15-618: Final Project Report

Paulina Davison (pldaviso)
Shrey Bagroy (sbagroy)

Summary:

We evaluate four methods to improve performance of concurrent sequential scans, which scan a table in-order, for the terrier database. The original design uses coarse-grained locking. We explore fine-grained locking, duplication of the data table, using a concurrent data structure, and designing a lock-free list. We present graphs which highlight the effectiveness of the fine-grained locking implementation and the lock-free list implementation, displaying the time elapsed and the number of items processed per millisecond of benchmarks run on an Intel Xeon Broadwell CPU on the CMU Parallel Data Laboratory network.

Background:

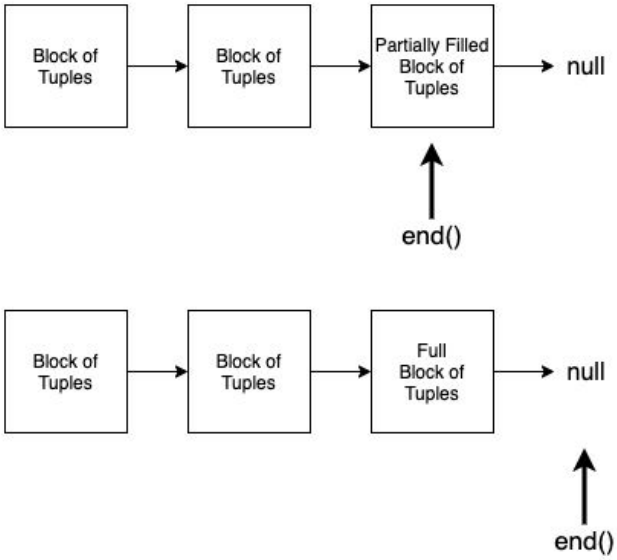
The Terrier Database is designed to support fully autonomous optimization of hybrid workloads (https://db.cs.cmu.edu/projects/peloton/). It is an in-memory, column-store database.

Our project improves the scan and insert functionality of the Terrier Database. The scan function implements a sequential scan of data, meaning that a thread reads contiguous data from as many of the blocks as it needs to. The scan function takes as inputs the context of the transaction that is requesting the scan, an iterator that marks the starting position of the scan, and a buffer that will be populated with the scanned tuples. The buffer parameter can be considered its output. The insert function inserts a new tuple into a free slot in the blocks. The ordering of the tuples does not matter, which allows different threads to insert into different blocks, allocating a new block and appending it to the list if needed. The insert function takes as inputs the context of the transaction that is requesting the insert and the tuple to insert. The insert function outputs the physical location, the block and offset in the block, that represents where the tuple was inserted.

The key data structure of this project is the data table, which stores the tuples of a single SQL table. The system requires that the data table be composed of

blocks that each store a set of tuples. In the Terrier Database design, each block is 1 MB. Though the order of the blocks does not matter, the system must be able to determine when a new block needs to be allocated. Blocks are allocated lazily, which means that a new block will only be allocated if all other blocks are full or unavailable because they are being inserted into by another thread. Though the data table must support scanning and inserting tuples, it is not required to support deleting tuples. This also means that the data structure used to implement the data table does not need to support deletion. In the original design, the data table is a doubly linked list of blocks, implemented by the Standard Template Library (STL) list. In the process of improving parallelization, we explore other possibilities including using a concurrent ordered map and using a singly linked list.

In the original design of the scan function, there are two places that locks are being used (in databases they are called latches - please overlook this - the implementation uses spin locks). Both are for iterators. The first is in a function called end() that is used to update the pointer to the last block of the linked list. The end pointer will either point to the last partially filled block in the list or the nullptr at the end of the list, if the last block is full:



A lock is taken in the end function to make sure that the end pointer is not being updated by multiple threads simultaneously. The overloaded operator ++ also takes a lock. This operator is responsible for moving a pointer to the next tuple in a block or, if at the end of a block, the next block.

In the scan function there is a pointer defined as start_pos, which points to an attribute in the block. There is a while loop that supports the start_pos pointer in moving through each of the tuple attributes in the blocks. In this while loop predicate, the end function is being called to determine that the start_pos pointer is still in scope. In the while loop before the next loop is run, the ++ operator is called on the start_pos pointer to move it to the next tuple. This means that locks are taken twice in every while loop iteration. In addition, since both of these locks are being updated at the tuple-level, not the block-level, they are being acquired more often than is necessary. In this design, The bottleneck to the system is the iterator and the locking that is being done to support it.

Though the scan function and insert function are both amenable to data parallelization, e.g. scanning or inserting a vector of tuples rather than individual tuples, the system currently provides the flexibility to specify any number of tuples to be scanned and requires that only one tuple be inserted at a time. We chose to work within this design decision.

To measure the performance of our improvements, we chose to use the metrics of time to complete each benchmark and the number of tuples (in the millions) that were processed per second. The first metric provides a measure of the speedup obtained. The second provides a measure of throughput, which is important because of the system may want to support a large number of tuples (in the millions) being concurrently scanned and/or inserted.

Approach

In this section, we discuss the methodology that we followed in order to make inserting into and reading the database more efficient. We provide the following as an overview of our pipeline:

- Writing benchmarks: Evaluate the current (baseline) implementation and test our implementations against the baseline.
- Fine-grained locking: Locking specific parts of the code as opposed to holding the lock throughout the function execution.
- Data table copy: Providing a snapshot of the data table (i.e linked list) to the iterator as opposed to the most up-to-date version.
- Lock free iterator functions: Removing locks from iterator++() and iterator.end()
- Concurrent data structures: Using unordered maps to swap out the underlying linked list.
- Lock-free list: Modifying the underlying linked list to allow for lock-free insertion.

For coherence, we provide this pipeline all together in this order, however, the subsequent methods we mention came as a result of analyzing the results of the previous steps and brainstorming about the best possible next steps. We provide all our methodology in this section and present the results (what did/didn't work) as a part of the next section.

We start with the existing implementation of the Terrier codebase available in this repository. The codebase for Terrier is written in C++, the specifics can be found on the wiki. As a result, all the code we wrote as a part of this project is also in C++. Throughout the project, we worked with a shared address space model which is the programming model that Terrier uses. Thanks to the DB lab, we got access to an Intel Xeon Broadwell CPU (E5-2630 v4 @ 2.20GHz) on the CMU Parallel Data Laboratory network, which we use for all of our experiments. This CPU has two sockets with 10 cores per socket and 20 threads per socket.

Since our task was to improve a specific parallel component (inserting/reading from the database), we do not change the underlying paradigm and instead

maintain consistency with the rest of the codebase. We now elaborate on each step of our pipeline.

Benchmarking

Our task is to improve the current implementation of database insertions and reads in the Terrier system. In order to measure the performance of the current (baseline) implementation and evaluate the improvement in performance through our experiments, we first needed to write some performance benchmarks.

The first benchmark(s) we write is called the ConcurrentScanXX. This benchmark highlights the Scan() operation in the terrier database which reads through the database. This benchmark has XX number of threads (varying from 1 to 32) which indicates the number of concurrent threads scanning the database. The benchmark creates a dummy database, inserts 10 million tuples and then spawns XX threads to scan through the database. Each thread is mapped to a physical (or virtual, in the case of >20 threads) core.

As described in the background section, the bottleneck for the read/insert operations is the iterator. Specifically, the current implementation of the iterator requires locks whenever we read the end of the iterator or try to increment its position. This bottleneck is what is (primarily) responsible for the suboptimal performance we find in the ConcurretScanXX benchmarks. To expose this bottleneck, we write the ConcurrentIteratorXX benchmark which first builds a dummy database (as above) and then simply traverses the blocks (not the individual tuples) in the database.

The two benchmarks mentioned above insert tuples sequentially and then concurrently read through the database table by calling Scan(). We write the ConcurrentInsertXX benchmark(s) to evaluate the performance of concurrently inserting into the database. As a part of this benchmark, each of the XX writers (again, between 1 and 32) insert 10 million tuples into the database.

Lastly, we write a workload to ensure the correctness of any modifications we make to the Terrier codebase. Though we ensure throughout development that our changes pass every existing unit tests written for the database, we write this

workload in order to ensure there are no concurrency issues, specifically data races, occurring between the insert function and the scan function. This workload uses 16 threads to concurrently scan and insert into the database.

Fine-grained locking

The current implementation for Terrier uses coarse-grained latches (custom scoped spin locks) to protect the critical section in the database insert() function as well as all the operations performed by the database iterator (operator++(), end()). These latches are acquired at the beginning of these functions and are held for the entire scope (usually until the end of the function). Our first strategy is to replace the coarse-grained locking with fine-grained locking. Specifically, we introduce standard mutexes and wrap them around parts of the code that are not thread-safe.

Duplicating the data table

Our next strategy specifically targets the latches around the database iterator operations. We find that the current implementation always acquires a lock when incrementing the iterator or reading the last block. In an attempt to reduce the synchronization overhead of acquiring these locks, we experiment with creating a snapshot (i.e, a copy) of the current database and feeding it to this iterator. As a result, the iterator only ever works on this acquired local copy which eliminates the need for locks in these operations completely.

Concurrent data structures

To reduce the amount of time spent on synchronization overhead associated with acquiring the lock, we attempt to change the underlying database data structure from a linked list to a concurrent data structure. Concurrent data structures are intended to allow concurrent insert and update. We use a concurrent ordered map from Intel's Threading Building Blocks (TBB) library, which implements concurrent functionality using both fine-grained locking and lock-free techniques.

Lock-free list

On analyzing the results we obtain from our previous experiments (see the Results section), we find that the expected behavior of the Terrier database system allows us to get away with no synchronization at the time of reading from

the linked list as long as we can synchronize linked list inserts. Since the system does not require a Delete() function, we can get away with a lock-free linked list that supports Scan() (without locks) and a lock-free implementation of Insert(). The pseudocode for our implementation of the Insert function is as below:

```
  void insertIntoList(RawBlock* new_block)
  {
    Node* new_node = new Node(new_block, nullptr);
    Node* current_last_node = last_node;
    Node* expected_last_node = nullptr;
    while
(!current_lastnode.next.compare_exchange_weak(expected_lastnode,
new_node)) {
      if (expected_last_node) {
        last_node_ = expected_last_node;
        expected_last_node = nullptr;
      }
    }
    last_node_ = new_node;
  }
```

Results

In this section, we present the results we obtain from the approach we followed. First, we present the performance statistics from our benchmarks corresponding to the current implementation of the Terrier database. As discussed in the Background section, our chosen metrics for performance are: 1) total time taken and 2) number of (database) items processed per second. The table below contains the baseline results.

Benchmark	**Time elapsed (ms)**	**Comparison with sequential version**	**# items processed/sec (in millions)**	**Comparison with sequential version**
ConcurrentScan1	2513	1.00x	3.79	1.00x
ConcurrentScan2	2997	0.84x	6.36	1.68x
ConcurrentScan4	3562	0.70x	8.43	2.22x
ConcurrentScan8	5541	0.45x	13.75	3.63x
ConcurrentScan16	5548	0.45x	27.53	7.26x
ConcurrentScan32	5708	0.44x	53.46	14.10x
ConcurrentInsert1	3493	1.00x	2.73	1.00x
ConcurrentInsert2	6742	0.52x	2.82	1.03x
ConcurrentInsert4	8211	0.43x	4.64	1.70x
ConcurrentInsert8	11166	0.31x	6.83	2.50x
ConcurrentInsert16	17918	0.19x	8.52	3.12x
ConcurrentInsert32	33989	0.10x	8.98	3.23x
ConcurrentIterator1	772	1.00x	12.35	1.00x
ConcurrentIterator2	1610	0.48x	11.85	0.95x
ConcurrentIterator4	4488	0.17x	8.5	0.69x

ConcurrentIterator8	18329	0.04x	4.16	0.34x
ConcurrentIterator16	60588	0.01x	2.52	0.20x
ConcurrentIterator32	123904	0.006x	2.46	0.19x

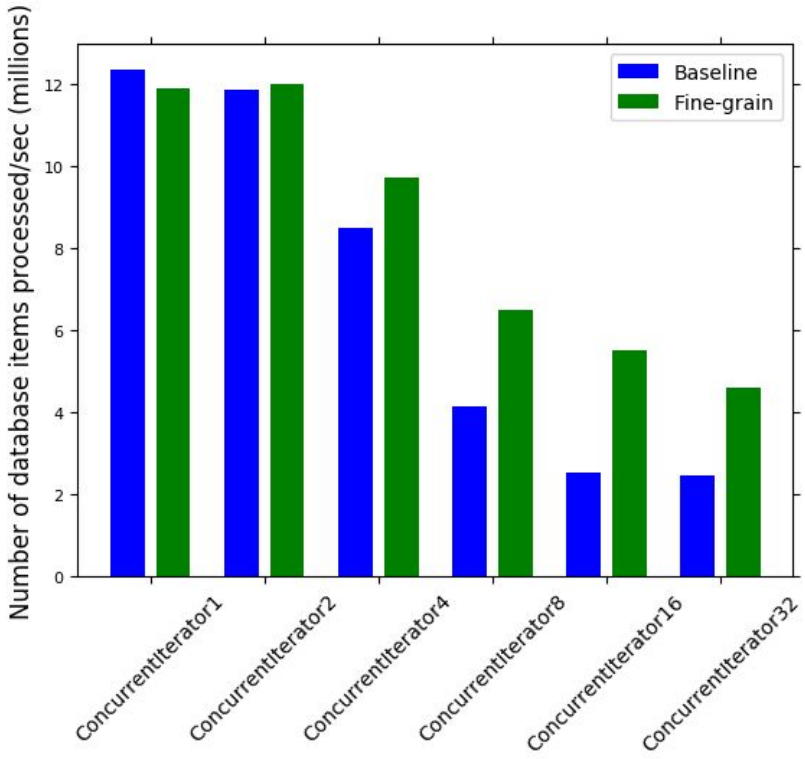
Following the discussion from the benchmark section, we know that the sequential version of all three benchmarks read (or insert, respectively) 10 million elements each. In the parallel implementations, each thread reads (or inserts) 10 million elements each. As a result, we are trying to achieve a higher workload within the same amount of time as the sequential version of the program. Thus, the ideal numbers in column 3, where we compare the elapsed time of the parallel version with the sequential version, should be 1.00x. For the last column, where we compare the number of database items processed, the ideal value for should be equal to the total number of threads, which would indicate a speedup of that much.

With this context, we evaluate the table above and find an overall poor performance for all three benchmarks. We find a considerable drop in the time elapsed (i.e an increased amount of time) for ConcurrentScan/ConcurrentInsert in column 2 with increasing number of threads. Further, we find that while there is an increase in the number of database items processed with an increasing number of threads, this increase is very small and not even close to being proportional to the number of threads, which is what one expects. This leads us to conclude that there is a bottleneck somewhere within these functions.

Our hypothesis is that the bottleneck lies in the iterator operations described earlier, i.e, the iterator increment (iterator++()) and reading the last block in the iterator (iterator.end()). To validate this hypothesis, we write the ConcurrentIteratorXX benchmarks. On evaluating the results corresponding to these benchmarks, we see that the performance drops substantially with an increasing number of threads. In fact, the last column showcases a major drop in the number of database items processed per second, indicating that it is actually better to use a sequential implementation as opposed to using more than one thread! Clearly, this benchmark exposes the contention issue between threads that is plaguing the system.

Since we validated our hypothesis that the bottleneck is due to synchronization overhead between threads in the iterator (and the Insert()) function, we then experimented with a host of different methods to mitigate this bottleneck. For brevity, we do not provide tables like the one above corresponding to each of the methods we experiment with. We provide speedup plots for the intermediate methods and a table for the best speedup we found across all our methods.

Fine-grained locking: We moved to a fine-grain locking scheme as opposed to having a function-wide (or scope-wide) latch. We see some minor improvements following this method, however, the speedup and increase in number of items processed is not substantial (plots below).

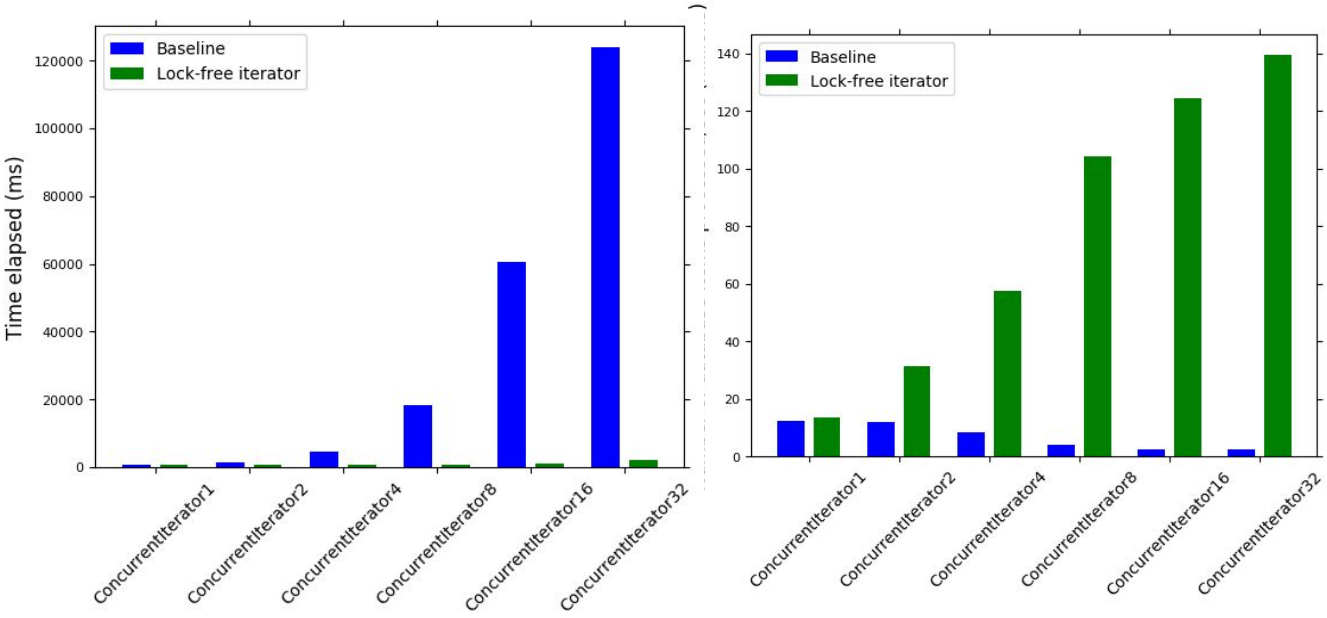


Duplicating data table: Our next idea was to duplicate the entire data table itself at the time of a concurrent read. This would allow the reader to work on a local copy of the database nodes, which would eliminate any contention with other threads since each would have a local copy of its own. However, a concern we had was in the situation that, at the time of a read, if another thread tries to insert into the database, the reader would have an outdated end() point of the iterator

since it is working on a local copy. Since this concern was more about the expected behavior of the database, we took this idea/concern to Matt (PhD student in the DB lab) and brainstormed the desired behavior. After diving deep into the required behavior of the database and the workloads it is built to handle, we discovered that we can get away with a Read() to the database having a "stale" end() pointer, i.e, *a Scan() through the database does not need to represent the most up to date linked list*. Once we established this theoretically, we realized that we could get away with no locks at all. As a result, we abandoned this idea and move to a lock-free iterator.

Lock free iterator functions: The concerns with read-write conflicts are that: 1) we want up-to-date information, and 2) there might be a consequential write-after-read which could potentially lead to incorrectness in the system. According to the expected behavior of the Terrier database, neither of these factors is a concern. As a result, we can completely remove locks put in place to protect the read-write conflict between the scan() function and insert() function. Specifically, we can entirely remove the latches from the iterator++() and iterator.end() functions.

While we established the theoretical correctness of the system without these latches, we still needed to check if this leads to any kinds of invalid memory accesses. To this end, we write a workload (described in benchmarks) which concurrently scans and inserts the database. We played around with the specifications of this workload to make sure there are (more than) enough situations where there would be a read-write conflict, yet we did not come across a single segmentation fault or invalid memory access. This experiment was an additional correctness check for our methodology. Once we passed this test, in addition to all the others in the system, we ran this version of the code through the benchmarks. We find a major performance boost in the benchmarks with the numbers getting a lot closer to the ideal statistics (plots below).

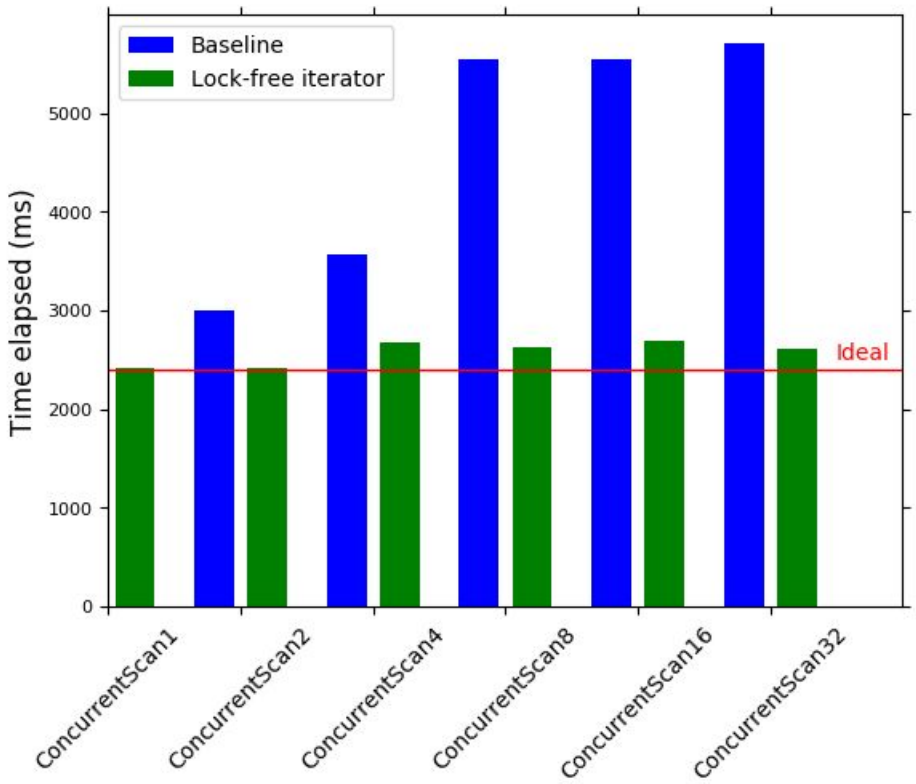


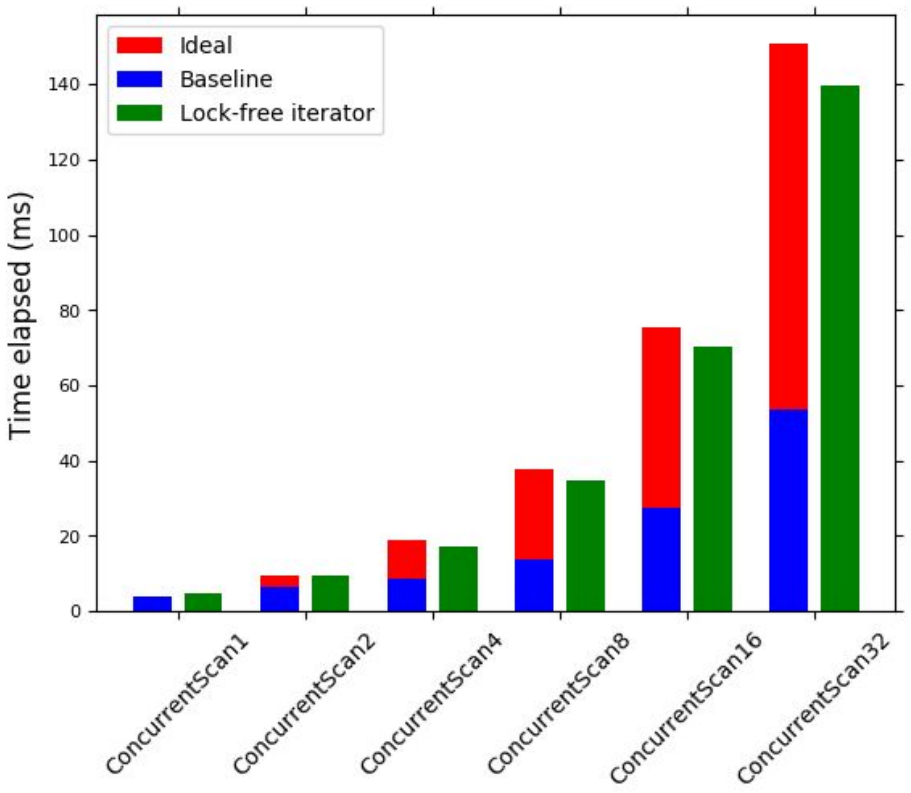
Concurrent data structures: Given that we (successfully) removed locks from the iterator functions, which primarily impact the Scan() function, our last bottleneck is the synchronization overhead of the lock in the Insert() function. Since concurrent data structures are meant to support concurrent inserts, we decided to replace the underlying linked list with a concurrent data structure that would allow concurrent inserts, ideally a concurrent linked list.

However, TBB does not have a linked list concurrent data structure. Instead it provides unordered/ordered sets and unordered/unordered maps. We first tried to use the unordered/ordered set because it fit the application requirements in two ways: the relational model used by the database does not require a specific ordering of inserted tuples and the library guarantees that any iterator that is currently traversing the list will “see” its members in the same order, which is required by the Scan() function. However, we soon realized that the Insert() function requires a consistent view of the last block in the data structure in order to realize that all the blocks have been filled and that a new block needs to be inserted. Since the unordered set is implemented internally as a hash map, it cannot guarantee this. We then tried to use a concurrent ordered map, keeping an atomically incrementing counter as the key. Unfortunately, we realized that there was an unavoidable data race between the counter and the insertion into the concurrent ordered map. As a result, this approach **did not work**.

Lock-free linked list: Since TBB did not support a lock-free implementation of a linked list, we decided to implement our own. We designed a (singly) linked list data structure that provides lock-free inserts and reads. The read function is implemented like a regular linked list; due to the lack of a read-write conflict, we can get away without any overhead at all. The Insert() function was written as per the code described in the Approach section. However, due to the large codebase and, consequently, it's dependencies between different parts of the code, we are still working on removing the last few bugs that are turning up because of our custom implementation.

Final result and best performing implementation: Due to the remaining implementation concerns with our (theoretically) best performing approach, the best performance we obtain is with the lock-free implementation of the iterator. The table with its performance on the benchmark workloads and corresponding speedup plots are below. We omit the results for the ConcurrentInsertXX benchmark(s) for brevity.





Benchmark	**Time elapsed (ms)**	**Comparison with sequential version**	**# items processed/sec (in millions)**	**Comparison with sequential version**
ConcurrentScan1	2424	1.00x	4.71	1.00x
ConcurrentScan2	2425	0.99x	9.34	1.98x
ConcurrentScan4	2673	0.91x	16.99	3.67x
ConcurrentScan8	2633	0.92x	34.69	7.36x
ConcurrentScan16	2693	0.91x	70.44	14.95x
ConcurrentScan32	2606	0.93x	139.73	29.6x
ConcurrentIterator1	592	1.00x	16.1	1.00x
ConcurrentIterator2	599	0.91x	31.84	1.97x
ConcurrentIterator4	652	0.91x	58.6	3.64x

ConcurrentIterator8	692	0.85x	104.66	6.5x
ConcurrentIterator16	1244	0.48x	122.65	7.61x
ConcurrentIterator32	2255	0.26x	135.33	8.4x

Analysis of results:

- We find an enormous increase in the performance as compared to the baseline numbers from the table earlier. The performance obtained, both in column 3 and column 5, are close to the theoretical upper bound.
- There is a dip in performance for ConcurrentIterator16 and ConcurrentIterator32. This can be attributed to the fact that the OS scheduler starts scheduling threads across different sockets after a certain threshold of threads (~20) on the Unix machine. As a result, there is an unavoidable communication overhead once we move beyond a certain number of threads. Since there is no way to avoid this overhead (on this machine) or (straightforward) method to quantify the time associated with this overhead, this is an unvalidated hypothesis. We do not see this dip in ConcurrentScan16 and ConcurrentScan32 since that is a much larger workload and the CPU can schedule work in the background (**latency hiding**).
- The biggest bottleneck in this system at the moment is the synchronization overhead in ConcurrentInsert; our lock-free linked list would potentially solve this problem but we are still debugging it.
- The workload was executed on CPUs and not GPUs. This was the correct choice because the execution environment for the Terrier database is always going to be CPUs; a GPU dependent database would not be very successful.