Concurrent Extendible Hash Table - Final Report

Authors: Shivang Dalal (shivangd), Ashwin Rao (arao2)

URL: https://sdtheslayer.github.io/Concurrent-Extendible-Hash-Table/

Project Summary

We implement three versions of a thread-safe extendible hash table (supporting concurrent inserts, retrievals and deletes): one using coarse grained locking, one using fine grained locking, and one lock-free implementation using atomic hardware operations like Compare And Swap. We benchmarked and compared performance (throughput, latency, speedup) and memory usage for each of these implementations across a diverse set of read and write workloads. Using these comparisons, we draw conclusions about the tradeoffs of the three approaches, and which implementation makes sense to use under different workloads. All our testing was done on 8-core CPUs on the GHC machines.

Background

Hash tables play a crucial role in database management systems (DBMS), particularly for joins and indexing. Traditional lock-based implementations often suffer from contention when multiple threads attempt to access the data structure simultaneously. Our lock-free design addresses these limitations by enabling parallel access without explicit synchronization mechanisms. We will also implement and compare performance with fine-grained and coarse-grained locking implementations and further discuss the different scenarios where these might be useful within a DBMS system.

Limitations of the Locking Approach

- Lock-based implementations create contention points.
- Traditional resizing mechanisms block concurrent access.
- High locking overhead even in low contention systems, especially with fine-grained locks.

Industry Applications of Extendible Hashing

- Database indexing systems.
- In-memory caching and in-memory databases.
- High-frequency trading platforms.
- Real-time analytics systems.

Extendible Hash Table

Before getting to our approach, first a brief about extendible hash tables and how they work. An extendible hash table is a dynamic data structure that implements a directory-based hashing scheme optimized for disk-based indexing and exact match queries. Unlike traditional hash tables that require complete rehashing during resizing, extendible hashing performs localized reorganization (rehashing) by splitting only the overflowing buckets and doubling the directory when needed. The system maintains two critical parameters: a global depth associated with the directory (determining the number of bits used for indexing) and a local depth for each bucket (indicating the number of bits needed to distinguish entries within that bucket). The operations on the extendible hash table are explained below, followed by a diagram-based example to depict how the hash table works.

Hashing a key to a bucket

- Apply a hash function to the input key.
- Extract the lowest ‘x’ bits based on the global depth of the directory.
- Follow the directory pointer to the required bucket.
- Insert, retrieve or delete the key from the bucket.

Inserting a key

Find the bucket to insert into. If the bucket is not full, insertion is simple – just the key value pair to the bucket. If the bucket is full, there are two scenarios:

1. If the bucket’s local depth < global depth, create two new buckets with local depth = old local depth + 1, and rehash all keys in the bucket to reorganize them between the two new buckets.
2. If the bucket’s local depth = global depth, do the same as above, but also increment the global depth (that is, double the directory size), and update the directory pointers to buckets based on their local depths (see example below).

Deleting a key

Find the bucket to delete from. If the bucket is not empty, simply delete the key-value pair. If the bucket is empty, there are two scenarios:

1. If the bucket’s sibling has equal local depth as this bucket, then delete this bucket and decrement the sibling bucket’s local depth.
2. Otherwise, do nothing.

Walking through an example

In the below example we are using the first two bits of the hashed key to index into the hash table (that is, global depth = 2). When we insert a third entry into the second bucket it overflows and we have to perform a global split. Since the bucket’s local depth is equal to the global depth, we double the directory size and update the directory’s pointers to buckets. Note that only the values in the overflowing bucket are rehashed between two new buckets with local depth 3 (bucket size = 2).

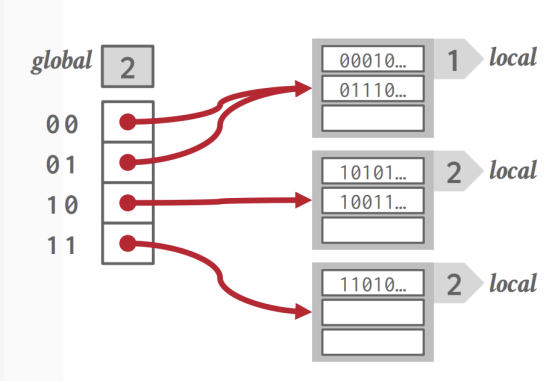


Fig.1 Initial buckets

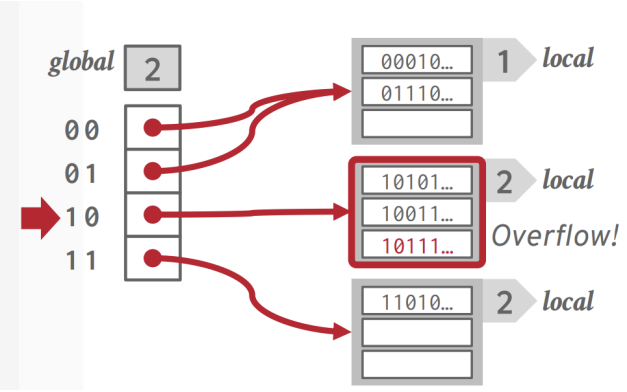


Fig.2 Trying to insert a third entry causes a overflow

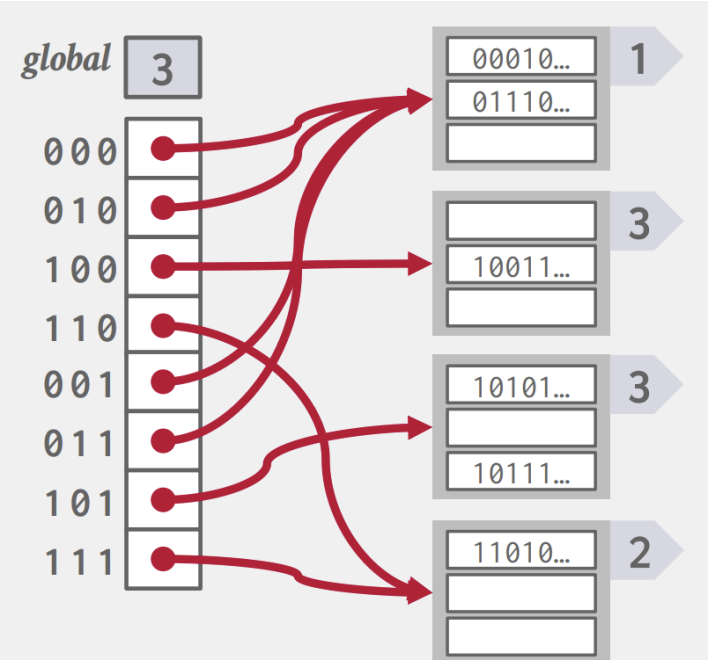


Fig.3 Resulting table after split but before insert.

Concurrent extendible hash tables

When ensuring thread-safety of the above operations, the directory is the main contention point across threads on insert and delete, since the directory is either read or modified in all extendible hash table operations. Further, thread-safety of individual bucket operations also needs to be handled, specifically while creating and deleting buckets. There are therefore two different contention points: the global directory, and each local bucket. Our coarse grained approach focuses on ensuring correctness of directory reads/writes in a multithreaded environment. Our fine-grained approach builds on this by reducing directory contention, with the tradeoff of ensuring correctness of concurrent bucket operations. Finally, our lock-free approach eliminates lock-based contention on both the directory and each bucket.

Lock based implementation

The locking-based extendible hash table implementation follows a two-tier architecture as follows:

1. A **Directory** class that manages the public-facing API of insert, get, remove and update. The directory maintains a vector of bucket pointers and handles dynamic growth through bucket splitting. It maintains the hash table's global depth.
2. A **Bucket** class, the objects of which store the actual key-value pairs. Each bucket implements a map-based storage system with configurable size limits, and maintains its own local depth mentioned above to manage the dynamic growth and splitting operations.

Coarse-grained locking implementation

The coarse-grained synchronization approach implements thread safety through a single directory-level shared mutex that guards all operations. The insight into this approach is that **all hash table operations must go through the directory**, so a directory-level lock is sufficient for correctness (although inefficient under load). This implementation prioritizes simplicity and correctness over performance by serializing all accesses to both the directory structure and individual buckets. The shared mutex (**directory_mutex**) enables read-write lock semantics, allowing multiple readers to access the structure simultaneously while ensuring exclusive access for writers. While this approach guarantees thread safety and prevents race conditions, it creates a significant bottleneck as all operations, regardless of whether they access different buckets, must acquire the same lock. This of course works very well under serial or light concurrent load as is visible from the benchmarks below, but quickly falls off under heavy concurrent load. The one saving grace it has is a lower memory footprint and low locking overhead (since there is only one lock to acquire), but that minor benefit is not worth the drop in performance.

Key Features:

- Single directory_mutex guards all operations
- Shared locks for read operations
- Exclusive locks for write operations
- Simple but limited concurrency

Fine-grained locking implementation

The fine-grained implementation significantly enhances concurrency by implementing a sophisticated two-level locking strategy. At the directory level, a shared mutex controls structural modifications, while individual bucket-level mutexes enable concurrent operations on separate buckets. This implementation carefully manages lock acquisition order to prevent deadlocks and minimizes critical sections by releasing locks as soon as possible. Specifically, care is taken to hold the directory lock for as little time as possible to prevent contention on the global directory. The system employs early lock release strategies, particularly in read operations, where the directory lock is released immediately after identifying the target bucket. Similarly, the write operations are optimized for the general case of an insert not splitting or merging a bucket (the directory lock is released early and reacquired **only if** the insert or delete operation needs to modify the directory, which is the rare case). These optimizations lead to a significant improvement in throughput and latency, especially for larger thread counts and buckets, with only a minor increase in memory footprint.

Key Optimizations:

- Directory-level lock for structural changes and modifications
- Bucket-level locks for data operations
- Lock ordering for deadlock prevention
- Early lock release strategies
- Smart lock management during bucket splits

Lock-free implementation

Both the above implementations still suffer from contention on directory and/or bucket locks. To mitigate this issue, one method is to remove locks altogether, and achieve thread-safety in a different way. We replace locks with hardware primitives and atomic operations like CAS (Compare and Swap). CAS allows a thread to update a value only if it has not been changed by another thread since the last read. This design allows multiple threads to operate on the hash table concurrently without blocking each other, enhancing performance in multi-threaded environments. The downside is the need to sometimes busy-wait in a loop to retry operations if CAS fails due to concurrent operations – even then, the benefit of removing locks is evident under light workloads, as seen in our benchmarks.

Our implementation of the lock-free extendible hash table has two components:

1. A lock-free Bucket with concurrent insert/get/remove operations, that we implement using a lock-free linked list. For better time complexity, we also keep the linked list sorted by key.
2. The actual extendible hash table implementation that uses the lock-free Bucket as a primitive.

Lock-free Linked List implementation

Each linked list node contains a key value pair and an atomic next pointer. The atomic pointer ensures that inserts and deletes from the linked list are thread-safe. The linked list also maintains an atomic pointer to the root, and an atomic count of the number of elements in the list for constant-time size calculation. Atomic updates to these atomic variables are done using the Compare and Swap operation. If CAS fails (for example, due to a concurrent operation), the operation is retried in a loop until it succeeds. Standard algorithms are used for insert, delete and find operations, modified to use CAS with retries.

Lock-free Extendible Hash Table implementation

The lock-free extendible hash table uses the same algorithms as the lock-based implementation. The key difference is that Directory and Bucket locks are replaced with atomic operations.

1. The **Bucket** class is replaced by the lock-free linked list implementation. This effectively replaces Bucket locks with atomic operations to ensure concurrent reads/writes to a bucket.
2. The **Directory** lock is removed. The lock-free extendible hash table instead contains an atomic array of pointers to buckets (each bucket is the head of a lock-free linked list), along with atomic variables to store the total number of elements and number of buckets in the extendible hash table. In the rare case that the directory is modified on insert/delete, the entire array is updated atomically using CAS (this is fast since the array is simply a double pointer).

Concurrent inserts, retrievals and deletes are implemented using lock-free Bucket primitives and CAS to update the buckets array in the directory when new buckets are created (on insert) or freed (on delete), with logic similar to previous implementations. Another optimization we implemented is initializing lock-free Buckets on-demand rather than all at once during hash table construction – this reduces the memory footprint of the extendible hash table. Because of this, we also require CAS during bucket initialization to ensure that each bucket is initialized only by a single thread (CAS fails when a thread tries to initialize an already initialized bucket).

The lock-free implementation avoids deadlocks since there is no locking. Even under high contention, some threads always make progress. In the benchmarking section, we explore how our lock-free implementation achieves significant speedup over lock-based versions for light and medium workloads, while speedup degrades with extremely heavy workloads due to higher contention on atomic variables, resulting in many CAS retries.

Experimental Results

All of our experiments were performed on the 8 core GHC cluster (on different machines but all having the same configuration). Hence most of our parallel experiments run up to 8 threads. We designed multiple benchmarks based on the use cases seen in database systems.

Benchmarking suite

The benchmarking suite implements two distinct testing approaches for evaluating concurrent hash table performance. For each approach, we measure both the duration takes as well as peak memory usage.

The Mixed Test represents a real-world scenario where multiple readers and writers operate simultaneously on the data structure - writers insert new values while readers attempt to retrieve them, simulating a concurrent access pattern common in hash table indexes.

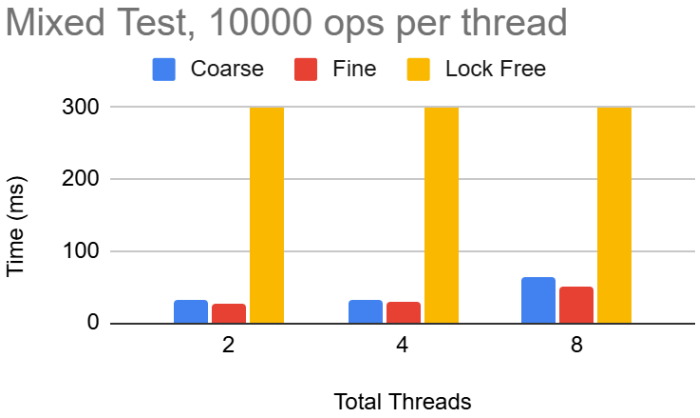
The CRUD Test, in contrast, follows a more sequential pattern where each thread performs a complete cycle of operations (Create, Read, Delete) in order, providing insights into how the data structure handles hash join type workloads.

The suite offers extensive configurability through command-line parameters to simulate various workload scenarios:
-r: Controls the number of dedicated reader threads
-w: Sets the number of writer threads for concurrent operations
-n: Defines the workload size by specifying operations per thread
-b: Configures the underlying bucket size of the hash table
-i: Determines test reliability by setting the number of iterations for averaging results

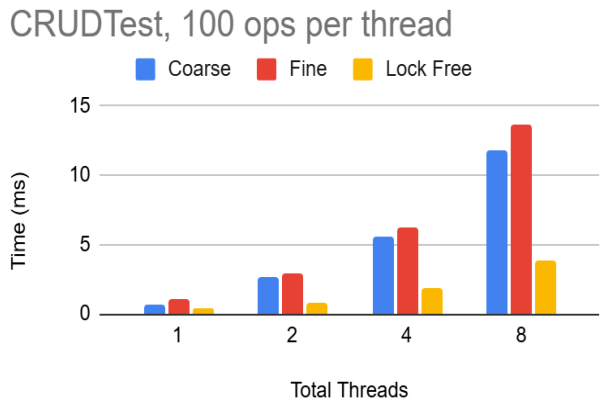
We used these parameters to run experiments and plot the below graphs and other analysis.

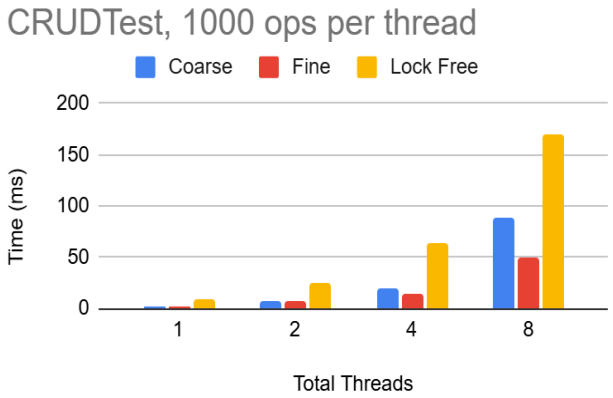
Throughput comparison

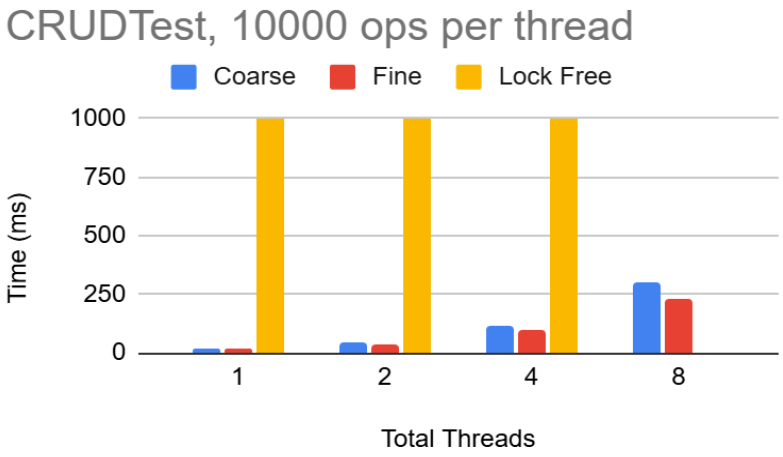
We start by doing a comparison **on the mixed test** with varying levels of operations per thread, with a bucket size of 20 and equal ratio between readers and writers. We repeat the experiment in a multithreaded environment with 2, 4 and 8 threads.



Next, we perform a similar comparison **on the CRUD test**, again with varying levels of operations per thread, bucket size of 20 and equal ratio between readers and writers. We repeat the experiment in a multithreaded environment with 2, 4 and 8 threads.







As can be seen from the graphs in both suites, **the lock free implementation performs the best under low contention and low throughput.** This is due to the absence of lock-based contention and overhead that slows down the lock-based implementations. However, as the load increases – especially at 10000 operations per thread – the lock-free performance falls off a cliff. As discussed before, **we attribute this to the sheer busy waiting and contention on the atomic operations at high contention,** including the performance of the lock free sorted list that uses atomic operations. At the highest load, the lock free implementation is up to 5-10x worse than the lock-based approaches. We could not get the lock-free implementation to perform 80k insert operations due to high contention, so the graph in this case is empty.

The comparison between coarse and fine is more straightforward, with the two starting out almost equal under light load in both the Mixed test and CRUD test. This makes sense due to less locking contention and overhead. Under heavy load however, the fine-grained version becomes the clear favourite, performing significantly better in the Mixed test with parallel readers and writers. This is observed in the CRUD test as well (with mostly writers) – fine grained performs significantly better than coarse grained. The explanation is simple: **the coarse-grained implementation suffers from high contention on the global Directory lock, resulting in poor performance.** The fine-grained implementation achieves much better concurrency since multiple threads can operate on different buckets simultaneously, while in the coarse-grained implementation, all threads are serialized on directory access.

Overall, our observations on throughput can be summarized as follows:

- The lock-free implementation is highly efficient on light and medium workload.
- The fine-grained locking implementation works the best under heavy workload.
- Coarse-grained implementation can be preferred when we require a simple implementation for light workloads.

Memory and Speedup comparison

Although we initially thought that the memory overhead of fine-grained locking would be significant, **all three variants remained under 5% of each other in all of our experiments.** Our analysis revealed that the locks itself occupied very little space per bucket compared to the actual data. Had we chosen a more compact representation of the hash table buckets it might have been more significant.

Similarly, throughout our experiments, the time to perform some number of inserts distributed across multiple threads was always worse than a single thread performing all operations. This is because the critical section (the amount of actual work that needs to be done by each thread) is not too significant – all we are doing is a couple of inserts. In a real world application, between each insert call, there would be a lot more computation – in this case, the system would benefit greatly from our parallelization. We did simulate this by making each thread sleep for a random amount of time after every operation, which made the parallelization overhead insignificant. Additionally, our computation time also includes the time it takes to spin-up a thread pool, and this negatively impacts our results comparing performance to a single-threaded test suite.

Conclusion

Based on our extensive experimentation and analysis of three different implementations of concurrent extendible hash tables, we can draw several important conclusions. The lock-free implementation showed superior performance under light to medium workloads but suffered significant degradation under heavy concurrent operations due to CAS retry overhead and atomic operation contention. The fine-grained locking approach emerged as the most balanced and practical solution, particularly for high-load scenarios, demonstrating significantly better performance than coarse-grained locking while maintaining similar memory overhead (under 5% difference). The coarse-grained implementation, while simple and effective for light workloads, proved inadequate for heavy concurrent access patterns. Contrary to our initial expectations, the memory overhead differences between implementations were minimal, suggesting that the choice between these implementations should primarily be based on expected workload patterns rather than memory constraints. These findings provide valuable insights for database system designers, indicating that fine-grained locking may be the most suitable choice for production environments where consistent performance under varying loads is crucial, while lock-free implementations could be beneficial for specialized light-workload scenarios.

References

Drawn some inspiration from the following research papers on lock-free extendible hash tables:

- Shalev and Shavit, "Split-ordered lists: Lock-free extensible hash tables," Journal of the ACM, 2006.
- Fatourou, Kallimanis, and Ropars, "An Efficient Wait-free Resizable Hash Table," 2018.

We also used CMU's Intro to DB (15-445) slides for explaining Extendible hashing.

Distribution of work and credit

Credit distribution: 50% - 50% between Shivang and Ashwin.
Focus areas:

1. Initial research (Shivang 50%, Ashwin 50%)
2. Coarse grained locking (Shivang 60%, Ashwin 40%)
3. Fine grained locking (Shivang 60%, Ashwin 40%)
4. Lock free hash table (Shivang 30%, Ashwin 70%)
5. Benchmarking (Shivang 50%, Ashwin 50%)
6. Report (Shivang 60%, Ashwin 40%)
7. Poster (Shivang 40%, Ashwin 60%)