the design options.

[4]In an LKVS, a Delete operation appends a tombstone marker in the store without physically deleting the data.

Algorithm 3 BatchedGC(Table-scan stream s, versioning m, ts_{Last})

```
1:  $k_{\text{prev}} \leftarrow$ NULL                                  ▷ $k_{\text{prev}}$ is previous key in the stream
2:  $q_d \leftarrow$ NULL   ▷ $q_d$ maintains potential key-value pairs for deletion
3:  for $\forall \langle k, v, ts \rangle \in s$ do          ▷ Stream data sorted by key and in
    time-ascending order (i.e. from past to now)
4:      if $k_{\text{prev}} == k$ then
5:          if $q_d$.size() $< m$ then
6:              $q_d$.enqueueToHead($\langle k, v, ts \rangle$)
7:          else if $q_d$.size == $m$ then
8:              $q_d$.enqueueToHead($\langle k, v, ts \rangle$)
9:              $\langle k, v', ts' \rangle \leftarrow q_d$.dequeueFromTail() ▷ All pairs in $q_d$ are
    of same key $k_{\text{prev}} = k$
10:             if $ts' \geq ts_{\text{Last}}$ then          ▷ $ts'$ is no older than $ts_{\text{Last}}$
11:                 emitToIndexDel($\langle k, v', ts' \rangle$)        ▷ emit the data to
    index-deletion stage
12:             end if
13:         end if
14:     else
15:         loop $q_d$.size()$> 0$                ▷ Clear $q_d$ for index deletion
16:             $\langle k, v', ts' \rangle \leftarrow q_d$.dequeueFromHead()
17:             if $\langle k, v', ts' \rangle$ is a Delete tombstone then
18:                 emitToIndexDel($q_d$.dequeueAll())
19:             end if
20:         end loop
21:         $k_{\text{prev}} \leftarrow k$
22:         $q_d$.enqueueToHead($\langle k, v, ts \rangle$)
23:     end if
24: end for
```

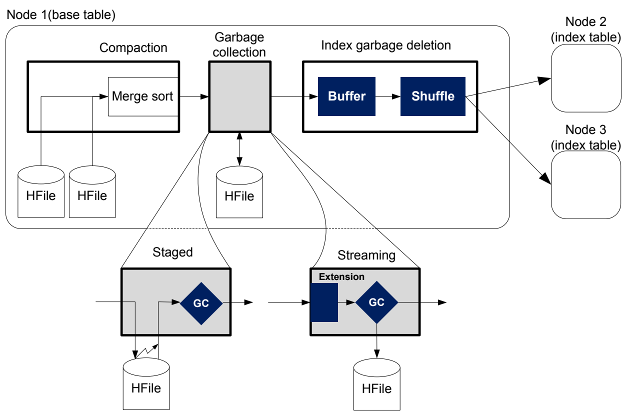


Fig. 4: Compaction-aware index repair

1) The Garbage Collection: We present two system designs for garbage collection, including a staged design that puts the garbage collection right after the compaction process, and a streaming design that couples the garbage collection inside the compaction process.

A staged design: The garbage collection subsystem is materialized as a staged component that runs after the previous compaction completes. As portrayed in Figure 4, the system monitors the number of sorted data files in the local machine. When an offline compaction process finishes, it reduces the number of on-disk files to one, upon which the monitor component triggers the garbage collection process. In this case, the garbage collection reloads the newly generated file to memory (and the file system cache may still be hot), scan it, and run Algorithm 3 to collect the obsolete data versions.

A streaming design: Alternatively, the garbage collection subsystem can be implemented by streaming the compaction's output stream directly to the garbage collection service. To be specific, as shown in Figure 4, the streaming garbage collection intercepts the output stream from the compaction while the data is still in memory; the realization of interception is described below. Then it runs the garbage collection computation in Algorithm 3; if the data versions are found to be obsolete, they are emitted but still being persisted (for fault-tolerance concerns). Comparing the staged design, the streaming design saves disk accesses, since the data stream is directly streaming in memory without being reloaded from disk.

a) Implementation: In terms of implementation, the staged design can be realized as an add-on module to the key-value store system, since its implementation and deployment require nothing more than a generic file-system interface. By contrast, the streaming design may need built-in support from the key-value store in order to hook its garbage collection actions to the compaction data flow.[5] Certain key-value stores expose such programming interface (e.g. CoProcessor [12] in HBase) to allow applications to hook external actions to an internal event.

In particular, we have implemented the staged design for Cassandra, and the streaming design on HBase. For HBase implementation, for hooking up garbage collection, we used the PRECOMPACT API available in HBase's CoProcessor framework. This API allows an application to view a stream of data sorted and merged from multiple old HFiles as being loaded from disk and to customize which key-value records to skip or to be written back to disk. Implementation of both our designs does not require any internal code-base change, and can be lively deployed on the running key-value store.

2) The Index Garbage Deletion: For data emitted from the garbage collection, they enter the stage of index garbage deletion; the goal of this stage is to delete the corresponding obsolete index entries in the index table. Since these index entries could potentially be distributed on any remote machines, the stage needs to issue a number of remote Delete calls. Then the remote index nodes, after receiving the calls, will store the Delete tombstones. After all base-table nodes finish sending the Delete calls, each index-table node will then trigger the index-side Compact which will eventually delete the obsolete index entries, physically.

In this stage, the performance bottleneck is on the sending of remote Delete calls which involve a large number of network messages. To improve network utilization, we propose to "combine" Delete calls to the same destination index node, and pack them into one (instead of multiple) RPC call. The data flow inside the stage of index garbage deletion is illustrated in Figure 4; the incoming data are first buffered in memory and later shuffled before being sent by a Delete call. The shuffle process sorts and combines the data key-value pairs based on the value. In the design of the garbage deletion subsystem, we expose a tunable knob to configure the maximal buffer size; The bigger the buffer is, the more bandwidth efficiency it can achieve at the expense of more memory overhead.

[5]Note that this requirement does not decrease the generality of the HINDEX system design, since it is always possible to modify the code-base of underlying key-value store systems.

C. Fault Tolerance

We consider a faulty networked system underneath the key-value store. The data flow in the offline index-repair process may drop some key-value pairs before the repair action is executed.

To enable and simplify the recovery, we enforce the following requirement in the regular execution path of index-repair process:

- Given a compaction and index-repair process with ts_{Last}, it does not physically delete any data of interest (i.e. with timestamp between now and ts_{Last}).

Note that for any data before ts_{Last}, physical deletion is enabled.

In addition, we assume that the operations in underlying LKVS are idempotent, that is, there are no additional effects if a Put (or Delete) is called more than once with the same parameters. Based on these two properties, we can easily support fault tolerance of the index-repair process. The logic is the following: Given a failed run of index repair, we can simply ignore its partial results and keep the old ts_{Last}. Upon the next run of index repair, the above property guarantees that all data versions from ts_{Last} to the present are still there in the system and the current run, if it succeeds, will eventually repair the index table correctly. Note that since the previously repaired data is not deleted, it may cause some duplicated operations which however do not affect the correctness due to idempotency.

VI. Experiments

This section describes our experimental evaluation of HINDEX. We first did experiments to study the performance characteristics of HBase, a representative LKVS, and then to study HINDEX's performance under various micro-benchmarks and a synthetic benchmark. Before all of these, we describe our experiment system setup.

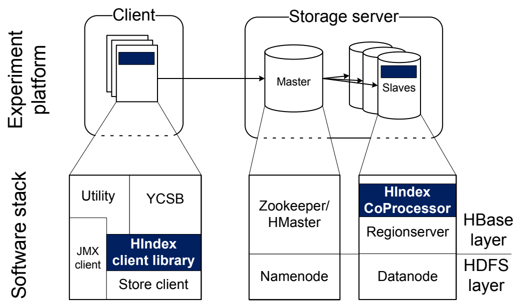


Fig. 5: Experiment platform and HINDEX deployment

A. Experiment System Setup

The experiment system, as illustrated in Figure 5, is organized in a client/server architecture. In the experiment, we use one client node and a 19-node server cluster, consisting of a master and 18 slaves. The client connects to both the master and the slaves. We set up the experiment system by using Emulab [13], [14]; all the experiment nodes in Emulab are homogeneous in the sense that each machine is equipped with the same 2.4 GHz 64-bit Quad Core Xeon processor and a 12 GB RAM. In terms of the software stack, the server cluster uses both Hadoop HBase and HDFS [15]. The HBase and HDFS clusters are co-hosted on the same set of nodes, as shown in Figure 5. Unless otherwise specified, we use the default configuration in the out-of-box HBase. The client side is based on the YCSB framework [16], an industry-standard benchmark tool for evaluating the key-value store performance. The original YCSB framework generates only key-based queries, and for testing our new API, we extended the YCSB to generate value-based queries. We use the modified YCSB framework to drive workload into the server cluster and measure the query performance. In addition, we collect the system profiling metrics (e.g. number of disk reads) through a JMX (Java management extension) client. For each run of experiments, we clean the local file system cache.

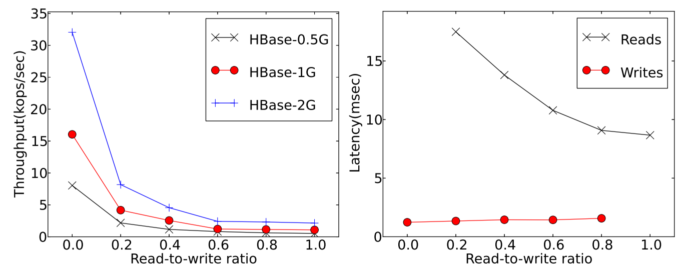
HINDEX *prototype deployment:* We have implemented an HINDEX prototype in Java and on top of HBase 0.94.2. The HINDEX prototype is deployed to our experiment platform in two components; as shown by dark rectangular in Figure 5, the HINDEX middleware has a client-side library for the online operations and a server-side component for the offline index repair. In particular, based on system hooks in HBase, the prototype implements both the staged garbage collection and streaming garbage collection in the server component.

Dataset: Our raw dataset consists of 1 billion key-value pairs, generated by YCSB using its default parameters that simulates the production use of key-value stores inside Yahoo!. In this dataset, data keys are generated in a Zipf distribution and are potentially duplicated, resulting in $20,635,449$ distinct keys. The data values are indexed. The raw dataset is pre-materialized to a set of data files, which are then loaded to the system for each experiment run. For query evaluation, we use 1 million key-value queries, be it either Write, ReadValue or ReadKey. The query keys are randomly chosen from the same raw dataset, either from the data keys or values.

B. Performance Study of HBase

Read-write performance: This set of experiments evaluates the read-write performance in the out-of-box HBase to verify that HBase is aptly used in a write-intensive workload. In the experiment, we set the target throughput high enough to saturate the system. We configure the JVM (on which HBase runs) with different heap sizes. We varied the read-to-write ratio [6] in the workload, and report the maximal sustained throughput in Figure 6a, as well as the latency in Figures 6b. In Figure 6a, as the workload becomes more read intensive, the maximal sustained throughput of HBase decreases, exponentially. For different JVM memory sizes, HBase exhibits the similar behavior. This result shows that HBase is not omnipotent but particularly optimized for write-intensive workloads. Figure 6b depicts the latency respectively for reads and writes (i.e. Get and Put) in HBase. It can be seen that the reads are much slower than writes, by an order of magnitudes. This result matches the system model of LKVS in which reads need to check more than one places on disk and the writes are append-only and fast. In the figure, as the workload becomes more read intensive, the read latency decreases. Because with read-intensive workload, there are fewer writes and thus fewer data versions in the system for a read to check, resulting in faster reads.

[6]In the paper, the read-to-write ratio refers to the percentage of reads in a read-write workload.

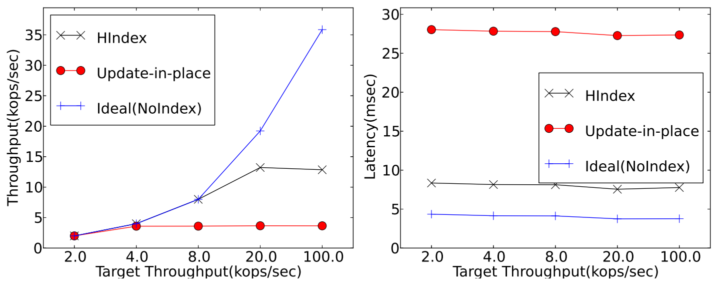


(a) Maximal sustained throughput (b) Latency

Fig. 6: HBase performance under different read ratios

C. HINDEX Performance

Online write performance: This experiment evaluates HINDEX performance under the write-only workloads. We drive the data writes from the client into the HBase server cluster. We compare HINDEX with the update-in-place indexing approach described in Section IV-A. We also consider the ideal case where there is no index structure to maintain. The results of sustained throughput are reported in Figure 7. As the target throughput increases, the update-in-place indexing approach hits the saturation point much earlier than HINDEX. While HINDEX can achieve a maximal throughput at about 14 thousand operations (kops) per second, the update-in-place indexing approach can only sustain at most 4 kops per second. Note that the ideal case without indexing can achieve higher throughput but can not serve the value-based queries. This result leads to a $3\times$ performance speedup of HINDEX. In terms of the latency, Figure 7b illustrates that HINDEX constantly outperforms the update-in-place approach under varied throughput.



(a) Maximal throughput (b) Latency

Fig. 7: Index write performance

Online read-write performance: In this experiment, we evaluate HINDEX's performance in the workload that varies from read-intensive workloads to write-intensive ones. We compare HINDEX on top of HBase against two alternative architectures:

the B-tree index in MySQL and the update-in-place indexing on HBase. For fair comparison, we use the same dataset in both HBase and MySQL, and drive the same workload there. MySQL is accessible to YCSB through a JDBC driver implemented by us, in which we reduce as much as possible the overhead spent in the JDBC layer. The results are shown in Figure 8. With varying read-to-write ratios, HINDEX on HBase is clearly optimized toward write-intensive workload, as can be seen in Figure 8a. On a typical write-intensive setting with 0.1 read-to-write ratio, HINDEX on HBase outperforms the update-in-place index on HBase by a 2.5× or more speedup, and the BTree index in MySQL by 10×. When the workload becomes more read-intensive, HINDEX may become less advantageous. By contrast, the update-in-place approach is more read-optimized and the BTree index in MySQL is inefficient, regardless of workloads. This may be due to that MySQL uses locking intensively for full transaction support, an overkill to our targeted use case. In terms of latency, the HINDEX on HBase has the lowest write latency at the expenses of relatively high read latency due to the extra reads to the base table. By contrast, the update-in-place index has the highest write latency (due to the reads of obsolete versions in the base table) and a low read latency (due to that it only needs to read the index table). Note that in our experiments, we use more write-intensive values for read-to-write ratios (e.g. more ticks in interval $[0, 0.5)$ than in $[0.5, 1.0]$).

Offline index repair performance: This experiment evaluates the performance of offline index repair with compaction. We mainly focus on the approach of compaction-triggered repair in the offline HINDEX; in the experiment we tested two implementations, with staged garbage collection and streaming garbage collection. For comparison, we consider a baseline approach that runs the batch index repair before (rather than after) the compaction (i.e. design option 1) in Section V-B). We also test the ideal case in which an offline compaction runs without any repair operations. During the experiment, we tested two datasets: a single-versioned dataset that is populated with only data insertions so that each key-value pair has one version, and a multi-versioned dataset populated by both data insertions and updates which results in 3 versions on average for each data value. While the multi-versioned data is used to evaluate both garbage collection and deletion during the index repair, the single-versioned dataset is mainly used to evaluate the garbage collection, since there are no obsoleted versions to delete. In the experiment, we have configured the buffer size to be big enough to accommodate all obsolete data in memory. [7] We issued an offline Compact call in each experiment, which automatically triggers the batch index repair process. Until the end, we collect the system profiling information. In particular, we collect two metrics, the execution time and the total number of disk block reads. Both metrics are emitted by the HBase's native profiling subsystem, and we implemented a JMX client to capture those values.

We run the experiment three times, and report the average results in Figure 9. The execution time is reported in Figure 9a. In general the execution time with multi-versioned dataset is much longer than that with the single-versioned dataset, because of the extra need for the index deletion. Among the four approaches, the baseline is the most costly because it loads the data twice and from the not-yet-merged small data files, implying that disk reads are mostly random accesses. The ideal case incurs the shortest time, as expected. Between the two HINDEX designs, the streaming garbage collection requires shorter time because it only needs to load the on-disk data once. To understand the performance difference, it is interesting to look at the disk read numbers, as shown in Figure 9b. We only show the results with the single-versioned dataset, because disk reads only occur in the garbage collection. The baseline approach incurs a similar number of disk reads to the staged design, because both approaches load the data twice from the disk. Note that the disk reads in the baseline approach are mostly random access while at least half of disk access in the staged HINDEX should be sequential; this leads to differences in their execution time. In Figure 9b, the ideal case has a similar cost to the streaming design, because both approaches load on-disk data once. From the single-versioned results in Figure 9a, it can be seen that their execution time is also very close to each other, due to that the extra garbage collection caused by the HINDEX approach is very lightweight and incurs few in-memory computations.

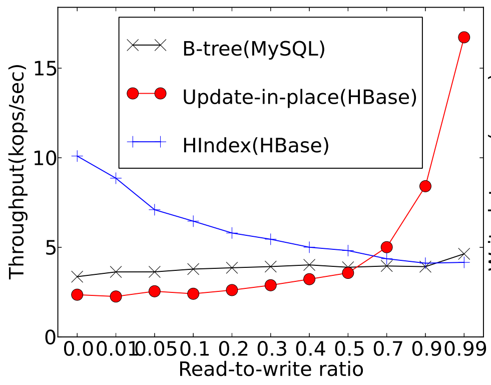
TABLE II: Overhead under Put and Compact operations

Name	Exec. time (sec)	Number of disk reads
HINDEX	**1553.158**	60699
Update-in- place index	4619.456	313662

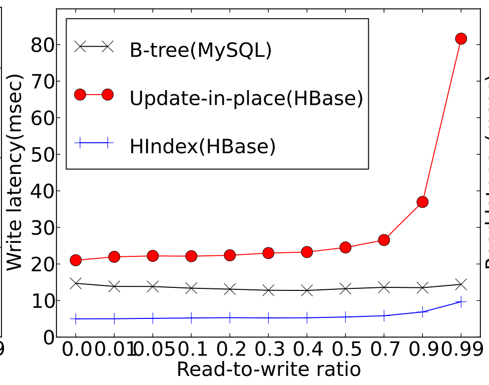
Name	Online	Offline	Online	Offline
HINDEX	**1093.832**	459.326	0	60699
Update-in- place index	4340.277	**279.179**	252964	60698

Mixed online and offline operations: In this experiment, we compare HINDEX and the update-in-place indexing approach as a whole package. In other words, we consider the online and offline operations together. Because the update-in-place approach already repairs the index in the online phase, there is no need to perform index repair in the offline time. For fair comparison, we run the offline compaction (without any repair actions) for the update-in-place index. In the experiment, the online workload contains a series of writes and the offline workload simply issues a Compact call and if any, the batch index repair. For simplicity, we here only report the results of streaming HINDEX. We report the execution time and the number of disk reads. The results are presented in Table II. In general, HINDEX incurs much shorter execution time and fewer disk reads than the update-in-place approach. For example, the execution time of HINDEX (in bold text in the table) is one third of that of the update-in-place approach. We break down the results to the online costs and offline costs, as in the bottom half of the table, which clearly shows the advantage of having the index repair deferred to the offline phase in HINDEX. Although the update-in-place index wins slightly in terms of the offline compaction (see the bold text "279.179" compared to "459.326" in the table), HINDEX wins big in the online computation phase (see bold text "1093.832" compared to "4340.277" in the table). It leads to overall performance gain of HINDEX. In terms of disk reads, it is noteworthy that HINDEX incurs zero costs in the online phase.

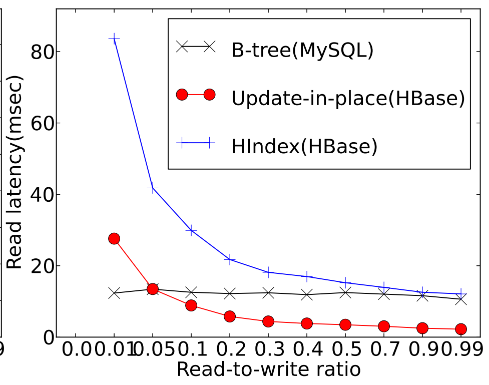
[7]We try to set up our experiment to be more bounded by local disk accesses than by the network communications, so that the offline index repair process is dominated by the garbage collection process than the deletion process.



(a) Maximal throughput

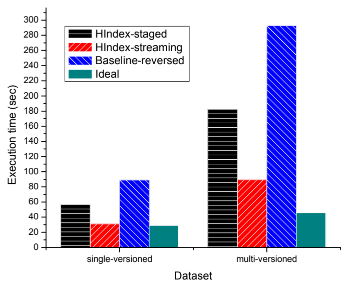


(b) Write latency

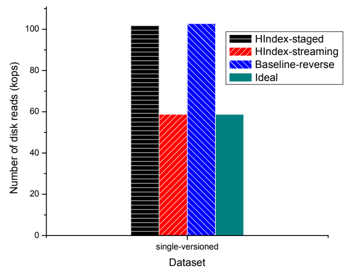


(c) Read latency

Fig. 8: Performance comparison between HIndex in HBase and MySQL



(a) Execution time



(b) Number of disk reads

Fig. 9: Performance of offline index repair

VII. Related Work

Log-structured systems and performance optimizations: Log-structured systems have been studied for more than two decades in the system community. The existing work largely falls under two categories, the unsorted LFS-like systems [17] and sorted LSM tree-like systems. While the former maintains a global log file in which data is appended purely by the write time, the latter organizes the data layout to a number of spill files, in each of which data is sorted based on the key. Log-structured systems generally rely on a garbage collection mechanism to reclaim disk space and/or re-organize the data layout. In particular, several heuristic-based garbage collection policies [18], [19] are proposed and adopted in LFS systems.

Recently, due to the burst of write-intensive workloads, log-structured design has been explored in the context of big data systems in the cloud. In addition to various LKVS systems, bLSM [20] aims at improving the read performance on log-structured stores; the idea is to decompose the cumbersome compaction process so that it can be run at fine granularity with costs being piggybacked with other concurrently running operations. Several key-value stores adopt the unsorted LFS design. Based on a farm of Flash/SSD storage nodes, FAWN [21] avoids the costly in-place writes on SSD by a sequential log and maintains an in-memory index that is updated in place and can speed up the random reads.

Optimizing the performance for LKVS can be divided into two aspects: the scalability/elasticity for cross-node communication efficiency, and the per-node performance. While proposed optimizations apply for elasticity aspects[22], [23], the essence of our HIndex work is to optimize the per-node IO performance in the context of secondary indexes layered over LKVS.

Secondary indexes on key-value stores: Recently, a large body of academic work [10], [24], [11] and industrial projects [25], [26], [27], [28], [29], [30], [31] emerge to build *secondary indexes* as middleware on scalable key-value store systems. Those systems can be largely categorized by their design choices in terms of: 1) whether the index is local or global, 2) how the index is maintained, and 3) the system implementation. Regarding choice 2), the index can be maintained synchronously, asynchronously, or in a hybrid way. Synchronous index maintenance indicates real-time query result availability at the expense of extra index update overhead. Asynchronous index maintenance means the whole index updates are deferred. The hybrid approach is essentially the append-only design (as in HIndex) in which only the expensive part of index maintenance is deferred. In terms of implementation, the index middleware can be in the client or server side, depending on the preference on system generality or performance. Our prior work [32] addresses flexible consistency in context of HBase indexing, but doesn't consider the asymmetric read-write performance in LKVS. We summarize these key-value store indexes in Table III.

In particular, Megastore [28] is Google's effort to support the cloud-scale database on the BigTable storage [1]. Megastore supports secondary indexes at two levels, namely the local index and global index. The local index considers the data from an "entity group" of machines that are close by. When the entity group is small, the local index is maintained synchronously at low overhead. The global index which spans cross multiple groups is maintained asynchronously in a lazy manner. Phoenix [26] is a SQL layer on top of HBase. Its secondary index is global and it considers two types of indices, a mutable index where base data updates overwrite previous versions and an immutable index (e.g. time series data) where data updates are append-only (semantically). While the immutable data index is easily maintained by a client (since an index entry never need to be deleted), the mutable data indexing needs to delete previous versions overwritten. It addresses the consistency issues when the data writes come to local nodes out of order, which may

TABLE III: Key-value indexing systems (– means uncertain and * means HINDEX is implemented on HBase and Cassandra.)

References	Local/Global	Index Mntn	Impl.
Phoenix [26]	Global	Hybrid	HBase-Client/Server
HyperTable Idx [31]	Global	–	HyperTable-Client
Huawei's Index [27]	Local	Sync	HBase-Server
Cassandra Index [29]	Local	–	Cassandra-Server
Megastore [28]	Local/global	Sync/Async	BigTable-Client
F1 [30]	Global	Sync	Spanner[33]-Client
PIQL [10], [34]	Global	Sync	SCADR [35]-Client
HINDEX	Global	Hybrid	General*-Client/Server

make it delete a wrong version.

While existing literature considers the append-only index maintenance (e.g. Phoenix [26]), it does not address the problem of scheduling expensive index-repair operations, the lack of which may lead to eventual index inconsistency and cause unnecessary cross-table check for query processing. By contrast, the HINDEX design is aware of the asymmetric performance characteristic in an LKVS and optimizes the execution of index repairs accordingly.

VIII. Conclusion

This paper describes HINDEX, a lightweight real-time indexing framework for generic log-structured key-value stores. The core design in HINDEX is to perform the *append-only* online indexing and compaction-triggered offline indexing. By this way, the online index update does not need to look into historic data for in-place updates, but rather appends a new version, which substantially facilitates the execution. To fix the obsolete index entries, HINDEX performs an offline batched index repair process. By coupling with the native compaction routine in an LKVS, the batch index repair achieves significant performance improvement by incurring no extra disk accesses. We implemented an HINDEX prototype based on HBase and demonstrate the performance gain by conducting experiments in real-world system setup.

Acknowledgments

The first author would thank Tao Zou for his comments. Ling Liu is partially supported by the National Science Foundation under Grants IIS-0905493, CNS-1115375, IIP-1230740, and a grant from Intel ISTC on Cloud Computing.

References

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data (awarded best paper!)," in *OSDI*, 2006, pp. 205–218.
[2] "http://hbase.apache.org/."
[3] "http://cassandra.apache.org/."
[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *SOSP*, 2007, pp. 205–220.
[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *Operating Systems Review*, 2010.
[6] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.
[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.
[8] M. Yabandeh and D. G. Ferro, "A critique of snapshot isolation," in *EuroSys*, 2012, pp. 155–168.
[9] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI*, 2010, pp. 251–264.
[10] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson, "Piql: Success-tolerant query processing in the cloud," *PVLDB*, vol. 5, no. 3, pp. 181–192, 2011.
[11] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: a distributed, searchable key-value store," in *SIGCOMM*, 2012, pp. 25–36.
[12] "https://blogs.apache.org/hbase/entry/coprocessor_introduction."
[13] "http://www.emulab.net/."
[14] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *OSDI*, 2002.
[15] "http://hadoop.apache.org/."
[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010, pp. 143–154.
[17] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *SOSP*, 1991, pp. 1–15.
[18] T. Blackwell, J. Harris, and M. I. Seltzer, "Heuristic cleaning algorithms in log-structured file systems," in *USENIX Winter*, 1995.
[19] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An implementation of a log-structured file system for unix," in *USENIX Winter*, 1993, pp. 307–326.
[20] R. Sears and R. Ramakrishnan, "blsm: a general purpose log structured merge tree," in *SIGMOD Conference*, 2012.
[21] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," in *SOSP*, 2009, pp. 1–14.
[22] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, elastic resource provisioning for nosql clusters using TIRAMOLA," in *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16, 2013*, 2013, pp. 34–41. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/CCGrid.2013.45
[23] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça, "Met: workload aware elasticity for nosql," in *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, 2013, pp. 183–196. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465370
[24] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan, "Asynchronous view maintenance for vlsd databases," in *SIGMOD Conference*, 2009, pp. 179–192.
[25] L. George, *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly, 2011.
[26] "http://phoenix.apache.org/secondary_indexing.html."
[27] "https://github.com/huawei-hadoop/hindex."
[28] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, 2011, pp. 223–234.
[29] "http://www.datastax.com/docs/1.1/ddl/indexes."
[30] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. O. K. Littlefield, D. Menestrina, S. E. J. Cieslewicz, I. Rae *et al.*, "F1: A distributed sql database that scales," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, 2013.
[31] "http://hypertable.com/blog/secondary_indices_have_arrived."
[32] W. Tan, S. Tata, Y. Tang, and L. L. Fong, "Diff-index: Differentiated index in distributed log-structured data stores," in *Proc. 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, March 24-28, 2014.*, 2014, pp. 700–711. [Online]. Available: http://dx.doi.org/10.5441/002/edbt.2014.76
[33] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally-distributed database."
[34] M. Armbrust, N. Lanham, S. Tu, A. Fox, M. J. Franklin, and D. A. Patterson, "The case for piql: a performance insightful query language," in *SoCC*, 2010, pp. 131–136.
[35] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh, "Scads: Scale-independent storage for social computing applications," in *CIDR*, 2009.