Storage Management in AsterixDB

Sattam Alsubaiee [#1], Alexander Behm [*2], Vinayak Borkar [#1], Zachary Heilbron [#1], Young-Seok Kim [#1], Michael J. Carey [#1], Markus Dreseler [+3], Chen Li [#1]
[#] Department of Computer Science, University of California, Irvine,
[*] Cloudera, [+] Hasso Plattner Institute Potsdam
[1] {salsubai,vborkar,zheilbro,youngsk2,mjcarey,chenli}@ics.uci.edu,
[2] alex.behm@cloudera.com, [3] markus.dreseler@student.hpi.uni-potsdam.de

ABSTRACT

Social networks, online communities, mobile devices, and instant messaging applications generate complex, unstructured data at a high rate, resulting in large volumes of data. This poses new challenges for data management systems that aim to ingest, store, index, and analyze such data efficiently. In response, we released the first public version of AsterixDB, an open-source Big Data Management System (BDMS), in June of 2013. This paper describes the storage management layer of AsterixDB, providing a detailed description of its ingestion-oriented approach to local storage and a set of initial measurements of its ingestion-related performance characteristics.

In order to support high frequency insertions, AsterixDB has wholly adopted Log-Structured Merge-trees as the storage technology for all of its index structures. We describe how the AsterixDB software framework enables "LSM-ification" (conversion from an in-place update, disk-based data structure to a deferred-update, append-only data structure) of any kind of index structure that supports certain primitive operations, enabling the index to ingest data efficiently. We also describe how AsterixDB ensures the ACID properties for operations involving multiple heterogeneous LSM-based indexes. Lastly, we highlight the challenges related to managing the resources of a system when many LSM indexes are used concurrently and present AsterixDB's initial solution.

1. INTRODUCTION

Social networks, online communities, mobile devices, and instant messaging applications are generating digital information at an increasing rate. Facebook has reported that the average number of content items shared daily as of May 2013 is 4.75 billion [13], while Twitter users are posting around 500 million tweets daily [29]. The growth of such data poses challenges for data-management systems to process such large amounts of data efficiently.

Traditional relational databases usually employ conventional index structures such as B^+-trees due to their low read latency. However, such traditional index structures use in-place writes to perform updates, resulting in costly random writes to disk. Today's emerging applications often involve insert-intensive workloads for which the cost of random writes prohibits efficient ingestion of data. Consequently, popular NoSQL systems such as [2, 4, 5, 11, 26] have adopted Log-Structured Merge (LSM) Trees [22] as their storage structure. LSM-trees amortize the cost of writes by batching updates in memory before writing them to disk, thus avoiding random writes. This benefit comes at the cost of sacrificing read efficiency, but, as shown in [26], these inefficiencies can be mostly mitigated.

Furthermore, much of today's newly generated data contains rich data types such as timestamps and locations in addition to textual content—especially due to the increasing adoption rate of smart phones. If such data is to be processed in real-time, then efficient methods for ingesting and searching the data are required. However, like the B^+-tree, conventional index structures that support rich types (e.g., the R-tree [14] for spatial data) do not scale well for such insert-intensive workloads since they are also update-in-place structures. Therefore, it is desirable to have a unified system that can efficiently handle such data with rich types.

In this paper, we present the storage system implemented in the AsterixDB BDMS (Big Data Management System) [1, 6, 8], which is such a unified system. The AsterixDB project, initiated in 2009, has been developing new techniques for ingesting, storing, indexing, and analyzing large quantities of semi-structured data and has been recently open-sourced in beta form. AsterixDB integrates ideas from three distinct areas, namely semi-structured data, parallel databases, and data-intensive computing, in order to provide a next-generation, open-source software platform that scales by running on large, shared-nothing commodity computing clusters.

Within AsterixDB's storage system is a general framework for converting conventional indexes to LSM-based indexes, allowing higher data ingestion rates. We show that converting an index without a total order (e.g., R-trees) to an LSM index is non-trivial if the resultant index is expected to have performant read and write operations. The framework acts as a coordinating wrapper for existing, non-LSM indexes so that designing and implementing specialized indexes from scratch can be avoided, saving development time.

We also show how to enforce the ACID properties across multiple heterogeneous LSM indexes and how AsterixDB ensures atomic update operations across a primary index and any number of secondary indexes. We also describe AsterixDB's novel yet simple recovery method based on both logical logging and index-component shadowing. This recovery scheme combines a no-steal/no-force buffer management policy with the write-ahead-logging (WAL) protocol so that crash recovery requires only redo—and not undo—operations. Moreover, the framework's concurrency control scheme, based on two-phase locking (2PL), is general enough to be applied to any type of secondary index since locking there is not required.

Lastly, we identify some challenges of managing the memory and disk resources of the system with a large number of LSM in-

Proceedings of the VLDB Endowment, Vol. 7, No. 10

dexes running concurrently. Most prior work in the literature reports performance results of LSM trees (or systems based on them) for individual LSM trees. Here we describe a number of important resource-management decisions that a system with many LSM indexes must consider in order to offer a good overall performance.

The rest of the paper is organized as follows. First, we provide background information. Next, we describe the framework for converting conventional index structures to LSM indexes, followed by how to enforce the ACID properties across multiple heterogeneous LSM indexes. We then discuss challenges that arise in managing the resources of a system with a large number of LSM indexes. Finally, we present some initial performance evaluation results of AsterixDB focusing on the LSM-based storage system.

2. BACKGROUND

We start by describing the general idea of the LSM-tree and its operations. We then provide an overview of the AsterixDB system, followed by a brief discussion of other related work.

2.1 LSM-tree

The LSM-tree [22] is an ordered, persistent index structure that supports typical operations such as insert, delete, and search. It is optimized for frequent or high-volume updates. By first batching updates in memory, the LSM-tree amortizes the cost of an update by converting what would have been a disk seek into some portion of a sequential I/O.

Entries being inserted into an LSM-tree are initially placed into a component of the index that resides in main memory—an *in-memory component*. When the space occupancy of the in-memory component exceeds a specified threshold, entries are *flushed* to disk. As entries accumulate on disk, the entries are periodically merged together subject to a *merge policy* that decides when and what to merge. In practice, two different variations of flush and merge are used. Block-based, "rolling merges" (described in [22]) periodically migrate blocks of entries from newer components (including the in-memory component) to older components that reside on disk—*disk components*—while maintaining a fixed number of components. On the other hand, component-based flushes migrate an entire component's worth of entries to disk, forming a new disk component, such that disk components are ordered based on their freshness, while component-based merges combine the entries from a sequence of disk components together to form a new disk component. Popular NoSQL systems commonly employ the component-based variation, where in-memory and disk components are usually called *memtables* and *SSTables*, respectively (e.g., in [11]). Throughout the rest of this paper, a reference to an LSM tree implies a reference to a component-based LSM-tree. As is the case in [22], a component is usually a B^+-tree. However, it is possible to use other index structures whose operations are semantically equivalent (e.g., Skip Lists [24]) for the in-memory component.

Disk components of an LSM-tree are immutable. Modifications (e.g., updates and deletes) of existing entries are handled by inserting control entries into the in-memory component. A delete (or "anti-matter") entry, for instance, carries a flag marking it as a delete, while an insert can be represented simply as the new entry itself. Control entries with identical keys must be *reconciled* during searches and merges by either annihilating older entries in the case of a delete, or by replacing older entries with a new entry in the case of an update. During merges, older deleted entries may be safely ignored when forming a new component, effectively removing them from the index.

Entries in an LSM-tree are scattered throughout the sequence of components, which requires range scans to be applied to all of the components. As entries are fetched from the components, the reconciliation described above is performed. A natural design for an LSM-tree range scan cursor that facilitates reconciliation is a heap of sub-cursors sorted on ⟨key, component number⟩, where each sub-cursor operates on a single component. This design temporally groups entries with identical keys, easing reconciliation.

Point lookups in unique indexes can be further optimized. Given key uniqueness in an LSM-tree, cursors can access the components one-by-one, from newest to oldest (i.e., in component number order), allowing for early termination as soon as the key is found. Enforcing key uniqueness, however, increases the cost of an insertion since an additional integrity check is now required: the index must first search for the key that is being inserted. This is in contrast to the typical usage of LSM-trees in popular NoSQL systems, where the semantics of insert usually mean "insert if not exists, else update" (a.k.a. "upsert"), where primary key uniqueness is not maintained. Throughout the rest of the paper, references to *insert* will assume the semantics "insert if not exists, else error if the key exists". As suggested in [27], a Bloom filter can be maintained in main memory for each disk component to reduce the chance of performing unnecessary disk I/O during point lookups, thereby decreasing the cost of performing the integrity check and point lookups in general.

Compared to a B^+-tree, the LSM-tree offers superior write throughput at the expense of reads and scans [16, 22]. As the number of disk components increases, search performance degrades since more disk components must be accessed. It is therefore desirable to keep few disk components by periodically merging multiple disk components into fewer, larger components in order to maintain acceptable search times. We refer interested readers to [26] for a recent study of advanced LSM merging strategies.

2.2 AsterixDB

AsterixDB is a parallel, semistructured information management platform that provides the ability to ingest, store, index, query, and analyze mass quantities of data. It supports use-cases ranging from rigid, relation-like data collections, whose types are predefined and invariant, to more flexible and complex data, where little is known a priori and the data instances are variant and self-describing. For this, AsterixDB has a flexible data model (ADM) that is a superset of JSON and a query language (AQL) comparable to languages such as Pig [21], Hive [7], and Jaql [15]. Through ADM and AQL, AsterixDB supports native storage and indexing of data as well as access to external data (e.g., data in HDFS). AsterixDB uses the Hyracks [9] data-parallel platform as its runtime engine. Hyracks sits at roughly the same level that Hadoop does in implementations of other high-level languages such as Pig, Hive, or Jaql.

2.3 Related Work

Utilizing memory to support efficient write-heavy workloads has also been suggested by the log-structured file system (LFS) [25]. Inspired by the LSM-tree [22], different variants of log-structured B^+-trees have been introduced in the literature and some have been deployed commercially [2, 4, 5, 11]. The bLSM-tree [26] uses advanced scheduling algorithms to provide predictable write performance. LogBase [12] has adopted log-only storage, where the logs are the actual data repository. The Bkd-tree [23] utilized the kd-tree to index multi-dimensional point data to provide efficient data ingestion for spatial workloads.

The partitioned exponential file [16] is the most closely related work to ours. It too offers a generic data structure for write-intensive workloads that can be customized for different types of data. The work described here differs in three ways:

1) We provide a more general approach for converting existing index structures to LSM-based index structures that avoids building specialized indexes from scratch, saving development time.
2) We show, in detail, how to enforce the ACID properties across multiple heterogeneous LSM indexes.
3) We provide a full implementation of the work in AsterixDB as an open-source software package.

There have been efforts to enforce the ACID properties in existing LSM-based storage systems. In many systems, the ACID properties are enforced at the granularity of a single-row operation over a single index [12, 26]. Some systems [3, 4] support secondary indexes with weaker consistency guarantees between the primary data store and the secondary indexes. Our work differs in that we show how to ensure the ACID properties when there are different types of LSM indexes such as LSM-based R-trees and LSM-based inverted indexes and when secondary index consistency is desired.

3. SECONDARY INDEX LSM-IFICATION

This section describes a generic framework for converting a class of indexes (that includes conventional B$^+$-trees, R-trees, and inverted indexes) with certain, basic operations into LSM-based secondary indexes. The framework provides a coordinating wrapper that orchestrates the creation and destruction of LSM components and the delegation of operations to the appropriate components as needed. Using the original index's implementation as a component, we can avoid building specialized index structures from scratch while enabling the advantages that an LSM index provides.

Applying the key ideas behind LSM-trees to index structures other than B$^+$-trees turns out to be non-trivial and must be done carefully in order to achieve a high-write throughput. In particular, we start by explaining why the process of reconciliation of index entries when "LSM-ifying" indexes such as the R-tree and inverted index is challenging.

3.1 Reconciliation Challenges

Entries in a log-structured data structure are descriptions of the state for a particular key. Since modification operations (such as inserts and deletes) on a log-structured data structure always produce a new state (entry) for a particular key, there must be a process for determining the correct state of the key amongst the existing states. This is called *reconciliation*. We define the correct state of the key to be the most recent state since that will provide the usual semantics expected of index operations.

The process of reconciliation was briefly outlined for LSM B$^+$-trees in Section 2.1, whereby entries corresponding to a single key were grouped by virtue of being returned in the native, key order of the underlying index. However, this luxury is only possible when the underlying index return entries grouped by their key. To explore the difficulties of reconciliation for an index lacking this grouping property, we analyze typical operations on an LSM index.

Search and merge operations: When performing a range scan on an LSM index, each component must be searched using the same predicate since matching entries may be distributed across the components. Each entry returned as the product of searching a component must be reconciled since it may or may not be the correct state for a particular key. Recall that in the LSM B$^+$-tree, this reconciliation is simple: in a manner reminiscent of the merge step of merge-sort, simply collect the next entry from each sub-cursor that has a matching key, retaining only the most recent version of the entry and discarding the others. In general, since the entries being returned from an LSM index's sub-cursors may not be totally ordered (e.g., from an R-tree sub-cursor), this approach cannot be used. One possible approach could be to perform a search for every key returned from a sub-cursor on all newer components, but this would be prohibitively expensive since it may incur multiple expensive, multi-path searches in an R-tree or a multitude—one for each token—of searches in an inverted index. Alternatively, all entries can be returned and then sorted using any comparator that guarantees that exactly equal values come together (e.g., byte-based sorting).

Insert and delete operations: Since modification operations on an LSM index produce entries in the in-memory component, it is possible to introduce multiple states for a particular key in a single component. The existence of multiple states in a single component could prohibit the possibility of determining the most-recent state unless additional information is carried in the entries. One example of such information is a sequence number. But, if the overhead of storing sequence numbers with each entry is to be avoided, reconciliation should be performed at modification time to ensure that at most one state per key resides in a single component. In a B$^+$-tree component this is simple: when modifying (e.g., deleting) an entry, the previous version of the entry will be discovered when traversing the tree. Unfortunately, this is not true, for example, in an R-tree since the location of an entry being inserted is not uniquely determined. One possible solution is to perform a search to find any identical keys before each modification, but doing so can greatly reduce the ingestion rate of an LSM R-tree since repeated multi-path, spatial searches will incur overhead. Furthermore, this issue is exacerbated when LSM-ifying an inverted index. Keys in an inverted index are usually ⟨token, id⟩ pairs. Thus, there exists a state for every token inserted into the index. In order to remove a document from the inverted index, the document will need to be retokenized and a delete entry will need to be added for every token that was produced, making both the delete operation and reconciliation during search and merge operations exorbitantly expensive.

3.2 Efficient Reconciliation

The issue of efficiently reconciling can be viewed equivalently as the problem of (in)validating a single entry. We can achieve this by maintaining data and control entries separately, in two different structures. Specifically, new incoming data entries can be batched into the underlying in-memory index, while control entries, representing deleted entries, are batched into an in-memory *B$^+$-tree*, called the *deleted-key B$^+$-tree*. The deleted-key B$^+$-tree is essentially a delete-list that stores the primary keys of the deleted entries. We chose to use a B$^+$-tree, but any data structure that provides similar operations in an efficient manner could be used.

Storing control entries separately avoids expensive multi-path searches for control entries since only the delete-list needs to be searched. For the inverted index, it also avoids expensive tokenization and token control entry insertion for delete operations since only a single entry needs to be made in the deleted-key B$^+$-tree.

An LSM index employing this approach maintains two data structures in memory: the original index structure and the deleted-key B$^+$-tree. Both structures are tightly coupled and will always be flushed to disk together, yielding disk components consisting of the original index structure and a deleted-key B$^+$-tree. As such, we refer to the pair of data structures as a single component. Figure 1 shows a secondary LSM R-tree index employing the above design alongside a secondary LSM B$^+$-tree that does not use the above design. We omit the use of the above design for LSM B$^+$-trees since the B$^+$-tree naturally groups entries, serving as an optimization. Each index has one in-memory component and one disk component. Note that the deleted-key B$^+$-trees are only used to validate

entries through point lookups. Thus, as an optimization, for every disk instance of the deleted-key B$^+$-tree, a Bloom filter containing the entries of the tree is maintained in memory to reduce the need to access pages of the deleted-key B$^+$-tree on disk. Index operations on this new structure are explained in Section 3.4.1.

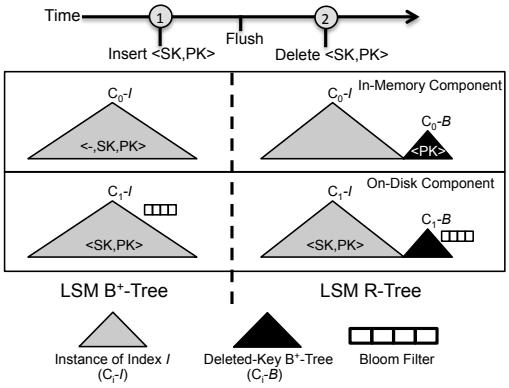


Figure 1: The final state of insertion, flushing, then deletion applied to a secondary LSM R-tree and a secondary LSM B$^+$-tree. Both indexes are storing entries of the form $\langle SK, PK \rangle$, where SK is a secondary key and PK is the associated primary key. The LSM R-tree handles deletion by inserting the primary keys of the deleted entries in its deleted-key B$^+$-tree, while the LSM B$^+$-tree handles it by inserting a control entry, denoted by $\langle -, SK, PK \rangle$, into its memory component.

3.3 Imposing a Linear Order

For a certain class of indexes lacking a total order, it is possible to impose a linear order on the index entries. For example, a Z-order curve or Hilbert curve can be used to impose a linear order on the index entries of an R-tree. This is especially useful for bulk-loading since the ordering can be applied during flushes and merges of LSM components. By doing so, a hybrid approach to efficient reconciliation becomes possible. In this hybrid strategy, incoming inserts and deletes are still maintained in two different in-memory data structures as explained above. However, when flushing the in-memory component to disk, the data and control (anti-matter) entries of the in-memory index structure and the deleted-key B$^+$-tree can be sorted based on the ordering criterion (e.g., Hilbert curve) and merged to form a single disk component that consists of the single, original index structure. This hybrid approach can benefit the performance of the LSM index in many aspects. First, the sorted mini-cursors design can now be used to search the disk components of the index, allowing deleted entries to be ignored on the fly. Second, since the participating disk components are already ordered based on the sorting criteria, the merging process is simple and efficient: scan the component's leaves and output the sorted entries into a new disk component. Third, the newly-formed, merged component will retain the original ordering. For the R-tree, retaining the ordering improves the performances of searches and generally produces a higher quality index as shown in [17].

3.4 LSM Generalization

Based on the above design, we can provide a generic framework to "LSM-ify" the secondary indexes of a system. Let entries be of the form $e = \langle SK, PK \rangle$ where SK is a secondary key and PK is the associated primary key. The framework requires that the following primitive operations be supported by every secondary, non-LSM index that is to be converted into an LSM index:

1. Bulk-load: Given a stream of entries $e_1, \ldots, e_m$, the bulk load operation creates a single disk component of the index. The bulk load operation is used for two reasons: to flush an in-memory component of the index into a new disk component and to merge multiple disk components into a single disk component.
2. Insert: This operation inserts a given entry e into the index.
3. Delete: The delete operation removes a given entry e from the index.

With those primitive operations, we describe the basic operations of the proposed general LSM index design and of the LSM B$^+$-tree, which are both implemented in AsterixDB.

3.4.1 General LSM Index Operations

Insert and delete operations: Since entries in the deleted-key B$^+$-tree refer only to disk components, inserted entries never need to be reconciled with deleted entries upon insertion. Thus, an insertion is performed by inserting an entry $e = \langle SK, PK \rangle$ into the in-memory index without the need to search for control entries in the in-memory deleted-key B$^+$-tree. To complement this, deletes are performed by deleting the given entry directly from the index structure and adding a control entry $e' = \langle PK \rangle$ to the in-memory deleted-key B$^+$-tree.

Search and merge operations: When answering a search query, all components of the LSM index must be examined. An entry $e = \langle SK, PK \rangle$ is part of the result set if it satisfies two conditions:

1. SK satisfies the query predicate, and
2. There does not exist a control entry $e' = \langle PK \rangle$ in the deleted-key B$^+$-tree of a newer component than e's component.

When merging components of an LSM index, the deleted-key B$^+$-trees of the participating components are also searched in the same manner to keep deleted entries out of the merged component.

3.4.2 LSM B$^+$-Tree Operations

Insert and delete operations: Insert operations are preceded by a search to check if an insert entry exists with the same key. If an insert entry already exists, an error will be thrown. Otherwise, the entry $e = \langle SK, PK \rangle$ will be added to the in-memory component. Delete, on the other hand, simply inserts an anti-matter entry $e' = \langle -, SK, PK \rangle$ into the in-memory component. If the in-memory component contains an entry that has the same key as the key being inserted or deleted, then the existing entry is simply replaced with the new entry (except, if the existing entry is an insert entry and the operation to be performed is an insert, which will throw an error, as mentioned above, to enforce primary key uniqueness).

Search and merge operations: When answering a range query, all components of the LSM B$^+$-tree must be examined. An entry $e = \langle SK, PK \rangle$ in a component is part of the result set if it satisfies two conditions:

1. SK satisfies the query predicate, and
2. There does not exist a more recent control entry $e' = \langle -, SK, PK \rangle$.

A point lookup query can be optimized to search components sequentially from newest to oldest until a match is found.

Similarly, when merging multiple components of an LSM B$^+$-tree, the participating components are searched as in the range query to avoid putting deleted entries into the merged component.

3.5 Indexes in AsterixDB

Data in AsterixDB is partitioned using hash-based partitioning on the dataset's primary key. All of the dataset's secondary indexes are local as in most shared-nothing parallel databases. Thus, secondary index partitions refer only to data in the local primary index partition. AsterixDB currently supports LSM-based B$^+$-trees, R-trees[1], inverted keyword, and inverted ngram secondary indexes. Secondary index lookups are routed to all of a dataset's partitions since matching data could be in any partition. These lookups occur in parallel and primary keys are the result of these lookups. The resulting primary keys are then used to lookup the base data from the primary index, sorting the keys first to access the primary index in an efficient manner. Inserting and deleting entries from a dataset requires that each of the dataset's indexes be mutated since AsterixDB maintains consistency across all indexes of a dataset. The primary index is always updated first, followed by updating the secondary indexes, if any exist.

4. PROVIDING RECORD-LEVEL ACIDITY

AsterixDB supports record-level, ACID transactions across multiple heterogeneous LSM indexes in a dataset. Transactions begin and terminate implicitly for each record inserted, deleted, or searched while a given DML statement is being executed. This is similar to the level of transaction support found in today's NoSQL stores. Since AsterixDB supports secondary indexes, the implication of this transactional guarantee is that all the secondary indexes of a dataset are consistent with the primary index.

AsterixDB does not support multi-statement transactions, and, in fact, a DML statement that involves multiple records can itself involve multiple independent record-level transactions. A consequence of this is that, when a DML statement attempts to insert 1000 records, it is possible that the first 800 records could end up being committed while the remaining 200 records fail to be inserted. This situation could happen, for example, if a duplicate key exception occurs as the 801st insertion is attempted. If this happens, AsterixDB will report the error as the result of the offending DML insert statement and the application logic above will need to take the appropriate action(s) needed to assess the resulting state and to clean up and/or continue as appropriate.

AsterixDB utilizes a no-steal/no-force buffer management policy and write-ahead-logging (WAL) to implement a recovery technique that is based on LSM disk component shadowing and index-level logical logging. During crash recovery, invalid disk components are removed and only the committed operations from in-memory components need to be selectively replayed. Log-based structures, due to their shadowing nature, create immutable components that are more amenable to replication than structures that do in-place modifications, providing more opportunity for us to explore fault tolerance when servers fail. Currently, we do not do anything to handle server failures and will address this in future work.

4.1 Concurrency Considerations

AsterixDB's concurrency control is based on two-phase locking (2PL) and follows the latch protocols described in ARIES/KVL [19] for the B$^+$-tree and GiST [18] for the R-tree. Transaction locks are only acquired on primary keys when accessing a primary index. Locks are never acquired when accessing a secondary index, which could lead to inconsistencies when reading entries of a secondary index that are being altered concurrently. To prevent these inconsistencies, secondary index lookups are always validated when fetching the corresponding records from the primary index.

Keys are only locked during insert, delete, and search operations. Flushing an in-memory component and merging disk components do not set locks on the entries of the components, nor do they generate log records.

4.2 Maintaining No-Steal

In a no-steal policy, uncommitted data is not allowed to be persisted. As updates fill the in-memory components of LSM indexes, memory pressure grows. This pressure must eventually be released by flushing the in-memory component to disk. Thus, in order to maintain the no-steal policy, AsterixDB prevents flushing an in-memory component until all uncommitted transactions that have modified the component are completed. Further, the system prevents new transactions from entering an in-memory component that is full until it is flushed, reset, and able to accommodate them. This is done by maintaining a reference count on each in-memory component that is incremented and decremented when a transaction enters and exits a component, respectively. A transaction enters a component before it performs an operation on the component. Then, when the transaction is committed (e.g., a commit log record is written), the transaction exits the component. A reference count of zero implies that there are no active transactions in the component, hence it is safe to flush. As an optimization, AsterixDB maintains two reference counts: one for write transactions and one for read transactions. Read transactions are always permitted to enter an in-memory component, regardless of whether the component is full or not (except for a very brief duration when the component is reset). Write transactions, however, continue to follow the stricter rules that were described when using a single reference count.

4.3 Abort

AsterixDB employs *index-level logical logging*, i.e., a single update operation in an LSM index generates a single log record. The logical log record format for LSM operations is shown in Figure 2(b). To undo an aborted transaction, all of its log records are read in reverse order via the previous LSN and are undone accordingly. Undoing an operation involves applying the inverse operation. For example, to undo a delete, an insertion is performed (and vis-versa). Unlike ARIES, CLRs are unnecessary during undo operations since we follow a no-steal policy; there is no possibility of partial effects in a component when aborting a transaction.

4.4 Crash Recovery

Whenever a new disk component is created, through a flush or merge operation, a *validity* bit is atomically set in a metadata page of the component to indicate that the operation has completed successfully. During crash recovery, any disk component with an unset validity bit is considered invalid and removed, ensuring that the data in the index is physically consistent.

By using a no-steal policy, only committed operations from in-memory components are selectively replayed during crash recovery and there is no need for an undo phase, unlike the repeating history step in ARIES [20]. Operations' idempotence is guaranteed by comparing the log sequence numbers (LSNs) of log records with an *index LSN*—an LSN that indicates the last operation that was applied to the index. This concept is reminiscent of the page LSNs used by ARIES to ensure idempotence. Each component of an index maintains a *component LSN* that indicates (in a similar manner to the index LSN) the last operation that was applied to the com-

[1] We implemented two versions: one that keeps anti-matter entries inside the disk R-trees (AMLSM R-tree), and another that uses a deleted-key B$^+$-tree with every disk component for maintaining control entries (LSM R-tree). We arbitrarily chose the LSM R-tree to be the default spatial index in AsterixDB, but we intend to switch to the AMLSM R-tree as a result of the performance evaluations performed for this paper.

ponent. The maximum value of all component LSNs of an index is the index LSN.

Recovery starts from a checkpoint file which indicates the LSN of the first log record that is needed for recovery—the *low water mark*. Checkpoint files are created the very first time the system is started, after each successful run of recovery, and when the system is shutdown, in order to reduce the time of future recovery runs. There are two phases in a post-recovery checkpoint. The first phase is to flush all in-memory components having redo effects to disk, and the second phase is to create a checkpoint file that holds the LSN where the first log record will be after recovery.

Crash recovery in AsterixDB is also two phased: an analysis phase followed by a redo phase. The analysis phase reconstructs committed transactions by reading log records in the forward direction starting from the log record indicated by the checkpoint LSN. The redo phase then replays the effects of the committed transactions as follows. First, log records are read in the forward direction starting from the smallest LSN created by the committed transactions. A given redo log record's operation is performed only if the transaction identified by the transaction ID had committed and the LSN of the log record is greater than the index LSN of the index identified by the index ID in the log record. The first condition guarantees the durability of committed transactions' effects and the second condition guarantees the idempotence of the redo operation by not redoing an effect made to disk already.

4.5 An Example

Figure 2(a) depicts the lifecycle of inserted and deleted records. In this example, there is a dataset with records consisting of three fields: Id (an integer key for the primary LSM B$^+$-tree, named $PIdx$), Loc (a two-dimensional point key for the secondary LSM R-tree, named $SIdx$), and $Name$. A component of $PIdx$ is denoted as PC_i, where PC_0 represents the in-memory component and PC_1 represents the oldest disk component. A component of $SIdx$ consists of an R-tree and a deleted-key B$^+$-tree, each of which is denoted as SC_i and BC_i as is done for the primary index. Older disk components have smaller values of i.

The timeline shows a sequence of five consecutive operations and their side-effects (e.g., flush and merge), where each operation is executed by a single transaction, starting from T1 to T5. Each panel in Figure 2(a) represents the final states of the indexes after the operation or side-effect is performed. For expository purposes, we use a flush policy that flushes in-memory components when the number of in-memory entries is two and a merge policy that merges disk components of an index when the number of disk components is two (other policies are described in Section 5). The size of each log record generated from each index is a fixed 40 bytes and the log does not contain any pre-existing log records.

The initial state of the indexes is empty before T1 is executed. Transaction T1 inserts the first record as follows. First, to enforce the primary key uniqueness constraint, T1 must ensure that the key $\langle 1 \rangle$ does not exist in the primary index by performing a search operation. If no match is found, an exclusive lock (X-lock) is acquired on the key and an index-level logical log record is generated that includes the index ID, operation type, entry image, etc. The log record's LSN is 0 since it is the first log record. Since the operation was applied to component PC_0, the component's LSN is set to 0 (denoted by the square bracket under the component).

Next, the entry $\langle 1, (10, 10), \text{“}Kim\text{”} \rangle$ is inserted into the in-memory component of the primary index, PC_0, followed by the insertion of the secondary index entry $\langle (10, 10), 1 \rangle$ into the in-memory component of the secondary index, SC_0 (for which no additional locking is required), which is preceded by writing the second log record whose LSN is 40. The LSN 40 becomes the component LSN (recall that log records are 40 bytes long). The result is shown in the right-hand side of panel T1. Also, the log records created by transaction T1 in Figure 2(a) are shown in Figure 2(c). The previous LSN of the first log record is set to -1, indicating that there is no prior log record generated by T1. The IDs $PIdx$ and $SIdx$ are the IDs of the primary and the secondary indexes, respectively.

Finally, a commit log record is written, the log buffer is flushed to disk to enforce WAL, and all locks held by T1 (the X-lock on key $\langle 1 \rangle$) are released. The two fields of the commit log record (not shown in the figure) of T1 would be set to $\langle T1, commit \rangle$.

The second record $\langle 2, (20, 20), \text{“}Sam\text{”} \rangle$ is inserted by T2 in the same way as in T1. The initial outcome of executing T2 is shown in the panel marked as T2-A. Since the memory budget for the memory components of PC_0 and SC_0 has been exhausted, both components are now flushed to disk, as shown in T2-B (also notice the Bloom filter that has been created for the flushed B$^+$-tree).

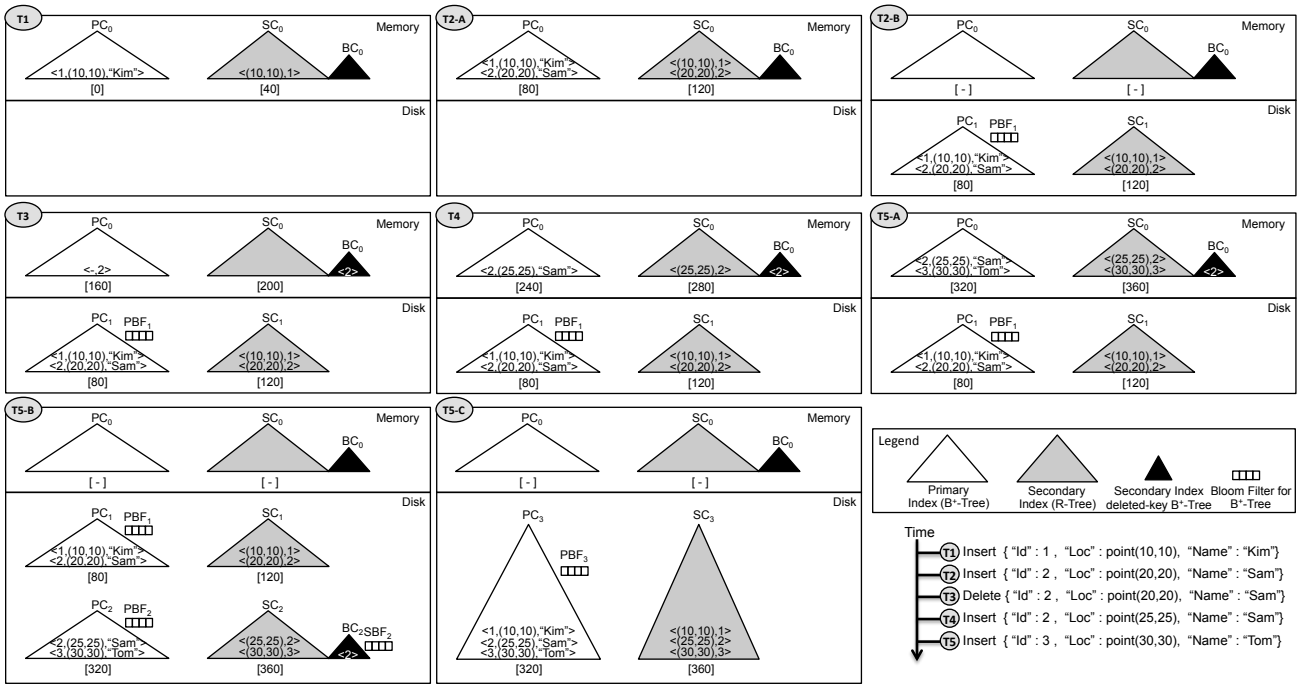
Now, a delete operation is issued by T3 to delete records with coordinates that fall within a radius of 5 degrees from a point with coordinates $\langle 22, 22 \rangle$. The delete operation is transformed to two consecutive operations: 1) find all records satisfying the delete condition and 2) delete the qualified records. In this example, AsterixDB's query optimizer will choose the R-tree as the access method to quickly identify all qualified records. Each retrieved entry must be validated through a primary index lookup. In AsterixDB, validation starts only after all the candidate qualified results are retrieved from the secondary index. This barrier-style validation is used to avoid complex solutions that would involve verifying that the entries in a secondary index page are still valid when consecutively acquiring and releasing the page's latch during a search operation. This validation solution is analogous to *revalidation after unconditional locking* in [19]. In AsterixDB, the barrier-style validation comes for free since the qualified candidate results from any secondary index are sorted (which is a blocking operation) on primary key before probing the primary index.

As shown in T3 of Figure 2(a), the entry $\langle (20, 20), 2 \rangle$ in SC_1 is going to be retrieved by the delete, followed by a primary index lookup for key $\langle 2 \rangle$, where transaction T3 acquires a shared lock (S-lock) on the primary key. Since the qualified record has been identified, now it must be deleted as follows. First, the S-lock on primary key $\langle 2 \rangle$ is upgraded to an X-lock and a log record is written for the primary index. Then, the control entry $\langle -, 2 \rangle$ is inserted into PC_0. Notice that the control entry in the primary index includes only the delete flag and the key without the associated payload. After that, a log record for the secondary index is written and the entry $\langle 2 \rangle$ is inserted into BC_0.

Figure 2(d) shows log records created from T3's delete operation. Notice that although the secondary index entry is inserted into the deleted-key B$^+$-tree, BC_0, the index ID of the second log record in the figure is set to $SIdx$; this is because our logging method is indeed index-level logical logging. Remembering the assumption that log records are 40 bytes long, the previous LSN of the second log record is 160 because the first log record of T3 is the fifth log record overall.[2] Finally, the commit log is written and flushed and the X-lock held by T3 on $\langle 2 \rangle$ is released.

Next, transaction T4 inserts a record $\langle 2, (25, 25), \text{“}Sam\text{”} \rangle$. Its primary key is equal to the deleted record key in the third operation, but the point now has different location coordinates. When the corresponding entry is inserted into PC_0, the control entry in PC_0 is replaced by the new record as shown in the figure. In contrast, the entry $\langle 25, 25, 2 \rangle$ is inserted into SC_0 without deleting the entry

[2] Commit log records are excluded in the LSN counting in this mock example.



(a) A Running Example.

Previous LSN	Transaction ID	Log Type	Index ID	Operation Type	Entry

(b) The Log Record Format.

-1	T1	Update	PIdx	Insert	<1,(10,10),"Kim">
0	T1	Update	SIdx	Insert	<(10,10),1>

(c) Log Records of T1.

-1	T3	Update	PIdx	Delete	<2,(20,20),"Sam">
160	T3	Update	SIdx	Delete	<(20,20),2>

(d) Log Records of T3.

Figure 2: A running example, the log record format, and example log records.

$\langle 2 \rangle$ from BC_0 (recall that entries in BC_0 only refer to disk components). The locking and logging actions performed on behalf of T4 will be similar to previous transactions.

Finally, transaction T5 inserts a record $\langle 3, (30, 30), \text{“}Tom\text{”} \rangle$ as shown in T5-A, which triggers a flush operation as shown in T5-B. Based on the used merge policy, the two disk components are now merged into one as shown in T5-C. In PC_3, the old entry $\langle 2, (20, 20), \text{“}Sam\text{”} \rangle$ in PC_1 has been superseded by the recent entry $\langle 2, (25, 25), \text{“}Sam\text{”} \rangle$ in PC_2 and PBF_3 has been created for the new disk component PC_3. Component SC_3 does not have an entry $\langle 20, 20, 2 \rangle$, as it was removed by the control entry in BC_2. Lastly, component BC_2 and its associated Bloom filter were removed since all deleted keys were consolidated during the merge.

5. RESOURCE MANAGEMENT

In this section, we discuss the challenges of managing the memory and disk resources of a system having a large number of LSM indexes running concurrently. We argue that such systems must address a number of important resource-management policies in order to offer good overall performance. We also outline AsterixDB's first cut on how to handle this complex problem.

5.1 Problem Space

As shown in Figure 3, memory in the AsterixDB system is divided into three different classes:

- *Working memory*, which is used during query processing for performing operations like sorts, joins, and aggregations.
- *Buffer cache*, which is used for buffering disk pages of LSM components.
- *In-memory component memory*, which is used for buffering in-memory components before they are flushed to disk.

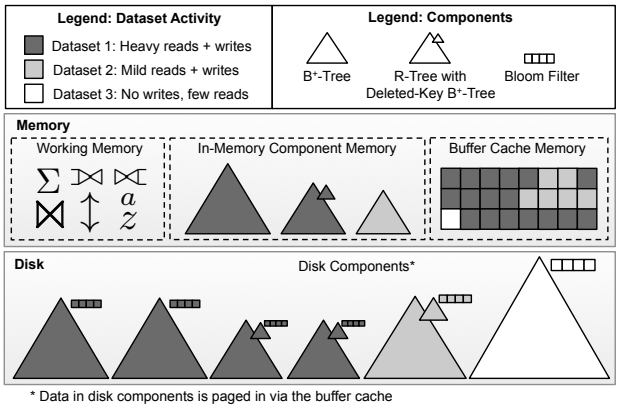


Figure 3: AsterixDB memory classes.

Deciding how much memory to allocate to each class–and when–is non-trivial and workload-dependent. This is true for non-LSM based systems as well ([10, 28]) and is exacerbated by in-memory components of LSM indexes. During complex analytical queries, for example, it would be beneficial to have memory allocated to the buffer cache for reading data from disk and to working memory for processing the data. However, during heavy data ingestion, it would be beneficial to allocate memory for buffering updates to the mutable components of the LSM indexes.

In a multi-LSM system like AsterixDB, there exists the additional complexity of further subdividing the mutable LSM component memory between each of the LSM indexes. Both the size and

rate of updates entering the LSM indexes may vary from index to index and may vary with time. The sizing of LSM components greatly affects the performance of the system: sizing a component too small causes the benefit of batching to be diminished, while sizing a component too large constrains the remaining resources of the system, including, possibly, the size of other LSM components. Furthermore, a system with many LSM indexes must deal with many concurrent and potentially large I/O requests. Scheduling too many I/O requests on the same disk may reduce the amount of sequential I/O that the batching effect is supposed to provide.

Contrast this with popular NoSQL systems that do not need to process complex queries: the only decision that needs to be made is how much memory to give to each machine for buffering in-memory components and, equivalently, how much memory to give to the file system for caching disk resident data. In such a system, there are essentially only two classes of memory: the LSM component memory and buffer cache memory. Working memory is non-existent since the system's APIs do not offer the ability to express sorts, joins, and aggregations.

5.2 Current Implementation

The first version of AsterixDB is designed with these memory considerations in mind, but implements a simple yet configurable approach to resource management, deferring a thorough investigation to future work. However, the current implementation is described here for the interested reader.

Working memory: Sort, join, and aggregation buffer sizes are statically configured to a constant size per memory-needing operator. Whenever such an operation is performed, the configured amount of memory is allocated to be used. Operations adhere to these budgets and do spilling to disk as needed in order to do so.

Buffer cache memory: Data of disk components reside in paginated structures. In order to read data from a disk component, the appropriate pages must be brought into memory. This is done through the use of a traditional buffer cache using a clock replacement policy. The size of the buffer cache is statically configured to a constant size and is allocated at system startup.

LSM component memory: Since each of the indexes in AsterixDB is LSM based, each mutation of a dataset will require updating its indexes by batching the updates in memory. Therefore, some portion of memory must be dedicated to hold the in-memory components of each index. In AsterixDB, datasets are statically configured with a constant amount of memory. Whenever a dataset is accessed, it must first be activated if it is not already. When the dataset is activated, the statically defined amount of memory is allocated and split amongst each dataset's indexes uniformly. Activating too many datasets could cause the system's available memory to be exhausted. AsterixDB provides an additional configuration constant that dictates the maximum amount of memory to be used for active datasets. In the event that this threshold is reached and a dataset needs to be activated, the system will attempt to deactivate another unused dataset using a least recently used (LRU) eviction policy. This strategy is analogous to what is done with pages in the buffer cache, except at the granularity of a dataset.

Using datasets as the granularity at which memory allocation takes place makes the most sense for a write-intensive dataset that sees frequent updates to all partitions. This scenario will provide good memory utilization since writing to all partitions of a dataset entails keeping all indexes (including secondary indexes) open and up-to-date. However, there may exist multiple partitions of a dataset on a particular machine in order to exploit disk-level parallelism. Therefore, even a small (e.g., one record) write will cause an entire dataset's worth of memory to be reserved, even though only $1/n^{\text{th}}$ of it is required (where n is the number of local partitions) since the write will only be routed to one partition.

In AsterixDB, a flush operation is scheduled whenever a dataset's memory allocation becomes occupied and all transactions that mutated the exhausted memory component have been committed. Currently, AsterixDB provides three different merge policies that can be configured per dataset: *constant*, *prefix*, and *no-merge*. The constant policy merges disk components when the number of components reaches some constant number k, which can be configured by the user. While the prefix policy (the default for AsterixDB) relies on component sizes *and* the number of components to decide which components to merge. Specifically, it works by first trying to identify the smallest ordered (oldest to newest) sequence of components such that the sequence does not contain a single component that exceeds some threshold size M and that either the sum of the component's sizes exceeds M or the number of components in the sequence exceeds another threshold C. If such a sequence of components exists, then each of the components in the sequence are merged together to form a single component. Finally, the no-merge policy simply never merges disk components. AsterixDB also provides a DML statement that allows compacting a dataset and all of its indexes by merging all the disk components of the indexes.

All flush and merge operations are submitted asynchronously to a global scheduler for that machine. The scheduler has visibility into which devices will be read and written to by a particular I/O operation. Thus, in theory, it has the ability to schedule these operations based on some efficient strategy. The global scheduler that is currently implemented simply schedules all I/O requests immediately, whenever they are submitted to the scheduler. We leave exploring advanced scheduling polices for future work.

6. PRELIMINARY EVALUATION

This section shows results from an initial experimental evaluation of AsterixDB's storage engine. Section 6.1 evaluates the system as a whole while 6.2 is a micro-benchmark that evaluates the "LSM-ification" framework from the perspective of an R-tree.

6.1 AsterixDB's Storage System

The following experiments demonstrate the scalability of AsterixDB's data-ingestion while varying the number of nodes in the cluster and the number of indexes being used. We show the performance impacts that different merge policies have for data ingestion and queries. In addition, we show the performance of range queries for different operating regions when using the default merge policy of AsterixDB, namely the prefix policy.

To evaluate the performance of data ingestion, we used a feature of AsterixDB called *data feeds*, which is a mechanism for having data continuously arrive into AsterixDB from external sources and to have that data incrementally populate a managed dataset and its associated indexes. We mimicked an external data source, Twitter, by synthetically generating tweets that resemble actual tweets. The synthetically generated tweets have fields such as user, message (the tweet itself), sending time, sender location, and other relevant fields. The tweets are generated in the Asterix Data Model (ADM) format with monotonically increasing 64-bit integer keys. The average size of a tweet was 1KB.

For each experiment, we used two sets of machines. The first set of machines was used to generate the synthetic tweets which are sent over the network to the second set of machines (the AsterixDB cluster) for ingestion. We empirically determined the minimum number of machines that can saturate a single-machine AsterixDB instance, in terms of transactions per second (TPS), and then used this number to scale the number of data-generation machines when

conducting a multi-machine AsterixDB experiment. The second set of machines is a cluster of eight IBM machines used to host an AsterixDB instance, each with a 4-core Xeon 2.27 GHz CPU, 12GB of main memory, and four locally attached 10,000 rpm SATA drives. Of the available 12GB, AsterixDB is given 6GB while the remaining free memory is locked in order to disable the OS's file system buffer cache. In each participating machine, we dedicated one disk to be used by the transaction log manager for writing its log records, while the three remaining disks are dedicated as data storage disks. The three storage disks are used as separate partitions of the tweets dataset by AsterixDB, hosting their associated indexes. Thus, a dataset in an 8-machine AsterixDB instance is partitioned into 24 partitions.

Currently in AsterixDB, all partitions of any one dataset and its indexes in a machine share the same memory budget for in-memory components (divided equally across the indexes). This implies that when the memory component of index in a partition is declared to be full, all memory components of the indexes for the same dataset in that machine are also declared to be full. Therefore, multiple consecutive flush requests are sent to the I/O scheduler for flushing the memory components of that dataset and its indexes for all the partitions located on the machine.

Table 1 shows the configuration parameters used throughout the experiments in 6.1, including the threshold value(s) of each merge policy described in Section 5. Unless specified, we used the prefix merge policy and the length of each experiment is 20 minutes, starting from an empty dataset. Finally, in all experiments, we used two memory components per index (i.e., double buffering) to avoid stalling ingestion during flushes.

Parameter	Value
Memory given for a dataset and its indexes in a machine	1GB
Data page size	128KB
Disk buffer cache size	3GB
Bloom filter target false positive rate	1%
Memory allocated for buffering log records (log tail)	16MB
Max component size of prefix merge policy	1GB
Max component count of prefix merge policy	5
Max component count of constant merge policy	3

Table 1: Settings used throughout these experiments.

6.1.1 Scalability

Figure 4 shows the average ingestion TPS as the number of nodes in the cluster is varied with different combinations of secondary indexes. In particular, Figure 4(a) shows the TPS when ingesting tweets into a dataset that has no secondary indexes ($NoIndexes$), or has one of a single, secondary LSM B$^+$-tree, LSM R-tree, LSM inverted keyword, or LSM inverted ngram index ($LSMB^+Tree$, $LSMRTree$, $LSMKeyword$, and $LSMNGram$, respectively).

As the number of nodes and offered workload increase, we observe near-linear increase in ingestion throughput. A dataset with no secondary indexes always achieves the highest TPS for the obvious reason that there is no overhead of maintaining secondary indexes. Adding a secondary LSM B$^+$-tree to the dataset reduces its ingestion rate, and the reduction is higher as we add more complex secondary indexes such as an LSM R-tree or an LSM inverted index. Clearly, the LSM ngram index is the most expensive index of the four secondary indexes since it requires gram-tokenizing all of the strings in a tweet followed by inserting the resulting ⟨token, id⟩ pairs into the index.

Figure 4(b) shows the TPS for a dataset with all of the four supported secondary indexes in AsterixDB in use. The results show that the upper bound on TPS is determined by the least upper bound of all of the indexes (here the $LSMNGram$).

Figure 4(c) shows the effect of adding additional secondary indexes to a dataset in a cluster of 8 machines. When an additional secondary LSM B$^+$-tree is added to the dataset, ingestion throughput dropped near-linearly due to the overhead of maintaining consistency between the primary and the secondary indexes.

6.1.2 Effects of Merge Policies

This section shows how three different merge policies–prefix, constant, and no-merge–affect ingestion throughput. In Table 2, we report the TPS and its ratio to the prefix policy for each merge policy. In addition, we show the average number of flush and merge operations that were performed per partition.

Regardless of the merge policy, the average flush count per partition is always proportional to the TPS. This is due to the fact that given the same amount of data to ingest and the same in-memory component size, a higher TPS indicates the memory component is always filled faster, incurring more flush operations.

Using the prefix merge policy yields the highest TPS for two primary reasons. First, it avoids merging large components by ignoring those that are larger than the size threshold M = 1GB. In addition, it always tries to find the *smallest* sequence of components, whose total size is larger than M, to be considered for a merge. Second, the policy avoids stacking too many small components through the use of the predefined count threshold, $C = 5$. When the policy fails to find a sequence of components based on size consideration, it still considers merging small components when their count exceeds C. Reducing the number of small disk components improves the performance of both inserts (by reducing the cost of enforcing the primary key uniqueness constraint) and search queries. Recall that to ensure primary key uniqueness, each insert into the primary index must be preceded by searching all components of the index for a duplicate. Thus, as more disk components are accumulated, the cost of maintaining uniqueness increases since more and more components must be searched.

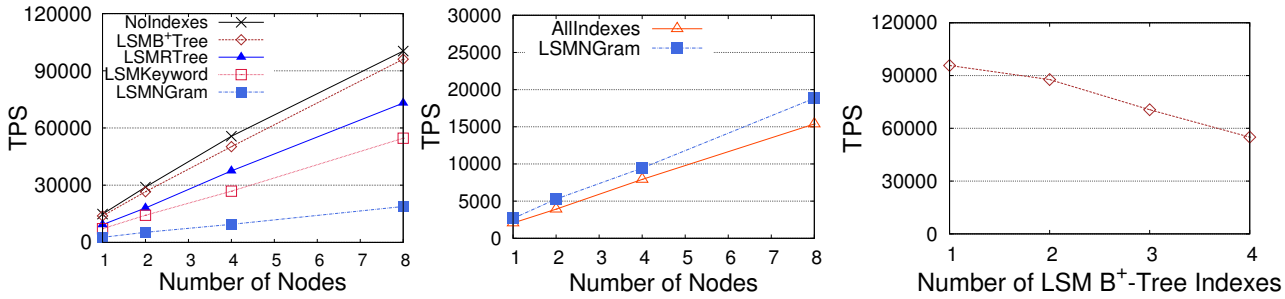
When the constant policy is used, we observe that the TPS is lower than the TPS achieved by the prefix policy. This is because the constant policy never allows the number of disk components to exceed three, resulting in fewer large disk components after merging. Also without considering a component's size, the constant policy will merge small components with large components, resulting in all previously ingested data being read and written to/from disk, so as more data is ingested, merges become more and more costly.

Merge Policy	TPS	TPS Ratio to Prefix	Flush Count	Merge Count
Prefix	102,232	1.00	47	12
Constant	77,774	0.76	35	17
No-Merge	78,101	0.76	35	0

Table 2: Merge policies effect on ingestion throughput.

6.1.3 Range Query Performance

In this experiment, we ingested a total of 490GB worth of synthetic tweets into a dataset with a secondary LSM B$^+$-tree index on a randomly generated, 32-bit integer field, where both the primary and secondary indexes are using the prefix merge policy. We queried the dataset in three different operating regions: 1) during ingestion, 2) post-ingestion, and 3) post-ingestion and post-compaction, where each index's disk components are manually compacted (merged) to form a single component per index. The experiment starts by ingesting 420GB worth of tweets. Then, during ingestion, random range queries are submitted sequentially to AsterixDB. Once data ingestion ended (490GB worth of tweets were ingested), 2000 random range queries were submitted sequentially. The dataset was then compacted so that every index has exactly one



(a) With/out a Secondary Index. (b) With All Secondary Indexes. (c) Multiple Secondary LSM B$^+$-Trees.

Figure 4: Data-ingestion throughput when varying number of nodes and using different types of secondary indexes.

disk component. After compaction, another 2000 random range queries were submitted sequentially. All queries had range predicates on the secondary key. Thus, the secondary index is always searched first, followed by probing the primary index for every entry returned from the secondary index. In addition, the queries were generated randomly such that the result sets have an equal probability to have a cardinality of 10, 100, 1000, or 10000 records.

Table 3 shows the results of this experiment. Out of all three operating regions, querying the dataset while it is ingesting data produces the slowest response time for the obvious reason that the system resources (CPU and disk) are being contended for by both inserts and queries. We also observed that the performance of the queries is worse when there is an ongoing merge, as merges are both CPU and I/O intensive operations.

Avg. Result Cardinality	Avg. Response Time While Ingesting	Avg. Response Time After Ingestion	Avg. Response Time After Compaction
10	1023	139	138
100	1185	191	184
1000	2846	634	500
10000	11647	3747	3381

Table 3: Range query performance (in milliseconds).

Once ingestion is over, the query performance improved by a large margin due to reduced resource contention. On the other hand, surprisingly, the performance of queries after the final compaction improved by only a small margin. The reason for the comparable performance is two fold. First, before probing a primary index, the entries are sorted based on the primary key; therefore, good cache locality is achieved when accessing the primary index, mitigating the negative effects of having more disk components. Second, each primary index probe is a point lookup that makes use of the Bloom filters on the primary-index disk components, greatly reducing the chance of unnecessary I/O.

Finally, for all operating regions, there is a considerable overhead for small range queries, which is caused by the Hyracks job initialization. For each request, a new job is created and executed. As job creation includes the translation and optimization of an AQL program and execution includes the distribution and start of the job on all cluster nodes, this overhead can be significant for small jobs/queries. We tried an AQL query that does not touch persistent data to estimate this overhead. AsterixDB took an average of 43ms to execute our "no-op" query. We plan on fixing this next.

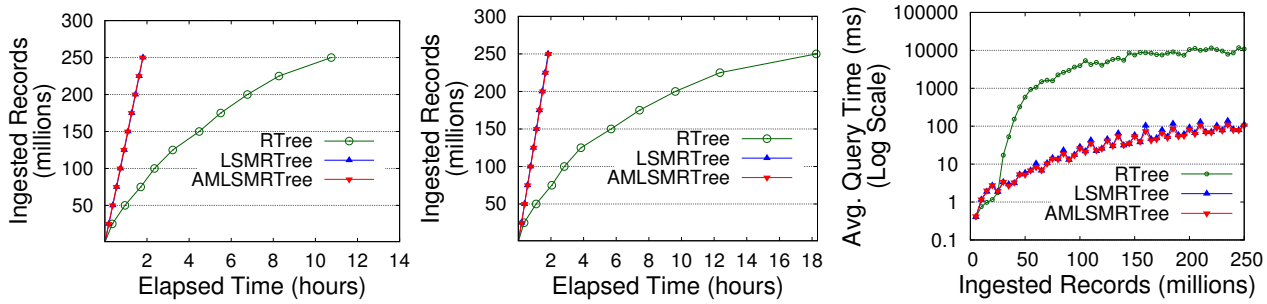
6.2 LSM-ification Framework

Next we show the detailed performance characteristics of a secondary index implemented using the "LSM-ification" framework. We compare an LSM R-tree ($LSMRTree$), an AMLSM R-tree ($AMLSMRTree$), and a conventional R-tree [14] ($RTree$). For both LSM indexes, the Hilbert curve is used to order the entries within each disk component. The following experiments were performed as micro-benchmarks where the indexes were accessed directly, bypassing compilation, job setup, transactions, and other code paths that are incurred by AQL statements submitted to AsterixDB.

We first empirically determined that the best ingestion and query performance for the LSM indexes can be obtained when their in-memory and disk components page sizes are 0.5KB and 32KB, respectively. On the other hand, a 2KB page size yielded, by far, the best ingestion performance for the R-tree. However, larger page sizes such as 16KB yielded much better query performance, but performed poorly for ingestion. We decided to use a 2KB page size for the R-tree since we are focusing on ingestion-intensive workloads. All experiments in this section use a single machine and utilize a single disk. All records and queries are sent from a client residing on a different machine. The machine configurations are the same as the experiments in 6.1. The R-tree used a 1.5GB LRU buffer cache, while each of the two LSM indexes used a 1GB LRU buffer cache for caching disk component pages and 0.5GB for their in-memory components. In addition, we used the prefix merge policy for both LSM indexes.

These micro-experiments employed data inspired by a real dataset. The source dataset contained event occurrence data that had time and location stamps, but the location information available in the source data was just at the (*city, state*) level. To convert this data into a more GPS-like spatial dataset for use in our micro-experiments, we synthetically augmented it as follows: we geotagged all of the US records (of which there were approximately 120 million) and ended up with roughly 4500 unique data points. We then used that point data to generate records following the frequency distribution of the provided dataset, but we shifted the points by adding random values in the range of [-0.25,0.25] to the latitude and longitude values as a way to mimic a more realistic fined-grained spatial dataset. The final dataset thus has approximately 4500 clusters, each with many distinct data points. The size of each record is 40 bytes (four double values representing each record's bottom-left and upper-right corners and a 64-bit integer representing a monotonically-increasing primary key). Note that since the spatial location being indexed is a point, the minimum bounding rectangle (MBR) for the point has corners with identical data (the point itself). It would be possible to optimize space in this case, but that is not currently done.

We generated query MBRs that provide result sets with similar cardinalities when querying each index as follows. Since many of the clusters are overlapping, the frequency distribution of each cluster was pre-computed based on the percentage of the overlapping area. For example, if two clusters overlap each other by 20% of their area, the frequency distribution of each cluster is incremented by 20% of the (original) distribution of the other cluster in order to compensate. To create an MBR for a query, a cluster is first chosen



(a) Raw Inserts. (b) Inserts with Concurrent Queries. (c) Queries during Ingestion.

Figure 5: Ingestion and query performance of the R-tree, LSM R-tree, and AMLSM R-tree.

at random. Then, the size of the MBR is determined based on the pre-computed frequency distribution of the chosen cluster such that the size will be relatively small if the cluster is highly populated and vice-versa. The size was restricted so as not to exceed the cluster size, avoiding the creation of queries that may span clusters, which may lead to very large result sets (this occurs when the number of records in the cluster is less than the requested result cardinality). Based on the MBR's size, we choose the MBR center randomly within the bounds of the cluster such that the MBR is fully contained inside the cluster. The average result set size of the queries was 2361 with 250 million records ingested into an index.

6.2.1 *Ingestion Performance*

Raw inserts: Figure 5(a) shows the raw insert performance when ingesting a total of 250 million records from a single stream. The R-tree took roughly 11 hours to ingest all of the data. Both LSM indexes were able to ingest the same amount of data in less than two hours. In the figure, the LSM indexes' curves are on top of each other because they behave similarly when the workload does not contain deletes (i.e., append-only workloads), as expected.

We also observe from Figure 5(a) that the LSM indexes maintain the same ingestion rate throughout the ingestion period. The first reason for this is that random disk I/O is avoided by LSM indexes. Second, in contrast to a primary index, a secondary index does not enforce key uniqueness, avoiding the overhead of checking for duplicate keys. Third, the prefix merge policy bounds the merge cost by ignoring disk components that are larger than a specified threshold (1GB in our experiments). On the other hand, the R-tree suffers from performance degradation over time. As more data is ingested, the cost of performing in-place index writes becomes increasingly expensive since the height of the tree grows without bound.

Inserts with concurrent queries: Similar to the previous experiment, a stream of 250 million sequential insert operations was sent to the indexes. In addition, a second, concurrent stream of queries were submitted sequentially during ingestion. Figure 5(b) shows the performance of these queries. For both LSM indexes, the effect was very minimal, mainly because their inserts are mostly CPU bound while queries are I/O bound. Therefore, queries are not contending with inserts for resources. On the other hand, the R-tree's ingestion ability was negatively effected, with elapsed time up from 11 hours to 18 hours. The main reason is that in-place inserts and queries are both I/O bound, causing resource contention between the two operations.

6.2.2 *Spatial Range Query Performance*

Query performance after ingestion: Here, again, 250 million records were ingested into each of the three indexes. Then, 1000 queries were submitted sequentially to each index. After that, the two LSM indexes were each compacted into a single disk component. Then, the same 1000 queries were again submitted to the LSM indexes. Before compaction, each LSM index had 12 disk components resulting from 78 flush and 19 merge operations, for a total size of 11.3GB. After compacting the disk components, the size of the single component was 11.3GB in both LSM indexes. The final R-tree index size was 17.6GB. The difference in size is due to the fact that the LSM indexes fully pack the pages of on disk components since they are immutable, resulting in better space utilization. Table 4 shows the results of this experiment.

After ingestion, the average query response time for the basic R-tree was 25 and 29 times slower than that of the LSM R-tree and the AMLSM R-tree, respectively. The LSM indexes' query times were much better since they incurred fewer buffer cache misses compared to the R-tree, which can be attributed to: 1) the LSM R-tree and the AMLSM R-tree use the Hilbert curve to order their entries in every disk component which improves the clustering quality of the R-trees' entries, 2) the pages of the LSM disk components are fully utilized, and 3) the disk page size of the LSM indexes is larger than the R-tree page size. The LSM R-tree performed slightly better than the AMLSM R-tree because the AMLSM R-tree returns records sorted based on Hilbert order to reconcile index entries. Since there are no deleted records in this experiment, the time spent to maintain the Hilbert order while answering queries is wasted. On the other hand, the Bloom filters associated with every deleted-key B^+-tree in the LSM R-tree handle this special case (an empty Bloom filter) in an efficient manner. After compacting the LSM indexes, their performance dramatically improved since spatially-adjacent entries from different disk components are now packed into a single component. Thus, the LSM indexes will have to access fewer disk pages when answering a query.

Index Name	After Ingestion		After Compaction	
	Avg. Response Time	Avg. Cache Misses	Avg. Response Time	Avg. Cache Misses
RTree	2543.2	560.6	-	-
LSMRTree	86.7	24.7	17.5	8.1
AMLSMRTree	100.8	24.7	16.5	8.1

Table 4: Spatial range query performance (in milliseconds) and number of buffer cache misses after ingestion and compaction.

Query performance during ingestion: In this experiment, a stream of 250 million inserts were again sent concurrently with another stream that submits queries sequentially during the ingestion. Figure 5(c) shows the corresponding query performance (notice that the Y axis is using a log scale). Each data point represents the average query time of all the queries that were submitted since the last data point. When the index had less than 25 million records, the R-tree query performance was slightly better because all of its pages were cached in memory, and because queries that were submitted to

the LSM indexes contended with flush and merge operations (there were 9 flushes and 2 merges during the ingestion of the first 25 million records). When the R-tree had ingested around 30 million records, pages from the buffer cache started to be evicted, which led to significant performance degradation. Both LSM indexes provided comparable query performance throughout the ingestion process. Their query times show some variance (the sawtooth shape) due to resource contention of ongoing merge operations.

Effect of deletion: Finally, we also study the impact that deletion has on query performance for the LSM R-tree and AMLSM R-tree. Again, we used a single stream that inserts 250 million records with a modification that now causes a 1% chance to delete an existing record instead of inserting a new record. Therefore, by the end of the experiments, the stream has submitted roughly 2.5 million delete operations to each index. We also used a second, concurrent stream that sent queries sequentially during the ingestion process.

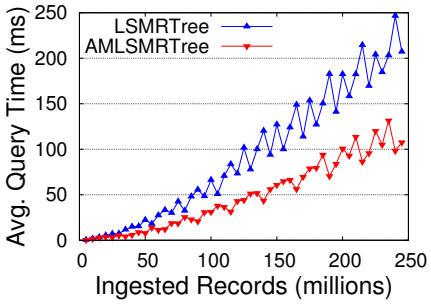


Figure 6: Effect of deletion on query performance.

Figure 6 shows the performance of queries in this case. Again, each data point represents the average query time of all the queries that were submitted since the last data point. The AMLSM R-tree consistently provided faster query response time than the LSM R-tree, almost by a factor of two, due to the handling of entry reconciliation. The AMLSM R-tree reconciliates entries "on the fly" as it accesses the entries based on Hilbert order from different disk components. On the other hand, the LSM R-tree reconciliates entries by probing all of the Bloom filters that are associated with the deleted-key B^+-trees of the newer components, and possibly the deleted-key B^+-trees themselves, which is more costly than reconciliation of entries in the AMLSM R-tree.

7. CONCLUSIONS

In this paper, we presented the storage engine implemented in the AsterixDB system. We described the framework in AsterixDB that leverages existing implementations of conventional indexes (e.g., the R-tree) to convert them to LSM-based indexes, which allowed us to avoid building specialized index structures from scratch and enables the advantages that an LSM index provides. Further, we explained how AsterixDB enforces the ACID properties across multiple heterogeneous LSM indexes. We also discussed the challenges that a system like AsterixDB faces for managing its disk and memory resources when dealing with many LSM-based indexes. Finally, we shared results from a preliminary evaluation of AsterixDB's storage engine that shows its performance characteristics in different settings and we presented the results for a set of micro-benchmarks to evaluate the "LSM-ification" framework.

Acknowledgements The AsterixDB project has been supported by an initial UC Discovery grant, by NSF IIS awards 0910989, 0910859, 0910820, and 0844574, and by NSF CNS awards 1305430 and 1059436. The project has also enjoyed industrial support from Amazon, eBay, Facebook, Google, HTC, Microsoft, Oracle Labs, and Yahoo! Research.

8. REFERENCES

[1] AsterixDB. http://asterixdb.ics.uci.edu/.
[2] Cassandra. http://cassandra.apache.org/.
[3] CouchDB. http://couchdb.apache.org/.
[4] HBase. http://hbase.apache.org/.
[5] LevelDB. https://code.google.com/p/leveldb/.
[6] S. Alsubaiee et al. Asterix: scalable warehouse-style web data integration. In *IIWeb*, 2012.
[7] Apache Hive, http://hadoop.apache.org/hive.
[8] A. Behm et al. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3), 2011.
[9] V. R. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.
[10] K. P. Brown et al. Towards automated performance tuning for complex workloads. In *VLDB*, 1994.
[11] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. *ACM TOCS.*, 26(2), 2008.
[12] S. Chen et al. Log-based architectures: using multicore to help software behave correctly. *ACM SIGOPS Oper. Syst. Rev.*, 45(1), 2011.
[13] Facebook. Facebook's growth in the past year. https://www.facebook.com/media/set/?set=a.10151908376636729.1073741825.20531316728.
[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.
[15] Jaql, http://www.jaql.org.
[16] C. Jermaine et al. The partitioned exponential file for database storage management. *The VLDB Journal.*, 16(4), 2007.
[17] I. Kamel et al. On packing R-trees. In *CIKM*, 1993.
[18] M. Kornacker et al. Concurrency and recovery in generalized search trees. In *SIGMOD*, 1997.
[19] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *VLDB*, 1990.
[20] C. Mohan et al. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS.*, 17(1), 1992.
[21] C. Olston et al. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.
[22] P. O'Neil et al. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4), 1996.
[23] O. Procopiuc et al. Bkd-tree: A dynamic scalable kd-tree. In *SSTD*, 2003.
[24] W. Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6), 1990.
[25] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, 1991.
[26] R. Sears et al. bLSM: a general purpose log structured merge tree. In *SIGMOD*, 2012.
[27] D. G. Severance et al. Differential files: Their application to the maintenance of large databases. *ACM TODS.*, 1(3), 1976.
[28] A. J. Storm et al. Adaptive self-tuning memory in DB2. In *VLDB*, 2006.
[29] Twitter Blog. New Tweets per second record, and how!, August 2013. https://blog.twitter.com/2013/new-tweets-per-second-record-and-how.