A Scalable Linearizable Multi-Index Table

Gali Sheffi
Yahoo Research, Oath
Haifa, Israel
gsheffi@oath.com

Guy Golan-Gueta
VMWare research
Herzlia, Israel
ggolangueta@vmware.com

Erez Petrank
Technion
Haifa, Israel
erez@cs.technion.ac.il

***Abstract*—Concurrent data structures typically index data using a single primary key and provide fast atomic access to data associated with a given key value. However, it is often required to atomically access information via multiple primary and secondary keys, and even through additional properties that do not naturally represent keys for the given data. We present lock-free and lock-based algorithms of a table with multiple indexing, supporting linearizable inserts, deletes, and retrieve operations. We have implemented Java versions of our algorithms and evaluated their performance on a multi-core machine. The results show that the proposed table implementations are scalable and more efficient than any existing available alternative for in-memory realizations of a multi-index table.**

***Keywords*-Parallel Algorithms; Concurrent Data Structures; Progress Guarantees; Lock-Freedom;**

I. Introduction

The rapid deployment of highly concurrent machines has resulted in the acute need for concurrent algorithms and their supporting concurrent data structures. Concurrent data structures are designed to utilize all available cores in order to achieve faster performance. The literature contains plenty of designs for concurrent data structures that implement sets (or maps). Typically, these data structures consider a key and data associated with it as an item. Such an item can be inserted into the data structure, deleted from it, and the item can be quickly retrieved using its key. However, in practice, data sometimes contains various different keys [1], [2]. For example, a student may have an identity number (such as an S.S.N.), a driving license number, a name, a phone, an address, etc., and it is often useful to be able to access the student's record using any of his properties. If a student record is kept in a data structure using the identity number as the key, then it would be costly (if at all possible) to locate a student record according to his phone number. There are two common solutions for this problem (i.e, solutions that provide concurrent multi-indexing): (1) STM - software transactional memory (usually not used in practice), and (2) ad hoc combinations of several concurrent maps (in many cases, such combinations do not provide the wanted result. see [3]).

We present constructions of a concurrent table. The table has entries that can be quickly accessed via their primary key but also via secondary keys and properties that would not normally be considered as keys. For example, we may ask if there is a student in the class who lives on 5th Ave. Our constructions are linearizable, meaning that an item is inserted to (or removed from) the table simultaneously through all its indices. We propose lock-free and lock-based constructions and compare them against each other, against a simple construction that employs coarse-grained locking and against an STM table.

In a sequential setting, the solution to this problem is simple: employ several data structures, one to index each relevant property. This way, an item can be quickly found using any of the indexed fields. While such a solution is easy to implement in a sequential environment, it is not simple to make it correct, i.e., linearizable [4], when operations are executed on the table concurrently. Linearizability means, for example, that the existence of an item in the table does not depend on which index is used to search for it. The operation of inserting (or removing) elements simultaneously and instantly into (or from) several concurrent data structures brings up interesting algorithmic challenges.

In the context of a table, it is useful to deal with standard unique keys but also with non-unique properties of the item. For example, if we try to insert an item to the table with an identity number that already exists in the table, then the insertion should fail. The item already exists in the table. But inserting an item with an address that already exists in the table is fine. There could be two items in the table that represent two students who live in the same building. Interestingly, we found out that even the *contains* operation is not trivial to implement for a property that is non-unique.

On the practical level, while using several indices to organize data is often needed, existing solutions are either very heavy (in the sense that they provide much additional functionality at a cost) or very slow (because they do not support secondary keys). Databases often allow multiple indexing into the data, but their optimizations are mainly focused on disk operations. In particular, they are not designed to provide efficient in-memory concurrency for multi-core

Most of this work was done when the first author was a masters student at the Technion. This work was supported by the Israel Science Foundation grant No. 274/14.

machines (as in concurrent data structures). The concurrent table proposed in this paper is a simple concurrent structure that supports multiple indices, but is meant to hold a fast and scalable in-memory table, which quickly locates an item given any of its keys or properties.

A different practical alternative is to use a concurrent table that is indexed only by a single primary key. Many concurrent data structures that implement a key-value store can be used to implement such a table (e.g. [5]). However, finding an item given a secondary key requires traversing the table. This is not only inefficient, but it is not simple to implement a linearizable *contains* operation without holding a coarse-grained lock on the table during the entire traversal. One alternative to using a coarse-grained lock is to employ a snapshot mechanism, (e.g., [6], [7]). But that would still require a costly traversal of the table.

II. Preliminaries

Our model for concurrent multi-threaded computation follows the linearizability model of [4]. In particular, we assume an asynchronous shared memory system where a finite set of deterministic threads communicate by executing atomic operations on some finite number of shared variables.

A. Table Terminology

A *table* is a set of n-tuples $\langle e_1, \ldots, e_n \rangle$ where all tuples have the same number of elements. A table T has $n = N_T$ fields (given T, N_T is a constant), denoted by $f_1, \ldots, f_{N_T}$, where each field represents a different element of the tuples. Each field f_i can be either *unique* or *non-unique*. If f_i is unique, then any two different tuples $\langle e_1, \ldots, e_n \rangle$ and $\langle e'_1, \ldots, e'_n \rangle$ in T satisfy $e_i \neq e'_i$. Otherwise (i.e., f_i is non-unique), T may contain two different tuples $\langle e_1, \ldots, e_n \rangle$ and $\langle e'_1, \ldots, e'_n \rangle$ such that $e_i = e'_i$.

Each table field may be of a specific type (such as Integer, Float or String). We assume that there exists a total order $<$ of all possible elements such that for any two elements $e_i \neq e'_i$ either $e_i < e'_i$ or $e'_i < e_i$. We also assume that there exist two special elements $-\infty$ and ∞. The elements $-\infty$ and ∞ represent the minimal and the maximal possible element that can be stored in a tuple.

B. Table Operations

The proposed table supports the *add*, *remove* and *retrieve* operations. The *add* operation receives a tuple $\langle e_1, \ldots, e_n \rangle$ and adds it to the table if and only if for every unique field number i the table does not contain a tuple $\langle e'_1, \ldots, e'_n \rangle$ such that $e_i = e'_i$. The *remove* operation receives a value val and a unique field number i, and tries to remove a tuple $\langle e_1, \ldots, e_n \rangle$ such that $e_i = val$ from the table. It will fail is no such tuple exists in the table. The *retrieve* operation receives a value val and a field number i and returns all tuples $\langle e_1, \ldots e_n \rangle$ that satisfy $e_i = val$. A simpler *contains* operation can be implemented by calling the *retrieve* operation, and returning *true* if *retrieve* returns at least one tuple, and *false* otherwise.

III. The Algorithm

We chose to implement multiple indexing of a concurrent table by maintaining multiple data structures, each indexing a different key (or property) of an item. In order to simplify the design, we chose simple concurrent (ordered) linked-lists to maintain the indexed set (or multi-set). Next, in order to speed up the access time, we built skip-lists on top of the linked-lists. The linked-lists are carefully designed to maintain correctness and linearizability and make sure that items appear to be inserted to or removed from all indices at once. Being a member of the table means being a member of all linked-lists. Thus, an item can either be in all linked-lists, or in none. The skip-lists are implemented in a more relaxed manner and are only used to provide fast indexing. They do not need to always properly hold an updated view of all items in the table.

In this section we describe the algorithm. We start by defining some variants of the CAS operation, which are widely used in our algorithms (in Section III-A). We then present the data structure we use (in Section III-B), and continue with some algorithmic challenges (in Section III-C), a detailed description of the lock-free table algorithm (in Section III-D), a short description of the lock-based variant (in Section III-E) and a summary of the lock-free variant's correctness proof (in Section III-F). Finally, we explain the additional fast skip-list indexing (in Section III-G). For lack of space, we put the formal proof of the lock-free variant, and the full description and formal proof of the lock-based variant in [8].

A. Extending the CAS Operation

The *compareAndSwap* (CAS) operation is a standard synchronization instruction widely available by hardware. It receives three inputs: $addr$, exp and new, and it executes the following atomically. If the memory content at location $addr$ equals the input value exp, then it assigns new as the new content of memory location $addr$. Otherwise, the memory is not changed. The CAS operation returns true if the content was indeed replaced and false otherwise. The literature contains many CAS extensions (see [9]), enabling the comparison and swap of two or more memory locations.

In our lock-free implementation, we often use the CAS operation as described above (for example, see line 31 or 46 in Figure 3) but also in an extended manner in which a pointer has an additional associated flag by which it can be atomically marked. Marked atomic pointers were first used in [10] and later in many other papers (see [5], [11]). The extended CAS atomically modifies a reference and an associated boolean flag. It receives five input parameters, $addr$, $expPtr$, $newPtr$, $expFlag$ and $newFlag$. It compares the contents of the pointer and boolean flag of the reference object in $addr$ to $expPtr$ and $expFlag$ (respectively), and if both are equal, it modifies them to $newPtr$ and $newFlag$. These two comparisons and two assignments are executed atomically, returning true when successful and false otherwise. For example, this appears in line 48 or 49 in Figure 3. This extended CAS is supported by existing programming environments such as the Java runtime library.

Practically, this can be done in C by using the least-significant-bit of the pointer (which is always zeroed in practice). Thus, one needs to modify a single word, which can be reduced to the simple CAS. Or, in Java, this can be done by allocating a new object that holds the two fields, and modifying the pointer to it in a single atomic swap in order to execute a modification of the two fields simultaneously.

B. Internal Data Structure

Our table organizes the stored tuples as a collection of sorted linked-lists. In each list L_i the tuples are sorted according to the i-th field.

We call the object that represents a tuple *a record*. Each record consists of (1) an array $data$ with N_T entries, (2) an array $next$ with N_T entries, and (3) a $status$ flag. The $data$ array stores the represented tuple. Specifically, if record R represents the tuple $\langle e_1, \ldots, e_n \rangle$ then $R.data[i] = e_i$ for any $1 \leq i \leq n$. The $next$ array contains pointers that maintain the internal lists of the table. If $L_i = R_1 \rightarrow R_2 \rightarrow \ldots \rightarrow R_{k+1}$ then $R_j.next[i]$ points to R_{j+1} for every $1 \leq j \leq k$. In addition to the pointer, each $next[i]$ field contains a $marked$ boolean flag. The pointer and boolean flag can be read and updated simultaneously and atomically, using a single CAS execution (for more details, see Section III-A).

In order to implement a simultaneous insertion (or deletion) of a record into all the linked-lists at once, we use the *status* field (a more general *status* field is being used in [12]). The status of a record is set to *Pending* while it is being inserted into all the lists. While the status is *Pending*, the algorithm does not consider this record as part of the table, even if it has already been inserted into some of the lists. Upon completion of insertions to all linked-lists, the record's status is modified to *InTable*, and it is exactly at this point that the record becomes a member of the table (this is considered as the logical insertion of the record into the table). To remove a record from all lists simultaneously, it is enough to mark the status field as *Removed* and from that point and on the algorithm considers it deleted. Deletion proceeds by unlinking the record from all the linked-lists.

If the table has a unique field, then a thread may fail to insert a record because its relevant property already appears in the list of that unique field. This happens also in the standard case of a single-key list, and the failed attempt is not even visible to the other threads. But in the case of a table we need to handle the case of a successful insertion into one list and then a failed insertion to a second list. For example, a student may have changed a driving license (e.g., due to moving to a different state), but still have the same identity number. In this case, it may happen that we attempt to insert a record for this student while an old record representing him is present in the table. Our algorithm will succeed to insert the record using the first index (the driving license number) because the old record of this student has a different driving license number. But when it attempts to insert this record into the second index (the identity number), it will discover that the student already appears in the table and then the insertion would return failure. When this happens, we mark the status of the inserted record as *Failed* and execute a protocol that eventually removes this record from the list.

We use two special records, *head* and *tail*, which represent the tuples $\langle -\infty, \ldots, -\infty \rangle$ and $\langle \infty, \ldots, \infty \rangle$, respectively. Each list L_i starts with the *head* record and ends with the *tail* record. The records *head* and *tail* are not considered table items, and therefore they are never added to or removed from the table. Both *head* and *tail*'s statuses are always set to *InTable*.

In Figure 1 we present an example of a table with three fields ($N_T = 3$). The arrows represent the next[] references, and in the i-th list, records are sorted according to their i-th field. Notice that the table does not contain two tuples with the same unique field, and that if a record's status is not *InTable*, it is not necessarily fully linked.

C. Algorithmic Challenges

We propose a lock-free and fine-grained locked-based implementation of the concurrent table using the ideas raised in Section III-B. Let us highlight some algorithmic challenges that must be handled when designing the lock-free table variant (for the lock-based variant, see Section III-E).

Interestingly, for the lock-free algorithm, the *retrieve* (or even a *contains*) operation is not trivial when one searches for a property that is non-unique. Note that it suffices to retrieve one record with the required property, and this is still non-trivial. Of course, if we find a record with the required property in the table, then we simply retrieve this record. The problem is that if we do not find a record with this property, we cannot determine in a linearizable manner that it is not in the table. During a search, it is possible that a record with a given property is continuously present in the table, but the *retrieve* method does not find any record that matches this property. This can happen, for example, in the following scenario. When the search finds the first record with the given property, that record has a pending status. It has not yet been added to the table, but it is in the process. Next, the search goes to the next record and discovers that its status is removed. If there are no more records with this property, the search may deduce erroneously that this property is not in the table. However, it is possible that before the second

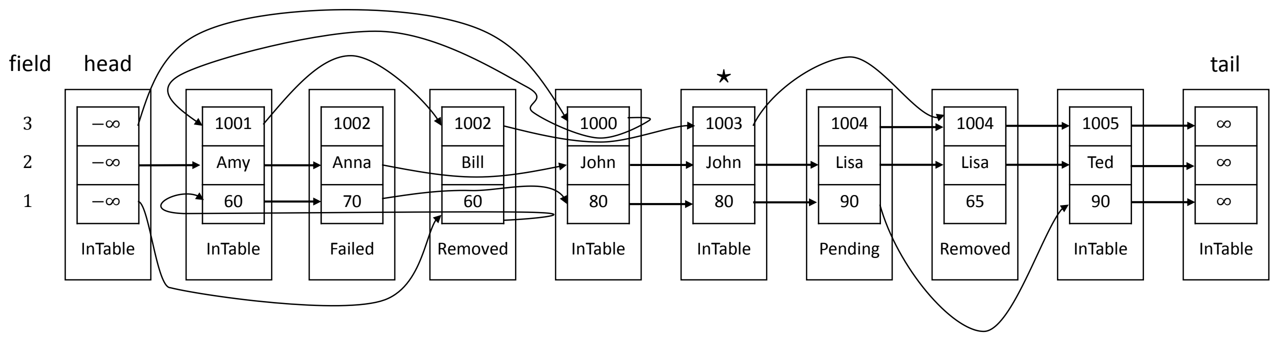

Figure 1. A table with three fields ($N_T = 3$). The first and second fields are non-unique and the third field is unique. Besides the *head* and *tail* sentinels, there are four records with an *InTable* status (i.e. the table contains four tuples)

record was removed, the first record has been completely added to the table. And so a record with this property has been part of the table for the entire execution of the search.

The problem is that the traversal of records with the required property is not atomic. This problem resembles the challenge of obtaining a snapshot of a lock-free data structure (which is challenging) and there exist in the literature various ways to deal with it in a lock-free or even wait-free manner (e.g., [7]). We chose a simple and efficient lock-free solution for this problem. First, we maintain an invariant that a new record with a given property is always added before all existing records with that property. Given this invariant, it is possible to traverse the sublist (of records with the given property) repeatedly until we find two equal views of it. And if these two views do not contain a record that is properly in the table, then we can determine that there is a point in the execution in which no record with this property was present. Whenever we do not obtain the same view, another operation has made progress and so lock-freedom is maintained.

Moving to the proposed fine-grained *lock-based* design, we would like to lock a record (or two) in each list in order to execute an insertion (or a deletion) of a record. A natural locking strategy would be to keep one lock for each record and lock the needed records in a disciplined order one by one starting from the one needed for the first linked-list until the one needed for the last linked-list. Interestingly, this creates a potential deadlock.

To see that, one needs to notice that the order of records in the different linked-lists may differ. So a record A can precede record B in the first list, but record B can precede A in the second list. Therefore, when two insertions are executed concurrently, we can see the following problem. The first thread acquires a lock to record A in order to execute the insertion to the first linked-list, and then the first thread waits to acquire the lock of record B in order to execute the insert to the second linked list. At the same time, a second thread locks B in order to insert a different record to the first linked-list and then waits for the lock of record A in order to insert a record to the second linked list, and we get a deadlock.

To avoid such deadlock, we allocate a lock per record per property. Now, we obtain an order on all locks that we can use to maintain discipline in the order of acquiring locks. In this order, locks of a preceding linked-list precede all locks of a subsequent linked-list and locks in between a linked-list are ordered according to the order of their records in that list. Using this ordering, a thread that waits for a lock in the second linked-list is independent of other threads acquiring locks in the first linked-list or in any other linked-list, and dead-lock is prevented.

D. The Lock-Free Table

In this section we describe the lock-free table design. While all operations present interesting algorithmic challenges, one unexpected challenge was to obtain linearizability for *retrieve* (at the end of this section). We say that a record is logically in the table if it is physically in all internal linked-lists and its status is *InTable* (for the formal definition, see Definition 1 in Section III-F). This means that the successful insertion of a new record into the table includes adding it to all lists and then changing its status from *Pending* to *InTable*. The removal of a record includes changing its status (does not necessarily include its physical removal from the lists) from *InTable* to *Removed*.

The find method

Before describing the *add*, *remove* and *retrieve* operations, we are going to describe the *find* auxiliary method to be used by *add* and *remove* (shown in Figure 2). The *find* method receives a value val and a field number i and returns two records, $pred$ and $curr$, such that (1) $pred.data[i] < val$, (2) $curr.data[i] \geq val$ and (3) $curr$ is $pred$'s successor in L_i.

The method starts its traversal from *head*. The $pred$ and $curr$ references are advanced throughout L_i, until the relevant two records are found and returned (line 14).

In addition to finding the described records, the *find* method also helps physically removing records that have

```
1:  find(val,i)
2:      retry:                                        ▷ Starting a new search
3:      pred ← head
4:      curr ← pred.next[i]
5:      while (true) do                   ▷ While desired records not yet found
6:          succ, marked ← curr.next[i]
7:          while (marked) do                ▷ While curr is logically removed
8:              snip ← CAS(pred.next[i], curr, succ, false, false)
9:              if (!snip)                      ▷ If the physical removal failed
10:                 goto retry
11:             curr ← pred.next[i]
12:             succ, marked ← curr.next[i]
13:         if (curr.data[i] $\geq$ val)
14:             return pred, curr                ▷ Return the current records
15:         pred ← curr
16:         curr ← succ
```

Figure 2. The lock-free *find* method

been logically removed (see [10]). Records are removed from the table during unsuccessful *add* and successful *remove* executions.

A record R is physically removed from L_i only after its $next[i]$ pointer is marked. The *find* method is the only one in charge of physical removals, and it executes them in the following way: Each time $curr$ is advanced, the method checks if $curr$'s $next[i]$ pointer is marked. If $curr$'s $next[i]$ pointer is not marked, the traversal continues. Otherwise, the method attempts to physically remove $curr$ from L_i using a CAS operation (line 8). It does it by setting $pred$'s successor in L_i to $succ$. If the execution is successful, then $pred$'s successor in L_i becomes $succ$, meaning $curr$ is no longer in L_i. Otherwise, The traversal starts again from *head*.

The add operation

The *add* operation (shown in Figure 3) receives a tuple tup, creates a new corresponding record R with the data of tup and a *Pending* status and tries to add R to the table. In order to enable lock-freedom, the adding procedure is divided into smaller steps, allowing other executing threads to help the insertion (if it prevents them from finishing their own insertions). After creating R, it is added to each list (according to their order) using the *helpAdding* method. When returning from the *helpAdding* method, if the insertion was successful, R's status is either *InTable* or *Removed* (The later is possible if another thread has removed it after its insertion and before this check of its status). In this case, the whole operation is considered successful and it returns $true$ (line 27). If the insertion was unsuccessful, R's status is set to *Failed*, and all of R's $next$ pointers are marked (lines 21-25), for a physical removal during a later *find* execution.

The *helpAdding* method (line 28) receives a record R and a field number $1 \leq m \leq N_T$, and it tries to insert R into $L_m, L_{m+1} \ldots, L_{N_T}$, according to their order. This allows helping to insert a record given that it already has been inserted into the first $m-1$ lists. After finishing the physical adding procedure (the loop in lines 29-30), the executing

```
17: add(tup)
18:     newRecord ← makerecord(tup)          ▷ Creating the new record
19:     helpAdding(newRecord,1)             ▷ 1 stands for the first list
20:     if (newRecord.status == Failed)    ▷ Insertion was unsuccessful
21:         for (i=1 to $N_T$) do
22:             succ, marked ← newRecord.next[i]
23:             while (!marked) do
24:                 CAS(newRecord.next[i], succ, succ, false, true)
25:                 succ, marked ← newRecord.next[i]
26:         return false
27:     else  return true                    ▷ Insertion was successful

28: helpAdding(newRecord, m)
29:     for (i=m to $N_T$) do
30:         helpAddingField(newRecord, i)
31:     CAS(newRecord.status, Pending, InTable)      ▷ Logical insertion

32: helpAddingField(newRecord, i)
33:     while (newRecord.status == Pending) do
34:         succ ← newRecord.next[i]                     ▷ For a later CAS
35:         pred, curr ← find(newRecord.data[i], i)
36:         tmp ← curr
37:         while (tmp.data[i] == newRecord.data[i]) do
38:             if (tmp == newRecord)
39:                 return               ▷ The record is already in the list
40:             tmp ← tmp.next[i]
41:         if (Unique(i))
42:             if (curr.data[i] == newRecord.data[i])
43:                 if (curr.status == Pending)
44:                     helpAdding(curr, i+1)           ▷ Help adding curr
45:                 if (curr.status == InTable)          ▷ Insertion failed
46:                     CAS(newRecord.status, Pending, Failed)
47:                     break
48:         if (CAS(newRecord.next[i], succ, curr, false, false))
49:             if (CAS(pred.next[i], curr, newRecord, false, false))
50:                 return              ▷ The record was added to the list
```

Figure 3. The lock-free *add* operation

thread tries to change R's status from *Pending* to *InTable* using a CAS operation (line 31).

The *helpAddingField* method is in charge of the insertion of the new record R into a list L_i, where i and R are provided as parameters. The method returns immediately when finding out that R's status is not *Pending* anymore, and keeps trying to add it otherwise.

Every insertion trial starts by saving R's current successor in L_i in the local variable $succ$ for a later use (line 34). Next, The *find* method is called in order to locate R in L_i (line 35). Our next concern is making sure that R has not been already inserted into L_i, before trying to physically insert it. In order to make sure that R has not been already inserted into L_i, we go over all records R' satisfying $R'.data[i] = R.data[i]$ that follow $curr$ in L_i (lines 36-40), and stop the current insertion procedure if one of them is R (line 39).

Now, if i is a unique field number, adding R to L_i may cause a violation of the uniqueness property. Therefore, before we try to add R to L_i, we have to make sure that there does not exist any other record with R's i-th property, which is logically in the table. If $curr$'s i-th property is not equal to R's i-th property (checked in line 42), then we can continue in the adding procedure. Otherwise, we need to make sure that either $curr$ is not logically in the table (and we can continue in our adding procedure) or is logically in the table (and the adding procedure must fail).

If $curr$'s status is *Pending*, then $curr$ has not been fully added to all lists, and we are not yet able to determine whether its insertion procedure is going to succeed or fail. In this case, we help adding $curr$ to $L_{i+1}, \ldots, L_{N_T}$ (line 44), since it is guaranteed that it has already been inserted into $L_1, \ldots, L_i$. Now, if $curr$'s insertion has been successful, and its status has been set to *InTable*, R's insertion would violate the table's uniqueness property. Therefore, in this case, we set R's status to *Failed*, which notifies all helping threads that its insertion is no longer relevant (line 46). Otherwise, if either $curr$'s insertion has not been successful (its status is *Failed*) or it had been successful but $curr$ has already been logically removed thereafter (its status is *Removed*), we can proceed with the adding procedure.

At this stage, regardless of i being a unique or a non-unique field number, we try to insert R into L_i. We do it by making $curr$ be its successor (line 48) and make $pred$ be its predecessor (line 49) in L_i. If both CAS executions are successful then R is successfully inserted into L_i. Otherwise, we start a new insertion trial.

```
51: **remove(val, i)**
52:     pred, victim ← **find**(val, i)                ▷ We do not need pred
53:     **if** (victim.data[i] $\neq$ val)
54:         return false                             ▷ There is no such item
55:     **else if** (!CAS(victim.status, InTable, Removed))
56:         return false
57:     **for** (j=1 to $N_T$) **do**
58:         succ, marked ← victim.next[j]
59:         **while** (!marked) **do**
60:             CAS(victim.next[j], succ, succ, false, true)
61:             succ, marked ← victim.next[j]
62:     return true                    ▷ The record was logically removed
```

Figure 4. The lock-free *remove* operation

The remove operation

The *remove* operation (shown in Figure 4) receives a value val and a unique field number i, and tries to logically remove a record R such that $R.data[i] = val$. In line 52, the *remove* operation calls the *find* method in order to determine whether there exists a record R which is in the table and for which $R.data[i] = val$. The potential victim for removal is assigned into the $victim$ local variable. If $victim$'s i-th property is not val, then it is guaranteed that there is no other relevant record, and the operation fails (line 54). Otherwise, if the attempt to logically remove $victim$ from the table (by setting $victim$'s status to *Removed*) fails, the operation is considered to be a failure, and it returns in line 56.

If the trial to logically remove $victim$ from the table is successful, then the operation is considered successful. Before returning $true$ in line 62, we mark all of the victim's $next$ pointers, for its future physical removals by *find* executions.

```
63: **retrieve(val, i)**
64:     **while** (true) **do**
65:         tmpSet ← $\emptyset$        ▷ For saving pointers to the relevant records
66:         curr ← head                                      ▷ First traversal
67:         **while** (curr.data[i] $<$ val) **do**
68:             curr ← curr.next[i]
69:         **if** (curr.data[i] $>$ val)
70:             return tmpSet                  ▷ There are no relevant records
71:         first ← curr                ▷ Saving the first relevant record
72:         **while** (curr.data[i] == val) **do**
73:             status ← curr.status
74:             **if** ((status == InTable) **or** (status == Pending))
75:                 tmpSet ← tmpSet $\cup$ $\{\langle$curr, status$\rangle\}$
76:             curr ← curr.next[i]
77:         curr ← head                                     ▷ Second traversal
78:         **while** (curr.data[i] $<$ val) **do**
79:             curr ← curr.next[i]
80:         **if** (first == curr)                ▷ No new records were added
81:             tuples ← $\emptyset$                      ▷ For returning the tuples
82:             valid ← true
83:             **for** ($\langle$R, status$\rangle$ in tmpSet) **do**
84:                 **if** (R.status == status)
85:                     **if** (status == InTable)
86:                         tuples ← tuples $\cup$ {R.data}
87:                 **else**
88:                     valid ← false
89:                     break
90:             **if** (valid)
91:                 return tuples
```

Figure 5. The lock-free *retrieve* operation

The retrieve operation

The *retrieve* operation (shown in Figure 5) receives a value val and a field number i, and returns the set of all tuples $\langle e_1, \ldots e_n \rangle$ satisfying $e_i = val$ (meaning all arrays $R.data$ for which R is logically in the table and $R.data[i] = val$).

The operation starts by traversing L_i to find the first record for which $data[i] \geq val$. If there does not exist a record with val as its i-th property, meaning the first record found satisfies $curr.data[i] > val$, then the set returned is empty (line 70). If a relevant record is found during this traversal, a pointer to the first relevant record is saved in the local variable $first$ (line 71). After saving it, pointers to all of the following relevant records are also saved, together with their statuses (lines 72-76), in the $tmpSet$ set. Relevant records are records R such that $R.data[i] = val$ and whose status is either *Pending* or *InTable*.

After all the relevant seen records are saved, the operation starts a second traversal (line 77) and checks whether the first record R satisfying $R.data[i] = val$ is the same one as in the first traversal (line 80), saved in the $first$ local variable. If it is not, then the whole procedure starts again (from line 64). Otherwise, due to the fact that a new record with a given property is always added before all existing records with that property (as mentioned in Section III-C), it is guaranteed that the snapshot saved in $tmpSet$ is a correct snapshot of all records in the table that have val in their i-th field (for more details, see Section III-F), and this snapshot can be linearized when the second traversal starts.

After making sure that there are no new relevant records in L_i, the operation goes over the saved pairs of records and statuses $\langle R, status \rangle$, and checks whether $R.status$ is still $status$ (lines 83-89). If all pairs satisfy this condition, all the relevant tuples (represented by saved records with an *InTable* status) are added to the $tuples$ set (line 86), and returned as output (line 91).

E. The Lock-Based Table

Moving to the proposed fine-grained *lock-based* design, we use the same overall strategy of maintaining a collection of linked-lists to provide the multiple indexing, but we use lists with fine-grained locking. We relegate the detailed description and proof of the lock-based table variant to [8].

F. Correctness

The full correctness proof of both table variants appears in [8]. It includes a linearizability proof for both variants, a proof that the lock-free table variant (presented in Section III-D) is indeed lock-free, and a proof that the lock-based variant (presented in detail in [8]) is dead-lock free. Here, we are going to provide a short summary of the lock-free variant's correctness proof.

Basic definitions

Let T be a table with N_T fields, implemented using the lock-free implementation described in Section III-D. Before showing that our implementation is linearizable and lock-free, we need to show that throughout any parallel execution of T's operations, T behaves according to its sequential specification (as described in Section II-B). Meaning, for every completed table operation, it returns the result it would return if the operations were executed one by one, in the order of their linearization points.

Given a record R, we would like to be able to define the exact terms that determine whether R is considered as a member of T. In Section III-D, we referred to it as being logically in T. We start by presenting Definition 1, which is a formal definition of being logically in the table.

Given two records, R_1 and R_2, we say that R_1 is an i-predecessor of R_2 (or that R_2 is the i-successor of R_1) if $R_1.next[i]$ points to R_2. In addition, we say that R_1 i-precedes R_2 (or that R_2 i-follows R_1), if there exist records $R_{i_0}, \ldots, R_{i_k}$ such that $R_{i_0} = R_1$, $R_{i_k} = R_2$ and for every $0 < j \leq k$, R_{i_j} is the i-successor of $R_{i_{j-1}}$. Using the concept of i-following, we can now define the i-reachability term. We say that a record R is i-reachable if R i-follows *head*. Now, we can define formally what being logically in T means:

Definition 1: Given a record R, we say that R is logically in T if its status is *InTable* and for every $1 \leq i \leq N_T$, R is i-reachable.

Our basic assumption is that no thread ever tries to add or remove a tuple with either an ∞ or $-\infty$ field (notice that we do not have to rely on this assumption, since it can be checked during the execution of the *add* and *remove* operations, respectively). Meaning, no thread ever tries to add or remove *head* or *tail* from T. Given this assumption, it suffices to show that the following invariants always hold:

1) For every $1 \leq i \leq N_T$, *head* and *tail* are i-reachable.
2) A record R is logically in T if its status is *InTable*.
3) For every $1 \leq i \leq N_T$ and two records R_1 and R_2, if R_2 i-follows R_1 then $R_1.data[i] \leq R_2.data[i]$.
4) For every unique field number $1 \leq i \leq N_T$ and two records R_1 and R_2, if R_2 i-follows R_1 and both are logically in T, then $R_1.data[i] < R_2.data[i]$.

We prove that the above invariants always hold in [8]. Our proof relies on some basic observations (which are also proven in [8]): (1) A record's $next[i]$ pointer is marked only if the record's status is either *Removed* or *Failed*, (2) A marked pointer cannot become unmarked, (3) A record can only be physically removed from a certain list L_i when executing line 8 of the *find* method, and only if its $next[i]$ pointer is marked, and (4) A record's status can only change from being *Pending* to being *InTable* or *Failed* and from being *InTable* to being *Removed*.

Notice that invariant 2 guarantees that a record's status can be *InTable* only if it is i-reachable for every $1 \leq i \leq N_T$. This means that (as stated in Section III-D), changing a record's status from *Pending* to *InTable* is indeed its logical insertion into T, and changing a record's status from *InTable* to *Removed* is indeed its logical removal from T. This means that records which are not logically in T are not treated as T's members, even if they are i-reachable for some $1 \leq i \leq N_T$.

Linearizability

The table implementation described in Section III-D is linearizable. Therefore, for every parallel execution (we give a detailed definition of an execution in [8]) of the table's operations, one can assign a linearization point to each completed operation and some of the uncompleted operations so that the linearization point of each operation occurs after the operation starts and before it ends, and the results of these operations are the same as if they had been performed sequentially, in the order of their linearization points (see [4]).

We shall define linearization points for each of the completed *add*, *remove* and *retrieve* operations, and some of the uncompleted ones. In addition, it will be convenient to define linearization points for calls to the *find* method, terminated during that execution. The first goal is to show that at any time, the set of tuples represented by records which are currently logically in the table is exactly the set of tuples for which there exists an already linearized *add* call inserting them into the table, and there does not exist an already linearized *remove* call removing them from the table. The second goal is to show that the output of each operation matches the table's content. For instance, that a *remove* call with an input (val, i) will return false if and only if at its linearization point, there does not exist a record R which is logically in T, and for which $R.data[i] = val$.

The find *method:* First, we define linearization points for the *find* method. We wish to choose a linearization point for the *find*(val, i) so that the two records $pred$ and $curr$, returned as output, satisfy (1) $pred.data[i] < val \leq curr.data[i]$, (2) $pred$ and $curr$ are both i-reachable at the linearization point, and (3) $curr$ is $pred$'s i-successor at the linearization point. Notice that by the time the method returns, $pred$ and $curr$ may no longer satisfy those conditions. In [8] we prove that such a linearization point always exists (it may differ between different executions).

The add *operation:* Assume an *add*(tup) execution that has terminated, and for which $newRecord$ is the record created in line 18. If the execution is successful, we define its linearization point as the execution of line 31, in which $newRecord$'s status becomes *InTable*. As explained above, it is guaranteed that there exists such point during the execution, and $newRecord$ is indeed logically in the table at this point. If the execution is unsuccessful, then the operation's linearization point is set to be the read of $curr$'s status in line 45 (executed by the adding thread or by another helping thread). At this point, there exists a record R, which is logically in the table, and for which $R.data[i] = newRecord.data[i]$ for some unique field number i. Inserting $newRecord$ into the table at this point would break the uniqueness property of our table and thus, returning false is valid in this case.

The remove *operation:* Consider a *remove*(val, i) execution that has terminated. If the execution is successful (returning true in line 62), then the executing thread must have executed a successful logical removal of the table record $victim$ in line 55 (notice that $victim.data[i] = val$). This logical removal is considered as the operation's linearization point in this case.

If the execution is unsuccessful, there are two different scenarios. If $victim.data[i] \neq val$ (causing the return in line 54), at the linearization point of the *find* method, invoked in line 52, there does not exist a record R, which is logically in the table and for which $R.data[i] = val$. In this case, this point is considered as the *remove* linearization point as well. Otherwise, $victim.data[i] = val$ the operation returns in line 56. At the linearization point of the *find* method, invoked in line 52, $victim$ is i-reachable. If $victim$'s status at this point is *InTable*, then since the CAS operation in line 55 fails, some other thread has meanwhile logically removed $victim$ from the table. Since i must be a unique field, right after that logical removal there does not exist any record R which is logically in the table and for which $R.data[i] = val$. That logical removal (by another thread) is considered as the operation's linearization in this case. The only remaining scenario is the one in which $victim$'s status is not *InTable* at the *find* linearization point. The choice of linearization point in this case is delicate, and is fully described in [8].

The retrieve *operation:* For each *retrieve*(val, i) execution that has terminated, we want to set a linearization point for which the set of tuples returned by the operation is the set of all tuples $t = \langle e_1, \ldots, e_{N_T} \rangle$ which are logically in the table and satisfy $t.e_i = val$. In other words, the set of tuples returned by the operation is the set of all $data$ arrays for which $data[i] = val$ and there exists a record R such that R is logically in the table at the linearization point, and $R.data = data$. The operation's linearization point is either set to be the last update of the last $next[i]$ pointer read during the loop in lines 67-68, or to the last execution of line 68 (if this update occurs before the operation's invocation).

When the operation returns an empty set in line 70, there does not exist any record satisfying the above condition at the linearization point. Therefore, the returned set is empty. The case of returning a non-empty set (in line 91) is more complicated. First, it is guaranteed that the record saved in the $first$ local variable (in line 71) is the first record in L_i whose i-th property is val. Given that, we still need to show that the returned set of tuples indeed represents the exact set of records which are logically in the table at this point (notice that the condition checked in line 72 guarantees that $e_i = val$ for all of the returned tuples).

Using our observation regarding the status changes during the execution, the condition checked in lines 84-85 guarantees that all returned tuples represent records with an *InTable* status at the chosen linearization point. Meaning, the returned set of tuples indeed represents only records which are logically in the table at this point. It still remains to show that no relevant records are skipped.

Assume by contradiction that a relevant record, which is logically in the table at the linearization point, is skipped. As mentioned in Section III-D, after the traversal in lines 67-68, new records R satisfying $R.data[i] = val$ can only be inserted into L_i before the record saved inside the $first$ local variable. Since this record is the first such record at the linearization point, all relevant records were in L_i while executing lines 72-76. Moreover, due to our status changes observation, their statuses must have been either *Pending* or *InTable* during this loop. Therefore, the skipped relevant record must have been saved in $tmpSet$, together with its status, in line 75. The condition checked in line 84 guarantees that the status of this skipped record at this point is equal to its status when it is read in line 73. By our assumption, the record is skipped, meaning, its status is *Pending*, at this point and the linearization point as well -

a contradiction. Therefore, the returned set of tuples indeed represents the exact set of records which are logically in the table at the linearization point.

After assigning a linearization point to each completed table, we end our linearization proof with the following theorem:

Theorem 1: The implementation given in Figure 2, 3, 4 and 5 is a linearizable implementation of a table.

Progress

Let us sketch the proof that the table implementation presented in Section III-D is lock-free. Suppose in way of contradiction that this is not the case. Therefore, there is some execution for which no executing operation terminates after a certain point, and no new operation is invoked as well (this assumption is fine since a finite set of executing threads). We then show that there must be a finite number of status changes, pointer markings, and physical insertions and removals during that execution (anything different would derive a contradiction). Therefore, there exists a point for which the table becomes stable. Notice that any list traversal should terminate after this point.

When the list is stable, every *retrieve* execution must terminate (since this operation's termination depends only on the table's stability). Moreover, since there are no more pointer markings and physical removals from the lists, every *find* and *remove* executions must terminate. It still remains to show that the process of adding a new record into the table must also terminate after this point. For the detailed proof, see [8]. We conclude with the following theorem:

Theorem 2: The implementation given in Figure 2, 3, 4 and 5 is a lock-free implementation of a table.

G. Adding a Fast Skip-List Indexing

We improve access to the table by using a skip-list index [13]. In the above sections, each table operation traverses at least one linked-list until reaching the desired record. Such linked-list traversals may yield poor performance. Moreover, in the lock-free version, traversals might restart due to unsuccessful physical removals of marked records. To improve performance, we employ an index mechanism which enables getting to a record's predecessor in a list faster (without traversing from the head of the list). For example, in an execution of *retrieve(*$1005, 3$*)* with the table of Figure 1, instead of starting from the *head* record, our index can be used to start traversing from the record which represents $\langle 80, \text{John}, 1003\rangle$ (marked with $\star$).

For each list L_i we use a linearizable skip-list S_i that serves as an index to the records in L_i. S_i represents a sequence of record pointers that are sorted by their i-th field. S_i supports operations *insert(R)* and *remove(R)* which are used to insert and remove record pointers; their implementation ensure that each pointer may appear at most once in S_i (i.e., S_i is a set). It also supports operation *getPrev(v)* that returns the latest record pointer R in S_i that satisfies $R.data[i] < v$. As noted in [13], such S_i can be realized by a standard linearizable skip list; in our work it is implemented as simple wrapper of Java's *ConcurrentSkipListSet*.

Using the index

In operations *find(*v, i*)* and *retrieve(*v, i*)*, instead of starting the traversal from *head* we start from the record R returned by $S_i.getPrev(v)$. As described below, the content of S_i does not necessarily represent the exact content of the table. In particular, the returned record R may have already been removed from the table. In such a case we invoke $S_i.getPrev(R.data[i])$ and continue in the same manner until *getPrev* returns a valid record (i.e., *R.status* = *InTable*). This process will necessarily stop after a finite number of steps, since *head* cannot be removed from S_i.

Updating the index

Each S_i is updated in a way that attempts to (but does not always) ensure that S_i points to the valid records in the table. When the table is created, *head* is added to all S_i indexes. After completing a successful insertion (or deletion) of a R into (or from) the table, we add (or remove) R to all S_i indexes. Thus, S_i may point to a record that has already been deleted from the table. Furthermore, a removal of record R from the table can be fully executed before R is added to all indexes - in such a case, $S_i.remove(R)$ might be executed before $S_i.add(R)$, resulting in leaving R in S_i forever (even though it is not in the table anymore). To prevent such cases, after adding R to the indexes we check the status of R. If its status is not *InTable* then we explicitly remove R from all the indexes.

IV. Measurements

We compared the performance of our lock-free and lock-based table against each other, and against a global lock-based table and an STM table. All implementations use the skip-list optimization of Section III-G.

Each experiment consists of 10 trials. A trial is a run in which each thread executes randomly chosen operations drawn from the workload distribution, on a table consisting of two unique fields and three non-unique fields. We ran four experiments.

In the first three experiments (presented in the first row of Figure 6), we used a workload distribution with 50% *retrieve*, 25% *add* and 25% *remove* operations. In the last three experiments, we used a workload distribution with 90% *retrieve*, 5% *add* and 5% *remove* operations.

For each workload distribution, we ran the same three tests, which differed in the key range size. In the first experiment, unique fields were selected uniformly at random from the range [0,255], in the second one, from the range $[0,10^4-1]$, and in the third one, from the range $[0,10^6-1]$. For creating repetitions in the non-unique fields (which

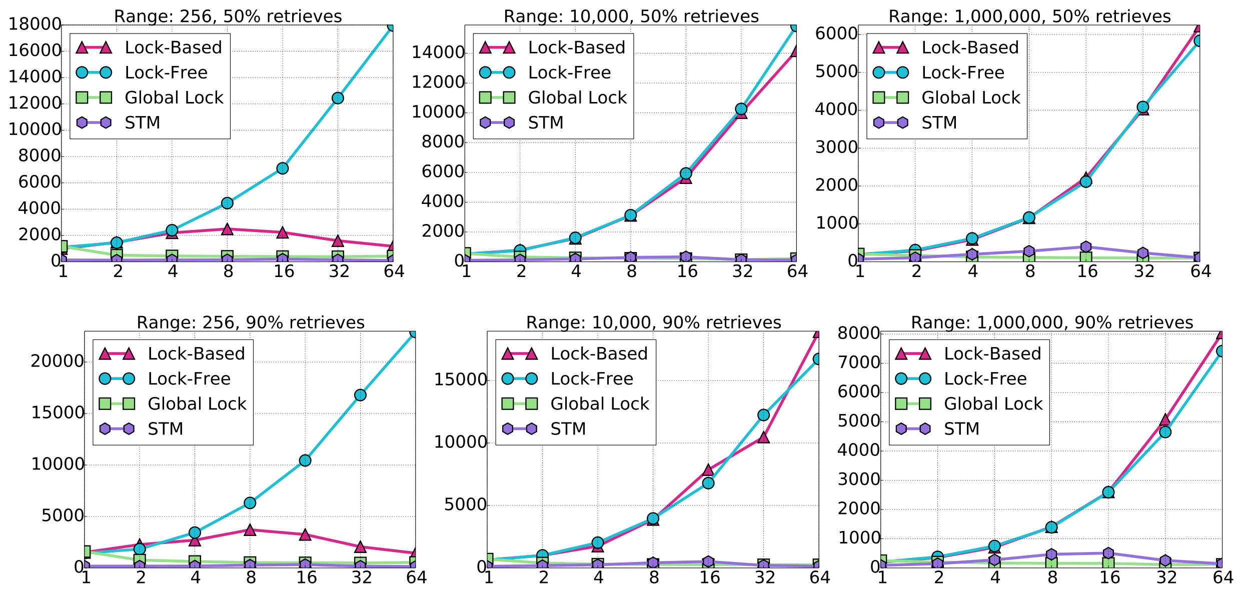

Figure 6. Throughput graphs. The x-axis is the number of threads. The y-axis is the throughput (operations per millisecond).

yields a realistic table content), they were selected uniformly at random from the range [0,63] for the first experiment, from the range $[0,25 \cdot 10^2 - 1]$ for the second experiment, and from the range $[0,25 \cdot 10^4 - 1]$ for the third experiment. The table was initiated with 128 records in the first experiment, with $5 \cdot 10^3$ records in the second experiment, and with $5 \cdot 10^5$ records in the third experiment. The graphs present the average throughput (operations executed per millisecond) over all trials.

We ran our experiments on a machine with 2 Intel Xeon E5-2686 v4 processors, each with 16 cores. Each core has 2 hardware threads (hyper-threading is enabled). All implementations are in Java. We used Java's *AtomicMarkableReference* for the lock-free version and Java's $ConcurrentSkipListSet$ (e.g. [14]) for the skip-list optimization. For our STM table, we adopt the TL2 [15] implementation from Synchrobench [16].

As shown in Figure 6, for a small range of fields (when contention is high), our lock-free table is far-superior to the other versions and scales well. For large fields ranges, our lock-free and lock-based versions present a similar performance, and are far superior to the other two alternatives.

V. Related Work

Many sophisticated concurrent data structures (see [11], [17]–[21]) have been developed and used in modern software systems. The *concurrent Map* is a notable data structure which is widely used in real life concurrent programs [3]. In this work, we extend the Map's capabilities by adding support for efficient thread-safe multiple indexes. We hope that our new data structure will practically extend the Map's usability in concurrent software; especially in programs where the same information is indexed and accessed via several different properties (see [22], [23]).

Many (relational) database systems have built-in support for multiple primary and secondary indexes [1], [24]. Typically, database indexes can be correctly used in the presence of concurrency (e.g., by using transactions [24]). Our table data structure can be seen as an in-memory representation of the well-known database table. Our data structure is much more restricted than a standard database table (e.g., it does not support common database operations like *join* [1] and *snapshot* [6], and is not required to access hard disks). However, its restricted functionality enables creating efficient in-memory implementations which are optimized for modern multi-core machines.

Recently, several multi-index implementations were developed for NOSQL datastores (e.g., see [2], [25]–[29]). However, these implementations have been designed to work on several distributed machines, and are based on complex and expensive mechanisms for handling persistency and inter-machine synchronization. These multi-index implementations utilize a simple in-memory concurrency control which resembles global lock synchronization, and it is not clear how to convert them into effective in-memory data structures that can be used by multi-threaded applications.

A table with multi-index support can be simply realized via general-purpose software transactions [13], [16]. However, in contrast to our specialized table implementations, general transactions provide limited performance due to their high runtime overhead [16], [30], and their excessive thread contention [13].

References

[1] R. Ramakrishnan and J. Gehrke, *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[2] A. Tai, M. Wei, M. J. Freedman, I. Abraham, and D. Malkhi, "Replex: A scalable, highly available multi-index data store," in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, A. Gulati and H. Weatherspoon, Eds. USENIX Association, 2016, pp. 337–350. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/tai

[3] O. Shacham, *Verifying atomicity of composed concurrent operations*. University of Tel-Aviv, 2012.

[4] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[5] J. Vu, "The art of multiprocessor programming by maurice herlihy and nir shavit," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 52–53, 2011. [Online]. Available: http://doi.acm.org/10.1145/2020976.2021006

[6] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling concurrent log-structured data stores," in *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, L. Réveillère, T. Harris, and M. Herlihy, Eds. ACM, 2015, pp. 32:1–32:14. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741973

[7] E. Petrank and S. Timnat, "Lock-free data-structure iterators," in *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, ser. Lecture Notes in Computer Science, Y. Afek, Ed., vol. 8205. Springer, 2013, pp. 224–238. [Online]. Available: https://doi.org/10.1007/978-3-642-41527-2_16

[8] "A scalable linearizable multi-index table." [Online]. Available: http://www.cs.technion.ac.il/~erez/Papers/table.pdf

[9] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, ser. Lecture Notes in Computer Science, D. Malkhi, Ed., vol. 2508. Springer, 2002, pp. 265–279. [Online]. Available: https://doi.org/10.1007/3-540-36108-1_18

[10] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, ser. Lecture Notes in Computer Science, J. L. Welch, Ed., vol. 2180. Springer, 2001, pp. 300–314. [Online]. Available: https://doi.org/10.1007/3-540-45414-4_21

[11] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank, "Wait-free linked-lists," in *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Baldoni, P. Flocchini, and B. Ravindran, Eds., vol. 7702. Springer, 2012, pp. 330–344. [Online]. Available: https://doi.org/10.1007/978-3-642-35476-2_23

[12] K. Fraser and T. L. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, p. 5, 2007. [Online]. Available: http://doi.acm.org/10.1145/1233307.1233309

[13] A. Spiegelman, G. Golan-Gueta, and I. Keidar, "Transactional data structure libraries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. Berger, Eds. ACM, 2016, pp. 682–696. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908112

[14] "Java language home page," http://java.sun.com/. [Online]. Available: http://java.sun.com/

[15] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, ser. Lecture Notes in Computer Science, S. Dolev, Ed., vol. 4167. Springer, 2006, pp. 194–208. [Online]. Available: https://doi.org/10.1007/11864219_14

[16] V. Gramoli, "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 1–10.

[17] A. Braginsky, N. Cohen, and E. Petrank, "CBPQ: high performance lock-free priority queue," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, ser. Lecture Notes in Computer Science, P. Dutot and D. Trystram, Eds., vol. 9833. Springer, 2016, pp. 460–474. [Online]. Available: https://doi.org/10.1007/978-3-319-43659-3_34

[18] A. Braginsky and E. Petrank, "A lock-free b+tree," in *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*, G. E. Blelloch and M. Herlihy, Eds. ACM, 2012, pp. 58–67. [Online]. Available: http://doi.acm.org/10.1145/2312005.2312016

[19] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, A. W. Richa and R. Guerraoui, Eds. ACM, 2010, pp. 131–140. [Online]. Available: http://doi.acm.org/10.1145/1835698.1835736

[20] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, C. Cascaval and P. Yew, Eds. ACM, 2011, pp. 223–234. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941585

[21] S. Timnat and E. Petrank, "A practical wait-free simulation for lock-free data structures," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, J. E. Moreira and J. R. Larus, Eds. ACM, 2014, pp. 357–368. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555261

[22] S. Benchmarks, "Standard performance evaluation corporation," 2000.

[23] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[24] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[25] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB - The Definitive Guide: Time to Relax*. O'Reilly, 2010. [Online]. Available: http://www.oreilly.de/catalog/9780596155896/index.html

[26] A. Cassandra, "Apache cassandra," *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, p. 13, 2014.

[27] K. Chodorow and M. Dirolf, *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2010. [Online]. Available: http://www.oreilly.de/catalog/9781449381561/index.html

[28] A. Khetrapal and V. Ganesh, "Hbase and hypertable for large scale distributed storage systems," *Dept. of Computer Science, Purdue University*, pp. 22–28, 2006.

[29] R. Klophaus, "Riak core: Building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010, p. 14.

[30] C. Cascaval, C. Blundell, M. M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: why is it only a research toy?" *Commun. ACM*, vol. 51, no. 11, pp. 40–46, 2008. [Online]. Available: http://doi.acm.org/10.1145/1400214.1400228