Brie: A Specialized Trie for Concurrent Datalog

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

Pavle Subotić
Amazon
UK
psubotic@amazon.com

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

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

Abstract

Modern Datalog engines are employed in industrial applications such as graph databases, networks, and static program analysis. To cope with the vast amount of data in these applications, Datalog engines must employ specialized parallel data structures. In this paper, we introduce the Brie, a specialized data structure for high-density relations storing large data volumes. It effectively compresses dense data in a lock-free fashion and obtains up to 15× higher performance in parallel insertion benchmarks compared to state-of-the-art alternatives. Furthermore, when integrated into a Datalog engine running an industrial points-to analysis, runtime improves by a factor of 4× with a compression ratio of up to 3.6× are obtained.

Keywords Datalog, Data Structures, Concurrency

1 Introduction

Modern Datalog engines are increasingly employed for large-scale data processing tasks, including graph databases querying [32], network verification [14] and program analysis [19] among many others. Due to the often *giga-scale* sized data sets inherent in these application domains, a scalable, high-performance implementation of a Datalog engine is paramount.

A vital ingredient for achieving high-performance in Datalog engines is a *parallel evaluation*, i.e., the ability to compute logical evaluation operations concurrently in separate threads of a shared memory system. Designing an effective parallel evaluation strategy, however, is not limited to the evaluation algorithm alone. The choice of underlying *parallel data structure* for modeling relations is crucial for achieving high performance [20, 34]. While previous techniques have argued for various data structures, based on best *average case* performance [20, 36], in practice, achieving peak performance requires the use of specialized, highly tuned data structures that exploit particular characteristics for a given workload.

To incorporate a wide variety of specialized data structures for different workloads, we have implemented a data structure interface for relations in Soufflé [19]. Instead of relying on a single relational data structure, Soufflé can utilize a wide-range of data structures and can choose a data structure for a relation that is most suitable for achieving peak performance.

In this paper, we introduce a specialized concurrent data structure for Soufflé called the Brie, which has been designed for *high-density* relations with a *large data volume*. Logical 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 [15] and trie data structures [27, 28]. The Brie has several key advantages over both B-trees and tries. 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. For example, in our industrial use case, the Brie uses on average 2× less space than a highly optimized B-tree. Bries also do not require complex locking mechanisms, because they do not re-organize the tree data structure by balancing data. Unlike the trie, the Brie has a high degree of cache friendliness. However, note that Bries only perform well for particular workloads exhibiting relations with high-density data.

We have implemented the Brie data structure in Soufflé to store relations. We evaluated Brie's performance on parallel micro-benchmarks and an industrial program analysis benchmark, i.e., a points-to analysis for OpenJDK. On micro benchmarks, the Brie is up to 15× faster than state-of-the-art data structures. When used in Soufflé for a points-to analysis, the Brie is 4× faster than B-trees.

PMAM'19 , February 17, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6290-0/19/02…$15.00
https://doi.org/10.1145/3303084.3309490

The contributions of this paper are summarised as follows:

- We outline the Soufflé parallel evaluation strategy – an implementation agnostic data structure framework that allows seamless integration of specialized data structures.
- We describe the Brie data structure, which is a concurrent lock-free data structure for storing relations, and its integration into Soufflé.
- We evaluate the Brie compared to other state-of-the-art data structures on both micro-benchmarks and an industrial sourced benchmark.

The remainder of the paper is organized as follows: In Section 2 we describe the parallel evaluation strategy of Soufflé and the Soufflé data structure framework. In Section 3 we introduce the Brie data structure. In Section 4 we evaluate the Brie data structure. In Section 5 we present related work where we compare the Brie to existing state-of-the-art data structures, and we conclude in Section 6.

2 Parallel Datalog Evaluation

In this section, we describe the parallel execution strategy and the data structure framework of Soufflé. Soufflé synthesizes a parallel C++ program from a Datalog program [19]. A Datalog program consists of a set of facts, called *input relations* (given as data or in code) and logical rules that compute additional derived relations, called *output relations*. For example, let *parent* be a binary input relation. The two logical rules

$$sg(X, Y) \text{ :- } parent(P, X), parent(P, Y). \quad (1)$$

$$sg(X, Y) \text{ :- } sg(P1, P2), parent(P1, X), parent(P2, Y). \quad (2)$$

implicitly define the content of the output relation *sg* computing the set of individuals in the same generation of, e.g., a family tree defined by the *parent* relation. Entries like *parent*(*Marge*, *Bart*) model direct parent-child relations. The first rule computes all pairs (X, Y) sharing an arbitrary common parent P, and the result is stored in relation *sg*. The second rule is a recursive rule that computes pairs inductively, i.e., if $P1$ is a parent of X, and $P2$ is a parent of Y, where $P1$ is in the same generation as $P2$, then it follows that X must be in the same generation as Y. Thus, the entry (X, Y) shall be added to *sg*. While this is a simple example, real-world use-cases comprise hundreds of relations, connected through hundreds of (potentially mutually recursive) rules.

For the given example, Soufflé [19, 30] produces OpenMP C++ code similar to the simplified C++ code shown in Figure 1. The C++ code outlines the iterative semi-naïve [1] evaluation of a Datalog program, computing a least fixed point. STL data structures are used to store relations. The evaluation proceeds in two stages, the first from line 5 to line 13 evaluates rule 1, and the second from line 15 to line 34 evaluates rule 2.

```
1  using Tuple = array<size_t,2>;
2  using Relation = set<Tuple>;
3  Relation evaluate(const Relation &parent){
4   Relation sg;
5   // sg(X,Y) :- parent(P,X), parent(P,Y).
6   for (const auto &t1: parent) {
7    auto l = parent.lower_bound({t1[0],0});
8    auto u = parent.upper_bound({t1[0]+1,0});
9    for (auto it=l; it!=u; ++it) {
10    Tuple t2({t1[1], (*it)[1]});
11    sg.insert(t2);
12   }
13  }
14
15  // sg(X,Y) :- sg(P1,P2), parent(P1,X), parent(P2,Y).
16  Relation deltaSg = sg;
17  while(!deltaSg.empty()){
18   Relation newSg;
19   for (const auto &t1: deltaSg){
20    auto l1 = parent.lower_bound({t1[0],0});
21    auto u1 = parent.upper_bound({t1[0]+1,0});
22    for (auto it1=l1; it1!=u1; ++it1) {
23     auto l2 = parent.lower_bound({t1[1],0});
24     auto u2 = parent.upper_bound({t1[1]+1,0});
25     for (auto it2=l2; it2!=u2; ++it2) {
26      Tuple t2({(*it1)[1], (*it2)[1]});
27      if (sg.find(t2) == sg.end())
28       newSg.insert(t2);
29     } // end of for it2
30    } // end of for it1
31   } // end of for deltaSg
32   sg.insert(newSg.begin(), newSg.end());
33   deltaSg.swap(newSg);
34  } // end of while
35  return sg;
36 }
```

Figure 1. Synthesised sequential C++ code for Same Generation Example using STL sets

In the first stage, we iterate over the *parent* relation (line 6). For each tuple $t1 \equiv (P, X)$ in the *parent* relation, we iterate over the subset of *parent* matching the first element (lines 7 to 9), finding tuples (P, Y). Assuming that *parent* is ordered, we can efficiently find this subset of tuples via a range traversal between an upper and lower bound in log-linear time. Finally, we take the second element of each tuple, create $t2 \equiv (X, Y)$, and insert it into the *sg* relation (line 11).

In the second stage, the rule to be processed is recursive. According to the semi-naïve algorithm, auxiliary relations *deltaSg* and *newSg* are created, to store the new tuples generated in the previous iteration and current iteration respectively. We first iterate over *deltaSg* (line 19), finding tuples $t1 \equiv (P1, P2)$. Then, we iterate over *parent* twice (lines 20 to 22, and lines 23 to 25), to find tuples $(P1, X)$ and $(P2, Y)$ matching $t1$. Finally, the tuple $t2 \equiv (X, Y)$ is inserted into *newSg* (line 28).

Except for the insertion operations on line 11 and 28, all operations within the nested for loop are read-only operations or targeting non-shared memory locations. Thus, if a data structure provides an efficiently synchronized insertion operation, the parallel potential of Datalog query processing could be harnessed by merely parallelizing the for loops on line 6 and 19.

Since Datalog evaluation operates on sets of tuples, it is important to choose a suitable set data structure. A candidate data structure 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.
- *empty*() determines whether the set is empty.

However, there is no universal best data structure to store relations, and therefore we provide a framework for incorporating portfolios of data structures. Any data structure using this framework should implement the interface defined above. Generally, elements in the set should also be sorted to allow efficient range queries to be performed when tuples require filtering, however, this is not mandatory.

It is important to note that in a concurrent setting, seminaïve evaluation will either have (1) multiple writers to a relation, but no reads, or (2) multiple reads from a relation, but no writes. For example, in Figure 1, the nested loops in lines 19 to 31 read from relations *deltaSg* and *parent*, and write to relation *newSg*. Therefore, no relation is read from and written to at the same time, and as a result, read operations do not need to be synchronized. However, the synchronization of writes is critical. Concurrent write throughput of the data structure will significantly impact performance, since vast amounts of tuples may be produced in the nested loops.

3 A Specialized Trie for Datalog

To improve the performance of Soufflé for dense workloads, we have designed a data structure combining lock-free, efficient inserts and cache friendly range queries. We refer to this data structure as a *Brie* – for being structurally based on a trie and utilizing a node-blocking scheme like the one found internally in B-trees.

Figure 2 illustrates the overall structure of a Brie. A fixed-height trie provides the foundation of a Brie. Thus, stored tuples are encoded within the edges, instead of placing them into nodes as is the case for B-trees or classic binary search trees. Each inner node maintains a map linking integers (the components of the tuples to be stored) to child node pointers. This map is based on the principles of quadtrees combined

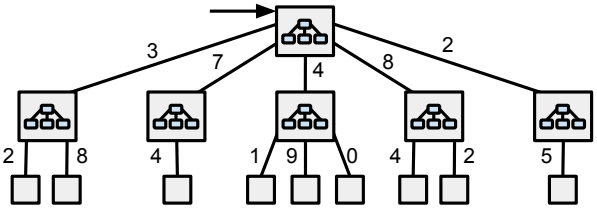


Figure 2. Overview on the structure of a Brie.

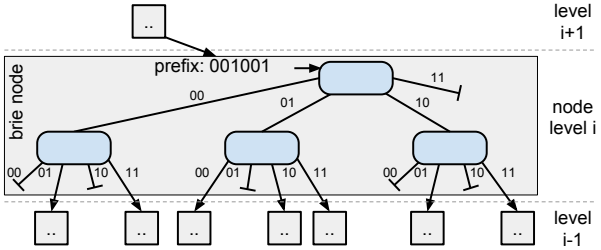


Figure 3. Internal structure of an inner node of a Brie.

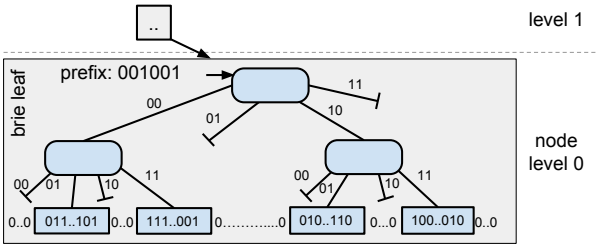


Figure 4. Internal structure of a leaf node of a Brie.

with the node blocking feature of B-trees. Furthermore, on the leaf node level, bit-masks are utilized for compressing the information of the presence or absence of values into single bits.

Each inner node has to maintain a mapping between integers and child node pointers. Figure 3 illustrates the structure utilized for this purpose. Within every inner node of the Brie, a blocked quadtree realizing a map of integers to sub-brie-tree pointers is maintained. The map's keys are stored along the path from a root node to its leaf nodes, where the associated child node pointers are stored. However, multiple levels of the inner quadtree are collapsed into single nodes for improved cache efficiency. While Figure 3 illustrates the collapse of two levels for clarity, the actual implementation collapses a configurable number of levels into a single node. For the experiments covered in this paper, an empirically determined optimal value of 6 levels has been selected.

Branches that do not contain any child node pointers are pruned. Furthermore, the root is formed by the first node exhibiting more than a single child. Thus, 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. During insert operations, this prefix is shortened as necessary.

Figure 4 outlines the structure of leaf level nodes in the Brie. 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 $\log_2(W)$ levels of the index structure are collapsed into a single machine word W bits wide.

The Brie structure provides efficient operations for insertions, membership tests, and range queries for processing Datalog queries. Furthermore, insert operations can be conducted lock-free utilizing atomic operators for setting individual bits in the leaf node level or exchanging pointers in inner node levels, as outlined by Algorithm 1.

The *insert* procedure inserts new elements into a Brie by using the *getLeaf* procedure to navigate within nodes over nested quadtrees. Non-existing nodes are inserted by the *ensureExistence* procedure (line 17) while navigating the Brie in a top-down fashion. Furthermore, the *ensurePrefixCovered* function is utilized to adapt maintained common prefixes and grow nested quadtrees in height whenever necessary (line 20). Details of the latter function are covered in Algorithm 2.

For every new value to be inserted in an inner quadtree, the shared prefix is iteratively shortened by 6 bits until it also covers the new value. In every step, a new root node is added to the local nested quadtree and its height is increased by one. The tree thus grows bottom up, in a similar fashion to B-trees. Function *raiseLevel* realizes the growing step. However, items in the Brie are never moved around - their position is kept static.

To synchronize insertion operations, three critical regions need to be protected: (1) the insertion of a new Brie node, (2) the insertion of a bit in a Brie leaf node, and (3) the raising of a nested quadtree.

The first of those is realized by line 6 of Algorithm 1. In this step, a pointer referencing an empty subtree is replaced by a pointer to a newly created subtree. The insertion is protected by a compare-and-swap operation (CAS), ensuring only one subtree is ever to be inserted at a given position into the tree. In case of a collision, where two threads attempt to insert two different subtrees at the same location into the Brie, one thread will succeed while the other will notice that another subtree has been inserted. The latter will then discard its temporal subtree and continue using the already inserted structure.

The second operation requiring synchronization among threads is the insertion of bits to mark the presence of tuples in leaf nodes of the Brie. This operation is covered by line 39 of Algorithm 1. A simple atomic *boolean or* operator is sufficient to ensure correct updates. Conflicting updates do not have to be detected, since update effects are implicitly aggregated and no cleanup operations are necessary.

The third critical region is given by the growing of the btree-node local quadtrees. Its implementation is given by the function *raiseLevel* in Algorithm 2. Conceptually, its synchronization follows the CAS approach. However, unlike the previous cases, the value of the memory location to be updated is not a simple scalar, but a small struct maintaining the shared prefix information. This information comprises the common prefix, its length, and the root node pointer. All this information is retrieved atomically in line 12, used for computing a new prefix, prefix length, and root node,

Algorithm 1 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
```

Algorithm 2 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
```

and compare-and-swapped in as a replacement in line 28. As for the insertion of a Brie node, concurrent updates are detected and temporary values discarded. 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, offers according atomic utilities to access those features or functionally equivalent substitutes.

While for updates (1) and (3) conflicts are expensive, the probability of such cases occuring is low. Typical work load patterns will direct threads to insert elements in distinct parts of a Brie, thus avoiding 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.

3.1 Discussion

Compared to conventional B-trees, Bries offer a number of benefits. In particular, insertion operations are computationally cheaper, since navigation steps do not depend on search operations over ranges of keys. Instead, in each step the following point can be directly addressed. Additionally, insert operations do not require insertions into ordered arrays, necessitating the movement of lists of keys as it would be the case for B-trees.

On the memory usage side, 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 demonstrated in the evaluation section, the efficiency of this compression capability depends heavily on the correlation of entries to be stored. If common prefixes are sparse, memory utilization may be significantly higher than is the case with B-trees or other data structures.

Besides the Brie's more efficient lookup and insertion operations, the Brie insertion operation can also be effectively synchronized for concurrent access, as outlined above. The presented synchronization scheme 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.

4 Evaluation

In this section, we evaluate our Brie data structure in comparison to other data structures. Our Brie data-structure is implemented in the open-source Soufflé engine [12] (cf. `src/Brie.h`). The outcome of our evaluations is to validate the following claims.

- **Claim-I:** Our Brie data structure uses less memory than storing the data directly for high-density data.
- **Claim-II:** The sequential performance of our Brie data structure outperforms state-of-the-art data structures for high-density data.
- **Claim-III:** The parallel performance of our Brie data structure outperforms state-of-the-art data structures for high-density data.
- **Claim-IV:** Bries outperform state-of-the-art B-tree data structure when being used for maintaining relation data in Soufflé [19] for an industrial benchmark, i.e., performing a simple points-to analysis for OpenJDK with 7 MLOCs.

The Brie data structure has been implemented using C++ and integrated as a data structure representing relations in the Soufflé Datalog compiler [19]. Synchronization operations in our Brie implementation are based on C++'s concurrency memory model [5] and its atomic value wrappers. They are thus implemented to be portable among different architectures.

The performance characteristic of Brie is evaluated through a set of benchmarks, evaluating the execution time of the most frequently used operators within the Soufflé runtime. Furthermore, we evaluate the memory requirements of our structures, since memory constraints are commonly the limiting factor for scaling up analyses.

Besides our Brie implementation (denoted as *brie*), we include additional reference data structures in our evaluation. For the evaluation of memory consumption and sequential performance we include:

- C++'s *std::set* 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 *std::hash_set*, providing theoretically superior insertion performance of $O(1)$, but no efficient support for range queries
- a state-of-the-art B-tree implementation provided by Google [18], denoted as *google btree*
- a sequential version of Soufflé 's high-performance concurrent B-tree [20], denoted as *sequential btree*

For the evaluation of parallel operations we include:

- a parallel version of Soufflé 's high-performance concurrent B-tree implementation with optimistic locking [20], denoted as *optimistic btree*
- the *concurrent_unordered_set* implementation of Intel's TBB library version 2017_U7 [29] denoted as *tbb::hash_set*, representing an industry standard, state-of-the-art concurrent set implementation; since it is hash based, no efficient range queries are supported

The 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[1]) equipped with 256GB memory using GCC 5.4.0 with -O3 optimization. GCC's OpenMP implementation is used as the underlying runtime system, threads are pinned to cores, and sockets are filled before threads are assigned to additional sockets. All sources are publicly available[2].

4.1 Memory Requirements

The first aspect to be investigated in order to determine the suitability of the Brie data structure for representing relations within a Datalog query processor is its memory consumption. In theory, our data structure exhibits $O(N)$ memory requirements. However, the memory usage of the Brie depends on the *density* of those entries.

For instance, 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.

[1]we ignore hyper-threading capabilities in our evaluation for brevity
[2]https://tinyurl.com/ycb3gqvc (available on github, anonymised for review)

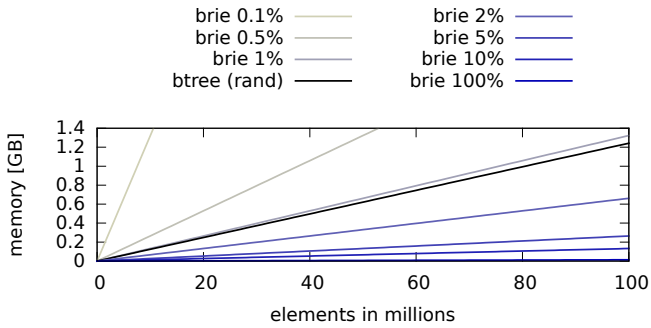


Figure 5. B-tree vs. Brie with different point densities (10.000 samples each)

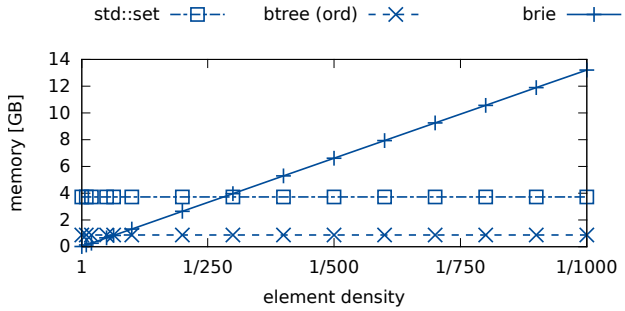


Figure 6. Memory requirements for 100M points depending on data point density.

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

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

To evaluate the memory requirements of the Brie data structure, the density of data points needs to be considered.

The memory requirements of Bries filled with data of various densities are illustrated by Figure 5. Results labeled "brie X%" are obtained by maintaining a data density of X%. The results show the linear correlation between the number of contained elements and the total memory consumption. Furthermore, it can be observed that a higher data point density leads to a higher efficiency of the Brie structure. For a sufficiently high-density the Brie provides higher storage efficiency compared to the B-tree, while for lower densities B-trees (vastly) outperform Bries.

To quantify the impact of the data density on the memory requirements of a Brie, Figure 6 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.

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.

4.2 Sequential Performance

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

For evaluating the performance of the insert operation, we insert varying numbers of elements into initially empty sets and measure the number of inserts per second. 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 Bries towards data point densities, point sets with varying densities are inserted into Brie structures.

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 a random sequence. The number of queries processed per second is recorded.

Finally, for the evaluation of the range query operation, we focus on the cost of scanning (or 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 elements traversable in each second.

Figure 7 summarizes the measured performance for all three operators. As in the previous experimental setup, "brie X%" denotes the utilization of the Brie structure with a point set exhibiting a density of X%. Each evaluation has been repeated at least three times, generally yielding neglectable variations in processing time, and thus throughput. Thus, variance indicators have been omitted for clarity.

The results show the nearly linear scaling of ordered inserts on the investigated B-trees and Brie configurations (Figure 7a), compared to a gradual performance degradation observed for growing set sizes for randomly ordered inserts (Figure 7b). This is due to the higher probability of cache misses when inserting elements into data structures exceedingly outgrowing available cache sizes. A similar effect is observed in the random query case, while the scans are not affected — due to their in-order memory operations.

Given sufficiently high-density, the results demonstrate the speedup gained by the Brie structure over the B-tree – between 5-17x faster for insert and query operations. In those cases, Bries even outperform hash-based set implementations exhibiting theoretically superior runtime complexity. For range scans, however, the need of decoding the bit-encoded values in the Brie increases the scan time by roughly a factor of two, compared to Soufflé 's B-tree, bringing the Brie on par with Google's B-tree.

4.3 Parallel Performance

To evaluate the parallel insert performance, we evaluate the parallel scalability of our contestants. We insert 100M 2D points into an initially empty set using a varying number of threads. Figure 8 summarizes the obtained results for all our contestants when gradually scaling the computation up to 4×8 cores, and thus beyond the boundaries of a single socket.

As we can observe, TBB's concurrent set implementation experiences performance penalties for each additional socket getting involved in the computation.

To the contrary, while Soufflé 's optimistic B-tree does suffer from crossing socket boundaries, it keeps benefiting from additional parallel resources.

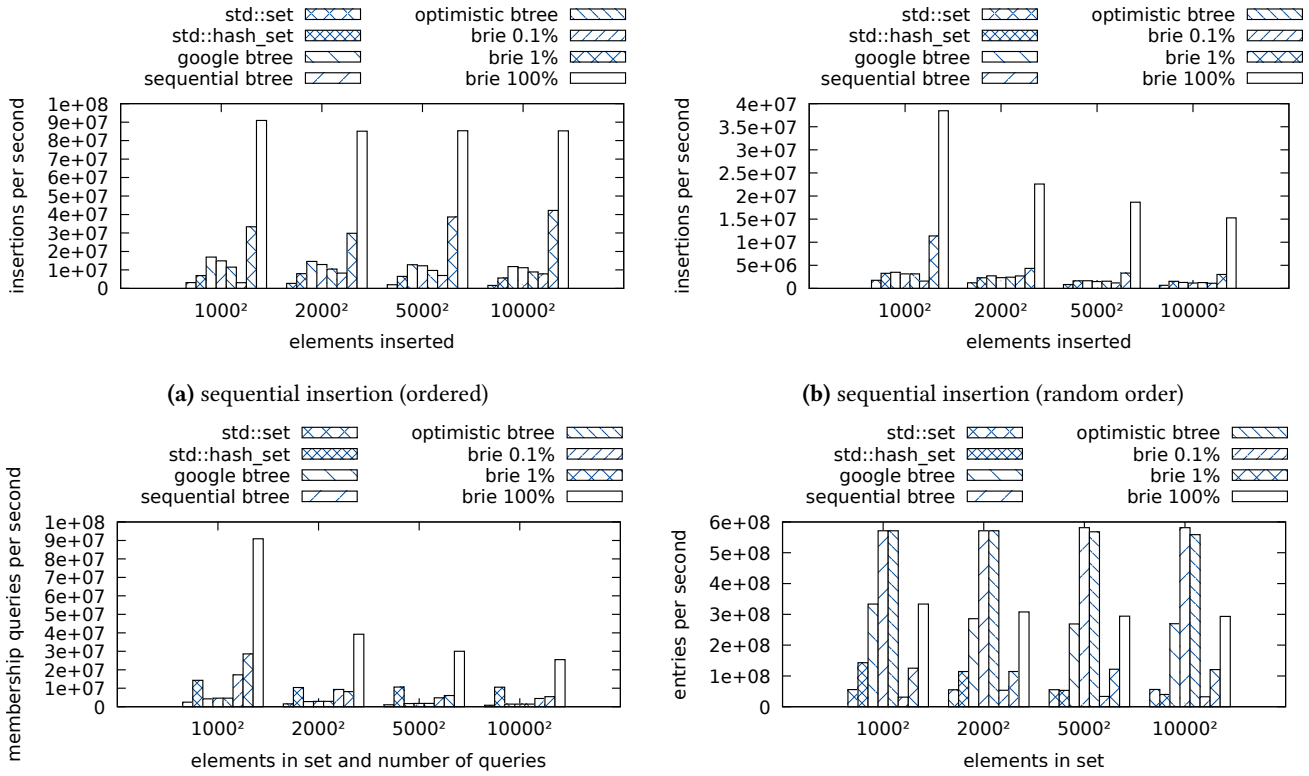
Figure 8 also illustrates the parallel scalability of Bries being filled with sets of various densities. For sufficiently dense data sets, Bries outperform B-trees. In those cases Bries reach a parallel efficiency of up to 80% for 32 cores. However, when becoming too sparse, the high memory usage and the associated memory management overhead and low data locality result in a loss of efficiency.

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 350x faster than TBB's concurrent set.

4.4 Processing Datalog Queries

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 9 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



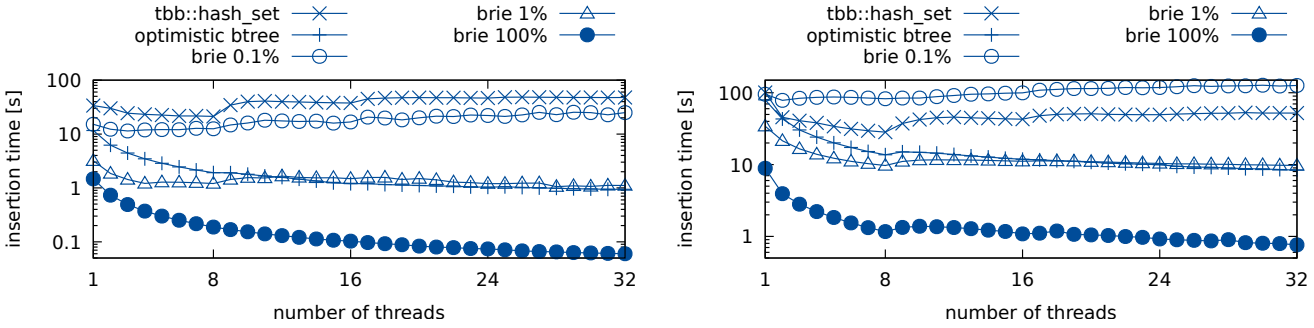
(a) sequential insertion (ordered)

(b) sequential insertion (random order)

(c) membership test (random order)

(d) full-range scan

Figure 7. Sequential performance of performance critical set operations.



(a) ordered insertion

(b) random order insertion

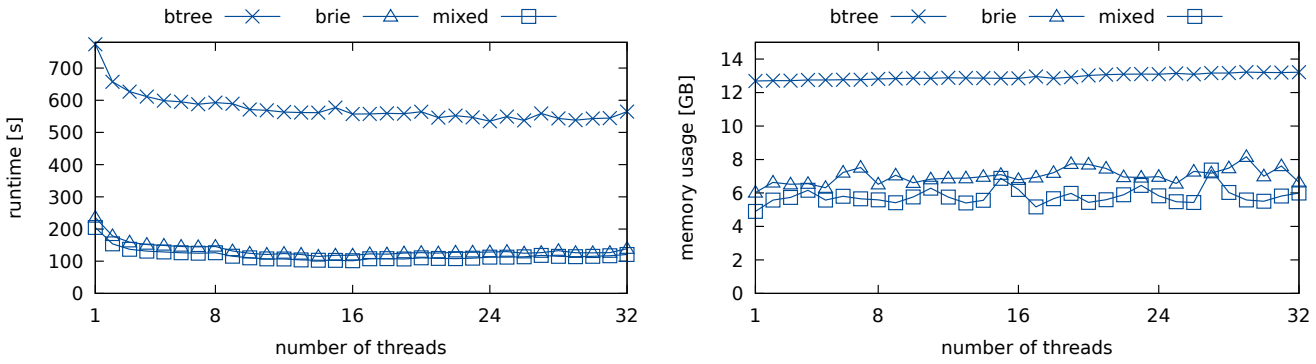
Figure 8. Comparison of insertion times of 100M 2D points, using ideal density for each data structure.

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 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.

5 Related Work

Data Structures for Datalog. Previous Datalog implementations have focused on various kinds of alternative data



(a) runtime **(b)** memory

Figure 9. 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.

structures including binary decision diagrams [37], Hash-sets, e.g., [16, 32] and B-trees, e.g., [7, 19]. In our experience, while B-trees (as implemented in PA-Datalog / Logicblox ver. 3 and Soufflé) have shown to be the most scalable for large ruleset/dataset benchmarks [19], certain use cases can be enhanced by the use of specialized data structures such as Bries. 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 [35].

Parallel Datalog Engines. There has been a multitude of parallelization efforts of Datalog in the past, e.g., see [10, 31, 33, 38] mainly focusing on rewriting techniques and top-down evaluations. Recently a number of state-of-the-art engines have employed fine-grain parallelism to bottom-up evaluation schemes. The work in [39] uses an in-memory parallel evaluation of Datalog programs on shared-memory multi-core machines. Datalog-MC hash-partitions tables and executes the partitions on cores of a shared-memory multi-core system using a variant of hash-join. To parallel evaluate Datalog, Datalog rules are represented as and-or trees that are compiled to Java. Logicblox version 4, uses persistent functional data structures that avoid the need for synchronization by virtue of their immutability, where insertions efficiently replicate state via the persistent data structure. A particular performance-focused approach has been proposed by Martınez-Angeles et al. who implemented a Datalog engine running on GPUs [25]. Their basic data structure is an array of tuples, allowing for duplicates. Thus, after every relational operation, explicit duplicate elimination is performed – which for some cases vastly dominates execution time. Also, the potentially high number of duplicates occurring in temporary results quickly exceeded the memory budget on GPUs. The applicability of this approach has only been demonstrated for small Datalog queries. We point the reader to [2, 30] for performance comparisons between engines on large ruleset/dataset benchmarks.

Concurrent Tree Data Structures. B-trees themselves are among the most successful data structures for indexed data. Lots of research effort has been addressing locking techniques [15]. However, most of these focus on the secondary storage use case [22]. For in-memory usage, a modified B-tree known as B-link tree managed to eliminate the need for read locks [23]. Unfortunately, we have not been able to obtain an implementation for comparison. An alternative approach has achieved good results by utilizing hardware transactional memory features available on some recent Intel architectures for synchronizing B-tree insert operations [21]. 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 [8] is similar in spirit to our work with an optimistic concurrency which allows invisible readers. The approach in [13] maintains interval information to determine the placement of data. The data structure in [17] 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 [3, 4, 6, 9, 11, 24, 26–28].

6 Conclusion

In this paper, we argue for the need for various relational data-structures in the evaluation of Datalog programs. We introduced the Brie data structure and have shown that for high-density use cases, parallel insertion performance can exceed the performance of state-of-the-art B-tree implementations by a factor of 15. Integrated into a Datalog engine, Bries can out-perform best average case performing B-tree data structures by a factor of 4, as demonstrated through an industrial real-world use case.

Acknowledgments

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

References

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

[2] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting Doop to Soufflé: A Tale of Inter-engine Portability for Datalog-based Analyses. In *SOAP*. ACM, New York, NY, USA, 25–30.

[3] Maya Arbel and Hagit Attiya. 2014. Concurrent Updates with RCU: Search Tree As an Example. In *PODC*. ACM, New York, NY, USA, 196–205.

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

[5] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 68–78.

[6] Anastasia Braginsky and Erez Petrank. 2012. A Lock-free B+Tree. In *SPAA*. ACM, New York, NY, USA, 58–67.

[7] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception analysis and points-to analysis: better together. In *ISSTA*. ACM, New York, NY, USA, 1–12.

[8] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. *SIGPLAN Not.* 45, 5 (Jan. 2010), 257–268.

[9] Trevor Brown and Joanna Helga. 2011. Non-blocking K-ary Search Trees. In *OPODIS*. Springer-Verlag, Berlin, Heidelberg, 207–221.

[10] S. Cohen and O. Wolfson. 1989. Why a Single Parallelization Strategy is Not Enough in Knowledge Bases. In *PODS*. ACM, New York, NY, USA, 200–216.

[11] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2012. A Speculation-friendly Binary Search Tree. In *PPoPP*. ACM, New York, NY, USA, 161–170.

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

[13] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical Concurrent Binary Search Trees via Logical Ordering. *SIGPLAN Not.* 49, 8 (Feb. 2014), 343–356.

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

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

[16] Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. 2011. *Z*- An Efficient Engine for Fixed Points with Constraints. In *CAV (Lecture Notes in Computer Science)*, Vol. 6806. Springer, 457–462.

[17] Shane V. Howley and Jeremy Jones. 2012. A Non-blocking Internal Binary Search Tree. In *SPAA*. ACM, New York, NY, USA, 161–171.

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

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

[20] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. A Specialized B-tree for Concurrent Datalog Evaluation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA.

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

[22] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226.

[23] Philip L. Lehman and s. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 650–670.

[24] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*. IEEE Computer Society, Washington, DC, USA, 302–313.

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

[26] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *PPoPP*. ACM, New York, NY, USA, 317–328.

[27] Rotem Oshman and Nir Shavit. 2013. The SkipTrie: Low-depth Concurrent Search Without Rebalancing. In *PODC*. ACM, New York, NY, USA, 23–32.

[28] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. *SIGPLAN Not.* 47, 8 (Feb. 2012), 151–160.

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

[30] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *CC*. ACM, New York, NY, USA, 196–206.

[31] Jürgen Seib and Georg Lausen. 1991. Parallelizing Datalog Programs by Generalized Pivoting. In *PODS*. ACM, New York, NY, USA, 241–251.

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

[33] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. 2012. Optimizing Large-scale Semi-Naïve Datalog Evaluation in Hadoop. In *Datalog 2.0*. Springer-Verlag, Berlin, Heidelberg, 165–176.

[34] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. 2013. Graph Queries in a Next-generation Datalog System. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1258–1261.

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

[36] Pavle Subotic, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. *PVLDB* 12, 2 (2018), 141–153.

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

[38] Ouri Wolfson and Avi Silberschatz. 1988. Distributed Processing of Logic Programs. *SIGMOD Rec.* 17, 3 (June 1988), 329–336.

[39] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. 2017. Scaling up the performance of more powerful Datalog systems on multicore machines. *VLDB J.* 26, 2 (2017), 229–248.