Diff-Index: Differentiated Index in Distributed Log-Structured Data Stores

Wei Tan
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA
wtan@us.ibm.com

Sandeep Tata*
Google, Inc
Mountain View, CA 94043, USA
tata@google.com

Yuzhe Tang†
Georgia Institute of Technology
Atlanta, GA 30332, USA
yztang@gatech.edu

Liana Fong
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA
llfong@us.ibm.com

ABSTRACT

Log-Structured-Merge (LSM) Tree gains much attention recently because of its superior performance in write-intensive workloads. LSM Tree uses an append-only structure in memory to achieve low write latency; at memory capacity, in-memory data are flushed to other storage media (e.g. disk). Consequently, read access is slower comparing to write. These specific features of LSM, including no in-place update and asymmetric read/write performance raise unique challenges in index maintenance for LSM. The structural difference between LSM and B-Tree also prevents mature B-Tree based approaches from being directly applied. To address the issues of index maintenance for LSM, we propose *Diff-Index* to support a spectrum of index maintenance schemes to suit different objectives in index consistency and performance. The schemes consist of *sync-full*, *sync-insert*, *async-simple* and *async-session*. Experiments on our HBase implementation quantitatively demonstrate that *Diff-Index* offers various performance/consistency balance and satisfactory scalability while avoiding global coordination. *Sync-insert* and *async-simple* can reduce 60%-80% of the overall index update latency when compared to the baseline *sync-full*; *async-simple* can achieve superior index update performance with an acceptable inconsistency. Diff-Index exploits LSM features such as versioning and the flush-compact process to achieve goals of concurrency control and failure recovery with low complexity and overhead. Diff-Index is included in IBM InfoSphere BigInsights, an IBM big data offering.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases*

General Terms

Design, Experimentation, Performance

Keywords

log-structured merge (LSM) tree, secondary index, CAP theorem, distributed databases, failure recovery

1. INTRODUCTION

In recent years, scale-out data stores, popularly referred as *NoSQL* [28], are rapidly gaining attention to support Internet-scale applications. Examples are BigTable [7], Dynamo [14], PNUTS [11], Espresso [23], Cassandra [2] and HBase [3]. These stores provide limited operations (mainly CRUD) compared to relational databases, and excels in elasticity on commodity hardware. Such stores are widely used by applications that perform CRUD operations while needing scalability.

As applications built on NoSQL stores grow, the need for database features such as secondary indexes becomes important. Consider a social review application (similar to yelp.com) tracking reviews posted by users about restaurants, bars, clubs, etc. A common query is to list all reviews by a certain criteria, e.g., of a particular restaurant, or by a particular user. The schema in Figure 1 shows three tables: *Reviews*, *Users*, and *Products*. In a NoSQL store, tables are usually partitioned across a cluster of servers. A given table can only be partitioned using one attribute, say by *ReviewID* for *Reviews*. As a result, to efficiently answer queries such as "find all reviews for a given restaurant" or "find all re-

*Work done while author was at IBM Almaden Research Center.

†Work done while author was an intern at IBM T. J. Watson Research Center.

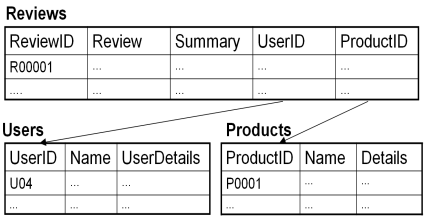

Figure 1: Review Schema.

Table 1: LSM tree vs. B-tree

Features	LSM	B-Tree
Optimized for	write	moderate read and write
Write	append-only, fast	in-place, slower
Write API	put for both insert and delete	insert and delete
Read	relatively slow	relatively fast
Usage	BigTable, HBase, Cassandra, etc	many RDBMS

views written by a given user", we need secondary indexes on *Reviews*.

Challenges of secondary index maintenance in LSM

Internet-scale workloads become more write-intensive with the proliferation of click streams, GPS and mobile devices. Because of this, Log-Structured-Merge (LSM) Tree [22] gains much recent attention. LSM has superior performance in write-intensive workloads because of its log-like data structure. However, index maintenance in a distributed LSM store has not been systematically addressed.

Firstly, existing index approaches on B-Tree (in this paper B-Tree refers to B-Tree, B+ Tree and other B-Tree variants) cannot be straightforwardly applied to LSM due to their structural difference. See a comparison of B-Tree and LSM in Table 1. B-Tree handles updates in an **in-place** manner, and is optimized for moderate read-and-write workloads; LSM generates only sequential I/O in writes, with **no in-place update**, and is optimized for write-intensive workloads. As illustrated later in Section 2, in LSM *inserts* and *updates* are indistinguishable, and are implemented with the same *put* operator; while in B-tree insert and update are two separate operators. This implies that in LSM a *put* does not know if it is actually an *insert* or *update*. As a consequence, the associated index update needs to incur a *read* into the data table, in order to get the old index value and later remove it. That is, a *read* operation is added into the path of a *write* in index maintenance. Different from B-tree, in LSM a read is many times slower than a write. This no in-place update and asymmetric read-write latency, and its implication to index update, needs to be considered and addressed.

Secondly, LSM stores, like other NoSQL systems, are by-design partitioned to handle big data. Therefore, base data and index will be both distributed. How to maintain consistency between base and index, and minimize the maintenance overhead is also challenging.

Thirdly, how to avoid yet another logging system for failure recovery is key to reduce the overhead of index maintenance and improve overall system performance. Moreover, this failure recovery protocol has to be reconciled with the aforementioned index maintenance procedures.

Contribution of Diff-Index

This paper presents *Diff-Index* (Differentiated Index), a novel and systematic approach to add global secondary index on a distributed LSM store, without compromising its write latency and scalability. To the best of our knowledge, we are the first to build effective indexes in LSM taking into account its intrinsic features: no in-place update and versioned record, read is much slower than write, and distributed nature.

Diff-Index provides a spectrum of *synchronous* and *asynchronous* index maintenance schemes, and a carefully designed, lightweight concurrency control and failure recovery protocol. Our contribution is as follows:

- We investigate the challenge of secondary index maintenance in distributed LSM stores, addressing its no in-place update feature, asymmetric read/write performance and distributed nature.

- We design *Diff-Index* comprising four index update schemes, and quantitatively analyze the performance of them.
  1) *sync-full*: to complete all index update tasks synchronously;
  2) *sync-insert*: to insert new index synchronously but lazily-repair old index entries;
  3) *async-simple*: to asynchronously execute index updates and guarantee eventual execution;
  4) *async-session*: to achieve read-your-write semantics on top of *async-simple*.

- We design the concurrency control and failure recovery protocol by exploiting LSM feature such as versioning and flushing, so as to minimize additional complexity on top of LSM.

- We implement Diff-Index on HBase. Experiment results show that Diff-Index offers a variety of performance/consistency trade-offs, and scale well without introducing any global locking or coordination.

Diff-Index is now a feature enhancement of HBase in *IBM InfoSphere BigInsights* V2.1 [19], an IBM big data product built on top of the open-source Hadoop ecosystem. Diff-Index enables BigInsights to create, maintain and utilize index for data stored in HBase.

The rest of the paper is organized as follows. Section 2 introduces LSM and HBase. Section 3 discusses consistency of index and motivates the design of Diff-Index. Sections 4 and 5 present synchronous and asynchronous index schemes in Diff-Index, respectively. Section 6 discusses the IO cost and ACID features. Section 7 introduces the implementation in IBM BigInsights. Section 8 presents comprehensive experiments that explore the latency, throughput and consistency of Diff-Index under different settings of the extended YCSB workload. We describe related work in Section 9 and conclude in Section 10.

2. BACKGROUND

This section lays out the problem domain that motivates the design of Diff-Index. Section 2.1 introduces the LSM model; Section 2.2 introduces HBase as an LSM implementation.

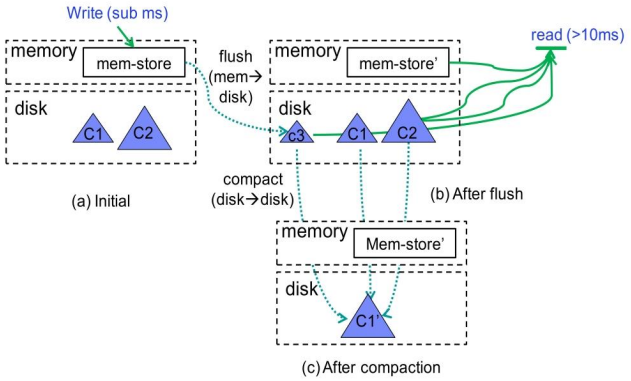

Figure 2: Log Structured Merge tree.

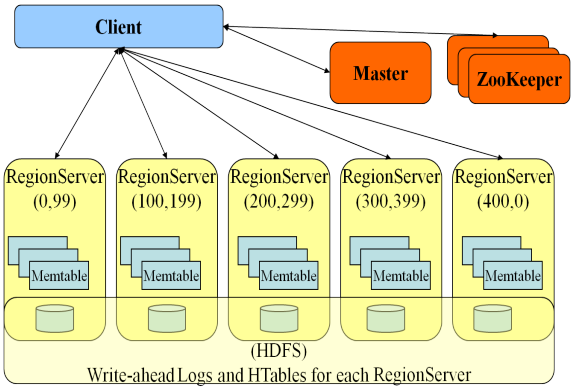

Figure 3: HBase architecture.

2.1 LSM-Tree Model

LSM model was introduced in 1996 [22] and received a reviving interest after Google released BigTable [7]. It prevails in workloads with a high rate of inserts and deletes. The LSM-tree defers and batches data changes by cascading them from a memory to disk. A LSM-Tree consists of: an in memory store using tree or map structure, and several immutable disk stores using B-tree or variants. For example, the LSM tree in Figure 2(a) consists of an in memory store (block *mem-store*) and two disk stores *C1* and *C2*. Writing into LSM equals an insertion into the mem-store. In practice a write usually also involves an append to a commit log for durability purpose. Therefore, an LSM write, including a memory operation and a sequential I/O, is very fast. When mem-store reaches to a certain volume, its content is *flushed* to disk. For example, the mem-store in Figure 2(a) is flushed into a new disk store *C3* in Figure 2(b). After the flush the the mem-store becomes empty and denoted as *mem-store'* in Figure 2(b). With this procedure, every write is virtually an append; an update or deletion to an existing record is achieved by adding a new version (or a tombstone, in case of deletion) into the mem-store. A newer version is usually marked by a monotonic timestamp. By this means LSM has no in-place update and keeps multiple versions of a record. To retrieve one or multiple versions of a record, the mem-store and all disk stores need to be scanned, as shown in Figure 2(b). Therefore a read may include multiple random I/O and is relatively slow. To alleviate this and consolidate multi-versions of a record into a single place, disk stores are periodically *compacted*. As an example, *C1*, *C2* and *C3* are compacted into *C1'*, as shown in Figure 2(c).

In Section 4.1 we will revisit these two features of LSM, i.e., no in-place update and asymmetric read/write latency, and demonstrate that they impose unique challenges in index maintenance.

2.2 HBase Data Model and Architecture

Apache HBase is a LSM store mimicking BigTable. Since our implementation is based on HBase, we briefly introduce its architecture and API. In HBase, data is organized in *tables*. A table contains *rows* that are identified by a (primary) *row key*. Each row may contain an arbitrary number of named *columns* with corresponding values and *timestamps*. Columns are grouped into column families. Each column family is partitioned and stored on multiple nodes, and on each node it is stored as a LSM-tree.

We list a subset of HBase's API:

put(table, key, colname, colvalue): Insert a column value into a row.

get(table, key, colname): Read a column value and its timestamp from a row.

delete(table, key, colname): Delete a column from a row.

Although not shown, there are also multi-column versions of these methods. For example, the multi-column version of Put allows many columns in one row to be updated with a single call. Most APIs can also specify a timestamp to either put a row with a timestamp or get with a given timestamp. Without losing generality, we consider a single column-family table in this paper.

As an example in Figure 3, in each each table, the key space is partitioned into *Regions* and each region is assigned to a *RegionServer*. The RegionServer is responsible for serving puts and gets for the keys that fall in the Region. A given RegionServer may serve more than one Regions. The client library caches a copy of the partition map and can route a request for a given key to the appropriate RegionServer. HBase *Master* is the management node dealing with tasks such as table creation and destroy. *ZooKeeper* [18] is the cluster management node dealing with region assignment, node failure, etc.

When a put request arrives at a region server, it first assigns a timestamp to the put, and then writes this request to the Write-ahead-Log (WAL). This timestamp is local to the RegionServer and is a monotonically non-decreasing long integer generated by System.currentTimeMillis() call in Java. The WAL is a file on HDFS and it can guarantee durability. Once the put has been logged, the RegionServer applies the update to the *Memtable* (Memtable is the HBase term of the aforementioned LSM mem-store). Periodically, the contents of the Memtable are flushed to the on-disk LSM component called *HTable*. HTables also reside in HDFS and are therefore replicated and durable.

3. MOTIVATION OF DIFF-INDEX

3.1 Global vs. Local Index

In a distributed and partitioned data store, there are two types of secondary indexes: *local* and *global*. As evident from the name, a local index indexes data in a given re-

gion and co-locates with that particular region. In contrast, a global index indexes all regions in a table, and is potentially itself partitioned across all nodes. The advantage of a global index is in the handling of highly selective queries, i.e., those with a small result set. This is because a global index has an overall knowledge of data location, and allows sending queries to regions that actually contain the required data. Its drawback is that the update of a global index incurs remote calls and therefore results in a longer latency, as data and index are not necessarily collocated. On the other hand, a local index has the advantage of faster update because of its collocation with a data region; its drawback is that every query has to be broadcast to each region, and therefore costly especially for highly selective queries. Both local and global indexes can be valuable in different settings [9]. In summary, we focus on **global indexes** to better support selective queries on big data. This selection leads to a distributed index which is subject to CAP theorem.

3.2 Consistency of Index

CAP theorem [6] indicates that any networked and replicated data store can enjoy two of the three properties: *consistency*, *availability* and network *partition-tolerance*. In a distributed data store, partition-tolerance is always desirable and this indicates a balance has to be made between consistency and availability. In a lot of circumstances availability means latency of a request, so latency and consistency needs to be balanced, subject to CAP theorem.

From the perspective of CAP theorem, index can be considered as a replication of the base data it indexes. Full ACID guarantee on index updates is not always desirable in Internet-scale, write-intensive workloads because of the following reasons:
1. All-or-nothing semantics restricts the availability of the system. In some cases when index cannot be synchronized, users still want the work to proceed.
2. Protocols such as two-phase commit can provide transaction guarantee to index updates; however in a distributed environment this increases latency.
3. Many workloads are write intensive and read from time to time, such that index update can be delayed. Also, sometimes it is acceptable that the index as a replication does not catch up with base data.

3.3 Session Consistency

Session consistent read offers a read-your-write semantics while still getting most of latency and throughput benefits of asynchronously maintained indexes. We use the social review application in Figure 1 to explain it. Consider two different users taking the following sequence of operations:

User 1	User 2
1.View reviews for product A.	View reviews for product B
2.Post review for product A.	
3.View reviews for Product A.	View reviews for Product A

User 1 views the reviews for product A while User 2 browses the reviews for a different product B. Next, User 1 posts a review for product A, and asks to view all reviews for product A – possibly to see if his review gets listed correctly. If this query is served by looking up an index on column ProductID of table Reviews, and if this index is maintained asynchronously, it is possible that User 1, at time=3 does not see his own review when listing all the reviews for product A, because the index has not yet been updated yet. This is an undesirable effect, and may lead to User 1 assuming that his review was lost and resubmitting it – later to find his review getting listed twice.

Under session consistency, User 1 would be guaranteed to see his review at time=3 when he lists all reviews for product A. However, User 2 is not guaranteed to see User 1's review of product A when his query arrived at time=3, after the review had been posted. For Web 2.0 applications like this, such a notion of consistency is adequate and can be attractive if it provides better latency and throughput guarantees. Section 5.2 describes the design and implementation of session consistency in Diff-Index.

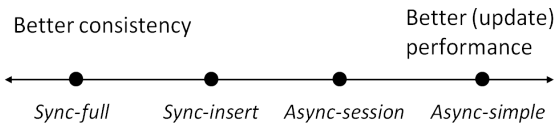

Figure 4: The spectrum of Diff-Index schemes.

3.4 Configurable Consistency in Diff-Index

Compared to solutions that can guarantee a strongly consistent index [13], Diff-Index is desirable for workloads requiring low latency, high availability and with intensive writes. Diff-Index provides consistency levels as follows (see Figure 4):

- *Causal consistent*: once a put operation returns SUCCESS to the client, both data and its associated index entry are persisted in the data store. This is the strongest consistency level in Diff-Index and achieved by the *sync-full* scheme to be introduced in Section 4.1.

- *Causal consistent with read-repair*: index is causal consistent when client check both index and base table. This is achieved by scheme *sync-insert* to be introduced in Section 4.2.

- *Eventually consistent*: when a data insert returns SUCCESS to the client, its associated index entry is not updated right away but will eventually be. This is the weakest consistency level in Diff-Index and achieved by scheme *async-simple* in Section 5.1.

- *Session consistent*: on top of *async-simple*, a client session is aware of its own actions, i.e., read-your-write semantics is enforced. This is achieved by scheme *async-session* in Section 5.2.

In our design, schemes can be chosen in a per index manner, i.e., each index can choose one from the four schemes. Users can choose which Diff-Index schemes to use depending on the workload and consistency requirements. Here are some general principles:
(1) use sync-full or sync-insert when consistency is needed;
(2) use sync-full when read latency is critical;
(3) use sync-insert when update latency is critical;
(4) use async-simple or sync-session when consistency is not a concern;
(5) use async-session when read-your-write semantics is needed.

Ideally Diff-Index should be able to adaptively choose a scheme by understanding consistency requirements and observing workload characteristics such as read/write ratio. Currently user selection is required and we leave adaptive scheme selection for future work.

4. SYNCHRONOUS SCHEMES

Before introducing Diff-Index schemes, we go through the operations incurred in an index update accompanying a base data update. We use update/write/insert/put interchangeably given the nature of LSM. Now we introduce notations of key-value pairs and operations in base and index tables.
A record, row, or key/value pair: $\langle k, v, ts\rangle$ in which k is the key, v is the value, and ts is the timestamp. We omit ts when it has no particular meaning in the context.
LSM Tables: base or data table (B), and index table (I);
Operations on LSM: write–*put*() (P), read–*get*() (R), delete–*delete*() (D);
Operations on base table: *put*()–P_B, *get*()–R_B, and *delete*()–D_B;
Operations on index table: *put*()–P_I, *get*()–R_I, and *delete*()–D_I.

Remark: Here we use notions that are generally applicable to any LSM store. Readers can easily relate these notions to HBase terminology. For example, in base table, the key of a Key/Value pair is HBase's rowkey plus column name. In Diff-Index we make the index table a key-only one, i.e., an index row uses the concatenation of the index_value and rowkey of the base entry as its rowkey, with a null value.

4.1 Scheme Sync-full

Sync-full consists of steps shown in Algorithm 1: **SU1**– put in base table, **SU2**– put in index table, **SU3**– read in base table, and **SU4**– delete in index table. Since **SU1** is not introduced by index update so we formulate the latency (L) of index update scheme *syn-full* as follows:

$$L(sync\text{-}full) = L(P_I) + L(R_B) + L(D_I) \quad (1)$$

Using sync-full, an index is up-to-date after Algorithm 1 returns successfully. As we discussed in Section 2.1, one feature distinguishing LSM from B-tree is that, $L(R_B)$ is much larger than $L(P_I)$ and $L(D_I)$. Therefore, $L(R_B)$ is the major contributor of $L(sync\text{-}full)$. As a result, adding index introduces a large latency $L(sync\text{-}full)$ on top of $L(P_B)$. This inspires us seeking alternatives to optimize scheme sync-full and reduce latency.

Algorithm 1 Index update in sync-full

When a put $\langle k, v_{new}, t_{new}\rangle$ to base table is observed, do **SU1** to **SU4**:
SU1. Put data into base table: $P_B(k, v_{new}, t_{new})$;
SU2. Put index into index table: $P_I(v_{new} \oplus k, null, t_{new})$ or $P_I(v_{new} \oplus k, t_{new})$ to omit the null value;
SU3. Read the value of key k before time t_{new}: $v_{old} \leftarrow R_B(k, t_{new} - \delta)$;
SU4. Delete old index from t_{new} from index table: $D_I(v_{old} \oplus k, t_{new} - \delta)$.

Remark: in Algorithm 1, δ stands for a infinite small time unit; in HBase implementation we choose 1 millisecond as it is the smallest time unit. $R_B(k, t_{new} - \delta)$ means read from the base table, the value of row k before timestamp t_{new}.

4.2 Scheme Sync-insert

In scheme sync-full, by running steps **SU1** and **SU2** only and skip steps **SU3** and **SU4**, the major component $L(R_B)$ from the right-hand side of Equation 1 is removed. This scheme, which we call *sync-insert*, shortens index update latency significantly:

$$L(sync\text{-}insert) = L(P_I) \quad (2)$$

Let us look at the implication of this scheme. In a base table, new data $\langle k, v_{new}, t_{new}\rangle$ invalidates $\langle k, v_{old}, t_{old}\rangle$ for any $t_{old} < t_{new}$. This is caused by LSM's semantics: for a given key, a value with a more recent timestamp overwrites any value with an older timestamp. However, in index table new index entry $\langle v_{new} \oplus k, t_{new}\rangle$ and the old one $\langle v_{old} \oplus k, t_{old}\rangle$ co-exist, because they are with different keys, i.e., $v_{old} \oplus k$ and $v_{new} \oplus k$, respectively. So this scheme leaves stale entries in index table. This is acceptable until those entries are referenced by a query. Therefore to get accurate result a double-check routine has to be enforced during querying. As seen in Algorithm 2, sync-insert requires an additional step **SR2** during index read, compared to sync-full. Actually this additional read-and-delete step is saved earlier in the index update process of sync-insert.

Algorithm 2 Index read in sync-insert

Input: Index value v_{index}.
Output: List of rowkeys in base table K.
$K \leftarrow \emptyset$
SR1: Read index table and get a list of rowkeys with timestamps: $List(\langle k, ts\rangle) \leftarrow R_I(v_{index})$
SR2:
for all $\langle k, ts\rangle$ in $List(\langle k, ts\rangle)$ **do**
  read base table and get newest value of k with timestamp: $\langle v_{base}, tb\rangle \leftarrow R_B(k)$
  if $v_{index} == v_{base}$ **then**
    $\langle v_{index} \oplus k\rangle$ is an up-to-date index entry and add k to K
  else
    $\langle v_{index} \oplus k\rangle$ is stale and delete $\langle v_{index} \oplus k, ts\rangle$ from index table: $D_I(v_{index} \oplus k, ts)$
  end if
end for
return K

4.3 Concurrency Control

Concurrency control is important for a distributed LSM store, and is orthogonal to the underlying data structure. Different concurrency control approaches can be built on the same abstract LSM model. For a couple of examples: HBase uses MVCC in each region, writes are sequenced in a region and reads are non-blocking by following snapshot isolation; MegaStore (an enhancement of BigTable) [4] uses a single timestamp-oracle across regions, and lock-based two-phase commit for distributed transactions. Index maintenance protocols have an impact on the correctness of concurrent access. In both sync and async schemes, as seen in Algorithms 1 and 4, we enforce that **an index entry must have the same timestamp as the base entry it associated with**. We enforce this rule for easy concurrency

control and failure recovery. Here, we discuss the issue of concurrency control and leave the failure recovery issue to Section 5.3

The generated index entry carries the same timestamp as a new incoming base entry, so that in sync-full, **SU4** can safely delete the old index record before timestamp t_{new}. Please note that in both $R_B(k, t_{new} - \delta)$ and $D_I(v_{old} \oplus k, t_{new} - \delta)$, the δ is important. In $R_B(k, t_{new} - \delta)$, you want to read the latest version right before the base put at t_{new}; without the δ you could have read exactly the same put at t_{new}, which is incorrect. In $D_I(v_{old} \oplus k, t_{new} - \delta)$, you want to delete the value right before t_{new}; without the δ you could have deleted the index just put at t_{new}, if $v_{new} == v_{old}$; this is also incorrect.

In LSM *put* is a generic term for both *insert* and *update*, and a data entry having a newer timestamp (version) invalids the previous version. For *delete*, LSM writes a *tombstone* with a timestamp representing the deletion time. In other words, deletion is handled similarly as put in LSM, i.e., deletion can be treated as a put with a null value and a timestamp. Therefore, in the remaining part of this paper we only discuss the case of put and the approach can be extended to handle deletion in a straightforward way.

5. ASYNCHRONOUS SCHEMES

In the previous section we already see that, the index update latency of sync-full is high since it requires synchronously deleting the old index entry, which in turn requires a read of the previous version of the value in the base table. The sync-insert scheme offers a lower latency in exchange for a worse read latency. It may be an attractive tradeoff for write-heavy applications. We next describe another scheme named *async-simple* that offers low latency on both updates and reads by relaxing consistency.

5.1 Scheme Async-simple

The idea behind async-simple is to use an in-memory queue called asynchronous update queue (AUQ) to temporarily store all base updates that require index processing, and acknowledge the client as soon as the base update has been logged and added into AUQ. Index processing is done by a background service called the asynchronous processing service (APS) that de-queues from AUQ, inserts new values into the index table, and deletes corresponding old values if present. The procedure at update-time and during background processing is outlined in Algorithms 3 and 4, respectively.

With the asyc-simple scheme, there will be a time window when a piece of base data has been updated but its index has not been fully updated. If partially updated indexes are accessed by a client, it is possible that an item that was already updated in the base table either still associates with an old index (when neither the new value nor the delete has been delivered), or disappears from the index (when the delete has been delivered but not the new value), or associates with two different index values (when new value has been delivered but not the delete). This is similar to the guarantees provided by PNUTS [1].

5.2 Session Consistency and Scheme Async-session

While some applications can live with eventual consistency, Terry *et al.* [27] argue that sometimes stronger semantics are highly desirable. To avoid the cannot-see-your-own-write abnormality shown in Section 3.3, we propose an approach that provides session consistency – we call it *async-session*. In this scheme, any index look-up is guaranteed to contain updates to the base data that were made in the same session. We next detail the process for session consistency.

Algorithm 3 Index update in async-simple

When a put $\langle k, v_{new}, t_{new} \rangle$ to base table is observed, do **AU1** to **AU2**:
AU1. Do base put $P_B(k, v_{new}, t_{new})$; If an index is defined, add the put to AUQ;
AU2. Acknowledge put SUCCESS to the client.

Algorithm 4 Background index processing in async-simple

BA1. If AUQ is not empty, dequeue a put $\langle k, v_{new}, t_{new} \rangle$ from it;
BA2. Read from base table the value for k at $t_{new} - \delta$: $v_{old} \leftarrow R_B(k, t_{new} - \delta)$;
BA3. Delete old index from t_{new}: $D_I(v_{old} \oplus k, t_{new} - \delta)$;
BA4. Insert new index $P_I(v_{new} \oplus k, t_{new})$.

Consider the sample interaction below:

```
session s = get_session()
put(s, table, key, colname, colvalue)
getFromIndex(s, table, colname, colvalue)
end_session(s)
```

put() and getFromIndex() behave exactly like their regular counterparts except that they take an extra argument: a session ID `s` that is generated by the get_session() call. The getFromIndex() call guarantees that it sees all updates made to any index by the operations in session `s`. We implemented a session consistent variant to each HBase API described in Section 2.2.

The basic technique used to provide session consistency is to track additional state in the client library, and we implement session consistency in HBase Java client library. In a get_session() call, the client library creates a random session ID, adds an entry in a data structure that tracks live sessions, and returns this ID to the application. For all operations that use the session consistent version of the APIs, the client library maintains a set of client-local, private, in-memory tables associated with the session ID and updates them every time the server returns success for a request. To service session consistent reads, the client library first runs a regular read that gets results from the appropriate RegionServer, but before returning to the application, it merges this read with the contents of the private table in the client library. The intuition behind this approach is to track session-local state that might not be synchronously maintained in indexes. The main challenge with this approach is managing the memory allocated to each session storing the private table subsets.

When a base table is updated, the client library submits this request as a regular call, but also requests that the server returns the old value and the new timestamp assigned to the update. The library uses the old value to generate delete markers for the keys in the secondary index table(s). It also uses the same logic as in the server to generate new entries based on new base records and inserts them into the session-private hash table.

On getting requests, if there is data in session-private data

structure, the results returned by the server get combined with the data in the data structure. The session-private data are garbage collected when a session expires or the client issues an end_session() call.

For implementation, a "client" is not necessarily at an end-user's machine, but usually at the middleware layer, say in an application server which acts as the client of the LSM store. Despite an application server's ample capacity to store session information, a session should not stay and grow indefinitely. We configure a maximum limit for session duration: if a session is inactive longer than this limit (say 30 minutes), then it is destroyed and data garbage collected. An application that issues a request under this session ID after 30 minutes will get a session expiration notification and start a new session. Experiments in Section 8 show that, it is highly unlikely that asynchronous index updates take more than 30 minutes to complete, so such an expiration mechanism makes sense. We also developed a mechanism to monitor the memory usage of a session, and automatically disable session-consistency when out-of-memory is to occur, most likely in update-intensive workloads.

5.3 Failure Recovery of AUQ

As analyzed earlier in this section, schemes async-simple and async-session provide lower latency for both updates and reads, with sacrifice in consistency. However, this asynchronism brings about complication in terms of failure recovery to guarantee eventual consistency. Since the recovery logic of LSM is dependent on both the abstract data model and the specific implementation, we first briefly review the recovery protocol in HBase. We then introduce the design of Diff-Index failure recovery protocol by leveraging the generic LSM features and the recovery protocol of HBase.

Failure recovery of HBase

Recall Figure 3, HBase's failure recovery relies on these facts: (1) data in in-memory MemTables have their write-ahead-logs (WAL) persisted in HDFS, (2) on-disk HTables themselves persist on HDFS, and (3) HDFS is fault-tolerant and accessible by any participant node. When one RegionServer fails, HBase does the following: the ZooKeeper detects the failure using a heart-beat mechanism, retrieves the WAL for that RegionServer from HDFS and splits it into separate logs, i.e., one for each Region. Then, the ZooKeeper re-assigns each Region that used to be hosted by the failed RegionServer to a new RegionServer. Each of these newly assigned `RegionServers` replays the WAL corresponding to the new region to which it has been assigned, to restore the MemTable data. After WAL replay the RegionServer links the MemTable with the earlier-persisted HTables from HDFS, and finally re-open the region for reads and writes. In the event that the HBase master fails, a new node takes over as a new master since all the state is stored in highly-available Zookeeper.

AUQ failure recovery protocol

In Diff-Index, indexes are also stored as LSM tables. When a RegionServer fails, all the base and index tables hosted by this server will be recovered by the specific LSM implementation, such as HBase. When asynchronous schemes are used and the AUQ is not empty, those pending requests in AUQ do not necessarily reach their target regions. While base and index tables are both recovered by HBase, these pending requests in AUQ need to be handled as well. In other words, the recovery protocol needs to guarantee that index update operations in AUQ are also recovered correctly during a region restore. A straightforward solution is to add yet another log for AUQ to ensure its durability, however this adds to the overhead and complexity to the system.

As discussed in the previous paragraph, in HBase data that have been flushed to HTable is persistent; data that are not flushed and still in MemTable are vulnerable but can be rebuilt from WAL in server failure. Here an important milestone to watch out is the *flush* point. During a flush, a MemTable is emptied and data in it is persisted to one or more HTables; in the meanwhile, the WAL corresponding to the flushed/persisted data is deleted (this process is also called WAL roll-forward). We can divide all the pending requests PR in AUQ in two sets, $PR(MemTable)$ and $PR(Flushed)$, based on whether the base data is still in MemTable or already flushed, i.e.,

$$PR = PR(MemTable) \cup PR(Flushed) \tag{3}$$

and,

$$PR(MemTable) \cap PR(Flushed) = \emptyset \tag{4}$$

Data in $PR(MemTable)$ and $PR(Flushed)$ have the following behavior in the region recovery process:
(a) each base put in $PR(MemTable)$ will be replayed during the region recovery;
(b) none of base put in $PR(Flushed)$ will be replayed during the region recovery process, since they are all flushed and persisted in a HTable.

This finer categorization of PR in AUQ inspires us to design an AUQ recovery protocol without adding yet another log, when the following two requirements are satisfied.

(1) ensure that $PR(Flushed) = \emptyset$ at any time, and

(2) further associate the recovery of $PR(MemTable)$ with the recovery of the base MemTable.

The reason of (1) is that, if base puts in a MemTable remain un-processed in the AUQ after the MemTable flushes, they become dangling and not able to be reconstructed from WAL replay, since their corresponding WAL has already been rolled forward after flush. Figure 5 illustrates how to achieve (1): We add a `preFlush()` coprocessor hook to block the AUQ from receiving new entries, and wait until the APS drains the AUQ ("1. pause & drain"). Only after that we do "2. flush" and "3. roll forward" of WAL. With this draining-AUQ-before-flush constraint, WAL acts as the log for both MemTable and AUQ.

During a WAL replay, (2) is achieved by: each base put replayed is also put into AUQ again by Diff-Index, regardless of whether or not it has been delivered to index tables by AUQ before the failure. This ensures delivery of index update but as a side-effect, some puts may enter AUQ and get delivered more than once. This behavior is correct with respect to LSM semantics, because an index entry in Diff-Index is always with the same timestamp of its base entry; while in LSM semantics, adding the same entry with the same timestamp for multiple times, is *idempotent*.

Low performance impact of the recovery protocol

This draining-AUQ-before-flush approach will slightly delay flush when the system is under a heavy write load. We show in Section 8 that in practice, this delay is reasonable. One may also raise concern about the excessive delivery of indexes during a failure, despite its idempotency. We argue

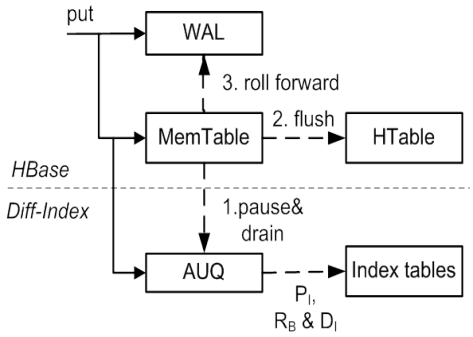

Figure 5: Recovery process of AUQ.

Table 2: I/O cost of Diff-Index schemes

Scheme	Action	Base Put	Base Read	Index Put	Index Read
no-index	update	1	0	0	0
	read	–	–	–	–
sync-full	update	1	1	1+1	0
	read	0	0	0	1
sync-insert	update	1	0	1	0
	read	0	K	K	1
async-simple	update	1	[1]	[1+1]	0
	read	0	0	0	1

that, since failure recovery occurs relatively rarely, the simplicity of this design outweighs the potential excessive (but semantically correct) index update.

In summary, a light-wight failure recovery is accomplished by (1) draining AUQ before LSM flush, (2) ensuring index and base data having the same timestamp, and (3) re-add every base put to AUQ during WAL replay. Through this careful design exploiting features of both generic LSM and HBase implementation, we avoid adding a separate logging and recovery mechanism for index; instead we make index recovery a subroutine in base WAL replay.

6. DIFF-INDEX PROPERTIES

In this section we discuss the I/O cost and ACID properties of Diff-Index schemes.

6.1 I/O Cost

Table 2 summarizes the complexity of Diff-Index schemes, in terms of the read/write operations involved. For each scheme, two actions, i.e., index update and index read, are considered. For both actions, the number of put in base table (Base Put), read in base table (Base Read), put in index table (Index Put) and read in index table (Index Read), are counted, respectively. For example, in the sync-full row, an index update involves 1 Base Put, 1 Base Read , 1 Index Put and (possibly) 1 Index Delete (summarized as "1+1" in column Index Put); in the sync-insert row, an index read involves 1 Index Read, K Base Read (assuming index read returns K rows), and (possibly) K Index Delete. In the async-simple row, "[ ]" indicates asynchronous operations. As previously discussed, we do not distinguish put and delete because their I/O costs are similar in LSM.

6.2 ACID Properties

Atomicity. When a base *put*() occurs, there are three follow-up operations shown in the right-hand side of Equation 1. Each individual operation, i.e., P_I, R_B, and D_I, is atomic. This is offered by the atomicity of *put*(), *read*() and *delete*(), in most LSM stores including HBase. If any of them fails, we do not roll-back the base *put*(). Instead, we insert failed tasks into AUQ where they will be retried by APS until eventually success. By this, as long as the base *put*() succeeds, the associated index update is guaranteed to eventually complete. In our schemes there is no global transaction so the three operations are not guaranteed to occur at the same time point. This of course has implications on consistency and isolation but with the advantage of no global locking and coordination.

Consistency. (1) Schemes sync-full and sync-insert are *causal consistent*. This means, once all index update operations complete and a SUCCESS is returned to the client, both base data and its index are persisted. Else, if any index operation fails, the base *put*() is still persisted and the failed operations is added to the AUQ. In this case, causal consistency is no longer guaranteed and AUQ is responsible for the *eventual consistency* of index updates.
(2) Schemes async-simple and async-session provide eventual consistency and session consistency, respectively.

Isolation. HBase provides per region server read committed semantics. We neither break this nor provide anything more advanced. Therefore, during an index update, other concurrent clients can see partial results. For example, a client may see the base data but not the corresponding index entry at a time point between P_B and P_I, since base and index entry may be on different region servers. Other systems such as MegaStore [4] have argued that such a scheme is useful when there is a strict latency requirement.

Durability. (1) As already seen in Section 5.3, the durability of asynchronous schemes are guaranteed by WAL and AUQ, and the drain-AUQ-before-flush policy.
(2) For synchronous schemes, i.e., in sync-full and sync-insert, operations P_I, R_B, and D_I are done after base *put*(). If any of them fails, we add the failed operation to AUQ which is responsible for the (eventual) re-execution of it. By this means, causal consistency degrades to eventual consistency, and the durability is guaranteed by the AUQ durability protocol in (1).

7. IMPLEMENTATION ON HBASE AND IBM BIGINSIGHTS

We implement Diff-Index on HBase, as HBase is a widely used LSM store and also a part of *IBM InfoSphere BigInsights* V2.1 [19]. BigInsights is an IBM big data product built on top of the open-source Hadoop ecosystem. It has an IBM SQL engine called *Big SQL* to manipulate data in Hadoop and HBase. Diff-Index facilitates Big SQL to create secondary indexes for HBase, maintain them, and use index to speed up queries.

In Figure 6, Diff-Index has a server side component (right) and a client side one (left). The server side contains three coprocessors, i.e., *SyncFullObserver*, *SyncInsertObserver* and *AsyncObserver*. SyncFullObserver and SyncInsertObserver implements schemes sync-full and sync-insert, respectively. AsyncObserver implements async-simple, and async-session with the help of the client-side. Coprocessor is a plug-in fea-

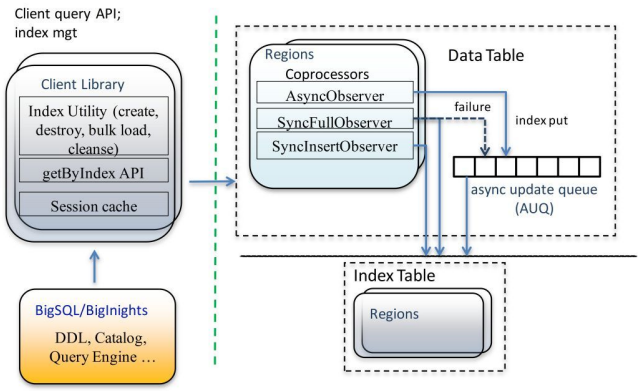

Figure 6: Diff-Index implementation on HBase and IBM BigInsights.

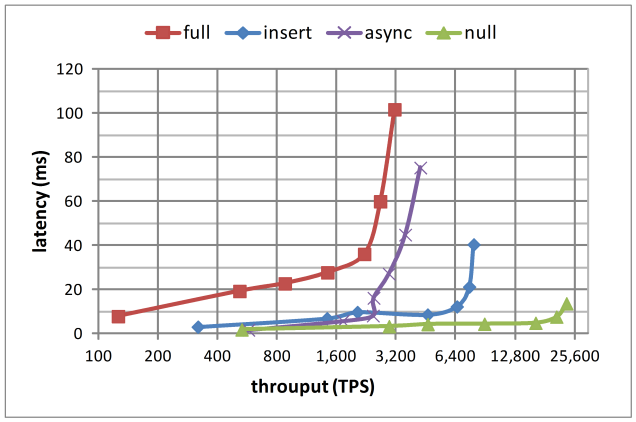

Figure 7: Update performance.

ture of HBase and its function is comparable to triggers and stored procedures in RDBMS. Coprocessor provides a means to extend HBase's function without intruding into its core code. The three coprocessors are deployed in each index-enabled table. They listen to and intercept each data entry made to the hosting table, and act based on the schemes they implement. Each index in a table can choose one scheme out of four offered by Diff-Index.

The client side component consists of (1) a utility for index creation, maintenance and cleanse; (2) the index read API *getByIndex*; and (3) a session cache help enforce session consistency with the server-side AsyncObserver. In case of sync-insert, a check-and-clean routine mentioned in Algorithm 2 is added in getByIndex. For async-session, results from index and session cache are combined in getByIndex. The client library in turn is integrated with Big SQL. In Big SQL, *DDL* component creates indexes using a CREATE INDEX statement; *Catalog* stores index metadata and also put a copy in HBase table descriptor; *Query Engine* uses index metadata in query planning, and accesses indexes via the aforementioned *getByIndex* API in query execution.

Besides those already discussed in this paper, other salient features of Diff-Index include: the support for composite index, indexing dense columns[1], and indexing columns with customer encoding scheme. For more details about Big SQL and index usage please refer to [25].

8. EXPERIMENTS

8.1 Experiment Setup

Unless otherwise mentioned, experiments are conducted on a 10 machine cluster. Each machine is with two quad-core Intel Xeon E5440@2.83 GHz, 32 GB of RAM, 2 TB hard disk, and Ubuntu 10.04.1 LTS. A HDFS is deployed with 3-way replication. HDFS NameNode, Hadoop JobTracker, ZooKeeper and HBase Master are all deployed in one machine. We used YCSB (Yahoo! Cloud Serving Benchmark) [12] as the workload driver and deploy it in a second machine. Each of the remaining 8 nodes is deployed with three processes, i.e., HDFS DataNode, Hadoop TaskTracker, and HBase RegionServer. We assign 1 GB heap each to DataNode and TaskTracker; we assign 8 GB heap to RegionServer process and 25% (i.e., 2GB) of it to block cache.

We extend YCSB by adding a *item* table in which each row has a unique *item_id* as the rowkey and 10 columns. Among them, *item_title* and *item_price* are two columns to index. Therefore we also add two index tables *item_title* and *item_price*. The other 8 columns are each fed with 100 byte long random byte arrays of no particular meaning. Altogether each row is approximately of 1 KB in size and consumes 1.5 KB storage in HBase with overhead (including space for column family name, column name, timestamp, and amortized space for HFile index and BloomFilter). The data table *item* contains 40 million rows and is approximately 60 GB in size. Each index entry is approximately 60 bytes. We evenly distribute the data and index table among all 8 region servers. Therefore each server contains approximately 7.5 GB of base data, which makes the read disk-bounded.

We test the system with 1 to 320 client threads to achieve various throughput. Each client thread continuously submits read/write request to the system. A completed request will be followed up by another one immediately. Update performance is measured with 1 million requests or 30 minutes, whichever takes longer; this is to make sure that flush and compaction both occur frequently during the workload. Read is measured with a warmed block cache and 15 minutes run. For a fair comparison with sync-full, we turn off the client buffer in both YCSB and coprocessors. As a consequence, the throughput we report is not as good as those in [12]. We argue that rather than the absolute numbers, the relative performance of different schemes are more interesting. For simplicity, we use *async* for *async-simple*, *full* for *sync-full*, *insert* for *sync-insert*, and *null* for no index. We use index *item_price* to test range query and index *item_title* for all other experiments.

8.2 Experiment Results

Performance of index update. We have shown that in a moderate cluster and data set, query-by-index is 2-3 orders of magnitude faster compared to parallel-table-scan [15]. Now we test index update latency of Diff-Index schemes, under different throughput (measured in transactions per

[1]A dense column is a column comprising multiple fields each of which is with a different type and encoding. Using dense columns, which is basically combining multiple columns into one, can reduce the storage overhead brought by a KV store like HBase.

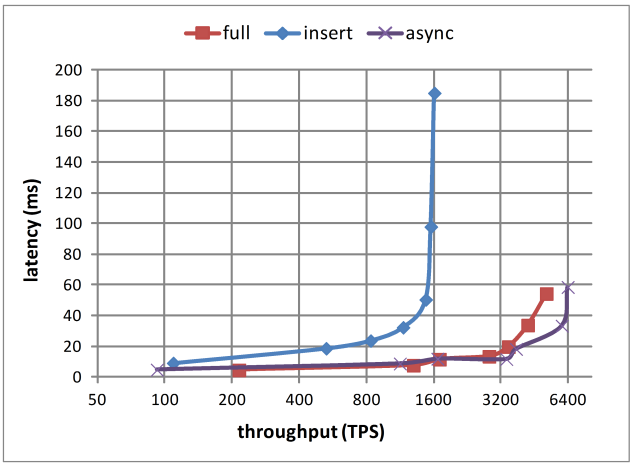

Figure 8: Read performance.

second, TPS). For comparison purpose we also include the latency of base put with no index. In Figure 7, the latency of sync-insert is approximately two times of a base put, because it does one additional index put besides a base put. Sync-full's latency can be five times higher since it involves a base read. In terms of async, its latency is close to no-index when the workload is low; its latency increases and surpasses that of sync-insert when the workload gets higher since the background AUQ competes for system resource.

Performance of index read. Similarly, we test index read latency under different throughput. We first use an exact match query that returns only one row. In Figure 8, sync-full achieves very low latency since it only needs to access the relatively smaller index table; sync-insert's latency is much higher because it involves an additional base table read to check if the index is stale. In terms of async, its read latency is close to sync-full however the result obtained is not guaranteed to be consistent.

Figures 7 and 8 shows that, Diff-Index schemes make different trade-offs among update and read performance, and consistency. Again, we emphasize that the throughput of the system can be further optimized by enabling client buffer for update, tuning Hadoop and HBase configuration parameters, etc. But it is not within the scope of this paper.

Range query with index. We then test the latency of range queries, using 10 concurrent client threads and with selectivity varying from 0.0001% (40 rows in result) to 0.1% (40k rows in result), by setting the query range for *item_price*. In Figure 9, sync-insert has a much larger latency as selectivity grows lower (numbers increasing from 0.0001% to 0.1%). This is because that, when selectivity gets lower, a lot of rows are returned from index query; each of these rows needs to be double checked and involves a base read.

Diff-Index on cloud. We also conduct experiments in a larger cluster in IBM Research Compute Cloud (RC2) [24], an IaaS cloud for internal IBM use, to test the scalability of Diff-Index. In RC2 we create a virtual cluster consisting of 42 virtual machines 40 of which are Hadoop DataNodes and HBase RegionServers, and load 200 million records. Each node is with four virtual CPUs each at 3.0 GHz, 8 GB RAM and 55 GB hard disk. Compared to the in-house cluster with

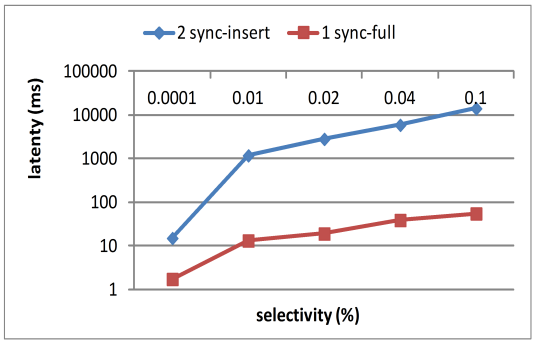

Figure 9: Read latency under different selectivity.

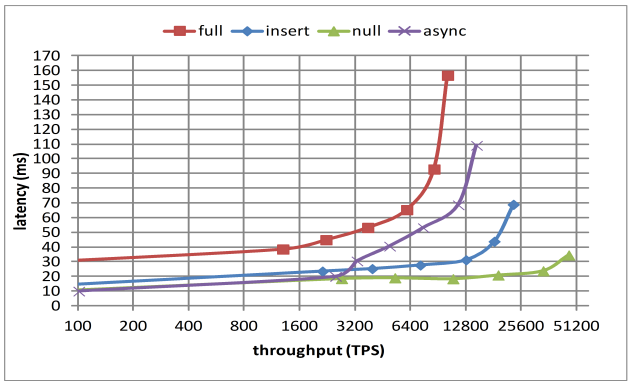

Figure 10: Diff-Index update performance in IBM Research Compute Cloud (RC2).

8 data servers and 40M rows, this virtual cluster is five times as large, in terms of data nodes and data volume.

Figure 10 shows the update performance of different Diff-Index schemes. If linear scale-out could be achieved, 1) 40-region-server cluster should achieve 5x TPS versus the 8-region-server cluster; 2) the update latency of $5x$ TPS in this 40-region-server cluster should be close to that of x TPS in the 8-region-server cluster. The experimental results shows that, compared to Figure 7, 1) the 40-region-server cluster reaches less than 4x TPS; 2) the latencies of all schemes in $5x$ TPS case, are a couple of times larger than those of x TPS in the 8-server-cluster, respectively. Despite the sub-linear scale-out, the relative performance of all Diff-Index schemes remain in RC2. The reasons preventing linear scale-out are the following: 1) the virtual servers are less powerful than the physical servers in configuration; 2) virtualization brings a layer of indirection; 3) we also observed contention in both network and disk I/O among virtual machines.

Index consistency in async-simple. In scheme async-simple, index update operations are pushed to the AUQ to be processed, and therefore there is a time-lag between 1) when a data entry is visible in base table and 2) when its associated index is visible. We measure the distribution of this time-lag under different transaction rates, from 600 TPS to 4000 TPS. To measure staleness, the AUQ in every region server records two timestamps T_1 and T_2 for an index update it handles. T_1 is the timestamp of the base data entry in base table, denoting the time when the data persists in base table; T_2 is the timestamp when AUQ has completed all

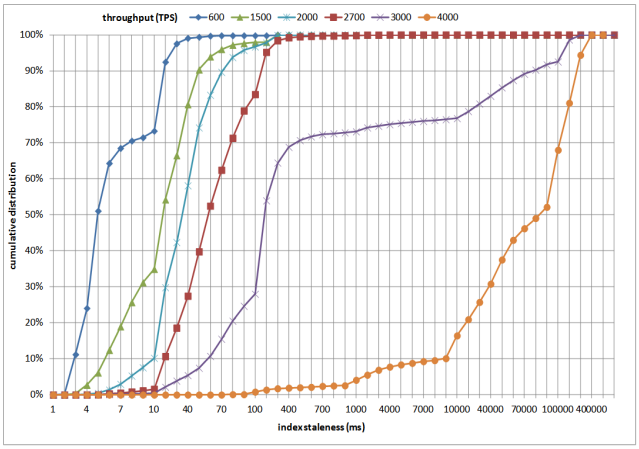

Figure 11: Time-lag between data and index.

(asynchronous) update tasks for this base entry. The index-after-data time-lag for a record is the difference of T_2 and T_1. Excessive measurement may affect the accuracy, and we avoid this by sampling only a small fraction (0.1%) of the inserted data entries.

Figure 11 shows that, staleness of async index grows with the transaction rate. When the system load is modest (600-2700 TPS), most index entries are updated within 100 ms. When system load gets higher, the background AUQ process contends for resources and the system is close to saturate. In 4000 TPS, index can be up to several hundred seconds late.

For the sync-full and async curves in Figure 7, async reaches a throughput 30% higher than sync-full (4200 vs. 3200 TPS). This moderate higher throughput is credited to the batching of operations in AUQ. Moreover, if the high load is transient, by assigning a large-size AUQ the workload surge can be largely absorbed and much higher throughput can be achieved by using async.

The experiments in this section illustrate that the performance and scalability of Diff-Index is satisfactory. The relative cost of different Diff-Index schemes matches well with the complexity analysis in Section 6.1 and Table 2.

9. RELATED WORK

Diff-Index is related to research in both database management and distributed systems.

B-tree vs. LSM. There are many studies in B-Tree based indexes [10, 16, 20]. However, the structural difference between LSM and B-Tree prevents B-Tree based approaches from being directly applied. For example, Equation 1 indicates that in LSM, the latency of index maintenance consists of $L(P_I)$, $L(R_B)$ and $L(D_I)$. Compared to B-Tree: (1) In LSM, $L(P_I)$ and $L(D_I)$ is much smaller and $L(R_B)$ is much larger; (2) RDBMS distinguishes insert and update, so R_B (i.e., read base table to get old value) is done during the put and not needed again in index update. Therefore Equation 1 for B-tree is: $L(sync\text{-}full) = L(P_I) + L(D_I)$. This illustrates that R_B is a unique and significantly expensive operation involved in LSM index and no prior work has addressed it.

On LSM side, bLSM [26] improves LSM's read performance by using a geared scheduler and BloomFilters. While complementary to Diff-Index, we expect bLSM to improve the performance of all Diff-Index schemes.

NoSQL stores with index. Some NoSQL stores explored ways of supporting secondary indexes. MongoDB [21] and Cassandra support local indexes. Chen *et al.* [8] propose a framework for DBMS-like indexes in the cloud, and evaluate its performance using distributed hash and B+ tree structures, respectively. Compared to them, Diff-Index is specifically tailored for LSM stores, exploiting LSM features and with a finer grained spectrum of index maintenance schemes. Huawei has recently released its HBase index implementation [17], which is local and with synchronous update only. In comparison, Diff-Index is global and optimized for highly selective queries; Diff-Index also offers a broader spectrum of index schemes. Google Spanner [13] supports transactional consistent index across data centers, with a relative high commit latency (50-150 ms). Diff-Index is more desirable when update latency is of paramount importance and index inconsistency can be tolerated, being complementary to Spanner having consistent indexes with a higher latency. Diff-Index is inspired by the idea of asynchronous view maintenance [1] in Yahoo PNUTS. In comparison, Diff-Index addresses and leverages the specific features of LSM, for consistency/latency trade-off, concurrency and failure recovery.

Consistency of replicated data. Many systems including Dynamo [14] focus on how to balance consistency and performance in replicated data stores. While Google Spanner [13] and OMID [29] support global transaction and synchronous replication, we argue that Diff-Index is very useful when data ingestion rate is of much higher priority compared to index freshness. In many real-life workloads we observed, even when index cannot catch up, data ingestion does not tolerate interruption or excessive delay.

10. CONCLUSION

This paper addresses the issue of index maintenance in one popular category of NoSQL stores, i.e., LSM store. We demonstrated that, the unique features of LSM, i.e., no in-place update, asymmetric read/write latency, and the distributed nature, make it challenging to maintain a fully consistent index with reasonable update performance. We proposed *Diff-Index*, a spectrum of index update schemes consisting of *sync-full*, *sync-insert*, *async-simple* and *async-session*. Guided by CAP theorem, these schemes offer different trade-offs between index update latency and consistency. Experiments on HBase show that *sync-insert* and *async-simple* can reduce 60%-80% of the overall index update latency when compared to the baseline *sync-full*; *async-simple* can achieve superior index update performance with an acceptable inconsistency; the read performance of *sync-insert* is acceptable when query selectivity is high. These quantitative results show that Diff-Index offers a variety of performance/consistency trade-offs, and scale well without introducing any global locking or coordination.

As far as we know, Diff-Index is the first global index design on LSM, offering a wide spectrum of index maintenance schemes that balance performance and consistency. We exploit the LSM-specific features to achieve light-weight concurrency control and failure recovery. We implement Diff-Index on HBase based on its newly introduced coprocessor framework. While the implementation is on HBase but the design principle are equally applicable to other LSM stores

such as Cassandra and LevelDB. Diff-Index is included in IBM InfoSphere BigInsights, an IBM big data offering. In future work we plan to investigate workload-aware scheme selection, and index compression [5] in Diff-Index.

11. ACKNOWLEDGMENTS

We thank colleagues in IBM InfoSphere BigInsights team, especially Bert Van der Linden and Deepa Remesh, for constructive discussions on use cases and features of HBase index.

12. REFERENCES

[1] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous View Maintenance for VLSD Databases. In *SIGMOD*, pages 179–192, 2009.

[2] Apache.org. Apache Cassandra. `http://cassandra.apache.org/`, 2012.

[3] Apache.org. Apache HBase. `http://hbase.apache.org/`, 2012.

[4] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[5] B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat. Efficient index compression in DB2 LUW. *PVLDB*, 2(2):1462–1473, Aug. 2009.

[6] E. Brewer. Pushing the CAP: Strategies for consistency and availability. *IEEE Computer*, 45(2):23–29, 2012.

[7] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[8] G. Chen, H. T. Vo, S. Wu, B. C. Ooi, and M. T. Özsu. A framework for supporting DBMS-like indexes in the cloud. *PVLDB*, 4(11):702–713, 2011.

[9] D. Choy and C. Mohan. Locking protocols for two-tier indexing of partitioned data. In *International Workshop on Advanced Transaction Models and Architectures*, 1996.

[10] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[11] 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*, 1(2):1277–1288, Aug. 2008.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC '10*, pages 143–154, New York, NY, USA, 2010. ACM.

[13] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, and P. Hochschild. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[14] 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*, volume 41, pages 205–220. ACM, 2007.

[15] L. Fong, Y. Gao, X. Guerin, Y. Liu, T. Salo, S. Seelam, W. Tan, and S. Tata. Toward a scale-out data-management middleware for low-latency enterprise computing. *IBM Journal of Research and Development*, 57(3/4):6:1–6:14, 2013.

[16] G. Graefe. B-tree indexes for high update rates. *SIGMOD Rec.*, 35(1):39–44, Mar. 2006.

[17] Huawei. Secondary index in HBase. `https://github.com/Huawei-Hadoop/hindex`, 2013.

[18] P. Hunt, M. Konar, F. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 10, 2010.

[19] IBM. What is New in IBM InfoSphere BigInsights v2.1. (`http://www-01.ibm.com/software/data/infosphere/biginsights/whats_new.html`), 2013.

[20] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *ICDE*, 2013.

[21] MongoDB. MongoDB Indexes. `http://docs.mongodb.org/manual/indexes/`, 2012.

[22] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[23] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradar, C. Beaver, G. Brandt, M. Gandhi, K. Gopalakrishna, W. Ip, S. Jgadish, S. Lu, A. Pachev, A. Ramesh, A. Sebastian, R. Shanbhag, S. Subramaniam, Y. Sun, S. Topiwala, C. Tran, J. Westerman, and D. Zhang. On brewing fresh espresso: Linkedin's distributed data serving platform. In *SIGMOD*, pages 1135–1146, New York, NY, USA, 2013. ACM.

[24] K. D. Ryu, X. Zhang, G. Ammons, V. Bala, S. Berger, D. M. D. Silva, J. Doran, F. Franco, A. Karve, H. Lee, J. A. Lindeman, A. Mohindra, B. Oesterlin, G. Pacifici, D. Pendarakis, D. Reimer, and M. Sabath. RC2-a living lab for cloud computing. In *USENIX LISA*, pages 1–14. USENIX Association, 2010.

[25] C. M. Saracco and U. Jain. What's the big deal about Big SQL? (`http://www.ibm.com/developerworks/library/bd-bigsql/`), 2013.

[26] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *SIGMOD*, pages 217–228, 2012.

[27] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data, 1994.

[28] S. Tiwari. *Professional NoSQL*. John Wiley and Sons, 2011.

[29] M. Yabandeh and D. Gómez Ferro. A critique of snapshot isolation. In *EuroSys '12*, pages 155–168, New York, NY, USA, 2012. ACM.