Fast Parallel Equivalence Relations in a Datalog Compiler

Patrick Nappa
The University of Sydney
Sydney, Australia
patricknappa@gmail.com

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

Pavle Subotić
Amazon
London, United Kingdom
psubotic@gmail.com

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

***Abstract*—Modern parallelizing Datalog compilers are employed in industrial applications such as networking and static program analysis. These applications regularly reason about equivalences, e.g., computing bitcoin user groups, fast points-to analyses, and optimal network routes. State-of-the-art Datalog engines represent equivalence relations verbatim by enumerating all possible pairs in an equivalence class. This approach inhibits scalability for large datasets.**

In this paper, we introduce EQREL, a specialized parallel union-find data structure for scalable equivalence relations, and its integration into a Datalog compiler. Our data structure provides a quadratic worst-case speed-up and space improvement. We demonstrate the efficacy of our data structure in SOUFFLÉ, which is a Datalog compiler that synthesizes parallel C++ code. We use real-world benchmarks and show that the new data structure scales on shared-memory multi-core architectures storing up to a half-billion pairs for a static program analysis scenario.

***Keywords*-Parallel Data Structures, Equivalence Relation, Datalog Compiler, Semi-naïve Evaluation**

I. INTRODUCTION

Parallelizing Datalog compilers [16], [42] for shared-memory multi-core computers have seen a proliferation in large-scale applications including static program analysis [27], program security [25], program optimizations [23], cloud computing [2], and networking [24]. Datalog provides a succinct logic representation for the application semantics, enabling users to rapidly build and prototype scientific/industrial large-scale applications.

Recently, modern state-of-the-art Datalog engines, such as SOUFFLÉ [16], have demonstrated performance on a par with hand-crafted tools, while maintaining the ease of use and rapid-prototyping capabilities of the Datalog language. SOUFFLÉ achieves this by utilizing parallel evaluation through specialized parallel data structures [18], [17].

Many common applications of Datalog encode notions of equivalence relations. An equivalence relation R is a binary relation which is *reflexive* (if x is in the domain of R, then $(x, x) \in R$), *symmetric* (if $(x, y) \in R$, then $(y, x) \in R$), and transitive (if $(x, y), (y, z) \in R$, then $(x, z) \in R$). For example, for a Bitcoin user group analysis [29], the relation encoding whether two users are the same users is an equivalence relation. Examples of equivalence relations also appear in points-to analyses [34], [19], SCCs for graph analyses [36], and optimal network routes [37], all of which are well-suited to implementation in Datalog.

However, current state-of-the-art Datalog engines require equivalence relations to be expressed *explicitly* using default data structures [4]. This explicit encoding of equivalence relations may incur up to a quadratic overhead compared to an optimized solution, often causing bottlenecks in the Datalog evaluation.

At the same time, the union-find [9], [20] data structure appears to be well suited for handling equivalence relations. Union-find is typically implemented as a disjoint forest of trees. Each tree represents a single equivalence class and a tree node represents an element in the equivalence relation. Storing an equivalence relation in a union-find data structure requires only a linear amount of space in the number of elements, far improving on the quadratic blow-up of an explicit representation. A union-find data structure becomes *self-computing* because of the implicit computations of rules for reflexivity, symmetry and transitivity. For example, inserting the equivalence pairs $(1, 2)$ and $(2, 3)$ into an empty union-find data structure implicitly introduces additional pairs such as $(1, 1)$ by reflexivity, $(2, 1)$ by symmetry, and $(1, 3)$ by transitivity. Although this data structure shows a lot of promise, a union-find data structure cannot easily be adopted for Datalog engines, as several key factors hamper their integration.

1) The semi-naïve evaluation strategy, which is the de-facto standard for modern Datalog engines [1], cannot accommodate self-computing data structures and must be adapted.
2) The typical disjoint forest implementation of the union-find data structure has the problem that it assumes a dense and a priori known domain, where elements are values from 1 to n and n is fixed. This is an obstacle for practical use in a Datalog engine because the domain of any relation may contain a set of arbitrary values and may grow throughout evaluation. Therefore, an on-the-fly densification (i.e. finding a unique mapping between the domain elements and numbers from 1 to n) is required so that efficient union-find data structure can be implemented.
3) In order to integrate the data structure into the Datalog evaluation algorithm, we need data structures to expose

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00015

IEEE computer society

an interface mimicking that of an ordinary relation. Therefore, we must provide an abstraction of the underlying union-find data structure to enable its use in Datalog evaluation.

4) The final challenge is effective parallelization of the union-find data structure.

In this paper, we present EQREL, a novel parallel *equivalence relation* data structure designed for parallelizing Datalog compilers. Our data structure allows the *implicit* storage and self-computation of equivalence relations, incurring only a linear amount of storage compared to the quadratic blow-up of an explicit representation. For a domain D of n elements, an equivalence relation may have up to n^2 pairs. Therefore, for an explicit representation, up to $\mathcal{O}(n^2)$ space is required, while EQREL stores each *element* once, using only $\mathcal{O}(n)$ space.

To incorporate equivalence relations into Datalog, we extend the semi-naïve evaluation strategy to support self-computing data structures. To ensure a dense domain, we employ a *densification* mechanism that allows the deployment of an efficient union-find data structure implementation. Lastly, we design the EQREL data structure to support concurrency, thus allowing integration into a parallel Datalog engine.

We have implemented the EQREL data structure in the parallel Datalog compiler SOUFFLÉ. Our data structure compactly stores data, which results in quadratic speed-up and space improvements over explicitly storing equivalence relations. We have evaluated our data structure on several real-world benchmarks that store up to half a billion pairs and demonstrated that it scales five orders of magnitude better than an explicit representation using B-Trees.

The contributions of our paper are summarized as follows:

- extending the parallel semi-naïve evaluation strategy for equivalence relations in a parallelizing Datalog compiler,
- designing a three-layered, self-computing data structure for efficient and parallel handling of equivalence relations, and
- providing experimental evaluation on industrial sourced applications.

The paper is structured as follows. In Section II we provide background and motivate the need for the EQREL data structure. In Section III, we describe the design of EQREL, and its integration into a parallelizing Datalog compiler. In Section IV, we evaluate the performance of our data structure for various use cases, compared to an explicit representation.

II. BACKGROUND AND MOTIVATION

We use the standard terminology for Datalog, taken from [1]. A tuple is of the form $R(c_1, \ldots, c_n)$, where the relation R has arity n, and each c_i is a constant. A Datalog program P is a finite set of logic *rules*, which compute a set of tuples from a set of input tuples. Each rule is a Horn clause of the form: $R_h(\boldsymbol{v}_h)$:- $R_1(\boldsymbol{v}_1), \ldots, R_k(\boldsymbol{v}_k)$ where the :- operator denotes a logical implication, each $R_i(\boldsymbol{v}_i)$ is an *atom*, where R_i is a relation and $\boldsymbol{v}_i$ is a vector of appropriate arity, containing *constants* or *variables*. Each atom can be negated, with restrictions (e.g., stratification). The atom on the left side of the :- operator is the *head* of the rule, and the k atoms on the right side is the body.

Each rule is read right-to-left as a universally quantified implication. Thus, if $R_1(\boldsymbol{v}_1), \ldots, R_k(\boldsymbol{v}_k)$ holds under an evaluation, then $R_h(\boldsymbol{v}_h)$ holds. We also distinguish input and computed relations. Any relation occurring only in the body of rules is part of the Extensional Database (EDB), or input. In contrast, any relation occurring in the head of any rule is part of the Intensional Database (IDB), and is computed from the rules in the Datalog program.

Apart from its use as a database query language, Datalog has been used as a logic specification language for specifying properties of systems that are checked using a Datalog solver. An example Datalog program is given in Figure 1. In this example, the `transaction` relation is the EDB, while the `same_user` relation is the IDB. This program implements a blockchain wallet analysis, which clusters users that appear to be the same. We assume that if two users sign the same transaction, then they both control that same private key and must be the same user. This is expressed in rule 1 (lines 1 and 2), where users u_1 and u_2 sign the same transaction *tx*, and thus are determined to be the same user. Rule 2 (lines 3 and 4) expresses the transitive property, that is that if u_1 and u_2 are the same user, and if u_2 and u_3 are the same user, then u_1 and u_3 should also be the same user. Note that

```
1 same_user(u1, u2) :- transaction(tx, u1),
2                      transaction(tx, u2).
3 same_user(u1, u3) :- same_user(u1, u2),
4                      same_user(u2, u3).
```

Figure 1: Blockchain Wallet Analysis

the blockchain example in Figure 1 demonstrates the use of an equivalence relation. While there are no explicit reflexive and symmetric rules, rule (1) captures the semantics of these properties, as users u_1 and u_2 are unordered and are maybe the same user. Therefore, equivalence relations may show up even when all 3 properties are not explicitly stated.

To evaluate this Datalog program, the de facto approach is known as semi-naïve evaluation [1]. Semi-naïve evaluation is a bottom-up evaluation approach, that starts from the input tuples and iteratively computes new tuples until a fixed point is reached. Semi-naïve evaluation uses auxiliary relations, namely, a *new* and a Δ-version of each recursive relation. Relation *new*^{k} contains the new tuples (including recomputed tuples) computed in iteration k, while relation Δ^k stores all new tuples (excluding recomputed tuples) generated in iteration k. These auxiliary relations avoid the recomputation

of previously computed tuples. For the example in Figure 1, Semi-naïve evaluation transforms the recursive rule in line 2 into two new rules:

$$new^{k+1}_{\texttt{same\_u}}(u_1, u_3) \text{ :- } \Delta\texttt{same\_u}^{k}(u_1, u_2), \texttt{same\_u}^{\texttt{k}}(\texttt{u}_2, \texttt{u}_3)$$
$$new^{k+1}_{\texttt{same\_u}}(u_1, u_3) \text{ :- } \texttt{same\_u}^{\texttt{k}-1}(\texttt{u}_1, \texttt{u}_2), \Delta\texttt{same\_u}^{\texttt{k}}(\texttt{u}_2, \texttt{u}_3)$$

In each rule, k denotes the current iteration of the evaluation. Once the evaluation of these two rules are completed for the current iteration, the contents of the *new* and Δ relations are merged into the main relation:

$$\Delta\texttt{same\_u}^{k+1} := new^{k+1}_{\texttt{same\_u}} \setminus \texttt{same\_u}^{\texttt{k}}$$
$$\texttt{same\_u}^{\texttt{k+1}} := \Delta\texttt{same\_u}^{k+1} \cup \texttt{same\_u}^{\texttt{k}}$$

For the given Datalog program, SOUFFLÉ performs several transformations, producing the OpenMP parallelized C++ code shown in Listing 2, which was simplified for the sake of readability. This C++ code implements a specialized *semi-naïve* bottom-up evaluation of the Datalog program, computing a least fixed point that coincides with the result of the logic specification.

In the first stage, we iterate over the `transaction` relation (lines 5 to 13). For each tuple t_1 in relation `transaction`, we iterate over the subset of `transaction` containing tuples with the same first element as t_1 (line 7). The resulting tuples are inserted into relation `same_user` (line 10).

In the second stage, we evaluate the recursive rule. The semi-naïve evaluation introduces auxiliary relations `delta_same_user` and `new_same_user` for storing the new tuples generated in the previous iteration and current iteration, respectively. We first iterate over `delta_same_user` (line 19), finding tuples $t_1 \equiv (u_1, u_2)$. Then, we iterate over `same_user` (line 20), to find tuple (u_2, u_3) matching t_1. Finally, the tuple $t_2 \equiv (u_1, u_3)$ is inserted into `new_same_user` (line 24).

In the example, we have used C++ STL containers for representing relations. However, Datalog engines use highly customized relational data structures. For example, SOUFFLÉ contains a framework that allows the integration of any set container [18], [17], provided the following operations exist:

- *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.

Since there is no universal best relational data structure for Datalog, the SOUFFLÉ framework offers a *portfolio* of relational data structures that provide applications a choice.

```
1  using Tuple = array<size_t,2>;
2  using Relation = set<Tuple>;
3  Relation evaluate(const Relation
       &transaction){
4   Relation same_user;
5   // same_user(u1,u2) :- transaction(tx,u1),
       transaction(tx,u2).
6   for (const auto &t1: transaction) {
7    for (const auto &t2 : transaction) {
8     if (t2[0] == t1[0]) {
9      Tuple t3({t1[1], (*it)[1]});
10     same_user.insert(t3);
11    }
12   }
13  }
14  // new_same_user(u1, u3) :-
       delta_same_user(u1, u2), same_user(u2,
       u3).
15  Relation delta_same_user = same_user;
16  while(!delta_same_user.empty()){
17   Relation new_same_user;
18   #pragma omp parallel for
19   for (const auto &t1: delta_same_user){
20    for (const auto &t2 : same_user) {
21     if(t2[0] == t1[1]) {
22      Tuple t3({t1[0], (*it1)[1]});
23      if (same_user.find(t3) ==
           same_user.end())
24       new_same_user.insert(t3);
25     }
26    } // end of for same_user
27   } // end of for delta_same_user
28   /* omitted code similar as above for
       new_same_user(u1, u3) :- same_user(u1,
       u2), delta_same_user(u2, u3) */
29   same_user.insert(new_same_user.begin(),
       new_same_user.end());
30   delta_same_user.swap(new_same_user);
31  } // end of while
32  return same_user;
33 }
```

Figure 2: Compiled C++ Code

This paper presents the integration of a specialized data structure designed for efficient computation and storage of equivalence relations. One of the major novelty of this new data structure is that it becomes self-computing; that is, the rules for reflexivity, symmetry, and transitivity is computed by the data structure itself (instead of performing rules in the semi-naïve evaluation to obtain new tuples).

A. Equivalence Relations in Datalog

Equivalence relations are binary relations that are reflexive, symmetric, and transitive. Any elements that are related by virtue of these properties are considered to be within the same equivalence class. We include a Datalog snippet demonstrating a binary relation with equivalence relation semantics in Figure 3. Note that the reflexivity in the

Datalog snippet is only partially specified because of the rule `relation(a,a) :- relation(_,a)` is subsumed by the symmetry rule (2).

```
1 relation(a,a) :- relation(a,_). // (1)
      reflexivity
2 relation(a,b) :- relation(b,a). // (2)
      symmetry
3 relation(a,c) :- relation(a,b), // (3)
      transitivity
4                  relation(b,c).
```

Figure 3: Explicit Equivalence Relations in Datalog

Using this explicit representation of an equivalence relation, the program derives many output tuples as a result of a single input tuple. For example, if the input to the above example was the tuple `relation(1, 2)`, the resulting output would be: `relation(1,1)`, `relation(1,2)`, `relation(2,1)`, `relation(2,2)`. If the EDB also contained `relation(2,3)`, five additional tuples would be part of the final computed knowledge; i.e., `relation(1,3)`, `relation(2,3)`, `relation(3,1)`, `relation(3,2)`, `relation(3,3)`. In this example, only a single equivalence class exists; i.e., $\{1, 2, 3\}$.

The complexity of an equivalence class representation differs from an explicit representation. Let $R \subseteq D \times D$ be an equivalence relation on a domain D. The overhead required to store an equivalence relation explicitly depends on the characteristics of the equivalence classes. The worst-case occurs when R is a single equivalence class, where $|D|^2$ storage overhead is required for an explicit representation, and the best-case is if each element in D is its own equivalence class, where only $|D|$ storage overhead is required. Meanwhile, regardless of the characteristics of the equivalence classes, an implicit equivalence class representation only has $|D|$ elements to deal with.

III. Parallel Equivalence Relation Data Structure

In this section, we present the design of the concurrent self-computing EQREL data structure for semi-naïve evaluation. The data structure has three layers, namely, (1) an equivalence relation layer, (2) a densification layer and (3) a disjoint-set layer. The equivalence relation layer provides an interface for the data structure, imitating an explicit relation representation with operations such as iteration over pairs and insertion; the densification layer compacts the domain of the equivalence relation such that a fast array-style implementation for the disjoint-set layer can be employed; and the disjoint set layer implements a wait-free union-find data structure.

A. Equivalence Relation Layer

The equivalence relation layer provides an abstraction layer, so that semi-naïve evaluation can use the data structure transparently (i.e., as if all equivalent pairs were stored explicitly). All operations performed on this layer interact with the lower layers, where the actual data is managed. The interface of this layer is designed to mimic the functionality of a binary relation stored explicitly, allowing operations such as iteration, set partitioning (for parallelization), and concurrent insertion. The equivalence relation layer includes an evaluation extension such that EQREL can be used for Δ-relations without compromising correctness of Datalog evaluation.

Evaluation Extensions: A major reason precluding the use of equivalence relations in state-of-the-art Datalog engines is that the Δ-relations may become an under-approximation in naïve equivalence relation implementations. Consider the example in Figure 4. The current relation in iteration k contains 3 equivalence classes: $\{a, b, c\}$, $\{f, g\}$, and $\{d, e\}$. The new knowledge generated in iteration $k + 1$ contains 2 equivalence classes: $\{b, f\}$ and $\{g, c\}$. If the standard semantics of semi-naïve evaluation were used (i.e., $\Delta_R^{k+1} := new_R^{k+1} \setminus R^k$), then the Δ-relation would be equal to the new relation. However, in this case the new relation should actually join the equivalence classes $\{a, b, c\}$ and $\{f, g\}$, and so tuples such as (a, f) would be implicitly generated. Such implicit tuples are not captured by the standard semi-naïve evaluation with EQREL, and so we require to *extend* the Δ-relation when it is computed. We denote this extension operator as $\odot$, so that

$$\Delta_{eqrel} R^{k+1} = new_R^{k+1} \odot R^k$$

The extension of the delta equivalence relation is implemented by Algorithm 1. This algorithm takes as input the current relation R^k and the new relation new_R^{k+1}, and computes an extended relation Δ_R^{k+1}. The algorithm iterates over each element e of new_R^{k+1}, and finds the equivalence class in R^k that contains e (denoted as *class*). The *class* is then inserted into Δ_R^{k+1} as an equivalence class, thus capturing the extension semantics, since there are implicit tuples generated from elements of *class*, and e. Algorithm 1 operates in amortized $\mathcal{O}(\alpha(n)n)$ time: each element is visited at most once where it will at most perform a constant number

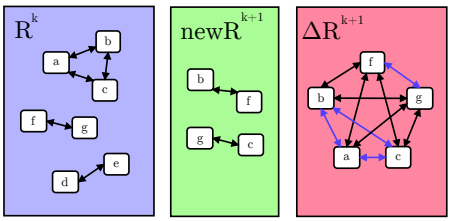

Figure 4: Resulting delta relation after the extension

of `find` or `union` queries to find the representatives of a class or to insert a pair into a relation.

Note, however, that the result of Algorithm 1 is an over-approximation of a Δ-relation in a standard semi-naïve evaluation. Superfluous pairs are marked in blue in Figure 4. This is a side-effect of storing the Δ-relation as an equivalence relation, which implicitly computes equivalence tuples. The over-approximation will not affect the correctness of semi-naïve evaluation: in the worst case, some recomputation of previously computed tuples will be performed during Datalog evaluation.

```
1:  procedure EXTEND(origR, newR)
2:      new_relation ← empty equivalence relation
3:      element_list ← empty set
4:      ▷ Add elements that exist in both sets to our worklist
5:      for element ∈ ELEMENTS(newR) do
6:          if element ∈ ELEMENTS(origR) then
7:              element_list.ADD(element)
8:      ▷ add classes from origR that contain an element
    from element_list
9:      for element ∈ element_list do
10:         class ← equivalence class in origR that contains
    element
11:         for child ∈ ELEMENTS(class) do
12:             new_relation.INSERT(element, child)
13:             ▷ Ensure we don't visit a class twice
14:             if child ∈ element_list then
15:                 element_list.REMOVE(child)
16:     ▷ add all classes within newR
17:     for class ∈ CLASSES(newR) do
18:         rep ← REPRESENTATIVE(class)
19:         for element ∈ class do
20:             new_relation.INSERT(rep, element)
21:     return new_relation
```

Algorithm 1: Return an extended relation

Iterators: The iterators of EQREL are required to simulate an explicitly represented binary relation. However, since the equivalence relation is represented implicitly via a union-find data structure, the construction of iterators for simulating an explicit representation is more involved. For the construction of the iterators, we process each equivalence class separately and produce for each equivalence a list that can produce the pairs for the iterators. These lists are volatile and are implemented using a cache mechanism, i.e., as soon as new pairs are inserted into EQREL, the lists for the iterators are discarded.

The necessity of using a cache mechanism for iterators is because the lower layers of the data structure store the full equivalence relation in a single large list. To iterate over a single equivalence class in this list would require to iterate over every element, checking which equivalence class that element belongs to, exhibiting a $\mathcal{O}(\alpha(n)n^2)$ worst-case complexity. By utilizing a caching mechanism where each equivalence class is stored in a separate list, a single equivalence class can be iterated over using a double nested for-loop. The cache mechanism exhibits $\mathcal{O}(\alpha(n)n)$ worst-case complexity runtime for constructing the iterators, and $\mathcal{O}(d^2)$ worst-case complexity runtime to iterate over an equivalence class, where d is the size of the equivalence class. Another advantage of the cache mechanism is that equivalence classes are stored more compactly in memory (rather than dispersed throughout a large array), thus leading to better cache coherence during iteration.

Internally, the caches for iterators are stored in a cache map, where the keys are the representatives of each equivalence class, and the values are the cache arrays storing that equivalence class. Thus, iterating over a subset of the relation (i.e., all pairs where the first/second element is fixed) is efficient, as finding the correct equivalence class is a lookup of the representative for that element.

To generate the caches, we first create the aforementioned mapping from each disjoint set to its corresponding cache list, using a specialized concurrent B-tree [18]. To fill the cache lists, we iterate through each disjoint set in the underlying union-find data structure. The cache generation algorithm is illustrated in Algorithm 2, showing that it interacts directly with the lower levels of the data structure. As the above algorithm is designed to be distributed across

```
1:  procedure GENERATE_CACHE(rel)
2:      for element ∈ rel.disjoint_set do
3:          drep ← rel.disjoint_set.FIND(element)
4:          sparse_rep ← rel.TOSPARSE(drep)
5:          sparse_element ← rel.TOSPARSE(element)
6:          ▷ Append sparse_element to a list determined by
    the representative of the equivalence class
7:          if sparse_rep ∉ rel.cache then
8:              rel.cache[sparse_rep] ← empty list
9:          rel.cache[sparse_rep].APPEND(sparse_element)
```

Algorithm 2: Generate the equivalence cache

parallel workloads, in the actual implementation we iterate over the elements by assigning portions of the disjoint set across different threads. Thus, it is important that we use a thread-safe list for the caches, which we describe in Section III-D.

Iterator Partitioning.: The EQREL data structure is designed to facilitate effective concurrent usage. To achieve load-balancing and improve cache coherence, the iteration space is partitioned so that each thread can iterate over their own portion of the data structure. For partitioning the data structure, we design a `partition(count)` operation, which generates approximately `count` iterators over the equivalence relation.

For this purpose, we introduce two new iterator creation procedures: `CLOSURE` generates an iterator that covers all pairs represented by an equivalence class, and `ANTERIOR` generates an iterator for an equivalence class with a fixed first element (i.e., iterating over all x for (c, x) where c is fixed). Our heuristic generates these partitions as demonstrated in Algorithm 3. If there are more disjoint sets than the number of partitions, we generate a `CLOSURE` iterator for each equivalence class (lines 4 to 8). Otherwise, we split up large equivalence classes, with one iterator for each element in the class (the element is fixed as the first element using `ANTERIOR`, lines 13 to 16), and create a `CLOSURE` iterator for small equivalence classes (lines 17 to 19).

```
1:  procedure PARTITION(rel, num_iters)
2:      iterators ← empty list
3:      ▷ Special case: supply an iterator per equivalence
    class
4:      if NUMCLASSES(rel) ≥ num_iters then
5:          ▷ Add an iterator that covers the entire class
6:          for class ∈ equivalence classes do
7:              iterators.APPEND(CLOSURE(class))
8:          return iterators
9:      ▷ Approximate pairs per equivalence class
10:     ppc ← SIZE(rel) ÷ num_iters
11:     for class ∈ CLASSES(rel) do
12:         ▷ if this class needs to be split up
13:         if SIZE(class) ≥ ppc then
14:             for element ∈ class do
15:                 ▷ generate iterator covering (element, *)
16:                 iterators.APPEND(ANTERIOR(element))
17:         else
18:             ▷ otherwise cover the entire class
19:             iterators.APPEND(CLOSURE(class))
20:     return iterators
```

Algorithm 3: Partition the equivalence relation to generate a number of iterators, which cover all pairs stored within

B. Densifier

The union-find implementation of the lower layer uses a contiguous array for processing and representing disjoint-sets efficiently. Within this array, the elements are encoded using their array index as an identifier; these identifiers we refer to as *dense* values. As elements within the input domain of the equivalence relation are not necessarily tightly encoded, we require a mapping between these *sparse* values and dense values. In addition to this sparse-to-dense mapping, we require an inverse mapping, for internal operations. We assign these sparse values dense values incrementally, on demand. A sparse value when densified will always resolve to the same dense value; similarly, for the dense-to-sparse mapping. The bijective mapping is implemented by two data structures; the sparse-to-dense mapping is stored within a specialized B-tree [18], whilst the dense-to-sparse mapping is stored using a custom thread-safe random-access list, discussed in Section III-D. Although the task of the densifier is basic, great care must be taken to implement the densifier efficiently.

We have applied slight modifications to a high-performance B-tree implementation [18] such that an atomic counter is incremented on each insert and that value is automatically inserted into newly created element nodes. It is this counter value that produces new dense values as shown in the insert procedure. When a dense value is newly created, the sparse value is inserted into the random-access list at the index of $dense - 1$. Retrieving a sparse value given a dense value is thus trivial.

C. Disjoint-Set Layer

Union-find is an efficient data structure to partition a set of elements D into disjoint sets. Conceptually, these disjoint sets partition the set of elements into equivalence classes. A union-find data structure must support the following operations: `make_set` which creates a new disjoint set with one element, `union` which merges the disjoint sets of two elements, and `find` which returns the representative of the disjoint set containing an element.

Union-find data structure represents a disjoint set as a tree where the root of the tree becomes the representative element of that disjoint set. The tree can be either be represented as a dynamic tree data structure which may result in a slower implementation. Alternatively, the tree can be represented by an array. For example, Anderson's parallel union-find implementation [3] stores the elements in an array whose array elements contain a record that contains two fields: the *parent* index, and a *rank*. The index of an array element represents the element itself, the parent index links the element to its parent in the tree. The rank represents the *quasi-height* of the element.

However, with an array representation, the domain of elements is assumed to be *fixed*, i.e., the size of the array determines all possible elements that can be stored. While evaluating rules, new elements may be generated arbitrarily, and so an expanding domain is required. C++'s `std::vector` may be suitable for this task, however, it is not concurrent, and inefficiently requires copying of elements when the underlying container is filled. We introduce a new high-performance implementation of Anderson's parallel union-find data structure using our custom concurrent expanding list data structure, *PiggyList* described in Section III-D. Note that Datalog relations are growing monotonically [1], and hence no deletion operations are required.

D. PiggyList

Named due to the expanding nature of the data structure, this similar to a simplified version of the Intel Threading

Building Blocks (TBB) `concurrent_vector` [15], instead supporting two modes of operation: appending, and random-access element creation. This list is used in multiple places in the EQREL layered data structure: (1) the list in the equivalence cache (operates in *append* mode), (2) the dense-to-sparse mapping in the densifier layer (operates in *random-access* mode), and (3) the array containing the disjoint-set forest (operates in *append* mode).

In append mode, elements are written to the next available index within a block, and if the available slots are depleted, a new block is created with a size double of the previous. A lookup-table is updated with the location of the new block.

For finding the corresponding *block index*, compiler intrinsic integer logarithms are used that only require slight modification for varying starting block sizes. This data structure is efficient with regards to locks - it is only necessary to lock when new blocks are added. However, it is a very rare event that new blocks are added, i.e., a logarithmic number of times due to the use of double-checked locks.

IV. EXPERIMENTS

In this section, we evaluate the performance of our EQREL data structure and its integration in the SOUFFLÉ Datalog compiler. Our experiments aim to validate the following claims:

- Claim I: The EQREL data structure is more scalable than an explicit representation of equivalence relations.
- Claim II: EQREL performs better than a state-of-the-art B-tree when integrated into a Datalog compiler, for real-world use cases containing equivalence relations.
- Claim III: The EQREL data structure uses less memory than an explicit representation in real-world Datalog benchmarks.

Our EQREL data structure is implemented in C++ and is open-source, available under the UPL license. In a Datalog program, a relation can be tagged as an equivalence relation using the `eqrel` qualifier. For such relations, the explicit equivalence rules (reflexivity, symmetry, and transitivity) are not required, and the synthesizer of SOUFFLÉ will employ the EQREL data structure to self-compute an equivalence relation.

The performance of the EQREL data structure is evaluated through a set of micro-benchmarks as well as real-world Datalog programs in the SOUFFLÉ engine. We compare the performance of the implicit representation of equivalence relations in EQREL to an explicit representation. The explicit representation uses a state-of-the-art B-tree designed for Datalog evaluation [18].

We have run our experiments on an Intel Xeon Gold 6130 CPU with 16 cores (32 threads) at 3.7 GHz, and 192 GB memory. The operating system is Fedora 29, with GCC version 8.3.1 used for compiling SOUFFLÉ synthesized programs.

A. Microbenchmarks

In this section, we evaluate the performance of EQREL on equivalence relations of different characteristics. The main experiment measures the execution time for the two most important operations in SOUFFLÉ: insertion and iteration. We compare the performance of EQREL with a state-of-the-art B-tree [18] which stores an explicit representation of the same data. For these benchmarks, we test four different size characteristics. Assuming we have n total elements, we test:

- n equivalence classes, with each element in its own equivalence class, and therefore the only pairs are those with the same element repeated. This case results in n pairs of elements, and is the best case for the explicit representation as the implicit representation does not gain any meaningful implicit information denoted as *tiny*).
- $\frac{n}{2}$ equivalence classes, with two elements per equivalence class, and therefore each equivalence class contains 8 pairs of elements. This case results in $4n$ pairs of elements. (denoted as *half*)
- $\sqrt{n}$ equivalence classes, each containing $\sqrt{n}$ elements. This results in $n^{\frac{3}{2}}$ pairs of elements (denoted as *sqrt*).
- 1 equivalence class, with all elements being in the same equivalence class. This results in n^2 pairs of elements and is thus the worst case for the explicit representation (denoted as *large*).

To evaluate the performance of the insertion operation, we insert varying numbers of pairs into EQREL and the B-tree. For EQREL, we insert n pairs, with the result implicitly representing all pairs in the equivalence relation. For the B-tree, we must insert each pair explicitly, and thus we expect a significant runtime and memory blow-up for *sqrt* and *large* cases. To evaluate the iteration operation, we start with data structures already containing a set of pairs representing an equivalence relation. Then, we iterate through this full set, measuring the amount of time taken to iterate through varying numbers of pairs.

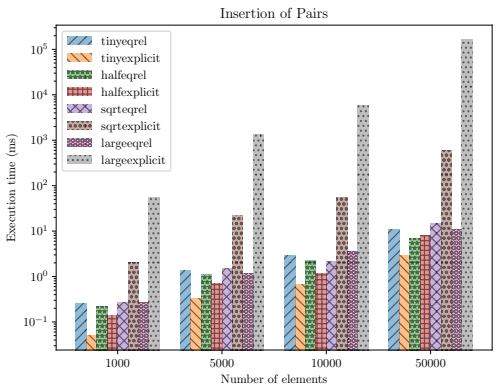

Figure 5: Performance of single-threaded insertion

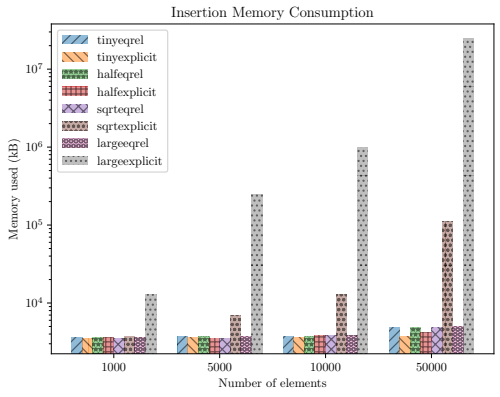

Figure 6: Memory usage of EQREL vs explicit

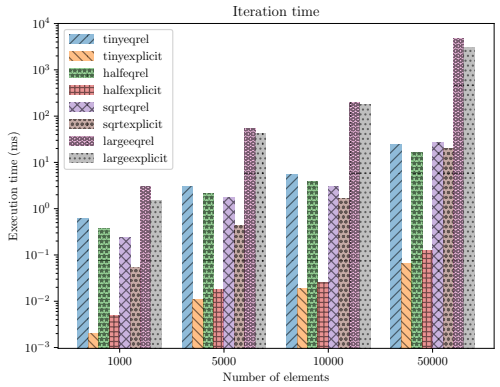

Figure 7: Performance of single-threaded scan over full relation

Figure 5 shows the performance of the insertion operation for EQREL compared to the explicit representation stored in a B-tree. We observe that for the *large* equivalence relation case, EQREL outperforms the explicit representation by multiple orders of magnitude for all input sizes, with the largest input size demonstrating an improvement of up to 4.2 orders of magnitude. This improvement is a result of the large difference between storing n tuples for EQREL, compared to n^2 tuples for the explicit representation. On the other hand, the *tiny* equivalence relations case is the worst case for EQREL, since both EQREL and the explicit representation must store n tuples and no implicit information is gained by using EQREL. However, even in this situation, EQREL exhibits comparable performance to the explicit B-tree, having an overhead of less than $4\times$ for the largest element domain. Moreover, we observe that EQREL performs similarly across all sizes of equivalence relations, as EQREL stores n tuples of data regardless of the sizes of the equivalence classes.

Figure 6 shows the memory usage for EQREL compared to the explicit representation. We observe a similar pattern, with the explicit representation requiring over 3.6 orders of magnitude more memory to store the *large* equivalence relation. We also observe that EQREL uses a constant amount of memory for all cases of equivalence class sizes when the number of total elements is the same. Therefore, we demonstrate that the memory usage of EQREL depends on the number of *elements* in the equivalence relation, rather than the number of *pairs*, as the explicit representation does. Thus, EQREL scales extremely well when the data contains large equivalence classes.

We also repeated the experiment with a larger dataset, to overcome the limitations of measuring memory via resident set size. Through these experiments, we determined the memory requirements to store a set of pairs forming an equivalence relation. The explicit B-tree required up to 9.8 bytes per pair in the *large* case and 12.4 bytes per pair in the *tiny* case. Comparatively, EQREL required 0.000034 bytes per pair in the *large* case, due to the extensive implicit information contained within. However, in the worst case of *tiny* equivalence classes, EQREL requires 34.3 bytes per pair due to the overheads of maintaining separate equivalence classes. In comparison, a direct encoding would require 8 bytes per pair, and thus EQREL significantly outperforms this when implicit information is stored.

Figure 7 shows the performance of the iteration operation, where we iterate over all pairs in an equivalence relation. Note that for iteration, since the result must be equal to the total number of explicit pairs (i.e. up to n^2 tuples), we expect an explicit representation to perform better than EQREL since EQREL is required to reconstruct implicitly stored information. We observe that EQREL performs similarly for *tiny*, *half*, and *sqrt* cases, indicating an overhead for building the caches required for iteration. EQREL performs slightly worse in the *tiny* case compared to *half*, as a result of each equivalence class requiring a separate cache array, and therefore the smaller caches lead to worse cache coherence. However, note that for the *sqrt* and *large* cases, EQREL performs within a $1.6\times$ overhead over the explicit representation, indicating that once the cache building overheads are overcome, iteration is reasonably efficient compared to the B-tree structure.

These microbenchmarks substantiate Claim I, i.e., that the EQREL data structure is more scalable for large equivalence classes than an explicit representation. The runtime speed-up of up to 4.2 orders of magnitude, and memory usage improvement of up to 3.6 orders of magnitude, demonstrate the suitability of EQREL for storing equivalence relations.

B. Industrial Scale Applications

Points-to analysis of the OpenJDK: Points-to analysis is a form of static program analysis which computes an abstract representation of the run-time memory configuration, i.e., set of all possible mappings between variables and objects. These forms of analyses are often costly for real-world programs and require abstractions that balance precision and scalability.

Steensgaard points-to analyses [34] provides an abstracted semantics for the interaction of points-to sets over the duration of the analysis; the points-to sets of variables are equivalence classes and merge when interacting with each other. For example, when a variable `y` is assigned to `x`, the points-to set of `x` and `y` merge. Traditionally, Steensgaard analyses were not amenable to representation in Datalog due to the large number of pairs required to be stored. However, with our EQREL data structure, Steensgaard analyses become tractable, even for large input sizes.

In this experiment, in order for the computation to be tractable for the explicit representation, we operate on a subset of the OpenJDK, namely only generating the points-to set for the `java.lang` libraries using a Steensgaard analysis. We test several versions of the points-to analysis:

1) *explicit* The explicit representation of the Steensgaard field-sensitive analysis in Datalog
2) *eqrel* Using EQREL to implicitly equivalence semantics of Steensgaard
3) *non-symmetric* A stripped explicit representation, removing the symmetry in order to make it more tractable

In Listing 1, the base Datalog program for the *explicit* program is shown. The *eqrel* program is achieved by removing the equivalence relation simulation rules and adding an EQREL annotation, whilst the *non-symmetric* program is achieved by removing just the symmetric rule on line 11.

Listing 1: Steensgaard Datalog Program

```
1  // allocation sites (x = new o())
2  vpt(x,o) :- alloc(x,o).
3  // assignments (x = f)
4  vpt(x,y) :- assign(x,y).
5  // load/store pairs (x.f = y; p = q.f;, q
       and x alias)
6  vpt(y,p) :- store(x,f, y),
7              load(p,q,f),
8              vpt(x,q).
9  // simulate equivalence relation making the
       relation vpt
10 vpt(x,x) :- vpt(x,_).   // (1) reflexive
11 vpt(y,x) :- vpt(x,y).   // (2) symmetric
12 vpt(x,z) :- vpt(x,y),   // (3) transitive
13             vpt(y,z).
14 // output the number of pairs
15 .printsize vpt
```

We set a time-out for the analysis in Figure 8 to be 9 hours. Despite this long time-out, only the 16- and 32-threaded finished with runtimes of over 8 and 6 hours, respectively.

All EQREL programs for each thread take approximately the same time, so that multiple threads provide no benefit to runtime, due to the effect of the multi-threading overhead on a program with a short duration, of around 100ms. The non-symmetric analysis still carries a significant margin of overhead - ranging from around 2 orders of magnitudes slower for one thread, to just under one order of magnitude slower at 32 threads.

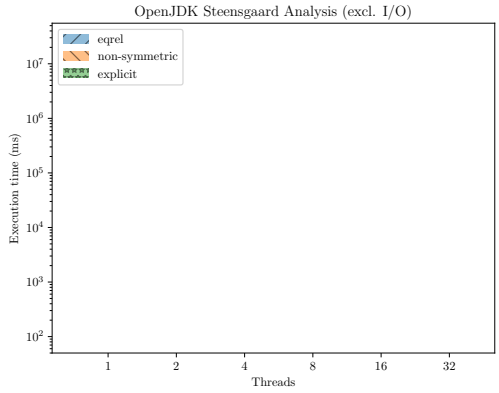

Figure 8: Execution time for the Steensgaard analysis

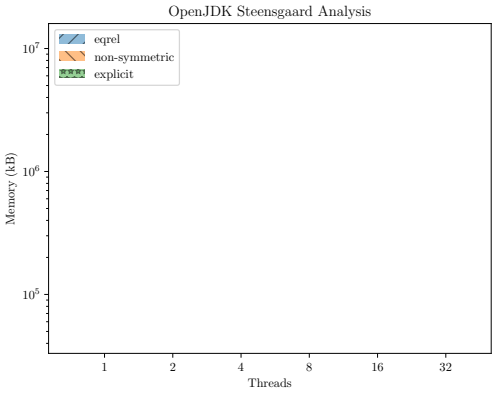

Figure 9: Memory Consumption of the Steensgaard analysis for a variety of threads

Despite the experiment running on only a subset of the OpenJDK, the explicit representation still requires 10GB of memory. As is expected, memory consumption is equivalent across all threads, for each program type. Whilst the non-symmetric program stores fewer pairs than the EQREL program, it carries an overhead of 18% as it must store these pairs explicitly. In addition, we ran this experiment with the full OpenJDK and observed that the EQREL version finished in under 6 seconds on a single thread, whilst the explicit representation timed out after a week.

Bitcoin user identification: The aim of this Datalog program is to take a set of Bitcoin transactions, and partition the associated wallets into disjoint sets based on the user submitting the transaction. A Bitcoin transaction consists of a set of input wallets and a set of output wallets. The user submitting the transaction specifies the amount of Bitcoin each input wallet contributes, and the amounts sent to each output wallet. Each wallet is represented in the transaction by a *public key*. The input wallets are associated with *private keys* held by the user submitting the transaction, which is

important for verifying the authenticity of the transaction. To determine whether two wallets are controlled by the same user, Reid and Harrigan [29] propose the following heuristic. All public keys input to the same transaction are considered to be controlled by a single user, as that user must have control of all the associated private keys. Notably, this heuristic can be represented by an equivalence relation, as it is reflexive (public keys are owned by the same user as their own user), symmetric (likewise), and transitive (inputs across multiple transactions may be shared). We wish to demonstrate the efficiency of the implicit representation over the previous method of explicit representation of equivalence relations. As our dataset, we use a subset of all Bitcoin transactions from 2017 containing over 200 million transaction/input pairs. We are only able to analyze a subset of the transactions due to computational and space requirements. In Table I, the left-hand column denotes how many pairs are loaded in as facts. In order to observe the scaling over both workload, and thread count, we run the EQREL and explicit Datalog programs over three sizes of input pairs, and up to 32 threads.

Whilst the mean size of each disjoint set is less than two (as is the case in the `halfeqrel` and `halfexplicit` benchmark in IV-A), the number of pairs is larger than this would indicate, due to the presence of several large equivalence classes.

In this experiment, we seek to provide a second backing to Claims II and III, in that for another real-world dataset, the EQREL version scales well for large inputs over a number of threads, and outclasses the performance of the equivalent explicit representation.

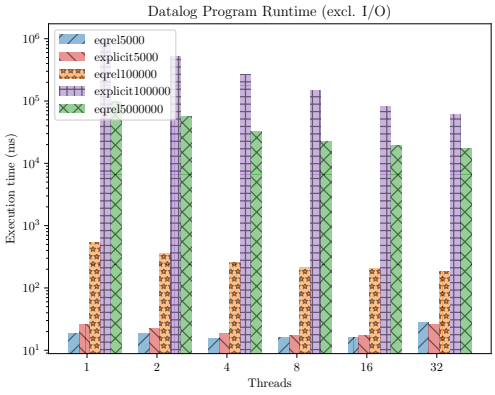

Figure 10: Execution time of a Bitcoin user analysis over varying sized inputs

Not included in the graph is the runtime for the explicit Datalog program for 5 million input transaction pairs, all tests timed out after 10 hours, whilst all EQREL programs finished under 100 seconds for that input size.

From Figure 12, we observe that for EQREL, the scalability across threads improves for inputs larger than 100k, improving 7× from 1 to 32 threads. For the 5000 input run,

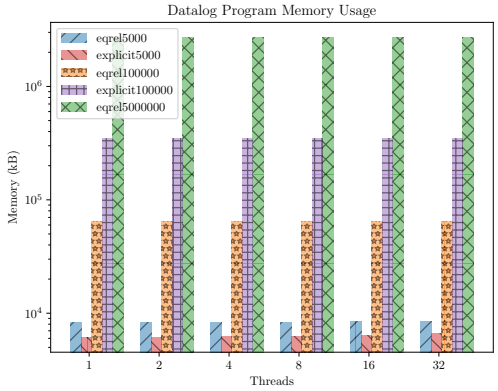

Figure 11: Memory consumption of a Bitcoin user analysis over varying sized inputs

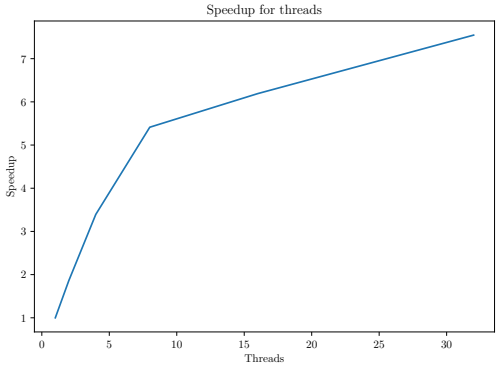

Figure 12: Speedup of the Bitcoin user analysis

there was no benefit running on multiple threads, that again, the overhead from multi-threaded on such a brief program renders threading irrelevant. Moreover, the EQREL version outperforms the explicit version even on the 5000 input run, despite little implicit information is gained. An exception to this is the 32 thread 5000 run, wherein both run times spike due to hyper-threading artefacts.

Comparison to other systems: SOUFFLÉ is an open source, parallelizing compiler, whilst competing Datalog engines are either sequential and/or closed source. For this reason, SOUFFLÉ was the most suitable candidate for implementing the EQREL data structure. However, we would expect similar asymptotic speedups if EQREL techniques were integrated into other bottom-up Datalog engines. Due to the performance disparity on large benchmarks (see experimental evaluation in e.g., [31], [35]), we compare against other engines (BDDBDDB [39] and LogicBlox [5]) using smaller datasets (10k Bitcoin transactions, and a Steensgaard analysis on only the java.lang.String subset), thus establishing SOUFFLÉ as a fair baseline. A comparison is shown in Table II.

As observed in Table II, there are multiple orders of

Size	PubKeys	Transactions	Classes	Singletons	Largest Class	Mean Size	Same User Pairs
5000	4685	4117	3803	3303	11	1.23	7947
100000	76129	54555	43335	38185	2768	1.76	13351193
5000000	3442156	2737910	1776519	1469558	61384	1.94	12096911888

Table I: Statistics of the input data set

Engine	Bitcoin (10k Tx)	Steensgaard (small)
Souffle(eqrel)	32ms	2.6ms
Souffle(explicit)	57ms	910ms
BDDBDDB	2.29s	1.56s
LogicBlox	2.58s	9.17s

Table II: Performance difference between Datalog engines

magnitudes in difference in runtime between performance in SOUFFLÉ EQREL and other engines. When comparing to handwritten code, as far as we are aware of, there is no parallel Steensgard's alias analysis available in a production compiler. We conducted an initial experiment with LLVM/Seahorn [11] that contains a sequential Steensgaard's analysis. We used a benchmark with 62 KLOCs that translated to ≈20K nodes. The analysis took in total 11s where 0.94s was spent on the construction of the equivalence relation. In contrast, our OpenJDK benchmark with ≈47K nodes (see Figure 13) ran in 0.099s - just under ≈10x faster. These real-world Datalog benchmarks substantiate Claims II and III, demonstrating the superior scalability of EQREL compared to an explicit representation. We observe runtime speed-ups of up to 5.4 orders of magnitude, and a memory usage improvement of up to 2.4 orders of magnitude for these real-world use cases. Furthermore, in these Datalog programs, there are no situations where the EQREL is outperformed by an explicit representation.

V. RELATED WORK

Data Structures for Datalog: Previous Datalog implementations have focused on relational data structures including binary decision diagrams [38], Hashsets e.g., [12], [22] and B-trees e.g., [7], [16]. While our experience is that B-trees (as implemented in Logicblox ver. 3 and SOUFFLÉ [16]) have shown to be the most scalable for large ruleset/dataset benchmarks [18], [35], certain use cases may benefit heavily from a more specialized data structure taking into account certain properties of the particular use case. For instance, a Brie data structure introduced in [17] demonstrates significant benefits for highly dense data. For *must-alias* program analysis, [19] introduces both an engine-level and a Datalog-level implementation of a specialized data structure approximating must-alias relations. The EQREL data structure presented in this paper is designed to facilitate efficient storage and processing of equivalence relations.

Special Handling of Equivalences: In the area of semantic web, equivalence relations are also prevalent in certain datasets. For example, with the OWL 2 RL language, the `sameAs` relation is a congruence, which subsumes an equivalence relation. Therefore, specialized handling of `owl:sameAs` has been developed [21], [28], [6], using union-find data structures to store relationships between objects. For Datalog, [13] presents a modular framework for Datalog evaluation, allowing to plug in a specialized equivalence relation evaluation algorithm with a focus on incremental evaluation. However, their approach does not introduce new data structures, and no memory usage improvements are possible.

Parallel Datalog Engines: There has been a multitude of parallelization efforts of Datalog in the past [33], [14], [8], [10], [32], [41], [40] 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. In [42] 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 evaluate Datalog in parallel, 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 [26]. 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 [4], [31] for performance comparisons between engines on large ruleset/dataset benchmarks.

VI. CONCLUSION

We have presented the design, implementation, and evaluation of a novel concurrent equivalence relation data structure for Datalog. We have proposed a three-layered data structure architecture that provides both seamless integration in Datalog engines such as SOUFFLÉ and the performance to scale to industrial sized applications.

REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "Boom analytics: exploring data-centric, declarative programming for the cloud," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 223–236.

[3] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. ACM, 1991, pp. 370–380.

[4] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis, "Porting doop to souffle: A tale of inter-engine portability for datalog-based analyses," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2017. New York, NY, USA: ACM, 2017, pp. 25–30.

[5] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1371–1382.

[6] B. Bishop, A. Kiryakov, Z. Tashev, M. Damova, and K. I. Simov, "Owlim reasoning over factforge." in *ORE*. Citeseer, 2012.

[7] M. Bravenboer and Y. Smaragdakis, "Exception analysis and points-to analysis: better together," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 1–12.

[8] S. Cohen and O. Wolfson, "Why a single parallelization strategy is not enough in knowledge bases," in *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '89. New York, NY, USA: ACM, 1989, pp. 200–216.

[9] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Commun. ACM*, vol. 7, no. 5, pp. 301–303, May 1964. [Online]. Available: http://doi.acm.org/10.1145/364099.364331

[10] S. Ganguly, A. Silberschatz, and S. Tsur, "A framework for the parallel processing of datalog queries," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. New York, NY, USA: ACM, 1990, pp. 143–152.

[11] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 343–361.

[12] K. Hoder, N. Bjørner, and L. de Moura, "μz– an efficient engine for fixed points with constraints," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 457–462.

[13] P. Hu, B. Motik, and I. Horrocks, "Modular materialisation of datalog programs," 2019.

[14] G. Hulin, "Parallel processing of recursive queries in distributed architectures," in *Proceedings of the 15th International Conference on Very Large Data Bases*, ser. VLDB '89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 87–96.

[15] Intel, "Threading building blocks - high performance concurrent data structures," Dec 2017. [Online]. Available: https://www.threadingbuildingblocks.org/

[16] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: on synthesis of program analyzers," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 422–430.

[17] H. Jordan, P. Subotić, D. Zhao, and B. Scholz, "Brie: A specialized trie for concurrent datalog," in *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'19. New York, NY, USA: ACM, 2019, pp. 31–40.

[18] ——, "A specialized b-tree for concurrent datalog evaluation," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 327–339.

[19] G. Kastrinis, G. Balatsouras, K. Ferles, N. Prokopaki-Kostopoulou, and Y. Smaragdakis, "An efficient data structure for must-alias analysis," in *Proceedings of the 27th International Conference on Compiler Construction*. ACM, 2018, pp. 48–58.

[20] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[21] V. Kolovski, Z. Wu, and G. Eadon, "Optimizing enterprise-scale owl 2 rl reasoning in a relational database system," in *International Semantic Web Conference*. Springer, 2010, pp. 436–452.

[22] M. S. Lam, S. Guo, and J. Seo, "Socialite: Datalog extensions for efficient social network analysis," in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ser. ICDE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 278–289.

[23] C. Liu, L. Ren, B. T. Loo, Y. Mao, and P. Basu, "Cologne: A declarative distributed constraint optimization platform," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 752–763, 2012.

[24] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan, "Declarative routing: extensible routing with declarative queries," in *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4. ACM, 2005, pp. 289–300.

[25] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref, "Secureblox: customizable secure distributed data processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 723–734.

[26] C. A. Martınez-Angeles, I. Dutra, V. S. Costa, and J. Buenabad-Chávez, "A datalog engine for gpus," *Declarative Programming and Knowledge Management*, pp. 152–168, 2014.

[27] M.Bravenboer and Y.Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 243–262.

[28] B. Motik, Y. Nenov, R. E. F. Piro, and I. Horrocks, "Handling owl: sameas via rewriting," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[29] F. Reid and M. Harrigan, "An analysis of anonymity in the bitcoin system," in *Security and privacy in social networks*. Springer, 2013, pp. 197–223.

[30] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 196–206.

[31] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 196–206.

[32] J. Seib and G. Lausen, "Parallelizing datalog programs by generalized pivoting," in *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '91. New York, NY, USA: ACM, 1991, pp. 241–251.

[33] M. Shaw, P. Koutris, B. Howe, and D. Suciu, "Optimizing large-scale semi-naïve datalog evaluation in hadoop," in *Proceedings of the Second International Conference on Datalog in Academia and Industry*, ser. Datalog 2.0'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 165–176.

[34] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996, pp. 32–41.

[35] P. Subotic, H. Jordan, L. Chang, A. Fekete, and B. Scholz, "Automatic index selection for large-scale datalog computation," *PVLDB*, vol. 12, no. 2, pp. 141–153, 2018.

[36] D. Suthers, "Ics 311 16: Disjoint sets and union-find," 2015. [Online]. Available: https://www2.hawaii.edu/~janst/311/Notes/Topic-16.html

[37] B. Thau Loo, "Datalog and its application to network routing design," 2010. [Online]. Available: https://www.cis.upenn.edu/~boonloo/research/talks/fmin-loo.pdf

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

[39] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," *SIGPLAN Not.*, vol. 39, no. 6, pp. 131–144, Jun. 2004. [Online]. Available: http://doi.acm.org/10.1145/996893.996859

[40] O. Wolfson and A. Ozeri, "A new paradigm for parallel and distributed rule-processing," *SIGMOD Rec.*, vol. 19, no. 2, pp. 133–142, May 1990.

[41] O. Wolfson and A. Silberschatz, "Distributed processing of logic programs," *SIGMOD Rec.*, vol. 17, no. 3, pp. 329–336, Jun. 1988.

[42] M. Yang, A. Shkapsky, and C. Zaniolo, "Scaling up the performance of more powerful datalog systems on multicore machines," *VLDB J.*, vol. 26, no. 2, pp. 229–248, 2017.

A. Artifact Appendix

A.1 Abstract

This artifact contains the benchmarking suite for the paper *Fast Equivalence Relations in Datalog*. This paper presents a new equivalence relation data structure designed for efficient Datalog evaluation.

The format of these benchmarks is to reproduce our results in the form of raw timing/measurement data, and compiling these into the same graphs as are available within the final paper, for comparison.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Extension of semi-naïve evaluation to accomodate implicit information
- **Data set:** OpenJDK, Bitcoin (both open-source and included)
- **Run-time environment:** Linux
- **Hardware:** 32GB RAM, 32 threads, AMD64
- **Execution:** Full suite takes ≈16 hours
- **Metrics:** Runtime and memory usage
- **Output:** CSV raw data, PDF graphs
- **Experiments:** Run via provided shell scripts, with variations in execution time dependent on CPU/RAM speeds.
- **How much disk space required (approximately):** 2GB
- **How much time is needed to prepare workflow (approximately):** 15 minutes (depending on internet speed)
- **How much time is needed to complete experiments (approximately):** 16 hours
- **Publicly available:** Yes
- **Code licenses (if publicly available):** Universal Permissive License (UPL)
- **Data licenses (if publicly available):** UPL
- **Workflow framework used:** Docker, shell scripts, Python
- **Archived:** DOI: 10.5281/zenodo.3346193

A.3 Description

A.3.1 How delivered

This artifact is available either as a prebuilt Docker image on the Dockerhub repository, or a Dockerless version is available for download from Zenodo (https://doi.org/10.5281/zenodo.3346193).

Refer to section A.4 for instructions on how to install/manage the experiments.

A.3.2 Hardware dependencies

In order to run the experiments as is, 32GB of RAM and 32 CPU threads are required. These are enforced in the several scripts that are provided. Refer to A.7 in how to remove these, and modifying the number of threads.

A.3.3 Software dependencies

Docker, or a Debian-based system if you wish to build without Docker - refer to A.4 for installation instructions.

A.3.4 Data sets

All data sets are included in the Docker image and the Dockerless version.

A.4 Installation

The easiest way to run is with Docker, where we have prebuilt all requirements necessary for Soufflé, and our benchmark suite. Otherwise, we also have a section which describes setting up for a non-Docker environment.

In order to run these experiments, you must run on a machine with at least 32GB of RAM, in addition to at least 32 threads.

These experiments are built using version 1.5.1 of the Souffle Datalog compiler, source code available at https://github.com/souffle-lang/souffle.

A.4.1 Docker installation

Fetch the image from Dockerhub - this is around 1.5 GB in size.

```
docker pull pnappa/pact2019_eqrel
```

Start and enter the container.

```
sudo docker container run -it \
pnappa/pact2019_eqrel /bin/bash
```

You may now start running the experiments, refer to A.5 for instructions. Make sure that after running the experiments, you keep your shell open.

A.4.2 Dockerless

This requires a Debian-based system (tested on Ubuntu LTS 18.04.2).

Download the archive from https://doi.org/10.5281/zenodo.3346193, and extract to find all the relevant files.

Install souffle (version 1.5.1):

```
sudo apt install ./souffle_1.5.1-1_amd64.deb
```

As we render using Latex, you'll need to install the following:

```
sudo apt install texlive-base latexmk dvipng \
texlive-latex-extra time python3-pip
```

Install python3. You'll also need matplotlib to graph.

```
pip3 install matplotlib
```

A.5 Experiment workflow

Each experiment is run by a `runner.sh` script, which handles compilation of C++ code, executing the actual benchmarks, and generating result PDFs.

Note, for sake of runtime, these experiments are only run once, in the paper we repeated the experiments 10 times. If you wish to increase the number of repeats, modify the `repetitions` variable in each of the `runner.sh` scripts for each of the following experiments.

A.5.1 Microbenchmarks

In this benchmark we evaluate the performance of EQREL, comparing the execution time of the two most important operations in

```
#!/bin/bash

# name of the running docker instance
instancename=$(docker ps | grep "pact2019_eqrel" | awk '{print $10}' | head -n 1)

# copy microbenchmark files back
docker cp "$instancename:/artifact/microbenchmarks/pairinsertionsingletime.pdf" .
docker cp "$instancename:/artifact/microbenchmarks/pairinsertionsinglemem.pdf" .
docker cp "$instancename:/artifact/microbenchmarks/pairiterationsingletime.pdf" .

# copy bitcoin files back
docker cp "$instancename:/artifact/bitcoin_same_user/bitcoingraphmem.pdf" .
docker cp "$instancename:/artifact/bitcoin_same_user/bitcoingraph.pdf" .

# copy openjdk analysis files back
docker cp "$instancename:/artifact/openjdk_javalang_steensgaard/jdkgraph.pdf" .
docker cp "$instancename:/artifact/openjdk_javalang_steensgaard/jdkgraphmem.pdf" .
```

Figure 1. Bash script to extract generated PDF graphs from Docker image

SOUFFLÉ- insertion and iteration. We compare the performance with a state-of-the-art B-tree implementation which stores the equivalence relation explicitly.

To run, cd into the `microbenchmarks` directory and run:

`./runner.sh`

This will take approximately 40 minutes to complete.

The resulting graphs will be generating using the python script launched as part of `runner.sh`, and will emit three PDFs:

- `pairinsertionsingletime.pdf` which graphs the runtime (excluding IO) for explicit and eqrel insertions for varying domain types, and input sizes.
- `pairinsertionsinglemem.pdf` graphs the memory usage of the above experiment.
- `pairiterationsingletime.pdf` graphs the iteration time over the domain types versus the input sizes.

A.5.2 Bitcoin Same-User Analysis

In this benchmark, we run a Datalog program that performs a user analysis on a fragment of the Bitcoin blockchain. We compare EQREL with an explicit representation, over varying number of threads, and input sizes (subsets of the blockchain).

To run, cd into the `bitcoin_same_user` directory, and run:

`./runner.sh`

It will take around 45 minutes to complete.

The resulting graphs will be generating using the python script launched as part of `runner.sh`, and will emit two PDFs:

- `bitcoingraph.pdf` which graphs the runtime (excluding IO) for explicit and eqrel representations over varying input sizes of blockchain transactions
- `bitcoingraphmem.pdf` graphs the memory usage of the above experiment.

A.5.3 OpenJDK Steensgaard Points-to Analysis

This runs an experiment over the `java.lang.*` subset of the OpenJDK, whose fact files are generated using a proprietary Oracle Labs tool. We perform several Steensgaard points-to analyses on this language subset; EQREL, an explicit equivalence relation version, and a non-symmetric version.

To run, cd into the `openjdk_lang_steensgaard` directory and run:

`./runner.sh`

It will take around 16 hours to complete.

The resulting graphs that will be emitted as part of `runner.sh` are:

- `jdkgraph.pdf` graphs the solving time (and size calculation) of the analysis vs the number of threads for a variety of Datalog programs.
- `jdkgraphmem.pdf` graphs the memory consumption of the above experiment.

A.6 Evaluation and expected result

The artifact as a whole generates several PDF graphs based on the results from running the experiments. Follow the instructions in A.5 to completion, and keep the Docker container running.

In order to extract the PDF files from the container once the experiments have finished running, you may use the following script. Save the following script described in Figure 1, and run as a superuser (e.g. `sudo bash ./download.sh`):

The resulting graphs should reproduce the relevant results in the paper, demonstrating a quadratic speed up for larger equivalence class datasets (Bitcoin, OpenJDK).

A.7 Experiment customization

One is able to modify the `runner.sh` and `grapher.py` scripts to modify the number of threads and size of the inputs that are consumed.

To modify the number of threads used, the `threads` variable in `runner.sh`, and the `threads` variable in `grapher.py` should be modified.

To modify the size of inputs, the arguments to `run_program` in `runner.sh`, and the `counties` variable in `grapher.py` should be modified.