Trie Memtables in Cassandra

Branimir Lambov
DataStax
Nesebar, Bulgaria
blambov@gmail.com

ABSTRACT

This paper discusses a new memtable implementation for Apache Cassandra which is based on tries (also called prefix trees) and byte-comparable representations of database keys. The implementation is already in production use in DataStax Enterprise 6.8 and is currently in the process of being integrated into mainstream Apache Cassandra as CEP-19. It improves on the legacy solution in the performance of modification and lookup as well as the size of the structure for a given amount of data. Crucially for Cassandra (a database running under the Java Virtual Machine), it also reduces garbage collection and general memory management complexity by operating on blocks of fixed size in large preallocated buffers. We detail the architecture of the solution and demonstrate some of the performance improvements that we have been able to achieve with it.

PVLDB Reference Format:
Branimir Lambov. Trie Memtables in Cassandra. PVLDB, 15(12): 3359 - 3371, 2022.
doi:10.14778/3554821.3554828

1 INTRODUCTION

Apache Cassandra [4] is a distributed NoSQL database [13, 30] whose storage design is a log-structured merge (LSM) tree [24, 27]. Like all LSM tree systems, Cassandra uses an in-memory buffer, called "memtable", to place incoming writes according to their logical order before flushing them to on-disk files (so called SSTables, for "sorted string tables" as originally called by Google's BigTable [3]). The performance of the in-memory buffer determines the peak write throughput that the database as a whole can achieve, while the amount of data that can be placed in it is a primary determinant of the achievable sustained throughput, as larger buffers enable a lower number of subsequent compaction passes over the data.

Traditionally, the data structures and algorithms that databases use to order data have been comparison-based. That is, to make decisions on how to place an item in an indexing structure, this item's key is compared to the key of other items. Cassandra is no exception, and its current memtable implementation uses a hierarchy of comparison-based data structures to organize data: a concurrent skip list is used to index database partitions[1], and separate B-Trees are used to index rows within a partition, columns within a row, and individual cells within a complex column.

From a practical point of view, comparison-based structures have some inherent inefficiencies that can lead to compromises in performance. Keys can can comprise of multiple components where it is common for elements to share the same value for all but the last component, and for comparisons to have to compare an equal prefix time and again. Keys can be in a multitude of different types, necessitating multi-morphic virtual calls to apply the correct comparison logic. Making decisions about the branch to continue operating with usually requires binary search, which is notorious for the unpredictability of its branching. Keys must be in comparable forms (e.g. `BigDecimal`), which can waste space or create a lot of object churn. Finally, keys must be fully present for a comparison, and the cache space required to avoid going to main memory for the most often consulted keys is large. Theoretically, the worst-case lookup complexity in comparison-based structures is $O(k \log n)$ (for n entries and key length k), which can be improved on if we know a bit more about the structure of the keys.

Cassandra's language of implementation is Java, and it uses the Java Virtual Machine's garbage collector as the primary memory management mechanism. Because of this, there are two factors that are the main consideration when determining the memtable size limits in Cassandra:

- the overall memory and Java heap size, and
- the efficiency of garbage collection (GC).

The current memtable solution in Cassandra can use off-heap memory to store the data coming with write requests, but uses only on-heap memory for the indexing structures necessary to structure and locate each piece of data (we will call this the "memtable metadata"). As a result, with the typical workloads where a row size is in the low hundreds of bytes, the on-heap footprint of a memtable dominates.

This has two downsides:

- The memtable size is limited by the heap size. To make use of pointer compression, this is usually set to under 32 GB of memory even on large machines, and needs to accommodate a lot of other transient and non-transient data.
- Garbage collection has to process the memtable metadata. Since memtables tend to stick around for a while, they are usually promoted to older GC generations and are only freed in full GC cycles.

The larger the fraction of the heap that the memtable takes, the worse the GC complexity becomes, and high GC complexity causes long stop-the-world pauses which in turn appear in the high percentile request latencies. Because garbage collection cycles in Cassandra are costly, one of the less evident side effects of this problem is an increased pressure on the Cassandra developers,

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554828

[1] Data in Cassandra is organized in partitions, distributed amongst nodes according to a hashing function and modified atomically, each containing a sorted set of rows of a number of columns, where each column is either a single data cell or a collection of cells.

regardless of the component of the database they work in, to reduce or eliminate object allocations, which makes the database code harder to write and understand, as well as more error-prone.

The considerations above indicate the potential of significant performance gains to be realized by developing a new memtable implementation. Our contribution addresses the inefficiency of comparison-based indexes by using a data structure that utilizes additional organization imposed on the database keys, and the garbage collection cost by hiding the internal composition of the index from the garbage collector.

The additional structure we impose on the database keys is the requirement to represent them as sequences of bytes that can be compared by their lexicographic order (called "byte-comparable representations"). Once this is in place, the comparison logic can be dramatically simplified, and comparisons can be decided early, without reading the entirety of each key, when prefixes differ. More importantly, data structures such as tries can be used to operate on these representations, which gives prefix compression and greater lookup and insertion efficiency with $O(k)$ worst-case complexity (for key length k).

Our solution, contributed to Apache Cassandra in CEP-19 [19] and included in DataStax Enterprise 6.8, improves Cassandra's memtables by using a state-of-the-art implementation of in-memory tries built to offer:

- Fast lookup and insertion based on the translation of keys to their byte-comparable representation.
- Sharing of key prefixes to reduce the space needed to store keys.
- Internal memory management that enables large parts of the memtable metadata to be stored off-heap or in large on-heap buffers which are opaque to the garbage collector.

2 BYTE-COMPARABLE REPRESENTATIONS

To be able to use machinery based on byte comparison, we need a method of translating any key to its byte-comparable representation. The translation must be such that for any two keys, the result of lexicographically comparing the unsigned bytes of the byte-comparable representation is equal to the result of comparing the keys.

Keys in Cassandra are formed of multiple components of predefined types. We use the type to define a translation that satisfies several requirements, and combine the translations of the individual components into a flattened translation of the whole key.

One way to achieve this is to require the following two properties for the translation ψ_T of a given type T and comparison function θ_T:

(1) Comparison equivalence:
$\forall x, y \in T, \vartheta(\psi_T(x), \psi_T(y)) = \theta_T(x, y)$
(2) Prefix-freedom:
$\forall x, y \in T, \psi_T(x)$ is not a prefix of $\psi_T(y)$,

where ϑ stands for lexicographic comparison on the unsigned bytes of a sequence.

If we have this and we attach together the bytes of the translations of a component, we can achieve comparison- equivalence for the composite type, i.e. the key:

$$\forall x_1, y_1 \in T_1, x_2, y_2 \in T_2, \dots x_n, y_n \in T_n,$$

$$\vartheta(\oplus_{i \leq n}(\psi_{T_i}(x_i)), \oplus_{i \leq n}(\psi_{T_i}(y_i))) = \theta_T(x, y)$$

where $\oplus$ is the concatenation operator, and $\theta_T(x, y)$ stands for lexicographic comparison of the sequences x_i and y_i using the corresponding type T_i.

The actual translation depends on the type, for example:

- For unsigned fixed-length integers, directly use the big-endian sequence of bytes.
- For signed fixed-length integers, invert the sign bit and use the sequence as above.
- For IEEE floating point numbers, invert the sign bit, and also invert all other bits if the number is negative.
- For blobs of varying size, encode all 00s in the blob as 00 01 and terminate the sequence with 00 00.

Cassandra already provides such a translation [11], using a tweaked variation of the above. Another example of byte ordered translation is given by Orderly [10].

3 TRIES

A trie (also called prefix tree) [14, 16] is a data structure that describes a mapping between sequences and associated values. It has a close relationship with finite state automata and was developed to encode words in a language. The trie terminology talks about "characters", "words" and "alphabet", which in our application map to bytes of the byte-comparable representation, the sequence that encodes it, and the possible values of a byte.

A trie can be defined as a tree graph in which vertices are states, some of which can be final and contain associated information, and where edges are labeled with characters. A valid word in the trie is encoded by a path starting from the root of the trie where each edge is labeled with the next character of the word, and ending in a final state which contains the "payload" associated with the word.

Finding the payload associated with a word is a matter of following the edges (also called "transitions") from the initial state labeled with the consecutive characters of the word, and retrieving the payload associated with the state at which we end up. If that is not a final state, or if at any point in this we did not find a transition in the trie matching the character, the trie does not have an association for the word. The complexity of lookup is thus $O(k)$ transitions, where the cost of taking a transition can be taken to be constant, thus this complexity is theoretically optimal.

When the items stored in a trie are lexicographically ordered (e.g. byte-comparable), a trie is also an ordered structure. A trie can be walked in order and it is also possible to efficiently list the items between two given keys.

From a storage space perspective, one of the main benefits of a trie as a data structure for storing a map is the fact that it completely avoids storing redundant prefixes. All words that start with the same sequence store a representation of that sequence only once. If prefixes are commonly shared, this can save a great deal of space.

Tries can be used as mutable or immutable structures, both in memory or on disk. For Cassandra, a mutable in-memory structure in which we can insert, modify or delete mappings can be used for memtables, and an immutable on-disk structure can be employed as

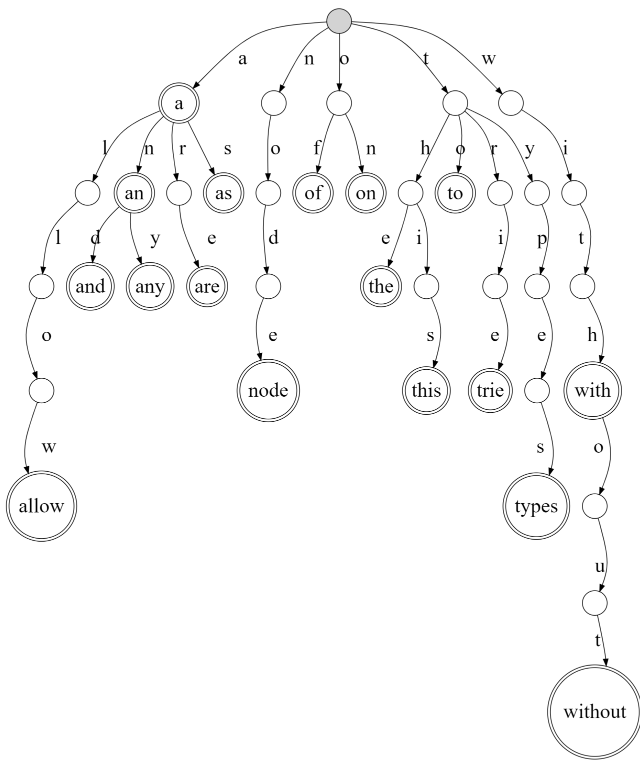

Figure 1: Sample trie mapping several words to themselves. The paths lead to "final" or "payload" nodes that contain a copy of the key.

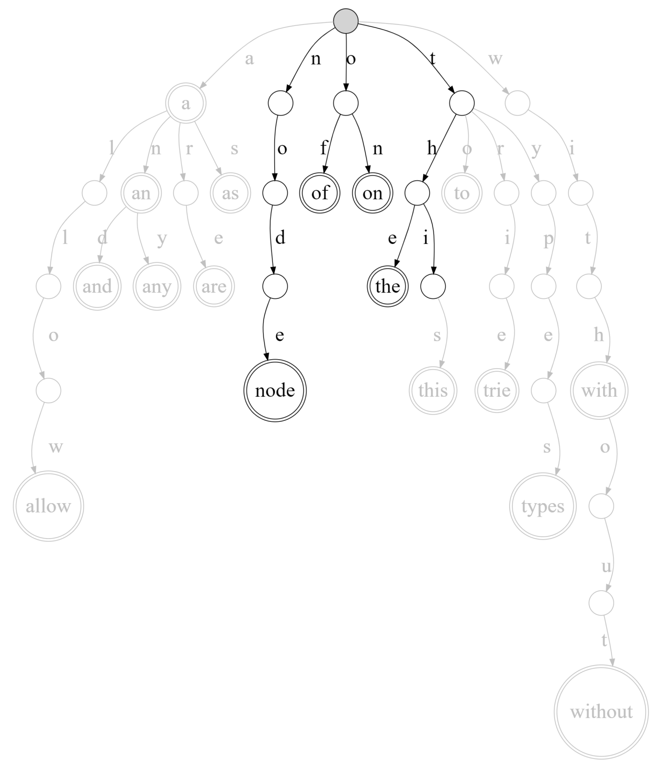

Figure 2: An example of slicing the trie on Figure 1 with the range "bit"-"thing". Processing only applies on boundary nodes (root, "t", "th", "thi"), where we throw away the transitions outside the range. Subtries like the ones for "n" and "o" fall completely between "b" and "t" thus are fully inside the range and can be processed without any further restrictions.

a primary or secondary SSTable index, or fully replacing SSTables. In this paper we focus on the former application. DataStax's trie-indexed SSTable format and storage attached index [9] are examples of the latter.

4 IMPLEMENTING TRIES

There exist many variations on the concept of a trie and various implementations, including examples of applications for a database [23, 31]. Some of the considerations that determine the efficiency of the structure are:

The size of the character set. It is well known [14] that equivalent tries and automata can be easily built with arbitrarily smaller or larger character sets. For example, one can use 8 single-bit transitions in place of byte ones. This can be used to replace wider tries with deeper ones, with the result of trading time for space efficiency. A somewhat extreme example is given by the so-called "succinct tries" [12] which store the structure with extremely low overhead but require lookup in external structures (constructed from the succinct representation) to take transitions.

As each transition may require one or more data fetches from main memory, it is preferable to have fewer, and employ wider alphabets to use more information in each step. In practice it is also possible to vary the alphabet size, i.e. to use sub-transitions or bundle multiple together.

The method of storing transition information. The most straightforward approach is to use a two dimensional array with nodes in one dimension and characters in the other. Equivalently, one can have node objects each using an array to store target nodes per each possible character. We call this the "dense" format. It is pretty quick in finding and adding information, but requires space which is proportional to the number of nodes multiplied by the number of characters. As the latter is not small, this structure is prohibitively large.

The common alternative is to use an ordered map indexed by transition in the node object, most efficiently by storing a sorted array of characters and array of target nodes. We call this the "sparse" format, and it uses space proportional to the number of transitions. The downside is that lookup in this structure is slower (using binary or linear search), and modification is much more difficult.

A third option is to use a combination of the two: use typed nodes, where some nodes use dense storage and others sparse, depending on the number of transitions from the node. In practice, tries usually tend to form a structure where the nodes closer to the root have many transitions. As they are most often consulted and

take up most of the time in aggregate, it makes a lot of sense to use a dense format for them. On the other hand, the nodes further from the root usually have fewer transitions and form the bulk of the nodes in the trie, and for them it makes sense to use the more compact sparse format.

Usage of multi-byte transitions. A further special case of the above is a situation where there is a sequence of nodes that have only one transition, for example as the remainder of a key after a unique prefix has been determined. It is valuable to be able to store such sequences compactly, so that a character can take just one byte and multiple transitions are easier to take. A common solution to this is to permit "multi-byte transitions", an approach often called a "radix tree" or "Patricia trie" [16, 26].

Data locality and cache line packing. Data fetches from main memory into the cache usually progress in chunks of at least 32 bytes. If it is possible to pack related information in these chunks, and avoid extending beyond a cache line size, the cache efficiency and access complexity can be improved.

Memory management. A straightforward approach is to use Java objects representing trie nodes. Such nodes can be polymorphic, and reference arrays containing transition characters and children.

This solution has multiple drawbacks: the object overhead for such objects can be multiple times the data size stored, additional pointer-chasing hops and memory fetches may be required to fetch the arrays and, as discussed in the introduction, complex structures of Java objects with medium-term lifetimes are very difficult to handle efficiently by a garbage collector.

Concurrent operation. Locking structures are easier to implement, but to achieve better performance it is often preferable to use lock-free ones. Complex designs, however, are very hard to implement with the latter. The single-producer-multiple-consumer option provides a good middle ground where writes are applied sequentially, but multiple reads can proceed concurrently with any write.

The method of presenting the trie structure. In an LSM tree's use case, the ability to merge and slice trie content efficiently is extremely important. If the internal structure of a trie is available for these operations, they can also benefit from the prefix-sharing effect as all comparisons between nodes of two tries are effectively comparisons between all the keys that share the same prefix.

It is natural to present the content of a trie as a tree of nodes where each node can be queried about its transitions and child nodes. Transformations such as merging would then create combination nodes representing the transformed node (e.g. the union of all source nodes, where each child is also constructed by creating a union of the relevant children from the sources). Unfortunately this presentation can create a large amount of intermediate objects, which can pose a problem for garbage collection or other memory management.

5 DESIGN OF CASSANDRA'S IN-MEMORY TRIE

5.1 Layout

One of the main considerations of the design has been the need to avoid complex structures of objects on the heap. To this end we opted to allocate large chunks of memory (on- or off-heap byte buffers as requested by the user) in which we store a trie's structure (i.e. the trie's nodes). This means that we have to perform our own memory management inside these chunks, and that this management needs to be as simple as possible.

Considered in combination with the cache line packing point from the list above, this led to the decision to pack data in 32-byte "blocks" that form a unit of memory allocation and release in the buffer. 32 bytes are not enough to store all the data for a node with many transitions, but this problem can be overcome by breaking up large nodes into sub-transitions.

Logically, the trie operates on nodes with byte transitions. Internally, the nodes are laid out in blocks, but there is no 1:1 correspondence between nodes and blocks. There are five different types of nodes in our tries:

- `Split` nodes have many children and are laid out using sub-transitions. The node id points to the top level of the split node, which contains pointers to four middle blocks corresponding to the different values of the two most significant bits in the transition byte. The mid-level blocks contain pointers to eight bottom blocks corresponding to the next three bits, and the bottom blocks point to eight children each corresponding to the least significant three bits in the transition byte. Any pointer in this structure can be null (for example, there can be a null mid block for 01, meaning that this node has no mapping for any byte between 0x40 and 0x7F), and the space it takes is between 3 and 37 blocks.
- `Sparse` nodes have between 2 and 6 transitions and occupy one block. They list the possible byte values and the child pointers but, to permit modifications concurrent with reads, do not store them in order (note: searching within a small array is usually faster using direct rather than binary search). For in-order iteration they also maintain an "order word" which encodes how the children are ordered.
- `Chain` nodes handle the single-child case. As described under multi-byte transitions in the previous section, single-child nodes most often come in sequences; in such blocks we store sequences of 1 to 28 chain nodes, where the last has an explicit pointer, and the previous ones implicitly point to the next.
- `Leaf` nodes point to content and have no further children. They do not have corresponding blocks; at this time our tries map to Java objects which cannot be stored in byte buffers, thus we also maintain a separate content array and point directly to an index in it.
- `Prefix` nodes hold content that is associated with some prefix, i.e. content that falls on an intermediate node in the trie rather than a leaf. Prefixes augment a node by placing the content information either in free space within its block (e.g. the first 16 bytes of the leading block of a `split` node) or by pointing to it from a separate block.

`Leaf` nodes are the most common, followed by `chain` and `sparse`. `Split` and `prefix` nodes are rare, which makes this storage efficient. Also, when we bring a block from main memory, we can make the most of it by either going deep and following multiple transitions (`chain`), by having the width of all the possible children

(sparse) or the targets of a three-bit choice (split) depending on the complexity of the state at the node where we are positioned.

To avoid using space inside a crowded block, and to make it possible to point to specific nodes in chain blocks, we store the type of node in the node pointer. As we work inside a large byte buffer, which in Java are limited in size to 2 GB, to identify a 32-byte block we need the top 27 bits of a positive integer. We use negative integers to identify leaf nodes, and the last 5 bits of positive ones to distinguish between split, sparse, prefix, and the exact node in a chain block.

The properties of the various node types are summarized in Table 1. Further examples and details can be found in the code's bundled documentation [21].

5.2 Traversal

The critical operation that tries in Cassandra need to support is the ability to iterate the keys and content of a bounded range from a union of several tries. For this to be efficient, we need a method of presenting the internal structure of the trie with minimal intermediate object creation.

To do this we chose a stateful "cursor" paradigm which lists the nodes of a trie in lexicographic order. That is, if we implicitly associate each node in the trie with the path that is followed to reach it from the root, all nodes of the trie will have associated words, and when we place these in lexicographic order, the step between any two consecutive ones either appends a new letter, or truncates the word to a shorter length and appends a new letter. In the graph this corresponds to either taking the first child transition of the current node, or (when it does not have children), ascending to the closest parent that still has any children left, and moving to its first remaining child.

This is implemented in its most basic version as a single operation, advance, which moves the cursor to the next node in lexicographic order and returns information about the reached descend-depth as well as the label of the transition that was taken. This information is sufficient to recreate all visited keys, because after each advance the descend-depth specifies which character in the key needs to be replaced with the returned, as well as the length of the new path.

Crucially, when two or more cursors are advanced in parallel, the descend-depth and incoming transition are also sufficient to determine which one is earlier in iteration order, which in turn permits efficient implementation of unions and intersections. See the bundled documentation [22] for further details.

5.3 Modification

Modification in this structure is done by a single writer, but readers can progress in parallel with a writer. This is permitted by the structure of the individual node types, where we can add information without invalidating existing data (for example, to add a new child to a sparse node we append the new target and transition byte without moving earlier values and creating invalid intermediate states).

The mutation process is recursive, where we track the nodes corresponding to positions in the trie on the way down, and apply

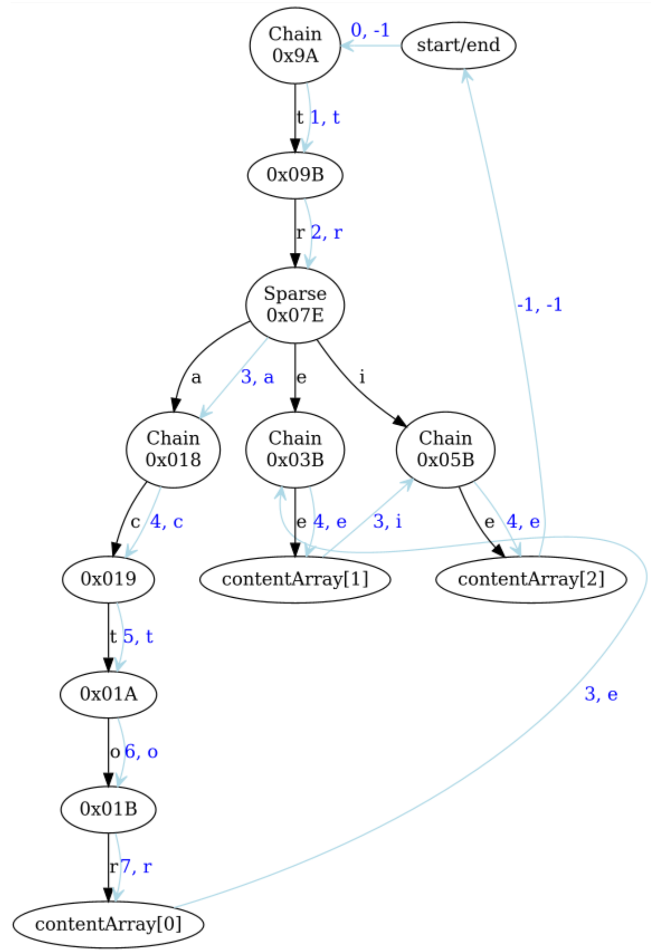

Figure 3: A sample trie with a cursor walk over it. Arrows in black show the trie structure. The labels inside the nodes are the pointer values with which the nodes is reached, or the content which is associated with the leaves. Arrows in blue show the progress of the cursor in response to advance calls together with the associated descend depth and incoming character.

any added information on the way back. While there is no corresponding trie data on the way back, we can create chain nodes by adding bytes; eventually we reach a node where we have to add a new child. If the node permits it (e.g. a sparse node with less than 6 children), we do so with a volatile write — any new reader is then guaranteed to see the new branch, which is already fully prepared.

If the node cannot be expanded (e.g. a chain node with a different transition byte), we copy it, switching to the next wider type and adding the new child. We then ascend to the parent and remap its child pointer (with a volatile write as above), but also leave the original version of the node in place for any reader that may still be active and positioned on it.

Table 1: Node types and their properties

	`leaf`	`chain`	`sparse`	`split`	`prefix`
children	0	1	2-6	>6	—
blocks per node	0	1/28 to 1	1	3-37	0 (embedded) / 1 (standalone)
usage	leaf node with content	multiple single-child transitions	few children	many children	adds content or meta-data
typical location in trie	leaf	before leaf	in the middle	towards the root	prefix entries; boundaries
lookup complexity	—	equality check	linear lookup	bit manipulation + pointer hops	—
pointer offset	negative	00-1B points to specific node	1E	1C	1F
layout	content index encoded as ~ `pointer`	00: trans to 01 ... 1A: trans to 1B 1B: trans to child 1C-1F: child pointer	00-03: child pointer 0 04-07: child pointer 1 ... 18: trans 0 19: trans 1 ... 1E-1F: order word base-6 encoded	head block 10-13: pointer for 00 14-17: pointer for 01 18-1B: pointer for 10 1C-1F: pointer for 11 mid and tail 00-03: pointer for 000 04-07: pointer for 001 ... 1C-1F: pointer for 111	embedded 00-03: content index 04: child ptr offset 05-1F: augmented node (`chain` or `split`) standalone 00-03: content index 04: 0xFF 05-1B: unused 1C-1F: augmented node pointer

6 THE TRIE MEMTABLE

The elements above are combined in the `TrieMemtable` class, which is the new memtable implementation for Cassandra that users can switch to via its new memtable API [18].

Currently our solution combines a sharded trie partition map with the existing mechanisms for storing data inside a partition. In other words, the only component that we have changed is the existing skip-list partition map, which we have replaced with several tries split by hash ranges. This will not be the ultimate form that the solution is expected to take, but it already provides some dramatic advantages.

In the trie memtable we map each partition key to a byte-ordered representation of its decorated version (i.e. partition key augmented with a hash token, where order is determined by the hash value first). This representation forms the path in the trie, and the payload attached to the end of this path is the partition content.

Unlike other memtables, we do not store the partition key with the partition content. Instead, we only store it as the trie path. This improves memory usage and permits the memtable to take full advantage of the prefix-sharing benefit of tries. The disadvantage is that the key needs to be translated back on retrieval — however, this only needs to happen on range queries and memtable flushes, because point queries (which are most operations in Cassandra) are already supplied with a copy of the required key.

As our tries cannot be written to concurrently and the memtable must be able to scale its write performance with added resources, we cannot use a single trie for the map. Instead, we partition the hash space for which the node is responsible into equally-sized ranges and create a separate trie for each range. The number of such tries (called "shards") is by default set to be the number of CPU threads available to the process, and can be adjusted by the user. Shards are also a mechanism for overcoming the 2GB trie size limitation, which only applies to individual shards rather than the whole memtable.

Because normally the database attaches a hash to each key which should be close to evenly distributed in the hash space, each shard should receive a close-to-equal part of the writes to the memtable, and thus the performance of the memtable should normally be close to optimal.

There are two exceptions to this: non-hashing partitioners, such as the one used for legacy secondary indexes, and workloads where writes to a single partition dominate. While at the moment this means we can't unqualifiedly recommend using the new memtable in these scenarios, the improved lookup performance and memory management may still provide a benefit compared to the legacy solution. Sharding is further discussed and tested in section 8.1.

In addition to the sharded representation, the memtable also provides a merged view on the whole memtable that is used for range queries and flushes. This view is constructed by performing a union of the per-shard tries as they are created. As each source is reflected live in the union, this view is always up-to-date with the full content.

7 PERFORMANCE

We will start with some microbenchmarks, comparing the read and write speed using the legacy skip-list memtable and the new trie implementation. We use a JMH microbenchmark called `ReadTest-SmallPartitions` that is included in the Cassandra code on a

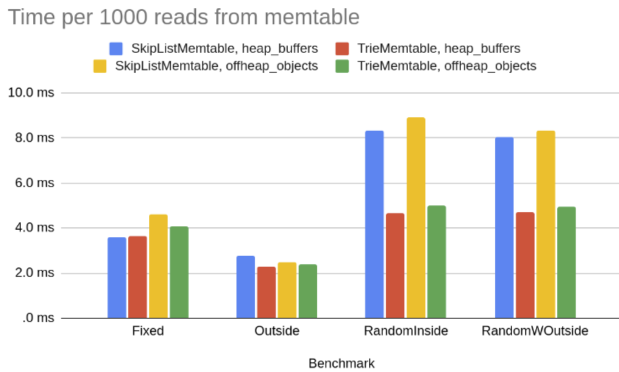

Figure 4: Read microbenchmark, lower is better

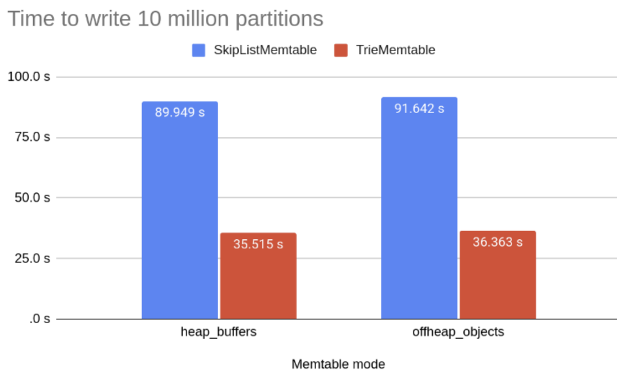

Figure 5: Write microbenchmark, lower is better

Core i7-6700K machine with 32GB of RAM under Ubuntu 18 and memtable configured to use up to 8GB of on- and off-heap memory. The benchmark uses a key-value table (i.e. one that does not have a clustering key and each row is only identified via its partition key), fills it with a given number of writes and runs a test of repeatedly reading batches of partitions from the table. Here we used 10 million partitions and batches of 1000 reads, and ran the test both for the default "heap_buffers" memtable mode (where the stored data is on the heap in byte buffers), as well as the "offheap_objects" mode (where the stored data is in directly-allocated off-heap memory) which is preferable when higher throughput is required.

The results are summarized in Figure 4. The legacy format is only able to keep up with the trie memtable when it is repeatedly being asked to return the same partition. In the more interesting case of random lookup with or without looking for non-existing content, trie memtables are almost twice as fast.

The time to prepare the run, i.e. to fill in the memtable with the necessary data, shown in Figure 5, is also dramatically faster with the new solution regardless of the memory mode.

Figure 6 shows the memtable size after the data is loaded. The memtable mode has historically only affected where the table's inserted data is placed — any indexing structures remain on heap.

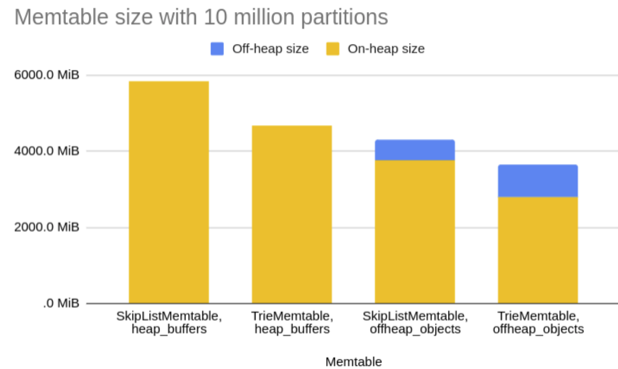

Figure 6: Size microbenchmark, lower is better

With the new implementation we can also move the partition index off heap which can be easily seen in the graph.

For all configurations, including off-heap trie, the on-heap metadata dominates the space used. Even so, the new solution is able to reduce the heap size of the memtable by 20 to 30%. In practical terms this is reflected in better garbage collection and compaction efficiency. We expect this to improve much further with later additions to the solution.

Let's examine how the new memtable behaves with larger-scale tests. To this end, we used a long-running high-throughput test[2] that we use to evaluate the throughput that the database can sustain up to a very large data size. We performed this test both on the CASSANDRA-17240 branch in Apache Cassandra [17] which introduces this solution, as well as on DataStax's "Converged Cassandra" branch [8], which includes additional improvements (notably, improved compaction) that let the implementation perform to the best of its ability. The tests were carried out using DataStax's Fallout [15, 25] deployment, using a single `i3.4xlarge` instance as server and 11 `i3.xlarge` instances as clients sharing one `i3`-equivalent node in our lab. The following settings were changed in `cassandra.yaml` to avoid some known throughput bottlenecks:

```
memtable_allocation_type: offheap_objects
memtable_flush_writers: 8
memtable_heap_space: 16384MiB
memtable_offheap_space: 16384MiB
concurrent_reads: 256
concurrent_writes: 256
commitlog_total_space: 51200MiB
commitlog_segment_size: 320MiB
commitlog_compression:
    class_name: LZ4Compressor
disk_access_mode: mmap_index_only
file_cache_size: 8192MiB
compaction_throughput: 0MiB/s
concurrent_compactors: 30
key_cache_migrate_during_compaction: false
```

[2]The test was performed using the script at https://gist.github.com/blambov/ce4e22a217d8d62fc959d7e6e24eb77e. This script makes use of DataStax software deployments and hardware, which can be substituted by the open-source Fallout [6] in combination with NoSQLBench [7] (which is an evolution of the EBDSE module used in the test, and is compatible with its scripts), run on public cloud (e.g. Amazon's EC2).

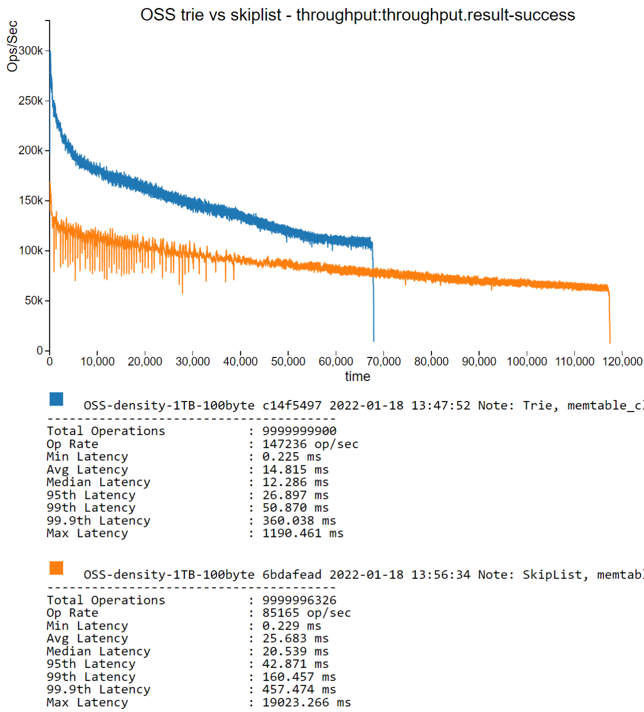

Figure 7: Throughput test using the CASSANDRA-17240 branch. The graph plots the throughput achieved in the Y axis against test time in the X axis. Trie in blue, skip list in orange. Higher values are better.

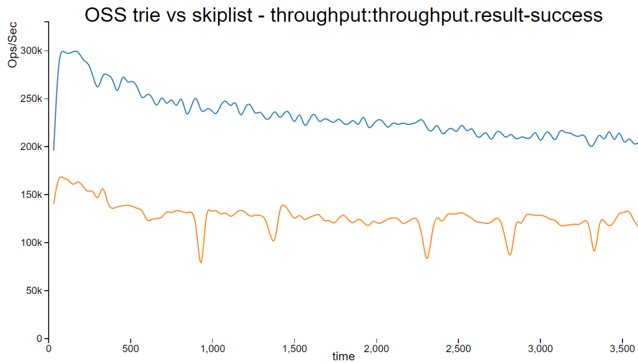

Figure 8: First hour of throughput test.

The main part of the test is a long-running 90:10 write:read high-throughput stage that shows the maximum throughput that can be achieved and sustained in writing over 1 TB of data in 100-byte partitions. Like the microbenchmarks above, this is also a key-value test (rows are only identified by a partition key). To be more representative of real workloads, this test skews reads quadratically towards accessing recently-written data. The main factors affecting the performance of this stage are the performance of the memtable in accepting writes, serving reads and flushing its content to disk,

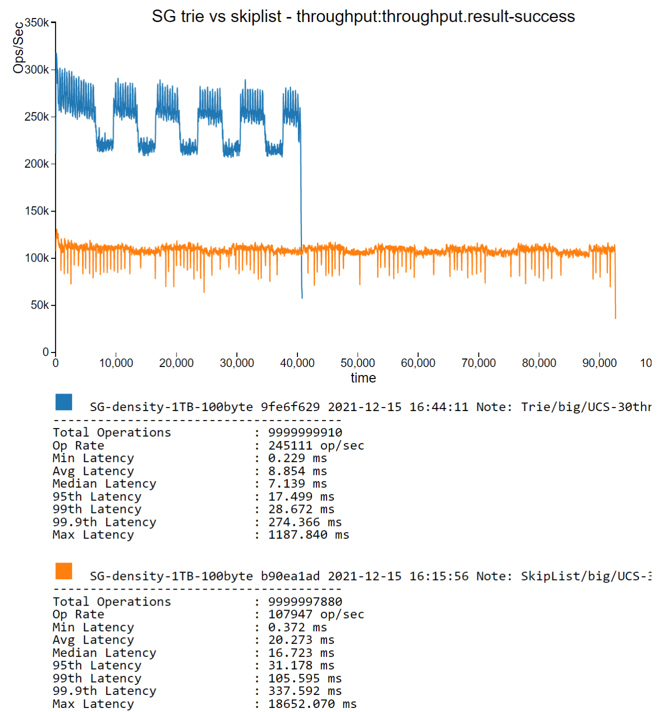

Figure 9: Throughput test using the Converged Cassandra branch. Higher values are better.

as well as, as the test progresses and accumulates data, the efficiency of compaction — if compaction is not able to perform at the same level as the LSM buffer, SSTables accumulate and reads become progressively more difficult until they start dominating the execution time of operations. We can see both effects in Figure 7.

The process finishes in ~ 19 hrs for the trie memtable (in blue) and ~ 33 for the skip-list (in orange), which corresponds to average throughput of ~ 145 vs ~ 85 thousand operations per second (of which 10% are reads). The throughput starts high for both memtables, and quickly drops as other bottlenecks take over the computation time. This is most evident for the trie memtable, where we can see a sharp drop in throughput as compaction is unable to keep up with the load and allows SSTables to accumulate, eventually (near the end of the test) slowing down writes to a level that it is able to maintain. The drop is not as sharp for the legacy solution, which is both due to every operation taking more time and thus the effect of accumulation taking longer to appear, as well as the fact that it starts off closer to the achievable compaction throughput.

Figure 8 is a zoomed view of the first hour of the test which is less influenced by compaction and clearly demonstrates close to doubling of Cassandra's burst performance, as well as flattening of the curve due to the relieved garbage collection pressure.

As we have previously recognized compaction as a bottleneck, we have also worked on separate improvements[3] which we have included in our Converged Cassandra open-source branch [8]. With

[3]Most notably the unified compaction strategy [1] and cursor compaction process [20], which are yet to be proposed to mainline Cassandra and publicized.

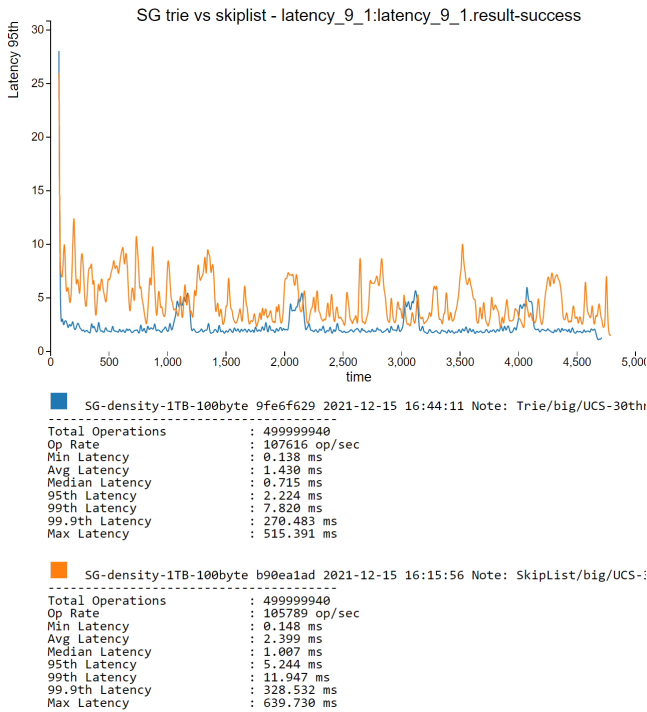

Figure 10: Latency test at fixed throughput (110k ops/sec), where 10% of the operations are reads and 90% are writes. The graph plots 95-percentile latency in the Y axis against test time in the X axis. Trie in blue, skip list in orange. Lower values are better.

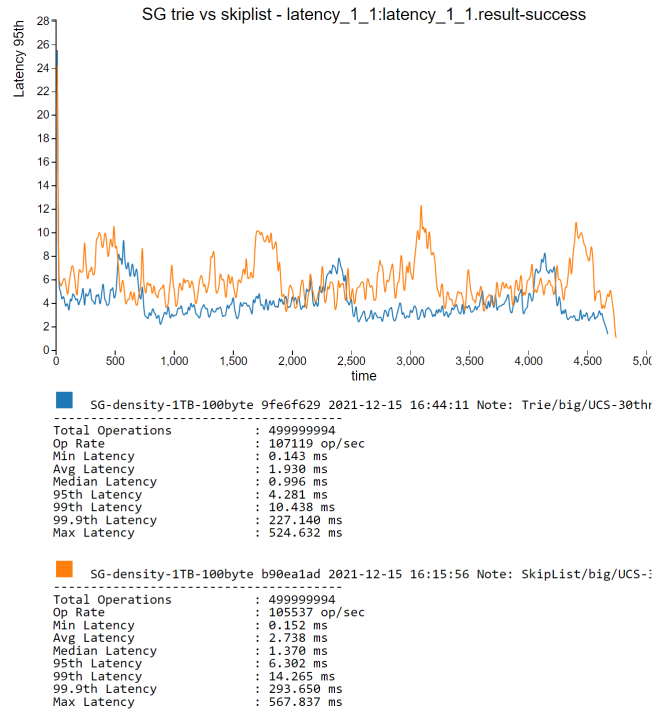

Figure 11: Latency test at fixed throughput (110k ops/sec), where half of the operations are reads. Lower values are better.

these improvements in place for both memtable solutions, the same test shows sustained levels of throughput beyond double of what is achieved with the legacy memtable, as shown in Figure 9. One of their effects is that compaction can use more resources and in effect slow down incoming writes to keep the LSM tree in a proper state; this can be seen as the periodic dips in the performance graph, caused by compaction threads operating on multiple compaction levels in parallel and stealing more time from the mutation threads. Even in the low points of these dips, the trie memtable outperforms the skip list over two-fold.

The increased throughput is complemented by lower latencies at fixed throughput for all latency percentiles, shown in Figures 10 and 11 for 10 and 50% reads respectively[4].

We also looked at some stats produced during the execution of the test to validate some of the effects that the new solution is expected to bring:

Metric	Trie	Skip-list
Memtable switch count	1510	1982
Level-0 SSTable size	812 MiB	611 MiB
Old-gen GC total time	0 s	696 s
Young-gen GC total time	4290 s	10144 s

As expected, the more compact representation and the movement of structure off-heap allows the new memtable to flush less frequently and produce 30% larger SSTables on Level 0 for the same memtable memory allocation. The total garbage collection time is also dramatically reduced. Note that even though the duration of the test is different with the two memtable implementations, it performs the same amount of work which makes this a fair comparison.

8 ALTERNATIVES CONSIDERED

8.1 Sharding vs. fully concurrent structures

As commodity hardware started to become wider in recent years, one performance problem with Cassandra that became apparent was the fact that as the number of cores and hardware threads available on a node increases, the concurrency overheads imposed by sharing the skip-list map become significant enough to become a bottleneck for the performance of the database. Sharding the map, i.e. splitting it into multiple independent maps each serving a close-to-equal share of the owned token space as described in section 6, can overcome this problem and permit much higher concurrency.

[4] We only provide latency results with the Converged Cassandra branch, because we were unable to configure mainline Cassandra in a way that allows compaction to keep up with the writes. Lagging compaction leads to an SSTable distribution and resulting latencies that are not stable enough to provide trustworthy information.

One of the intermediate stages in the development of this solution, available as part of the introduction of pluggable memtables to Cassandra 4.1 with CEP-11 [18] and introduced as part of the thread-per-core architecture of DataStax Enterprise 6.0, was the sharding of the existing skip-list solution.

Cassandra 4.1 further allows one to choose whether the database should serialize writes on a shard, i.e. block other threads from writing while one is executing, or still permit concurrent writes on the individual shard. In a busy node the former typically permits higher overall throughput at the expense of a small increase in write latency, but is not as well suited to skewed workloads or ones using a non-hashing partitioner.

Our density test was also performed with the two variations of the sharding skip-list memtable; the results are presented in Figure 12. Sharding provides a very clear throughput benefit which is retained throughout the test, and the locking variation, which (by virtue of blocking threads instead of wasting CPU time and memory on competing writes) gives compaction more time to work, is able to maintain much higher throughput throughout the test.

This graph also shows that the trie solution's achievable insertion and lookup performance is far higher than the effect of sharding alone. Towards the end of the test, as more data is accumulated and performance starts being near completely determined by the accumulation of SSTables with uncompacted data, the blocking skip-list is able to perform almost[5] at the same level as the trie.

We also ran a variation of the test[6] to exercise workload skew, executed for 1/10 of the duration of the full density test.

On skewed workloads (Figure 13) the blocking solutions are still better when 1/10 of the writes go to a single partition, lose some ground when the dominating partition is referenced by every fifth write, and become pretty bad when every second write is to the congested partition. In the latter case, the concurrent skip-list is further benefiting from repeatedly hitting the same path and not having to create any new mappings to reach levels of throughput well beyond the ones in the well-distributed case, while the trie suffers further from[7] tracking lock congestion to be able to help operators identify this problem and possibly switch to an alternative. In all cases, the trie significantly reduces the garbage collection workload (Figure 14).

The results indicate that as long as the workload is only moderately skewed, the sharding solutions fare better than the fully concurrent ones even when they fully block the shard. As expected, highly skewed workloads are a weakness of the sharding solutions that block, including the new trie memtable as it stands today.

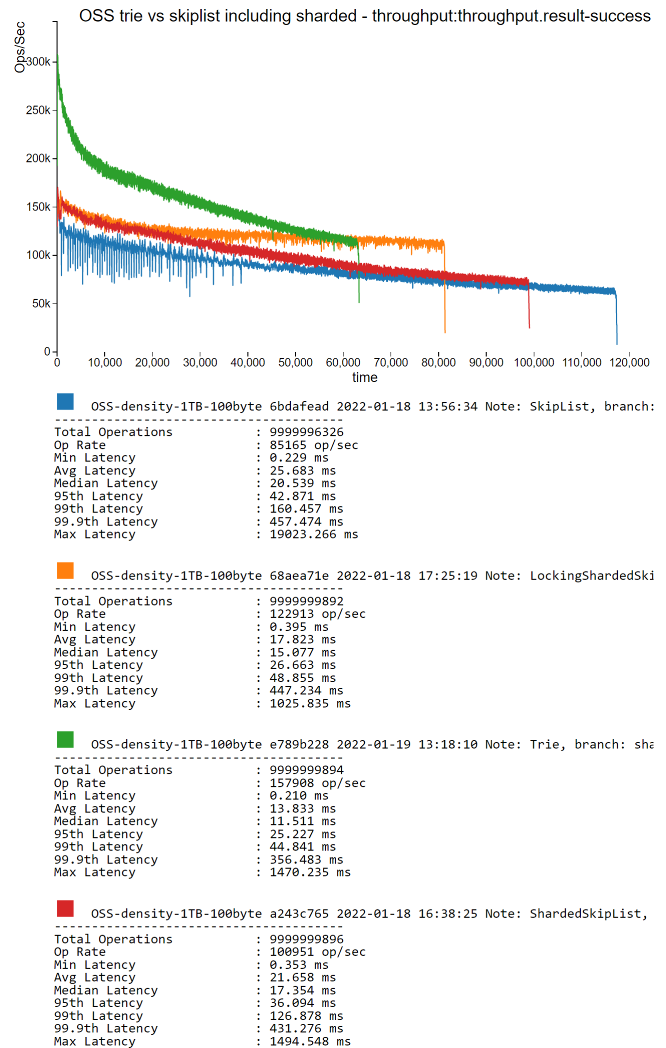

Figure 12: Throughput graph including skip-list sharding. Trie in green, blocking sharded skip-list in orange, concurrent sharding skip-list in red, skip-list in blue.

While it is possible to improve the solution's standing for these workloads, this would be at the expense of higher complexity for the ones that are more typical for Cassandra, where data is distributed among many partitions and no single partition has a double-digit percentage share of all writes.

We prefer instead to invest effort into decreasing the time a lock is held by improving the performance of individual writes, e.g. by replacing the current B-Tree hierarchy with an extension of the memtable trie, and increasing the efficiency of locking, e.g. by taking advantage of asynchronous processing or virtual threads [29]. The legacy skip-list implementation is still available as an option for those users and use cases that do exhibit high skew.

On the other hand, the skip list partition map, introduced primarily to support concurrent modification, appears not as well suited to

[5]This graph plots number of operations per second against test time; as some tests complete earlier, the performance at a certain density is not represented at the same horizontal position in the graph for all tests. In particular, even though the trie graph appears to show lower performance than the blocking skip-list at some instants of the test (e.g. @60ks), the trie graph at this point in time works with over 1 TB of accumulated data, while the blocking skip-list one is still at less than 800 GB; the performance of the skip-list for the same density is represented later (@80ks). Moreover, the latter graph has had the same amount of time to compact less information and is thus expected to benefit from better structuring of the data and read performance is less of a bottleneck.

[6]Using the script at https://gist.github.com/blambov/00e8dbff5a97f321d30d0ef992465a08. The most important difference is the introduction of a `domination_op` that changes a specified fraction of all requests to hit a specific partition.

[7]Confirmed by testing a modified version that uses the same lock mechanism as the blocking skip-list.

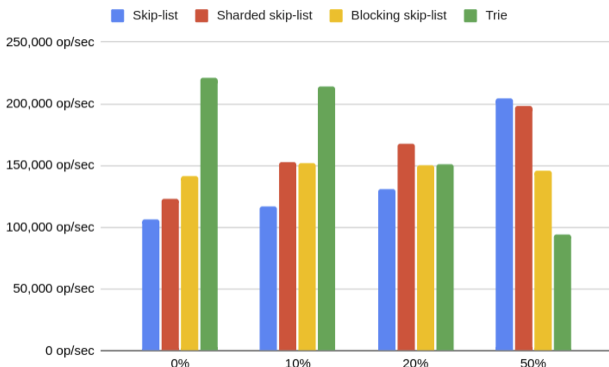

Figure 13: Average throughput on skewed workloads, higher is better. Graphs the average write throughput over 100 GB of the density workload for increasing levels of skew. The skew percentage specifies the ratio of writes and reads hitting the exact same partition.

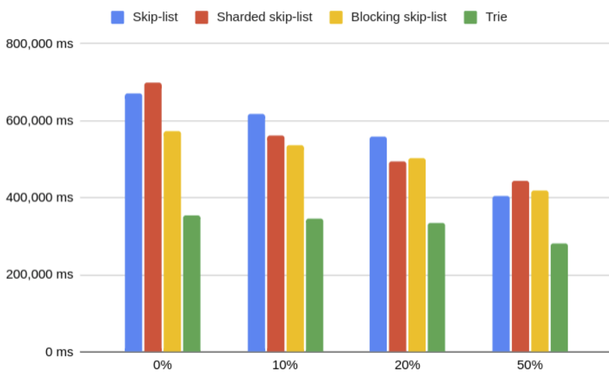

Figure 14: Total garbage collection time for skewed workloads, lower is better.

Cassandra's typical workloads as a single-writer model combined with sharding.

8.2 B-Trees

B-Trees and their variations are the comparison-based default go-to structures for databases [28]. Cassandra already implements in-memory B-Trees, and if we move away from the requirement to support concurrent writers as the previous section suggests, it becomes possible to use Cassandra's B-Tree implementation for the partition map and thus for all levels of the structural hierarchy of a memtable. That is, a memtable could be implemented as a sharded B-Tree to partitions, each being a B-Tree to rows containing a B-Tree to the columns within the partition and potentially another B-Tree for the cells inside a complex column.

Converting this implementation to avoid using heap memory for its blocks is relatively easy if we store the keys as objects on heap or in a separate memory region: B-Tree nodes have a predetermined size, and we can either use separate sections for leaf and internal node store or adjust the key count for leaves to make it match the size of an inner node.

Storing the key elsewhere has the disadvantage of requiring an extra pointer-chasing hop to read each key; additionally, the whole key is usually required to make decisions, which means that the cache space required to carry out these decisions without referring to main memory is on the order of the size of the key. While an inner-node decision in the B-Tree can carry more information than a trie block advance, it requires much more memory, and a lot more than a single cache line fetch. As finding the decision point in a B-Tree node requires ordered keys, modification concurrent with reads in such structures is only possible if we use a mechanism similar to the order word of our `sparse` nodes, used on every access, which adds further complexity. This combination of factors would make B-Trees with detached key store much less efficient than a trie.

On the other hand, storing variable-size keys with the B-Tree node may improve the fetch time for all keys of the node, but does not reduce the cache space need and no longer offers a fixed node size. Managing the tradeoff between reserved size and memory and cache requirements and the handling of various edge cases for nodes that can accept insertions is extremely difficult, thus the most feasible implementation of collocated keys would keep nodes immutable and copy on every modification. If not reused, old copies can very quickly dominate the used space and thus this kind of B-Tree requires a method of collecting and reusing variable-size nodes, a highly non-trivial undertaking, early in the development cycle; there are high chances such a mechanism would have worse consequences than leaving it to the garbage collector.

Conceptually, one can understand the memtable B-Tree hierarchy as a 4-level trie with infinite branching on every level. Furthermore, one could easily imagine further segregation in levels along the components of a key, e.g. a separate level for each component of a clustering key, necessary in order to avoid repetition of keys and to some extent mitigate the problem of having to store large keys for comparisons.

Alternatively, or as a further step, one could also make use of byte-comparable keys in a B-Tree identifying, for each B-Tree node, the common prefix and by storing in each key position only the bytes or bits that are sufficient to decide the direction of a comparison. This approach is known as a "prefix B-Tree" [2] and could in theory greatly reduce the required space per key, especially the cache space required to hold the information necessary to make decisions on a B-Tree node, at the cost of more complicated logic.

We consider what is described in the last paragraphs a complex attempt of shoehorning a trie into the framework of a B-Tree. Using a trie directly is more straightforward and drastically simpler.

8.3 ART

The Adaptive Radix Tree (ART) [23] is an existing trie implementation for database indexes that shares some key ideas with this work, including the concept of typed nodes, a special treatment of multi-byte paths, and tagged pointers for recognizing leaves. We were unable to identify an ART implementation that can offer concurrent reads, which some of our early thread-per-core deployments have shown to be very desirable. Another crucial difference

is this work's usage of fixed block sizes which facilitates better cache efficiency as well as the simplified memory management that has been a key advantage of this work.

Independently, an adaptive radix tree memtable for Cassandra may soon be available in Intel's persistent memory solution [5].

8.4 SuRF

Succinct tries [12] are a method of storing tries with what is considered to be optimal levels of space overhead, which have been applied to construct an LSM database index called Succinct Range Filter (SuRF) [31].

Because of the compactness of their representation, succinct tries are not suitable for mutable structures, and even less so for ones that should be able to be read concurrently with updates. As demonstrated by SuRF, their strength is in immutable structures such as the on-disk SSTables. Appendix A of [31] also describes the mechanism for supporting mutations in combination with SuRF: by using an LSM tree, including a dynamic trie as a write buffer. In other words, this work is complementary to SuRF or other immutable on-disk tries.

It is also worth noting that efficient access in succinct tries requires external structures, which, even as it is theoretically optimal, is not practically as efficient in a fully in-memory structure because of the additional search steps, pointer chasing hops and associated branch prediction and cache inefficiency.

9 FUTURE WORK

This is the initial application of the developed trie infrastructure for Cassandra. We believe the results it has shown are just a small part of what can be achieved and are already working on further development, but nevertheless these results are substantial enough to be worth sharing in their own right.

The next iterations of the trie memtable should increase the reach of the trie to lower levels of the data hierarchy. More specifically, at this point the trie only maps to partitions, while separate B-Trees are used to index rows within the partitions. It is possible to extend the map in a way that allows a single trie to reach the level of rows, include information related to both partitions and rows, and be able to retrieve rows or trie-backed views of partitions. This should bring significant further space savings for all workloads, and improve performance for workloads that utilize clustering keys.

This requires some further development of the trie code and interfaces, including:

- Improvements on the traversal paradigm to permit descending to specific children.
- Write atomicity support which enables in-progress readers to see a consistent view of the partition they operate on.
- Recycling and reuse of blocks.

A more complex issue is the handling of range tombstones, deletion markers that span sections of the row space within a partition. This can be done by attaching a parallel tombstone trie to each partition (or possibly any point in the trie) that contains such a deletion.

Subsequently, it is possible to go further to the lowest level of the memtable hierarchy, the cell, to completely cover all memtable indexes within the trie, which in turn would eliminate the on-heap footprint of memtables, bringing the associated garbage collection improvements.

Another direction of further development is the question of applying this paradigm to SSTables, the immutable on-disk structures that memtables are transformed to on flush. As immutable structures they offer further optimization opportunities and as on-disk ones they have different data locality and caching characteristics, but would also benefit from the advantages of representing the data in a trie such as smaller footprint, faster lookup and prefix sharing for storage as well as processing.

REFERENCES

[1] Stefania Alborghetti, Justin Chu, Dimitar Dimitrov, Branimir Lambov, and Alex Sorokoumov. 2022. *Unified Compaction Strategy.* Retrieved 2022-06-22 from https://github.com/datastax/cassandra/blob/ds-trunk/doc/unified_compaction.md

[2] Rudolf Bayer and Karl Unterauer. 1977. Prefix B-Trees. *ACM Trans. Database Syst.* 2, 1 (mar 1977), 11–26. https://doi.org/10.1145/320521.320530

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[4] Cassandra community. [n.d.]. *Apache Cassandra.* Retrieved 2022-06-22 from https://cassandra.apache.org/_/index.html

[5] Cassandra community. [n.d.]. *CASSANDRA-13981: Enable Cassandra for Persistent Memory.* Retrieved 2022-06-22 from https://issues.apache.org/jira/browse/CASSANDRA-13981

[6] Cassandra community. [n.d.]. *Fallout.* Retrieved 2022-06-22 from https://cassandra.tools/fallout

[7] Cassandra community. [n.d.]. *NoSQLBench.* Retrieved 2022-06-22 from https://github.com/nosqlbench/nosqlbench

[8] DataStax. [n.d.]. *Converged Cassandra.* Retrieved 2022-06-22 from https://github.com/datastax/cassandra

[9] DataStax. [n.d.]. *Storage-Attached Indexing (SAI).* Retrieved 2022-06-22 from https://docs.datastax.com/en/storage-attached-index/6.8/sai/saiTOC.html

[10] Nick Dimiduk. [n.d.]. *Orderly.* Retrieved 2022-06-22 from https://github.com/ndimiduk/orderly

[11] Dimitar Dimitrov, Branimir Lambov, and Jacek Lewandowski. 2022. *Byte-comparable translation of types (ByteComparable/ByteSource).* Retrieved 2022-07-01 from https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/utils/bytecomparable/ByteComparable.md

[12] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms.* Springer, 326–337. https://doi.org/10.1007/978-3-319-07959-2_28

[13] Jing Han, Haihong E, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *2011 6th International Conference on Pervasive Computing and Applications.* 363–366. https://doi.org/10.1109/ICPCA.2011.6106531

[14] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[15] Guy Bolton King, Sean McCarthy, Pushkala Pattabhiraman, Jake Luciani, and Matt Fleming. 2021. Fallout: Distributed Systems Testing as a Service. https://doi.org/10.48550/ARXIV.2110.05543

[16] Donald E Knuth. 2014. *Art of computer programming, volume 3: Sorting and searching.* Addison-Wesley Professional.

[17] Branimir Lambov. 2022. *CASSANDRA-17240 branch.* Retrieved 2022-06-22 from https://github.com/blambov/cassandra/tree/CASSANDRA-17240

[18] Branimir Lambov. 2022. *CEP-11: Pluggable memtable implementations.* Retrieved 2022-06-22 from https://cwiki.apache.org/confluence/display/CASSANDRA/CEP-11%3A+Pluggable+memtable+implementations

[19] Branimir Lambov. 2022. *CEP-19: Trie memtable implementation.* Retrieved 2022-06-22 from https://cwiki.apache.org/confluence/display/CASSANDRA/CEP-19%3A+Trie+memtable+implementation

[20] Branimir Lambov. 2022. *Compaction process optimization.* Retrieved 2022-06-22 from https://github.com/datastax/cassandra/blob/ds-trunk/src/java/org/apache/cassandra/io/sstable/compaction/cursors.md

[21] Branimir Lambov. 2022. *MemtableTrie Design.* Retrieved 2022-06-22 from https://github.com/blambov/cassandra/blob/CASSANDRA-17240/src/java/org/apache/cassandra/db/tries/MemtableTrie.md

[22] Branimir Lambov. 2022. *Trie interface.* Retrieved 2022-06-22 from https://github.com/blambov/cassandra/blob/CASSANDRA-17240/src/java/org/apache/cassandra/db/tries/Trie.md

[23] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[24] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[25] Patrick McFadin. 2020. *Databases Should Be the Most Boring Thing in Your Data Center*. Retrieved 2022-06-22 from https://www.datastax.com/blog/databases-should-be-most-boring-thing-your-data-center

[26] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)* 15 (1968), 514 – 534.

[27] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[28] Alex Petrov. 2019. *Database Internals*. O'Reilly Media, Inc.

[29] Ron Pressler and Alan Bateman. [n.d.]. *JEP 425: Virtual Threads*. Retrieved 2022-06-22 from https://openjdk.java.net/jeps/425

[30] Christof Strauch. 2011. NoSQL databases. Retrieved 2022-07-01 from https://www.christof-strauch.de/nosqldbs.pdf

[31] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336. https://doi.org/10.1145/3183713.3196931