Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP

Chen Luo*
University of California, Irvine
cluo8@uci.edu

Pınar Tözün†
IT University of Copenhagen
pito@itu.dk

Yuanyuan Tian
IBM Research - Almaden
ytian@us.ibm.com

Ronald Barber
IBM Research - Almaden
rjbarber@us.ibm.com

Vijayshankar Raman
IBM Research - Almaden
ravijay@us.ibm.com

Richard Sidle
IBM
ricsidle@ca.ibm.com

ABSTRACT

The rising demands of real-time analytics have emphasized the need for Hybrid Transactional and Analytical Processing (HTAP) systems, which can handle both fast transactions and analytics concurrently. Wildfire is such a large-scale HTAP system prototyped at IBM Research - Almaden, with many techniques developed in this project incorporated into the IBM's HTAP product offering. To support both workloads efficiently, Wildfire organizes data differently across multiple *zones*, with more recent data in a more transaction-friendly zone and older data in a more analytics-friendly zone. Data *evolve* from one zone to another, as they age. In fact, many other HTAP systems have also employed the multi-zone design, including SAP HANA, MemSQL, and SnappyData. Providing a *unified* index on the large volumes of data across multiple zones is crucial to enable fast point queries and range queries, for both transaction processing and real-time analytics. However, due to the scale and evolving nature of the data, this is a highly challenging task. In this paper, we present Umzi, the *multi-version* and *multi-zone* LSM-like indexing method in the Wildfire HTAP system. To the best of our knowledge, Umzi is the first indexing method to support evolving data across multiple zones in an HTAP system, providing a consistent and unified indexing view on the data, despite the constantly on-going changes underneath. Umzi employs a flexible index structure that combines hash and sort techniques together to support both equality and range queries. Moreover, it fully exploits the storage hierarchy in a distributed cluster environment (memory, SSD, and distributed shared storage) for index efficiency. Finally, all index maintenance operations in Umzi are designed to be non-blocking and lock-free for queries to achieve maximum concurrency, while only minimum locking overhead is incurred for concurrent index modifications.

1 INTRODUCTION

The popularity of real-time analytics, e.g., risk analysis, online recommendations, and fraud detection etc., demands data management systems to handle both fast concurrent transactions (OLTP) and large-scale analytical queries (OLAP) over fresh data. These applications ingest data at high-speed, persist them into disks or shared storage, and run analytical queries simultaneously over newly ingested data to derive insights promptly.

The necessity of real-time analytics prompts the emergence of Hybrid Transactional and Analytical Processing (HTAP) systems, e.g., MemSQL [7], SnappyData [28], SAP HANA [21], and among others. HTAP systems support both OLTP and OLAP queries in a single system, thus allowing real-time analytics over freshly ingested data. Wildfire [15] is a large-scale HTAP system, prototyped at IBM research - Almaden. Many of the techniques developed in this research project have been incorporated into the IBM Db2 Event Store offering [4]. Wildfire leverages the Spark ecosystem [10] to enable large-scale data processing with different types of complex analytical requests (SQL, machine learning, graph analysis, etc), and compensates Spark with an underlying engine that supports fast transactions with snapshot isolation and accelerated analytics queries. Furthermore, it stores data in open format (Parquet [8]) on shared storage, so that other big data systems can access consistent snapshots of data in Wildfire. The back-end shared storage that Wildfire supports includes distributed file systems like Hadoop Distributed File System (HDFS) and GlusterFS [2], as well as object-based storage on cloud like Amazon S3 and IBM Cloud Object Storage.

To support efficient point lookups and range queries for high-speed transactional processing and real-time analytics, fine-grained indexing is mandatory in a large-scale HTAP system like Wildfire. However, indexing large volumes of data in an HTAP system is highly non-trivial due to the following challenges.

Challenges due to shared storage. First of all, for large-scale HTAP, memory-only solutions are not enough. As a result, most HTAP systems, including Wildfire, persist data in highly-available fault-tolerant shared storage, like HDFS and Amazon S3, etc. However, most of these shared storage options are not good at random access and in-place update. For example, HDFS only supports append-only operations and optimizes for block-level transfers, and object storage on cloud allows neither random access inside an object nor update to an object. To accommodate the unique characteristics of shared storage, index operations, e.g., insert, update and delete, have to leverage sequential I/Os without in-place updates. Naturally, LSM-like index structures are more appealing.

Furthermore, a typical shared storage prefer a small number of large files to a large number of small files. This is not only because of the overhead in metadata management, e.g., the maximum number of files supported by an HDFS cluster is determined by how much memory is available in the namenode, but more importantly because of the reduced seek time overhead when accessing larger files.

Finally, accessing remote shared storage through networks for index lookups is costly. Thus, indexing methods on HTAP must fully exploit the storage hierarchy in a distributed cluster environment for efficiency. Particularly, nowadays, we can take advantage of large memories and SSDs in modern hardware. Due to the large scale of data in HTAP systems, however, only the most frequently accessed portions of indexes can be cached locally,

*†Work done while at IBM Research - Almaden.

while leaving cold entries in shared storage. Effective caching mechanisms must be developed to facilitate index lookup.

Challenge due to evolving nature of data. Since HTAP systems have to support both transactional and analytical workloads efficiently, many of them [7, 14, 21, 23, 28] store data in different organizations, typically one organization good for transactions on the more recent data and one organization good for analytics on the older data. We call the different data organizations *zones*. As data age in the system, they evolve from the transaction-friendly zone to the analytics-friendly zone. In Wildfire, transactions first append writes into a transaction log, which is then groomed into columnar data blocks. The groomed data is further periodically post-groomed to a more analytics-friendly organization that is optimal for queries by creating data versions, data partitioning, and larger data blocks. SAP HANA organizes data into a read-optimized main store and a write-optimized delta store. Writes are first buffered into the row-major delta store, which is further transformed into the columnar main store to facilitate analytical queries. Some loosely-coupled HTAP solutions employ NoSQL stores, like HBase [3] or Cassandra [1], for operational workloads, and periodically copy data from the NoSQL stores into files in columnar format like Parquet or ORC-File on the shared storage, so that SQL-on-Hadoop engines, like Hive [35] or SparkSQL [13], can efficiently query them. The data evolution across different zones in these HTAP systems/solutions is constantly on-going, posing a significant challenge to building and maintaining indexes.

Existing indexing solutions on multi-zone HTAP systems either support index on the transaction-friendly zone only, like in SnappyData [28] and the loosely coupled HTAP solutions, or support separate indexes on different zones, like in MemSQL [7]. First of all, being able to efficiently query historical data is very important for real-time analytics, especially for analytical queries that are part of a transaction in the true HTAP scenario. As a result, the index needs to cover both recent data and historical data. Secondly, having separate indexes on different zones exposes a divided view of data. This requires queries to perform extra work to combine index query results that span multiple zones. In particular, with the constant evolving nature of HTAP data, it is non-trivial for queries to make sure that there is no duplicate or missing data in the final results. Therefore, it is highly desirable to have a consistent and unified index across the different zones in an HTAP system.

Contributions. To tackle the challenges of indexing in a large-scale HTAP system, we present Umzi, the multi-version and multi-zone LSM-like index in the context of Wildfire. Umzi provides a consistent and unified indexing view across the groomed and post-groomed zones in Wildfire. To the best of our knowledge, Umzi is the first unified multi-zone indexing method for large-scale HTAP systems.

Umzi employs an LSM-like structure with lists of sorted index runs to avoid in-place updates. A novel index-run format that combines hash and sort techniques is introduced to flexibly answer equality/range queries as well as the combination of both using the index. Runs are organized into multiple levels as in today's NoSQL systems, e.g., LevelDB [6] and RocksDB [9]. A new run is added into the lowest level, i.e., level 0, and runs are periodically merged into higher levels within a zone. However, when data evolve from one zone to another, an *index evolve* operation is introduced to build new index runs in the new zone and garbage-collect obsolete index runs from the old zone in a coordinated way, so that the entire index is always in a consistent state. To fully exploit the storage hierarchy, lower index levels can be made non-persistent to speed up frequent merges, and we dynamically adjust cached index runs from memory and SSD to speed-up index lookups and transactional processing. In Umzi, all operations are carefully designed to be non-blocking such that readers, i.e., index queries, are always lock-free while only negligible locking overhead is incurred for index maintenance.

Paper organization. Section 2 provides the background on Wildfire. Section 3 describes an overview of the Umzi index. Section 4 presents the internal structure of Umzi components. Section 5 describes index maintenance operations in Umzi. Section 6 discusses some design decisions of Umzi to exploit the storage hierarchy. Section 7 introduces methods for processing index queries, i.e., range scans and point lookups. Section 8 reports the experimental evaluation of Umzi. Section 9 surveys related work. Finally, Section 10 concludes this paper.

2 BACKGROUND

2.1 Wildfire

Wildfire [15] is a distributed multi-master HTAP system consisting of two major pieces: Spark and the Wildfire engine. Spark serves as the main entry point for the applications that Wildfire targets, and provides a scalable and integrated ecosystem for various types of analytics on big data, while the Wildfire engine adds the support for high-speed transactions, accelerates the processing of application requests, and enables analytics on newly-ingested data. All inserts, updates, and deletes in Wildfire are treated as *upserts* based on the user-defined primary key. Wildfire adopts last-writer-wins semantics for concurrent updates to the same key, and snapshot isolation of quorum-readable content for queries, without having to read the data from a quorum of replicas to satisfy a query.

A table in Wildfire is defined with a *primary key*, a *sharding key*, and optionally a *partition key*. Sharding key is a subset of the primary key, and it is primarily used for load balancing of transaction processing in Wildfire. Inserted records are routed by the sharding key to different shards. A table shard is replicated into multiple nodes, where one replica serves as the shard-leader while the rest are slaves. Any replica of a shard can ingest data. A table shard is the basic unit of a lot of processes in Wildfire, including grooming, post-grooming, and indexing (details later). In contrast, the partition key is for organizing data in a way that benefits the analytics queries. Typically, the sharding key is very different from the partition key. For example, an IoT application handling large volumes of sensor readings could use the device ID as the sharding key, but the date column as the partition key to speed up time-based analytical queries.

Wildfire adds the following three hidden columns to every table to keep track of the life of each record and support snapshot isolation as well as time travel. The column beginTS (begin timestamp) indicates when the record is first ingested in Wildfire; endTS (end timestamp) is the time when the record is replaced by a new record with the same primary key; prevRID (previous record ID) holds the RID (record ID) of the previous record with the same key.

In order to support both OLTP and OLAP workloads efficiently within the same system, Wildfire divides data into multiple zones where transactions first append their updates to the OLTP-friendly zone, which are gradually migrated to the OLAP-friendly zone. Figure 1 depicts the data lifecycle in Wildfire across multiple zones.

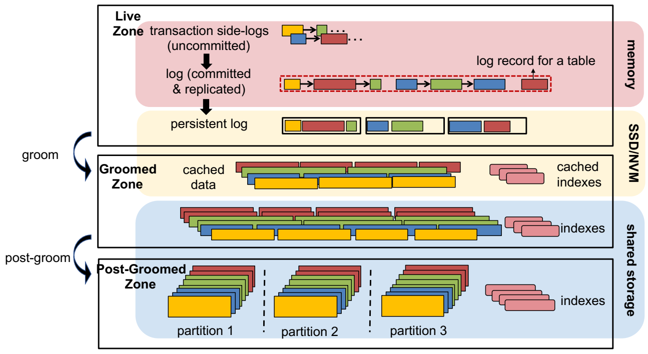


Figure 1: Data Lifecycle in Wildfire

Live Zone. A transaction in Wildfire first appends uncommitted changes in a transaction local side-log. Upon commit, the transaction tentatively sets beginTS for each record using the local wall clock time (beginTS is reset later in the groomed zone), and appends its side-log to the committed transaction log, which is also replicated for high-availability. The committed log is kept in memory for fast access, and also persisted on the local SSDs using the Parquet columnar-format. Since this zone has the latest ingested/updated data, it is called the *live zone*.

Groomed Zone. To bound the growth of the committed log and resolve conflicts from different replicas, each shard in Wildfire has a designated *groomer* which periodically (e.g., every second) invokes a *groom* operation to migrate data from the live zone to the groomed zone.

The groom operation merges, in the time order, transaction logs from shard replicas, resolves conflicts by setting the monotonic increasing beginTS for each record, and creates a Parquet columnar-format data file, called a *groomed block*, in the shared storage as well as the local SSD cache. Each groomed block is uniquely identified by a monotonic increasing ID called *groomed block ID*. Note that the beginTS set by the groomer is composed of two parts. The higher order part is based on the groomer's timestamp, while the lower order part is the transaction commit time in the shard replica. Thus, the commit time of transactions in Wildfire is effectively postponed to the groom time. The groomer also builds indexes over the groomed data.

Post-Groomed Zone. The groomer only sets the beginTS for each record. EndTS and prevRID still need to be set to support snapshot isolation and time travel. In addition, up to the groomed zone, data is still organized according to the sharding key, and they need yet to be re-organized based on the more analytics-friendly partition key. To achieve these tasks, another separate process, called *post-groomer*, periodically (e.g., every 10 minutes) performs *post-groom* operations to evolve the newly groomed data to the post-groomed zone.

The post-groom operation first utilizes the post-groomed portion of the indexes to collect the RIDs of the already post-groomed records that will be replaced by the new records from the groomed zone. Then, it scans the newly groomed blocks to set the prevRID fields using the RIDs collected from the index, and re-organizes the data into post-groomed blocks on the shared storage according to the OLAP-friendly partition key. The post-groomer also uses the same set of RIDs from the index to directly locate the to-be-replaced records and sets their endTS fields. Since the post-groomer is carried out less frequently than the groomer, it usually generates much larger blocks, which results in better access performance on shared storage. At the end, the post-groomer also notifies the indexer process to build indexes on the newly post-groomed blocks.

While both groomed and post-groomed blocks reside in the shared storage, based on the access patterns of a node, they are also cached in the local SSDs of that node, similarly to the indexes.

2.2 LSM-tree

Since Umzi employs an LSM-like structure, here we brief introduce the background of LSM-trees. Interested readers can refer to [27] for a survey of recent research on LSM-trees. The Log-Structured Merge-tree (LSM-tree) is a write-optimized persistent index structure. It first appends all writes into a memory table. When the memory table is full, it is flushed into disk as a sorted run using sequential I/Os. A query over an LSM-tree has to look up all runs to perform reconciliation, i.e., to find the latest version of each key.

As runs accumulate, query performance tends to degrade. To address this, runs are periodically merged together to improve query performance and reclaim disk spaced occupied by obsolete entries. Two LSM merge policies are commonly used in practice, i.e., leveling and tiering [27]. In both merge policies, runs are organized into levels, where a run at level $L + 1$ is T times larger than the run at level L. The leveling policy optimizes for queries by limiting only one run per level. A run is merged with the one at the higher level when its size reaches a threshold. In contrast, the tiering policy optimizes for write amplification by allowing multiple runs at each level. Multiple runs at level L are merged together into a new run at level $L + 1$.

3 UMZI OVERVIEW

In Wildfire, depending on the freshness requirement, a query may need to access data in the live zone, groomed zone, and/or the post-groomed zone. We choose to build indexes over the groomed zone and the post-groomed zone. Indexing does not cover the live zone for a few reasons. First, since groomer runs very frequently, data in the live zone is typically small, which alleviates the need for indexing. Secondly, to support fast data ingestion, we cannot afford maintaining the index on a record basis.

In a nutshell, Umzi employs an LSM-like [31] structure with multiple runs. Each groom operation produces a new index run, and runs are merged over time to improve query performance. Even though Umzi is structurally similar to LSM, it has significant differences from existing LSM-based indexes. First, existing LSM-based indexes either store the records directly into the index itself, e.g., LevelDB [6] and RocksDB [9], or store the records separately with the assumption that each record has a fixed RID, e.g., WiscKey [25] [1]. However, both approaches do not work well in Wildfire. Storing records into the LSM-tree incurs too much write amplification during merges, significantly affecting ingestion performance. While for the second approach, as data evolve from one zone to another, RIDs also change.[2] To accommodate the multi-zone design and the evolving nature, Umzi divides index runs into multiple zones accordingly. Runs in each zone are chained and merged, as in LSM indexes. When data evolve from one zone to another, Umzi performs the evolve operation

[1] The same assumption is generally held by non-LSM-based indexes as well

[2] In Wildfire, an RID is identified by the combination of zone, block ID, and record offset.

to migrate affected index entries to the target zone accordingly. Since data evolution in Wildfire is accomplished by a number of loosely-coupled distributed processes, it is critical for index evolve operations to require minimum coordination among distributed processes and incur negligible overhead for queries.

Furthermore, Umzi targets at multi-tier storage hierarchies in distributed environment, including memory, SSD and shared storage. Index runs are persisted on shared storage for durability, while Umzi aggressively exploits local memory and SSD as a caching layer to speed up index queries. Umzi dynamically adjusts cached index runs based on the space utilization and the age of the data, even without ongoing queries. To improve merge performance and avoid frequent rewrites on shared storage, Umzi allows certain lower levels to be non-persisted, i.e., their runs are only stored in local memory and optionally SSD.

Even though we present Umzi with two zones, i.e., groomed zone and post-groomed zone, this structure does not limits the applicability of Umzi to other systems. It is straightforward to extend Umzi to support other HTAP systems with arbitrary number of zones. To this end, one needs to structure Umzi with multiple run lists, each of which corresponds to one zone of data. When data evolves from one zone to another, the indexing process should be notified to trigger an index evolve operation to migrate index entries accordingly.

As mentioned in Section 2, a table shard is the basic unit of groomer and post-groomer processes in Wildfire. This is also the case for indexing. In the distributed setting of Wildfire, each Umzi index structure instance serves a single table shard. There are a number of indexer daemons running in the cluster. Each runs independently, and is responsible for building and maintaining index for one or more index structure instances. As a result, this paper describes the Umzi index design from the perspective of one table shard.

The following four sections describe the detailed design of Umzi and its index maintenance operations. Without loss of generality, we assume Umzi is used as a primary index throughout the paper.

4 INDEX STRUCTURE

This section details the internal structure of Umzi. We first describe the index definition of Umzi, followed by the singe-run storage format and multi-run structure respectively.

4.1 Index Definition

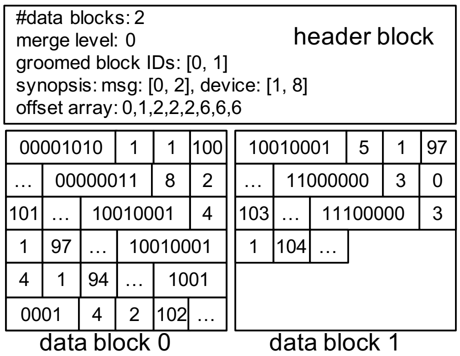
Umzi is designed for supporting both equality queries and range queries, as well as facilitating index-only access plans if possible. Index definitions in Umzi reflect these design goals. An index is defined with key columns plus optionally included columns. Index key columns can be a composition of equality columns (for equality predicates) and sort columns (for range predicates). Included columns are extra columns to enable efficient index-only queries. If equality columns are specified, we also store the hash value of equality column values to speed-up index queries, which makes Umzi a combination of hash and range index. In an example IoT application, the user can define the deviceID as the equality column, while the message number as the sort column. As a special case, the user could leave out the equality column(s), which makes Umzi a complete range index. Similarly, omitting the sort columns would turn Umzi into a hash index. The flexibility of this index structure helps Umzi to answer equality queries, range queries, and combinations of the two, with one index.

	hash	device	msg	beginTS	RID
0	0000 0101	1	1	100	...
1	0010 0011	8	2	101	...
2	1001 0001	4	1	97	...
3	1001 0001	4	1	94	...
4	1001 0001	4	2	102	...
5	1001 0001	5	1	97	...
6	1110 0000	3	0	103	...
7	1110 0000	3	1	104	...

(a) Example Run Data

	offset
000	0
001	1
010	2
011	2
100	2
101	6
110	6
111	6

(b) Offset Array



(c) Physical Layout of An Index Run

Figure 2: Example Index Run: *device* is the equality column, *msg* is the sort column, and there is no included columns. For simplicity, we assume the hash value takes only one byte.

4.2 Run Format

Each run in Umzi can be logically viewed as a sorted table containing the hash column, index key columns, included columns, beginTS and RID. As mentioned before, the hash column stores the hash value of equality columns (if any) to speed up queries with equality predicates. The beginTS column indicates the timestamp when the indexed record is inserted, which is generated by groomers in Wildfire (Section 2). The RID column defines the exact location of the indexed record. Index entries are ordered by the hash column, equality columns, sort columns, and descending order of beginTS. We sort the beginTS column in descending order to facilitate the access of more recent versions. All ordering columns, i.e., the hash column, equality columns, sort columns and beginTS, are stored in lexicographically comparable formats, similar to LevelDB [6], so that keys can be compared by simply using memory compare operations when processing index queries. Figure 2 shows an example index run, where *device* is the equality column and *msg* is the sort column. There is no included columns in this example, and we assume the hash value takes only one byte. Figure 2a shows the index rows in this run, where the hash value is shown in the binary format.

An index run is physically stored as a header block plus one or more fixed-size data blocks. The header block contains the metadata information of the index run, such as the number of data blocks, the merge level this run belongs to (Section 5), and the range of groomed block IDs to which this run corresponds.

To prune irrelevant runs during index queries, we also store a synopsis in the header block. The synopsis contains the range (min/max values) of each key column stored in this run. A run can be skipped by an index query if the input value of some key column does not overlap with the range specified by the synopsis. Figure 2c shows an example index run layout which contains one header block and two data blocks.

When equality columns are specified in the index definition, we store in the header block an offset array of 2^n integers to facilitate index queries. The offset array maps the value of the most significant n bits of hash values to the offset in the index run. When processing index queries, the offset array can be used to provide a more compact start and end offset for binary search. For example, Figure 2b shows the offset array with the most significant 3 bits of hash values from Figure 2a.

4.3 Multi-Run Structure

An example multi-run structure of Umzi is shown in Figure 3. Similar to LSM indexes, Umzi contains multiple runs. A groom operation produces a new run to level 0, and runs from lower levels are gradually merged into higher levels to improve query performance. Each run is further labeled with the range of groomed block IDs, where larger IDs correspond to newer groomed blocks.

In this meanwhile, to accommodate the multi-zone design and data evolving nature of the HTAP systems, Umzi divides index runs in multiple zones accordingly. For concurrency control purpose, runs in each zone are chained together based on their recency into a list, where the header points to the most recent run. We will further discuss concurrency control of Umzi in Section 5.1. Based on this multi-zone design, Umzi only merges runs within the same zone. When data evolves from one zone to another, an index evolve operation is triggered to migrate index entries to the target zone accordingly.

The assignment of levels to zones are configurable in Umzi. For example in Figure 3, levels 0 to 5 are configured as the groomed zone, while levels 6 to 9 are configured as the post-groomed zone.

5 INDEX MAINTENANCE

In this section we describe index maintenance operations in Umzi, including index build, merge, and evolve. Before presenting the details of index maintenance operations, we first discuss concurrency control in Umzi since index maintenance is performed concurrently with queries. Finally, we also briefly discuss how recovery is performed in Umzi.

5.1 Concurrency Control

Umzi aims at providing non-blocking and lock-free access for queries. To this end, Umzi relies on atomic pointers and chains runs in each zone together into a linked list, where the header points to the most recent run. All maintenance operations are carefully designed so that each index modification, i.e., a pointer modification, always results in a valid state of the index. As a result, queries can always traverse run lists sequentially without locking to get correct results. To minimize contentions caused by concurrent index maintenance operations, each level is assigned a dedicated index maintenance thread. A short duration lock is acquired when modifying the run list to prevent concurrent modifications. Note that the locking overhead is negligible since locks are only used to prevent concurrent modifications to the run list, which happens infrequently. Moreover, these locks never block any index queries.

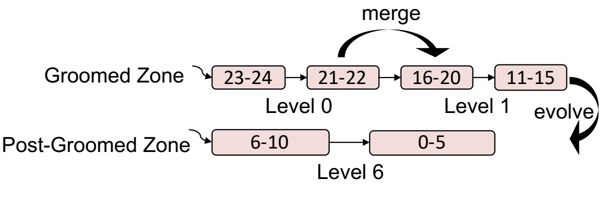


Figure 3: Multi-Run Structure in Umzi

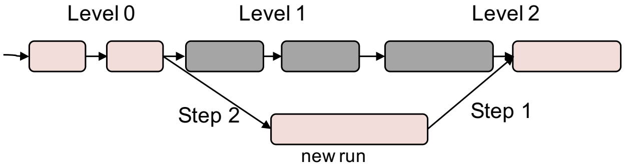


Figure 4: Index Merge Example

5.2 Index Build

After a groom operation is completed, Umzi builds an index run over the newly groomed data block. This is done by simply scanning the data block and sorting index entries in ascending order of hash values, sort columns, equality columns and descending order of beginTS. Along with writing sorted index entries back to data blocks, the offset array can be computed on-the-fly. Finally, the new run becomes the new header of the run list for the groomed zone. Note that the order of pointer modifications is important to guarantee the correctness for concurrent queries, where the new run must be set to point to the header before the header pointer is modified.

5.3 Index Merge

In order to easily trade-off write amplification and query performance, Umzi employs a hybrid merge policy similar to [20]. This policy is controlled by two parameters K, the maximum number of runs per level, and T, the size ratio between runs in adjacent levels. Each level L maintains the first run as an active run, while the remaining runs are inactive. Incoming runs from level $L - 1$ are always merged into the active run of level L. When the active run in level L is full, i.e., its size is T times larger than an inactive run in level $L - 1$, it is marked inactive and a new active run is created. Finally, when level L contains K inactive runs, they are merged together with the active run in level $L + 1$.

After a merge, the new run replaces old runs in the run list. As shown in Figure 4, this is done by first setting the new run point to the next run of the last merged run, and then set the previous run before the first merged run point to the new run. A lock over the run list is acquired during the replacement, since otherwise pointers could be concurrently modified by other maintenance threads. There is no need for a query to acquire any locks when traversing the list; it sees correct results no matter whether the old runs or the new run are accessed.

5.4 Index Evolve

In Wildfire, the post-groomer periodically moves groomed data blocks to the post-groomed zone. After a post-groom operation, groomed data blocks are marked deprecated and eventually deleted to reclaim storage space. As a result, index entries in Umzi must be migrated as well so that deprecated groomed blocks are no longer referenced. However, index evolving in distributed HTAP systems is non-trivial due to the following problems.

First, HTAP systems like Wildfire are often composed of several loosely-coupled distributed processes. The post-groomer in Wildfire is a separate process running on a different node from the indexer process. In a distributed environment, one requirement for index evolving is to minimize the coordination among multiple processes. Second, an index evolve operation must apply multiple modifications to the index. This requires index evolve operations to be carefully designed to ensure the correctness for concurrent queries without blocking them.

To tackle the first problem, an index evolve operation in Umzi is performed asynchronously by the indexer process with minimum coordination, as shown in Figure 5. Each post-groom operation is associated with a post-groom sequence number (PSN). After a post-groom operation, the post-groomer publishes the metadata for this operation and updates the maximum PSN. In the meanwhile, the indexer keeps track of the indexed post-groom sequence number, i.e., IndexedPSN, and keeps polling the maximum PSN. If IndexedPSN is smaller than the maximum PSN, the indexer process performs an index evolve operation for IndexedPSN+1, which guarantees the index evolves in a correct order, and increments IndexedPSN when the operation is finished. Note that asynchronous index evolution has no impact on index queries since a post-groom operation only copies data from one zone to another without producing any new data. For a query, it makes no difference to access a record from the groomed zone or post-groomed zone.

For the concurrency control issue, we decompose the index evolve operation into a sequence of atomic sub-operations. Each sub-operation is guaranteed to result in a valid state of the index, ensuring that concurrent queries always see correct results when traversing the run lists. Specifically, an index evolve operation for a given PSN is performed as follows. First, the indexer builds an index run for post-groomed data blocks associated with this PSN, and adds it atomically to the post-groomed run list. Note that this run still contains the range of groomed block IDs it corresponds to. Second, the indexer atomically updates the maximum groomed blocked ID covered by the post-groomed run list, based on the newly built run. All runs in the groomed run list with end groomed block ID no larger than this value would be automatically ignored by queries since entries in these runs are already covered by the post-groomed run list. Finally, these obsolete runs are garbage collected from the groomed run list. Note that during each step, a lock over the run list is acquired when modifying a run list to prevent concurrent modifications by other maintenance threads.

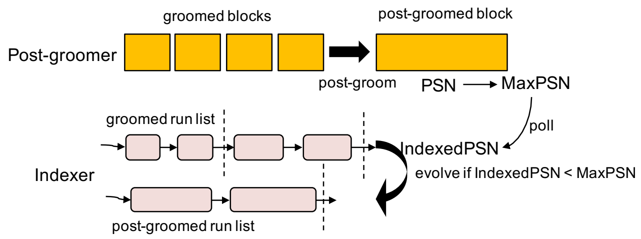


Figure 5: Interaction between Post-groomer and Indexer

We further illustrate the index evolve operation with an example depicted in Figure 6. Suppose the groomed blocks 11 to 18 have been post-groomed, and the indexer now performs an index evolve operation for this post-groom operation accordingly. First, the indexer builds a new run labeled 11-18 for the newly post-groomed data, and atomically adds it to the post-groomed run list. The indexer then atomically updates the maximum groomed blocked ID of the post-groomed run list from 10 to 18. At this moment, run 11-15 will be ignored by subsequent queries since it is fully covered by run 11-18. Finally, this obsolete run is garbage collected from the groomed list, which concludes this index evolve operation.

It is straightforward that each step of an evolve operation only makes one modification to the index, and is thus atomic. Between any two of the above three steps, the index could contain duplicates, i.e., a record with the same version could appear in both a groomed run and a post-groomed run. Moreover, even after the last step of the index evolve operation, the index may still contain duplicates since groomed blocks consumed by a post-groom operation may not align perfectly with the boundaries of index runs. However, duplicates are not harmful to index queries. Since a query only returns the most recent version of each key, duplicates are removed on-the-fly during query processing (Section 7).

5.5 Recovery

We assume runs in Umzi are persisted in shared storage. After each index evolve operation, the maximum groomed blocked ID for the post-groomed run list and IndexedPSN are also persisted. However, an indexer process could crash, losing all data structures in the local node. To recover an index, we mainly need to reconstruct run lists based on runs stored in shared storage, and cleanup merged and incomplete runs if any.

A run list can be recovered by examining all runs in shared storage. Runs are first sorted in descending order of end groomed blocked IDs, and are added to the run list one by one. If multiple runs have overlapping groomed block IDs, the one with largest range is selected, while the rest are simply deleted since they have already been merged.

6 UMZI ON MULTI-TIER STORAGE HIERARCHY

Recall that Umzi is designed for large-scale distributed HTAP systems running on multi-tier storage hierarchy, i.e., memory, SSD, and distributed shared storage. Even though shared storage provides several key advantages for distributed HTAP systems such as fault tolerance and high availability, it brings significant challenges when designing and implementing an indexing component. Shared storage generally does not support in-place

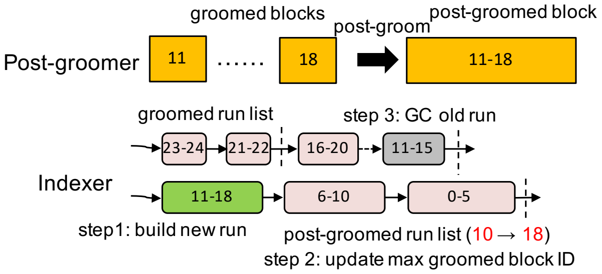


Figure 6: Index Evolve Example

updates and random I/Os, and prefers a small number of large files to reduce metadata overhead. Furthermore, accessing shared storage through networks is often costly, incurring high latency for index queries.

So far, we only discussed how Umzi eliminates random I/Os and in-place updates by adopting an LSM-like structure. In this section, we present solutions adopted by Umzi to improve storage efficiency in a multi-tier storage hierarchy.

6.1 Non-Persisted Levels

In a traditional LSM design, on-disk runs of all levels are persisted on disk equivalently. Even though this design garbage collects olds runs after a merge, it introduces a large overhead on shared storage because of writing a large number of (potentially small) files. To avoid frequently rewriting a large number of small files on shared storage, Umzi supports non-persisted levels, i.e., certain low levels of the groomed zone can be configured as non-persisted.

Runs in persisted levels are always stored in shared storage for fault tolerance and can be cached in local memory and SSD to speedup queries. However, runs in non-persisted levels are only cached in local memory and optionally spilled to SSD if memory is full, but they are not stored in shared storage for efficiency.

Introducing non-persisted levels complicates the recovery process of Umzi, since after a failure all runs in non-persisted levels could be lost. To address this, Umzi requires level 0 must be persisted to ensure that we do not need to rebuild any index runs from groomed data blocks during recovery. Moreover, if level L to K are configured as non-persisted, runs in level L-1 cannot be deleted immediately after they are merged into level L. Otherwise, if the node crashes, we would again lose index runs since the new run is not persisted in shared storage. To handle this, when merging into non-persisted levels, old runs from level L-1 are not deleted but rather recorded in the new run. When the new run is finally merged into a persisted level again, i.e., level K+1, its ancestor runs from level L-1 can be safely deleted.

6.2 Cache Management

As mentioned before, accessing shared storage through networks is often costly and incurs high latencies for index queries. To address this, Umzi aggressively caches index runs using local memory and SSD, even without ongoing queries. We assume most frequently accessed index runs fit into the local SSD cache so that shared storage is mainly used for backup. However, when the local SSD cache is full, Umzi has to remove some index runs to free up the cache space. For this purpose, we assume recent data is accessed more frequently. As the index grows, Umzi dynamically purges old runs, i.e., runs in high levels, from the SSD cache to free up the cache space. In contrast, when the local SSD is spacious, Umzi aggressively loads old runs from shared storage to speedup future queries.

To dynamically purge and load index runs, Umzi keeps track of the current cached level that separates cached and purged runs, as shown in Figure 7. When the SSD is nearly full, the index maintenance thread purges some index runs and decrements the current cached level if all runs in this level have been purged. When purging an index run, Umzi drops all data blocks from the SSD while only keeps the header block for queries to locate data blocks. On the contrary, when the SSD has free space, Umzi loads recent runs (in the reverse direction of purging) into SSD, and increments the current cached level when all runs in the current cached level have been cached. Umzi further adopts a write-through cache policy when creating new index runs during merge or evolve. That is, a new run is directly written to the SSD cache if it is below (lower than) the current cache level.

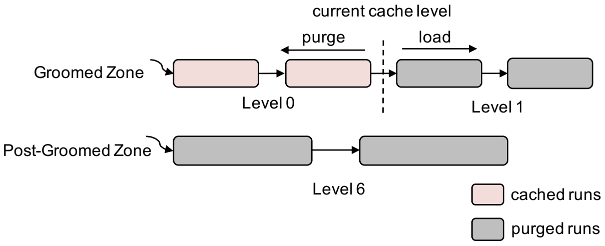


Figure 7: Cache Management in Umzi

7 INDEX QUERY

In this section, we discuss how to process index queries on Umzi. Since Umzi is a multi-version index, a query has to specify a query timestamp (queryTS), and only the most recent version for each matching key is returned, i.e., the version with largest beginTS such that beginTS ≤ queryTS. In general, two types of index queries are supported. The range scan query specifies values for all equality columns (if any) and bounds for the sort columns, and returns the most recent version of each matching key. The point lookup query specifies the entire index key (i.e., the primary key), and at most one matching record is returned.

A query first collects candidate runs by iterating the run lists and checking run synopses. A run is considered as a candidate only if all column values as specified in the query satisfy the column ranges in the synopsis. Also note that all runs are read from the SSD cache. In case that a query must access purged runs, we first transfer runs from shared storage to the SSD cache on a block-basis, i.e., the entire run data block is transferred at a time, to facilitate future accesses. After the query is finished, the cached data blocks are released, which are further dropped in case of cache replacement. Depending on the query type, the details of query processing are as follows.

7.1 Range Scan Query

For a range scan query, we first discuss how to search a single run to get matching keys. Results returned from multiple runs are further reconciled, since results from newer runs could override those from older runs, to guarantee only the most recent version is returned for each matching key.

7.1.1 Search Single Run. Searching a single run returns the most recent version for each matching key in that run. Since a run is a table of sorted rows, the query first locates the first matching key using binary search with the concatenated lower bound, i.e., the hash value, equality column values, and the lower bound of sort column values. If the offset array is available, the initial search range can be narrowed down by computing the most significant n bits of the hash value (denoted as i) and taking the i-th value in the offset array.

After the first matching key is determined, index entries are then iterated until the concatenated upper bound is reached, i.e., the concatenation of the hash value, equality column values, and the upper bound of sort column values. During the iteration, we further filter out entries failing timestamp predicate beginTS ≤ queryTS. For the remaining entries, we simply return for each key the entry with the largest beginTS, which is straightforward since entries are sorted on the index key and descending order of beginTS.

Consider again the example run in Figure 2. Recall that *device* is the equality column, while *msg* is the sort column. Consider a range scan query with *device* = 4, 1 ≤ *msg* ≤ 3, and queryTS = 100. We first take the most significant 3 bits of hash(4) = 1001 0001, i.e., 100, to obtain the initial search range from the offset array, i.e., 2 to 6. The first matching key is still entry 2 after binary search with the input lower bound (1001 0001, 4, 1). We then iterate index entries starting from entry 2. Entry 2 is returned since it is the most recent version for key (4, 1), while entry 3 is filtered out since it is an older version of entry 2. However, entry 4 is filtered out because its beginTS 102 is beyond the queryTS 100. We stop the iteration at entry 5, which is beyond the input upper bound (1001 0001, 4, 3).

7.1.2 Reconcile Multiple Runs. After searching each run independently, we have to reconcile results returned from multiple runs to ensure only the most recent version is returned for each matching key. In general, two approaches can be used for reconciliation: the set approach and the priority queue approach.

Set Approach. In the set approach, the query searches from the newest to the oldest runs sequentially, and maintains a set of keys which have already been returned to the query. If a key has not been returned before, i.e., not in the set, it is added to the set and the corresponding entry is returned to the query; otherwise, the entry is simply ignored since we have already returned a more recent version from the newer runs. The set approach mainly works well for small range queries since it requires intermediate results to be kept in memory during query processing.

Priority Queue Approach. In the priority queue approach, the query searches multiple runs together and feeds the results returned from each run into a priority queue to retain a global ordering of keys, which is similar to the merge step of merge sort. Once keys are ordered, we can then simply select the most recent version for each key and discard the rest without remembering the intermediate results.

7.2 Point Lookup Query

The point lookup query can be viewed as a special case of the range scan query, where the entire primary key is specified such that at most one entry is returned. As an optimization, one can search from newest runs to oldest runs sequentially and stop the search once a match is found. Here we can use exactly the same approach from above to search the single run, where the lower bound and upper bound of sort column values are the same.

For a batch of point lookups, we first sort the input keys by the hash value, equality column values, and sort column values, to improve search efficiency. The sorted input keys are searched against each run sequentially from newest to oldest, one run at a time, until all keys are found or all runs to be searched are exhausted. This guarantees that each run is accessed sequentially and only once.

8 EXPERIMENTAL EVALUATION

In this section, we report the experimental evaluation of Umzi. We first evaluate the index build performance, index query performance, and further study the end-to-end query performance with concurrent data ingestion. We first outline the general experiment setup, then report and discuss the experimental results.

As mentioned in Section 3, there are a distributed cluster of indexer daemons running in Wildfire, each independently responsible for building and maintaining index for one or more Umzi index structure instances (one per table shard). As a result, Umzi scales up and down nicely with more or less indexer daemons. Since the goal of our experiments is to demonstrate the performance of Umzi index structure, we focus on a single shard setting for our experiments.

Note that since all experiments were conducted inside Wildfire, which is closely tied to an IBM product, we cannot report absolute performance numbers. As a result, we report normalized performance numbers, with the normalization process explained for each experiment.

8.1 Experiment Setup

All experiments are performed against a single table shard on a single node using a dual-socket Intel Xeon E5-2680 server (2.40GHZ) with 14 cores in a socket (28 with hyper-threading) and 1.5TB of RAM. The operating system is Ubuntu 16.04.2 with Linux kernel 4.4.0-62. The node uses an Intel 750 Series SSD as the SSD cache. For end-to-end experiments, we use GlusterFS [2] as the shared storage layer.

We use a synthetic data generator to generate keys with include columns used in all experiments, where keys can be sequential or random. Note that our index only stores key and include columns instead of entire records, thus a key generator is sufficient for our experiments. Throughout the experiments, we consider three different index definitions as below:

- I1: one equality column, one sort column, and one include column
- I2: two equality columns and one include column
- I3: one equality column and one include column

Each column is a long type with 8 bytes. Unless otherwise noted, we use index definition I1 as the default case. Each of the following experiments was reported for three times, and the average number is reported.

Moreover, for the scope of this paper, we only focus on the time of index lookups, while omitting the time of retrieving records based on the fetched RIDs, since the latter depends on the record storage format and is orthogonal to the indexing method.

8.2 Index Building Performance

In the first set of experiments, we evaluate the performance of building index runs, which is the primitive operation for Umzi's index maintenance after a groom or post-groom cycle. Figure 8 shows the results for the time it takes to build an index run using the three different index definitions mentioned above as we increase the number of entries in a run. The running time is normalized against the time of building a run with 1000 tuples using I1. As the graph shows, index building almost scales linearly with the number of rows. Furthermore, the index building time for index I3 is always faster than I1 and I2, since I3 has one fewer key column. The impact of the number of indexed columns on the index building time, however, is negligible, compared to the overhead of sorting entries during index building.

8.3 Index Query Performance

Next set of experiments evaluate the performance of querying Umzi under various settings. By default, an index contains 20 runs, where each index run has 100000 entries. We execute index

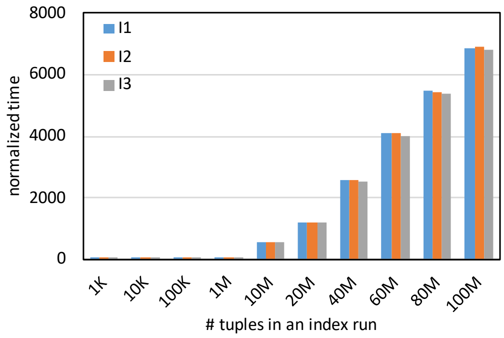


Figure 8: Index Building Performance

queries in a batch, where the default batch size is 1000. All index runs used in this set of experiments are cached in the local SSD.

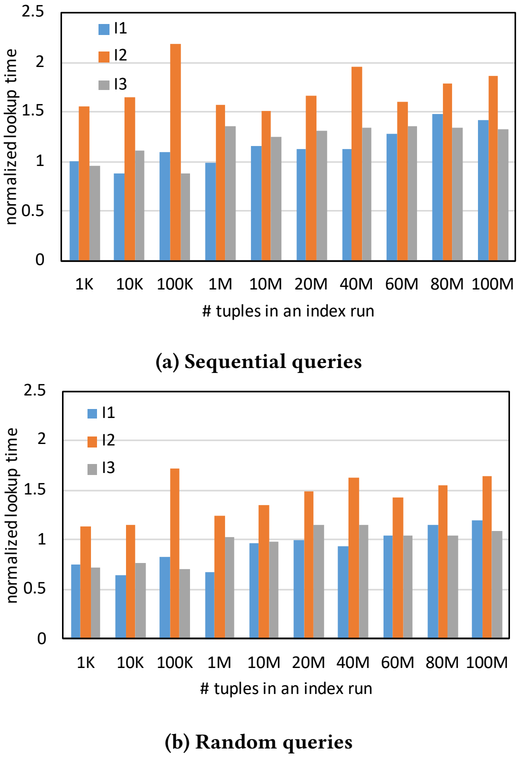
We consider two kinds of entry characteristics for the following experiments, i.e., ingested with sequential keys or random keys. Sequential keys are sequentially generated by our synthetic key generator to simulate the time correlated keys, while random keys are randomly sampled from a uniform distribution without any temporal correlation. We further consider two kinds of key distribution in index queries: sequential and random. As the name suggests, sequential and random queries use sequentially and randomly generated keys in a batch, respectively.

8.3.1 Single Run. We first evaluate the index query performance against a single run. For brevity, we only report experiment results with sequentially ingested keys in this experiment. Since entries in a run are sorted on hash values, there is no difference to use sequentially or randomly ingested data. Figure 9 shows the normalized lookup time with varying run sizes and index definitions.

The time is normalized against the index lookup time of the sequential query over the run with 1000 tuples under the index definition I1. In general, the query time increases with larger index runs, since search keys spread across the run and potentially more I/Os are required to process the query batch when the run size increases. However, the impact of run size is limited because we use the hash offset array to locate the initial search range, and further use binary search to locate the exact location. Moreover, index lookup performance of index definition of I1 is comparable to that of I3, but index lookup performance of index definition I2 is generally slower since I2 contains two equality columns, making the hash offset array less effective in terms of locating the initial search range.

8.3.2 Multiple Runs with Sequential Keys. In this section we evaluate the index query performance against multiple index runs with sequentially ingested keys. We vary the query batch size and the number of index runs for the lookup queries, and the scan range for the range queries. The experiment results are shown in Figure 10.

Figure 10a shows the impact of the batch size on the index lookup performance. The index lookup time per key is normalized against the lookup time of the sequential query with batch size one. In general, sequential queries perform much better than random ones since the run synopsis enables pruning most of the irrelevant runs and leaving only a small fraction of the runs need to be searched (except for the case of batch size 1, where the sequential queries take longer because of some variances in



(a) Sequential queries

(b) Random queries

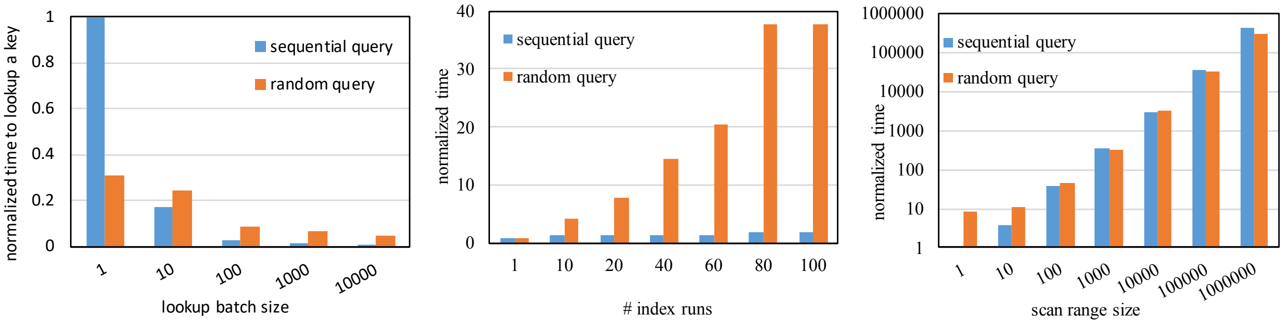
Figure 9: Single Run Query Performance

the experiments). Furthermore, batching greatly improves index lookup performance, since once an index block is fetched into memory for a the lookup of a particular key, no additional I/O is required to fetch that block again for looking up other keys in the batch.

Figure 10b shows the index lookup performance with varying number of index runs. The query time is normalized against the time it takes to complete the sequential query against one run. As the result shows, the number of runs has limited impact over the sequential queries, since most irrelevant runs are simply pruned because of the run synopsis. However, the time of the random queries grows almost linearly with more runs, since more runs need to be searched to complete the batch of lookups.

Finally, the performance of the range scan queries using the priority queue method is in Figure 10c. The time is normalized against the query time of the sequential query with range one. In general, the query time of the range scan query grows linearly with the query range, since larger ranges require more time to read index entries and output matching keys. Moreover, sequential or random ranges have little impact over the query performance, since the time of locating start position is negligible compared to scanning the index entries.

8.3.3 Multiple Runs with Random Keys. After investigating the index query time using sequentially ingested keys, this section evaluates the query performance against multiple index runs with randomly ingested keys. The results are shown in Figure 11. The numbers on the y-axes are normalized the same way as the corresponding numbers in Figure 10. In general, random keys render the run synopsis less useful, which decreases the performance of sequential queries since more runs need to be searched. However, the impact on the random queries is almost negligible,

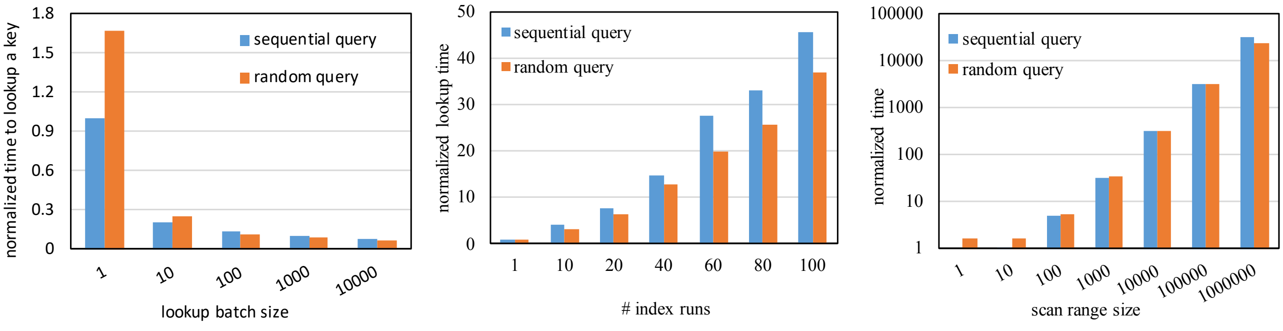


(a) Varying batch size

(b) Varying number of runs

(c) Varying scan ranges

Figure 10: Query performance of multiple runs with sequentially ingested keys



(a) Varying batch size

(b) Varying number of runs

(c) Varying scan ranges

Figure 11: Query performance of multiple runs with randomly ingested keys

since the pruning capability of the run synopses is anyway limited when we have random keys in the query batch. As a result, the performance of sequential queries becomes similar to that of random queries.

8.4 End-to-end Experiments

The last set of experiments evaluates the end-to-end performance of Umzi in Wildfire as we perform data ingest and index lookups concurrently while Umzi's index maintenance operations are also handled in the background. By default, for each experiment, we ingest roughly 100000 random records per second. The groomer runs every second, and the post-groomer runs every 20 seconds. We also submit batches of 1000 random index lookup queries continuously. Each experiment lasts for 100 seconds.

For this set of experiments, we generate data with update rates that mimic a realistic IoT application, where the recent data are updated more frequently. The update rates are calculated based on the groom cycles: the ingested data for the latest groom cycle updates $p\%$ of data from the last groom cycle, and $0.1 \times p\%$ of data from the last 50 cycles, and $0.01 \times p\%$ of data in the last 100 cycles. By default, we set $p\% = 10\%$.

With this experimental setup, we investigate the impact of concurrent readers, percentage of updates, purged runs, and index evolve operations on index lookup time. The results are summarized and discussed below.

8.4.1 Concurrent Readers. Figure 12 shows how varying the number of concurrent readers impact the average index lookup time. Each reader thread submits batches of 1000 lookup queries continuously. For brevity, we show results for only 4, 16, 28, 40, and 52 readers, and the experiment results are normalized against the lookup time with 1 reader from the beginning of the experiment. In addition, Figure 12 zooms into a 30 second period from the middle of the overall experiment to focus on the impact of concurrency as opposed to the index behavior over time as the index grows. As one can see, more concurrent readers have small impact on the query performance, which demonstrates the advantages of Umzi's lock-free design for the readers. The varying performance of the index lookup operation in this graph (and the rest of the graphs in this section) is due to the random input keys we generate for the index lookup requests. Based on the distribution of these random keys, the search for a key can lead to reading fewer or more index runs, which impacts the performance of a point lookup as seen previously.

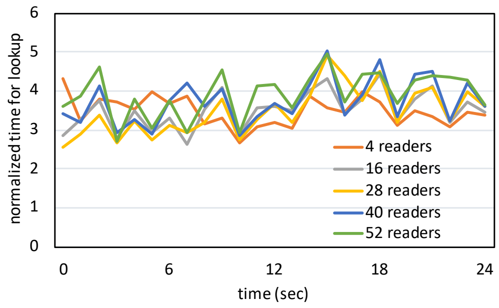


Figure 12: Performance with concurrent readers

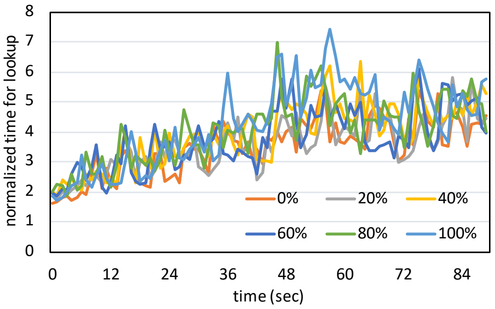


Figure 13: Varying percentage of update workloads

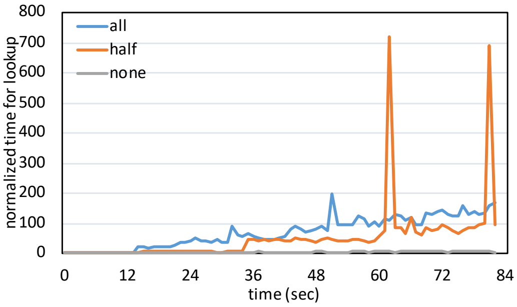


Figure 14: Performance with various purge levels

8.4.2 Updates. Figure 13 analyzes the impact of varying level of updates in a workload on the query performance. We change the update rate p% from 0% (read-only workload) to 100% (all ingested records are updates after the first groom cycle). As the graph demonstrates, updates have limited impact on the average query performance. The slightly increasing lookup time over time, which can be observed in all the experiments in this section, is due to the growing of the index run chain of Umzi.

8.4.3 Purged Runs. The impact of purged runs on the query performance is shown in Figure 14, where we manually set the purge level to control the percentage of purged runs. The running time is normalized against the performance of the no-purge case in the beginning of this experiment. As expected, Figure 14 emphasizes the significance of the SSD cache on the query performance. The latency of the lookup queries is much lower when all the index runs are cached (*none*) compared to the cases where the *half* or *all* of the runs are purged. Moreover, when some runs are purged, we observe unpredictable latency spikes. The reason is that when purged runs are first accessed after being merged or evolved, they have to be fetched from the shared storage into the local SSD cache on a block by block basis as the queries require them.

8.4.4 Index Evolve Operations. Finally, we evaluate the impact of the index evolve operations on the query performance by enabling/disabling the post-groomer. The results are shown in Figure 15. The running time is normalized against the lookup time in the beginning of the experiment where the post-groomer (including index evolution), is enabled. As the graph illustrates, the index evolve operation has certain overhead over the query performance, since often the query may experience several cache misses after runs have been evolved. However, the overhead again is limited, since in the meanwhile the index evolve operation reduces the total of number of runs, which in turn improves the query performance.

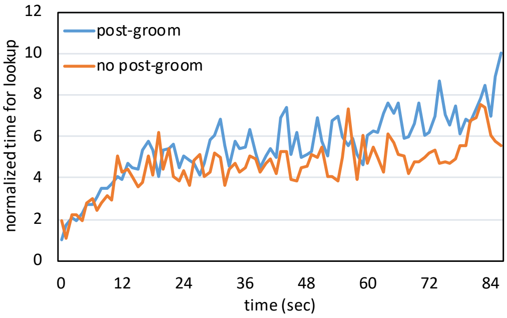


Figure 15: Impact of index evolve operations

9 RELATED WORK

In this section, we survey related work in indexing methods for HTAP systems, as well as LSM-like indexes.

Indexing in HTAP Systems. To satisfy the demand of fast transactions and analytical queries concurrently, many HTAP systems and solutions have been proposed recently. A recent survey of HTAP systems can be found in [32]. In-memory HTAP engines, e.g., SAP HANA [21], HyPer [22], Pelaton [14], Oracle TimesTen [23] and DBIM [29] take unique advantage of large main memories, e.g., random writes and in-place updates, while large-scale HTAP systems that go beyond the memory limit have to face inherently different challenges in the presence of disks and shared storage. MemSQL [7] supports skip-list or hash indexes over the in-memory row store, and LSM-like indexes over the on-disk column store. However, column store indexes cannot be combined with row indexes to provide a unified view for queries. SnappyData [28] only supports indexes over row tables, while providing no indexing support for column tables.

Another category of HTAP solutions typically glue multiple systems together to handle OLTP and OLAP queries. For example, one typical solution is to use a key-value store, such as HBase [3] or Cassandra [1], as the updatable storage layer, while resorting to SQL-on-Hadoop systems such as Spark-SQL [13] to process analytical queries with the help of data connectors. Other systems directly build upon updatable storage engines to handle both transactional and analytical queries, such as Hive [35] on HBase [3] and Impala [16] on Kudu [5]. In these solutions, indexes, if any, are exclusively managed by the storage engine to support efficient point lookups. However, none of these solutions support both fast ingestion and data scans, which is different from our system where ingested data evolves constantly to be more analytics-friendly.

LSM-like Index. The LSM-tree [31] is a persistent index structure optimized for write-heavy workloads. Instead of updating entries in place, which potentially requires a random I/O, the LSM-tree batches inserts into memory and flushes the data to disk using sequential I/O when the memory is full. It was subsequently adopted by many NoSQL systems, such as HBase [3], Cassandra [1], LevelDB [6] and RocksDB [9], for its superior write performance. Many variations of the original LSM-tree have been proposed as well. LHAM [30] is a multi-version data access method based on LSM for temporal data. FD-tree [24] is designed for SSDs by limiting random I/Os and uses fractional cascading [18] to improve search performance. bLSM [34] uses bloom filters to improve point lookup performance, and proposes a dedicated merge scheduler to bound write latencies. AsterixDB [12] proposes a general framework for using LSM-tree as secondary indexes. Ahmad and Kemme [11] present an approach to improve the merge process by offloading merge to dedicated nodes and a cache warm-up algorithm to alleviate cache misses after merge. LSM-Trie [36] organizes runs using a prefix tree-like structure to improve point lookup performance by sacrificing the ability to do range queries. WiscKey [25] reduces write amplification by only storing keys in the LSM tree while leaving values in a separate log. The work [26] presented efficient maintenance strategies for LSM-based auxiliary structures, i.e., secondary indexes and filters, to facilitate query processing. Monkey [19] is an analytical approach for automatic performance tuning of LSM trees. Dostoevsky [20] presents a lazy leveling merge policy for better trade-offs among query cost, write cost and space amplification. Accordion [17] optimizes LSM on large memories by in-memory flushes and merges. SlimDB [33] optimizes LSM-based key-value stores for managing semi-sorted data.

Different from existing work on LSM indexes, which focuses on a key-value store setting, Umzi is an end-to-end indexing solution in distributed HTAP systems. It supports the multi-zone design commonly adopted by large-scale HTAP systems, and evolves itself as data migrates without blocking queries. We further discuss how Umzi is designed to accommodate the multi-tier storage hierarchy, i.e., memory, SSD, and shared storage, to improve storage efficiency.

10 CONCLUSION

This paper describes Umzi, the first unified multi-version and multi-zone indexing method for large-scale HTAP systems in the context of Wildfire. Umzi adopts the LSM-like design to avoid random I/Os in shared storage, and supports timestamped queries for multi-version concurrency control schemes. Unlike existing LSM indexes, Umzi addresses the challenges posed by the multi-zone design of modern HTAP systems, and supports migrating index contents as data evolves from one zone to another. It also utilizes an interesting combination of hash and sort techniques to enable both equality and range queries using one index structure. Furthermore, it fully exploits the multi-level storage hierarchy of HTAP systems for index persistence and caching.

In the future, we plan to extend Umzi to build and maintain secondary indexes in HTAP systems. Then, we would like to perform more experimental evaluation on Umzi to study its performance under various workloads. Finally, we would also like to study other SSD cache management strategies, and evaluate their impact on query performance.

REFERENCES

[1] 2018. Cassandra. http://cassandra.apache.org/.

[2] 2018. GlusterFS. https://www.gluster.org/.

[3] 2018. Hbase. https://hbase.apache.org/.

[4] 2018. IBM DB2 Event Store. //https://www.ibm.com/us-en/marketplace/db2-event-store/.

[5] 2018. Kudu. https://kudu.apache.org/.

[6] 2018. LevelDB. http://leveldb.org/.

[7] 2018. MemSQL. http://www.memsql.com.

[8] 2018. Parquet. https://parquet.apache.org/.

[9] 2018. RocksDB. http://rocksdb.org/.

[10] 2018. Spark. http://spark.apache.org/.

[11] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction Management in Distributed Key-value Datastores. *Proc. VLDB Endow.* 8, 8 (April 2015), 850–861. https://doi.org/10.14778/2757807.2757810

[12] Sattam Alsubaiee et al. 2014. Storage Management in AsterixDB. *PVLDB* 7, 10 (2014), 841–852.

[13] Michael Armbrust et al. 2015. Spark SQL: Relational data processing in Spark. In *SIGMOD*. ACM, 1383–1394.

[14] Joy Arulraj et al. 2016. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*. ACM, New York, NY, USA, 583–598.

[15] Ronald Barber et al. 2017. Evolving Databases for New-Gen Big Data Applications.. In *CIDR*.

[16] MKABV Bittorf et al. 2015. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*.

[17] Edward Bortnikov et al. 2018. Accordion: Better Memory Organization for LSM Key-value Stores. *PVLDB* 11, 12 (2018), 1863–1875.

[18] Bernard Chazelle and Leonidas J Guibas. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica* 1, 1 (1986), 133–162.

[19] Niv Dayan et al. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 79–94.

[20] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *SIGMOD*. 505–520.

[21] Franz Färber et al. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[22] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*. IEEE Computer Society, Washington, DC, USA, 195–206.

[23] Tirthankar Lahiri et al. 2013. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13.

[24] Yinan Li et al. 2010. Tree Indexing on Solid State Drives. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 1195–1206.

[25] Lanyue Lu et al. 2017. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 5.

[26] Chen Luo and Michael J. Carey. 2018. Efficient Data Ingestion and Query Processing for LSM-based Storage Systems. *CoRR* abs/1808.08896 (2018). arXiv:1808.08896

[27] Chen Luo and Michael J. Carey. 2018. LSM-based Storage Techniques: A Survey. *CoRR* abs/1812.07527 (2018). arXiv:1812.07527 http://arxiv.org/abs/1812.07527

[28] Barzan Mozafari et al. 2017. SnappyData: A Unified Cluster for Streaming, Transactions and Interactive Analytics.. In *CIDR*.

[29] Niloy Mukherjee et al. 2015. Distributed Architecture of Oracle Database In-memory. *PVLDB* 8, 12 (2015), 1630–1641.

[30] Peter Muth et al. 2000. The LHAM Log-structured History Data Access Method. *The VLDB Journal* 8, 3-4 (Feb. 2000), 199–221.

[31] Patrick O'Neil et al. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385.

[32] Fatma Özcan et al. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *SIGMOD*. ACM, 1771–1775.

[33] Kai Ren et al. 2017. SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data. *PVLDB* 10, 13 (2017), 2037–2048.

[34] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[35] Ashish Thusoo et al. 2010. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*. IEEE, 996–1005.

[36] Xingbo Wu et al. 2015. LSM-trie: an LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 71–82.