Lazy Evaluation for Concurrent OLTP and Bulk Transactions

Lesley Wevers
University of Twente
Enschede, the Netherlands
l.wevers@utwente.nl

Marieke Huisman
University of Twente
Enschede, the Netherlands
m.huisman@utwente.nl

Maurice van Keulen
University of Twente
Enschede, the Netherlands
m.vankeulen@utwente.nl

ABSTRACT

Existing concurrency control systems cannot execute transactions with overlapping updates concurrently. This is especially problematic for bulk updates, which usually overlap with all concurrent transactions. To solve this, we have developed a concurrency control mechanism based on lazy evaluation, which moves evaluation of operations from the writer to the reader. This allows readers to prioritize evaluation of those operations in which they are interested, without loss of atomicity of transactions. To handle bulk operations, we dynamically split large transactions into transactions on smaller parts of the data. In this paper we present an abstract lazy index structure for lazy transactions, and show how transactions can be encoded to effectively use this data structure. Moreover, we discuss evaluation strategies for lazy transactions, where trade-offs can be made between latency and throughput. To evaluate our approach, we have implemented a concurrent lazy trie, on which we performed a number of micro benchmarks.

CCS Concepts

•Information systems → Database transaction processing; •Computer systems organization → Availability; •Computing methodologies → Concurrent algorithms;

Keywords

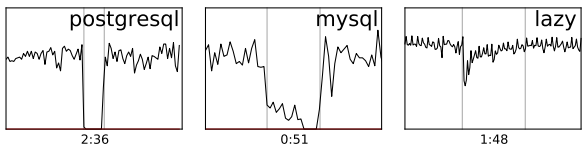
Transactions; Lazy Evaluation; Contention; Bulk Update

1. INTRODUCTION

Transactions allow multiple users to use a database concurrently while guaranteeing that their operations do not interfere. Modern database systems must handle hundreds of concurrent transactions, while providing high throughput and low latency under a variety of workloads. Standard approaches that address this challenge are two-phase locking (2PL) [5], optimistic concurrency control (OCC) [9] and multi-version concurrency control (MVCC) [2]. These approaches allow *uncontended* transactions to execute concurrently, but often fail to provide concurrency when transactions update the same data. 2PL degrades to sequential execution, as it blocks all but one of the conflicting transactions. Moreover, due to deadlocks, a conflicting transaction sometimes has to be aborted, which wastes the work that has already been performed. In contrast, OCC checks for conflicts after a transaction has executed, and aborts a transaction if a conflict has been detected. Under high contention, this often leads to repeated re-execution, and a lot of wasted work. Finally, MVCC allows a read-only transactions to execute on a snapshot of the database, which avoids concurrency bottlenecks in read-heavy workloads. However, update-heavy workloads are still problematic under MVCC. Consequently, in all standard concurrency control approaches, contended transactions experience degraded throughput and increased response latency.

The limitations of existing techniques are especially problematic for transactions that update data in bulk, as bulk operations usually contend with all concurrent transactions. To illustrate the problem, in earlier work we have measured the impact of bulk operations on the industry standard TPC-C benchmark in commonly used DBMSs [19]. Figure 1a and Figure 1b show the transaction throughput over time for PostgreSQL and MySQL when performing a bulk `UPDATE` operation while concurrently running the TPC-C benchmark. The results show long periods where the TPC-C workload is completely blocked. This is unacceptable for applications that require 24/7 availability.

In this paper we develop a concurrency control mechanism that can concurrently execute contended transactions. Figure 1c shows results obtained with our system in a micro-



(a) PostgreSQL (b) MySQL (c) Lazy

Figure 1: The effect of bulk updates on transaction throughput. The graphs plot relative transaction throughput over time. When the graph hits the x-axis, throughput is completely blocked. The grey lines mark the beginning and end of the transformation. The label below the graph shows the total execution time of the bulk operation.

IDEAS '16, July 11 - 13, 2016, Montreal, QC, Canada

http://dx.doi.org/10.1145/2938503.2938555.

benchmark, where we see that a workload of updates can continue executing during a bulk update. In our approach the bulk update commits *before* it is executed, and it is executed on-demand only for those values that are being accessed by concurrent transactions.

Lazy Transactions. The main idea of our approach is to split a transaction into operations on individual variables, and to move the responsibility of evaluating these operations from writers to readers. For this, we use a lazy evaluation mechanism that is inspired by that used in functional programming languages [8]. Instead of computing values and then writing them, transactions can write *suspended computations*, which are evaluated to a value upon reading. This allows a transaction to commit *before* these operations are executed. Moreover, readers evaluate only those operations in which they are interested, and do not have to wait for writers to evaluate their full set of operations.

For the correctness of our approach, suspended computations must evaluate to the same result as when the computations are evaluated immediately. If the data accessed by a suspended computation changes between committing and its evaluation, its result would change. To solve this, lazy operations can read from a stable snapshot that is acquired when the transaction commits. Similar to MVCC, read-only transactions can read from a snapshot without interfering with other transactions. For a transaction that updates the database, the execution is divided into four phases:

- **Preparation phase**: The transaction performs computations that do not depend on values in the database, and it may optimistically read from a snapshot.
- **Commit phase**: The transaction atomically queues lazy operations, and logically commits at the end of this phase.
- **Lazy phase**: This is the part of the transaction that is executed inside suspended computations, where it may read lazily from snapshots and perform computations.
- **Result phase**: The transaction computes its observable result from snapshots obtained in the commit phase.

We assume that the commit phase is executed sequentially, while the other phases can execute concurrently. Consequently, transactions should minimize work in the commit phase, and defer most work to the other phases to obtain concurrency. The main source of concurrency comes from concurrent reads on snapshots, where they force the evaluation of suspended computations. To minimize time in the commit phase, reads should be avoided there, as they may need to evaluate suspended computations by other transactions. Ideally, reads are deferred to the lazy phase by performing them inside suspended computations. However, this cannot always be done, as reads may be required to determine *where* suspended computations should be written. To cover this case, reads can also be performed optimistically in the preparation phase, while using the lazy phase to check whether the reads have not been modified by a concurrent transaction. In section 4 we show how the different phases can be used effectively by means of some examples.

Note that, for lazy evaluation of database operations, the end-user interface of a lazy DBMS is a programming language, and not an interactive interface as provided by current DBMSs. However, interactive transactions can still be executed using OCC, but this misses out on the concurrency benefits of laziness.

Lazy Bulk Transactions. To support non-blocking bulk updates, transactions must be able to write a large number of suspended computations without blocking the commit phase. We solve this by using laziness recursively on a tree-structured index over the data. Transactions can write suspended computations to the root of tree in constant time, which allows them to commit instantaneously. When these suspended computations are evaluated, they produce new branch nodes, which may contain more suspended computations over the next level of branches in the tree. This process continues until the leaves of the index are reached, where the data is stored. Readers can navigate the lazily constructed tree, and only evaluate those parts of the bulk operation in which they are interested.

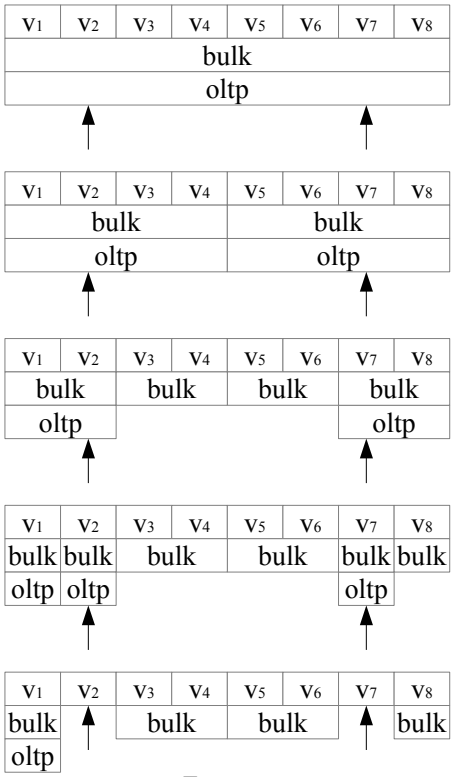


Figure 2

Figure 2 illustrates our approach on a very small database of eight values. Initially, two transactions `bulk` and `oltp` are committed by queueing their operations as suspended computations at the root of the tree. The `bulk` operation updates all values, while the `oltp` operation updates only v_1, v_2 and v_7. From top to bottom, we see how the evaluation of the lazy operations progresses, driven by demand from reads on v_2 and v_7, marked by the arrows. We can see that on every evaluation step, the lazy operations are split into operations over a smaller part of the data, which are then queued one level deeper in the tree. Operations are essentially pushed down from the root towards the leaves. In the final state, evaluation of the reads has finished, but we see that other parts of transactions are still left unevaluated. This shows that all transactions, `bulk`, `oltp` and the readers, all progressed concurrently without blocking one another, and the reads could finish without fully evaluating the transactions that preceded them.

Evaluation Strategies. Our approach provides freedom in scheduling the evaluation of suspended computations, where trade-offs can be made between throughput, latency, concurrency and memory consumption. For instance, suspended computations can be evaluated purely on-demand, or they can be evaluated immediately after committing a transaction. Importantly, a scheduler must take care that the system does not run out of memory by being too lazy. Moreover, trade-offs can be made in the granularity of laziness. We discuss evaluation strategies in section 5.

Implementation and Evaluation. We have developed an in-memory data structure that stores key-value pairs, and which provides lazy operations on single keys, key ranges, and sets of keys. Similar to B-trees [1], our data structure can be used to implement primary and secondary indexes in database systems. To implement snapshots, instances of our data structure are immutable, which is a standard technique in functional programming [15]. In section 3 we discuss the abstract interface of our data structure, and in section 4 we show how transactions can be encoded to use this interface effectively, using to the transaction phases discussed earlier. We have implemented a prototype of our data structure in Scala, where we faced some unique challenges due to lazy evaluation, which we discuss in section 6. To evaluate our approach, we have performed some experiments on our prototype implementation, which are described in section 7.

Contributions. We present the following contributions:

- A concurrency control mechanism where transactions are split into mostly independent operations on data, and where readers can evaluate operations on demand.
- An implementation of our approach in the form of a lazy index structure, which supports lazy operations on single keys, ranges of keys and sets of keys.
- A discussion on evaluation strategies for lazy transactions.
- An experimental evaluation, which shows that lazy evaluation can extract more concurrency from transactions than existing concurrency control mechanisms, and which allows large bulk operations to execute atomically without blocking a concurrent workload of OLTP transactions.

2. LAZY EVALUATION

In this section, we briefly review lazy evaluation, and introduce some notation. The basic idea of lazy evaluation is that the evaluation of an expression is delayed until its result is needed, and when an expression is first evaluated, its result is cached to avoid repeated evaluation [8]. For correctness, lazy evaluation must produce the same result as when the expression is evaluated immediately. As such, when the evaluation of an expression is delayed, its environment must be captured, so that the expression can be evaluated later in the same environment. In our setting, expressions are usually queries over the database. Thus, to ensure correctness, lazily evaluated queries must be performed on a snapshot of the database.

In our examples, we use a type `Lazy[V]` to denote a suspended computation, also called a *thunk*, that produces a value of type `V`. We construct a thunk for an expression `e` by writing `Lazy(() => e)`, and we can *force* the evaluation of a thunk `v` by writing `v.get`. For the caching aspect of laziness, the implementation of `get` must ensure that multiple or concurrent invocations of `get` must produce reference-equivalent results, so that structural sharing is preserved. Loss of sharing wastes memory, and can lead to duplication of thunks, which can in turn lead to duplication of computations.

In our implementation we evaluate thunks optimistically, which avoids the need for locks and provides good scalability under high contention. If multiple threads evaluate a thunk concurrently, they all compute their own version of the result. To ensure that concurrent invocations produce reference-equivalent results, we update a result pointer in the thunk using an atomic compare-and-set operation.

3. AN ABSTRACT LAZY INDEX

We now define an abstract interface for a lazy index, which differs from that of traditional non-lazy index structures. Implementations of this index can be used for primary and secondary indexes in lazy database systems. We assume that our index has type `Map[K,V]`, storing pairs of type `(K,V)` where `K` is the type of the keys, and `V` is the type of the values. We made the following general design decisions for the interface:

- Instances of our index are immutable, so that a snapshot is just a reference to a particular instance. Consequently, update operations produce a new version of the index instead of mutating an existing version.
- In contrast to the standard insert and remove operations on non-lazy maps, operations on our lazy map take functions of the type `() => Option[V]`, which evaluate either to `Some(v)` or `None`. In addition to allowing computing values on demand, this also allows the decision of whether to insert, update or remove values to be evaluated lazily.
- Bulk insertion of thunks should be done using a single operation, so that a lazy bulk operation can be queued in constant time, and an implementation can lazily split a bulk operation into smaller parts.

Update Operations. The following is a set of functions that we use in the examples and experiments to perform updates. All functions implicitly take a `Map[K,V]`, and return a new `Map[K,V]`. More functions can be defined in a similar fashion, and an implementation may provide specialized functions for additional performance.

`update(key: K, f: () => Option[V])`
This is a general operation that can insert, update and remove entries. If `f()` = `Some(v)`, then `(k,v)` is inserted, or `v` replaces the current value for `k`. Otherwise, if `f()` = `None`, the existing entry with key `k` is removed, or no entry is inserted.

`update_bulk(fs: Map[K,() => Option[V]])`
This operation queues multiple lazy updates in one operation by providing the updates as a map of update functions. The idea is that a transaction can build this map in the preparation phase. An implementation can lazily merge the updates with the state.

`update_range(low: K, high: K, f: K => Option[V])`
This operation updates the values of all entries where the key is between `low` and `high` inclusive. In contrast to the previous operations, the update function `f` receives the current key, so that it can look up the value currently associated with the key.

`map[W](f: K => Option[W])`
This operation is similar to `update_range`, but updates *all* values in the map, and allows changing the type of the returned map, which is necessary in schema transformations.

Read Operations. To read from the index, we use standard non-lazy functions such as `get(key: K) : Option[V]` and `aggregate(acc: T, f: (T,V) => T) : T`.

4. ENCODING LAZY TRANSACTIONS

We now show, through examples, how our lazy map can be used to execute transactions concurrently. We implement a transaction that atomically swaps two entries in a lazy map, under the condition that both entries exist. Our goal is to minimize the work in the commit phase, by deferring most work to the lazy phase and the preparation phase. While we assume that, due to laziness, each lazy write can be performed in constant time, we assume that reads and computations are generally expensive. As such, our primary goal is to avoid reads and computations in the commit phase. We first show a simple implementation that performs all operations in the commit phase, and then we show how the lazy and preparation phases can be used improve concurrency. To show how the different phases are used, we highlight them using the following color coding: preparation, commit, *lazy*, ***result***.

The State. To interact with the state, transactions must be able to get a snapshot of the current state, as well as update the state. For this, we define the following class:

```
class State[T](var current : T)
  def snapshot : T = current
  def update(transaction : T => T) =
    atomic { current = transaction(current) }
```

This class maintains a pointer `current` to the latest version of the state. A transaction can either use `snapshot` to obtain a snapshot, or use `update` to modify the state using a *transaction function* that gets the current state, and returns the updated state. In our examples, we assume that there is a global variable `state` with type `State[Map[K,V]]`.

Reading in the Commit Phase. We can now implement our example transaction by performing all operations in the commit phase as follows:

```
def swap(a,b) = state.update(s => {
  val va = s.get(a), vb = s.get(b)
  if(va != None && vb != None)
    return s.update(a, _ => vb).update(b, _ => va)
  else
    return s
})
```

We update the state by providing `update` with a transaction function that receives the current state `s`. In the transaction function, we first read the two values to be swapped from `s`. If both values are not `None`, we return a state where both values have been swapped. Otherwise, we abort by returning the original state.

While the updates can be evaluated lazily, and thus concurrently, the main limitation of this solution is that reading `a` and `b` may force the evaluation of thunks from concurrent transactions, which can be slow. In the next paragraphs we show how reads can be avoided in the commit phase by executing them either lazily or optimistically.

Lazy Reads. To perform work in the lazy phase, a transaction has to write thunks into the state that do this work. For our example, we can do this as follows:

```
def swap(a,b) = state.update(s => {
  val va = Lazy(() => s.get(a))
  val vb = Lazy(() => s.get(b))
  val c = Lazy(() => va.get != None && vb.get != None)
  return s.update(a, () => if(c.get) vb.get else va.get)
          .update(b, () => if(c.get) va.get else vb.get)
})
```

Instead of reading eagerly, we now construct two thunks `va` and `vb` that lazily read the values to be swapped. Moreover, we create a thunk `c` that encodes the commit decision, and which checks whether both values are present. We force the evaluation of `c` in the update function to decide whether to commit or abort. If we commit, the lazy reads are forced and the swapped values are written. Otherwise, we write the original values, so that the final state is equivalent to the initial state, effectively aborting the transaction. Because `a` and `b` are read inside thunks, the results of reading is shared between `c` and the update functions, which avoids duplicate reads. In this example, the thunks are constructed in the commit phase, but they can be created in the preparation phase as an optimization.

Optimistic Reads. A limitation of the lazy phase is that the keys on which to write thunks must be known. Sometimes, these keys must be determined from reads on the database. To avoid such reads in the commit phase, transactions can read optimistically using optimistic concurrency control [9]. The idea is that the transaction can read in the preparation phase, and then in the lazy phase check whether the reads have not been modified by a concurrent transaction. This can be encoded as follows:

```
def swap(a,b):
  var retry = true
  while(retry):
    val snapshot = state.snapshot
    val va = snapshot.get(a), vb = snapshot.get(b)
    if(va == None || vb == None) retry = false
    else:
      var c = null
      state.update(s => {
        c = Lazy(() => s.get(a) == va && s.get(b) == vb)
        return s.update(a, () => if(c.get) vb else va)
                .update(b, () => if(c.get) va else vb) })
      retry = !c.get
```

In the preparation phase, we obtain a snapshot of the current state, and optimistically read the values to be swapped. If the commit condition is not satisfied, the transaction is aborted without needing to enter the commit phase. Otherwise, we enter the commit phase, where we construct a thunk `c` that checks whether the values in the committed state are equivalent to the values that have been read optimistically. We lazily update the state with the swapped values, but only under the condition that the check is satisfied. If the check is not satisfied, a concurrent transaction has overwritten one of our reads, so the commit attempt has failed, and we have to write the original values to abort. After committing, the check is evaluated in the result phase to determine if the transaction has executed successfully. If

the attempt was not successful, we try again by re-executing from the beginning.

Discussion. In the above examples we have shown how reads can be performed in the commit phase, in the lazy phase, and optimistically in the preparation phase. The different phases have trade-offs, and multiple approaches may be combined to maximize concurrency. Optimistic reads can be used to check constraints before entering the commit phase, without the possibility of failure due to contention. However, when committing, it needs to be checked that optimistic reads have not been modified by other transactions, and re-execute if this is the case. In contrast, lazy reads always succeed, but they require that write locations are known in the commit phase. Moreover, optimistic reads can be used for interactive transactions, whereas the use of lazy reads requires all computations to be encoded inside thunks. If re-execution must be avoided, write locations can also be determined in the commit phase, but this blocks other transactions from committing.

5. EVALUATION STRATEGIES

Our approach provides a lot of freedom in scheduling the evaluation of transactions. A DBMS can exploit this to make trade-offs between throughput, latency, concurrency and memory consumption. In this section, we briefly discuss trade-offs that can be made in *when* to evaluate thunks, and we discuss trade-offs in the granularity of laziness.

Immediate and Delayed Evaluation. First, there is freedom in *when* to evaluate thunks. One extreme is to evaluate thunks immediately after committing a transaction, while another extreme is to delay evaluation of a thunk until its result is needed by another transaction. It is also possible to pick a strategy somewhere between these extremes, such as evaluating thunks during idle time. Delayed evaluation has been investigated by Faleiro et al. [6], and has the following trade-offs:

- **Latency**: Delaying evaluation can lead to increased read latency as more operations still need to be performed when reading.
- **Locality**: By delaying evaluation, multiple operations on the same data can be gathered and executed in batches. This can improve throughput in environments where data access has high latency, such as when data is stored on disk.
- **Blind writes**: Blind writes can make it unnecessary to evaluate expensive computations. The longer evaluation is delayed, the greater the chance that work is not needed anymore.
- **Temporal Load Balancing**: Delaying the evaluation of thunks allows spreading work over a longer period of time. Consequently, peak throughput can be higher for short periods of time compared to eager evaluation. In addition, processing power can be spent on high-priority transactions.

In practice, memory provides a hard practical constraint on the amount of laziness in a system: thunks take up memory, and they prevent snapshots from being garbage collected. To avoid these problems, thunks need to be evaluated eagerly to some degree. An important observation is that eager evaluation does not reduce concurrency compared to purely on-demand evaluation. In either case, multiple transactions can execute lazy operations concurrently, and execution of operations is still prioritized based on demand. A general conclusion from this is that immediate evaluation is sufficient for concurrency, and delayed evaluation is only an optimization for additional throughput.

Granularity. Another area where trade-offs can be made is the granularity at which laziness is used. Under fine-grained laziness, only those operations that are needed to provide the desired value are evaluated. However, it is also possible to evaluate more operations than strictly necessary. Coarse-grained evaluation has the following trade-offs:

- **Overhead**: laziness can have significant overhead for in-memory databases, thus coarse-grained laziness can boost throughput. However, for disk-based storage dominated by I/O latency, we expect that the overhead of laziness is not significant.
- **Locality**: spatial locality can be exploited by executing all pending operations on a cache line or a disk page at the same time, which improves throughput.
- **Latency**: more operations are evaluated than necessary, which increases latency.
- **Blind writes**: unnecessary work can be performed by evaluating operations that are overwritten by blind writes.

Discussion. It is important to note that evaluating a suspended computation may trigger an avalanche of reads that are scattered throughout the state, which recursively may trigger more reads. Delayed evaluation and coarse-grained evaluation make this effect stronger, so these techniques may need to be avoided if low latency is desired.

6. IMPLEMENTING A LAZY INDEX

We have build a prototype implementation[1] of our lazy index in Scala. In this section, we discuss some of the implementation details. In particular, it is challenging to keep search trees balanced under lazy evaluation, as previously investigated by Trinder [18]. As a solution, we use tries, which use a static balancing scheme. Moreover, we provide a method for compacting the trie after lazily removing values. Finally, we also discuss the role of granularity and delayed evaluation in our implementation.

Tries. Our implementation uses tries, which where first introduced by Fredkin [7]. In contrast to B-trees, which are dynamically balanced by storing keys in branches [1], a trie uses a static partitioning. Branches in a trie have an array of 2^k bins, where operations select the bin based on the binary prefix of the key. Consequently, tries do not need balancing, which makes them very suitable for concurrent data structures. For instance, tries are used to implement a concurrent map in the Scala standard library [13], and they are used to implement the transactional map in ScalaSTM [3]. Balancing is especially problematic for lazy operations, as it is unknown whether a thunk adds or removes a value until it is evaluated. The static partitioning used by tries

[1]Our implementation is available at https://github.com/utwente-fmt/lazy-transactions-IDEAS16

solves this problem. Moreover, tries have been used for main-memory database indexes [10]. Their wide branches and shallow depth makes them fast on modern architectures. Moreover, similar to B-trees, tries can be ordered, which allows fast operations on ranges of data. Finally, tries can efficiently be made persistent by path copying, i.e., copying only the nodes that are modified, while sharing unmodified nodes between the old and the new version [15].

Laziness in Tries. Adding laziness to a standard implementation of tries is straightforward: we can simply allow branches and leafs to store thunks, as follows:

```
Trie[K,V] = Empty
  | Leaf(key : K, value : Lazy[Option[V]])
  | Branch(children : Array[Lazy[Trie[K,V]]])
```

Lazy operations can be implemented similar to operations on non-lazy tries. However, instead of executing operations eagerly, thunks are written in place of branches and values. In our implementation, we only use laziness in branches, so that leaves store fully evaluated, non-optional values. This improves efficiency and memory consumption, but slightly increases the granularity of laziness: when reading only the key of a leaf, lazy operations on the value are also evaluated.

Compaction. In our implementation, lazy operations that remove entries can leave branches that only have empty subtrees. To remove unnecessary branches from the trie without blocking concurrent transactions, we rewrite a snapshot where empty subtrees are replaced with `Empty`. During compaction, transactions can keep producing new versions of the trie. As these new versions generally share many branches with the snapshot on which compaction is performed, the new versions are compacted as a side effect. Note that, to maximize sharing, instead of rewriting only the root branch of an empty subtree, all child branches should also be rewritten. This gives the following compaction algorithm:

```
def compact(node : Trie[K,V]) = node match
  case Branch(children):
    var allEmpty = true
    for(index : children):
      val compacted = compact(children[index].get)
      children[index] = compacted
      allEmpty &&= compacted == Empty
    return (if(allEmpty) Empty else node)
  case other:
    return other
```

This algorithm traverses the whole trie to find nodes to compact. As this is relatively expensive, we have implemented this as a manual operation, similar to how vacuuming is done in database systems. This can be optimized by keeping track of which nodes are already compacted, so that compaction can break of early. However, we do not do this in our prototype.

Granularity. We have implemented two variants of our trie: one where leaves store a single key-value pair, which results in fine-grained laziness, and one where leaves store 16 key-value pairs, which results in coarse-grained laziness. The broad leaves can exploit spatial locality for sequential operations, and they are more space efficient, scaling linearly at about 12 bytes per entry, compared to about 62 bytes per entry for the fine-grained leaves.

Delayed Evaluation. Our implementation evaluates suspended computations only on demand. However, our implementation is not robust under purely delayed evaluation, as the system may run out of heap space and stack space. To avoid this, transactions can force thunks immediately after committing by reading all the keys that the transaction has written.

7. EXPERIMENTAL EVALUATION

We now show the results of several micro-benchmarks on our prototype implementation. We show that our lazy map can execute bulk operations without blocking a concurrent workload of OLTP transactions, and we show that OLTP workloads can be executed concurrently, while handling larger transactions much better than existing methods. We test our prototype system in three configurations: immediate evaluation with fine-grained laziness, immediate evaluation with coarse-grained laziness, and delayed evaluation with fine-grained laziness. Our trie is configured to store 32 bins per branch, and wide leaves store up to 16 entries. The experiments are run on a quad-core Intel i7 3770 with 16GB RAM, which can run 8 threads in parallel. We use the Hotspot JVM version 1.8.0_66-b17 on Linux kernel 3.2.0.

7.1 Bulk Operations

First, we show that we can perform bulk operations without blocking OLTP workload. For the schema we use a `Map[Int,Int]`, populated with 10 million consecutive keys and random values. We test the impact on two workloads: a read-only workload where transactions atomically read 100 random keys, and a workload where transactions atomically update the values associated to 100 random keys. The bulk operation is a `map` operation which, for each mapped value, reads a number random values from the state. We vary the number of reads from 0 to 16 to show how the amount of work performed per update impacts concurrent transactions. The reads are representative for lookups in primary or secondary indexes, which are required in complex schema transformations. Performing no reads is representative of a computationally cheap data transformation on every element. In this experiment, we use fine-grained laziness, and all transactions evaluate thunks immediately after committing by reading the written keys from the state. Measurements are affected by the JVM garbage collector. To mitigate this, we run the JVM with the `-XX:+UseConcMarkSweepGC` flag to use the concurrent collector, and we use the `-Xint` flag, which runs the JVM in interpreted mode, so that the transformations run longer and the behaviour of laziness can be better observed. We use 64 worker threads for the OLTP workload, and one additional thread for the bulk update.

Figure 3 shows the results of our experiments. We see that the OLTP workload can continue executing while the bulk update is in progress. At the moment the transformation is committed, we see a sharp drop in throughput because subsequent operations must evaluate the bulk update for the data they access. Reads show a larger drop in throughput than writes. As the bulk update is evaluated over time, the throughput returns to the same level as before the bulk update. We also see that, as more reads are performed in the mapped function, the throughput reduction is larger, and it takes longer to return to the baseline performance.

Note that, because the system runs at maximum capacity, our results show a near worst-case scenario. If the sys-

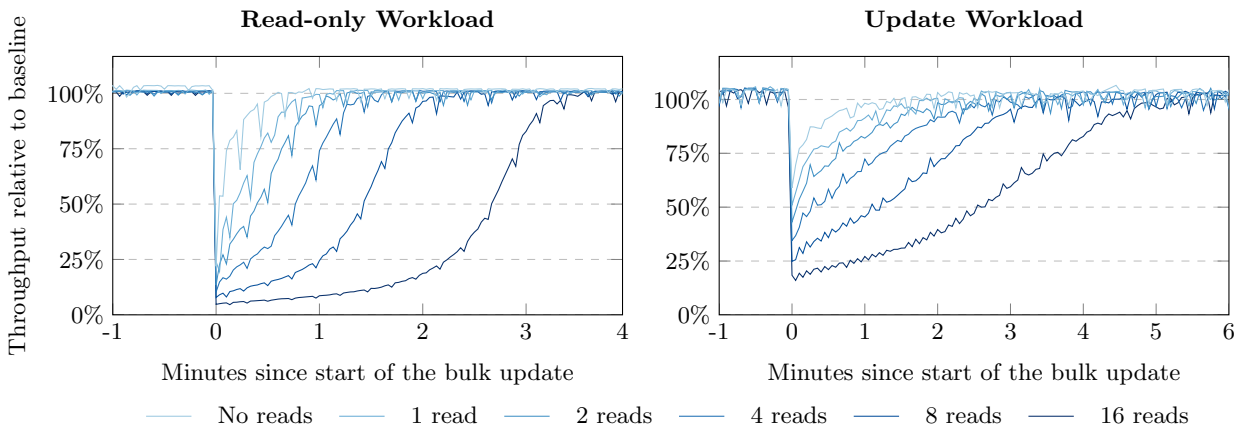


Figure 3: The impact of a `map` operation on the throughput of a concurrent OLTP workload. The number of reads in the mapped function is varied to show how the amount of work in the mapped function impacts the OLTP workload.

tem has sufficient spare capacity, laziness does not affect the throughput of the workload. Moreover, the uniform random reads strongly force the bulk update, while in practice it is more likely that reads focus on hotspots, which is more favorable to lazy evaluation.

We have also performed experiments with other evaluation strategies for bulk updates. Using delayed evaluation for the bulk operation has no measurable effect compared to immediate evaluation in our current experimental setup. When using coarse-grained laziness, the bulk update completes faster when no reads are performed in the bulk update, due to improved cache locality. However, when one or more reads are performed in the bulk update, throughput drops far below that of fine-grained laziness because each random access evaluates the bulk update on a range of values, which triggers many random reads.

7.2 OLTP Operations

We now show that our approach also provides good performance and scalability for OLTP transactions, and that it can handle some workloads with large overlaps in transaction footprints. We evaluate a number of different workloads on a `Map[Int,Int]` that is populated with one million consecutive keys associated to random values. We compare our solution to ScalaSTM [3], which is a state-of-the-art software transactional memory system for the JVM that implements optimistic concurrency control. In all experiments, we use 8 worker threads. We have run each experiment for two minutes, after a 10 second warm-up run, and we use the JVM arguments `-server -Xmx12g -Xms12g`.

Random and Sequential Access. First, we separately test random access and sequential access patterns under read-dominated and update-dominated workloads. Each worker thread randomly chooses whether to perform a read transaction or an update transaction. For random reads we randomly select keys and read the associated values from a snapshot. For sequential reads, we use an aggregate operation to compute a sum over a range of keys. For random updates, we use `update_bulk` to update multiple random keys in a single lazy operation. For sequential updates, we use the `update_range` operation to perform an operation on all values between two keys.

Figure 4 shows the sustained throughput of the various workloads for increasing transaction footprint sizes. First, in the read-dominated random-access workload we see good overall performance, which can be explained by the use of immutable data structures. From about 1024 operations per transaction, ScalaSTM shows a large drop in performance because OCC has trouble with the remaining 10% of writes, which are increasingly likely to overlap at larger transaction sizes. Laziness does not have this problem, and scales to large transactions. For random updates, we see a similar pattern, but the drop in performance for ScalaSTM is more pronounced. Lazy transactions are a bit slower than ScalaSTM for small update-only transactions. The main bottleneck is the allocation and initialization of new tree nodes, which is a drawback of immutable data structures. By using `update_bulk`, transactions share work between operations near the root of the tree, which makes larger transactions more efficient. In the bottom row, we see experiments with transactions that access the state sequentially. Similar to the random-access workload, lazy transactions scale well for larger transaction sizes. Moreover, sequential access is much more efficient than random access.

Finally, we see that coarse-grained laziness provides a substantial boost in throughput for sequential-access workloads. However, for random-access workloads the benefit is much smaller. Moreover, our experiments show that immediate evaluation is significantly faster than delayed evaluation for write-heavy workloads. Profiling shows that the reduced throughput is due to the garbage collector, and is not a problem of delayed evaluation in principle. Where immediate evaluation spends only 4.1% of the time in the garbage collector, delayed evaluation spends about 41% of the time collecting garbage due to the higher memory consumption.

Lazy Reads. The top row in Figure 5 shows a workload where transactions perform both reads and writes in a single transaction. In this experiment, write locations are known in advance, so transactions can write thunks that read lazily from a snapshot to compute the values that are written. We see similar behaviour as in the previous set of experiments. As more reads are performed per write, lazy transactions become relatively more efficient, while OCC struggles with the higher contention. OCC has keep track of all reads, and check whether they have been changed by concurrent transactions, while the lazy reads are performed uncontended on

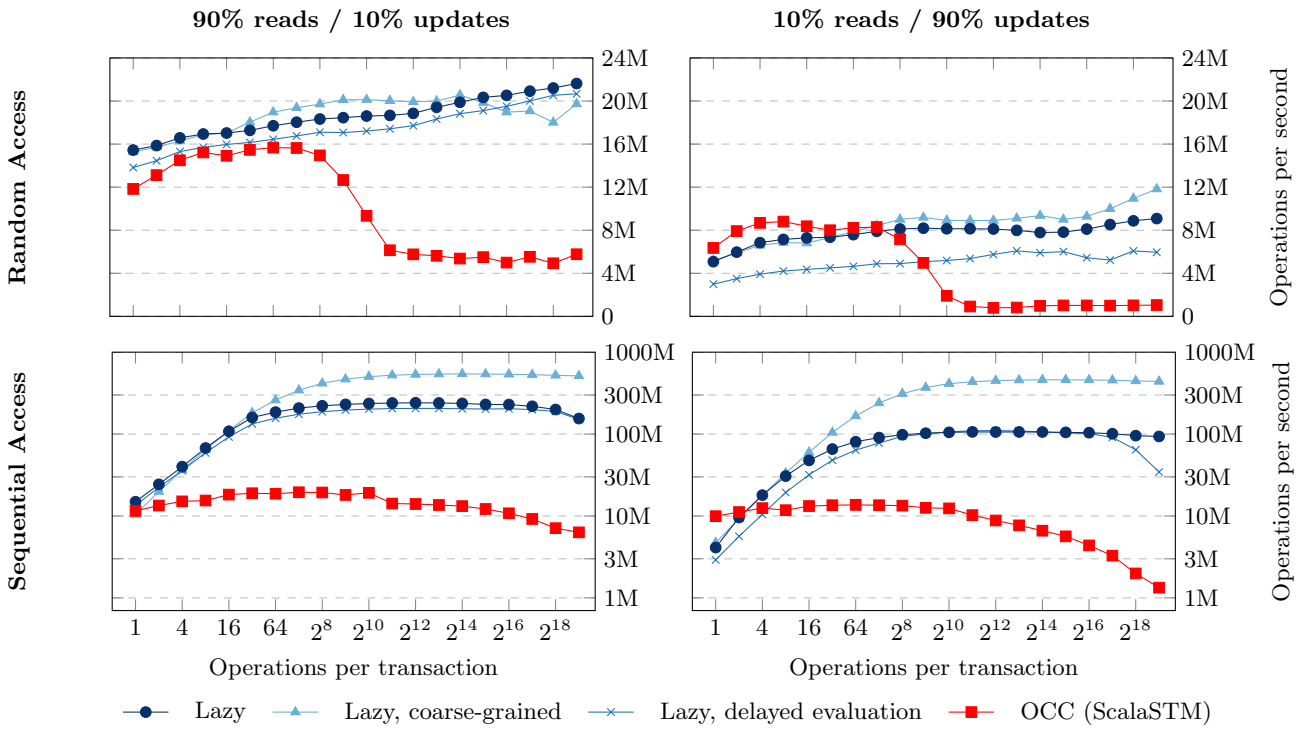


Figure 4: The effect of the size of transactions on throughput, where each transaction is read-only or update-only. On the left, transactions are read dominated, while on the right they are update dominated. Note the logarithmic axis on the sequential access plots.

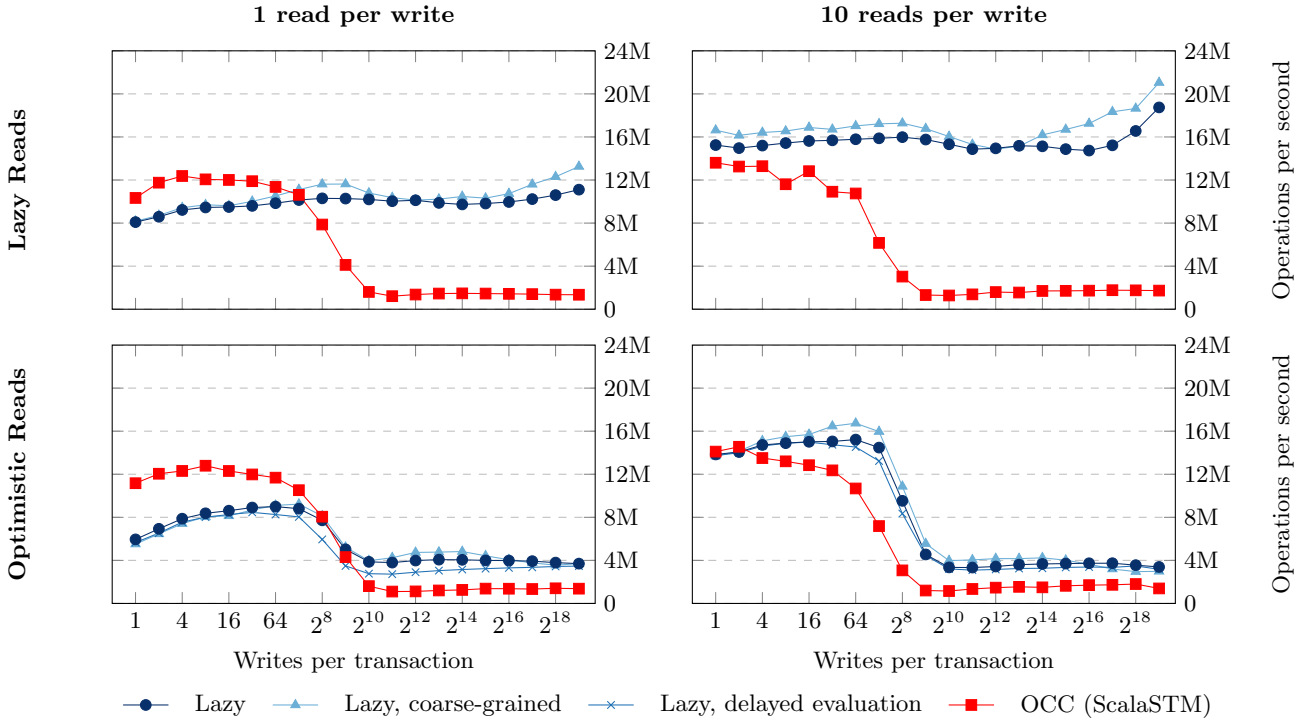


Figure 5: The effect of the size of transactions on throughput, where transactions perform a mixed read / write workload using lazy or optimistic reads.

a snapshot. Note that we could not run this experiment with delayed evaluation, because the transactions perform no eager reads that force the evaluation of thunks.

Optimistic Reads. In this experiment, we execute a workload similar to that of the lazy reads, but where transactions read optimistically. The workload consists of transactions that optimistically read a number of random keys, and write a number of random keys. The general procedure is that in the preparation phase, a transaction reads optimistically from a snapshot and constructs a tree of values to be written. In the commit phase, the tree is lazily merged with the state under the condition that the reads have not been changed by other transactions, which is checked lazily. If the check fails, the transaction is re-executed by reading in the commit phase. While this blocks other transactions while reading, it is guaranteed to succeed, and we can still extract some parallelism by writing the updates lazily.

The bottom row in Figure 5 shows that our version of OCC follows the same general pattern as ScalaSTM's OCC. When transactions perform one read per write, our implementation is less efficient than ScalaSTM for small transactions, but interestingly, it can handle large writes much better, being on average about 4 times faster than ScalaSTM. When the transactions perform 10 reads per write, our approach is faster in almost all cases.

8. RELATED WORK

The idea of using lazy evaluation in concurrency control mechanisms has been investigated in prior work, but has not yet received much attention in the literature. To our knowledge, we are the first to split transactions into independent parts, and evaluate these on demand using lazy evaluation. Moreover, this is the first concurrency control mechanism that can execute bulk operations without blocking concurrent transactions. In this section we compare our approach to other work on laziness in OLTP databases, and we look at other approaches to non-blocking bulk operations.

Laziness for OLTP. Trinder has investigated functional languages for transaction processing, where he also uses lazy evaluation as a concurrency control mechanism [18]. Our work is similar, but also different in many ways. For instance, we have developed a stand-alone lazy index structure, and we have investigated how transactions can extract concurrency from lazy evaluation. Another difference is that Trinder has investigated various ways to balance B-trees, while we avoid the need for dynamic balancing by using tries. Moreover, we have investigated alternative evaluation strategies, developed a concrete implementation, and we have performed experiments on modern hardware.

Faleiro et al. have investigated delayed evaluation of lazy OLTP transactions [6], and have shown the benefits for on-disk databases. Their approach to laziness differs from ours in many ways. Most importantly, they execute lazy transactions at the granularity of whole transactions, whereas we split transactions into smaller parts that can execute independently. Consequently, our approach extracts more concurrency from contended transactions, and we can execute bulk operations without blocking concurrent transactions. Moreover, we have shown that immediate evaluation provides many of the same concurrency benefits as delayed evaluation, while avoiding most problems with high latency and high memory consumption.

Haskell provides the `atomicModifyIORef` function, which allows a program to atomically apply an arbitrary function to an immutable data structure, swapping the pointer to that data structure with the result of this operation. While not documented in the literature, this mechanism is sometimes used as a basic STM mechanism, which can exploit laziness for concurrency. For instance, Sulzmann et al. have shown that this mechanism can be used to implement a concurrent linked list, which beats a linked list that is implemented in the Haskell STM on throughput [16]. To our knowledge, the performance characteristics of this mechanism have not been studied. Our research shows that `atomicModifyIORef` can be used as an STM, but requires that an appropriate data structure is used for laziness, and requires encoding transactions in a specific way.

Non-blocking Bulk Updates. We now compare lazy transactions to other approaches for non-blocking bulk updates and online schema changes. A general difference between these methods and our approach is that our solution is solved at the concurrency control level, and thus also allows bulk updates on a subset of the data.

Propagation-based methods copy and transform data from a source table to a target table, while propagating concurrent changes on the source table to the target table. Rönström has developed a method based on triggers [14], while Løland has developed a method based on logging [11]. The main benefit of laziness over these methods is that it allows the target table to be accessed immediately after committing. However, propagation covers cases where laziness may not work, such as online construction of indexes.

Neamtiu et al. show that laziness can be used for on-the-fly schema transformations [12]. Their approach is very different from ours, and is similar to propagation-based methods. They create a *shadow table* in the new schema, and copy data in a background thread. Queries and updates must ensure that the data has been transformed prior to accessing it. In contrast to our approach, they can only perform one schema transformation at a time, and cannot use this mechanism for OLTP transactions.

Sun et al. use laziness for online merging of B-trees [17], which is a special case of our approach. The idea is to let merge operations piggyback on user transactions to reduce I/O operations. Their approach differs from ours in that they update the tree in-place, and they use latches for synchronization on branches. In contrast, our approach uses persistent data structures, so that other lazy operations can still access the old versions, and our algorithm is lockless, which improves scalability for a high number of threads. To rebalance the B-tree during the merging process, they re-traverse the tree from the root if a split is required. A similar approach may work in our setting, but this would be complicated by the use of persistent data structures. As such, we use a trie that has static partitioning.

Another method for online transformation is rewriting of queries, which has been studied by Curino et al. [4]. They define the new schema as a view on the old schema, which effectively transforms a query on the new schema to a query on the old schema. This approaches is similar to lazy evaluation in the sense that transformations are performed on demand. The main difference is that transformed values

are not cached, and updates must either be translated and stored in the source schema, or stored in a delta table alongside the views.

9. CONCLUSIONS AND FUTURE WORK

Concurrency control is a major bottleneck in scaling transaction processing systems on parallel hardware. Existing mechanisms can only extract a limited amount of concurrency from transactions, and degrade as contention and the size of transactions increases. We have shown that lazy evaluation of transactions can extract significantly more concurrency than existing mechanisms, especially when transactions can be evaluated in a fine-grained manner. In particular, we have shown that lazy transactions can perform some types of bulk operations without blocking concurrent OLTP transactions. To the best of our knowledge, no prior concurrency control mechanism has this ability. We have also shown that optimistic concurrency control can be combined with lazy transactions, which means that any transaction that can be executed using OCC can also be executed in our mechanism with similar performance characteristics. In conclusion, lazy evaluation can extract significantly more concurrency from transactions than existing concurrency control techniques. However, many more challenges remain to be solved to come to a practical system.

Future Work. So far, we have only investigated our system in micro-benchmarks. We want to evaluate our system in a more realistic environment, for which we plan to implement the TPC-C benchmark using lazy transactions. Moreover, we want to investigate how laziness can be used on indexed data, which is important for realistic OLTP workloads. We can already perform operations involving indexes using OCC, but it may be possible to exploit laziness to further reduce contention. In addition to this, a natural extension to our work is investigating if lazy transactions can be used to perform non-blocking relational schema transformations. In particular, we want to develop lazy versions of schema transformation operators such as splitting and merging of tables, and removal of duplicate entries from tables.

What we have shown and implemented is currently still a first prototype, and there remains a lot of work to be done to come to a complete system. For instance, we would like to translate high-level user programs to lazy programs, and automatically optimize them to use laziness effectively. Another interesting direction for further research is on-disk storage, where we expect that laziness can mitigate I/O latency better than existing concurrency techniques. One idea is to restrict lazy evaluation to main memory, and store only the latest fully evaluated state on disk.

Traditional concurrency control methods have been studied for decades to optimize their performance. We expect that more optimizations and evaluation strategies can be found to improve the performance of lazy transactions. One limitation of our approach is that all transactions are applied to the root, which limits concurrency for very small transactions. This is particularly problematic on multi-processor NUMA architectures, where the communication cost between processors is high. Further research is needed to address this. Finally, our current implementation is fully persistent, while partial persistence suffices for lazy transactions. It would be interesting to see if this can be exploited to improve performance.

10. REFERENCES

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[3] N. G. Bronson, H. Chafi, and K. Olukotun. CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days*, 2010.

[4] C. A. Curino, H. J. Moon, M. Ham, and C. Zaniolo. The PRISM Workbench: database schema evolution without tears. In *ICDE'09*, pages 1523–1526. IEEE, 2009.

[5] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, Nov. 1976.

[6] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD'14*, pages 15–26. ACM, 2014.

[7] E. Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, Sept. 1960.

[8] P. Henderson and J. H. Morris Jr. A lazy evaluator. In *POPL'76*, pages 95–103. ACM, 1976.

[9] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[10] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE'13*, pages 38–49. IEEE, 2013.

[11] J. Løland and S.-O. Hvasshovd. Online, Non-blocking Relational Schema Changes. In *EDBT'06*, pages 405–422. Springer-Verlag, 2006.

[12] I. Neamtiu, J. Bardin, M. R. Uddin, D.-Y. Lin, and P. Bhattacharya. Improving Cloud Availability with On-the-fly Schema Updates. In *COMAD'13*, pages 24–34. Computer Society of India, 2013.

[13] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *ACM Sigplan Notices*, volume 47, pages 151–160. ACM, 2012.

[14] M. Ronström. On-Line Schema Update for a Telecom Database. In *ICDE'00*, pages 329–338. IEEE, 2000.

[15] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[16] M. Sulzmann, E. S. Lam, and S. Marlow. Comparing the Performance of Concurrent Linked-list Implementations in Haskell. In *DAMP'09*, pages 37–46. ACM, 2008.

[17] X. Sun, R. Wang, B. Salzberg, and C. Zou. Online B-tree merging. In *SIGMOD'05*, pages 335–346. ACM, 2005.

[18] P. Trinder. *A functional database*. PhD thesis, University of Oxford, 1989.

[19] L. Wevers, M. Hofstra, M. Tammens, M. Huisman, and M. van Keulen. Analysis of the Blocking Behaviour of Schema Transformations in Relational Database Systems. In *ADBIS'15*, pages 169–183. Springer, 2015.