Building a Join Optimizer for Soufflé

Samuel Arch[1(✉)], Xiaowen Hu[1], David Zhao[1], Pavle Subotić[2], and Bernhard Scholz[1]

[1] The University of Sydney, Sydney, Australia
{sarc9328,xihu5895,dzha3983}@uni.sydney.edu.au, bernhard.scholz@sydney.edu.au

[2] Microsoft, Belgrade, Serbia
pavlesubotic@microsoft.com

Abstract. Datalog has grown in popularity as a domain-specific language (DSL) for real-world applications. Crucial to its resurgence has been the advent of high-performance Datalog compilers, including Soufflé. Yet this high performance is unobtainable for users unless they provide performance hints such as join orders for rules.

In this paper, we develop a join optimizer for Soufflé that automatically computes high-quality join orders using a *feedback-directed optimization* strategy: In a profiling stage, the compiler obtains join size estimates, and in a join ordering stage, an offline join optimizer derives cost-optimal join orders. The performance of the automatically optimized joins is demonstrated using complex real-world applications, including DOOP, DDISASM, and VPC, surpassing the performance of un-tuned join orders by a geometric mean speedup of 12.07×.

Keywords: Datalog · Query optimization · Compilers

1 Introduction

In recent years, Datalog [1,13] has evolved from a recursive database query language to a domain-specific language (DSL) for crafting complex industrial-strength applications, including static program analysis [11,29], network analysis [7], smart contract de-compilation [21], and binary disassembly [18]. Datalog can express complex algorithms with a collection of relations and recursive rules. These applications expressed in Datalog are performance-sensitive for real-world workloads. For example, a static analysis tool such as DOOP [11] may run on software projects with millions of lines of code. Hence, Datalog used as a DSL necessitates compilation techniques for efficient execution [35].

In a bottom-up Datalog system, logic rules can be executed more than once due to recursion and operate over very large data sets. Hence, the runtime of rules is paramount for efficiency. The task of a join optimizer is to find high-quality *join orders*, i.e., the order in which to join the relations in a rule. There are a factorial number of possible left-deep join orders in the number of atoms

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
A. Villanueva (Ed.): LOPSTR 2022, LNCS 13474, pp. 83–102, 2022.
https://doi.org/10.1007/978-3-031-16767-6_5

in the rule, and the performance gap between good and bad join orders can be several orders of magnitude [30].

While automatically finding join orders is well-studied [20,33,36] and known as query optimization in the database community, databases are different to Datalog compilers. For instance, databases normally make no assumptions about future workloads and thus find join orders on the fly at *run-time*, which incurs overheads. Moreover, database workloads rarely contain recursion. Thus, these techniques do not support recursive rules very well. As a result, the database community has little to offer for implementing Datalog compilers whose challenge lies in the repeated execution of *unchanging*, *recursive* rules on different inputs with small variations [44].

For this reason, high-performance Datalog compilers such as Soufflé [26] allow users to optimize join orders using manual annotations at design time. This approach eliminates the overhead of finding join orders at run-time, instead shifting the burden to the developer. However, for non-experts, finding the right join order for large, complex rules can be an insurmountable task. To illustrate the challenge of finding fast join orders, we present a Datalog rule from the DDISASM [18] project, a binary dis-assembler written in Soufflé. DDISASM disassembles stripped binaries into re-assemblable assembly code in a series of analyses. The rule we select is from the Data Access Pattern (DAP) analysis where the atom `data_access_pattern(Address,Size,Multiplier,FromWhere)` represents a data access on an `Address` of size `Size` with multiplier `Multiplier` from address `FromWhere`. The rule derives new data accesses by propagating previous accesses, incrementing the address value by the multiplier to more accurately disassemble the binary.

For this rule there are 24 different join orders amongst the 4 positive atoms. We write the rule in two logically equivalent ways, with the order of the first 2 atoms swapped, illustrating 2 different join orders. In this rule, Soufflé joins the relations in the order in which they are written from left to right, but in general, the join order does not always coincide with the specified order of a rule[1].

```
propagated_data_access(EA+Mult,Mult,EA_ref) :-
    % Propagate previous access by multiplier
    data_byte(EA+Mult,_),
    propagated_data_access(EA,Mult,EA_ref),
    % No collision with next data access pattern
    !possible_data_limit(EA+Mult),
    % No collision with other data access pattern
    last_data_access(EA+Mult,Last),
    Last > EA,
    % No direct collision with data access
    data_access_pattern(Last,Size,Mult,_),
    Size+Last <= EA+Mult.
```

```
propagated_data_access(EA+Mult,Mult,EA_ref) :-
    % Propagate previous access by multiplier
    propagated_data_access(EA,Mult,EA_ref),
    data_byte(EA+Mult,_),
    % No collision with next data access pattern
    !possible_data_limit(EA+Mult),
    % No collision with other data access pattern
    last_data_access(EA+Mult,Last),
    Last > EA,
    % No direct collision with data access
    data_access_pattern(Last,Size,Mult,_),
    Size+Last <= EA+Mult.
```

Fig. 1. A rule from DDISASM written in two ways

When executing the rule in DDISASM, during disassembly of the *gamess* binary from the SPECCPU 2006 suite of binaries [22], Soufflé executes the first

[1] Soufflé relies on a greedy heuristic [15] at compile-time, selecting the next atom to join one at a time with the largest fraction of bounded attributes.

join order in 120.9s and the second join order in 0.02s, opening a performance gap of over 6000×. An expert user may provide join orders manually by using `.plan` statements [16] to hand-tune rule execution. However, finding good join orders is a tedious and time-consuming process, and breaks performance declarativeness. Users are expected to have a deep understanding of the rule execution strategies in Soufflé, if performance is needed.

In this paper, we present an *offline* feedback-directed strategy [40] for join ordering in Datalog compilers. With our new strategy we can derive high-quality join orders automatically. The strategy consists of a profiling and a join ordering stage. The profiling stage produces estimates for the expected number of tuples for each candidate join [36] using a representative input. The estimates are later ingested in the join ordering stage to derive high-quality join orders.

Our approach has the following advantages. First, the rule-set is known ahead of time, so we can perform lightweight *program-specialized* profiling of the Datalog program collecting statistics only for the smallest set of necessary join size computations. Second, the join ordering is performed at compile-time, so that the compiler can search for the cost-optimal join order without incurring any run-time overheads. Third, our approach uses a *recursive rule cost model*, guaranteeing cost-optimal join orders for recursive rules by relying on per-iteration statistics from the instrumented execution. Fourth, our approach is robust; the join orders generated by a representative input tend to generalize over large changes to the input.

We have implemented our new join optimizer in Soufflé [26] as an open-source contribution included in Release 2.3 [17]. We have conducted experiments on industrial-strength applications including DOOP [11], DDISASM [18] and VPC [7] and our join optimizer derives join orders outperforming hand-tuned ones by a geometric mean speedup of 1.09× and un-tuned orders by 12.07×. The contributions of this work are summarized as follows:

1. A novel adaptation of the *feedback-directed optimization strategy* for the join ordering problem.
2. A *program-specialized profiling strategy* for instrumenting Datalog programs to collect accurate join size estimations with automatic index selection.
3. A join optimizer with a *recursive rule cost-model* that finds minimum cost join orders for both recursive and non-recursive rules.

2 Background and Motivating Example

Optimizing joins in compiling Datalog engines is challenging and highly dependent on the rule evaluation strategy. Modern engines, such as Soufflé, frequently use a stratified bottom-up evaluation [1]. For stratified Datalog, an extended dependency graph is built for which the nodes are relations, and edges emanate from the head atoms to the body atoms of rules. The program can be stratified if and only if the dependency graph is free of cycles containing negated atoms. A stratum is a strongly connected component that may contain multiple mutually recursive relations (and their rules). Each stratum is computed by evaluating the

rule-set in a fixpoint loop, terminating when the rules fail to produce new facts and the strata are ordered according to their topological number. Datalog engines adopt semi-naïve evaluation [8] for computing the facts of each stratum. Semi-naïve evaluation has a fixpoint loop for each stratum and finds newly derived facts by considering only the new facts from the previous iteration of the fixpoint loop. As a result, recursive relations in the body of rules can be replaced by *delta relations* where a delta relation only contains the new facts derived from the last iteration. Therefore, a recursive rule $A_{n+1}(X_{n+1})\ :-\ A_1(X_1), ..., A_n(X_n)$ with n mutually recursive relations in the body, results in n *versions* of the recursive rule with semi-naive evaluation as follows:

$$\begin{aligned}
\texttt{new_}A_{n+1}(X_{n+1})\ &:-\ \texttt{delta_}A_1(X_1), A_2(X_2), ..., A_n(X_n).\\
\texttt{new_}A_{n+1}(X_{n+1})\ &:-\ A_1(X_1), \texttt{delta_}A_2(X_2), ..., A_n(X_n).\\
&\dots\\
\texttt{new_}A_{n+1}(X_{n+1})\ &:-\ A_1(X_1), A_2(X_2), ..., \texttt{delta_}A_n(X_n).
\end{aligned}$$

For the motivating example in Fig. 1, Soufflé places each semi-naïve rule-version into the same stratum, evaluating them in a fixpoint loop until no new fact is found. Since this rule has only one mutually recursive relation in the body, it has only one rule-version to compute. Hence the rule will be evaluated as shown in Fig. 2 with the semi-naïve rule-version evaluated inside the fixpoint loop.

```
delta_propagated = propagated
while delta_propagated ≠ ∅ do

    Eval(new_propagated(EA+Mult,Mult,EA_ref) :-
            delta_propagated(EA,Mult,EA_ref),
            data_byte(EA+Mult,_),
            ...)

    delta_propagated = new_propagated
    propagated = propagated ∪ new_propagated
    new_propagated = ∅
```

Fig. 2. Semi-naïve evaluation for the motivating example with `propagated_data_access` abbreviated as `propagated`

To evaluate each semi-naïve rule-version in the fixpoint loop, Soufflé uses indexed nested loop joins due to runtime and memory efficiency [41]. To evaluate each rule-version, the join order determines the loop-nest, placing the first relation in the outer-loop, the next relation in the next loop and so forth, continuing in this order until the entire loop-nest is unrolled. Finally, optimizations are applied to the loop-nest, i.e., rewriting scans over relations with filters into indexed scans. The join order is also known as a left-deep one since the result of each join is pipelined directly as input into the next join operator. For the two

formulations of the Datalog rule, Soufflé uses the order of relations as they are written in the rule resulting in the loop-nests shown in Fig. 3.

Examining the loop-nests in Fig. 3, the first join order iterates the cross-product of `data_byte` and `delta_propagated_data_access` (abbreviated as `delta_propagated`), filtering for tuples where the address accessed on `data_byte` is that of `delta_propagated` with the added multiplier, i.e., `NextEA = EA+Mult`. Hence, the complexity of the two outer loops of the loop-nest is of the order $\mathcal{O}(|\texttt{data_byte}| \times |\texttt{delta_propagated}|)$ excluding logarithmic factors. By comparison, using the second join order, `delta_propagated` is iterated in the outer loop grounding `EA` and `Mult`. Then, an efficient index scan can be performed on `data_byte` to quickly check if there exist any tuples with address `EA+Mult`. Thus, the corresponding complexity of the two outer loops of the loop-nest is $\mathcal{O}(|\texttt{delta_propagated}|)$.

We generated a profile for the *gamess* input from the SPECCPU2006 benchmark suite for DDISASM. From the profile, we observe that this rule produces only 53 tuples. However, despite producing almost no output, the first join order takes 120.9s to complete, iterating for each of the 1.5 million tuples of the `data_byte` relation through (on average) 567 tuples in the `delta_propagated` relation. On the other hand, the second join order only iterates through `delta_propagated` before using an index to check if a matching tuple exists in `data_byte`. As a result, the join order terminates in only 0.02s, a difference of over 6000×. Therefore, the above example demonstrates the utmost importance of good join orders for the performance in Soufflé.

Due to the complex semi-naive rule versions with constantly changing delta relations and the factorial number of possible left-deep join orders, manually finding high-quality join orders is tedious and time-consuming, as shown in the example. As a result, users require a deep understanding of the engine's internal rule execution strategies to find good join orders. Even then, an (expert) user only has a limited time budget for experimentation since there are too many potential join orders to explore. Therefore, users typically move on when they find a sufficiently fast join order and do not fully explore the solution space. Given that the performance gap is so vast between low-quality and high-quality join orders, it is paramount from a usability and a performance viewpoint that Datalog engines such as Soufflé find high-quality join orders automatically without user intervention.

```
for all t1 ∈ data_byte do
  for all t2 ∈ delta_propagated do
    /* NextEA = EA+Mult */
    if t1(1) = t2(1) + t2(2) do
      ...
        insert (...) into new_propagated
```

```
for all t1 ∈ delta_propagated do
  /* EA+Mult */
  if (t1(1) + t1(2), _) ∈ data_byte do
    ...
      insert (...) into new_propagated
```

Fig. 3. Indexed loop-nests for the two formulations of the motivating example

3 A Join Optimizer for Soufflé

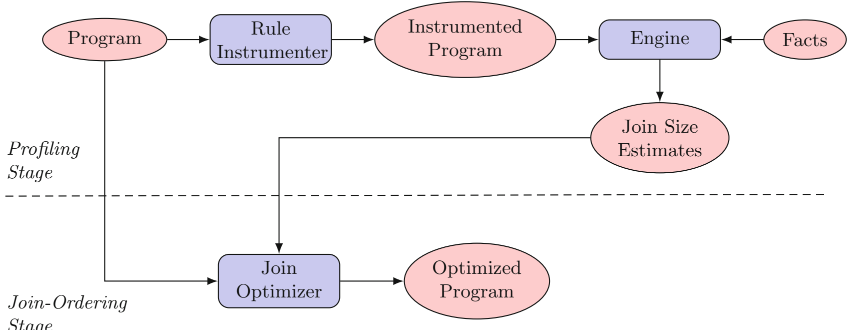


Fig. 4. The FDO Strategy for Join Optimization in Soufflé

To automatically derive join orders for the user, we propose an adapted *Feedback-Directed Optimization* (FDO) strategy [40] for join ordering. For our join optimizer, the FDO strategy has two stages (shown in Fig. 4), a *profiling stage* and a *join-ordering stage*. The profiling stage compiles and runs the Datalog program, using a representative input. As a side-effect of the run, statistics about the execution are produced. The join-ordering stage uses these statistics (the feedback) as input for its compilation. The statistics provide the join optimizer with estimates of the size of each candidate join and the join optimizer then uses these join size estimates and its cost model to derive *cost-optimal* left-deep join orders for all rules in the program using Selinger's algorithm [36]. The key challenges that we address with our join optimizer are:

(1) *Efficiently collecting a potentially exponential number of join size estimates.*
(2) *Developing a cost-model for recursive (and non-recursive) rules.*

We address Challenge (1) with a *program-specialized profiling strategy* that efficiently collects only the necessary statistics. We address Challenge (2) by introducing a *recursive rule cost model* for selecting join orders that computes join size estimates on an iteration-by-iteration basis. Note that for our approach, a representative input (i.e., a training data set) must be chosen in the profiling stage so that the produced join orders do not degrade the application's performance with normally occurring inputs. We demonstrate in Sect. 4 that our approach can generalize well (i.e. a representative input performs within 10% of the optimal for DOOP and DDISASM), and most inputs are representative.

3.1 The Profiling Stage

The profiling stage's task is to efficiently collect join size estimates for the join ordering stage. The join size estimate of an atom depends on its position in a

join order. For example, recall the loop-nest of Fig. 3 for the motivating example. The join size estimates for atoms `data_byte` and `delta_propagated` can change depending on whether they are placed in the first or second loop in the join order. The cost differences stem from the join attributes of the atoms, which are bound by values from outer loops and/or constants. In the example, the join uses the attributes `EA` and `Mult` bound by `delta_propagated` in the outer loop to join `data_byte` using the value `EA+Mult`. The first column of relation `data_byte` becomes a join attribute with value `EA+Mult`, and we want to estimate the number of tuples in `data_byte` that have this value.

The database literature [12,19] introduced the concept of selectivity, which measures the degree to which a predicate filters tuples. An accurate measure for selectivity can be found by projecting the set of tuples on the join attributes and counting the number of projected tuples. Then, the join size can be estimated by dividing the relation size by the number of projected tuples. More generally, the join size estimate can be expressed for an atom as,

$$f_{a_1,\ldots,a_k}(R) = \frac{|R|}{|\pi_{a_1,\ldots,a_k}(R)|} \tag{1}$$

using relational algebra [14] notation where $a_1, \ldots, a_k$ are join attributes[2] of the atom with relation R, where $\pi_{a_1,\ldots,a_k}(R)$ is the set of tuples in R projected on the join attributes and where $|R|$ is the cardinality of relation R. Note that the formula can be refined for constant attributes filtering out tuples that whose constants do not match.

According to Eq. 1, the join size estimate depends on which of the relation's attributes become join attributes, i.e., $a_1, \ldots, a_k$. When relations appear later in the join order, more attributes are bound; hence, the join size estimate becomes smaller. Note that there could be 2^m different join size estimates where m is the number of attributes of relation R (and even more considering constant attributes as well). However, only a small number of join size estimates are necessary to cover all possible join orders of a rule that may occur for a concrete rule. Since the rule set is given in a Datalog compiler, our program-specialized profiling strategy can compute the collection of potential join attributes ahead of time.

The algorithm for determining the collection of join size computations employs a variation of the *sideways information passing (SIP) graph* [1,9]. The graph G_ρ is constructed for a rule ρ of the shape $A_0(X_0) \coloneq A_1(X_1), \ldots, A_n(X_n)$. The vertices of the graph are the body atoms $A_i(X_i)$ for all i, $1 \le i \le n$. By abuse of notation, we assume that the arguments X_i are sets of variables (ignoring constants) that occur as arguments. There is a directed edge from one atom A_i to another A_j if its arguments X_i bind at least one argument in X_j. The set of incoming edges is denoted by function *in*, i.e., $A_j \in in(A_i)$. We denote the bindings variables themselves by $bvars(A_j, A_i)$ between two atoms A_i and A_j. We depict the SIP graph for the motivating example in Fig. 5.

[2] Not necessarily all attributes are join attributes.

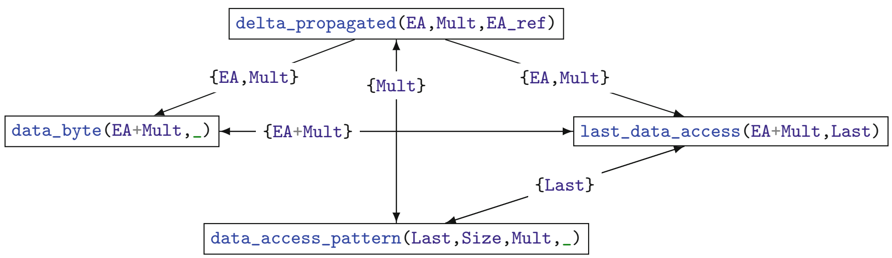


Fig. 5. The SIP graph for the motivating example

Only an atom's neighbours with incoming edges in the sideways information passing graph control the join attributes. From this observation, we can construct a simple algorithm that determines the collection of join attributes. Algorithm 1 computes for each relation the possible join attributes that can be bound in the SIP graph for a rule, i.e., the candidate joins on that relation. The algorithm considers for each rule ρ in the program P, its SIP graph G_ρ. Each atom A_i in the SIP graph finds its corresponding relation. Set B represents a collection of sets of variable bindings for all potential atoms A_j that could be placed before A_i. We perform a power-set construction for all possible subsets of set B. We make the union U of all p_i and compute J the join attributes in A_i bound by U. Finally, we add the join attribute set to the result set S_R.

Algorithm 1 ComputeUniqueJoins(P)

1: Let S be empty for all relations R in P, i.e., $S_R = \emptyset$
2: **for all** $\rho \in P$ **do**
3:   **for all** $A_i \in G_\rho$ **do**
4:     Let R be the relation for A_i
5:     Let $B = \{bvars(A_j, A_i) \mid A_j \in in(A_i)\}$ be the possible bindings passed to A_i
6:     **for all** $p_i \in \mathcal{P}(B)$ **do**
7:       Let $U = \bigcup p_i$ be all bindings from a subset of incoming edges
8:       Let J be the attributes in A_i bound by U
9:       $S_R = S_R \cup J$
10: **return** S

We illustrate the execution of Algorithm 1 on the motivating example when it encounters the atom `data_access_pattern(Last,Size,Mult,_)`. For this atom, there are two incoming edges, from `delta_propagated(EA,Mult,EA_ref)` and `last_data_access(EA+Mult,Last)`. Therefore, the set of all possible bindings B in the algorithm will hold $\{\{Mult\}, \{Last\}\}$. Any combination of these bindings can be possible (considering whether these atoms appear before or after `data_access_pattern` in the join order). Therefore, the power-set of B is enumerated $\{\{\{Mult\}\}, \{\{Last\}\}, \{\{Mult\}, \{Last\}\}\}$. Then for each subset, the bindings are collected with a set union operation to produce the set U. Set U

will take on the values $\{Mult\}, \{Last\}, \{Mult, Last\}$ for each subset. Since $Last$ corresponds to the first attribute, and $Mult$ corresponds to the third attribute, each binding will correspond to the join attributes $\{3\}, \{1\}, \{1, 3\}$. The set S_R then contains for the relation `last_data_access`, these sets of join attributes. The process continues for the remaining atoms, producing sets of join attributes (and hence candidate joins) for each relation.

We use the solution set S of Algorithm 1 to instrument the Datalog program at compile time. The instrumentation injects join size computations into the execution that will be evaluated during run-time to produce each join size estimate. The instrumentation of the program differentiates between recursive and non-recursive relations. For recursive relations, the join size computations are placed inside the fixpoint loop of the stratum (as determined by the semi-naive evaluation algorithm [8] explained in Sect. 2). Figure 6 shows the instrumented semi-naïve evaluation of the Datalog compiler. The join size computations on `delta_propagated` are placed in the fixpoint loop and the join size computations for other non-recursive relations, e.g., `data_byte` are placed in earlier strata as soon as they are fully computed.

```
delta_propagated = propagated
while delta_propagated ≠ ∅ do
    EstimateJoinSize(delta_propagated, { })
    EstimateJoinSize(delta_propagated, {2})

    Eval(new_propagated(EA+Mult,Mult,EA_ref) :-
             delta_propagated(EA,Mult,EA_ref),
             data_byte(EA+Mult,_),
             ...)

    delta_propagated = new_propagated
    propagated = propagated ∪ new_propagated
    new_propagated = ∅
```

Fig. 6. The instrumented semi-naïve evaluation for the motivating example

Computing Join Size Estimates. The join size computations are evaluated at run-time in the profiling stage to derive the join size estimates. Each join size estimate is found by counting the number of tuples in R and counting the unique tuples after the projection of R onto the join attributes (cf. Eq. 1). A naive way to compute the number of unique keys comprises the following steps: (1) projecting every tuple in the relation onto the join attributes, (2) sorting the projected tuples, and (3) iterating and counting the number of duplicates (and hence unique tuples). Note that the sorting step can be avoided by assuming that the tuples are already sorted on the join attributes ahead of time.

However, Soufflé's machinery facilitates a more efficient approach. Soufflé represents each relation R as a collection of multiple in-memory B-Tree indices [27]. Each index for a relation totally orders the tuples in R using a lexicographical ordering ℓ, i.e., $\ell = 1 \prec 2$ would order tuples by attribute 1 then break ties using attribute 2. For each join size estimate, we would like to find an index for R where its lex-order ℓ has the set of join attributes as a prefix, ensuring that the tuples projected on the join attributes will be traversed in sorted order. When the attributes for a join size form a prefix in the lex-order, we say that the join size estimate is covered by the index. To ensure every join size estimate is covered by an index, we can rely on Soufflé's automatic index selection algorithm [41]. The algorithm inspects the joins on each relation (called *primitive searches*) and ensures that each can be covered by an index, using the minimum total number of indexes for each relation. To achieve the same result with join size estimates, we represent each estimate as a primitive search with the same set of attributes, which guarantees that an index will cover it, eliminating the sorting step entirely.

Algorithm 2 EstimateJoinSize(R, J)

1: Let J be the set of join attributes.
2: Let $Dup = 0$ be the number of duplicates.
3: Let $R_\ell = LookupIndex(R, J)$
4:
5: **for all** $Curr \in R_\ell$ **do**
6:     **if** $\pi_{a_i \in J}(Prev) = \pi_{a_i \in J}(Curr)$ **then**
7:         $++Dup$
8:     $Prev = Curr$
9: **return** $\frac{|R_\ell|}{|R_\ell| - Dup}$.

An outline of the join size computation is shown in Algorithm 2. To extend the algorithm for constant join attributes, R_ℓ is first filtered for the tuples satisfying the constant attributes. Considering EstimateJoinSize(`delta_propagated`, $\{2\}$), Algorithm 2 will execute as follows. Firstly, an index R_ℓ will be found with the set of join attributes $\{2\}$ as a prefix. Since the relation has arity 3, the possible index orders are: $\ell = 2 \prec 1 \prec 3$ or $\ell = 2 \prec 3 \prec 1$ over the attributes of the relation. Next, the index is traversed in order, comparing the attribute in position 2 with the previous tuple. Since the index is sorted on attribute 2 first, duplicate values will appear in sequence. Finally, the relation size divided by the number of unique values on attribute 2 is retrieved as the estimated join size. For the *gamess* fact-set, on the first iteration, $|R_\ell| = 31,615$ tuples and $|R_\ell| - Dup = 23$ unique keys. Hence the expected size of the join for the first iteration on attribute 2 is $\frac{31,615}{23} = 1375$ tuples (rounding up). These join size estimates for each candidate join order can then be used to guide the cost model in the join ordering stage to find high-performance join orders.

Note that multiple join size computations can be covered by the same index. For instance, `data_access_pattern(Last,Size,Mult,_)` has the join size computations $\emptyset, \{3\}, \{1\}, \{1,3\}$. Instead of sorting 4 times (creating 4 distinct indexes), the 2 indexes $\ell = 3 \prec ...$ and $\ell = 1 \prec 3 \prec ..$ can cover them. Hence, our indexing technique for join size estimates creates little overhead for the profiling stage, producing the necessary statistics for the join ordering stage.

3.2 The Join Ordering Stage

The problem of finding join orders is well-known in the database literature [24, 25,30,33,37] as part of query optimization. A query optimizer's task is finding the fastest query plan (i.e. the fastest way of executing a query) efficiently using a cost-model [36]. The challenge is that each query has $n!$ possible left deep joins when there are n relations to join. In addition, for each possible join order, each relation can be accessed using different methods, i.e., a hash-join, sort-merge join, nested-loop join or a scan of the entire relation. In the context of Soufflé, every relation is accessed using indexed nested loop joins, and, therefore, each join order corresponds to exactly one query plan. However, there are still $n!$ possible left-deep join orders to consider.

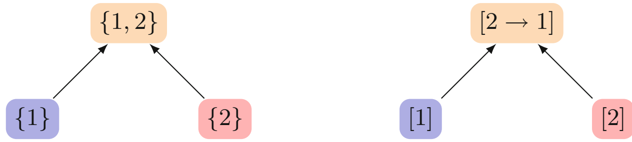


Fig. 7. An illustration of Selinger's algorithm which maps sets of atoms to minimum cost join orders

Selinger [36] observed that finding a minimum cost left-deep join order can be achieved without explicitly considering all $n!$ possible candidates. The insight is that if a join order for a subset of relations is sub-optimal, it can never appear in the optimal join order. Hence, the optimal join order can be found inductively through dynamic programming by considering the optimal join order for every subset of relations. The process is shown in Fig. 7 to join 2 relations. First, each subset of size 1 corresponds to a single minimum cost join order. Next, for the subset $\{1,2\}$, the algorithm considers removal of one relation i.e. the sets $\{1\}$ and $\{2\}$. For each of these subsets, it considers the minimum cost join order found for them, i.e. [1] and [2], and adds this to the cost of extending the join order to include the removed relation, i.e. $[1 \rightarrow 2]$ or $[2 \rightarrow 1]$. In this example, the cost of $[2 \rightarrow 1]$ is cheaper and is saved, with $[1 \rightarrow 2]$ never being considered further. This process continues up the lattice until the minimum cost join order for all relations is found, and the algorithm terminates. The algorithm has a complexity of $\mathcal{O}(n \times 2^n)$ since there are 2^n subsets to consider, and each subset must consider $\mathcal{O}(n)$ join orders from one level below. Hence, Selinger's algorithm

substantially improves upon the brute-force approach of considering all $n!$ left-deep join orders.

Selinger's algorithm (and other query optimization strategies) are designed for queries without recursion. Hence the literature offers little guidance for finding join orders for recursive rules. One approach would be to treat each recursive rule as if it were non-recursive, i.e. by taking the average size of a delta relation across all of the iterations of a given rule and using this average to drive the cost model. The disadvantage of this approach is that Selinger's algorithm would no longer guarantee high-quality join orders. For instance, a delta relation may be very large for the first iteration of the rule and then very small for subsequent iterations (as less new knowledge is derived). If the rule is executed for many iterations, placing the delta relation at the beginning of the join order is usually the most efficient. However, since the initial size of the delta relation is very high, its average might still be very high, and the join order chosen won't place the delta relation first, leading to worse performance. Although more sophisticated aggregate statistics could be used (for example, taking the geometric mean), any aggregate would fail to capture the costs for rules with multiple recursive relations that grow and shrink throughout different iterations.

To address this challenge, we develop a *recursive rule cost model* for Selinger's algorithm, which maintains accurate statistics for each candidate join on an *iteration-by-iteration* basis. Our approach relies on the fact that join size estimates are computed per iteration during the profiling stage as join size computations are placed inside the fix-point loop for each stratum. Under our cost model, the tuples for a particular candidate join are represented by a *vector* of length I where I is the number of iterations of the rule. Under our model, the cost of executing any particular join order is calculated as follows:

$$\begin{aligned}\text{CostToJoin}(\{R_1,\ldots,R_k\},S) = {} & \text{CostToJoin}(\{R_1,\ldots,R_k\}) \\ & + \text{Arity}(S) \\ & \times \sum_{i=1}^{I} \text{TuplesFromJoin}_i(\{R_1,\ldots,R_k\}) \\ & \times \text{ExpectedJoinSize}_i(S,J)\end{aligned}$$

The cost to join relations $\{R_1,\ldots,R_k\}$ with a new relation S is the cost to compute the previous join added to the new join cost. The new join cost is then calculated as the arity of the new relation S multiplied by the sum of tuple accesses across all i iterations from 1 to I by the join. For each iteration i, the number of tuples is calculated as the number of tuples from the previous join (the number of tuples from the outer loop) multiplied by the expected join size of S for this iteration, using join attributes J grounded by the previously joined relations.

To illustrate our approach, consider joining the first two atoms in the motivating example in Fig. 1. The algorithm estimates the cost of each choice between candidate join orders, as shown in Fig. 8. The cost of $(1, 2)$ (the left path shown

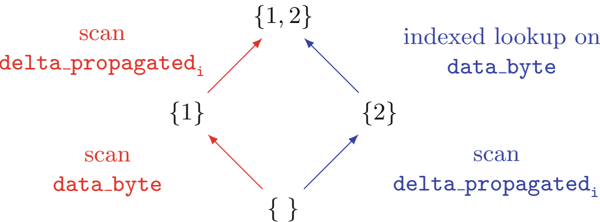


Fig. 8. The possible paths (and hence join orders) for the motivating example (Color figure online)

in red) with the cost model described previously is:

$$\text{Arity}(\texttt{data_byte}) \times \sum_{\texttt{i}=1}^{I} |\texttt{data_byte}|$$
$$+ \text{Arity}(\texttt{delta_propagated}) \times \sum_{\texttt{i}=1}^{I} |\texttt{data_byte}| \times |\texttt{delta_propagated}_{\texttt{i}}|$$

For the right path up the lattice (shown in blue) for order $(2, 1)$, the cost is:

$$\text{Arity}(\texttt{delta_propagated}) \times \sum_{\texttt{i}=1}^{I} |\texttt{delta_propagated}_{\texttt{i}}|$$
$$+ \text{Arity}(\texttt{data_byte}) \times \sum_{\texttt{i}=1}^{I} |\texttt{delta_propagated}_{\texttt{i}}| \times 1$$

Since the indexed lookup using the second join order only accesses a single tuple, the join order $[2 \rightarrow 1]$ is cheaper than $[1 \rightarrow 2]$ and saved. For the remaining atoms in the rule, the process continues up the lattice using dynamic programming, finding join orders for larger subsets of atoms by using the minimum cost join orders previously computed for each smaller subset. Eventually, the cheapest path up the lattice is found, corresponding to the minimum cost join order.

Overall, our adapted usage of Selinger's algorithm increases its time complexity from $\mathcal{O}(n \times 2^n)$ where n is the number of positive atoms to $\mathcal{O}(n \times 2^n \times I)$ where I is the number of iterations. However, the advantage of this approach is that the join sizes (and hence costs) for each iteration are considered separately, and the total cost is minimized, allowing the optimizer to select the cost-optimal join order for recursive rules.

Note that our join optimizer still finds high-quality join orders for rules that contain negated atoms, even though they are not considered explicitly in the algorithm. First, the join optimizer finds the join order considering only the positive atoms and unrolls the rule into a loop-nest. Then, the negated atoms,

which are evaluated as existence checks (see Sect. 2), are hoisted as high as possible in the loop-nest so that they can be evaluated as eagerly as possible. Since negated atoms act as sinks, they don't produce tuples in the loop-nest and hence don't affect the cost of a candidate join (which is determined by the number of tuples it generates). Therefore, our join optimizer performs well for rules with negated atoms unless the rule can terminate early by better placement of a negated atom, i.e. no tuples satisfy the negated atom.

4 Experimental Evaluation

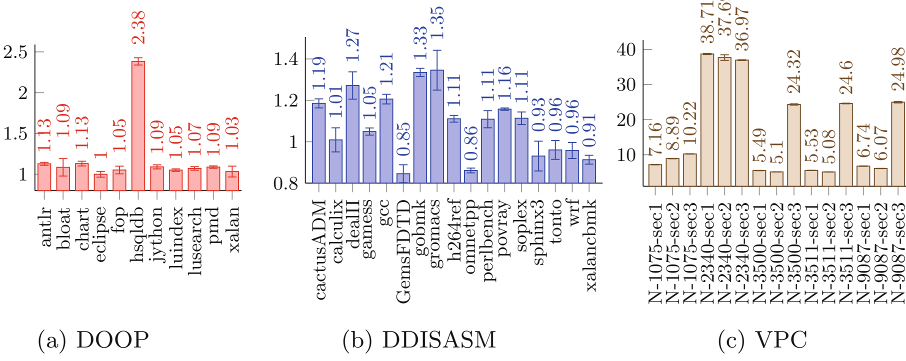
We have conducted several experiments to understand the quality of the proposed join optimizer for Soufflé. Specifically, we aim to answer the following questions:

1. What is the performance of the join optimizer in comparison with current join ordering heuristics in Soufflé?
2. Are the orders that the join optimizer produces on given training data sets robust for different inputs?
3. What is the overhead of the profiling and join ordering stages?

Experiments are run without virtualization using an AMD Ryzen Threadripper 2990WX (3GHz 32-Core Processor). The operating system is Ubuntu 20.10, with programs compiled using GCC 10.30. All performance-related benchmarks are run in single-threaded mode and executed three times for reliability.

We evaluate the join optimizer on the following industrial-strength Datalog applications consisting of hundreds of rules/relations. First, **DOOP** [11] is a framework supporting various types of static analysis of Java programs. We run the *context-insensitive* analysis on the Java programs present in the *DaCapo* [10] benchmark. Second, **DDISASM** [18] is a tool that transforms stripped binaries into re-assemblable assembly code through a series of analyses written in Datalog. We run DDISASM on the SPECCPU 2006 [22] suite of binaries. Third, **Virtual Private Cloud (VPC)** [7] is a benchmark taken from a real-world network security analysis framework deployed in Amazon Web Services.

Join Optimizer Performance. We compare the performance of Soufflé's heuristic join orders with the performance of the new join optimizer. Note that **DOOP** and **DDISASM** are hand-tuned Datalog implementations, whereas **VPC** is automatically generated Datalog code from a network model and is not hand-tuned. Figure 9 illustrates the speed-up achieved by the join optimizer for various input data sets of DOOP, DDISASM and VPC. The join optimizer delivers a performance speedup of up to 1.13× for DOOP. For DDISASM, we observe improved performance with the join optimizer on 11 out of 17 fact sets of up to 1.35× and a geometric mean speedup of 1.069× overall. For 6 out of 17 data sets, performance degrades (in the worst case on *GemsFDTD* by 0.85×). The slowdown is due to the join optimizer selecting new join orders that result in

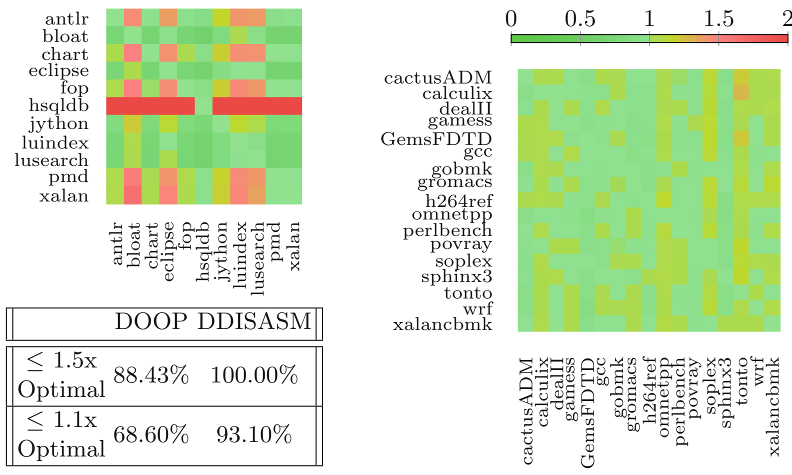


(a) DOOP (b) DDISASM (c) VPC

Fig. 9. Runtime Speed-up of the Join Orders Produced by the Join Optimizer.

extra indexes on relations, slowing down fact-sets that are write-heavy for certain relations.

For VPC, the join optimizer provides better speedups because the join orders for rules are not manually tuned. The speedup ranges from 5.08× up to 38.71× with a geometric mean of 12.07×. These experiments show that the join optimizer finds orders that, on average, perform well for DOOP and DDISASM even in comparison with hand-tuned join orders and delivers outstanding performance in the absence of hand-tuned joins such as VPC.



	DOOP	DDISASM
≤ 1.5x Optimal	88.43%	100.00%
≤ 1.1x Optimal	68.60%	93.10%

Fig. 10. A heat map representing the relative slowdown when optimizing on input A and evaluating on input B compared to optimizing on B and evaluating on B.

Training Data Robustness. For this experiment, we evaluate the robustness of the join optimizer using different data sets as training sets. For this experiment, we run DOOP with context-insensitive points-to analysis using the DaCapo benchmark suite and DDISASM using SPECCPU 2006. The results are shown in Fig. 10 as heat maps. A point in the heat map is a combination of a *training* input (on the x-axis) and a *test* input (on the y-axis). We generate the join orders for the *training* input using our join optimizer and use the same join orders for the *test* input. To evaluate the robustness of the training, we compute the slowdown for each combination. A ratio exceeding 1 is a performance slowdown indicating that the *training* input fails to generalize join orders for the *test* input. For DOOP, only 68.6% of the benchmarks have a slowdown ratio of 1.1× or lower, with 14 benchmarks having a slowdown ratio of at least 1.5×. The test input *hsqldb* is an outlier that generalizes poorly with a slowdown ratio of 3.68× to 6.10×. However, the join optimizer is quite robust for DDISASM's training data, with 93.1% of the runs having a slowdown ratio of less than 1.1×.

For both benchmarks, when a representative training input is selected (e.g., *pmd* for DOOP and *GemsFDTD* for DDISASM), the performance of the chosen join orders can generalize across the benchmark, staying within 10% of the optimal case. This shows that the join optimizer generalizes effectively when trained on a representative input. In any case, our experiments show that new join orders do not need to be re-derived for every new input, given the inputs have a similar structure. We conjecture that in most domains, the structure e.g., relative join sizes for different candidate join orders, would remain largely invariant.

Table 1. The overhead of the join optimizer at each stage

	DOOP			DDISASM			VPC		
	min	avg	max	min	avg	max	min	avg	max
Profiling Stage Slowdown	1.62×	2.61×	4.66×	1.05×	1.18×	1.20×	1.00×	1.18×	1.60×
Join Ordering Stage Time	1m28s	1m33s	1m37s	51s	1m	1m57s	< 1s		

Profiling and Join Ordering Overheads. For our next experiment, we are interested in determining the overhead of our program-specialized profiling and the overhead of running the join optimizer in the join ordering stage. From Table 1, the maximum overall slowdown is 4.66× for DOOP, the largest since there is complex mutual recursion, executing for many iterations. By comparison, the slowdown for DDISASM and VPC is 1.18× on average since they are mostly non-recursive. Overall, the slowdowns for the profiling stage are acceptable, ranging on average from 1.18× to 2.61×.

As shown from Table 1, the maximum time to run the join ordering stage is 1m37s for DOOP, 1m57s for DDISASM and less than 1 second for VPC. DOOP has a large overhead due to rules with larger bodies and rule iterations. DDISASM takes the longest due to some rules with much larger rule bodies than

in DOOP. Finally, since VPC has few iterations due to less mutual recursion, the join ordering stage finishes in less than 1 second. Overall, the join ordering stage of the join optimizer only takes a few minutes for complex industrial strength applications and occurs at compile-time, not run-time, which is acceptable.

5 Related Work

Evaluating Datalog as a DSL. LogicBlox [5] employs worst-case optimal joins, requiring users to manually rewrite and duplicate relations with different attribute orders to achieve satisfactory performance. BDDBDDB [42] uses binary decision diagrams to store relations, but high-performance relies on finding variable orderings, which is an NP-Hard problem [32]. Socialite [38] requires users to provide execution plans that enforce a join order. Other engines such as μZ [23], Flix [31], and PADatalog [4] also require the user to manual tune and provide performance hints that are no longer necessary when using Soufflé with the join optimizer. The technique in [39] provides cost estimates statically before execution.
Database Systems. Relational database management systems rely on cost-based optimization with both bottom-up [6,36] or top-down [20] approaches searching for a low-cost join order by estimating the actual execution cost of each candidate. Since the latency for each request is the sum of the optimization time and run-time, optimizers terminate after finding a sufficiently good join order. By comparison, Soufflé exhaustively runs its join optimizer at compile-time resulting in cost-optimal join orders, including for recursive rules. Additionally, Soufflé knows the rules apriori, allowing the join optimizer to collect only the necessary statistics, leading to a lightweight join optimization process that consistently produces high-quality join orders.
Auto-Scheduling for DSLs. There have been several compiling DSLs such as Halide [34], GraphIt [43], and TACO [28]. These DSLs separate the algorithm (what to compute) from the schedule (how to compute it). For instance, Halide allows users to provide schedules that control the degree of parallelization, vectorization, loop tiling and loop fusion. Auto-schedulers for these DSLs use sophisticated techniques [2,3] such as generating hundreds of thousands of random programs and schedules and training a machine-learned cost model for unseen schedules. Soufflé exposes only join-orders for optimization that are less sensitive to architectural details, allowing for simple cost models that rely on relational selectivities alone to achieve high-performance.

6 Conclusion

This paper presents an FDO strategy for join ordering targeting Datalog compilers. We have demonstrated that our optimizer is competitive with hand-tuned join orders while outperforming un-tuned orders by 12.07× on average. Our optimizer is lightweight, incurring an average slowdown of 2.61× for the slowest application during statistics collection. The join orders are also robust when

given a well-chosen, representative input. Overall, our join optimizer can save significant human effort spent manually tuning and enables users to achieve high-performance automatically without breaking declarativeness.

Acknowledgements. This work was generously supported by Fantom Foundation and by the Australian Government through the ARC Discovery Project funding scheme (DP210101984).

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases, vol. 8. Addison-Wesley Reading (1995)
2. Adams, A., et al.: Learning to optimize halide with tree search and random programs. ACM Trans. Graph. **38**(4), 1–12 (2019)
3. Anderson, L., Adams, A., Ma, K., Li, T.M., Jin, T., Ragan-Kelley, J.: Efficient automatic scheduling of imaging and vision pipelines for the GPU. In: Proceedings of the ACM on Programming Languages 5(OOPSLA), pp. 1–28 (2021)
4. Antoniadis, T., Triantafyllou, K., Smaragdakis, Y.: Porting doop to soufflé: a tale of inter-engine portability for datalog-based analyses. In: Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, pp. 25–30 (2017)
5. Aref, M., et al.: Design and implementation of the logicblox system. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1371–1382 (2015)
6. Astrahan, M.M., et al.: System R: relational approach to database management. ACM Trans. Graph. **1**(2), 97–137 (1976)
7. Backes, J., et al.: Reachability analysis for AWS-based networks. In: Dillig, Isil, Tasiran, Serdar (eds.) CAV 2019. LNCS, vol. 11562, pp. 231–241. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_14
8. Bancilhon, F.: Naive evaluation of recursively defined relations. In: Brodie, M.L., Mylopoulos, J. (eds) On Knowledge Base Management Systems. Topics in Information Systems. Springer, NY (1986). https://doi.org/10.1007/978-1-4612-4980-1_17
9. Beeri, C., Ramakrishnan, R.: On the power of magic. In: Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 269–284 (1987)
10. Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 169–190 (2006)
11. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pp. 243–262 (2009)
12. Bruno, N.: Automated Physical Database Design and Tuning, 1st edn. CRC Press Inc., Boca Raton (2011)
13. Ceri, S., Gottlob, G., Tanca, L., et al.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. Knowl. Data Eng. **1**(1), 146–166 (1989)

14. Codd, E.. F..: A relational model of data for large shared data banks. In: Broy, Manfred, Denert, Ernst (eds.) Software Pioneers, pp. 263–294. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-642-59412-0_16
15. Developers, S.: Soufflé documentation (2016). https://souffle-lang.github.io/pdf/abdulthesis.pdf
16. Developers, S.: Soufflé documentation (2016). https://souffle-lang.github.io/handtuning#profiler
17. Developers, S.: Soufflé release 2.3 (2022). https://github.com/souffle-lang/souffle/releases/tag/2.3
18. Flores-Montoya, A., Schulte, E.: Datalog disassembly. In: 29th USENIX Security Symposium (USENIX Security 2020), pp. 1075–1092 (2020)
19. Garcia-Molina, H., Widom, J., Ullman, J.D.: Database System Implementation. Prentice-Hall Inc., USA (1999)
20. Graefe, G.: The cascades framework for query optimization. IEEE Data Eng. Bull. **18**(3), 19–29 (1995)
21. Grech, N., Brent, L., Scholz, B., Smaragdakis, Y.: Gigahorse: thorough, declarative decompilation of smart contracts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1176–1186. IEEE (2019)
22. Henning, J.L.: SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput. Archit. News **34**(4), 1–17 (2006)
23. Hoder, Kryštof, Bjørner, Nikolaj, de Moura, Leonardo: μZ– an efficient engine for fixed points with constraints. In: Gopalakrishnan, Ganesh, Qadeer, Shaz (eds.) CAV 2011. LNCS, vol. 6806, pp. 457–462. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_36
24. Ioannidis, Y.E.: Query optimization. ACM Comput. Surv. **28**(1), 121–123 (1996)
25. Jarke, M., Koch, J.: Query optimization in database systems. ACM Comput. Surv. **16**(2), 111–152 (1984)
26. Jordan, Herbert, Scholz, Bernhard, Subotić, Pavle: SOUFFLÉ: on synthesis of program analyzers. In: Chaudhuri, Swarat, Farzan, Azadeh (eds.) CAV 2016. LNCS, vol. 9780, pp. 422–430. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_23
27. Jordan, H., Subotić, P., Zhao, D., Scholz, B.: A specialized b-tree for concurrent datalog evaluation. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 327–339 (2019)
28. Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The tensor algebra compiler. In: Proceedings of the ACM on Programming Languages 1(OOPSLA), pp. 1–29 (2017)
29. Lagouvardos, S., Dolby, J., Grech, N., Antoniadis, A., Smaragdakis, Y.: Static analysis of shape in TensorFlow programs. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming (ECOOP 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 15:1–15:29. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020)
30. Leis, V., et al.: Query optimization through the looking glass, and what we found running the Join Order Benchmark. VLDB J. **27**(5), 643–668 (2017). https://doi.org/10.1007/s00778-017-0480-7
31. Madsen, M., Yee, M.H., Lhoták, O.: From datalog to flix: a declarative language for fixed points on lattices. SIGPLAN Not. **51**(6), 194–208 (2016)
32. Meinel, C., Slobodová, A.: On the complexity of constructing optimal ordered binary decision diagrams. In: Prívara, I., Rovan, B., Ruzička, P. (eds.) MFCS 1994. LNCS, vol. 841, pp. 515–524. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58338-6_98

33. Neumann, T., Radke, B.: Adaptive optimization of very large join queries. In: Proceedings of the 2018 International Conference on Management of Data, pp. 677–692 (2018)
34. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Not. **48**(6), 519–530 (2013)
35. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in datalog. In: Proceedings of the 25th International Conference on Compiler Construction, pp. 196–206 (2016)
36. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Readings in Artificial Intelligence and Databases, pp. 511–522. Elsevier (1989)
37. Sellis, T.K.: Multiple-query optimization. ACM Trans. Database Syst. **13**(1), 23–52 (1988)
38. Seo, J., Guo, S., Lam, M.S.: Socialite: datalog extensions for efficient social network analysis. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 278–289. IEEE (2013)
39. Sereni, D., Avgustinov, P., de Moor, O.: Adding magic to an optimising datalog compiler. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 553–566. SIGMOD 2008, Association for Computing Machinery, NY (2008). https://doi.org/10.1145/1376616.1376673
40. Smith, M.D.: Overcoming the challenges to feedback-directed optimization (keynote talk). SIGPLAN Not. **35**(7), 1–11 (2000)
41. Subotić, P., Jordan, H., Chang, L., Fekete, A., Scholz, B.: Automatic index selection for large-scale datalog computation. Proc. VLDB Endow. **12**(2), 141–153 (2018)
42. Whaley, John, Avots, Dzintars, Carbin, Michael, Lam, Monica S..: Using datalog with binary decision diagrams for program analysis. In: Yi, Kwangkeun (ed.) APLAS 2005. LNCS, vol. 3780, pp. 97–118. Springer, Heidelberg (2005). https://doi.org/10.1007/11575467_8
43. Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., Amarasinghe, S.: GraphIt: a high-performance graph DSL. In: Proceedings of the ACM on Programming Languages 2(OOPSLA), pp. 1–30 (2018)
44. Zhao, D., Subotic, P., Raghothaman, M., Scholz, B.: Towards elastic incrementalization for datalog. In: Veltri, N., Benton, N., Ghilezan, S. (eds.) PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, 6–8 September 2021, pp. 20:1–20:16. ACM (2021)