Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Specializing Parallel Data Structures for Datalog

Herbert Jordan[1] | **Pavle Subotić**[2] | **David Zhao**[3] | **Bernhard Scholz**[3]

[1]Universität Innsbruck, Austria
[2]Amazon, United Kingdom
[3]The University of Sydney, Australia

Abstract

We see a resurgence of Datalog in a variety of applications including program analysis, networking, data integration, cloud computing, and security. The large scale and complexity of these applications need the efficient management of data in relations. Hence, Datalog implementations require new data structures for managing relations that (1) are parallel, (2) are highly specialized for Datalog evaluation, and (3) can accommodate different workloads depending on the applications concerning memory consumption and computational efficiency.

In this article, we present a data structure framework for relations that is specialized for shared-memory parallel Datalog implementations such as the SOUFFLÉ Datalog compiler. The data structure framework permits a portfolio of different data structures depending on the workload. We also introduce two concrete parallel data structures for relations, designed for various workloads. Our benchmarks demonstrate a speed-up of up to 6× by using a portfolio of data structures, compared to using a B-tree alone, showing the advantage of our data structure framework.

KEYWORDS:
Parallel Data Structure, B-Tree, Trie, Datalog

1 | INTRODUCTION

Datalog is a popular domain specific language (DSL) for specifying a range of important problems including static program analysis [1], network analysis [2], graph databases [3], block-chain analysis [4], among many others. Recently, Datalog has been shown to scale to industrial sized projects by use of effective shared-memory based parallelization techniques, and in particular, concurrent data structures [5] that provide the underlying mechanism to perform Datalog evaluation.

A Datalog program consists of a set of rules 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, previously computed relations. Datalog engines organize and evaluate the rules to compute output tuples. Efficient state-of-the-art Datalog evaluation strategies, known as *semi-naïve evaluation* strategies [6], 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 [6,7,8,9] is a data structure for representing relations that facilitates relational algebra operations including join, selection, and projection operations.

The data structures used in Datalog engines vary, including (in-memory) B-trees, tries, ordinary arrays, and BDDs. The use of specialized data structures has been argued in niche applications. For example, *bddbddb* [7] employs binary decision diagrams to store relations in points-to based static analysis. The *SociaLite* [3] engine employs hash-based sets for graph database problems. The *NoD*/μZ [10] engine employs a data structure optimized for Network analysis [2]. These engines argue that their respective data structures achieve the best average case performance for their respective domains. However, despite these strong performance results in certain situations, a single data structure ultimately fails to meet peak performance across diverse sets of benchmarks.

In this paper, we argue for the use of specialized data structures in Datalog engines. We discuss the design of efficient data structures that are specialized for an aspect of Datalog, e.g., evaluation algorithm, ruleset characteristics, and dataset characteristics. For example, since Datalog

conceptually operates on sets of tuples, and assumes monotonicity during the evaluation, a given tuple will only be inserted once in a given relation. Furthermore, the semi-naïve rule evaluation is performed in two phases [5], namely, a read phase and a write phase [6]. This information can be exploited in parallel execution to avoid unnecessary synchronization as writes cannot occur interleaved with these reads.

The approach above is reflected in the design of our Datalog engine Soufflé. Unlike other engines, Soufflé does not depend on a single data structure and does not attempt to provide performance by finding and employing a *universally best data structure*. Instead, Soufflé provides a data structure framework that allows the use of a *portfolio of data structures* during evaluation. In this way, the most effective data structure can be selected by users for a given relation, or automatically inferred based on various ruleset/dataset characteristics. Apart from improved overall performance, Soufflé provides researchers a framework to quickly and seamlessly prototype new experimental Datalog-Enabled Relational (DER) data structures for relations without the need for developing yet another Datalog engine. The framework has resulted in the development and integration of several novel specialized data structures and has allowed Soufflé to exhibit strong performance across a wide range of benchmarks including industrial grade benchmarks sourced from program analysis, network reachability and block-chain analysis use cases.

Within the Soufflé data structure framework, we have designed two pivotal, novel data structures that highlight the need for a portfolio of specialized data structures in Datalog engines.

The first data structure is a specialized concurrent data structure for Datalog that is based on in-memory B-trees and performs well for non-dense data. Our data structure is fundamental for the efficient parallel execution of relational algebra operations on contemporary shared-memory multicore machines. Unlike B-Tree implementations in engines such as Logicblox v3 [11], our B-tree is specialized to work with the semi-naïve Datalog evaluation and thus exploits the properties of the two phases of semi-naïve, and thus provides an efficient locking mechanism, as well as a 'hint' mechanism to reduce the overheads of tree traversal.

The second data structure is a specialized concurrent data structure for Soufflé called the *Brie*, which has been designed for *high-density* relations with a *large data volume* only. Datalog relations with these characteristics show up in applications such as points-to program analysis. The new data structure adopts design ideas from both B-tree [12] and trie data structures [13,14]. However, the BRIE has several key advantages over both B-trees and tries for dense data only. Compared to B-trees, the BRIE achieves a much higher *coding density* per element, by taking advantage of the similarities of maintained keys and thus reducing the amount of duplicated data stored. Compared to tries, the BRIE achieves superior data locality, thus improving cache coherence for dense input data sets.

We have implemented both data structures in the Soufflé data structure framework and have evaluated their performance on parallel micro-benchmarks and industrial Datalog benchmarks. We examine the cases where each data structure is effective and their combined usage to achieve high performance.

The contributions of this article are summarized as follows[1]:

- We present the properties of a generic data structure specification for Datalog relations. We enable the development of DERs via an interface so that data structures for applications can be adjusted depending on the workload. The data structure interface enables research in novel data structures for Datalog without the need to implement a whole Datalog engine.
- We present an in-memory B-tree that utilizes the data structure framework. The in-memory B-tree has been specialized for Datalog, exploiting the two phases (read/write) of semi-naïve evaluation. It has a low synchronization overhead and supports a wide range of applications that have a sparse to moderate data density.
- We present a BRIE data structure for data with high volume and high-density characteristics. The BRIE data structure uses some ideas of tries, with compressed representations of sub-trees, and path compression, while maintaining some of the cache-friendliness of B-trees.

We also conduct experiments with various industrial and research benchmarks showing the effectiveness of the novel data structures complying to the Soufflé's relational interfaces.

The article is structured as follows: In Section 2, we discuss the need of specialized Datalog relations for parallelized Datalog evaluation. In Section 3, we discuss the properties of a generic data structure interface for Datalog evaluation. In Section 4, we present the specialized in-memory B-tree data structure, and in Section 5, we present the Brie data structure. In Section 6, we evaluate both data structures. In Section 7, we survey related work, and in Section 8, we draw our conclusions.

[1]This journal article is based on previous conference articles [5,15] written by the co-authors.

```
using Tuple = array<size_t,2>;
using Relation = set<Tuple>;

Relation evaluate(const Relation &connection) {
 Relation tc = connection, deltaTc = connection;
 while(!deltaTc.empty()){
  Relation newTc;
  #pragma omp parallel for
  for (const auto &t1: deltaTc){
   auto l = connection.lower_bound({t1[1],0});
   auto u = connection.lower_bound({t1[1]+1,0});
   for (auto it=l; it!=u; ++it){
    auto& t2 = *it;
    Tuple t3({t1[0],t2[1]});
    if (tc.find(t3) == tc.end())
      newTc.insert(t3);
   }
  }
  tc.insert(newTc.begin(), newTc.end());
  deltaTc.swap(newTc);
 }
 return tc;
}
```

FIGURE 1 Synthesised C++/STL code for Path Example

2 | PARALLEL DATALOG EVALUATION

Datalog is a logic programming language, where logical specifications define the computation of input and output relations. Given a set of input relations, the output relations are computed by logic rules. This computation may also involve many intermediate relations in the case of complex Datalog programs.

For example, let *connection* be a binary input relation, representing a set of edges in a graph. The two logical rules:

$$tc(X, Y) \text{ :- } connection(X, Y). \tag{1}$$

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

implicitly define the content of the output relation *tc* computing the transitive closure of the *connection* relation. The first rule states that if there is a direct connection from node X to node Y, then (X, Y) is included in the transitive closure. The second rule is a recursive rule, adding transitive paths inductively. If there is a transitive path from node X to node Y, and there is a direct connection from Y to Z, then there is a transitive path from X to Z. While this is a simple example of Datalog, real-world use cases may comprise hundreds of relations, connected via hundreds of potentially mutually recursive rules.

The standard approach for Datalog evaluation is a bottom-up approach. A naïve bottom-up method proceeds in multiple iterations, each of which evaluates all rules in the program, and generate new tuples based on the tuples in the previous iteration. This evaluation proceeds until no new facts can be generated, where we say that a *fixpoint* has been reached.

However, the naïve evaluation will re-compute tuples that have already been generated in prior iterations, thus leading to wasted computational effort. Therefore, semi-naïve evaluation is an optimization which assumes that each iteration should depend on tuples generated in the immediate previous iteration. To this end, semi-naïve keeps track of auxiliary relations: a *new* relation, and a *delta* relation for each original relation, which keep track of new tuples generated in the current iteration. For our example, the recursive rule of the program would be modified as follows:

$$tc^0 := connection$$

$$\Delta_{tc}^0 := connection$$

$$\text{new}_{tc}^{i+1}(X, Z) := \Delta_{tc}^i(X, Y), connection(Y, Z).$$

$$\Delta_{tc}^{i+1} := \text{new}_{tc}^{i+1} \setminus tc^i$$

$$tc^{i+1} := tc^i \cup \Delta_{tc}^{i+1}$$

In this example, the new knowledge in iteration $i + 1$ depends directly on tuples computed in iteration i. Once the new knowledge is computed, two set operations are required to merge the new knowledge into the existing relation *tc*.

The Soufflé [8] Datalog engine compiles the transitive closure example to parallel C++ code as listed in Figure 1. The code outlines the anatomy of the underlying least fixed-point calculations of the semi-naïve evaluation [6] using STL's set data structures for relations. The synthesized function

`evaluate()` has the input relation *connection* as an argument and computes the output relation *tc*. The non-recursive rule (1) causes the initialization of relation *tc* with the tuples of relation *connection* (line 5).

The fixed-point calculation for the recursive rule (2) is performed in the while-loop from line 6 to line 21. In each iteration of the while-loop, new tuples are produced for relation *tc*. Thereby, the auxiliary relations *newTc* and *deltaTc* 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 *tc* will be the transitive closure of *connection*.

The semi-naïve algorithm thus unrolls recursive rules into a sequence of non-recursive rules. In the outlined implementation, these non-recursive rules are computed using a *nested-loop join*, i.e., a nested loop structure that filters tuples from relations and results in a potential insertion in the innermost loop. The filtering mechanism could be implemented by traversing over all tuples in a relation and checking an equality predicate to determine if a tuple should be included in an intermediate result. However, this full relation scans would be computationally inefficient. Instead, we maintain tuples lexicographically ordered and exploit this property to efficiently narrow down iteration ranges to tuples of matching attributes. This improve the search from a linear to logarithmic time complexity.

For example, the nested loop-join in lines 9-12 finds matching connections $t2 \equiv (Y, Z)$ for each newly discovered transitive connection $t1 \equiv (X, Y)$ in relation *deltaTc*. The adjacent connection $t2$ 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 $(u, v) < (u', v')$ if and only if $u < u' \vee (u = u' \wedge v < v')$. 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 connection set. With the lexicographical order, STL's red-black tree efficiently performs the range traversal of all matching/adjacent connections and no scan over the whole relation *connection* is necessary. If a matching connection pair is found, a new transitive connection $t3 \equiv (X, Z)$ (cf. line 14) is constructed. If $t3$ is not yet known (line 15), it will be added to the set of newly discovered transitive connections (line 16). Finally, newly discovered entries are merged into the tc relation and promoted to be the *deltaPath* set for the next iteration (line 19 and line 20).

Except the insert operation in line 16, all operations within the nested for-loop between line 9 and line 18 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 9) by providing a synchronized insert operation. However, STL's set implementation is not intended for concurrent use, and thus does not support synchronized insertion.

As outlined, for optimum performance of Datalog evaluation, indexed data structures are utilized, allowing the fast insertion and retrieval of tuples. However, due to the complexity of data access patterns in real-world Datalog programs, multiple indices may be required for each relation. The construction of these indices is covered in [16], where a polynomial time algorithm is described for the selection of optimal indices. Each index then uses a set data structure to store the underlying tuples in the order described by the index, and the B-tree and BRIE data structures presented in this paper are two candidates suitable for this task.

Datalog's rule evaluation has particular use patterns of the underlying set data structure. These use patterns can be exploited for better parallel performance. For example, the usage of a set data structure has two phases (cf. [17]): (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 9 and line 12 read pairs from the relations *deltaTc* and *connection* but do not insert new pairs into *deltaTc/connection*. Newly found pairs are inserted in relation *newTc* but no read operations on *newTc* exist in this loop nest. The semi-naïve evaluation guarantees these 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, and only the write operation requires synchronization. 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 way as well. Hence, the set data structure can exploit a tuple order to minimize the search overheads in the data structure.

3 | ABSTRACT RELATION SPECIFICATION

In this section, we describe the properties required for DER data structures that can be effectively employed for parallelized Datalog evaluation. The DER interface for data-structures facilitates the isolation of query processing logic from the underlying concurrent data structures. In general, the workloads differ significantly depending on the application and, hence, a single data structure cannot meet the needs of all applications. Therefore, we introduce a data structure framework that facilitates the design of new data structures in parallel Datalog engines. Our framework dictates properties for data structures to store Datalog relations so that they can seamlessly be deployed and swapped in a parallel Datalog engine. The resulting DER contract provides the foundation for Datalog engines built on top. It comprises basic API interface specifications as well as restrictions on the tuple types to be maintained and concurrent interaction patterns to be supported.

3.1 | Functional Requirements

A DER data structure stores a set of same-size tuples $\{t_1, \ldots, t_n\}$. The functional requirement of a set container is that no two tuples are the same in the set, i.e., $\forall i \neq j : t_i \neq t_j$. We assume that a tuple is a fixed-length vector of numbers. Note that tuples may have elements that describe strings, symbols or even more complex elements such as record [8]. However, we assume that there exists a string/symbol table that translates symbols or strings to numbers. This translation mechanism permits a compact and fixed-length representation of tuple elements even for more complex objects.

Any data structure that uses our framework implements an abstract data-type interface for a set container storing vectors of numbers. A DER data structure implements a set of operation that conforms to the operations that are necessary for executing a parallelized version of semi-näive evaluation [8,18]. A DER must implement the following operations:

- *insert*(t) inserts a fixed sized n-ary integer tuple t into a set of n-ary tuples concurrently, ignoring duplicates,
- *begin*$()$ and *end*$()$ provides iterators to traverse the set concurrently,
- *lower_bound*(a) and *upper_bound*(a) provides iterators to lower and upper bound values of a stored in the set, according to a set instance specific order,
- *find*(t) obtains an iterator to the tuple t in the set, if present, and
- *empty*$()$ determines whether the set is empty.

This interface mimics the STL set-container interface so that existing data-structure can be adapted quickly to a DER data-structure. The interface operations cover the functionality of storing tuples of fixed-length, set-enforcement, iterators for whole set traversals, range queries, membership/existence tests (find), and correct parallel execution. Beside the set-container functionality, a DER data structure requires a high-performance implementation of these operations. The DER operations are to be performed with little or no synchronization overhead on shared-memory multi-core computers. This can only be achieved by specializing a generic set-container for this purpose. In the following, we discuss the necessary specializations.

3.2 | Specializing for Parallel Tuple Insertion

In this section, we describe how a data structure can be specialized for tuple insertion, i.e., the *insert*(t) operation. Note that a tuple insertion already combines two primitive operations of DER: (1) the existence check whether the tuple already exists in the relation, and (2) the actual insertion of the tuple. The existence check is necessary for enforcing that the container remains a set, i.e., without the existence check a tuple may be introduced more than once into the set.

The combination of the two primitive operations imposes already a synchronization challenge for a DER data structure, since when a large number of multiple insertions are performed in parallel, a race condition may be imposed between the parallel insertions. One of the consequences of the data race will be the violation of the set property if the insertions are not synchronized properly. To obtain satisfactory performance and scalability, synchronization techniques with low synchronization overheads are essential.

Due to the existence check for set enforcement, classical linear data structures such as linked lists and arrays are ill-suited for the purpose of DER. The main reason is that the tuple existence check would exhibit a worst-case runtime complexity of $O(n)$ where n is the number of tuples. Variations of balanced search trees such as B-Trees [18] perform the existence in $O(\log n)$ and the variations of tries [15] perform the existence check in $O(W)$ where W is the (constant) bit length of a tuple.

Another observation in the semi-naïve evaluation is that a DER data structure has *two distinct phases*. In the first phase, the data structure is only read and hence no synchronization is required between multiple threads accessing the data structure simultaneously. In the second phase, parallel multiple insertions occur without reading/traversing the tuples of a relation. Between the two phases, the evaluation sets in place barriers so that no inter-leavings between tuple insertion operations and read-operations can happen. Naturally, these two phases can be exploited by a DER data structure - specializing the data structure purely for the purpose of Datalog evaluation.

Specifically, we have four auxiliary relations capturing the new, current, previous and delta knowledge of a recursive relation [7]. For evaluating a rule for a recursive relation, the auxiliary relations current, previous and delta knowledge are only read but not written to, i.e., they are used to iterate tuples over relations / and existence checks only. The auxiliary relation for new knowledge is the only one which receives tuples in a loop-nest and is not read from. For example, in our motivating example in Figure 1, the relation *newTc* is only written to inside the loop (cf. line 9 to line 18) but never read inside the loop-nest that evaluates new tuples for the rule *tc*(X, Z) :- *tc*(X, Y), *connection*(Y, Z).

As a consequence, we can specialize a DER data structure for synchronization purposes, i.e., we do not require any synchronization between read and write operations. Another consequence is that the order of the write operations is not important as long as all tuples that are inserted in

the set will materialize after the write phase. Other optimizations that are based on these two distinct phases are also possible. They are specific to a concrete DER data structure, as outlined in Section 4 and 5. For example, we exploit this assumption to propose a locking mechanism that avoids the overhead of a general purpose B-Tree for insertion.

3.3 | Specializing for Nested-Loop Joins

In this section, we describe how the order of attributes that tuples are stored in e.g., a B-Tree can be specialized in the data structure to facilitate low overhead range lookups, i.e., *lower_bound*(a) and *upper_bound*(a) operators. These range lookups are the building blocks of evaluating rules.

To achieve peak performance, modern Datalog engines generally assume an execution model that keeps all relations in memory [16]. In this sense, every relation is indexed. A Datalog engine requires that for all operations on a relation, the operations can be performed on an index to achieve improved performance. Hence, several replications of a relation are stored in memory that have different attribute orders. This *cluster of indexes* of a relation facilitates fast searches filtering tuples of relations. Such filtering operations are called *primitive searches*.

A primitive search can be naively achieved by conducting a linear scan over all tuples in a relation and checking the search predicate against each tuple. However, the time complexity of linear scans with n tuples is $\mathcal{O}(\mathsf{n})$, which is too costly for large relations considering that each primitive search is invoked repeatedly many times.

To enable faster searches on a relation, we require that DER data structures have an order among tuples. Given an ordered set of tuples, tuple lookups can be performed efficiently depending on the specifics of the data structure. For example, in some variations of a balanced-search tree, tuples can be found in logarithmic time rather than in linear time.

Since a tuple may have several elements, an order of tuples is imposed by element-wise comparison using a sequence over all attributes of the relation; this comparison is known as a *lexicographical order*. Let us denote an attribute sequence by $\ell = \mathsf{x}_1 \prec \mathsf{x}_2 \prec \cdots \prec \mathsf{x}_\mathsf{m}$ where $\prec$ denotes a chaining of elements to form a sequence.

Then, given ℓ that is formed by all attributes of a relation, a lexicographical order $\sqsubseteq_\ell \mathcal{D} \times \mathcal{D}$ is a total order (i.e., reflexive, asymmetric, transitive) defined over the domain $\mathcal{D}$ of the relation with respect to ℓ. For two tuples $\mathsf{a}, \mathsf{b} \in \mathcal{D}$, when $(\mathsf{a}, \mathsf{b}) \in \sqsubseteq_\ell \mathcal{D} \times \mathcal{D}$, we write $\mathsf{a} \sqsubseteq_\ell \mathsf{b}$ and we say that a is smaller than b with respect to ℓ. Note that $\mathsf{a} \sqsubseteq_\ell \mathsf{a}$, and for any two different tuples $\mathsf{a}, \mathsf{b} \in \mathcal{D}$, we either have $\mathsf{a} \sqsubseteq_\ell \mathsf{b}$ or $\mathsf{b} \sqsubseteq_\ell \mathsf{a}$ but not both.

A DER data structure has a single order that can support several primitive searches. However, not all primitive searches can be covered by a single index. Thus, the relation is replicated with different orderings. An evaluation heuristic such as literal orderings in rules impacts which data orderings are most memory efficient in the replicated relations. Note that different attribute sequences usually result in different lexicographical orders, and thus different indexes. That is, for tuples $\mathsf{a}, \mathsf{b} \in \mathcal{D}$ and attribute sequences ℓ and ℓ', it is possible that $\mathsf{a} \sqsubseteq_\ell \mathsf{b}$ and $\mathsf{b} \sqsubseteq_{\ell'} \mathsf{a}$.

Both data structures presented in Section 4 and 5 support arbitrary attribute orderings, and thus allow for modern Datalog engines to improve tuple search time complexity while reducing memory overhead using index covering techniques [16].

3.4 | Specializing for Sequences of Operations

Another observation is that a DER data structure may be traversed consecutively by multiple threads to perform a single operation each. Although some traversals are log-linear time, the traversal will become expensive for a large number of operations that are in the billions. To amortize the cost of traversal, DER data structures may provide a hint mechanism that caches previous traversals. The generic hint mechanism uses state associated with a DER storing summaries of previous traversals. In case of a new traversal, the state is revisited first and checked whether the old traversals can be reused. If this is the case, the information from the previous operations reduces the execution time of the current operation. Especially, when accessed data elements are approximately sorted the hint mechanism becomes more effective for ordered data structures such as B-Trees. The hint mechanism can be used for a wide range of operations such as tuple insertion, range queries, and existence checks.

3.4.1 | Interface Specifications

To describe interfaces of abstract data types like our DER, a type based notation can be utilized to provide a more formal description. Let D be the set of all instances of a DER, and the set E denote the set of all its potential elements. Then

$$\mathsf{contains} : (D, E) \to \mathbb{B}$$

and

$$\mathsf{insert} : (D, E) \to D$$

are type specifications for the required operators. The *contains* function takes a pair of a data structure instance $\mathsf{d} \in \mathsf{D}$ and and an element $\mathsf{e} \in \mathsf{E}$ and determines whether e is present in d by producing a boolean value $\mathsf{contains}(\mathsf{d}, \mathsf{e}) \in \{\mathsf{true}, \mathsf{false}\} = \mathbb{B}$. The *insert* function, on the other hand,

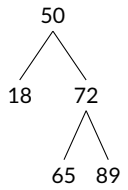


FIGURE 2 Binary search tree example.

accepts a data structure instance $\mathsf{d} \in \mathsf{D}$ and an element $\mathsf{e} \in \mathsf{E}$ and computes a potentially new instance $\mathsf{insert}(\mathsf{d}, \mathsf{e}) = \mathsf{d_r} \in \mathsf{D}$ such that the predicate

$$\mathsf{contains}(\mathsf{insert}(d, e), e)$$

holds. Thus, the data structure instance $\mathsf{d_r}$ obtained by processing the insertion function contains the inserted element e.

While this formalism is based on a functional perspective of the data structure interface, it serves as a reasonable abstraction of operations on data structures maintained within random access machines (RAM) underlying imperative or object-oriented programming languages. Modeling operations following such paradigms would require a more elaborate set of definitions regarding memory state and side effects of operations. By utilizing the functional point of view, such details, not required for our purposes, can be omitted.

Nevertheless, it is important to see that in non-functional context implementations of the *insert* operation above are typically destructive. Thus, the data structure passed as an argument is mutated over the course of function evaluation, and a modified version is returned. The original data structure, and the resulting data structure thus typically occupy mostly overlapping regions of the main memory.

Each operation on a data structure has a given runtime complexity, parameterized by the size of the data structure. For instance, the insertion of an element into a balanced tree requires $\mathcal{O}(\log(\mathsf{N}))$ steps, where N is the number of elements stored in the tree.

To perform sequences of operations, like the insertion of a range of elements into a given tree, those operations are chained. Thus, to insert the elements $\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{n} \in \mathsf{E}$ into a preexisting data structure $\mathsf{d}_0 \in \mathsf{D}$, the operations

$$\begin{aligned}
\mathsf{d}_1 &:= \mathsf{insert}(\mathsf{d}_0, \mathsf{e}_1) \\
\mathsf{d}_2 &:= \mathsf{insert}(\mathsf{d}_1, \mathsf{e}_2) \\
&\ldots \\
\mathsf{d}_\mathsf{n} &:= \mathsf{insert}(\mathsf{d}_{\mathsf{n}-1}, \mathsf{e}_\mathsf{n})
\end{aligned}$$

need to be evaluated. If the insertion operation is $\mathcal{O}(\log \mathsf{N})$, the overall sequence of operations will cost $\mathcal{O}(\mathsf{n} \log \mathsf{N})$. Thus, the overall cost of the sequence is the sum of its individual steps.

However, considering the concrete example of a balanced tree, performing sequences of single-element insertion operations could be conducted more efficiently — by sharing information among insertions.

For instance, consider the insertion of the value 32 into the binary search tree outlined in Figure 2. The new value is to be inserted as the right child of the node 18. At this point, it is known that the sub-tree to be rooted by the new value 32 is to contain all values between 18 and 50. Thus, if the next value to be inserted is in this range, navigation through the tree can be omitted, and the new element can be directly inserted as a child node of 32. Under these circumstances, the costs for inserting the second element is reduced from $\mathcal{O}(\log \mathsf{N})$ to $\mathcal{O}(1)$. To forward this kind of information from one insertion operation to the next, we introduce operator hints.

Let H be the set of all those hints. Then an insertion operation utilizing hints is specified by

$$\mathsf{insert}_H : (D, E, H) \to (D, H)$$

An insertion operation thus takes a data structure, the element to be inserted, and some operation hints as an input, and produces an updated pair of a data structure and a hint for subsequent operations. Analogous, the *contains* operation can be modified to include operator hints such that

$$\mathsf{contains}_H : (D, E, H) \to (\mathbb{B}, H)$$

The steps to perform sequences of element insertions get modified to

$$\begin{aligned}
(\mathsf{d}_1, \mathsf{h}_1) &:= \mathsf{insert}(\mathsf{d}_0, \mathsf{e}_1, \mathsf{h}_0) \\
(\mathsf{d}_2, \mathsf{h}_2) &:= \mathsf{insert}(\mathsf{d}_1, \mathsf{e}_2, \mathsf{h}_1) \\
&\ldots \\
(\mathsf{d}_\mathsf{n}, \mathsf{h}_\mathsf{n}) &:= \mathsf{insert}(\mathsf{d}_{\mathsf{n}-1}, \mathsf{e}_\mathsf{n}, \mathsf{h}_{\mathsf{n}-1})
\end{aligned}$$

using some initial operator hint $h_0 \in H$. Thereby, the overall runtime complexity can be reduced. In the binary search tree example outlined above, inserting a sequence of n consecutive elements into a gap of the initial tree reduced from $\mathcal{O}(n \log N)$ to $\mathcal{O}(\log N + n)$.

Naturally, one could argue to integrate hints into data structures directly, thereby eliminating the need for the extra parameter. However, keeping them apart allows client code to manage the context of insertion operations. For instance, assume two list of consecutive elements $a_1, \ldots, a_n \in E$ and $b_1, \ldots, b_n \in E$ have to be inserted into a binary search tree in an interleaved fashion. By maintaining operator hints within the data structure, the proximity of the elements ending up in the tree is lost, and the beneficial effect is not materializing. However, by maintaining two separate operator hints external to the data structure, those sequences can be inserted using their respective hint values, thus benefiting from this mechanism. While seemingly being a contrived example, this is precisely the situation to be faced in a concurrent environment where those two sequences are to be inserted by two distinct threads into a shared data structure.

In a concurrent environment, multiple threads might attempt to insert sequences of elements into a shared data structure. Allowing externally managed operator hints does not only enable those threads to benefit from the proximity of their processed elements, but also to maintain the actual operator hints within thread-private storage. Thus, updates to operator hints do not need to be synchronized — as it would be the case if hints would be managed internally.

3.4.2 | Data Structure Perspective

Operator hints effectively facilitate the realization of caches for partial results of former operator evaluations. For instance, in the binary search tree insertion case, the pointer to node 32 rooting the subtree covering the range $(18, \ldots, 50)$ can be maintained as a hint, thereby avoiding tree-navigation costs in future insert operations. These partial results of operations may be retrieved through caches faster than they might be re-computed using the management information included in the data structure.

Since hints thus exhibit the characteristics of caches, their design needs to scope with a range of questions: what information should be cached, what key is utilized to address the cached information, how is cached information invalidated, how much information is cached without losing its performance benefit, and what replacement policy for cache entries shall be used.

In the binary search tree example above, we outlined the utilization of a single cache entry formed by a node pointer addressing the most recently introduced tree node. We utilized the boundaries of its covered sub-tree to test the relevance of this information for future inserts. Fortunately, since simple binary search trees are never restructured, this information is never invalidated by future inserts, hence no invalidation has to be realized. Finally, since we only store a pointer to the last inserted node, we utilize a single-slot cache with least-recently-used (LRU) replacement policy.

Naturally, all those design decisions can be explored. In particular, by adding additional slots, more intricate patterns of insertions could benefit from operator hints. While a single hint entry improves the performance of one sequence of consecutive elements, caching two hints using an LRU policy will lead to $\mathcal{O}(\log(N) + n)$ insertion complexity also in the case of two interleaved sequences.

4 | OPTIMISTIC B-TREE

B-trees [19] are among the most widely utilized data structures for maintaining large quantities of keys. Their sorted and balanced nature provides $\mathcal{O}(\log n)$ worst-case execution time for insert and lookup operations, $\mathcal{O}(n)$ traversals, as well as $\mathcal{O}(\log n + r)$ complexity for range queries, where r is the size of the range. The mostly consecutive storage of keys in inner nodes and leaves further facilitates efficient utilization of memory caches due to a high degree of spatial locality of consecutive accesses. Like other data structures, B-trees require $\mathcal{O}(n)$ space. However, by blocking the storage of keys in nodes, the overhead of management information like pointers is effectively amortized, even for small key types.

While providing outstanding performance in many use cases demanding sequential accesses, state-of-the-art concurrent implementations of B-trees cannot cope well with insert-intensive workloads as imposed by parallel Datalog evaluation. In particular, within parallel sections, all threads are continuously attempting to insert data into a single shared B-tree instance, constituting a near worst-case scenario for typical locking schemes. Maintaining the consistency of the internal data organization requires the utilization of sophisticated locking mechanisms and schemes, typically limiting parallel scalability. Frequent lock state and data updates need to be propagated among CPU cores sharing access to B-trees, causing high communication demands on interconnects, easily leading to resource contention.

While imposing challenges on the scalability of insert operations, the Datalog use case also exhibits properties leading to relaxation of synchronization requirements. In particular, the Datalog evaluation scheme guarantees that no data structure is ever read and written concurrently. Membership tests, range queries, and traversals are never processed while performing insertions on the same tree. Thus, these operations do not need to be shielded from each other.

The remainder of this section outlines a concurrent B-trees design providing highly scalable insertion performance on multi-core and multi-socket architectures. It is based on a new optimistic fine-grained locking scheme using *optimistic read-write locks*[2]. By avoiding memory modifications in the most frequent read-only case – even when acquiring and releasing locks – the utilization of intra- and inter-socket communication channels is vastly reduced. Furthermore, by caching leaf-nodes reached by insertion and lookup operations within thread-private variables, a hint mechanism shortcutting consecutive tree navigation steps has been integrated to exploit temporal locality of leaf-node accesses.

4.1 | Synchronization

The two phases of the parallel semi-naïve evaluation of Datalog (cf. [17]) guarantee the strict separation of concurrent read- and write accesses to shared data structures. Data structures are either (concurrently) read by multiple threads, or written to. Thus, since lookups, emptiness checks, and range queries do not modify any shared state of a B-tree, only multiple concurrent executions of the B-tree's insertion operation need to be synchronized to isolate their effects from each other.

Previous work has been exploring a wide variety of locking schemes for B-trees [12]. Designs range from global locks, over per-node fine-grained locking using mutex or read/write locks, to implementations utilizing the capabilities of hardware transactional memory [20]. Many of those are limited in their scalability due to high bandwidth requirements caused by frequent cache line invalidations when updating lock state information. Especially the lock protecting the root node, which is involved at the beginning of every insert operation, causes a high amount of communication, resulting in a significant performance penalty, and thus in severely limited scalability.

4.1.1 | Optimistic Read-Write Lock

The *seqlock* [21] design constitutes a highly scalable lock solution for such frequent read / seldom write use case. Its design is based on the optimistic assumption that conflicts are very unlikely to occur. Under these circumstances, it can be cheaper to focus on reactive conflict detection than conventional proactive conflict prevention – an idea also previously exploited in database applications [22].

The central idea is to associate a *sequence number* to each shared blocks of data like e.g. a B-tree node. The sequence number covers two pieces of information: a version number incremented whenever the data is updated, and one bit to indicate whether an update is currently in progress. The progress bit is typically the last bit of the sequence number, turning even numbers into indications of stable states, without active writers, and odd numbers into indicators of ongoing updates.

Before accessing shared data, the sequence number is retrieved. If it is odd, a write operation is in progress, and the accessing thread (busy-) waits for its completion. Once even, the sequence number is recorded and the read operation on the shared data performed. To complete the read, the sequence number after reading the data is read again and compared to the recorded value. If identical, no concurrent update has occurred. If different, a conflict has been detected and the read operation may have to be repeated.

Write operations start by attempting to atomically increase an even sequence number to an odd number. On success, exclusive write access is guaranteed – though concurrent read accesses may be present. The writing thread may then continue by modifying the shared data, before finally increasing the now odd sequence number to the next even number. This unblocks new read or write operations and allows ongoing read operations to detect conflicts.

However, during B-Tree insert operations, threads are not clearly separated into reading or writing threads. Each thread passing through a node is a reading thread, but may later be mutated to a writing thread in cases a key needs to be inserted in a leaf node or an inner node gets split. Thus, the basic seqlock scheme needed to be customized to what we refer to as an *optimistic read-write* lock. Each thread starts as a reading thread, but may optionally upgrade its read permission to write permission, without invalidating read data. Figure 3 outlines the state diagram of every thread accessing a protected region using our optimistic read-write locks.

Each access follows the idea of a *transaction*, with a clearly indicated start and end. Each transaction starts by reading data, followed by an optional write phase, and consistency check at the end. Upon failure, the transaction is restarted.

To that end, the optimistic lock offers several operations. For read accesses, *start_read*, *valid*, and *end_read* are provided. The valid operation may be used during ongoing transactions to validate data read so far. The operations *try_upgrade_to_write*, *end_write*, and *abort_write* are offered to control the start and termination of the optional write phase. Write phases may be aborted in case no modification is performed although write access has been acquired. In this case, the sequence number is reverted back to its initial state (decremented), to avoid invalidating ongoing concurrent reads. Furthermore, the non-blocking operation *try_start_write* and the blocking operation *start_write* are offered to skip the initial read phase and enter directly into a write phase.

[2] Hence the name optimistic B-tree

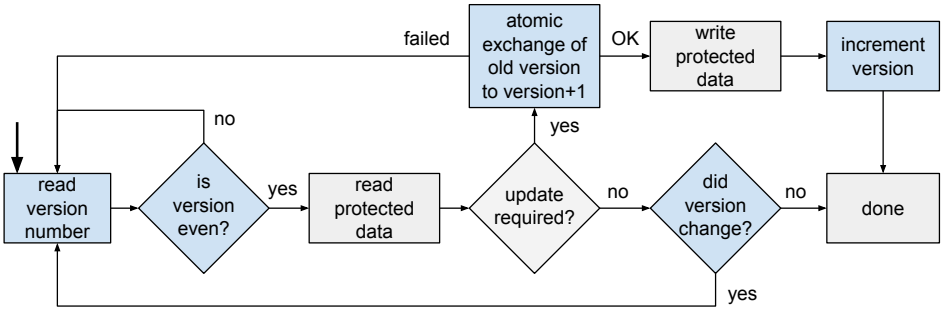


FIGURE 3 Outline of optimistic locking scheme.

4.1.2 | Optimistic B-tree Insertion

In our optimistic B-Tree, each node is equipped with an optimistic read-write lock to protect its internal state. Furthermore, the root node pointer is as well protected by an additional *root_lock*. The orchestration of these locks to facilitate a properly synchronized key insertion procedure is outlined in Algorithm 1.

Lines 2-9 cover the thread safe creation of a root node, only triggered when attempting to insert into an empty tree. In this case, an initial root-node is created by the first thread to pick up the root lock. After ensuring that at least one node is present, lines 11-49 cover the actual insertion procedure for inserting keys in non-empty trees.

Lines 13-17 are tasked to obtain a valid reference to the current root node, after which lines 20-33 navigate down the tree, chaining locks. When reaching a leaf node, the new key is inserted (lines 35-47). To that end, read permissions are upgraded to write permissions (line 36) and the new key is inserted (line 46) if there is sufficient space. If, however, the targeted leaf node is full, it needs to be split (line 40), and the insertion is restarted at the top. The splitting procedure is summarized in Algorithm 2.

Before performing modifications on the tree, the split procedure acquires write permissions on all nodes to be modified – bottom-up. The involved nodes comprise all nodes on the path from the targeted leaf node back up to the root node up and including the first non-full node. Lines 2-23 perform the corresponding navigation and locking steps. Once permissions have been acquired, nodes are split (line 26) and locks are released in reverse order (lines 28-35). As typical for B-trees, by splitting nodes the lower half of the keys are retained by the split node, while the upper half is moved to a new node, which is registered in the parent-nodes next-pointer list, referenced by the median key value.

The following invariants are obeyed at all time: almost all data stored in a node, including keys in leaf nodes as well as keys and pointers in inner nodes, are protected by the enclosing nodes' read/write lock. The only exception is given by a nodes parent pointer, which is protected by the parent node's lock or – in case of the root node – by the root pointer lock. Whenever reading any of those values, a read permission is acquired from its protecting lock, and after the value is obtained, its validity is verified before consuming the value by e.g. dereferencing a node pointer (e.g. line 28 in Algorithm 1). Analogous, modifications are encapsulated by the acquisition and release of write permissions. In all cases, detected conflicts are handled by restarting the overall insertion algorithm.

The outlined locking scheme detects read/write conflicts but sustains from spending time (and memory accesses) on preventing those. It is thus optimistic. However, conflicts may occur, and invalid values may be read from shared memory locations. It is thus essential to validate version numbers before using any obtained data. In particular, dereferencing corrupted values extracted for child-node pointers would cause segfaults in a best case scenario, and invalid, next to impossible to detect subtle memory corruptions in the worst case.

Since exclusive write-permissions are only acquired bottom up, and held read-permissions are not preventing other thread from acquiring write permissions, deadlocks are effectively avoided since no dependency cycles can be formed this way. However, rare lock contention situations may occur for inner nodes forming the root of multiple paths to leaf nodes only comprising fully occupied nodes.

The advantage of the presented optimistic locking scheme over similar fine-grained schemes using conventional locks is derived from optimizing the most frequent use case of a mere read of the state of an inner node. The navigation through an inner node only involves memory reads for handling lock operations, key search steps, and child-node pointer retrieval. Non of these reads to shared memory cause any cache line invalidation, and thus no bus communication. Using conventional locks, at least the lock state itself would have to be mutated twice – when acquiring the read permissions and when releasing it.

4.1.3 | Optimistic B-tree Implementation

Seqlocks impose a particular challenge to be implemented correctly using contemporary programming languages. Their basic idea is to permit parallel accesses, and detect and compensate for conflicts in rare cases. However, by doing so, seqlocks depend on properly defined semantics for race

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
```

conditions – a case frequently excluded from the definitions governing the memory model underlying programming languages. C++, in particular, states that any program with data races exhibits *undefined* behavior. Implementing seqlocks thus requires a careful encoding.

The problem of implementing seqlocks using modern C++ has been invested by Boehm in great detail [23]. The key idea is to exploit C++'s exception of data being accessed through atomic wrapper from any sort of race condition – by definition. Thus, as long as all shared data that is protected by seqlocks is accessed through atomic data type wrappers, a race condition free, valid seqlock implementation can be realized. In his paper, Boehm outlines several different refinements of this idea, minimizing the performance penalty introduced by those atomic wrappers. By carefully utilizing memory order arguments and synchronization fences, the native memory model of a targeted architecture can be exploited to the point where accesses through atomic wrappers yield (almost) identical sequences of assembly instructions as non-wrapped access, without sacrificing correctness.

For our implementation, we adopted a solution presented by Boehm. When entering a critical region, the sequence number is obtained using `memory_order_acquire`. Shared data is then read using `memory_order_relaxed`. Finally, to validating retrieved data, a `memory_fence_acquire` is included to force all those relaxed reads to be completed, before reading the lock's sequence number using `memory_order_relaxed` to complete the validation.

Algorithm 2 Optimistic B-tree node splitting procedure.

```
 1: procedure SPLIT(tree,node)
 2:     // write-lock path bottom-up
 3:     cur ← node
 4:     parent ← cur->parent
 5:     path ← ϵ
 6:     while true do
 7:         if parent != null then
 8:             while true do
 9:                 start_write(parent->lock)
10:                 if parent == cur->parent break
11:                 abort_write(parent->lock)
12:                 parent ← cur->parent
13:             end while
14:         else
15:             start_write(tree->root_lock)
16:         end if
17:         path ← append(path,parent)
18:
19:         // stop at root or non-full inner node
20:         if parent == null or !full(parent) break
21:         cur ← parent
22:         parent ← cur->parent
23:     end while
24:
25:     // conduct actual split
26:     < split node and propagate to parents >
27:
28:     // unlock path top-down
29:     for all parent in reverse(path) do
30:         if parent != null then
31:             end_write(parent->lock)
32:         else
33:             end_write(tree->root_lock)
34:         end if
35:     end for
36: end procedure
```

4.2 | Operation Hints

The data produced during the evaluation of Datalog queries using nested loop joins[6] may, in certain occasions, trigger membership queries and inserts following the same order as data is stored in tables. Thus, consecutive operations may require to navigate to consecutive keys stored within the same leaf node of the B-Tree. This spatial locality can be exploited to reduce the number of times the B-Tree has to be navigated top-to-bottom by consecutive calls. This is realized through the integration of an operation hint mechanism into our B-Tree design.

Essentially, each thread maintains pointers to the last leaf nodes reached in a B-Tree instance when performing an insertion operation or a membership test. Both hints are maintained separately within a thread-private variable, allowing effects to be bound to the operation type, tree instance, and thread context. When performing inserts (or membership tests), those hints referencing leaf nodes are tested first. If they reference a node in which the inserted key is to be located, the navigation from the root node to the leaf can be omitted. In case of a membership test, the leaf referenced by the matching hint can be inspected directly. In case of an insertion operation, keys can be directly inserted in the leaf node. In either case, if the provided operator hints are invalid for the operation at hand, they are ignored and the default membership test or insertion procedure is executed.

For example, when inserting the key $(3, 4)$ into a B-tree using standard lexicographical ordering, it may end up in a node storing keys ranging from $(2, 9)$ to $(4, 1)$. By maintaining the reference to this node, a subsequent insertion of e.g. the key $(3, 6)$ can be immediately performed on the node located by the previous insertion, since the key is as well contained in the range of elements covered by this leaf node. However, if this insertion is followed by inserting an out-of-range key like $(5, 2)$, the maintained hint is ignored, and a regular insertion starting at the B-Tree's root node is initiated. Hints for lookup operations operate analogously.

By storing operator hints in thread-private memory locations, typically maintained on the processing thread's stack, and passing them along as extra arguments to insert and lookup operations, thread-safe hint manipulations are automatically derived. Since insertion hints may cause the navigation-down phase to be skipped, no read-locks along the path from the root-node to the leaf node are obtained. These locks are, in the regular version of the insertion procedure as outlined in Algorithm 1, required to ensure that the correct leaf node has been reached. However, since from the boundaries of the leaf node in case of the utilization of a hint this property can be ensured, these missing read-locks can be tolerated. Nevertheless, write locks for splitting nodes need to be acquired bottom-up as usual. This bottom-up exclusive-lock-acquiring characteristic of our locking scheme thus facilitates the efficient utilization of insertion hints even in concurrent access contexts.

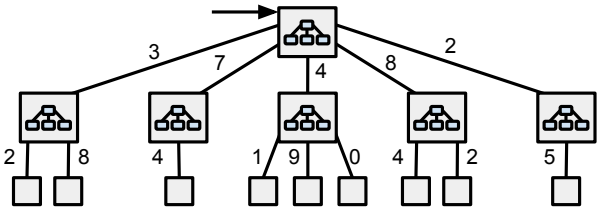


FIGURE 4 Overview of the structure of a BRIE.

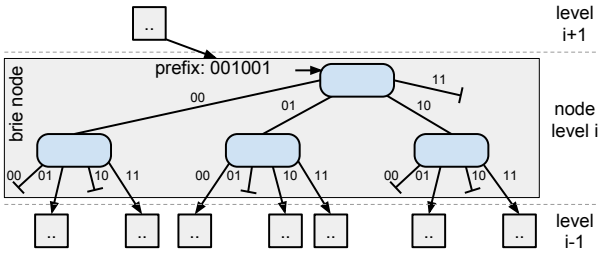
Finally, our C++ implementation covers various additional minor optimizations to improve its performance in the Datalog context: (1) most recursive operations are replaced by iterative versions to avoid excessive function-call overheads, (2) the involvement of node-rebalancing operations to reduce the number of necessary node splits and increase memory efficiency, (3) a linear B-tree merge algorithm, and (4) the utilization of customized 3-way comparison functions for B-tree entries. All these optimizations cover implementation details tuned to improve real-world execution performance. However, they do not contribute new algorithmic insights beyond what has been presented above. The implementation of our B-Tree is open-source and forms one of the foundations of the Soufflé Datalog engine [24]. The implementation can be found in the file `BTree.h` and is licensed under UPL V1.0.

5 | BRIE

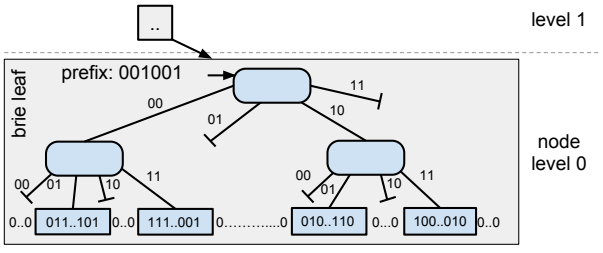
The optimistic B-tree is specialized for the concurrent access scheme exhibited by the Datalog evaluation scheme. Another approach is to specialize a data structure by considering the properties of the expected data to be maintained. For example, in the case of points-to analysis the data inserted in the most performance relevant relations are often *dense*, meaning that within the boundaries of a covered (n-dimensional) range a high number of elements will be present. This property can be utilized for designing a more efficient n-dimensional point data structure.

In this section, we introduce the *Brie*. The BRIE is structurally based on a trie and uses a node-blocking scheme found internally in B-trees. The trie based structure eliminates the need for tree reshuffling, and facilitates lock-free insertion operations, while the node-blocking scheme leads to an efficient, cache friendly consecutive storage of elements. Combined, these BRIE traits enable improved performance on dense data sets.

In Figure 4 the overall structure of a BRIE is shown. The foundation of the BRIE design is a fixed-height trie. Unlike B-trees or classic binary search trees, the BRIE encodes entries through its edges instead of explicit values in nodes. Each edge is labeled by an integer, forming a component of a stored tuple. Technically, this is realized by maintaining a map between labels and child node pointers. This mapping is based on quadtrees extended by the node blocking feature of B-trees. Moreover, bit-masks are used on the leaf node level to compress the information of the presence or absence of values, into a single bit.



(a) Internal structure of an inner node of a BRIE.



(b) Internal structure of a leaf node of a BRIE.

Figure 5a depicts the inner structure of an inner node of a BRIE. The mapping between tuple component values and child node pointers is realized by a blocked quadtree. The binary encoding of the map's keys is used for navigating the quadtree. On the leaf level of the inner node's quadtree pointers to the next level of the overall BRIE are stored.

To improve cache efficiency, multiple levels of the inner quadtree are collapsed into single nodes. Figure 5a illustrates this collapse for the two-level case. However, we note that the actual implementation collapses a configurable number of levels into a single node. For the experiments in Section 6, an empirically determined optimal value of 6 levels is selected.

Algorithm 3 Brie insertion procedure (simplified).

```
 1: // node handling utility
 2: procedure ENSUREEXISTENCE<T>(ptr)
 3:     // handle none-existing root
 4:     if ptr == nullptr then
 5:         new ← new T()
 6:         if !CAS(ptr,nullptr,new) then
 7:             // somebody else was faster
 8:             delete new
 9:         end if
10:     end if
11:     return ptr
12: end procedure
13:
14: // inner-node tree navigation utility
15: procedure GETLEAF(root,val)
16:     // handle potentially none-existing root
17:     ENSUREEXISTENCE<INDEXNODE>(root)
18:
19:     // adapt prefix if necessary
20:     ENSUREPREFIXCOVERED(root,val)
21:
22:     // navigate down the internal quadtree
23:     info = ATOMIC_LOAD(root->prefixInfo)
24:     cur ← info.root
25:     level ← info.level
26:     while level >= 0 do
27:         next ← cur->next[(val » level) & 0x3F]
28:         cur ← ENSUREEXISTENCE<INDEXNODE>(next)
29:         level ← level - 6
30:     end while
31:     return cur->value
32: end procedure
33:
34: // entry point for insertion of value
35: procedure INSERT(node,val,level)
36:     // handle leaf brie node
37:     if level == 0 then
38:         mask ← GETLEAF(node->index,val[0] » 6)
39:         ATOMIC_OR(mask,1«(val[0] & 0x3F))
40:         return
41:     end if
42:
43:     // navigate through inner brie node
44:     next ← GETLEAF(node->index,val[level])
45:
46:     // if next does not exist, create it
47:     ENSUREEXISTENCE<BRIENODE>(next)
48:
49:     // insert recursively
50:     INSERT(next,val,level-1)
51: end procedure
```

Another optimization is performed by pruning branches that do not contain any child node pointers. Additionally, the root is formed by the first node with more than a single child. In this way, a chain of nodes representing a common prefix of all keys at the top of the quadtree is avoided. Instead, the common prefix and its length is stored in an extra variable. This prefix is shortened as necessary during insert operations.

The structure of leaf level nodes in the BRIE is shown in Figure 5b. While conceptually identical to inner nodes, the child node pointers are replaced by machine word sized bitmaps indicating the presence or absence of entries. Thus, effectively the last $\log_2(\mathrm{W})$ levels of the index structure are collapsed into a single machine word W bits wide.

The insert operation for the brie is shown in Algorithm 3. The insert is synchronized using lock-free techniques that are based on atomic operators. These techniques facilitate the necessary means to set individual bits in the bitmaps maintained by leaf nodes and to atomically compare-and-swap pointers in inner node levels.

The *insert* operation adds a new element to a given BRIE by utilizing several auxiliary functions. The function *getLeaf* performs the node-internal quadtree navigation and manipulation within nodes. Furthermore, the generic function *ensureExistence* is used to create new nodes on demand while navigating the node-internal quadtree (lines 17 and 28) or the overall trie based metastructure (line 47). Finally, the *ensurePrefixCovered* function

Algorithm 4 Brie inner node shared prefix utilities.

```
 1: // a utility to check whether a given prefix is sufficient
 2: procedure COVERS(info,val)
 3:     i ← ATOMIC_LOAD(info)
 4:     if i.root == nullptr return false
 5:     mask ← (-1) « i.level
 6:     return val & mask == i.prefix
 7: end procedure
 8:
 9: // a utility to shorten the prefix by 6 bits
10: procedure RAISELEVEL(info,val)
11:     // get current state (atomic)
12:     oI ← ATOMIC_LOAD(info)
13:
14:     // create new prefix info struct (on stack)
15:     nI ← PrefixInfo()
16:     nI.level ← oI.level + 6
17:     nI.root ← new Node()
18:     nI.root->next[(oI.prefix » oI.level) & 0x3F] ← oI.root
19:
20:     // initialize or update prefix
21:     if oI.root == nullptr then
22:         nI.prefix ← val & ((-1) « 6)
23:     else
24:         nI.prefix ← oI.prefix & ((-1) « nI.level)
25:     end if
26:
27:     // compare and swap prefix index struct (atomic)
28:     if !CAS(info,oI,nI) then
29:         // somebody else was faster
30:         delete nI.root
31:     end if
32: end procedure
33:
34: // adapts the shared prefix
35: procedure ENSUREPREFIXCOVERED(node,val)
36:     while !COVERS(node->prefixInfo,val) do
37:         RAISELEVEL(node->prefixInfo,val)
38:     end while
39: end procedure
```

adapts the common prefixes stored as part of the node-internal quadtrees and grows those trees bottom-up whenever needed (line 20). Algorithm 4 outlines the details of this third function.

Whenever a new element is inserted into an inner quadtree, its key value may exceed the coverage of the currently maintained common prefix value (line 36). To include the new key, the number of leading significant bits of the common prefix is gradually reduced, until the new key is covered as well. Simultaneously, in each step a new root node is added. The function *raiseLevel* covers the synchronized introduction of a new root node and the corresponding update of management information. The latter comprises the root node pointer, the level counter, and the common prefix. Algorithm 4 uses a hard-coded increment of 6 bits per level, thus the utilization of $2^6 = 64$ child node pointers in each quadtree node. Overall, this results in inner quadtrees to facilitate high-output degree nodes grow bottom-up similar to a B-tree. However, no reshuffling of data nor node splitting is involved. Pointers once set within the BRIE are never updated or moved.

A synchronized implementation of the insert function has to protect three critical regions: (1) the insertion of a new BRIE node, (2) the setting of bits in leaf nodes, and (3) the update of the common prefix, and thus the raising of the height of nested quadtrees.

Line 6 of Algorithm 1 realizes the synchronized addition of BRIE nodes. In this line, a pointer to a newly created subtree is replacing a null-pointer marking an empty subtree. The exchange is conducted using a compare-and-swap (CAS) operation. This atomic operation ensures that only one thread inserts a new subtree ever successfully at any given position in the tree. Thus, in case multiple threads attempt to insert newly created subtrees at the same position simultaneously, only one of those will succeed. The remaining threads will detect the collision, discard the non-accepted temporary subtrees, and continue using the winning instance.

The second operation to be synchronized to handle concurrent modifications is in line 39 of Algorithm 3. It synchronization threads concurrently attempting to set bits in leaf-node level bit masks. Here we mark the presence of tuples in leaf nodes of the BRIE. The utilization of an atomic *boolean_or* operator is sufficient to ensure correct updates. Furthermore, update conflicts do not have to be detected, since their effects are implicitly aggregated.

The third critical region, located in the function *raiseLevel* in Algorithm 4, handles the growing of the BRIE-node local quadtrees. Conceptually, its synchronization also follows the CAS approach. However, unlike the previous case handling the first critical region, the value of the memory

location to be updated is not a simple scalar, but a small struct maintaining the common prefix, its length, and the root node pointer. In line 12, this current state of this struct is retrieved atomically and used for computing a new prefix, prefix length, and root node. Once completed, it is compare-and-swapped in as a replacement in line 28. This update of the prefix information contains concurrent updates detection and the discarding of temporary values in case of conflicts. The support for CAS operations on wider data types comprising multiple fields as utilized in this example depends on widely available platform specific support. C11 and C++14, as well as many other languages, offer atomic utilities to access those features or functionally equivalent substitutes.

The updates (1) and (3) may induce expensive conflicts, however, the probability of such conflicts occurring is low. Typical work load patterns will result in threads inserting elements in distinct parts of a BRIE, thus avoiding any conflicts. Non-conflicting inserts have very limited overhead over sequential, non-synchronized versions. In essence, the overhead is given by the utilization of a small number of atomic CAS operations instead of ordinary assignments.

When comparing conventional B-trees to the BRIE, our BRIE offers a number of key benefits:

Firstly, insertion operations are computationally cheaper. Since the navigation steps in the BRIE do not depend on search operations over ranges of keys, in each step, the following node can be directly addressed. Moreover, insert operations do not require insertions into ordered arrays and the movement of lists of keys as required in B-tree implementations.

Secondly, in the case of memory usage, the sharing of common prefix on both the trie and quadtree structures, as well as the utilization of a single bit to mark the presence of an entry on the leaf node level contribute to a reduction in the amount of memory required to store a given set of entries. However, as we demonstrate in Section 6, the efficiency of this compression capability depends heavily on the correlation of entries. Therefore, if common prefixes are sparse the memory utilization may be significantly higher compared to B-trees.

Lastly, with the presented synchronization scheme, synchronization is conceptually lock-free (assuming sufficient atomic operation support) and linearizable. The latter can be shown based on the sequentially consistent order of successful CAS operations.

6 | EVALUATION

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

- 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 $\mathcal{O}(1)$, yet no efficient support for range queries
- a state-of-the-art B-tree implementation provided by Google [25], 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 [26] 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*

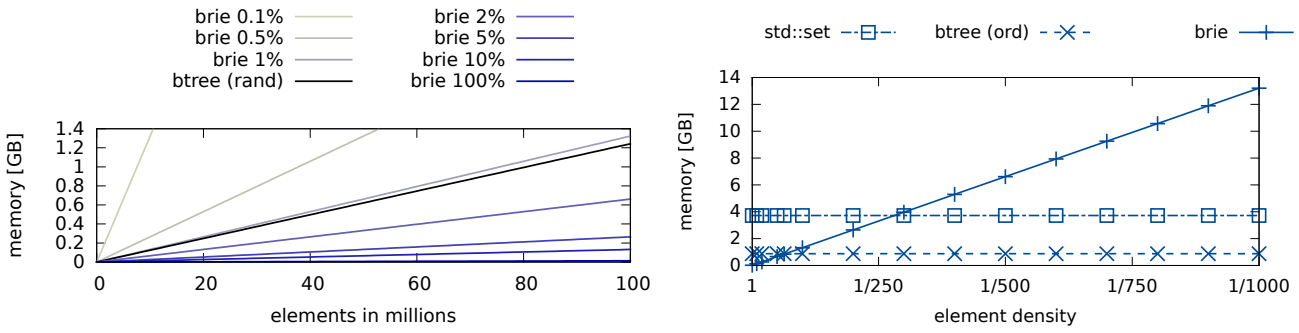
Table 1 provides a summary of all data structures included in our evaluation.

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 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.

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
brie	yes	our Brie

6.1 | Memory Requirements



(a) B-tree vs. BRIE with different point densities (10.000 samples each)

(b) Memory requirements for 100M points depending on data point density.

FIGURE 6 Memory usage of the B-tree and BRIE

The first aspect to be investigated in order to determine the suitability of our data structures for representing relations within a Datalog query processor is their memory consumption. In theory, both our data structures exhibit $\mathcal{O}(N)$ memory requirements. However, the memory usage of the BRIE depends on the *density* of those entries, while the memory usage of the B-tree is approximately consistent regardless of the characteristics of the data.

In the BRIE, the data points $(10, 11)$ and $(10, 101)$ will be represented through different 1 bits in different leaf nodes. On the contrary, the elements $(10, 11)$ and $(10, 12)$ would be represented through 1 bits in the same leaf node, consuming less memory. Since the values of stored entries are used to address the bit to be set for their representation, the actual values of the stored data points become relevant for the overall memory consumption.

Let $S \subseteq \mathbb{N}^n$ be a set of n-dimensional points and

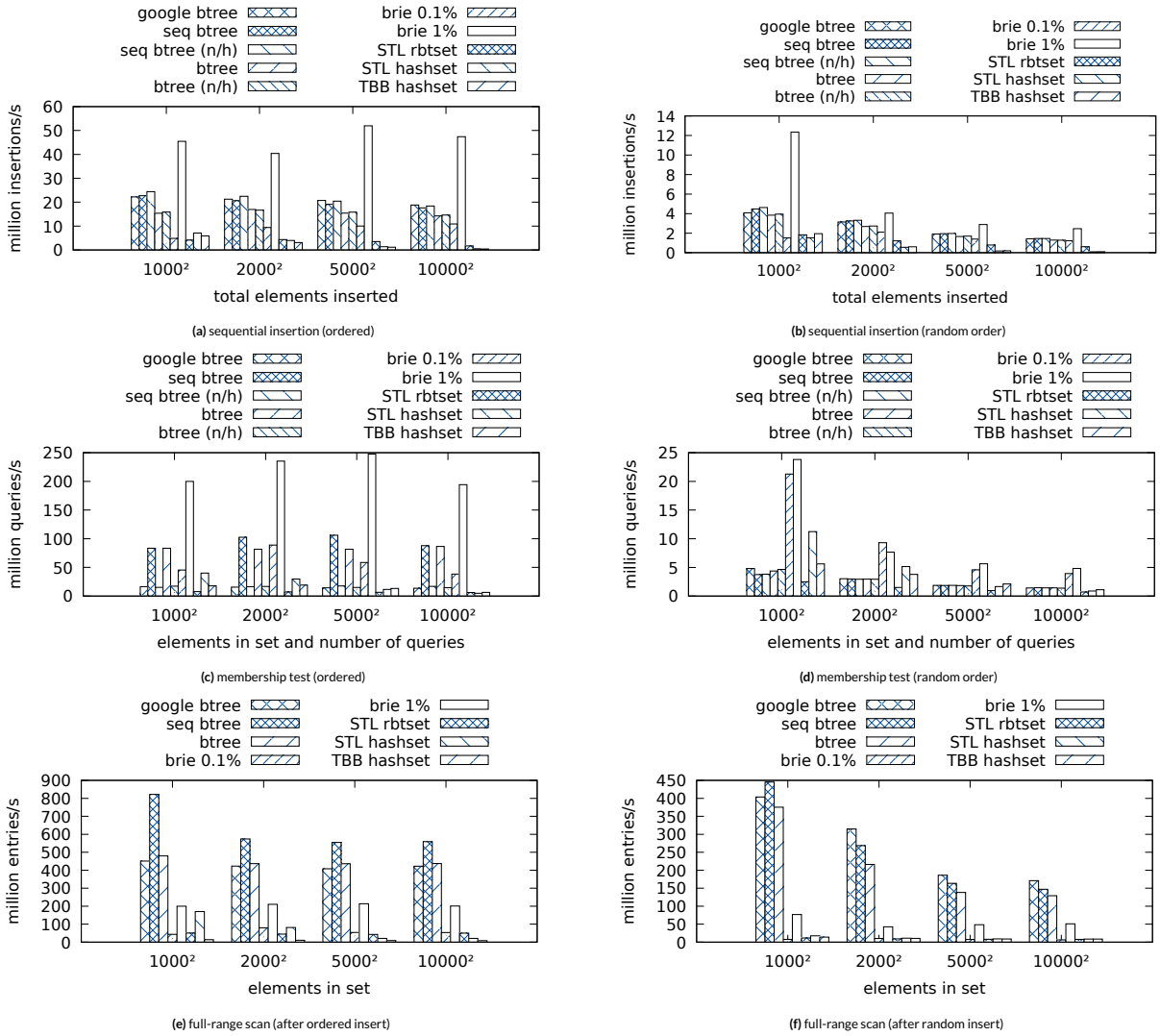
$$B(S) = \left\{ \bar{x} \in \mathbb{N}^n \mid \forall_{1 \leq i \leq n}. \exists \bar{l} \in S. \exists \bar{u} \in S. l_i \leq x_i \leq u_i \right\}$$

the set of points contained in the axis aligned bounding box of S. Then we define the *density* $d(S)$ of S by

$$d(S) = \frac{|S|}{|B(S)|} \in (0, \ldots, 1]$$

Therefore, to evaluate the memory requirements of the BRIE data structure, the density of data points needs to be considered.

For brevity, we focus on 2D data in this evaluation section. However, as additional evaluation showed, similar results are obtained when evaluating the memory consumption or performance of one- or higher-dimensional point sets.



(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 7 Sequential performance of performance critical set operations.

The memory usage of the BRIE filled with data of various densities is illustrated in Figure 6a. Results labeled "brie X%" are obtained by maintaining a data density of X%. These results demonstrate a linear scaling of memory usage with respect to the number of elements stored in the data structures. It also demonstrates that the BRIE becomes more efficient with higher data density, with the cut-off point being at approximately 1% density where the memory usage of BRIE and B-tree are equal. Therefore, with sufficiently high-density data, the BRIE provides higher storage efficiency compared to the B-tree, however with lower densities, the B-tree vastly outperforms the BRIE.

To quantify the impact of the data density on the memory requirements of a BRIE, Figure 6b summarizes the total memory usage of three different data structures to store 100M 2D data points of varying point density. Only the memory requirements of BRIEs are affected by the density. In total, to store a set of 2D points (2×4 bytes each), C++'s *std::set* requires 40 bytes, Google's B-tree between 8.8 and 10.5 bytes, SOUFFLÉ's B-tree 9.4 to 13.3 bytes, and our BRIE $\frac{0.142}{d(S)}$ bytes per point. Therefore, even at 2% density, our BRIE uses 6.4 bytes per point, which is smaller than a direct encoding using 8 bytes.

6.2 | 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. The result of this evaluation is a base-line reference of the performance of our data structures as well as insights on the overhead induced by the employed locking mechanisms.

For evaluating the performance of the insert operation, we insert varying numbers of 2D points[3] 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. Furthermore, due to the sensitivity of the BRIE towards the density of data, varying point densities were tested for the BRIE.

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 7 summarizes the collected performance data. The first row illustrates the sequential insertion performance. As can be observed, the BRIE outperforms all alternative data structures at 1% density, however, with 0.1% density the BRIE is less efficient than B-tree based data structures. Further experiments at 100% density showed that BRIEs outperformed our B-tree by up to 5× in the ordered case, and 11× in the random order case. Out of the remaining candidates, the B-tree data structures outperform the STL data structures and TBB's hash-based set. Despite the superior asymptotic runtime complexity of hash-based data structures, the cache-friendly organization of data in B-trees causes considerably fewer cache misses. This leads to superior performance compared to the random memory access pattern inherent in hash-based data structures. Among the B-tree data structures, the ordered insertions result in approximately 5× higher performance than random order insertions – partly due to the improved cache utilization and the reduced complexity of inserting elements within leaves in order. Furthermore, in the random order case, larger dataset sizes lead to weaker performance. While in both cases, inner nodes need to be passed to reach insertion points, in the random order case these inner nodes are more likely to trigger cache misses. This effect is even stronger in the BRIE, where cache misses are more likely to occur when the structures exceed the available cache sizes. Among our B-tree implementation, the operation hints cannot amortize their maintenance costs, and overhead caused by the locking mechanism can be observed (up to $\sim 25\%$ in ordered and $\sim 15\%$ in random).

Figure 7c and Figure 7d illustrate the observed query performance for ordered and unordered sequences of accesses. We observe that with sufficient data density, the BRIE outperforms all other data structures. With the random order case, the density has less effect, with 0.1% and 1% density performing similarly. This is due to cache effects, where the path in the tree up to an element may be cached and reused in the ordered case, but cannot provide benefits in the random order case. We also observe that hints provide up to a 6× performance boost for membership tests for B-trees in the ordered case, since navigating the tree can almost always be circumvented by using the hint. Secondly, the STL unordered set provides high performance for small datasets (up to 2× faster than TBB's hashset), however this advantage vanishes with larger data set sizes.

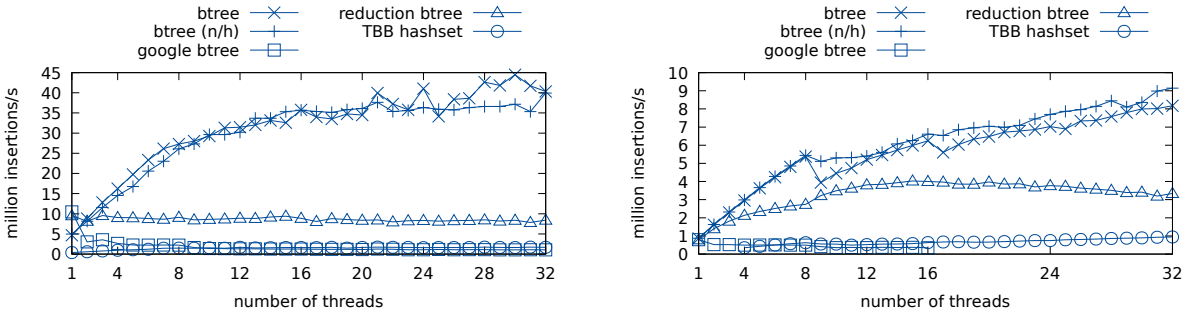
Finally, Figure 7e and Figure 7f summarize the rate at which elements stored in a data structure can be iterated through. Here, the bit-encoded values of the BRIE require reconstruction during the scan, and thus performance is lower than our B-tree by a factor of 2. However, the compact storage of data in B-trees 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 BRIEs with high-density data, with performance gains of up to 11× for insertions and membership tests.However, this advantage disappears when iterating through elements, and with lower data densities. Regardless of data density, B-tree based data structures are more performant compared to hash or red/black tree-based structures due to their cache efficiency. Also, these results demonstrate 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.

6.3 | Parallel Performance

To evaluate the parallel insertion performance, we evaluate the parallel scalability of our candidate data structures. We insert 100M 2D points into an initially empty set using a varying number of threads (strong scaling). Figure 8 summarizes the insertion throughput of our contestants with when gradually scaling up computation with multiple threads. 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 subsequent reduction step (*reduction btree*).

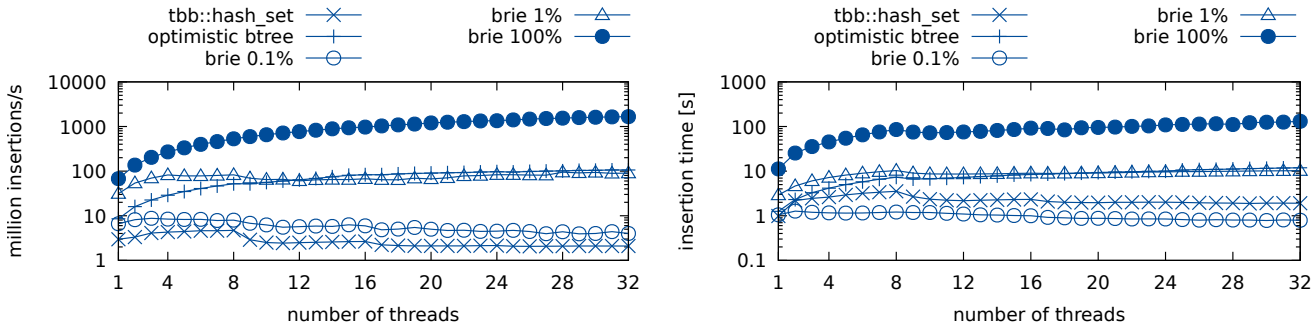
[3] 2D data is the most relevant case in many Datalog queries; besides, results remain similar for other dimensions



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

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

FIGURE 8 Parallel performance of insert operations.



(a) parallel insertion (ordered, multi socket, 4 × 8 cores)

(b) parallel insertion (random, multi socket, 4 × 8 cores)

FIGURE 9 Comparison of insertion times of 100M 2D points, using ideal density for each data structure.

While both techniques could be applied to any set implementation, for our evaluation we chose the fastest 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 hash-based set is not able to compensate for its sequential performance deficit through parallel scaling. Our optimistic B-tree outperforms TBB's hashset 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 8.8 and 10 in the 2 use cases. Furthermore, Figure 8a and Figure 8b 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 8b). Figure 8a 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 8b 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.

We also evaluate the performance of the BRIE compared to the best (our B-tree) and worst (TBB's hashset) thread-safe data structures. For a fair comparison, we evaluate three different data densities for the BRIE: 0.1%, 1%, and 100% density.

Figure 9 summarizes the obtained results. We observe that at peak efficiency, the BRIE achieves more than an order of magnitude faster performance compared to our B-tree, in both ordered and unordered cases. However, with sparse data, this advantage disappears, and at 0.1% density performance is below even that of TBB's hashset in the unordered case. In terms of parallel scalability, BRIEs achieve a parallel efficiency of up to

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

80% for 32 cores, in the best case with 100% density. However, when the data becomes too sparse, the high memory usage and associated memory management overhead and low locality result in a loss of efficiency.

We also observe that TBB's hashset implementation experiences performance penalties for each additional socket used in the computation. Both the BRIE and the B-tree also suffer slightly from crossing socket boundaries, however, they continue to benefit from additional parallel resources.

Overall, in the ordered as well as in the unordered insertion benchmarks, B-trees and BRIEs achieve their best performance with the full 32 cores. In the ordered insertion benchmark, B-trees outperform TBB's state-of-the-art concurrent set implementations by a factor of 22, while BRIEs outperform B-trees by an additional factor of 15, making BRIEs up to 350× faster than TBB's concurrent set. Therefore, the provided data demonstrates the superior parallel capabilities of our data structures, compared to the reference data structures.

6.4 | Processing Datalog Queries

For our last experiment, we integrate both our B-tree and BRIE into the SOUFFLÉ Datalog engine. We compare their performance to our other candidate data structures when used for evaluating real-world, large-scale Datalog analyses.

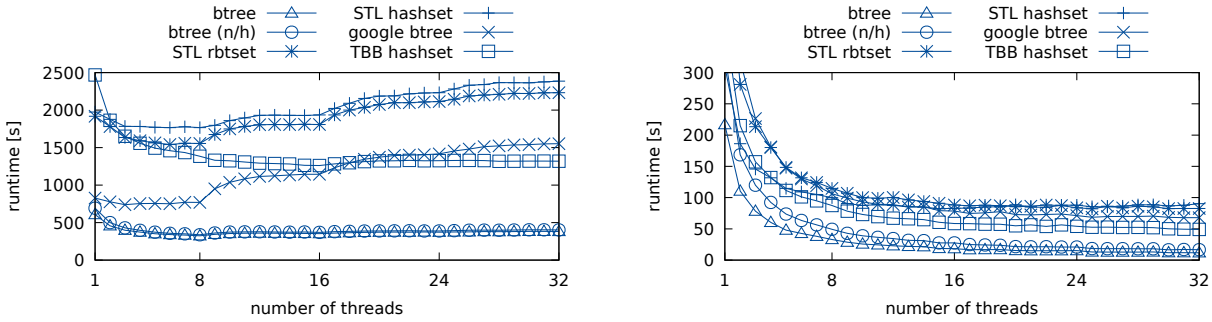
Our evaluation focus on the resulting system's ability to decrease overall processing time with increasing availability of parallel resources while keeping the processed problem size fixed (strong scaling). This corresponds to the most common use case of attempting to analyze a fixed data set – in the case of program analysis a fixed code base – as quickly as possible. On the other hand, the complex relationship between input data and inflicted workload renders designing weak scaling experiments impractical for all but the most trivial analysis.

6.4.1 | B-Tree

For our evaluation, we utilized two real-world benchmarks: a context-sensitive var-points-to analysis using the DOOP framework [1], 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.

Figure 10 illustrates the collected performance data for our two benchmarks. Note that Figure 10a 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 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%.

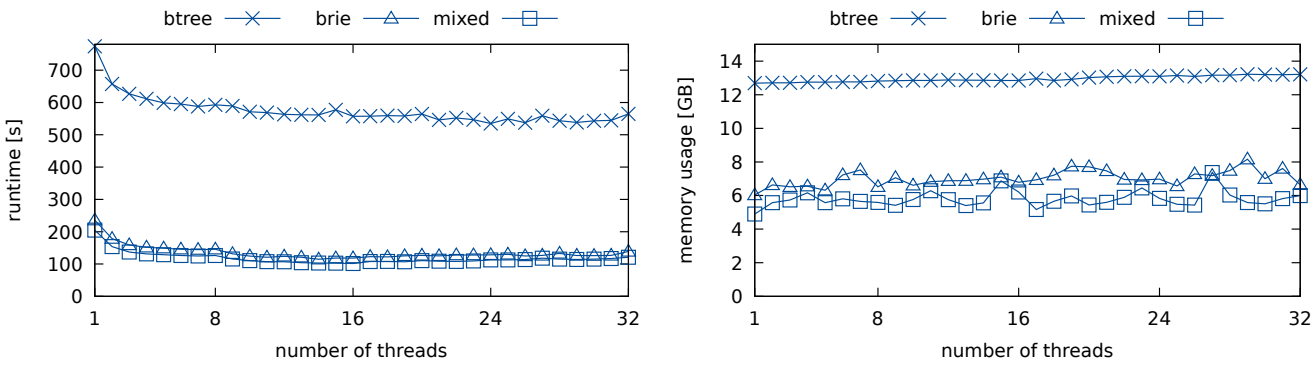
With the security vulnerability analysis, we observe that our B-tree performs approximately 2× better than the TBB hashset. Interestingly, with this workload, even the global locked data structures demonstrate some scalability, since it is read-heavy rather than write-heavy. We also note that this security analysis contains more relations with fewer tuples (1.2e7 out of the 1.6e7 produced tuples were concentrated in a single relation), and



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

(b) Security vulnerability analysis (read heavy).

FIGURE 10 Comparison of different data structures for two real-world applications relying on large scale Datalog queries.



(a) runtime

(b) memory

FIGURE 11 Comparison of SOUFFLÉ running a points-to analysis of OpenJDK with BRIE and B-tree data structures, with *mixed* denoting automatic selection of data structures.

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.

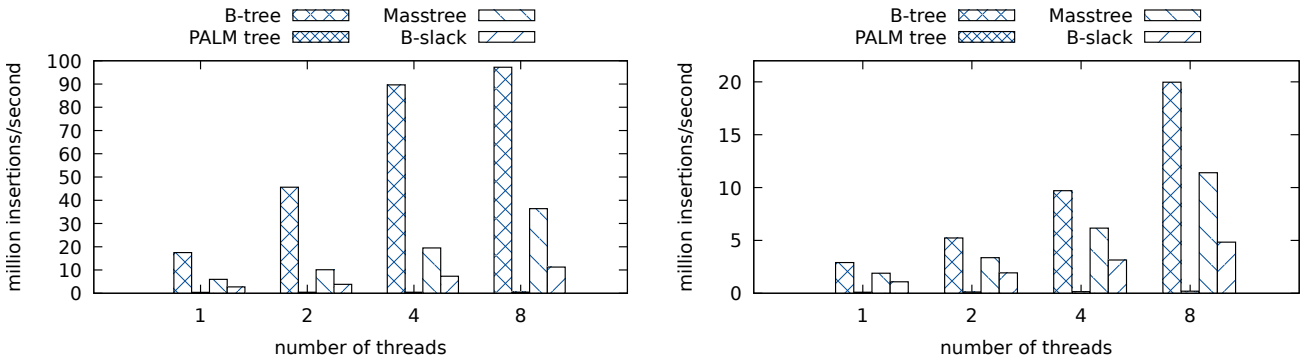
In both cases, the utilization of parallel resources provides performance improvements over the best sequential version. In the DOOP case, speedup of 1.9× can be obtained, and in the vulnerability case, a speedup of 8.4× compared to our B-tree run with 1 thread. These improvements are on top of the 1.3-3×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

6.4.2 | Brie

As demonstrated by the previous experiments, given the right circumstances, BRIEs offer vastly superior performance for (parallel) insertion and data query operations. Therefore, in practice, the BRIE data structure performs well when the workload consists of large volumes of high-density data. One example of such a workload is a points-to analysis of the OpenJDK dataset.

Figure 11 summarizes the performance of our BRIE data structure compared to B-tree when running SOUFFLÉ on this points-to analysis. We also include the performance when using an automatic *mixed* selection of data structures, whereby BRIE is used for relations with 2 or fewer dimensions, and B-trees in all other cases. The rationale for such a mechanism is that lower dimension data is more likely to exhibit high-density, and therefore the BRIE is more suitable than the B-tree.

We observe a runtime improvement of up to 4× by using BRIE instead of B-tree, and a memory usage reduction of up to 2×. The result suggests that much of the data in this workload is of high-density. Indeed, for the largest relation, *VarPointsTo*, we observed that B-tree used 11.9 bytes per tuple, while BRIE used 2.2 bytes per tuple. Being a binary relation, directly storing the data would use 8 bytes per tuple, and therefore the BRIE exploits density to store tuples 3.6× more efficiently than even a direct encoding. We also observe a slight performance improvement by using



(a) runtime (b) memory

FIGURE 12 Comparison of parallel 32 bit integer insertion throughput of our B-tree, PALM tree, Masstree, and B-slack (see Table 3)

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

a mixed data structure selection, suggesting that the workload also contains some higher dimension relations, with low data density. For these relations, BRIE performed worse than B-tree, and thus optimal performance is obtained by using a combination of the two.

6.4.3 | Comparison with Concurrent Tree Data Structures

In this section we compare our B-tree data structure to not alternative data concurrent data structures. These alternatives include *PALM tree* [27], *Masstree* [28], and *B-slack tree* [29]. We outline the limitations of these data structure's suitable for Datalog evaluation.

PALM tree [27] 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 [28] 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 [29] 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 [29].

We compared the performance of alternative data structures 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 Figure 12 and 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.

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

7 | RELATED WORK

Data Structures for Datalog

There have been a number of approaches for Datalog evaluation, based on various alternative data structures. For example, *bddbddb*[7] employs binary decision diagrams to store relations. While BDDs are particularly applicable for program analyses written in Datalog, their use also introduces the new issue of finding optimal variable orderings. Hash-based sets have also been used in systems such as μZ[10] and *SociaLite*[3]. Liu et al. propose a novel approach based on nesting arrays and linked lists, demonstrating a worst-case constant running time. However, experiments show that in practice, this technique is not scalable for large datasets. In our experience, B-trees (as used in PA-Datalog/LogicBlox version 3[30] and SOUFFLÉ[24]) have been the most suitable data structure for large ruleset/dataset benchmarks[8]. In particular, the work in[16] demonstrates the advantages of B-trees with novel indexing schemes specifically designed for these use cases. Moreover, by exploiting the property of semi-naïve evaluation that data structures are never read from and written to at the same time, we are able to further optimize the design of the data structure, using ideas from[17].

While B-trees demonstrate scalable performance for the general case, certain properties of the use case (e.g. density) may be exploited by specialized data structures. Therefore, engines such as μZ also include a framework for integrating different backends depending on the use case. For example,[2] implements a BDD-based and a bit vector-based backend to more effectively compute networking problems. For the program analysis use case, data with high-density is prevalent, and the Brie demonstrates an advantage in these situations.

Parallel Datalog Engines

The parallelization of Datalog engines has been well studied in the past[31,32,33,34,35,36,37,38,39]. These approaches are mainly focused on using distributed systems, and thus developing novel communication methods to efficiently evaluate Datalog in parallel. While most distributed Datalog approaches focus on top-down evaluation,[32,33] propose methods to parallelize semi-naïve evaluation. However, in a distributed setting, the main bottleneck for large-scale Datalog evaluation becomes the network communication overhead. Hence, approaches targeted to shared-memory multi-core machines have been developed, employing fine-grain parallelism to bottom-up evaluation schemes. For instance, Datalog-MC[40] uses hash tables internally, with range-partitioning providing the capability to execute each partition on separate cores of a shared-memory multi-core system, using a variant of hash-join. Each partition is then protected by a lock during evaluation, thus incurring appreciable overheads. To evaluate Datalog programs, Datalog-MC represents rules as and/or trees, which are compiled to Java. LogicBlox version 4[11] uses persistent functional data structures that avoid the need for synchronization. Instead, by virtue of their immutability, insertions efficiently replicate state via the persistent data structure. Another approach for parallelizing Datalog is to evaluate using GPUs. Martınez-Angeles et al. implement such a GPU-based engine[9]. Their basic data structure is an array of tuples, allowing for duplicates. After each relational operation, explicit duplicate elimination is performed, which leads to large overheads in some situations. Also, the potentially high number of duplicates occurring in temporary results quickly exceeded the memory budget on GPUs. This approach has been extended by Red Fox[41], which relies on a more sophisticated key-value store and allows more complicated queries than the earlier work. While GPU-based Datalog evaluation demonstrates significant speedups over traditional CPU-based techniques, they are limited by the amount of available memory for large-scale benchmarks. Moreover, complex use cases such as program analysis or network analysis have not been studied in GPU engines. For further performance comparisons between Datalog engines, we point the reader to[42,18].

Concurrent Tree Data Structures.

B-trees are among the most successful data structures for indexed data. The development of locking techniques for concurrent usage has been a well studied area[12]. However, most research focuses on the secondary storage use case[43]. For in-memory usage, a modified B-tree known as B-link tree managed to eliminate the need for read locks[44]. By adding a 'link' pointer to each node, pointing to the next node on the same level of the tree, Lehman et al. demonstrate that correct concurrent operation occurs even before the usual tree pointers are changed for a new split node. 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[20]. 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[45] is similar in spirit to our work with an optimistic concurrency which allows invisible readers. The approach in[46] maintains interval information to determine the placement of data. The data structure in[47] 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[48,14,49,50,51,13,52,53,54]. 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[29]

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.

8 | CONCLUSION

In this paper, we have presented a new approach to parallel Datalog evaluation that relies on a data structure framework that can incorporate multiple, specialized data structures. The framework, aside from allowing easy integration of novel data structures, represents a research platform to permit the investigation of new data structures for Datalog evaluation.

We have argued that since the appropriate choice of the data structure depends on the Datalog application, there is no need to find a single data structure for all applications and workloads. This resembles a very practical solution to fill the gap between the logic evaluation and the actual implementation of relations in the form of data structures.

To demonstrate the effectiveness of our approach we have presented two data structures that follow our framework: (1) an in-memory B-tree data structure providing an efficient base-line solution for a wide variety of relations and (2) the Brie data structure specialized for high-volume and highly dense relations. Our experiments have demonstrated that significant speed-ups (up to $6\times$) can be achieved by varying specialized data structure for given use-cases.

We will leave more detailed investigations of NUMA effects, heterogeneous architectures, and scaling beyond a single node to future work.

ACKNOWLEDGEMENT

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), the Austrian Research Promotion Agency (FFG) under contract no 868018, and a research donation from AWS.

References

1. Bravenboer M, Smaragdakis Y. Strictly Declarative Specification of Sophisticated Points-to Analyses. *SIGPLAN Not.* 2009; 44(10): 243–262.
2. Lopes NP, Bjørner N, Godefroid P, Jayaraman K, Varghese G. Checking beliefs in dynamic networks. In: ; 2015: 499–512.
3. Seo J, Park J, Shin J, Lam MS. Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis. *Proc. VLDB Endow.* 2013; 6(14): 1906–1917.
4. Brent L, Jurisevic A, Kong M, et al. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* 2018.
5. Jordan H, Subotic P, Zhao D, Scholz B. A Specialized B-tree for Concurrent Datalog Evaluation. In: PPoPP '19. ACM; 2019; New York, NY, USA.
6. Abiteboul S, Hull R, Vianu V., eds.*Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc. 1st ed. 1995.
7. Whaley J, Avots D, Carbin M, Lam MS. Using Datalog with binary decision diagrams for program analysis. In: ; 2005: 97–118.
8. Jordan H, Scholz B, Subotić P. Soufflé: On Synthesis of Program Analyzers. In: ; 2016.
9. Martınez-Angeles CA, Dutra I, Costa VS, Buenabad-Chávez J. A datalog engine for gpus. *Declarative Programming and Knowledge Management* 2014: 152–168.
10. Hoder K, Bjørner N, Moura dLM. μZ – An Efficient Engine for Fixed Points with Constraints. In: . LNCS 6806. Springer; 2011: 457-462.
11. LogicBlox I. Declartive cloud platform for applications that combine transactions & analytics. http://www.logicblox.com; .
12. Graefe G. A survey of B-tree locking techniques. *ACM Transactions on Database Systems (TODS)* 2010; 35(3): 16.
13. Oshman R, Shavit N. The SkipTrie: Low-depth Concurrent Search Without Rebalancing. In: ACM; 2013; New York, NY, USA: 23–32.

14. Prokopec A, Bronson NG, Bagwell P, Odersky M. Concurrent Tries with Efficient Non-blocking Snapshots. *SIGPLAN Not.* 2012; 47(8): 151–160.
15. Jordan H, Subotić P, Zhao D, Scholz B. Brie: A Specialized Trie for Concurrent Datalog. In: PMAM'19. ACM; 2019; New York, NY, USA: 31–40
16. Subotic P, Jordan H, Chang L, Fekete A, Scholz B. Automatic Index Selection for Large-Scale Datalog Computation. *PVLDB* 2018; 12(2): 141–153.
17. Shun J, Blelloch GE. Phase-concurrent Hash Tables for Determinism. In: SPAA '14. ACM; 2014; New York, NY, USA: 96–107.
18. Scholz B, Jordan H, Subotić P, Westmann T. On Fast Large-scale Program Analysis in Datalog. In: ACM; 2016; New York, NY, USA: 196–206.
19. BAYER R, MCCREIGHT E. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1972; 1: 173–189.
20. Karnagel T, Dementiev R, Rajwar R, et al. Improving in-memory database index performance with Intel® Transactional Synchronization Extensions. In: IEEE. ; 2014: 476–487.
21. Lameter C. Effective synchronization on Linux/NUMA systems. In: . 2005. ; 2005.
22. Menascé DA, Nakanishi T. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information systems* 1982; 7(1): 13–27.
23. Boehm HJ. Can seqlocks get along with programming language memory models?. In: ACM. ; 2012: 12–20.
24. Developers TS. Souffle – A Datalog Engine.. http://www.github.com/souffle-lang/souffle; . Accessed: 2019-01-05.
25. Inc. G. B-Tree Containers from Google. https://isocpp.org/blog/2013/02/b-tree-containers-from-google; . Accessed: 2017-02-14.
26. Reinders J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc." . 2007.
27. Sewall J, Chhugani J, Kim C, Satish N, Dubey P. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB* 2011; 4: 795-806.
28. Mao Y, Kohler E, Morris RT. Cache Craftiness for Fast Multicore Key-value Storage. In: EuroSys '12. ACM; 2012; New York, NY, USA: 183–196
29. Brown T. B-slack Trees: Space Efficient B-Trees. In: Ravi R, Gørtz IL., eds. *Algorithm Theory – SWAT 2014*Springer International Publishing; 2014; Cham: 122–133.
30. Aref M, Cate tB, Green TJ, et al. Design and implementation of the LogicBlox system. In: ACM. ; 2015: 1371–1382.
31. Shkapsky A, Yang M, Interlandi M, Chiu H, Condie T, Zaniolo C. Big data analytics with datalog queries on spark. In: ACM. ; 2016: 1135–1149.
32. Seo J, Park J, Shin J, Lam MS. Distributed socialite: a datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment* 2013; 6(14): 1906–1917.
33. Shaw M, Koutris P, Howe B, Suciu D. Optimizing Large-scale Semi-Naïve Datalog Evaluation in Hadoop. In: Springer-Verlag; 2012; Berlin, Heidelberg: 165–176.
34. Hulin G. Parallel Processing of Recursive Queries in Distributed Architectures. In: Morgan Kaufmann Publishers Inc.; 1989; San Francisco, CA, USA: 87–96.
35. Cohen S, Wolfson O. Why a Single Parallelization Strategy is Not Enough in Knowledge Bases. In: ACM; 1989; New York, NY, USA: 200–216.
36. Ganguly S, Silberschatz A, Tsur S. A Framework for the Parallel Processing of Datalog Queries. In: ACM; 1990; New York, NY, USA: 143–152.
37. Seib J, Lausen G. Parallelizing Datalog Programs by Generalized Pivoting. In: ACM; 1991; New York, NY, USA: 241–251.
38. Wolfson O, Silberschatz A. Distributed Processing of Logic Programs. *SIGMOD Rec.* 1988; 17(3): 329–336.
39. Wolfson O, Ozeri A. A New Paradigm for Parallel and Distributed Rule-processing. *SIGMOD Rec.* 1990; 19(2): 133–142.
40. Yang M, Shkapsky A, Zaniolo C. Scaling up the performance of more powerful Datalog systems on multicore machines. *VLDB J.* 2017; 26(2): 229–248.

41. Wu H, Diamos G, Sheard T, et al. Red fox: An execution environment for relational query processing on gpus. In: ACM. ; 2014: 44.

42. Antoniadis T, Triantafyllou K, Smaragdakis Y. Porting Doop to Souffle: A Tale of Inter-engine Portability for Datalog-based Analyses. In: ACM; 2017; New York, NY, USA: 25–30.

43. Kung HT, Robinson JT. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 1981; 6(2): 213–226.

44. Lehman PL, Yao sB. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. Database Syst.* 1981; 6(4): 650–670.

45. Bronson NG, Casper J, Chafi H, Olukotun K. A Practical Concurrent Binary Search Tree. *SIGPLAN Not.* 2010; 45(5): 257–268.

46. Drachsler D, Vechev M, Yahav E. Practical Concurrent Binary Search Trees via Logical Ordering. *SIGPLAN Not.* 2014; 49(8): 343–356.

47. Howley SV, Jones J. A Non-blocking Internal Binary Search Tree. In: ACM; 2012; New York, NY, USA: 161–171.

48. Brown T, Helga J. Non-blocking K-ary Search Trees. In: Springer-Verlag; 2011; Berlin, Heidelberg: 207–221.

49. Braginsky A, Petrank E. A Lock-free B+Tree. In: ACM; 2012; New York, NY, USA: 58–67.

50. Crain T, Gramoli V, Raynal M. A Speculation-friendly Binary Search Tree. In: ACM; 2012; New York, NY, USA: 161–170.

51. Levandoski JJ, Lomet DB, Sengupta S. The Bw-Tree: A B-tree for New Hardware Platforms. In: IEEE Computer Society; 2013; Washington, DC, USA: 302–313.

52. Arbel M, Attiya H. Concurrent Updates with RCU: Search Tree As an Example. In: ACM; 2014; New York, NY, USA: 196–205.

53. Natarajan A, Mittal N. Fast Concurrent Lock-free Binary Search Trees. In: ACM; 2014; New York, NY, USA: 317–328.

54. Basin D, Bortnikov E, Braginsky A, et al. KiWi: A Key-Value Map for Scalable Real-Time Analytics. *SIGPLAN Not.* 2017; 52(8): 357–369.