A Specialized B-tree for Concurrent Datalog Evaluation

Herbert Jordan
University of Innsbruck, Austria
herbert.jordan@uibk.ac.at

Pavle Subotić
University College London, UK
pavle.subotic.15@ucl.ac.uk

David Zhao
University of Sydney, Australia
dzha3983@uni.sydney.edu.au

Bernhard Scholz
University of Sydney, Australia
bernhard.scholz@sydney.edu.au

Abstract

Modern Datalog engines are employed in industrial applications such as graph-databases, networks, and static program analysis. To cope with vast amount of data, Datalog engines must employ parallel execution strategies, for which specialized concurrent data structures are of paramount importance.

In this paper, we introduce a specialized B-tree data structure for an open-source Datalog compiler written in C++. Our data structure has been specialized for Datalog workloads running on shared-memory multi-core computers. It features (1) an optimistic locking protocol for scalability, (2) is highly tuned, and (3) uses the notion of “hints” to re-use the results of previously performed tree traversals to exploit data ordering properties exhibited by Datalog evaluation. In parallel micro-benchmarks, the new data structure achieves up to 59× higher performance than state-of-the-art industrial standards, while integrated into a Datalog engine it accounts for 3× higher, overall system performance.

CCS Concepts • **Information systems** → **Data structures**; • **Software and its engineering** → *Constraint and logic languages*;

Keywords Optimistic Locking, B-Tree, Datalog Evaluation

ACM Reference Format:
Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019. A Specialized B-tree for Concurrent Datalog Evaluation. In *PPoPP '19: Symposium on Principles and Practice of Parallel Programming, February 16–20, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3293883.3295719

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295719

1 Introduction

In recent years, Datalog has gained increased popularity in the implementation of advanced data analyses. Applications vary from generic graph databases [50], network tools [18], to static program [2, 10, 23, 45], security [35], and cloud computing analysis [22, 28, 44]. In these applications, Datalog is used as a domain specific language that expresses application semantics naturally and concisely in a declarative fashion. Applications expressed in a few lines of Datalog may require several 100KLOCs in an imperative language, and, hence, automatically deriving the implementation from a Datalog specification is less error-prone and will greatly reduce the time for implementation, testing, and debugging of an application.

Datalog queries are defined on sets of relations – equivalent to tables in conventional relational database systems. However, not all data within the relations is provided explicitly. Deductive rules define how tuples in relations are formed from the content of other relations. It is the task of a Datalog engine to organize and evaluate the rules for computing the tuples. State-of-the-art Datalog evaluation strategies, known as *semi-naïve evaluation* strategies [1], reduce the evaluation of deductive rules to a sequence of relational algebra operations, which require an internal representation for logical relations. Thus, at the core of each Datalog engine[1, 28, 37, 52] is a data structure for representing logical relations that facilitates join, selection, and projection operations. The range of utilized data structures includes (in-memory) B-trees, tries, ordinary arrays, and BDDs – all providing their advantages and drawbacks. While several state-of-the-art engines have shown promise by the use of data structures such as hash-sets [25], B-trees [9], and OBDDs [52] to model relations, they still do not facilitate parallel evaluation strategies.

Despite the vast body of Datalog evaluation research, Datalog engines have been seen as sub-par when compared to hand-crafted programs. For example, Reps (1995) [43] reports that the Datalog implementation of an inter-procedural data-flow analysis is 4-6 times slower than hand-crafted C code. However, recently the Datalog engine Soufflé [28] has demonstrated performance on a par with hand-crafted tools for complex rulesets and large datasets. Unlike classical Datalog interpreters, Soufflé synthesizes a native parallel C++ program from a given Datalog specification. The generated C++ program has data structures that are specialized for the given deductive rules of a Datalog specification. This approach provides outstanding sequential performance. However, to utilize shared-memory multi-core machines, Soufflé is in need of an efficient concurrent data structure that facilitates the parallel evaluation of deductive rules.

In this paper, we introduce a specialized concurrent data structure for Datalog that is based on in-memory B-trees. Our data structure is fundamental for the efficient parallel execution of relational algebra operations on contemporary shared-memory multicore machines. Our in-memory B-tree implementation for Datalog enables Soufflé to evaluate very large scale analysis problems including context-sensitive points-to and network analysis for industrial-scale applications that were previously deemed too large for Datalog engines to perform.

Contributions. We make the following contributions:

- We design a specialized concurrent data structure for parallel semi-naïve evaluation that is based on in-memory B-trees supporting shared-memory multi-core machines.
- We present an optimistic fine-grained locking scheme for our B-tree implementation. We introduce a new *optimistic read-write lock* extending *seqlocks* [32] for our implementation. Our new synchronization scheme delivers high throughput in frequent write use-cases, even on multi-socket architectures.
- We present a "hint" mechanism for our B-trees operations that exploit spatial and temporal locality in tree accesses. By caching previous accesses, subsequent tree traversals may be eliminated.
- We provide experimental evidence that our specialized concurrent data structure based on in-memory B-trees is scalable and performs well on shared-memory multicore machines.

2 Parallel Datalog Evaluation

Datalog specifications define input and output relations. The input relations are given in the form of facts or input files. The output relations are generated by logic rules. For example, let *edge* be a binary input relation. The two logical rules

$$path(X,Y) \text{ :- } edge(X,Y). \tag{1}$$

$$path(X,Z) \text{ :- } path(X,Y), edge(Y,Z). \tag{2}$$

implicitly define the content of the output relation *path* computing the transitive closure of the edge relation. The first rule includes all tuples of relation *edge* in relation *path*. The second rule is a recursive rule and adds transitive paths inductively, i.e., if there is $(X,Y) \in path$ and $(Y,Z) \in edge$, then the pair (X,Z) will be in relation *path*. Real-world use-cases may comprise hundreds of relations, connected through hundreds of (potentially mutually recursive) rules.

```
1  using Tuple = array<size_t,2>;
2  using Relation = set<Tuple>;
3  Relation evaluate(const Relation &edge){
4   Relation path = edge, deltaPath = edge;
5   while(!deltaPath.empty()){
6    Relation newPath;
7    for (const auto &t1: deltaPath){
8     auto l = edge.lower_bound({t1[1],0});
9     auto u = edge.upper_bound({t1[1]+1,0});
10    for (auto it=l; it!=u; ++it){
11     auto& t2 = *it;
12     Tuple t3({t1[0],t2[1]});
13     if (path.find(t3) == path.end())
14       newPath.insert(t);
15    }
16   }
17   path.insert(newPath.begin(), newPath.end());
18   deltaPath.swap(newPath);
19  }
20  return path;
21 }
```

Figure 1. Synthesised C++/STL code for Path Example

Soufflé [28] compiles the transitive closure example to C++ code as listed in Figure 1. The code outlines the anatomy of the underlying least fixed-point calculations of the semi-naïve evaluation [1] using STL's set data structures for relations. The synthesized function `evaluate()` has the input relation *edge* as an argument and computes the output relation *path*. The non-recursive rule (1) causes the initialization of relation *path* with the tuples of relation *edge*. The fixed-point calculation for the recursive rule (2) is performed in the while-loop from line 5 to line 19. In each iteration of the while-loop, new tuples are produced for relation *path*. Thereby, the auxiliary relations *newPath* and *deltaPath* are used to avoid re-computations of already generated tuples. If no further tuples can be found, the fixed-point algorithm will stop, and relation *path* will be the transitive closure of *edge*.

The loop-nest in line 7-11 finds matching edges $t2 \equiv (Y,Z)$ for each newly discovered path $t1 \equiv (X,Y)$ in relation *deltaPath*. The adjacent edges $t2$ can be found efficiently because the STL set uses a lexicographical order over the edge set. The lexicographical order for two edges (u,v) and (u',v') is defined as $(u,v) \leq (u',v')$ if and only if $u < u' \vee (u = u' \wedge v \leq v')$. If the node-set is totally ordered (i.e., for each node there exists a unique number), then the lexicographical order is a total order of the edge set. With the lexicographical order, STL's red-black tree efficiently performs the range traversal of all matching/adjacent edges and no scan over the whole relation *edge* is necessary. If a matching edge pair is found, a new path edge $t3 \equiv (X,Z)$ (cf. line 12) is constructed. If $t3$ is not yet known (line 13), it will be added to the set of newly discovered paths (line 14). Finally, newly discovered entries are merged into the path relation and promoted to be the *deltaPath* set for the next iteration (line 17 and 18).

With the exception of the insert operation in line 14, all operations within the nested for-loop between line 7 and line 16 are read-only operations or targeting non-shared memory locations. Thus, the recursive rule can be made parallel (e.g., by merely parallelizing the outermost for-loop in line 7) by providing a synchronized insert operation. However, STL's set implementation misses this opportunity due to the lack of synchronisation.

Hence, a parallel Datalog evaluation requires a concurrent set data structure providing an STL-like interface. Elements in the set must be sorted so that range queries efficiently filter tuples. For the Datalog evaluation, the following concurrent operations are essential: *insert*(t) inserts a fixed sized n-ary integer tuple t into a set of n-ary tuples; ignoring duplicates, *begin*() and *end*() for container traversals, *lower_bound*(a) and *upper_bound*(a) for range queries, *find*(t) for checking the existence of tuple t, *empty*() for checking whether the container is empty. Note that no deletion operation is required since relations in Datalog can only grow (and never shrink).

The Datalog's rule evaluation has very specific use-patterns of the underlying set data structure. These use patterns can be optimized for better parallel performance. For example, the usage of a set data structure has two phases (cf. [51]): (1) either there are multiple writers but no readers, or (2) there are multiple readers but no writers. For the example in Figure 1, the nested loops in lines 7 and 10 read pairs from relation *deltaPath* and *edge* but do not insert new pairs into *deltaPath*/*edge*. Newly found pairs are inserted in relation *newPath* but no read operations on *newPath* exist in this loop nest. The semi-naïve evaluation guarantees this two phases for all involved sets in its rule evaluation, i.e., there are no interleaved reads and writes. Hence, read operations do not need synchronization; only the write operation must be synchronized. Furthermore, the tuples of the relations in the body of a rule are traversed in a sorted fashion, since the relations are sorted. Hence, newly found tuples are generated in a sorted fashion as well. Hence, the set data structure can exploit a tuple order for minimizing the search overheads in the data structure.

3 A Specialized B-tree for Datalog

A data structure that is highly suitable for the sequential evaluation of Datalog programs is the B-tree data structure [6]. B-trees are balanced binary search trees with insertions and searches in $O(\log n)$ worst-case execution time. They have efficient traversals in $O(n)$, and range queries in $O(\log n) + r$ where r is the size of the range. B-trees are memory efficient using $O(n)$ space. In high write-throughput situations, larger bucket sizes provide sufficient capacity to fill the buckets lazily, avoiding the reorganisation of the tree frequently. By consecutively storing data in memory, B-tree implementations greatly benefit from modern memory architectures using caches effectively.

However, current state-of-the-art implementations of B-trees cannot cope well with a parallel Datalog evaluation since their synchronisation mechanisms must have very little overheads. In the case of Datalog, we observe that read B-tree operations are always followed by several write-only operations. Thus, unlike in a traditional setup, we have two distinct phases for reading and writing. Another observation in Datalog workloads is that data is highly sorted. As a result, the same tree traversals frequently reoccur, and can be cached for future use.

In this section, we describe an in-memory B-tree data structure for parallel Datalog workloads that has a new optimistic fine-grained locking scheme using *optimistic read-write locks*. Our data structure has a high throughput in frequent write use-cases (even on multi-socket architectures). We introduce a "hint" mechanism for our B-trees operations that exploit spatial and temporal locality in tree traversals by caching them.

3.1 Synchronization

The parallel semi-naïve evaluation of Datalog guarantees two phases (cf. [51]) in a relational data structure, i.e., in every parallel context B-trees are either exclusively queried or written – or neither. However, a relation will never be queried while being modified. Therefore, we need a synchronisation mechanism for concurrent insertions only.

In the past, a large variety of locking schemes have been introduced [21]. Approaches range from globally locking the entire tree, over fine-grained mutex based locking, fine-grained read/write lock based locking, to hardware transactional memory based approaches [30]. Lock-based approaches on NUMA [24] architectures, especially beyond the single-socket boundary, suffer from the high bandwidth requirements due to frequent cache line invalidations when acquiring and releasing locks. In particular, the lock protecting the root node, which needs to be traversed for virtually every single operation, introduces a performance penalty for all operations.

Optimistic Read-Write Lock. A lock mechanism has been devised for concurrent heavy read / seldom write use-cases called *seqlocks* [32]. It represents an optimistic locking mechanism and was initially discussed in database applications [38]. Seqlocks solve the problem of synchronizing reader/writer threads that access shared data concurrently. A version number (aka. lease) is used to synchronise the access of the shared data. Before reading the shared data, the reader thread first records the version number and reads the shared data, which is not validated yet. After reading the shared data, the reader

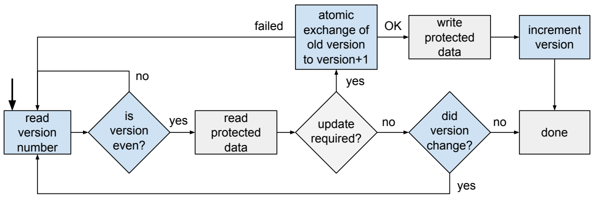


Figure 2. Outline of optimistic locking scheme.

thread verifies that the version number is even and has not changed. If the version number has changed or is odd, the read thread restarts itself. Otherwise, the read shared data is validated and can be consumed. When a writer thread begins writing shared data, it will wait until it reads an even version number and will increment the version number such that the version number becomes an odd number. After modifying the shared data the write thread increases the version number again. Note that concurrent writer threads are synchronized by observing an even number before writing.

For our B-Tree implementation we adapt seqlocks such that the locking of nodes in the B-tree and the locking of the root node reference become more efficient. Our new *optimistic read-write* lock has *read-potential-write* threads only (instead of having distinct read threads and write threads) and combines the roles of reader and writer threads. A read-potential-write thread reads the shared data and decides after reading the shared data whether it wants to modify the data. Thus, a read-potential-write thread may acquire a read lock first, inspects the shared data, and decides after the read whether a write to the shared data is necessary, and upgrades to a write-lock. Combining the roles of a read and writer thread provides better performance in our use-case, since shared data is read first and only after the inspection of the shared data, it may change to a write lock.

Figure 2 illustrates the state of our optimistic read/write-locking scheme. Our optimistic read/write lock provides eight operations: *start_read*, *valid*, *end_read*, *try_upgrade_to _write*, *end_write*, *abort_write*, *try_start _write*, and *start_write*. The *start_read* initiates a new read phase, returning a *lease*, i.e., the version number. This lease can be used to test whether concurrent updates occurred using the *valid* operator. The lease is also required to end the read phase using the *end_read* operation, or to attempt to enter a write phase through the *try_upgrade_to_write* operation. The *end_write* operation is used to mark the end of a write phase, and the *abort_write* terminates a write phase when no modification took place. Finally, the *try_start_write* operation attempts to directly enter a write phase. All of those are non-blocking. The only blocking operation is the *start_write* operation, which will block until a write access is granted.

Optimistic B-tree Insertion. Algorithm 1 outlines our optimistic B-tree insertion procedure. Each node is equipped with an optimistic read-write lock, and an additional *root_lock* protects the root node pointer. The thread safe initial creation of a root node is covered by lines 2-9. Lines 11-49 insert values in a non-empty tree. Lines 13-17 obtain a pointer to the current root node, while lines 20-33 navigate down the tree. Once a leaf node is reached, lines 35-47 inserts a new value. In case the targeted leaf node is full, the node splitting procedure outlined by Algorithm 2 is invoked. In this procedure, write permissions to nodes are requested bottom-up until a node that is not full – and thus does not need to be split – or the tree's root lock is reached (lines 2-23). Afterward, node splitting is performed (line 26) and the locks are released in reverse order (lines 28-35). Note that splitting is performed such that half of the original keys are kept in the existing node and the other half is moved to a newly created node. Since our data-structure can only grow (i.e. the evaluation does not require a delete operation), nodes in memory are never deleted or replaced by other nodes.

Algorithm 1 Optimistic B-tree insertion procedure.

```
 1: procedure INSERT(tree,val)
 2:     // safely initialize root node pointer
 3:     while tree->root == null do
 4:         if !(try_start_write(tree->root_lock)) continue
 5:         if tree->root == null then
 6:             tree->root ← < create new node >
 7:         end if
 8:         end_write(tree->root_lock)
 9:     end while
10:
11: restart:
12:     // safely obtain root node and its lock
13:     repeat
14:         root_lease ← start_read(tree->root_lock)
15:         cur ← tree->root
16:         cur_lease ← start_read(cur->lock)
17:     until end_read(root_lease)
18:
19:     // descent into the tree
20:     while true do
21:         // if value to be inserted is present => done
22:         if contains(cur,val) and valid(cur_lease) return
23:
24:         // process inner node
25:         if cur->inner then
26:             next ← find_next(cur,val)
27:             if !valid(cur_lease) goto restart
28:             next_lease ← start_read(next->lock)
29:             if !valid(cur_lease) goto restart
30:             cur ← next
31:             cur_lease ← next_lease
32:             continue
33:         end if
34:
35:         // request write access to located leaf node
36:         if !try_upgrade_to_write(cur_lease) goto restart
37:
38:         // make some space, if necessary
39:         if full(cur) then
40:             SPLIT(tree,cur)
41:             end_write(cur_lock)
42:             goto restart
43:         end if
44:
45:         // insert value into this leaf node
46:         < insert value in current node >
47:         end_write(cur_lock)
48:         return
49:     end while
50: end procedure
```

In all those steps, the following rules are obeyed: The values stored in a node as well as child node pointers are protected by the corresponding node's lock. However, the parent pointer is covered by the parent's lock or the tree's

Algorithm 2 Optimistic B-tree node splitting procedure.

```
procedure SPLIT(tree,node)
    // write-lock path bottom-up
    cur ← node
    parent ← cur->parent
    path ← $\epsilon$
    while true do
        if parent != null then
            while true do
                start_write(parent->lock)
                if parent == cur->parent break
                abort_write(parent->lock)
                parent ← cur->parent
            end while
        else
            start_write(tree->root_lock)
        end if
        path ← append(path,parent)

        // stop at root or non-full inner node
        if parent == null or !full(parent) break
        cur ← parent
        parent ← cur->parent
    end while

    // conduct actual split
    < split node and propagate to parents >

    // unlock path top-down
    for all parent in reverse(path) do
        if parent != null then
            end_write(parent->lock)
        else
            end_write(tree->root_lock)
        end if
    end for
end procedure
```

root lock, for the root node. Before reading any value, read access through the corresponding lock is acquired and before using the obtained information, the read phase is validated (e.g. line 28 in Algorithm 1). Similarly, every modification is enclosed by the handling of a corresponding write phase. The detection of conflicts is handled by restarting the insertion procedure.

The above locking scheme ensures that situations, where data is concurrently read and written, will be discovered and that write access is exclusive. However, since it is optimistic, it does not prevent concurrent read and write access in the first place. Thus, the code has to make sure that read data is validated by checking the version number before using it. This is in particular important for the utilization of child-node pointers, where utilizing a corrupted pointer could lead to segfaults.

Another complication is the requirement that the possibility of updating a read permit on a node to a write permit must only be ceded once it is ensured that the child node visited next during the insert is not going to split and thereby causing an update to the parent. This is a common requirement on all fine-grained locking schemes, and handled by lines 26-30 in Algorithm 1.

The major benefit of the optimistic locking scheme is that for the most frequent case of merely reading the state of an inner node, no write operation – and thus no cache line invalidation and no bus communication – is required. Neither the version number nor the node content has to be modified. For a conventional read/write lock at least the state of the lock would have to be altered.

Optimistic B-tree Implementation. Although theoretically sound, providing a sound implementation of a seqlocks using any programming language is difficult. At its core, seqlocks constitute data races, for which most programming languages do not provide any semantics. In particular, in our case, C++'s semantic defines the behavior of programs with data races as *undefined*. Consequently, implementations require special care.

Boehm investigated the problem of realizing a sound implementation of seqlocks using modern C++ and its associated memory model [7]. In his work, several ways to realize sound seqlock implementations are discussed. Unfortunately, each of those require not only the adaptation of the lock code but also the wrapping of all data elements accesses within the critical region inside of C++'s atomic wrapper type. Furthermore, to keep the resulting performance penalty low, explicit memory order constraints need to be provided. Ultimately, by carefully selecting memory orders and introducing fences, this allows the compiler to omit code relying on the native memory model of the target platform (e.g. x86), avoiding explicit atomic operations, and thus overhead. For our implementation, we adapted the solution presented by Boehm based on the utilization of acquire fences. We perform synchronization in the following steps: (1) read the version number with `memory_order_acquire`, (2) keys / pointers / element counters are all read with `memory_order_relaxed`, (3) before validating the version number we use `memory_fence_acquire`, and (4) followed by reading the version number with `memory_order_relaxed`.

3.2 Operation Hints

The nested for loops (implementing a nested loop joins [1]) in the Datalog evaluation computation maintain an order (lexicographical) on the data being searched or inserted into a given relation. As a result, this process results in locating or inserting data in close temporal proximity in a B-tree. Hence, by storing previous tree traversals, the new operation may be able to short cut the traversal and reuse the locality of a previous operation.

For example, we have two consecutive insert operations for the pair $(7, 10)$ and $(7, 4)$ that are close according to their lexicographical order. When searching for the existence of both pairs, the second insert operations most likely will try to locate the same leaf node and the tree traversal of the first insert operations can be utilized by checking whether the leaf node of $(7, 10)$ may possibly contain the pair $(7, 4)$ as well. If the pair $(7, 4)$ is not contained by the leaf node, a new tree traversal is initiated. This check has very little computational overheads.

To exploit the spatial and temporal locality of data items, we introduce the *hint mechanism*, which stores pointers of recently accessed leaf nodes. If the tuple to be inserted/queried is within the range covered by a retained leaf node, the tree traversal for finding the leaf node can be skipped. The hint mechanism also minimizes the synchronisation overhead caused by locking caused by the traversal. Note that since tree nodes are never deleted nor moved, the hint pointers never become a dangling reference/invalidate themselves.

Hints are maintained in thread local storage and passed on as arguments in each invocation of an operation. Hence, the hint mechanism has very little computational and memory overheads for parallel B-trees. However, achieving compatibility with tree synchronization schemes remains challenging for operation hints. In principle, such schemes have two options for safely acquiring sequences of locks to avoid deadlock: either a top-down or a bottom-up scheme. Both schemes ensure non-cyclic dependencies between threads holding locks, thereby ensuring deadlock free processing. Introducing hints for insertion operations may result in threads, upon a successful reuse of a previous tree traversal, starting their navigation through the B-tree at the leaf level and walk upwards in case nodes need to be split. Thus, a bottom-up policy for acquiring node locks is required and simpler, top-down based synchronization mechanisms are not compatible.

In our optimistic B-tree design, locks are also acquired top-down (see Algorithm 1). However, in those steps merely read-permissions are acquired. Thus, when entering the phase of mutating the tree, actual exclusive write-permissions are acquired bottom-up (see Algorithm 2). This can not lead to a deadlock since read-permission acquirement is non-blocking and threads owning those cannot prevent other threads from acquiring write permissions. Thus, the deadlock relevant type of permission is the write lock acquired bottom-up. Consequently, our locking mechanism is compatible with operation hints directly jumping to node leaves during insertions, skipping the navigation through the tree.

Implementation wise, our B-tree provides a factory function for initial operation hints. Every thread may invoke this function to obtain default hints, which can then be passed to B-tree operations where they are gradually updated. All four most frequently utilized operations, lookups, inserts, lower- and upper-bound queries, are equipped with such hints, tracing located nodes independently. For each of those, the last accessed leaf-node is maintained as a hint for future invocations.

Implementation Notes. Our C++ implementation includes a number of tuning optimizations to specifically target improved Datalog evaluation performance: (1) collapsing recursive operations to iterative ones, (2) implementing a custom 3-way comparator for tuples, and (3) a specialized merge operation which leverages the structure in one B-tree when merged into another. These tuning optimizations target implementation details and reduce the overall runtime of the operations; they do not provide any new algorithmic insights. The C++ implementation is an open-source implementation used in the Soufflé Datalog engine [16], and is licensed under the UPL V1.0 license. The data-structure is implemented as a C++ template and can be found in the file `BTree.h`.

4 Experiment

In this section, we evaluate the performance of our specialized concurrent B-tree data structure using benchmarks. Those benchmarks evaluate the execution time of the most frequently utilized operators within the Soufflé Datalog engine. Additionally, we evaluate the performance of our B-tree running Soufflé on real-world Datalog programs.

Besides our B-tree implementation (denoted as *btree* or *optimistic btree*), we include the following additional data structures in our evaluation. For the sequential performance we evaluate:

- C++'s *std::set*, denoted as *STL rbtset*, as an example of a balanced tree based in-memory data structure (red-black tree) satisfying all requirements stated for Datalog relations
- C++'s hash based *std::unordered_set*, for clarity denoted as *STL hashset*, providing theoretically superior insertion and lookup performance of $O(1)$, yet no efficient support for range queries
- a state-of-the-art B-tree implementation provided by Google [20], denoted as *google btree*, to evaluate the quality of our optimistic B-tree implementation
- a sequential version of our B-tree, to evaluate the impact of our locking scheme on the performance of operators with and without operation hints respectively, denoted as *seq btree* and *seq btree (n/h)*

For the evaluation of parallel operations we evaluate:

- the *concurrent_unordered_set* implementation of Intel's TBB library version 2017_U7 [42] denoted as *TBB hashset*, representing an industry standard, state-of-the-art concurrent hash-based set implementation
- an implementation of our B-tree without operation hints denoted as *btree (n/h)*
- a parallel reduction based set implementation where insertions take place on thread private set instances, before being merged in a parallel reduction step; the implementation uses Google's B-tree with OpenMP's user-defined reduction operation support; it is denoted as *reduction btree*

Table 1 provides a summary of all data structures included in our evaluation.

Most experiments presented in this section have been conducted on a 4-socket Intel(R) Xeon(R) CPU E5-4650 system (8 cores each, 32 total) equipped with 256GB memory using GCC 5.4.0 with -O3 optimization. For these multi-threaded

Table 1. Summary of investigated data structures.

designation	thread safe	description
STL rbtset	no	C++ standard library's set, implementing a red-black tree
STL hashset	no	C++ standard library's unordered set; a hash based set
google btree	no	Google's B-tree
TBB hashset	yes	Intel Threading Building Blocks' concurrent unordered set
seq btree	no	a sequential verions of our B-tree
seq btree (n/h)	no	our sequential B-tree without hints
reduction btree	yes	Google's B-tree combined with bulk inserts through parallel reduction
btree	yes	our optimistic B-tree
btree (n/h)	yes	our optimistic B-tree without hints

experiments, GCC's OpenMP implementation is used as the underlying parallel runtime system, threads are pinned to cores, and sockets are filled before threads are assigned to additional sockets. Single-threaded experiments have been conducted on an Intel Xeon Gold 6130 system equipped with 192GB memory, using GCC 5.5.0 with -O3 optimization.

4.1 Sequential Performance

In the first step, we evaluate the execution time required for the three most performance critical operations: inserts, membership tests, and range queries.

For evaluating the performance of the insert operation, we insert varying numbers of 2D points[1] into initially empty sets, measure the overall execution time and compute the achieved throughput in inserts/s. Thereby we distinguish between an ordered and unordered use case. In the ordered, the elements are inserted in their lexicographical order, in the unordered, a random order is employed.

For the membership query benchmark, we insert the same sets of elements into our candidate data structures, followed by querying for each element once in order and in a random sequence. The computation time of all queries is recorded and the amortized query performance in queries/s obtained.

Finally, for the evaluation of the range query operation, we are focusing on the cost of scanning (aka iterating) through a range of elements, since the cost of locating the start of a range is already covered by the membership test. Thus, for this benchmark we are measuring the number of entries visited per second while iterating once through the entire set of elements within a relation. Note that hints are not applicable to the iteration operation.

Figure 3 summarizes the collected performance data. The first row illustrates the sequential insertion performance. As can be observed, B-tree based implementations clearly outperform alternative data structures in the ordered as well as in the random insertion load case. This even holds up against hash-based data structures, which exhibit theoretically superior asymptotic runtime complexity. However, the cache-friendly organization of data within B-trees causes significantly less cache misses, and thus results in superior performance compared to the random memory access pattern inherent in hash-based data structures. Among B-trees ordered insertions result in approximately 5× higher performance than random insertions – in part due to improved cache utilization and the reduced complexity of inserting elements within leaves in order. Furthermore, for the random order case, a larger number of elements in the sets leads to weaker performance. While in both cases more inner nodes need to be passed to reach insertion points, in the random case those inner nodes are more likely to trigger cache misses. Also, in both cases operation hints can not amortize their maintenance costs and overhead caused by the locking scheme can be observed (up to $\sim$ 25% in ordered, and $\sim$ 15% in random).

Figure 3c and Figure 3d illustrate the observed query performance for ordered and unordered sequences of accesses. While all data structures provide roughly the same performance, two outliers can be observed: for one, in the ordered case, hints provide an up to 6× performance boost for membership tests to B-trees, since navigating the tree can almost always be circumvented thanks to the hint. The second outlier is provided by STL's unordered set providing high performance for small datasets (e.g 2× faster than the TBB version). This advantage vanishes with growing data set sizes.

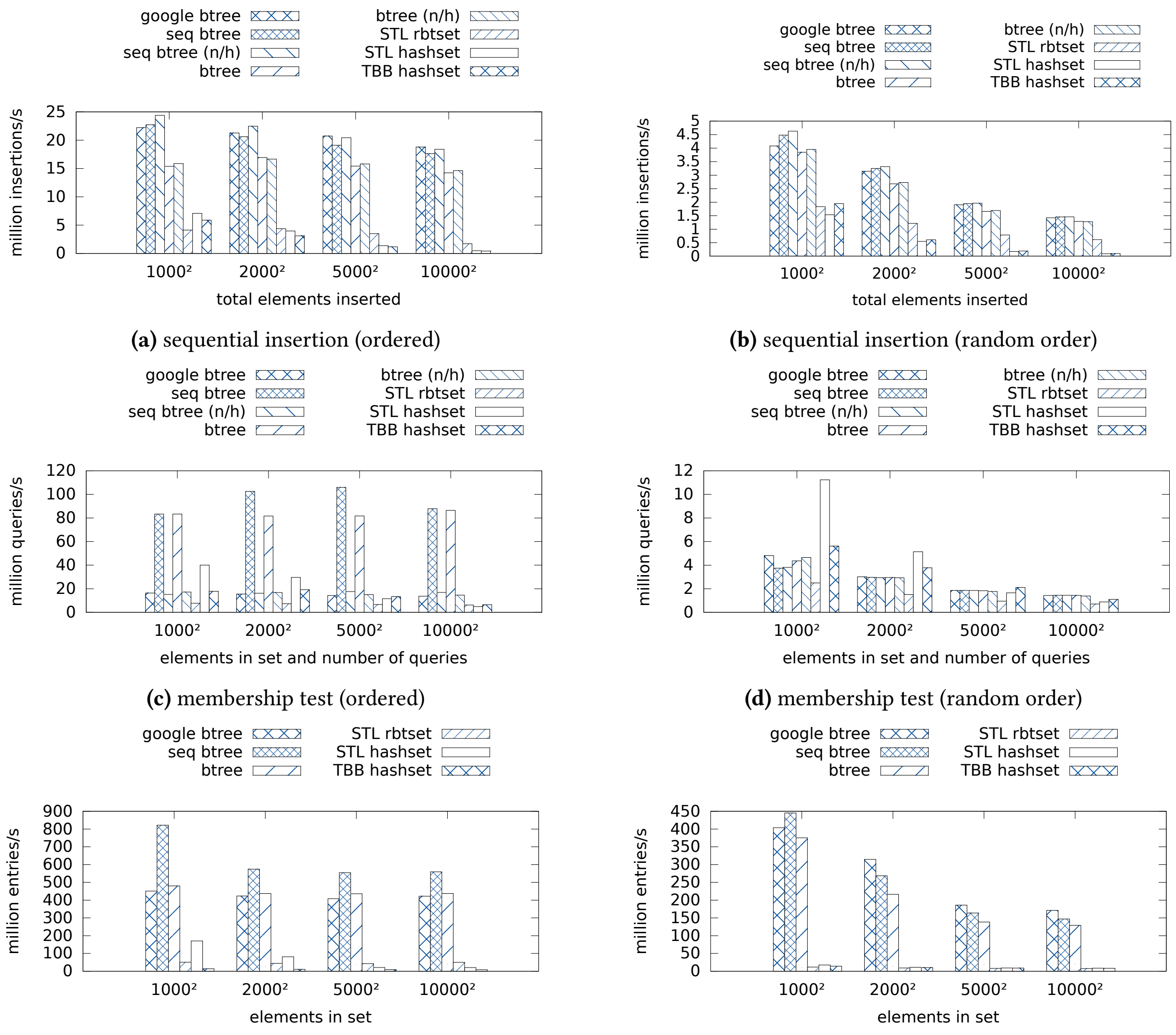
Finally, Figure 3e and Figure 3f summarize the rate at which elements stored in a data structure can be iterated through. Here, the compact storage of data in B-trees also facilitates efficient iterations. The filling order, however, has an impact on the filling grade of leaf nodes, affecting the overall efficiency. Higher filling rates, as caused by in-order inserts, lead to a more compact tree, involving fewer nodes. Since every node switch is a potential cache miss, this leads to improved iteration speed. Finally, as can be observed, the integration of synchronization techniques, and thus the necessary wrapping of key elements into atomic types, is causing a performance deficit for our optimistic B-tree compared to its sequential equivalent.

Overall, the data underlines the superiority of B-tree based data structures over hash or red/black tree based structures due to their cache efficiency. Also, it demonstrates that our optimistic B-tree implementation exhibits sequential performance characteristics comparable to Google's state-of-the-art B-tree implementation. However, while Google's solution is thread unsafe, our concurrent B-tree is able to scale well in a parallel environment.

4.2 Parallel Performance

To evaluate the parallel insert performance, we evaluate the parallel scalability of our contestants. To that end, we insert 100M 2D points into an initially empty set using a varying number of threads (strong scaling). Figure 4 summarizes the obtained results for all our contestants when gradually scaling the computation up to the full size of our two test

[1] 2D data is the most relevant case in many Datalog queries; besides, results remain similar for other dimensions



(a) sequential insertion (ordered)

(b) sequential insertion (random order)

(c) membership test (ordered)

(d) membership test (random order)

(e) full-range scan (after ordered insert)

(f) full-range scan (after random insert)

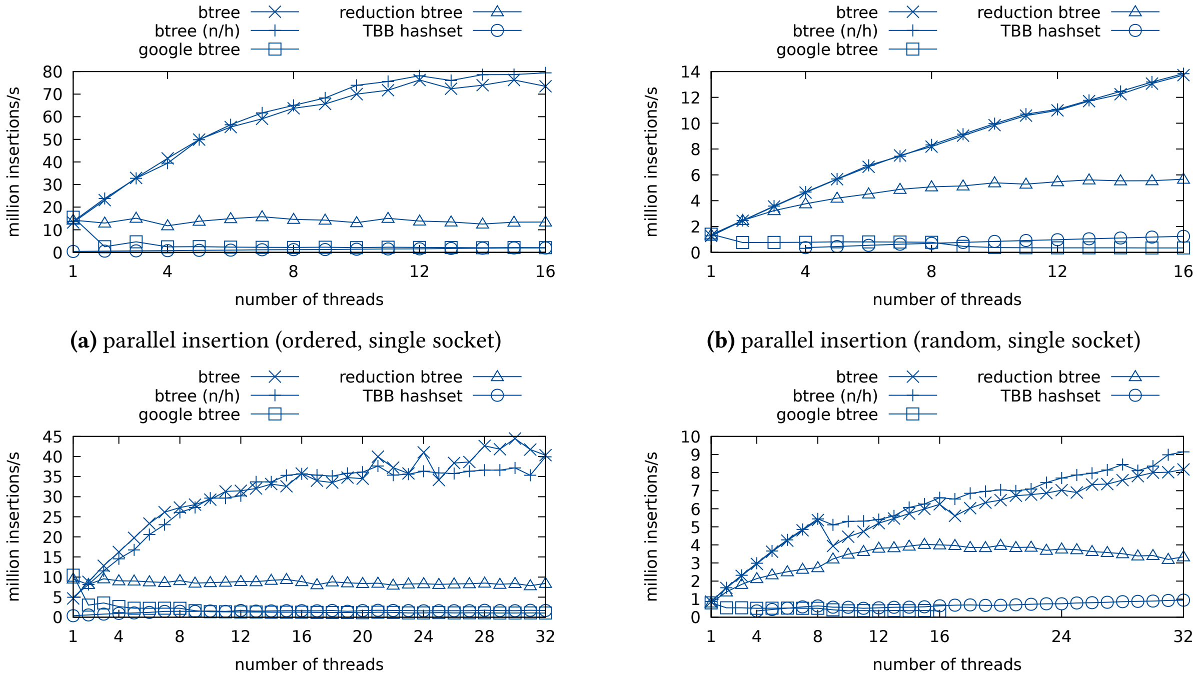
Figure 3. Sequential performance of performance critical set operations.

systems. Besides the three thread-safe data structures (btree, btree (n/h), and TBB hashset) we included two configurations realizing synchronization through external means: one variant with a global lock synchronizing insertions (*google btree*) and another where inserts are performed by threads on thread-private copies and merged through a subsequent reduction step (*reduction btree*). While both techniques could be applied to any set implementation, for our evaluation we chose the fasted sequential external option available to us – the Google B-tree.

Among the five contestants, only the global-lock based approach failed – predictably – to gain performance improvements through the utilization of more than a single thread. The remaining data structures manage to do so in at least one scenario. In absolute terms, however, TBB's implementation is not able to compensate for its sequential performance deficit through parallel scaling. Our optimistic B-tree outperforms TBB's implementation by at least a factor of 8.5 with 1 thread, and up to a factor of 59 with more threads.

The reduction based approach manages to obtain speedups over sequential performance in the random order cases, where the computation effort for the thread-local insertions is dominating the concluding merge. By reducing this domination – either by making the thread-local insertions more efficient (ordered) or by reducing the number of elements inserted by each individual thread (more threads) the potential of this approach is reduced.

Regarding our B-trees, it can be observed that the operation hints cause little differences in the parallel insertion performance in the given use cases – as it has been observed in the sequential experiment above. Both manage to obtain a speedup of approximately 5.8, 10.8, 8.8, and 10 in the 4 use



(a) parallel insertion (ordered, single socket)

(b) parallel insertion (random, single socket)

(c) parallel insertion (ordered, multi socket, 4x8 cores)

(d) parallel insertion (random, multi socket, 4x8 cores)

Figure 4. Parallel performance of insert operations.

cases. Furthermore, Figure 4c and Figure 4d illustrate the scalability of the optimistic locking approach beyond socket boundaries, while TBB's hashset sustains significant performance loses beyond the first socket boundary (not visible in 4d). Figure 4c thereby constitutes a use case performing most operations within NUMA boundaries. This is due to the partitioning of the elements to be inserted among the threads in the benchmarks, their pinning to cores, as well as the default first-touch NUMA policy. It thus demonstrates the achievable performance when being NUMA aware, while Figure 4d does not have this advantage. Consequentially, clear performance drops whenever growing beyond a single NUMA domain are observed. Our data structure is not inherently NUMA aware.

The provided data demonstrates the superior parallel capabilities of our data structure within single and multi-socket environments, compared to the reference data structures.

4.3 Processing Datalog Queries

For our last experiment, we adapted the Soufflé Datalog engine to utilize different data structures in its evaluation process. On top of this, we conducted large-scale, real-world analyses and evaluated the resulting performance.

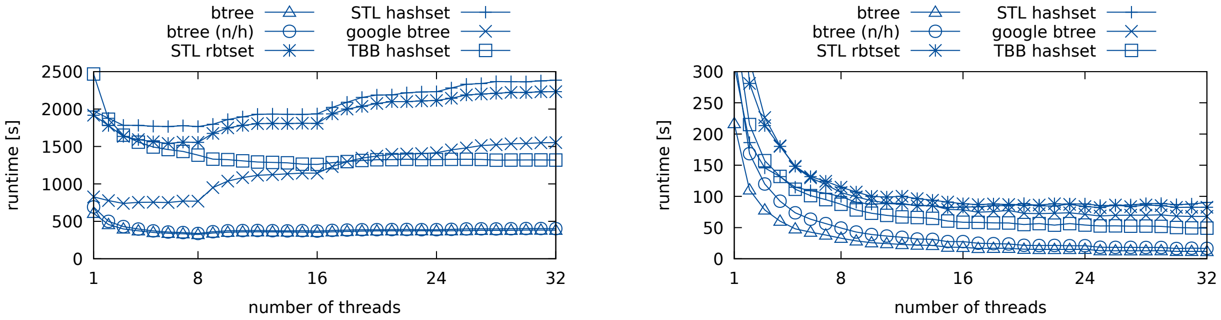
For our evaluation, we utilized two real-world benchmarks: a context-sensitive var-points-to analysis using the Doop framework [10], and a security vulnerability analysis for an Amazon EC2 network. The Doop analysis was run on the suite of DaCapo benchmarks, comprising 11 different Java programs. Both analyses comprise 100s of relations and rules. Table 2 summarizes the properties of those two benchmarks, along with runtime statistics for the inserts, membership tests, and lower/upper bound calls.

Table 2. Real-World Datalog Benchmark Properties.

Datalog Property	Doop on DaCapo (avg. per benchmark)	Amazon EC2 security vulnerability
relations	493	287
rules	810	236

Evaluation Statistics	Doop on DaCapo (avg. per benchmark)	Amazon EC2 security vulnerability
inserts	8.3e7	2.1e7
membership tests	1.5e8	4.2e9
lower_bound calls	2.1e8	2.5e9
upper_bound calls	2.1e8	2.5e9
input tuples	8.3e6	3515
produced tuples	2.5e7	1.6e7

Figure 5 illustrates the collected performance data for our two benchmarks. Note that Figure 5a presents the total time for analysis of all 11 DaCapo benchmarks for brevity. For the Doop static program analysis, we observe that the performance of our B-tree with a single thread is approximately 1.5× faster than the nearest reference data structure, the Google B-tree. In parallel workloads, our B-tree implementation provides better scalability than the reference data structures with global locks. Compared to the concurrent TBB hashset equipped Datalog engine, the optimistic B-tree based



(a) Context sensitive var-points-to analysis (insertion heavy).

(b) Security vulnerability analysis (read heavy).

Figure 5. Comparison of different data structures for two real-world applications relying on large scale Datalog queries.

counterpart maintains similar scalability, however, performance is approximately 4× better than the TBB hashset for all numbers of threads. Additionally, the usage of operation hints improved performance by up to 10%.

With the security vulnerability analysis, we observe that our B-tree performs approximately 2× better than the TBB hashset. Interestingly, with this workload, even the global locked data structures demonstrate some scalability, since it is read-heavy rather than write-heavy. We also note that this security analysis contains more relations with fewer tuples (1.2e7 out of the 1.6e7 produced tuples were concentrated in a single relation), and as a result, the hash-based data structures perform better compared with the Doop program analysis. Additionally, for this analysis, our B-tree with operation hints performs almost 1.5× as well as our B-tree without hints, indicating that ordered queries play an important part in this workload.

With regards to operation hints, collected statistics show that for the single-threaded Doop analysis, up to 54% of operations resulted in an operation hint hit, and up to 52% hits for the 16 thread case. With the security vulnerability analysis, these rates reach 77% for the single-threaded case and 76% for the 16 thread case. This further suggests that the security vulnerability analysis heavily involves ordered data, and therefore demonstrates a larger performance improvement with the addition of operation hints.

In both cases, the utilization of parallel resources provides performance improvements over the best sequential version. In the Doop case, speedup of 1.9× can be obtained, and in the vulnerability case, a speedup of 8.4× compared to our B-tree run with 1 thread. These improvements are on top of the 1.3-3× sequential improvement compared to other reference data structures.

4.4 Comparison with Concurrent Tree Data Structures

Our B-tree data structure is not the only concurrent tree data structures. Alternatives include *PALM tree* [48], *Masstree* [36], and *B-slack tree* [12]. However, there are a number of reasons why such data structures are not suitable for Datalog evaluation.

PALM tree [48] is a concurrent lock-free B+ tree implementation. It uses an internal synchronization strategy, where elements to be inserted are added to an internal queue and processed concurrently by the data structure itself. Moreover, PALM tree uses AVX instructions, and therefore only supports single integer keys and not tuples as required for Datalog processing.

Masstree [36] is built as a client/server architecture, designed for use-cases requiring persistence across reboots. Hence, it is not optimized for use in an in-memory Datalog engine. Moreover, Masstree only supports strings as keys and thus would require a significant redesign of Soufflé .

B-slack tree [12] constitutes a variation of B-tree that constrains the overall fill grade of all child nodes. As a result, B-slack trees exhibit better worst-case space complexity than conventional B-trees. B-slack trees weakens structural constraints and decouples the insertion from the rebalancing steps making the locking of B-trees more local. However, B-slack trees do not specify the locking scheme. Absolute insertion performance or parallel scalability have not been investigated in [12].

In addition to the practical limitations of these alternative data structures, we compared their performance to our B-tree. Due to the limitations of the data structures for our Datalog workload, we provide an additional set of benchmarks inserting and reading a set of 10,000,000 fixed-size 32 bit integers, in both sequential and random order. The results are in Table 3. Note that Masstree was evaluated using the included benchmark utility, as the client/server architecture could not be integrated with our microbenchmark suite. The results demonstrate that our B-tree exhibits up to 3× better sequential insertion performance, and up to 1.5× better random insertion performance, compared to the next best data structure (Masstree). Moreover, our data structure exhibited up to 8× higher throughput than B-slack tree, and 200× higher throughput than PALM tree.

Table 3. Throughput inserting 32 bit integers

	Throughput (ordered/random) [10^6 elements/second]			
Threads	B-tree	PALM tree	Masstree	B-slack
1	17.5/2.91	0.38/0.10	5.99/1.90	2.73/1.09
2	45.61/5.23	0.44/0.13	10.11/3.37	3.83/1.94
4	89.64/9.71	0.47/0.16	19.52/6.17	7.32/3.15
8	97.19/16.97	0.49/0.20	36.38/11.41	11.29/4.84

The repo http://github.com/souffle-lang/ppopp19 contains the experiments without some industrial benchmarks whose source-code could not be disclosed.

5 Related Work

Data Structures for Datalog. Previous Datalog implementations have focused on various kinds of alternative data structures including binary decision diagrams [52], Hashsets e.g., [25, 47] and B-trees e.g., [9, 28]. In our experience, the use of B-trees (as implemented in PA-Datalog / Logicblox ver. 3 and Soufflé [16]) have shown to be the most scalable for large ruleset/dataset benchmarks [28]. In particular, the work presented in [29] demonstrates the advantages of B-trees in novel indexing schemes specifically design for use cases with complex rulesets and large datasets.

Parallel Datalog Engines. There has been a multitude of parallelization efforts of Datalog in the past [14, 19, 27, 46, 49, 53, 54] mainly focusing on rewriting techniques and top-down evaluations. Recently a number of state-of-the-art engines have employed fine-grain parallelism to bottom-up evaluation schemes. The work in [55] uses an in-memory parallel evaluation of Datalog programs on shared-memory multi-core machines. Datalog-MC hash-partitions tables and executes the partitions on cores of a shared-memory multi-core system using a variant of hash-join. To parallel evaluate Datalog, Datalog rules are represented as and-or trees that are compiled to Java. Logicblox version 4, uses persistent functional data structures that avoid the need for synchronization by virtue of their immutability, where insertions efficiently replicate state via the persistent data structure. A particular performance focused approach has been proposed by Martınez-Angeles et al. who implemented a Datalog engine running on GPUs [37]. Their basic data structure is an array of tuples, allowing for duplicates. Thus, after every relational operation, explicit duplicate elimination is performed – which for some cases vastly dominates execution time. Also, the potentially high number of duplicates occurring in temporary results quickly exceeded the memory budget on GPUs. The applicability of this approach has only been demonstrated for small Datalog queries. We point the reader to [3, 44] for performace comparissons between engines on large ruleset/dataset benchmarrks.

Concurrent Tree Data Structures. B-trees themselves are among the most successful data structures for indexed data. Lots of research effort has been addressing locking techniques [21]. However, most of those focus on the secondary storage use case [31]. For in-memory usage, a modified B-tree known as B-link tree managed to eliminate the need for read locks [33]. Unfortunately, we have not been able to obtain an implementation for comparison. An alternative approach has obtained good results by utilizing hardware transactional memory features available on some recent Intel architectures for synchronizing B-tree insert operations [30]. Their evaluation shows comparable parallel scalability to our optimistic approach. However, special hardware support is required for those and multi-socket systems have not been evaluated. Concurrency has been investigated in several tree-like data structures for general use. For example, the data structure in [11] is similar in spirit to our work with an optimistic concurrency which allows invisible readers. The approach in [17] maintains interval information to determine the placement of data. The data structure in [26] is based on single-word reads, writes, and compare-and-swap where its algorithm operations only contend if concurrent updates affect the same nodes. Other concurrent tree-like structures have been presented in [4, 5, 8, 13, 15, 34, 39–41]. The involved design decisions are orthogonal to the locking scheme presented by our work. Realizing a C++ version of the B-slack tree utilizing our seq-lock-based synchronization scheme has the potential of yielding a highly scalable concurrent implementation. A recent approach [12] constrains the overall fill grade of all child nodes providing better worst-case space complexity than conventional B-trees and weakens the structural properties of the tree to improve the locality of modifications. However, the design does not specify a node locking scheme for a concurrent implementation.

6 Conclusion

In this paper, we presented the design of a specialized concurrent B-tree that is employed by the Soufflé Datalog engine. Our relation data structure is a modified concurrent in-memory variant of the widely utilized B-tree. Its novel optimistic locking scheme along with novel operation hints enables concurrent insertions to scale well beyond socket boundaries, achieving up to 59× faster insertion times, and up to 3× faster Datalog evaluation than industry standard, state-of-the-art concurrent set implementations.

Acknowledgments

We thank Cristina Cifuentes and Oracle Labs in Brisbane where early versions of this work was done, Byron Cook and the ARG team at Amazon Web Services. This research was supported partially by the Australian Government through the ARC Discovery Project funding scheme (DP180104030) and the Austrian Research Promotion Agency (FFG) under contract no 868018.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu (Eds.). 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc.

[2] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. 2015. Combining Type-analysis with Points-to Analysis for Analyzing Java Library Source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2015)*. ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/2771284.2771287

[3] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting Doop to Soufflé: A Tale of Inter-engine Portability for Datalog-based Analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017)*. ACM, New York, NY, USA, 25–30. https://doi.org/10.1145/3088515.3088522

[4] Maya Arbel and Hagit Attiya. 2014. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14)*. ACM, New York, NY, USA, 196–205. https://doi.org/10.1145/2611462.2611471

[5] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A Key-Value Map for Scalable Real-Time Analytics. *SIGPLAN Not.* 52, 8 (Jan. 2017), 357–369. https://doi.org/10.1145/3155284.3018761

[6] R. Bayer and E. M. Mccreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Inf.* 1, 3 (Sept. 1972), 173–189. https://doi.org/10.1007/BF00288683

[7] Hans-J Boehm. 2012. Can seqlocks get along with programming language memory models?. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. ACM, 12–20.

[8] Anastasia Braginsky and Erez Petrank. 2012. A Lock-free B+Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. ACM, New York, NY, USA, 58–67. https://doi.org/10.1145/2312005.2312016

[9] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception analysis and points-to analysis: better together. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1572272.1572274

[10] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. *SIGPLAN Not.* 44, 10 (2009), 243–262. https://doi.org/10.1145/1639949.1640108

[11] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. *SIGPLAN Not.* 45, 5 (Jan. 2010), 257–268. https://doi.org/10.1145/1837853.1693488

[12] Trevor Brown. 2014. B-slack Trees: Space Efficient B-Trees. In *Algorithm Theory – SWAT 2014*, R. Ravi and Inge Li Gørtz (Eds.). Springer International Publishing, Cham, 122–133.

[13] Trevor Brown and Joanna Helga. 2011. Non-blocking K-ary Search Trees. In *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS'11)*. Springer-Verlag, Berlin, Heidelberg, 207–221. https://doi.org/10.1007/978-3-642-25873-2_15

[14] S. Cohen and O. Wolfson. 1989. Why a Single Parallelization Strategy is Not Enough in Knowledge Bases. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '89)*. ACM, New York, NY, USA, 200–216. https://doi.org/10.1145/73721.73742

[15] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2012. A Speculation-friendly Binary Search Tree. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/2145816.2145837

[16] The Souffle Developers. [n. d.]. Souffle – A Datalog Engine. http://www.github.com/souffle-lang/souffle. ([n. d.]). Accessed: 2019-01-05.

[17] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical Concurrent Binary Search Trees via Logical Ordering. *SIGPLAN Not.* 49, 8 (Feb. 2014), 343–356. https://doi.org/10.1145/2692916.2555269

[18] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 469–483.

[19] Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. 1990. A Framework for the Parallel Processing of Datalog Queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. ACM, New York, NY, USA, 143–152. https://doi.org/10.1145/93597.98724

[20] Google. [n. d.]. B-Tree Containers from Google. https://isocpp.org/blog/2013/02/b-tree-containers-from-google. ([n. d.]). Accessed: 2017-02-14.

[21] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Transactions on Database Systems (TODS)* 35, 3 (2010), 16.

[22] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Foundations and Trends in Databases* 5, 2 (2013), 105–195.

[23] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An Efficient Tunable Selective Points-to Analysis for Large Codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017)*. ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/3088515.3088519

[24] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[25] Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. 2011. μZ – An Efficient Engine for Fixed Points with Constraints. In *Proceedings of the International Conference on Computer Aided Verification*, Vol. LNCS 6806. Springer, 457–462.

[26] Shane V. Howley and Jeremy Jones. 2012. A Non-blocking Internal Binary Search Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. ACM, New York, NY, USA, 161–171. https://doi.org/10.1145/2312005.2312036

[27] G. Hulin. 1989. Parallel Processing of Recursive Queries in Distributed Architectures. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB '89)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 87–96.

[28] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *International Conference on Computer Aided Verification*.

[29] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2017. Optimal On The Fly Index Selection in Polynomial Time. *CoRR* abs/1709.03685 (2017). arXiv:1709.03685 http://arxiv.org/abs/1709.03685

[30] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. 2014. Improving in-memory database index performance with Intel® Transactional Synchronization Extensions. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 476–487.

[31] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[32] Christoph Lameter. 2005. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, Vol. 2005.

[33] Philip L. Lehman and s. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 650–670. https://doi.org/10.1145/319628.319663

[34] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[35] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, USA, 18–18.

[36] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[37] Carlos Alberto Martınez-Angeles, Inês Dutra, Vıtor Santos Costa, and Jorge Buenabad-Chávez. 2014. A datalog engine for gpus. *Declarative Programming and Knowledge Management* (2014), 152–168.

[38] Daniel A Menascé and Tatuo Nakanishi. 1982. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information systems* 7, 1 (1982), 13–27.

[39] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/2555243.2555256

[40] Rotem Oshman and Nir Shavit. 2013. The SkipTrie: Low-depth Concurrent Search Without Rebalancing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2484239.2484270

[41] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. *SIGPLAN Not.* 47, 8 (Feb. 2012), 151–160. https://doi.org/10.1145/2370036.2145836

[42] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.".

[43] Thomas W. Reps. 1995. *Demand Interprocedural Program Analysis Using Logic Databases.* Springer US, Boston, MA, 163–196. https://doi.org/10.1007/978-1-4615-2207-2_8

[44] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226

[45] Bernhard Scholz, Kostyantyn Vorobyov, Padmanabhan Krishnan, and Till Westmann. 2015. A Datalog Source-to-Source Translator for Static Program Analysis: An Experience Report. In *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*. IEEE Computer Society, 28–37.

[46] Jürgen Seib and Georg Lausen. 1991. Parallelizing Datalog Programs by Generalized Pivoting. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '91)*. ACM, New York, NY, USA, 241–251. https://doi.org/10.1145/113413.113435

[47] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1906–1917. https://doi.org/10.14778/2556549.2556572

[48] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB* 4 (08 2011), 795–806.

[49] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. 2012. Optimizing Large-scale Semi-Naïve Datalog Evaluation in Hadoop. In *Proceedings of the Second International Conference on Datalog in Academia and Industry (Datalog 2.0'12)*. Springer-Verlag, Berlin, Heidelberg, 165–176. https://doi.org/10.1007/978-3-642-32925-8_17

[50] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. 2013. Graph Queries in a Next-generation Datalog System. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1258–1261. https://doi.org/10.14778/2536274.2536290

[51] Julian Shun and Guy E. Blelloch. 2014. Phase-concurrent Hash Tables for Determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '14)*. ACM, New York, NY, USA, 96–107. https://doi.org/10.1145/2612669.2612687

[52] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. 2005. Using Datalog with binary decision diagrams for program analysis. In *APLAS*. 97–118.

[53] Ouri Wolfson and Aya Ozeri. 1990. A New Paradigm for Parallel and Distributed Rule-processing. *SIGMOD Rec.* 19, 2 (May 1990), 133–142. https://doi.org/10.1145/93605.98723

[54] Ouri Wolfson and Avi Silberschatz. 1988. Distributed Processing of Logic Programs. *SIGMOD Rec.* 17, 3 (June 1988), 329–336. https://doi.org/10.1145/971701.50242

[55] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. 2017. Scaling up the performance of more powerful Datalog systems on multicore machines. *VLDB J.* 26, 2 (2017), 229–248. https://doi.org/10.1007/s00778-016-0448-z