A Multi-tenant Relational OLTP Database at Salesforce

Vaibhav Arora
Salesforce
vaibhav.arora@salesforce.com

Subho Chatterjee
Salesforce
schatterjee@salesforce.com

Terry Chong
Salesforce
tchong@salesforce.com

Thomas Fanghaenel
Salesforce
tfanghaenel@salesforce.com

Pat Helland
Salesforce
phelland@salesforce.com

Jamie Martin
Salesforce
jameison.martin@salesforce.com

Kaushal Mittal
Salesforce
kaushal.mittal@salesforce.com

Nat Wyatt
Salesforce
nwyatt@salesforce.com

Abstract

Salesforce Database (SalesforceDB) is a relational OLTP database that directly supports the Salesforce multi-tenant application model using an LSM-based (Log-Structured Merge) storage engine. LSM-based OLTP databases offer many advantages including excellent write throughput and reduced cross-node coordination in shared storage architectures. However, LSMs pose challenges in managing the performance cost of reading data across multiple levels. Our read-dominated workloads comprise both single-record probes and range scans.

SalesforceDB implements well-known read optimization techniques including Bloom filters, leveling-based compaction, and fence pointers. While these techniques improve both probe and scan performance, they are insufficient for our workloads.

This paper presents three additional optimizations to LSM read performance: a location cache to accelerate key probes, range filters to improve short range scan performance, and early tombstone pruning to minimize tombstone overhead in queue-organized tables. All three optimizations are deployed in production, and we demonstrate their efficacy on real-world workloads.

1 Introduction

Salesforce Customer Relationship Management (CRM) applications run on data. As Salesforce grew to multiple regions and hundreds of instances, managing its multitenant workload on vendor database systems with high reliability became increasingly challenging. At the same time, Salesforce was moving its infrastructure to public cloud [7]. In developing its own RDBMS, the goals were more synergy with Salesforce applications via a cloud native, multi-tenant architecture, higher overall reliability, scalable and simplified operations and problem management. It needed to meet the growing performance and scale requirements of the CRM applications. In addition, there were opportunities to leverage the Salesforce multitenancy model inside the database for security, efficiency, and new product features. The architecture chosen for this database system was a fork of Postgres on LSM storage and access. More details on this journey will be published elsewhere; this paper illustrates some techniques that were used to get good OLTP read performance on an LSM-based database design in a commercial setting.

LSM trees [28] are widely used in storage engines and data management systems including RocksDB [5], HBase, CockroachDB [34] etc. SalesforceDB chose an LSM based storage architecture because immutable data files eliminate update-in-place operations, and this immutability [20] is leveraged for several purposes in the overall design. It reduces cross-node coordination in shared storage architectures and eliminates entire classes of corruptions and concurrency bottlenecks that plague in-place-update engines during B-tree splits and page modifications. Immutable LSMs enable use of simple storage interfaces like S3 rather than requiring POSIX-compliant storage. LSMs also facilitate various multi-tenancy use-cases within SalesforceDB.

LSMs excel at supporting write-heavy workloads, but struggle with read performance due to their multi-level storage architecture. Our workloads are read-dominated and comprise many single record lookups as well as range-scans. SalesforceDB supports multi-version concurrency control (MVCC), and each read is executed as of a particular snapshot. Reads on the LSM access and combine data across multiple levels of the LSM tree. SalesforceDB employs well known techniques like using Bloom filters in each data file in the LSM, fence pointers (which track boundary keys / time bounds in a data file), and leveled compaction to speed-up reads. While these methods enhance probe as well as scan performance, they are insufficient for our workloads.

A substantial portion of the scans in SalesforceDB access the data via secondary indexes. Certain tables in the application schema are especially wide with a large number of columns, and benefit from index access. SalesforceDB stores both base and index relations in index-organized tables [29]. For non-covered index scans, a scan has to look up the base table record corresponding to the index record. This encompasses setting up the fetch of the base table record and performing a fetch across the LSM levels to retrieve the record. This can result in non-covered index scans being expensive and negate some of the benefits of using indexes.

Another challenge is executing short-range scans, that scan a relatively narrow range of records. In these scenarios, the cost of the scan is dominated by trying to determine the starting position of the scan, and not the amount of the data to be scanned. For a

LSM based database, the scan has to be set up for each LSM level. Several of these scans at different LSM levels typically don't return results incurring extra setup cost as compared to traditional databases making in-place updates.

A third challenge for reads on the LSM pertains to handling deletes. Deletes in the LSM are handled via tombstones or delete markers [28]. Until the tombstones are permanently removed from persistence via compaction, queries can experience performance degradation due to the overhead of accessing and disregarding these tombstones during a scan. This effect is significantly worsened for tables organized as queues, which are commonly used by the application for processing messages in priority and FIFO order and have a high turnover rate of records. Dequeue operations involve scanning a few records in insertion order, and then subsequently processing and deleting them. These operations end up scanning and ignoring tombstones accumulated from previous dequeues, and end up slowing down further over time.

In this paper, we list three distinct techniques implemented to speed-up reads on a production-grade relational transactional database running on top of an LSM.

First, we introduce a location cache that caches the physical location of the latest version of the record in the LSM. This helps speed up non-covered index scans significantly, and avoids the extra setup as well as the cost of looking across LSM levels for the base table record. The cache is designed to be synchronization-free for lookups, providing efficient read performance. Cache population is synchronized with invalidation, which occurs during flushes and in a delayed manner after compaction. The cache entries are designed to be compact, so that the location cache can easily scale to support large database sizes (in the hundreds of terabytes), with a relatively low memory footprint.

Second, SalesforceDB employs range-filters to reduce the cost of short-range scans. We present a persistent range-filter design, inspired by the in-memory trie-based filters introduced in SuRF [40]. The filters are persisted for each immutable data file as it is written to the LSM. Range-filters in SalesforceDB help speed-up range scans by avoiding the scan setup work if the range is empty for a particular LSM level. SalesforceDB's range filter, CRaFT (Counting Range Filter Tries), is trie based, and each data file's filter is partitioned into fixed-sized blocks. This allows bringing blocks of the filter into memory as needed, an important feature for TB sized databases like SalesforceDB. Persisting the range-filter for immutable files avoids coordination with updates.

Third, we demonstrate how we enhance our compaction technique to take tombstones into account. We describe an optimization to allow removal of tombstones at higher levels in the LSM, thereby shortening their lifetime. In addition, SalesforceDB's compaction strategies aggressively drive compactions in keyspaces with tombstones. These two approaches dramatically reduce the impact of tombstones on query performance. Persisting the tombstone information in data files themselves ensures that this scheme can scale to large database sizes, and works with compactions executing across the cluster.

All three techniques scale to large database sizes supported by SalesforceDB (10s-100s of terabytes) without adding overhead on reads. We present data collected from our production fleet that demonstrates the effectiveness of these techniques.

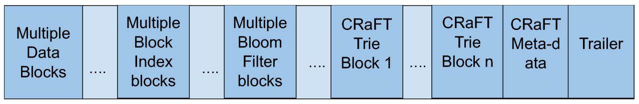


Figure 1: Data File Layout in SalesforceDB

2 Workload and LSM Overview

2.1 OLTP Workload

Salesforce provides a rich platform for Enterprise applications [38]. The application drives an extremely diverse, read-heavy workload comprising high volumes of small read-write transactions, some large transactions, and a mix of queries that include real time analytics, and a high volume of targeted operational queries for page loads and APIs. Customer written queries in SOQL (Salesforce Object query language) [38] are translated into SQL queries over its object-relational platform.

2.2 LSM Overview

SalesforceDB employs a single LSM for the entire multi-tenant database, mapping relational data to key-value pairs. The application data comprises base table and indexes, and both are stored as different relations in the LSM standalone secondary index [29]. Index keys are stored as composite keys, comprising the secondary index column values followed by the primary key of the corresponding base table record.

SalesforceDB uses a leveled LSM [32] optimized for read performance, configured with a fanout of five and eight levels. To support large database sizes, SalesforceDB performs partial compactions [17] with compaction processes executing per LSM level on multiple nodes across the cluster for horizontal scalability.

Each data file / SSTable is 2 GB in size and divided into 64 KB fixed-size blocks. A data file comprises data blocks, a block index, Bloom filter blocks, range-filter blocks (CRaFT), and a trailer block containing metadata corresponding to that data file. Block index blocks store keys for each data block. This structure allows for fast lookup of the data block that potentially holds a target key by using a binary search. Within each data block, records are spread across slots in key order, with each slot storing a key and a record entry, plus a slot directory to enable binary searching of keys within the data block. Figure 1 illustrates the data file layout in SalesforceDB.

Every database node in SalesforceDB maintains an interval-tree-based in-memory index to efficiently identify data files overlapping with a key or key range. Furthermore, each node has a read-only block cache populated with 64 KB blocks.

3 Location Cache

The Salesforce application utilizes secondary indexes extensively, and the database needs to ensure effective traversal of indexes for optimizing performance. SalesforceDB stores the secondary index as a separate relation, in the form of a lookup table. A secondary index stores composite keys, one for each base table row. The index entry consists of the index key fields followed by the primary key fields of the corresponding base table record. An index scan is a sequential scan on the secondary index relation that finds the qualifying index entries. With the primary key information extracted

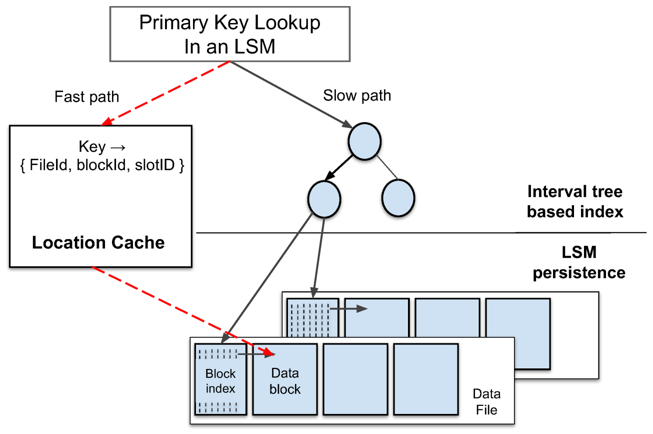


Figure 2: Location Cache Lookup

from an index entry, the corresponding base table row can be found by doing a point-lookup into the base table via the primary key index embedded therein.

The index-to-base lookup poses one of the biggest overheads in LSM based databases. A primary key lookup in an LSM tree is not a constant time operation, it is a logarithmic function of the size of the database. First, the interval-tree index needs to be searched to find the data files overlapping with a key. Bloom filters help in early exiting the search in levels where the data doesn't exist. But in levels that have a potential version of the target record, a data file needs to be searched. That involves searching another data structure (block index) to find the block that holds the record, followed by searching within the data block. All of these are logarithmic operations that may need to be performed on a number of levels of the LSM, depending on the overall size of the database. This is in stark contrast to traditional database systems, which provide a direct pointer lookup for the index to base record via a Row Identifier (aka RID).

To address this challenge, we present a data-structure called a location cache. The location cache is a very simple, light-weight, in-memory runtime cache that tracks the physical location of the most recent version of a record in the on-disk components of the LSM tree. Given the primary key of a base table record, the location cache can give information about the data file, block, and slot within a block where this record exists. The location cache is implemented as a simple direct-key lookup cache, hashing the primary key onto a simple array of fixed-size location descriptors. Location cache hits can result in false positives due to hash collisions. To avoid false positives, a key comparison between the lookup key and the key of the record in the target location needs to be performed. Only if the two keys match, the cache hit is a true positive.

As illustrated in Figure 2, when going through the location cache, the primary key lookup can be done before we set off on any of the logarithmic operations. Hence, by using a direct-mapped location cache, we can implement secondary indexes that provide access to a range of records in linear time.

The location cache is populated whenever a probe through the cache encounters a miss, and needs to go through a full LSM probe. Subsequent lookups for the same record can be served from the cache. The cache entry points to the latest version of the record present in persistence. Lookups that need to find something older than the most recent version of a record in persistence do not benefit from it. But that is a small minority of all cases. This is owing to the fact that the records which are recently modified are only periodically updated to persistence on flush, whereas the persistence location of other records only changes on compaction, and the frequency of that reduces as we go down the LSM tree, where majority of the data is present.

The location cache is designed so that record reads never block on synchronization. Cache population and invalidation require synchronization, but reads that cannot immediately insert a new location skip cache population rather than blocking on infrequent invalidations. Since every read operation specifies a snapshot at which the operation should be performed, the validity of read with respect to a snapshot can be determined after reading the record, without needing any coordination. This design allows for easy scaling of the location cache across multiple concurrent threads.

Location cache invalidations occur on flush, when a new immutable file with a new batch of updated records is added to the LSM tree. Only the cache entries corresponding to the set of newly written records are invalidated. Compaction events change the physical location of records, but not their contents. Therefore, location cache invalidations after compaction events can be done in a lazy fashion, as long as it is guaranteed that the old files containing the records are not recycled before the invalidation occurs. The lazy invalidation after compaction is essential to reduce synchronization as compaction occurs across nodes, as well as improve cache performance, and amortizing invalidation cost. Once a data file has been fully compacted out of the LSM, and has to be garbage collected, bulk invalidation is performed by scanning the cache and invalidating entries corresponding to a list of data files.

The location descriptors in the cache are fixed-size 64-bit values. Thus, a relatively small cache size can support large database sizes, while still enabling invalidations to be fast by utilizing SIMD instructions. The location cache size is set to a function of block cache size. Like with the block cache, location cache attempts to cover the working set. Consequently, the vast majority of hits in the location cache points to record locations in blocks that are still in the block cache, and will not incur physical I/O. This synergy between the block cache and the location cache alleviates the need for the latter to contain full record images.

The runtime overhead to maintain the cache is very small, and the penalty incurred on a cache miss is so large that performance improvements are noticeable even with very small caches and low hit ratios. In our experiments, we have found a location cache size of 10% of the block cache size has worked well and achieved hit ratios in excess of 75%. Sizing of the location cache is a topic of its own and future workloads may merit different sizes.

For memory-resident workloads, a successful location cache lookup significantly reduces data stalls due to processor cache misses from being over 250 to under 10 cache misses. This corresponds to a reduction in key lookup time from 10-15 microseconds to a few hundred nanoseconds, which is comparable to the performance of index lookups in traditional database systems.

4 CRaFT Range Filters

Salesforce OLTP workload comprises many short range scans, which are particularly challenging for LSM-based databases. These are narrow scans where the cost of the scan is dominated by the time spent in finding the potential position of the key range in the data file, rather than processing the scan results themselves.

An example of this is a nested loop join pattern which is frequently executed. There are scenarios where the workload comprises joins between two tables, with both of them being sufficiently large and one of them being an order of magnitude larger than the other. In such cases, we perform index nested loop joins using the smaller table. Nested loop joins are executed in other cases as well.

These nested loop joins break down into an outer table scan, and multiple inner scans, with an inner scan for each qualified outer table record encountered. For certain nested loop join scenarios in our workloads, many of these inner scans result in zero or one record per scan.

In a LSM-based database like SalesforceDB, a scan involves three logical steps:

- **Identifying relevant data files**: Determining which data files intersect with the key range being scanned.
- **Setting up the LSM scan**: Establishing the starting position of the key range within each identified data file. This involves binary searching across block index blocks to find the possible position of the key range in the data block.
- **Scanning data blocks**: Processing the data blocks and merging results across levels.

Each inner scan results in scanning data files across multiple LSM levels, with many of them returning zero rows, spending a significant portion of time in Step 2 above in setting up the scan. And if no records are present in the data file, the third step is wasteful for that file. This results in lower performance for such queries as compared to an update in-place RDBMS.

In addition to the nested loop join scenario, data files in L0 can have a wide keyspace, as they comprise the entire set of keys recently modified by the database. Consequently, they often overlap with the ranges of many range scans, requiring the scan to check these files. However, if the range of interest wasn't recently modified, it results in a scenario where the scan setup phase is wasteful.

To handle such scans more efficiently, SalesforceDB employs range filters. The range-filters, named CRaFT (Counting and Range Filter Tries), employ the LOUDS (Level Order Unary Degree Sequence) encoding of the trie described in SuRF (Succinct Range Filters) [40]. As a result of using this encoding scheme, CRaFT can support both variable length keys and queries, a requirement for SalesforceDB.

SalesforceDB stores CRaFT filters for each data file to describe its contents. These filters are partitioned and persisted across multiple fixed sized blocks in the data file. This automatically keeps the filters with the data they describe in the persistent LSM, avoiding any special memory management, auxiliary persistence artifacts and extra recovery handling. This is especially relevant for large databases. As CRaFT filters are maintained in fixed sized-blocks, they can be loaded and removed from the block cache as and when needed. Unlike global filters, CRaFT doesn't have to coordinate with updates, as filters are maintained at granularity of an immutable data file, and incoming updates have no impact on them.

To enable partitioning CRaFT, a two-level scheme is used, dividing the filter into CRaFT metadata and CRaFT trie blocks / partitions. CRaFT leverages the fact that keys being inserted in the data file are already sorted. These keys are divided into multiple independent tries that each fit into a 64K sized block. The CRaFT metadata block serves as an index for these trie blocks. Each data file typically consists of one or two metadata blocks and typically hundreds of trie blocks. The filter is probabilistic and can return false positives, but it guarantees no false negatives. Figure 1 illustrates the data file layout in SalesforceDB and how CRaFT fits in the layout.

Once the scan has determined that the queried range might be present in a data file, it will first check CRaFT filters to see if a range is present in that file. Each range lookup first looks at the metadata block, and then has to only lookup at most one CRaFT trie block to determine whether a range exists in the data file. In the cases where the range isn't present, the scan can completely avoid the scan setup that includes binary searching across hundreds of block index blocks to find the scan start position, as well scanning the data block. This improves the scan performance as multiple subscans across LSM levels can avoid the costly scan setup.

CRaFT is also used to determine key count for a range. This count is then leveraged during compaction to ascertain potential overlap of key ranges across levels, which subsequently guides the decision on which key spaces should be compacted. However, a detailed description of this aspect is beyond the scope of the paper.

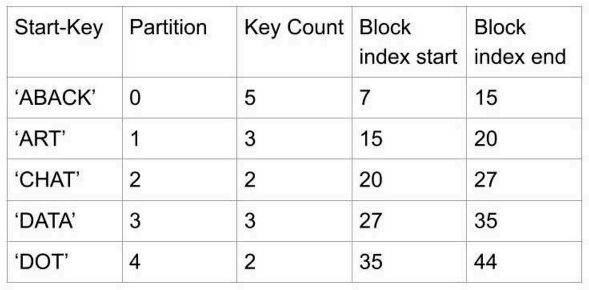
4.1 CRaFT Layout

CRaFT metadata provides a quick mechanism to identify the trie block(s) that needs to be searched for a given range. It is a sorted array storing the first key of each CRaFT trie partition, along with associated metadata for each block.

We store the following information for a trie block into CRaFT metadata (Figure 3):

- Start Key of the trie block: The first key in the trie block.
- Partition number of the trie block: We order the craft partitions in the order in which they are present in the data file.
- Key count: Number of keys in the trie block.
- Block Index starting block: The block number in the block index that contains the starting key in the trie block.
- Block Index ending block: The block number in the block index that contains the last key in the trie block.

CRaFT Trie Blocks / Partitions comprise an independent trie over a sorted sub-set of keys in the data file. Each CRaFT trie block stores the trie in LOUDS-S (LOUDS-Sparse) encoding of the tries, as described in SuRF. This includes LOUD-S and Has-Child bit vectors, and Label and Suffix vectors. The trie blocks also store some metadata for processing LOUDS trie: the number of trie nodes (n), the number of prefix nodes (m), Rank and Select metadata. In CRaFT, these data-structures are persisted in the block within a data file.



Start-Key	Partition	Key Count	Block index start	Block index end
'ABACK'	0	5	7	15
'ART'	1	3	15	20
'CHAT'	2	2	20	27
'DATA'	3	3	27	35
'DOT'	4	2	35	44

Figure 3: Craft Metadata Layout

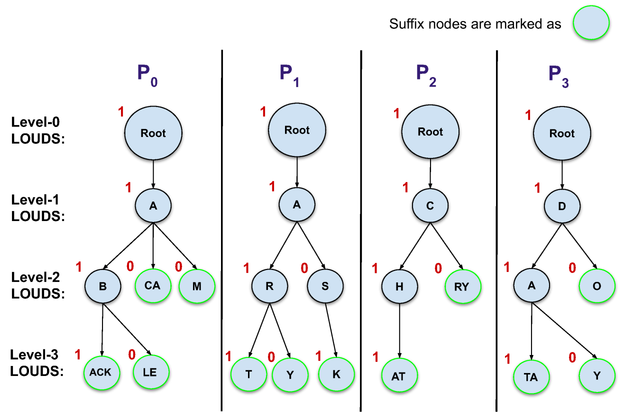


Figure 4: Craft Trie Partitions

Figure 4 illustrates an example of how keys in a data file are divided into multiple trie partitions. The figure also illustrates the prefix and suffix nodes.

4.2 Building CRaFT

CRaFT filters are generated each time a new data file is created during flush or compaction. When inserting a key into a data file, a data block entry is written, and then the corresponding block index, CRaFT and bloom filter entries are generated. Since CRaFT is employed at granularity of a data file, the keys are added to CRaFT in sorted order, and during each key insertion, the previous and next key being inserted are already known. Based on this sorted-awareness, the prefix and suffix portion of the key in the trie can be determined. This eliminates the need of navigating the trie from root to leaf nodes on each key insertion.

At a given instance, the keys are written to a single CRaFT trie block. The LOUDS, and HasChild, as well as the label and suffix vectors are incrementally maintained as keys inserted into the data file are processed. Once a trie partition's total size exceeds the block size, Rank and Select metadata is updated, the block is persisted and the CRaFT metadata block is incrementally maintained by adding an entry for this trie block. The block index information of the starting and ending keys is known at this point, and is updated. Keys are written to new trie blocks until the data file is full.

4.3 Querying CRaFT

For querying CRaFT to check whether a range(startKey, stopKey) is present:

- Binary searches are performed across the sorted keys in CRaFT metadata blocks to find the trie partitions with a key greater than the startKey, and the stopKey respectively.
- If startKey and stopKey correspond to different trie partitions, then the range exists.
- If startKey and stopKey correspond to the same CRaFT trie partition, a lookup is performed to see if the range exists in that block. This search efficiently scans bit vectors stored within the CRaFT block. The search is very similar to looking up the range in SuRF. Certain optimizations are employed for speed up but we omit them for brevity.
- If the range doesn't exist, the scan doesn't have to look at this data file further.
- If the range does exist, then the starting block index position in the craft metadata block entry is read for the startKey and stopKey, and block index search for scan setup is constrained to those block index blocks. This helps us speed-up the scan setup if the range exists, and ensures that the CRaFT is not wasteful in that scenario.

5 Handling Tombstones - Early Tombstone Pruning & Tombstone-Aware Compactions

The Salesforce application employs numerous tables as queues, and their usage patterns present distinct performance hurdles for LSM systems. The application utilizes these "queue-organized" tables for message processing and scheduling work based on these messages. Using these tables ensures that the messages are handled according to both their priority and their order of arrival.

These queue-organized tables manage records with brief lifecycles, resulting in rapid data turnover. Similar to other updates in SalesforceDB, deletes are processed by appending a new version of the record, which includes a specific marker indicating the deletion. These records are labeled as tombstone records. At high throughput, the swift churn in these tables leads to a substantial accumulation of tombstones. This build-up of tombstones is exacerbated in traditional LSM based systems where tombstones cannot be removed until they reach the lowest level, which could take a long time in large databases. Consequently, operations against these queue-organized tables often incur significant performance costs having to skip over these tombstones.

A critical operation that's impacted is dequeuing. Dequeuing requires scanning the tables to find the 'K' oldest records, which means they often encounter a large number of tombstones before locating active, valid records. This results in wasteful processing of already deleted records and diminished query performance. Dequeuing then adds more tombstones to mark records as processed, making further operations even slower. This behavior is illustrated in Figure 5.

SalesforceDB incorporates two optimizations to mitigate the challenges posed by tombstone accumulation: Early Tombstone Pruning and a Tombstone-Aware Compaction Policy. Early Tombstone Pruning leverages the knowledge of initial record versions to safely remove tombstones before they reach the last level; Tombstone

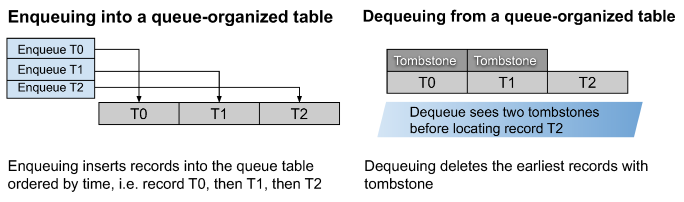

Enqueuing into a queue-organized table
Enqueue T0
Enqueue T1
Enqueue T2
T0
T1
T2
Enqueuing inserts records into the queue table ordered by time, i.e. record T0, then T1, then T2
Dequeuing from a queue-organized table
Tombstone
Tombstone
T0
T1
T2
Dequeue sees two tombstones before locating record T2
Dequeuing deletes the earliest records with tombstone

Figure 5: Impact of Early Tombstone Pruning

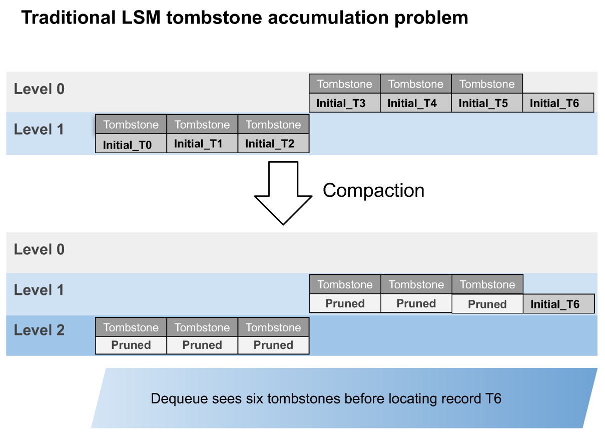


Figure 6: Compaction has to move tombstones to last level before they can be removed

aware compaction accelerates this process by triggering compaction on data files with high tombstone density. These two techniques reduce the impact of tombstones on querying queue organized tables.

5.1 Early Tombstone Pruning

Pruning refers to the cleanup activity executed during compaction, which involves physically discarding obsolete data versions that are no longer accessible to queries.

Early tombstone pruning relies on differentiating between initial record insertions and subsequent updates. Given that SalesforceDB adheres to SQL semantics, information regarding a record's initial status is a byproduct of existence checks conducted during INSERT statement executions. This mechanism enables SalesforceDB to maintain metadata that identifies initial record versions, representing the oldest extant version for a specific key.

The compaction process is enhanced to leverage this distinction in record versions. It recognizes that an entire chain of record versions, encompassing an initial record and up to and including its corresponding tombstone, can be pruned at any level of the LSM tree, not exclusively at the lowest level. This approach marks a significant deviation from traditional LSM compaction strategies.

By enabling the pruning of complete record chains (initial record + tombstone) on higher (smaller) levels of the LSM, the performance impact of tombstones for short-lived records is substantially reduced. Figures 6 and 7 illustrate how this prevents both the initial

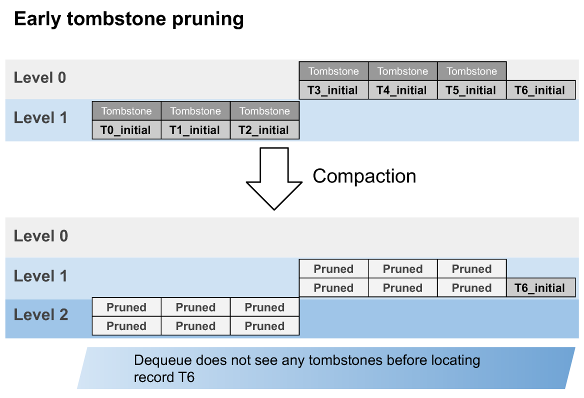


Figure 7: Compaction removes tombstones as soon as they sees them together with the initial records with early tombstone pruning

record and its associated tombstone from propagating deeply into the LSM tree, thereby minimizing their influence on subsequent read operations and overall storage footprint.

5.2 Tombstone-Aware Compaction Policy

To further expedite the removal of deleted records, SalesforceDB employs a tombstone-aware compaction policy. This policy utilizes storage-level metrics to continuously assess the concentration of tombstones across different key ranges.

SalesforceDB tracks the number of tombstones in a data file, storing this metric in the file's trailer block. The compaction mechanism has a trigger based on a threshold of the number of tombstones in data files, and prioritizes compaction operations on key ranges that exhibit elevated concentrations of tombstones. This compaction strategy accelerates the movement of tombstones to lower LSM levels, facilitating their elimination and associated overhead. Persisting tombstone information within data files enables compactions across multiple nodes to access this information, and ensures the scheme can scale seamlessly with the database size.

Early tombstone pruning and tombstone-aware compaction work in synergy with each other. Early tombstone pruning effectively curtails the lifecycle of tombstones, by enabling their elimination at higher LSM levels. Concurrently, the tombstone-aware compaction policy accelerates this process by prioritizing compactions in ranges with tombstones, and increasing the rate at which they are pushed to lower levels in the LSM, and meet the initial record. Collectively, these optimizations significantly enhance the efficiency of range scans and overall impact of tombstones on system performance.

6 Evaluation

This section highlights the effectiveness of the techniques discussed in the paper with data from production clusters. The databases were processing live production traffic, executing a diverse mix of operations outlined in Section 2. Each database emits periodic metrics that have been aggregated and presented from a week-long execution period. Each of the three techniques have been independently benchmarked, but here their efficacy is demonstrated across a large fleet of production clusters with default settings (all three techniques enabled).

6.1 Location Cache

The location cache was sized at 10% of the block size, and configured to be 64 GB, for a block cache size of 640 GB. When measuring hourly averaged cache hit rates, we observed a p50 value of 78%, p90 of 56%, and p95 of 45%, illustrating the robustness of the cache. The average time to perform index to base record lookup when not going through the location cache was observed to be averaging 13.5 micro-seconds, and when going through the cache, the item can be accessed in a few hundred nanoseconds, speeding up index to base lookups considerably.

6.2 CRaFT

We observed that for 417 billion scans across LSM levels, 86.45% of the scans were filtered by CRaFT. For the L0 level in the LSM, 95.34% of the scans were empty and filtered. We observe a false positive rate of only 4.17% across all levels. To quantify its benefits, we also micro-benchmarked CRaFT and observed it to reduce the scan time for an empty data file by 30% for an in-memory workload, illustrating its overall benefit.

6.3 Early Tombstone Pruning and Tombstone-Aware Compaction

These two techniques in combination resulted in an average of 61% of the tombstones across all database tables and indexes being pruned before getting to the last level. With SalesforceDB's LSM supporting TBs of data, it can take a considerable amount of time for the data to reach the last level. Therefore, pruning tombstones early greatly benefits queue-organized tables with a large volume of deletes.

7 Related Work

LSM-based [28] storage engines are the backbone of numerous data management systems like AsterixDB [10], LevelDB [2], RocksDB [5], WiredTiger [8], MyRocks [26] (built on RocksDB), CockroachDB [34] (on Pebble [4]) and many others [1, 39].

Many strategies exist to accelerate reads on LSMs. Monkey [13] and Elastic BF [23] speed-up point lookups with bloom filter optimizations. REMIX [43] and Disco [42] build index-like structures over the LSM's data files to reduce the setup cost of range scans and probes. Autumn [41] proposes gradually changing the capacity ratio between levels as database size grows. Various other compaction strategies have been proposed to speed-up reads [14], choose a design to optimize for read performance vs write-amplification [15], or drive compaction based on read accesses [2]. These approaches complement the techniques presented here.

Qader et al. [29] survey the class of techniques used for storing index data in the LSM. Index information can be embedded within the data files / SSTables storing the base relation by storing per-segment bloom filters / zone maps on secondary indexes attributes. SineKV [22] proposes decoupling values from primary keys and storing pointers to those values in the secondary index. SalesforceDB stores the secondary index as a separate relation, and stores composite keys, comprising the index column values followed by the primary key. The index is maintained eagerly avoiding any extra read validations [24].

For speeding-up index to base lookup, Next [33] proposes storing the index records in the same data files / SSTables as the corresponding base table records. An in-memory global index is then built to then get to corresponding data files for an index lookup. This approach has several drawbacks. First, it can not efficiently support building new indexes, a frequent operation for the Salesforce application, and often on large tables. Additionally, its global index presents challenges: it needs to be maintained during flush and compactions, and checkpointed and recovered. In contrast, Location Cache provides a near constant-time index-to-base lookup without imposing additional overhead on database operations like recovery or the index structure within the data files.

SlimDB [30] and Chucky [16] replace the bloom filter with a cuckoo-filter based global filter, and store the mapping of a key to the corresponding level-id. Similar to Location Cache, these approaches also map a key's hash to its physical location. However, there are several differences in these approaches. First, Location Cache stores a fine-grained location (data file, block, and slot). This granularity allows bypassing costly lookups through the block index and within the block once a key is found in the cache. Second, unlike SlimDB and Chucky, Location cache is employed as a cache, rather than a replacement of the bloom filter. This helps simplify the cache updates and maintenance, as all keys don't have to be in the cache, and if writing to the cache is stuck behind a costly cache bulk invalidation, a cache update can be simply discarded. Third, unlike Chucky, Location Cache doesn't eagerly invalidate records on every compaction. Invalidations after compaction can be deferred and performed in bulk during garbage collection of data files. Fourth, unlike SlimDB and Chucky, Location Cache doesn't maintain a list of keys with hash collisions or extension tables. Hash collisions are handled by simply verifying the retrieved key as a part of processing the record pointed to by the cache. This helps keep the data-structure relatively simple array of fixed-size location descriptors hashed by key.

GRF [36] is a global range filter storing locations of different key versions, mapped to a key's fingerprint. This enables using the filter to answer queries with multiple snapshots. While the location cache only stores the latest version's location, it validates results to ensure that the record returned from the cache does indeed satisfy the snapshot of the query. However, GRF restricts compaction policies, and cannot be used with the widely employed partial merge policy.

Several range filters [9, 11, 12, 18, 19, 21, 25, 27, 35, 37, 40] have been proposed to enhance range scan performance. Similar to DIVA [18] and ARF [9], SuRF [40] is a trie-based filter, and can support both variable-length queries and variable-length keys, which is a requirement for SalesforceDB. Unlike learned filters [11, 35], SuRF is built deterministically over a set of keys and does not require any model training. However, many of these range filters, like SuRF, offer only an in-memory design. An in-memory filter is not suitable for SalesforceDB due to the large database size. CRaFT illustrates a mechanism for persisting the trie-based range filter, SuRF, over fixed size blocks, and incorporating them in the data files themselves. This approach eliminates the need for auxiliary persistence files and simplifies recovery, as no special mechanisms are required.

RocksDB's Partitioned Index/Filters [3], like CRaFT's division into metadata and trie blocks, utilize a two-level index structure. This approach is designed to minimize the number of block index blocks that must be loaded into the block cache from a data file. In cases where CRaFT lookup finds that a key range exists, CRaFT metadata also helps in reducing the amount of block index blocks that have to be looked at.

Sarkar et al. [32] surveyed the compaction design, noting the two class of tombstone based compaction triggers: density based [5] and Time to Live (TTL) based [31]. Lethe [31] proposes the TTL-based approach motivated by legal and compliance use case. SalesforceDB also employs a tombstone density based trigger and selection. It's architectured in a way that tombstone information is present in persistence in the data files so that compactions performed across nodes can access that information, and no other auxiliary memory and persistence structure needs to be maintained. Single-delete [6] in RocksDB provides a mechanism to early prune tombstones if there is only a single version of the record in LSM, but is not as generic of an approach as SalesforceDB's early tombstone pruning that applies to every record chain implicitly.

8 Conclusion

Salesforce chose an LSM-based architecture for its OLTP database to achieve reliability, operational simplicity on cloud infrastructure, and multi-tenancy support. While LSMs excel at write throughput, they present read performance challenges at scale. We introduce innovative methods to speed up reads on LSMs, as well as adapt and engineer existing academic schemes to function effectively within a large production LSM database.

This paper presented three such production-deployed techniques addressing LSM read performance: a location cache accelerating record probes, CRaFT range filters reducing empty scan overhead, and early tombstone pruning with tombstone-aware compaction mitigating the impact of tombstones on query performance. Production measurements from thousands of database instances demonstrate these techniques were beneficial in improving OLTP read performance on an LSM-based architecture in a demanding commercial environment.

9 Acknowldegements

We are grateful to the reviewers for their insightful comments and suggested improvements, which have helped us improve the paper. Although this paper describes three techniques for optimizing read performance in SalesforceDB, we would like to acknowledge our colleagues in the broader SalesforceDB team, who have helped design, build, deploy and run the system in production. We also want to thank some of our past colleagues, Diego Ongaro, Rohit Agrawal, Aditya Shetty and Anusha Sivananainthaperumal, who worked on some of the approaches described in the paper.

References

[1] Apache Cassandra. https://cassandra.apache.org/.
[2] LevelDB. https://github.com/google/leveldb.
[3] Partitioned Index/Filters in RocksDB. https://rocksdb.org/blog/2017/05/12/partitioned-index-filter.html.
[4] Pebble. https://www.cockroachlabs.com/blog/pebble-rocksdb-kv-store/.
[5] RocksDB. https://github.com/facebook/rocksdb.
[6] Single Delete in RocksDB. https://github.com/facebook/rocksdb/wiki/Single-Delete.
[7] The Salesforce Platform. https://architect.salesforce.com/fundamentals/platform-transformation/.
[8] Wired Tiger. https://www.mongodb.com/docs/manual/core/wiredtiger/.
[9] K. Alexiou, D. Kossmann, and P.-Å. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *Proceedings of the VLDB Endowment*, 6(14):1714–1725, 2013.
[10] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, et al. Asterixdb: A scalable, open source bdms. *arXiv preprint arXiv:1407.0454*, 2014.
[11] G. Chen, Z. He, M. Li, and S. Luo. Oasis: An optimal disjoint segmented learned range filter. *Proceedings of the VLDB Endowment*, 17(8):1911–1924, 2024.
[12] M. Costa, P. Ferragina, and G. Vinciguerra. Grafite: Taming adversarial queries with optimal range filters. *Proceedings of the ACM on Management of Data*, 2(1):1–23, 2024.
[13] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94, 2017.
[14] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 2018.
[15] N. Dayan and S. Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*, pages 449–466, 2019.
[16] N. Dayan and M. Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, pages 365–378, 2021.
[17] N. Dayan, T. Weiss, S. Dashevsky, M. Pan, E. Bortnikov, and M. Twitto. Spooky: granulating lsm-tree compactions correctly. *Proceedings of the VLDB Endowment*, 15(11):3071–3084, 2022.
[18] N. Eslami, I. O. Bercea, and N. Dayan. Diva: Dynamic range filter for var-length keys and queries. *Proceedings of the VLDB Endowment*, 18(11):3923–3936, 2025.
[19] N. Eslami and N. Dayan. Memento filter: A fast, dynamic, and robust range filter. *Proceedings of the ACM on Management of Data*, 2(6):1–27, 2024.
[20] P. Helland. Immutability changes everything. *Communications of the ACM*, 59(1):64–70, 2015.
[21] E. R. Knorr, B. Lemaire, A. Lim, S. Luo, H. Zhang, S. Idreos, and M. Mitzenmacher. Proteus: A self-designing range filter. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1670–1684, 2022.
[22] F. Li, Y. Lu, Z. Yang, and J. Shu. Sinekv: Decoupled secondary indexing for lsm-based key-value stores. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1112–1122. IEEE, 2020.
[23] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu. {ElasticBF}: Elastic bloom filter with hotness awareness for boosting read performance in large {Key-Value} stores. In *USENIX ATC*, pages 739–752, 2019.
[24] C. Luo and M. J. Carey. Efficient data ingestion and query processing for lsm-based storage systems. *arXiv preprint arXiv:1808.08896*, 2018.
[25] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2071–2086, 2020.
[26] Y. Matsunobu, S. Dong, and H. Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.
[27] B. Mößner, C. Riegger, A. Bernhardt, and I. Petrov. bloomrf: On performing range-queries in bloom-filters with piecewise-monotone hash functions and prefix hashing. *arXiv preprint arXiv:2207.04789*, 2022.
[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta informatica*, 33(4):351–385, 1996.
[29] M. A. Qader, S. Cheng, and V. Hristidis. A comparative study of secondary indexing techniques in lsm-based nosql databases. In *Proceedings of the 2018 International Conference on Management of Data*, pages 551–566, 2018.
[30] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. Slimdb: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment*, 10(13):2037–2048, 2017.
[31] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A tunable delete-aware lsm engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 893–908, 2020.
[32] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis. Constructing and analyzing the lsm compaction design space (updated version). *arXiv preprint arXiv:2202.04522*, 2022.

[33] J. Shi, J. Yang, G. Cong, and X. Li. Next: A new secondary index framework for lsm-based data storage. *Proceedings of the ACM on Management of Data*, 3(3):1–25, 2025.

[34] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1493–1509, 2020.

[35] K. Vaidya, S. Chatterjee, E. Knorr, M. Mitzenmacher, S. Idreos, and T. Kraska. Snarf: a learning-enhanced range filter. *Proceedings of the VLDB Endowment*, 15(8):1632–1644, 2022.

[36] H. Wang, T. Guo, J. Yang, and H. Zhang. Grf: A global range filter for lsm-trees with shape encoding. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.

[37] Z. Wang, Z. Zhong, J. Guo, Y. Wu, H. Li, T. Yang, Y. Tu, H. Zhang, and B. Cui. Rencoder: A space-time efficient range filter with local encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 2036–2049. IEEE, 2023.

[38] C. D. Weissman and S. Bobrowski. The design of the force. com multitenant internet application development platform. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 889–896, 2009.

[39] Z. Yang, C. Yang, F. Han, M. Zhuang, B. Yang, Z. Yang, X. Cheng, Y. Zhao, W. Shi, H. Xi, et al. Oceanbase: a 707 million tpmc distributed relational database system. *Proceedings of the VLDB Endowment*, 15(12):3385–3397, 2022.

[40] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*, pages 323–336, 2018.

[41] F. Zhao, Z. Miller, L. Reznikov, D. Agrawal, and A. El Abbadi. Autumn: A scalable read optimized lsm-tree based key-value stores with fast point and range reads. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 2824–2837. IEEE Computer Society, 2025.

[42] W. Zhong, C. Chen, X. Wu, and J. Eriksson. Disco: A compact index for lsm-trees. *Proceedings of the ACM on Management of Data*, 3(1):1–27, 2025.

[43] W. Zhong, C. Chen, X. Wu, and S. Jiang. {REMIX}: Efficient range query for {LSM-trees}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 51–64, 2021.