Scaling-Up In-Memory Datalog Processing: Observations and Techniques

Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang,
Aws Albarghouthi, Paraschos Koutris, Jignesh M. Patel
University of Wisconsin-Madison
Madison, WI, USA
{zhiwei, jiangqiao, zuyu, aws, paris, jignesh}@cs.wisc.edu

ABSTRACT

Recursive query processing has experienced a recent resurgence, as a result of its use in many modern application domains, including data integration, graph analytics, security, program analysis, networking and decision making. Due to the large volumes of data being processed, several research efforts across multiple communities have explored how to scale up recursive queries, typically expressed in Datalog. Our experience with these tools indicate that their performance does not translate across domains—e.g., a tool designed for large-scale graph analytics does not exhibit the same performance on program-analysis tasks, and vice versa.

Starting from the above observation, we make the following two contributions. First, we perform a detailed experimental evaluation comparing a number of state-of-the-art Datalog systems on a wide spectrum of graph analytics and program-analysis tasks, and summarize the pros and cons of existing techniques. Second, we design and implement our own general-purpose Datalog engine, called RecStep, on top of a parallel single-node relational system. We outline the techniques we applied on RecStep, as well as the contribution of each technique to the overall performance. Using RecStep as a baseline, we demonstrate that it generally outperforms state-of-the-art parallel Datalog engines on complex and large-scale Datalog evaluation, by a 4-6X margin. An additional insight from our work is that it is possible to build a high-performance Datalog system on top of a relational engine, an idea that has been dismissed in past work.

1. INTRODUCTION

Recent years have seen a resurgence of interest from the industry and research community in the Datalog language and its syntactic extensions. Datalog is a recursive query language that extends relational algebra with recursion, and can be used to express a wide spectrum of modern data management tasks, such as data integration [7, 15], declarative networking [16], graph analysis [22, 23] and program analysis [29]. Datalog offers a simple and declarative interface to the developer, while at the same time allowing powerful optimizations that can speed up and scale out evaluation.

Development of Datalog solvers (or engines) has been a subject of study in both the database community and programming languages community. The database community independently developed its own tools that evaluate general Datalog programs, both in centralized and distributed settings. These include the LogicBlox solver [9], as well as distributed, cloud-based engines such as BigDatalog [25], Myria [26], and Socialite [23].

In the programming languages (PL) community, it has been observed that a rich class of fundamental static program analyses can be written equivalently as Datalog programs [20, 29]. The PL community has extensively implemented solvers that target all (or a subset) of Datalog. This line of research has resulted in several Datalog-based tools for program analysis, including BDDBDDB [28], Souffle [21], and more recently Graspan [27].

About a year ago, we started working on the problem of developing a scalable Datalog engine. We therefore tested state-of-the-art Datalog engines from both communities. Our experience with these tools indicate that their performance does not translate across domains: Tools such as LogicBlox and BDDBDDB were unable to scale well with large input datasets prevalent in other domains. Even Souffle, the best-performing tool for program analysis tasks, is not well-suited for tasks outside program analysis, such as graph analytics (which also require the language support for aggregation). BigDatalog, Myria, and Socialite, on the other hand, can support only simple Datalog programs with limited recursion capabilities (*linear and non-mutual recursion*)—however, the majority of program analyses typically correspond to Datalog programs with complex structures.

To address this divide, we ask two questions: (*i*) *what are the performance characteristics of existing parallel Datalog engines on tasks from different application domains? (ii) can we design and implement an efficient parallel general-purpose engine that can support a wide spectrum of Datalog programs?* To answer these questions, we perform a detailed experimental evaluation of different Datalog engines across tasks from graph analytics and program analysis, and compare their performance with our own in-memory paral-

lel Datalog engine, which uses a relational data management system as a backend. We systematically examine the techniques and optimizations necessary to transform a naïve Datalog solver into a highly optimized one. As a consequence of this work, we also show that—contrary to anecdotal and empirical evidence [13]—it is possible to use effectively an RDBMS as a backend for Datalog through careful consideration of the underlying system issues.

Our Contribution. In this paper, we make the following contributions:

1. **Benchmarking.** We perform an extensive comparison of four state-of-the-art Datalog engines on a multicore machine. We consider benchmarks both from the graph analysis domain (e.g., transitive closure, reachability, connected components), and the program analysis domain (e.g., points-to and dataflow analyses) using both synthetic and real-world datasets. We compare these systems across both runtime and memory usage. A summary of our findings is depicted in Table 1.

2. **Techniques and Guidelines.** We study the challenges of building a recursive query processing engine on top of a parallel RDBMS, and consider a spectrum of techniques that solve them. Key techniques include (*i*) a lightweight way to enable query re-optimization at every recursive step, (*ii*) careful scheduling of the queries issued to the RDBMS in order to maximize resource utilization, and (*iii*) the design of specialized high-performance algorithms that target the bottleneck operators of recursive query processing (set difference, deduplication). We also propose a specialized technique for graph analytics that can compress the intermediate data through the use of a bit-matrix to reduce memory usage. We systematically measure the effect of each technique on performance.

3. **Implementation.** We implement our techniques as part of RecStep, a Datalog engine that is built on top of QuickStep [19], which is a single-node in-memory parallel RDBMS. RecStep supports a language extension of pure Datalog with both stratified negation and aggregation, a language fragment that can express a wide variety of data processing tasks.

4. **Evaluation.** We experimentally show that RecStep can efficiently solve large-scale problems in different domains using a single-node multicore machine with large memory, and also can scale well when given more cores. RecStep outperforms in almost all the cases the other systems, sometimes even by a factor of 8. In addition, the single-node implementation of RecStep compares favorably to cluster-based engines, such as BigDatalog, that use far more resources (more processing power and memory). Our results show that (*i*) it is feasible to build a fast general-purpose Datalog engine using an RDBMS as a backend, and (*ii*) with the trend towards powerful (multi-core and large main memory) servers, single node systems may be sufficient for a large class of Datalog workloads.

Organization. In Section 3, we present the necessary background for Datalog and the algorithms used for its evaluation. Section 4 gives a summary of the architecture design of RecStep, while Section 5 discusses in details the techniques and optimizations that allow us to obtain high-performant evaluation. Finally, in Section 6 we present our extensive experimental evaluation.

2. RELATED WORK

We now compare and contrast our work with existing Datalog engines.

Distributed Datalog Engines. Over the past few years, there have been several efforts to develop a scalable evaluation engine for Datalog. Seo et al. [23] presented a distributed engine for a Datalog variant for social network analysis called Socialite. Socialite employs a number of techniques to enable distribution and scalability, including delta stepping, approximation of results, and data sharding. The notable limitation is Socialite's reliance on user-provided annotations to determine how to decompose data on different machines. Wang et al. [26] implement a variant of Datalog on the Myria system [11], focusing mostly on asynchronous evaluation and fault-tolerance. The BigDatalog system [25] is a distributed Datalog engine built on a modified version of Apache Spark. A key enabler of BigDatalog is a modified version of RDDs in Apache Spark, enabling fast set-semantic implementation. The BigDatalog work has shown superior results to the previously proposed systems which we discussed above, Myria and Socialite. Therefore, in our evaluation, we focus on BigDatalog for a comparison with distributed implementations. Comparing our work with the above systems, we focus on creating an optimized system atop a parallel in-memory database, instead of a distributed setting. The task of parallelizing Datalog has also been studied in the context of the popular MapReduce framework [2, 3, 24]. Motik et al. [18] provide an implementation of parallel Datalog in main-memory multicore systems.

Datalog Solvers in Program Analysis. Static program analysis is traditionally the problem of overapproximating runtime program behaviors. Since the problem is generally undecidable, program analyses resort to overapproximations of runtime facts of a program. A large and powerful class of program analyses, formulated as context-free language reachability, has been shown to be equivalent to Datalog evaluation. Thus, multiple Datalog engines have been built and optimized specifically for program-analysis tasks. The BDDBDDB Datalog solver [28] pioneered the use of Datalog in program analysis by employing binary decision diagrams (BDD) to compactly represent the results of program analysis. The idea is that there is lots redundancy in the results of a program analysis, due to overapproximation, that BDDs help in getting exponential savings. However, recently a number of Datalog solvers have been used for program analysis that employ tabular representations of data. These include the Souffle solver [21] and the LogicBlox solver [9]. Souffle (which has been shown to outperform LogicBlox [5]) compiles Datalog programs into native, parallel C++ code, which is then compiled and optimized for a specific platform. All of these solvers do not employ a deep form of parallelism that our work exhibits by utilizing a parallel in-memory database. The Graspan engine [27] takes a context-free grammar representation and is thus restricted to binary relations—graphs. However, Graspan employs worklist-based algorithm to parallelize fixpoint computation on a multicore machine. As we show experimentally, however, our approach can outperform Graspan on its own benchmark set.

Table 1: Summary of Comparison Between Different Systems. *CPU Efficiency* is defined as the reciprocal of the product of the overall performance (runtime) of the system supporting multi-core computation and the number of CPU cores given for computation - the greater number suggests higher CPU efficiency. We refer readers to [8] for more details.

	Graspan	Bddbddb	BigDatalog	Souffle	RecStep
Scale-Up	*yes*	*no*	*yes*	*yes*	*yes*
Scale-Out	*no*	*no*	*yes*	*no*	*no*
Memory Consumption	*low*	*low*	*high*	*medium*	*low*
CPU Utilization	*medium*	*poor*	*high*	*medium*	*high*
CPU Efficiency	*low*	*-*	*medium*	*high*	*high*
Hyperparameters Tuning	*yes*	*yes*	*yes*	*no*	*no*
Requried	*(lightweight)*	*(complex)*	*(moderate)*		
Mutual Recursion	*yes*	*yes*	*no*	*yes*	*yes*
Non-Recursive Aggregation	*no*	*no*	*yes*	*yes*	*yes*
Recursive Aggregation	*no*	*no*	*yes*	*no*	*yes*

Other Graph Engines. By now, there are numerous distributed graph-processing systems, like Pregel [17] and Giraph [12]. These systems espouse the *think-like-a-vertex* programming model where one writes operations per graph vertex. These are restricted to binary relations (graphs); Datalog, by definition, is more general, in that it captures computation over hypergraphs.

3. BACKGROUND

In this section, we provide the background on the syntax and evaluation strategies of the Datalog language. Then, we discuss the language extensions we use in this work.

3.1 Datalog Basics

A Datalog program P is a finite set of rules. A *rule* is an expression of the following form:

$$h :- p_1, p_2, \ldots, p_k.$$

The expressions $h, p_1, \ldots, p_k$ are *atoms*, i.e., formulas of the form $R(t_1, \ldots, t_\ell)$, where R is a table/relation name (predicate) and each t_i is a *term* that can be a constant or a variable. An atom is a *fact* (or *tuple*) when all t_i are constants. The atom h is called the *head* of the rule, and the atoms $p_1, \ldots, p_k$ are called the *body* of the rule. A rule can be interpreted as a logical implication: if the predicates $p_1, \ldots, p_k$ are true, then so is the head h. We assume that rules are always safe: this means that all variables in the head occur in at least one atom in the body. We will use the lower case $x, y, z, \ldots$ to denote variables, and $a, b, c, \ldots$ for constants.

The relations in a Datalog program are of two types: IDB and EDB relations. A relation that represents a base (input) table is called EDB (extensional database); EDB relations can occur only in the body of a rule. A relation that represents a derived relation is called IDB (intentional database), and must appear in the head of at least one rule.

EXAMPLE 1. *Let us consider the task of computing the transitive closure (TC) of a directed graph. We represent the directed graph using a binary relation* arc(x,y)*: this means that there is a directed edge from vertex* x *to vertex* y*. TC can be expressed through the following program:*

$$r_1 : \mathtt{tc(x,y)} :- \mathtt{arc(x,y)}.$$
$$r_2 : \mathtt{tc(x,y)} :- \mathtt{tc(x,z)}, \mathtt{arc(z,y)}.$$

In the above program, arc *is an* EDB *relation (input), and* tc *is an* IDB *relation (output). The program can be interpreted as follows. The first rule* r_1 *(called the* base rule*) initializes the transitive closure by adding to the initially empty relation all the edges of the graph. The second rule* r_2 *is a* recursive rule*, and produces new facts iteratively: a new fact* tc(a,b) *is added whenever there exists some constant* c *such that* tc(a,c) *is already in the relation (from the previous iterations), and* arc(c,b) *is an edge in the graph.*

Stratification. Given a Datalog program P, we construct its *dependency graph* G_P as follows: every rule is a vertex, and an edge (r, r') exists in G_P whenever the head of the rule r appears in the body of rule r'. A rule is *recursive* if it belongs in a directed cycle, otherwise it is called non-recursive. A Datalog program is recursive if it contains at least one recursive rule. For instance, the dependency graph of the program in Example 1 contains two vertices r_1, r_2, and two directed edges: (r_1, r_2) and (r_2, r_2). Since the graph has a self-loop, the program is recursive. A *stratification* of P is a partition of the rules into strata, where each stratum contains the rules that are in the same strongly connected component of G_P. The topological ordering of the strongly connected components also defines an ordering in the strata. In our running example, there exist two strata, $\{r_1\}, \{r_2\}$.

3.2 Datalog Evaluation

Datalog is a declarative query language, and hence there are different algorithms that can be applied to evaluate a Datalog program. Most implementations of Datalog use *bottom-up* evaluation techniques, which start from the input (EDB) tables, and then iteratively apply the rules until no more new tuples can be added into the IDB relations, reaching a *fixpoint*. In the *naïve evaluation* strategy, the rules are applied by using all the facts produced so far. For our running example, we would initialize the IDB relation tc with $\mathtt{tc}^0 \leftarrow \mathtt{arc}$. To compute the $(i+1)$ iteration, we compute $\mathtt{tc}^{i+1} \leftarrow \pi_{x,y}(\mathtt{tc}^i \bowtie \mathtt{arc})$. The evaluation ends when $\mathtt{tc}^{i+1} = \mathtt{tc}^i$. The naïve evaluation strategy has the drawback that the same tuples will be produced multiple times throughout the evaluation.

In the *semi-naïve evaluation* strategy, at every iteration the algorithm uses only the *new* tuples from the previous iteration to generate tuples in the current iteration. For instance, in the running example, at every iteration i, we maintain together with $\mathtt{tc}^i$ the facts that are generated only

in the i-th iteration (and not previous iterations), denoted by $\Delta\mathtt{tc}^i = \mathtt{tc}^i - \mathtt{tc}^{i-1}$. Then, we compute the $(i+1)$ iteration of $\mathtt{tc}$ as $\mathtt{tc}^{i+1} \leftarrow \pi_{x,y}(\Delta\mathtt{tc}^i \bowtie \mathtt{arc})$. The running example is an instance of *linear recursion*, where each recursive rule contains at most one atom with an IDB. However, many Datalog programs, especially in the context of program analysis, contain non-linear recursion, where the body of a rule contains multiple IDB atoms. In this case, the Δ relations are computed by taking the union of multiple subqueries (for more details see [1]).

Semi-naïve evaluation can be further sped up by exploiting the stratification of a Datalog program: the strata are ordered from lower to higher according to the topological order, and then each stratum is evaluated sequentially, by considering the IDB relations of prior strata as EDB relations (input tables) in the current stratum. In our implementation of Datalog, we use the semi-naïve evaluation strategy in combination with stratification.

3.3 Negation and Aggregation

In order to enhance the expressiveness of Datalog for use in modern applications, we consider two syntactic extensions: *negation* and *aggregation*.

Negation. Datalog is a monotone language, which means that it cannot express tasks where the output can become smaller when the input grows. However, many tasks are inherently non-monotone. To express these tasks, we extend Datalog with a simple form of negation, called *stratified negation*. In this extension, negation is expressed by adding the symbol $\neg$ in front of an atom. However, the use of $\neg$ is restricted syntactically, such that an atom $R(t_1, \ldots, t_\ell)$ can be negated in a rule if (i) R is an EDB, or (ii) any rule where R occurs in the head is in a strictly lower stratum.

EXAMPLE 2. *Suppose we want to compute the complement of transitive closure, in other words, the vertex pairs that do not belong in the closure. This task can be expressed by the following Datalog program with stratified negation:*

$$\mathtt{tc}(\mathtt{x},\mathtt{y}) \coloneq \mathtt{arc}(\mathtt{x},\mathtt{y}).$$
$$\mathtt{tc}(\mathtt{x},\mathtt{y}) \coloneq \mathtt{tc}(\mathtt{x},\mathtt{z}), \mathtt{arc}(\mathtt{z},\mathtt{y}).$$
$$\mathtt{node}(\mathtt{x}) \coloneq \mathtt{arc}(\mathtt{x},\mathtt{y}).$$
$$\mathtt{node}(\mathtt{y}) \coloneq \mathtt{arc}(\mathtt{x},\mathtt{y}).$$
$$\mathtt{ntc}(\mathtt{x},\mathtt{y}) \coloneq \mathtt{node}(\mathtt{x}), \mathtt{node}(\mathtt{y}), \neg\mathtt{tc}(\mathtt{x},\mathtt{y}).$$

Aggregation. We further extend Datalog with aggregation operators. To support aggregation, we allow the terms in the head of the rule to be of the form $\mathtt{AGG}(x, y, \ldots)$, where $x, y, \ldots$ are variables in the body of the rule, and $\mathtt{AGG}$ is an aggregation operator that can be $\mathtt{MIN}$, $\mathtt{MAX}$, $\mathtt{SUM}$, $\mathtt{COUNT}$, or $\mathtt{AVG}$. We allow aggregation not only in non-recursive rules, but inside recursion as well, as studied in [14]. In the latter case, one must be careful that the semantics of the Datalog program lead to convergence to a fixpoint; in this paper, we assume that the program given as input always converges ([30] studies how to test this property). As an example of the use of aggregation, suppose that we want to compute for each vertex the number of vertices that are reachable from this vertex. To achieve this, we can simply add to the TC Datalog program of Example 1 the following rule:

$$r_3 : \mathtt{gtc}(\mathtt{x}, \mathtt{COUNT}(\mathtt{y})) \coloneq \mathtt{tc}(\mathtt{x},\mathtt{y}).$$
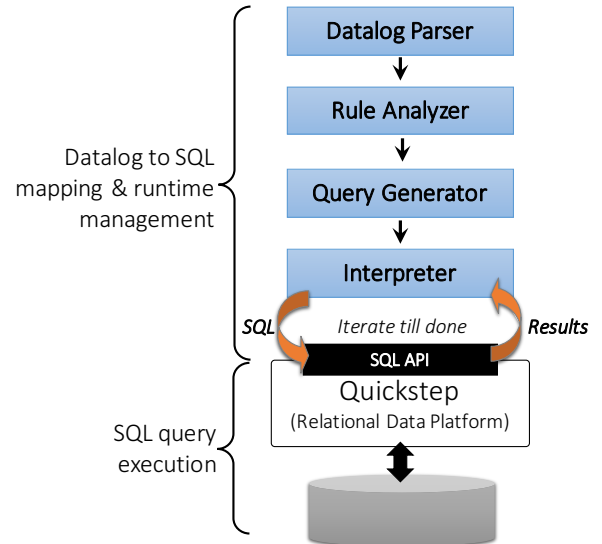

Figure 1: Architectural overview of RecStep

We should note here that it is straightforward to incorporate negation and aggregation in the standard semi-naïve evaluation strategy. Since negation can only be applied when there is no recursion, it can easily be encoded in a SQL query using the difference operator. Aggregation can be similarly encoded as group-by plus aggregation.

4. ARCHITECTURE

In this section, we present the architecture of RecStep. The core design choice of RecStep is that, in contrast to other existing Datalog engines, it is built on top of an existing parallel in-memory RDBMS (QuickStep). This design enables the use of existing techniques (e.g., indexing, memory management, optimized operator implementations) that provide high-performance query execution in a multicore environment. Further, it allows us to improve performance by focusing on characteristics specific to Datalog evaluation.

Overview. The architecture of our system is summarized in Figure 1. The Datalog program is read from a `.datalog` file, which, along with the rules of the Datalog program, specifies the schemas of IDB and EDB relations. The parsed program is first given as input to the *rule analyzer*. The job of the rule analyzer is to preprocess the program: identify the IDB and EDB relations, verify the syntactic correctness of the program, construct the dependency graph and stratification, and build the necessary mapping information (e.g., for joins). Next, the *query generator* takes the output of the rule analyzer and produces the necessary SQL code to evaluate each stratum of the Datalog program using semi-naïve evaluation. Finally, the *interpreter* is responsible for the evaluation of the program. It starts the RDBMS server, creates the IDB and EDB tables in the database, and takes care of the loop control for the semi-naïve evaluation in each stratum. It also controls the communication and flow of information between the RDBMS server.

Execution. We now describe how the interpreter executes a Datalog program, outlined in Algorithm 1.

The Datalog rules are evaluated in groups and order given by the stratification. The IDB relations are initialized so that they are empty (line 2). For each stratum, the interpreter

Table 2: Notation used in Algorithm 1

Function	Description
idb(s)	returns relations that are heads in stratum s
rules(R, s)	returns rules of stratum s with R as head
uieval(r)	evaluates all the rules in the set r
analyze(R)	call to the RDBMS to collect statistics for R
dedup(R)	deduplicates R

Algorithm 1 Execution Strategy for Datalog program P

1: **for each** IDB R **do**
2:     $R \leftarrow \emptyset$
3: // $\mathcal{S}$ is a stratification of P
4: **for each** stratum $s \in \mathcal{S}$ **do**
5:     **repeat**
6:         **for each** $R \in$ idb(s) **do**
7:             $R_t \leftarrow$ uieval(rules(R, s))
8:             analyze(R_t)
9:             $R_\delta \leftarrow$ dedup(R_t)
10:             analyze(R_δ, R)
11:             $\Delta R \leftarrow R_\delta - R$
12:             $R \leftarrow R \uplus \Delta R$
13:         **if** s is non-recursive **then**
14:             **break**
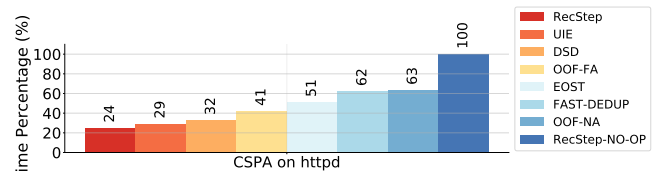15:   **until** $\forall R \in$ idb(s), $\Delta R = 0$

Figure 2: The effect of different optimizations techniques on the CSPA analysis on `httpd` dataset. The figure shows the effect of *turning off* each optimization on runtime, depicted as a percentage over the runtime of RecStep with all optimizations turned off (RecStep-NO-OP).

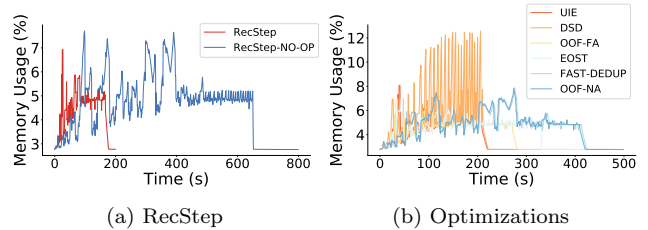(a) RecStep   (b) Optimizations

Figure 3: The effect of *turning off* each optimization of CSPA analysis on `httpd` on memory usage. The memory consumption with all optimization turned on/off is shown separately in Figure 3a for comparison purposes.

enters the control loop for semi-naïve evaluation. Note that in the case where the stratum is non-recursive (i.e., all the rules are non-recursive), the loop exits after a single iteration (line 13). In each iteration of the loop, two temporary tables are created for each IDB R in the stratum: ΔR, which stores only the new facts produced in the current iteration, and a table that stores the result at the end of the previous iteration. These tables are deleted right after the evaluation of the next iteration is finished.

The function uieval executes the SQL query that is generated from the query generator based on the rules in the stratum where the relation appears in the head (more details on the next section). We should note here that the deduplication does not occur inside uieval, but in a separate call (dedup). This is achieved in practice by using `UNION ALL` (simply appending data) instead of `UNION`.

Finally, we should remark that the interpreter calls the function analyze() during execution, which tells the backend explicitly to collect statistics on the specified table. As we will see in the next section, analyze() is a necessary feature to achieve dynamic and lightweight query optimization.

5. OPTIMIZATIONS

This section presents the key optimizations that we have implemented in RecStep to speed up performance and efficiently utilize system resources (memory and cores). We consider optimizations in two levels: Datalog-to-SQL-level optimizations and system-level optimizations.

For Datalog-to-SQL-level optimizations, we study the translation of Datalog rules to a set of SQL queries. An effective translation minimizes the overhead of catalog updates, selects the optimal algorithms and query plans, avoids redundant computations, and fully utilizes the available parallelism. In terms of system-level optimizations, we focus on bottlenecks that cannot be resolved by the translation-level

optimizations, and modify the back-end system by introducing new specialized data structures, implementing efficient algorithms, and revising the rules in the query optimizer.

We summarize our optimizations as follows:

1. *Unified IDB Evaluation* (UIE): different rules and different subqueries inside each recursive rule evaluating the same IDB relation are issued as a single query.
2. *Optimization On the Fly* (OOF): the same set of SQL queries are re-optimized at each iteration considering the change of IDB tables and intermediate results.
3. *Dynamic Set Difference* (DSD): for each IDB table, the algorithm for performing set-difference to compute Δ is dynamically chosen at each iteration, by considering the size of IDB tables and intermediate results.
4. *Evaluation as One Single Transaction* (EOST): the evaluation of a whole Datalog program is regarded as a single transaction and nothing is committed until the end.
5. *Fast Deduplication* (FAST-DEDUP): a memory-efficient implementation for high-performance deduplication.

Apart from the above list of optimizations, we also provide a specialized technique that can speed up performance on Datalog programs that operate on dense graphs called PBME. This technique represents the relation as a bit-matrix, with the goal of minimizing the memory footprint of the algorithm. We next detail each optimization; their effect on runtime and memory is visualized in Figure 2 and Figure 3.

5.1 Datalog-to-SQL-level optimizations

Unified IDB Evaluation (UIE). For each IDB relation R, there can be several rules where R appears in the head of the rule. In addition, for a nonlinear recursive rule, in which the rule body contains more than one IDB relation, the IDB relation is evaluated by multiple subqueries. In this case, instead of producing a separate SQL query for each rule and

Individual IDB Evaluation:

```
// Evaluate and write results separately
INSERT INTO pointsTo_tmp_mDelta0
    SELECT a0.y AS y, p1.x AS x FROM …
INSERT INTO pointsTo_tmp_mDelta1
    SELECT l0.y AS y, p2.x AS x FROM …;
    …
INSERT INTO pointsTo_tmp_mDelta6
    SELECT p1.x AS y, p2.x AS x FROM …;
// Merge results from seperate queries
INSERT INTO pointsTo_mDelta
    SELECT * FROM pointsTo_tmp_mDelta0
        UNION ALL
    SELECT * FROM pointsTo_tmp_mDelta1
        UNION ALL
        …
    SELECT * FROM pointsTo_tmp_mDelta6;
```

Unified IDB Evaluation:

```
// Evaluate and write results as a whole
INSERT INTO pointsTo_mDelta
    SELECT a0.y AS y, p1.x AS x FROM …
        UNION ALL
    SELECT l0.y AS y, p2.x AS x FROM …
        UNION ALL
        …
    SELECT p1.x AS y, p2.x AS x FROM …;
```

Figure 4: Example UIE in Andersen analysis.

then computing the union of the intermediate results, the query generator produces a *single* SQL query using the `UNION ALL` construct. We call this method *unified* IDB *evaluation* (UIE). Figure 4 provides an example of the two different choices for the case of Andersen analysis.

The idea underlying UIE is to fully utilize all the available resources, i.e., all the cores in a multi-core machine. Quick-Step does not allow the concurrent execution of SQL queries, and hence by grouping the subqueries into a single query, we maximize the number of tasks that can be executed in parallel without explicitly considering concurrent multi-task coordination. In addition, UIE mitigates the overhead incurred by each individual query call, and enables the query optimizer to jointly optimize the subqueries (e.g., enable cache sharing, pipelining instead of materializing intermediate results). The latter point is not specific to QuickStep, but generally applicable to any RDBMS backend (even ones that support concurrent query processing).

Optimization On the Fly (OOF). In Datalog evaluation, even though the set of queries is fixed across iterations, the input data to the queries changes, since the IDB relations and the corresponding Δ-relations change at every iteration. This means that the optimal query plan for each query may be different across different iterations. For example, in some Datalog programs, the size of ΔR (Algorithm 1) produced in the first few iterations might be much larger than the joining EDB table, and thus the hash table should be preferably built on the EDB when performing a join. However, as the ΔR produced in later iterations tend to become smaller, the build side for the hash table should be switched later.

In order to achieve optimal performance, it is necessary to re-optimize each query at every iteration (lines 8, 10 in Algorithm 1) by using the latest table statistics from the previous iteration. However, collecting the statistical data (e.g., size, avg, min, max) of the whole database at every iteration can cause a large overhead, since it may be necessary to perform a full scan of all tables. To mitigate this issue, our solution is to control precisely at which point which statistical data we collect for the query optimizer, depending on the type of the query. For instance, before joining two tables, only the size of the two tables is necessary for the optimizer to choose the right side to build the hash table on (the smaller table), as illustrated in the previous example.

In particular, we collect the following statistics:

- For *deduplication*, the size of the hash table needs to be estimated in order to pre-allocate memory. Instead of computing the number of distinct values in the table (which could be very expensive), we instead use a conservative approximation that takes the minimum of the available memory and size of the table.
- For *join processing*, we collect only the number of tuples and the tuple size of the joining tables, if any of the tables is updated or newly created.
- For *aggregation*, we collect statistics regarding the min, max, sum and avg of the tables.

The effect of OOF can be seen in Figure 4. Without updating the statistics across the iterations, the running time percentage jumps from 24% to 63% (OOF-NA). On the other hand, if we update the full set of statistics, the running time percentage increases to 41% (OOF-FA).

Dynamic Set-Difference (DSD). In semi-naïve evaluation, the execution engine must compute the *set difference* between the newly evaluated results (R_δ) and the entire recursive relation (R) at the end of every iteration, to generate the new $\Delta R \leftarrow R_\delta - R$ (line 12 in Algorithm 1). Since set difference is executed at every iteration for every IDB in the stratum, it is a computational bottleneck that must be highly optimized. There exist two different ways we can translate set difference as a SQL query.

The first approach (One-Phase Set Difference – OPSD) simply runs the set difference as a single SQL query. The default strategy that QuickStep uses for set difference is to first build a hash table on R, and then R_δ probes the hash table to output the tuples of R_δ that do not match with any tuple in the hash table. Since the size of R grows at each iteration (recall that Datalog is monotone), this suggests that the cost of building the hash table on R will constantly increase for the set difference computation under OPSD.

An alternative approach is to use a translation that we call Two-Phase Set Difference (TPSD). This approach involves two queries: the first query computes the intersection of the two relations, $r \leftarrow R \cap R_\delta$. The second query performs set difference, but now between R and r (instead of R_δ). Although this approach requires more relational operators, it avoids building a hash table on R.

We observe that none of the two approaches always dominates the other, since the size of R and R_δ changes at different iterations. Hence, we need to *dynamically* choose the best translation at every iteration. We guide this choice using a simple cost model, presented in full detail in [8].

5.2 System-level Optimizations

Evaluation as One Single Transaction (EOST). By default, QuickStep (as well as other RDBMSs) view each query that changes the state of database as a separate transaction. Keeping the default transaction semantics in Quick-Step during evaluation incurs I/O overhead in each iteration due to the frequent insertion happening to IDB tables, and the creation of tables storing intermediate results. Such frequent I/O actions are unnecessary, since we only need to commit the final results at the end of the evaluation. To avoid this overhead, we use the *evaluation as one single transaction* (EOST) semantics. Under these semantics, the data is kept in memory until the fixpoint is reached (when
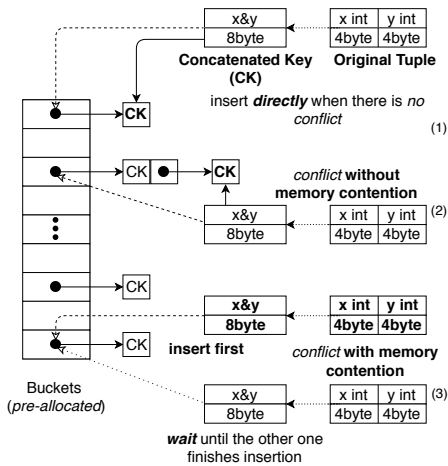
Figure 5: Example of applying fast deduplication algorithm on table with two integer attributes in RecStep

Algorithm 2 Parallel Bit-Matrix Evaluation of TC

1: **Input:** $arc(x, y)$ - EDB relation, number of threads k
2: **Output:** $tc(x, y)$ - IDB relation
3: // M_{arc}: virtual bit-matrix of $\texttt{arc(x, y)}$
4: Construct bit-matrix M_{tc} of $\texttt{tc(x, y)}$
5: $M_{tc} \leftarrow M_{arc}$
6: Partition the rows of M_{tc} into k partitions
7: // the k threads evaluate k partitions in parallel
8: **for each** row i in partition p **do**
9: $\quad \delta \leftarrow \{u \mid M_{tc}[i, u] = 1\}$
10: $\quad$ **while** $\delta \neq \emptyset$ **do**
11: $\quad\quad \delta_n \leftarrow \emptyset$
12: $\quad\quad$ **for each** $t \in \delta$ **do**
13: $\quad\quad\quad$ **for each** j s.t. $M_{arc}[t, j] = 1$ **do**
14: $\quad\quad\quad\quad$ **if** $M_{tc}[i, j] = 0$ **then**
15: $\quad\quad\quad\quad\quad \delta_n \leftarrow \delta_n \cup \{j\}$
16: $\quad\quad\quad\quad\quad M_{tc}[i, j] \leftarrow 1$
17: $\quad\quad \delta \leftarrow \delta_n$

there is enough main memory), and only the final results are written to persistent storage at the end of evaluation.

To achieve EOST, we slightly modify the kernel code in QuickStep to pend the I/O actions until the fixpoint is reached (by default, if some pages of the table are found *dirty* after a query execution, the pages are written back to the disk). At the end of the evaluation, a signal is sent to QuickStep and the data is written to disk. For other popular RDBMSs (e.g., PostgreSQL, MySQL, SQL Server), the start and the end of a transaction can be explicitly specified, but this approach is only feasible for a set of queries that are pre-determined.[1] However, in recursive query processing the issued queries are dynamically generated, and the number of iterations is not known until the fixpoint is reached, which means that similar changes need to be made in these systems to apply EOST.

Fast Deduplication. In Datalog evaluation, *deduplication* of the evaluated facts is not only necessary for conforming to the set semantics, but also helps to avoid redundant computation. Deduplication is also a computational bottleneck, since it occurs at every iteration for every IDB in the stratum (line 10 in Algorithm 1); hence, it is necessary to optimize its parallel execution.

To achieve this, we use a specialized *Global Separate Chaining Hash Table* implementation that uses a *Compact Concatenated Key* (CK), which we call CCK-GSCHT. CCK-GSCHT is a global latch-free hash table built using a compact representation of $\langle key, value \rangle$ pairs, in which tuples from each data partition/block can be inserted *in parallel*. Figure 5 illustrates the deduplication algorithm using an example in which CCK-GSCHT is applied on a table with two integer attributes (src int, dest int).

Based on the approximated number of distinct elements from the query optimizer, RecStep *pre-allocates* a list of buckets, where each bucket contains only a pointer. An important point here is that the number of pre-allocated buckets will be as large as possible when there is enough memory, for the purpose of minimizing conflicts in the same bucket, and preventing memory contention. Tuples are assigned to
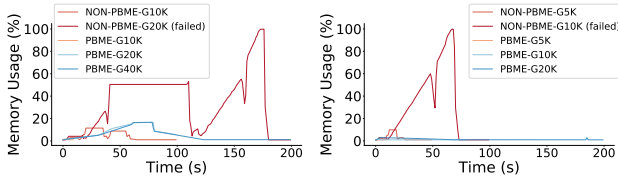
each thread in a round-robin fashion and are inserted in parallel. Knowing the length of each attribute in the table, a compact CK of fixed size [2] is constructed for each tuple (8 bytes for two integer attributes as shown in Figure 5). The compact CK itself contains all information of the original tuple, eliminating the need for explicit $\langle key, value \rangle$ pair representation. Additionally, the key itself is used as the hash value, which saves computational cost and memory space.

5.3 Parallel Bit-Matrix Evaluation

In our experimental evaluation, we observed that the usage of memory increases drastically during evaluation over dense relations. By default, QuickStep uses hash tables for joins between tables, aggregation and deduplication. When the intermediate result becomes very large, the use of hash tables for join processing becomes memory-costly. In the extreme, the intermediate results are too big to fit in main memory, and are forced to disk, incurring additional I/O overhead, or even out-of-memory errors. This phenomenon was observed in both graph analytics and program analysis. In both cases, a Datalog program starts with sparse input relations with a relatively small active domain, but end up with large and dense output relations. A typical example of this behavior is transitive closure on graphs.

Inspired by this observation, we exploit a specialized data structure, called *bit-matrix*, that replaces a hash map during join and deduplication in the case when the graph representing the data is *dense* and has relatively small number of vertices. This data structure represents the join results in a much more compact way under certain conditions, greatly reducing the memory cost compared to a hash table. In this paper, we only describe the bit matrix for binary relations, but the technique can be extended to relations of higher arity. At the same time, we implement new operators directly operating on the bit-matrix, naturally merging the join and deduplication into *one single operation* and thus avoid the materialization cost of the intermediate results. We call this

---

[1] To fully achieve EOST, transactional databases also need to turn off features such as checkpoint, logging for recovery, etc

[2] The inputs of Datalog programs are usually integers transformed by mapping the *active domain* of the original data (if not integers). Thus the technique can also applied to data where the original type has varied length.

(a) Transitive Closure (b) Same Generation
Figure 6: Memory Saving of PBME on TC and SG

(a) CPU Utilization (b) Memory Usage
Figure 7: PBME with Coordination v.s Non-Coordination on SG on the `G20K` dataset.

technique *Parallel Bit-Matrix Evaluation* (PBME). Our experiments (Figure 6) show that PBME improved performance for transitive closure (TC) and same generation (SG). The latter is expressed through the following program:

$$sg(x, y) :- arc(p, x), arc(p, y), x! = y.$$
$$sg(x, y) :- arc(a, x), sg(a, b), arc(b, y).$$

We next describe the bit-matrix technique and show how to apply it for *TC* (Algorithm 2). We discuss the algorithm for SG in the full paper [8]. Note that we construct a matrix only for each IDB, but for convenience of illustration, we use matrix notation for the EDBs as well (line 3 in Algorithm 2).

The Bit-Matrix Data Structure. Let $R(x, y)$ be a binary IDB relation, with active domain $\{1, 2, \ldots, n\}$ for both attributes. Instead of representing R as a set of tuples, we represent it as an $n \times n$ bit matrix denoted M_R. If $R(a, b)$ is a tuple in R, the bit at the a-th row and b-th column, denoted $M_R[a, b]$ is set to 1, otherwise it is 0. The relation is updated during recursion by setting bits from 0 to 1. We decide to build the bit-matrix data structure only if the memory available can fit both the bit matrix, as well as any additional index data structures used during evaluation.
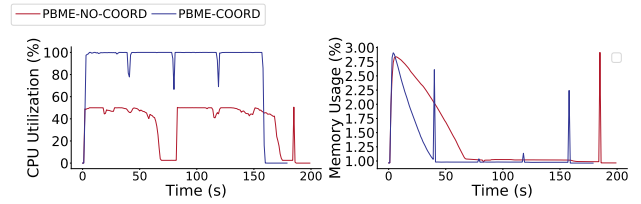
One of the key features of PBME is *zero-coordination*: each thread is only responsible for the partition of data assigned to it and there is almost no coordination between different threads. Algorithm 2 outlines the PBME for transitive closure. For the evaluation of TC (Algorithm 2), the rows of the IDB bit-matrix M_{tc} are firstly partitioned in a round-robin fashion (line 6). For each row i assigned to each thread, the set δ stores the new bits (paths starting from i) produced at every iteration (line 12-16). For each new bit t produced, the thread searches for all the bits at row t of M_{arc}, and computes the new δ (line 13-16).

The Effect of Skew. While unnoticeable in TC, data skew across the threads was observed in SG. To analyze the effect of data skew, we implement a variant of PBME with coordination (PBME-COORD) and compare it with PBME without coordination. PBME-COORD mitigates the data skew by rebalancing the workloads between threads. In the case where there is no skew, coordination only incurs unnecessary overhead. We refer readers to [8] for more detail.

The comparison of the two algorithms is shown in Figure 7. The CPU utilization of PBME with coordination is almost 100% throughout the whole evaluation of SG, and it takes less time to finish compared to PBME without coordination. This demonstrates that skew can indeed affect performance.

6. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of RecStep. Our experimental evaluation focuses on answering the following two questions:

1. How does our proposed system scale with increased computation power (cores) and data size?
2. How does our proposed system perform compared to other parallel Datalog evaluation engines?

To answer these two questions, we perform experiments using several benchmark Datalog programs from the literature: both from traditional graph analytics tasks (e.g., reachability, shortest path, connected components), as well as program analysis tasks (e.g., pointer static analysis). We compare RecStep against a variety of state-of-the-art Datalog engines, as well as a recent single-machine, multicore engine (Graspan), which can express only a subset of Datalog.

6.1 Experimental Setup

We briefly describe here the setup for our experiments.

System Configuration. Our experiments are conducted on a bare-metal server in Cloudlab [6], a large cloud infrastructure. The server runs Ubuntu 14.04 LTS and has two Intel Xeon E5-2660 v3 2.60 GHz (Haswell EP) processors. Each processor has 10 cores, and 20 hyper-threading hardware threads. The server has 160GB memory and each NUMA node is directly attached to 80GB of memory.

Other Datalog Engines. We compare the performance of RecStep with several state-of-the-art systems that perform either general Datalog evaluation, or evaluate only a fragment of Datalog for domain-specific tasks.

1. BigDatalog [25] is a general-purpose distributed Datalog system implemented on top of Apache Spark.[3]
2. Souffle [21] is a parallel Datalog evaluation tool that compiles Datalog to a native C++ program. It focuses on evaluating Datalog programs for the domain of static program analysis.[4]
3. BDDBDDB [28] is a single-thread Datalog solver designed for static program analysis. Its key feature is the representation of relations using binary decision diagrams (BDDs).
4. Graspan [27] is a single-machine disk-based parallel graph system, used mainly for interprocedural static analysis of large-scale system code.

---

[3]BigDatalog exhibits significant performance improvements over Myria and Socialite, and therefore we do not compare against them.

[4]Recent work has shown that Souffle outperforms LogicBlox [5]. Indeed, our early attempts using LogicBlox confirm that its performance is not comparable to other parallel Datalog systems. Thus, we exclude LogixBlox from our experimental evaluation.

Table 3: Summary of Datalog Programs and Datasets in Performance Evaluation

Graph Analytics	Datasets	Reference
Transitive Closure (TC)	[G5K, G10K, G10K-0.01, G10K-0.1, G20K, G40K, G80K]	[25]
Same Generation (SG)	[G5K, G10K, G10K-0.01, G10K-0.1, G20K, G40K, G80K]	[25]
Reachability (REACH)	[livejournal, orkut, arabic, twitter], [RMAT]	[25]
Connected Components (CC)	[livejournal, orkut, arabic, twitter], [RMAT]	[25]
Single Source Shortest Path (SSSP)	[livejournal, orkut, arabic, twitter], [RMAT]	[25]

Program Analysis	Datasets	Reference
Andersen's Analysis (AA)	7 synthetic datasets	-
Context-sensitive Dataflow Analysis (CSDA)	[linux, postgresql, httpd]	[27]
Context-sensitive Points-to Analysis (CSPA)	[linux, postgresql, httpd]	[27]

6.2 Benchmark Programs and Datasets

We conduct our experiments using Datalog programs that arise from two different domains: *graph analytics* and *static program analysis*. The graph analytics benchmarks are those used for evaluating BigDatalog [25]. Below, we present them in detail (with the exception of TC and SG, which are described earlier in the paper).

Reachability (REACH)

$$\texttt{reach}(\texttt{y}) :- \texttt{id}(\texttt{y}).$$
$$\texttt{reach}(\texttt{y}) :- \texttt{reach}(\texttt{x}), \texttt{arc}(\texttt{x},\texttt{y}).$$

Connected Components (CC)

$$\texttt{cc3}(\texttt{x}, \texttt{MIN}(\texttt{x})) :- \texttt{arc}(\texttt{x}, \_).$$
$$\texttt{cc3}(\texttt{y}, \texttt{MIN}(\texttt{z})) :- \texttt{cc3}(\texttt{x}, \texttt{z}), \texttt{arc}(\texttt{x},\texttt{y}).$$
$$\texttt{cc2}(\texttt{x}, \texttt{MIN}(\texttt{y})) :- \texttt{cc3}(\texttt{x}, \texttt{y}).$$
$$\texttt{cc}(\texttt{x}) :- \texttt{cc2}(\_, \texttt{x}).$$

Single Source Shortest Path (SSSP)

$$\texttt{sssp2}(\texttt{y}, \texttt{MIN}(0)) :- \texttt{id}(\texttt{y}).$$
$$\texttt{sssp2}(\texttt{y}, \texttt{MIN}(\texttt{d1} + \texttt{d2})) :- \texttt{sssp2}(\texttt{x}, \texttt{d1}), \texttt{arc}(\texttt{x},\texttt{y},\texttt{d2}).$$
$$\texttt{sssp}(\texttt{x}, \texttt{MIN}(\texttt{d})) :- \texttt{sssp2}(\texttt{x}, \texttt{d}).$$

The static analysis benchmarks include analyses on which Graspan was evaluated [27], as well as a classic static analysis called Andersen's analysis [4].

Andersen's Analysis (AA)

$$\texttt{pointsTo}(\texttt{y}, \texttt{x}) :- \texttt{addressOf}(\texttt{y}, \texttt{x}).$$
$$\texttt{pointsTo}(\texttt{y}, \texttt{x}) :- \texttt{assign}(\texttt{y}, \texttt{z}), \texttt{pointsTo}(\texttt{z}, \texttt{x}).$$
$$\texttt{pointsTo}(\texttt{y}, \texttt{w}) :- \texttt{load}(\texttt{y}, \texttt{x}), \texttt{pointsTo}(\texttt{x}, \texttt{z}), \texttt{pointsTo}(\texttt{z}, \texttt{w}).$$
$$\texttt{pointsTo}(\texttt{z}, \texttt{w}) :- \texttt{store}(\texttt{y}, \texttt{x}), \texttt{pointsTo}(\texttt{y}, \texttt{z}), \texttt{pointsTo}(\texttt{x}, \texttt{w}).$$

Context-sensitive Points-to Analysis (CSPA)[5]

$$\texttt{valueFlow}(\texttt{y}, \texttt{x}) :- \texttt{assign}(\texttt{y}, \texttt{x}).$$
$$\texttt{valueFlow}(\texttt{x}, \texttt{y}) :- \texttt{assign}(\texttt{x}, \texttt{z}), \texttt{memoryAlias}(\texttt{z}, \texttt{y}).$$
$$\texttt{valueFlow}(\texttt{x}, \texttt{y}) :- \texttt{valueFlow}(\texttt{x}, \texttt{z}), \texttt{valueFlow}(\texttt{z}, \texttt{y}).$$
$$\texttt{memoryAlias}(\texttt{x}, \texttt{w}) :- \texttt{dereference}(\texttt{y}, \texttt{x}), \texttt{valueAlias}(\texttt{y}, \texttt{z}),$$
$$\texttt{dereference}(\texttt{z}, \texttt{w}).$$
$$\texttt{valueAlias}(\texttt{x}, \texttt{y}) :- \texttt{valueFlow}(\texttt{z}, \texttt{x}), \texttt{valueFlow}(\texttt{z}, \texttt{y}).$$
$$\texttt{valueAlias}(\texttt{x}, \texttt{y}) :- \texttt{valueFlow}(\texttt{z}, \texttt{x}), \texttt{memoryAlias}(\texttt{z}, \texttt{w}),$$

---

[5]Graspan's analysis is context-sensitive via method cloning [29]—therefore, calling context does not appear in the rules, but in the data.

$$\texttt{valueFlow}(\texttt{w}, \texttt{y}).$$
$$\texttt{valueFlow}(\texttt{x}, \texttt{x}) :- \texttt{assign}(\texttt{x}, \texttt{y}).$$
$$\texttt{valueFlow}(\texttt{x}, \texttt{x}) :- \texttt{assign}(\texttt{y}, \texttt{x}).$$
$$\texttt{memoryAlias}(\texttt{x}, \texttt{x}) :- \texttt{assign}(\texttt{y}, \texttt{x}).$$
$$\texttt{memoryAlias}(\texttt{x}, \texttt{x}) :- \texttt{assign}(\texttt{x}, \texttt{y}).$$

Context-sensitive Dataflow Analysis (CSDA)

$$\texttt{null}(\texttt{x}, \texttt{y}) :- \texttt{nullEdge}(\texttt{x}, \texttt{y}).$$
$$\texttt{null}(\texttt{x}, \texttt{y}) :- \texttt{null}(\texttt{x}, \texttt{w}), \texttt{arc}(\texttt{w}, \texttt{y}).$$

To evaluate the benchmark programs, we use a combination of synthetic and real-world datasets, which are summarized in Table 3. To give a better view of the performance evaluation, we briefly summarize some of the datasets and corresponding Datalog programs here. For more details, readers can go to the reference in Table 3.
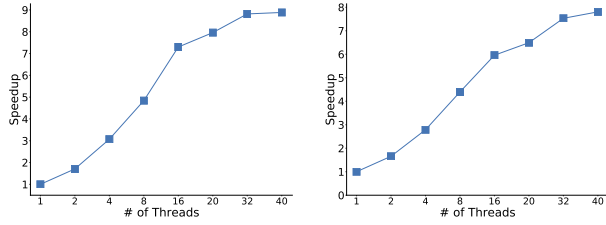
Gn-p graphs are graphs generated by the GTgraph synthetic graph generatorc [10], where n represents the number of total vertices of the graph in which each pair of vertices is connected by probability p. Each pair of vertices in Gn omitting p is connected with probability 0.001. All Gn-p graphs are very dense considering their relatively small number of vertices. SG and TC generate very large results when evaluation is performed on Gn-p, (at least a few orders of magnitude larger than the number of vertices). RMAT graphs are graphs generated by the RMAT graph generator [10], with the same specification in [25], RMAT-n represents the graph that has n vertices and $10n$ directed edges ($n \in \{1M, 2M, 4M, 8M, 16M, 32M, 64M, 128M\}$ in our evaluation experiments). livejournal, orkut, arabic, twitter are all large-scale real-world graphs which have tens of millions of vertices and edges. For the Andersen's analysis, seven datasets are generated ranging from small size to large size based on the characteristics of a tiny real dataset available at hand, numbered from 1 to 7. The graph representations of the datasets are small and produce moderate number of tuples. linux, postgresql, httpd are all real system programs used for CSDA and CSPA experiments in [27].
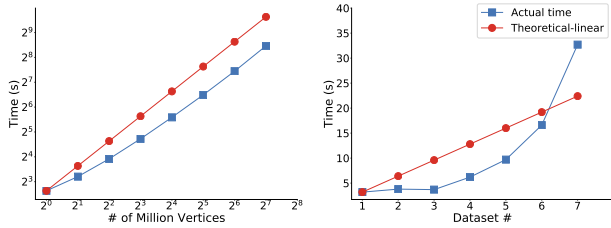
6.3 Experimental Results

We first evaluate the scalability of our proposed engine, and then move to a comparison with other systems.

6.3.1 Scalability

Scaling-up Cores. To evaluate the speedup of RecStep, we run the CC benchmark on livejournal graph,

(a) CSPA on `httpd`   (b) CC on `livejournal`

Figure 8: Scaling-up on Cores

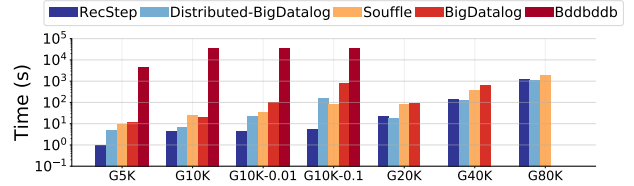(a) CC on `RMAT`   (b) AA on synthetic dataset

Figure 9: Scaling-up on Datasets: the x-axis of 9a represents the number of vertices of the corresponding `RMAT` graph datasets (`RMAT-1M` to `RMAT-128M`); the synthetic datasets are numbered from 1 to 7 and the x-axis of 9b suggests the corresponding dataset number.
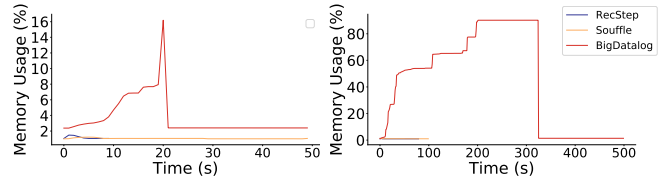

(a) Transitive Closure

(b) Same Generation

Figure 10: Performance Comparison of TC and SG

(a) Transitive Closure   (b) Same Generation

Figure 11: Memory Usage of TC and SG (`G10k`)

and the CSPA benchmark on the `httpd` dataset. We vary the number of threads from 2 to 40. Figure 8 demonstrates that for both cases, RecStep scales really well using up to 16 threads, and after that point achieves a much smaller speedup. This drop on speedup occurs because of the synchronization/scheduling primitive around the common shared hash table that is accessed from all the threads. Recall that 20 is the total number of physical cores.

Scaling-up Data. We perform experiments on both a graph analytics program (CC on `RMAT`) and a program analysis task (AA on the synthetic datasets) using all 20 physical cores (40 hyperthreads) to observe how our system scales over datasets of increasing sizes. From Figure 9a, we observe that the time increases nearly proportionally to the increasing size of the datasets. In Figure 9b, we observe that for datasets numbered from 1 to 3, the evaluation times on these three datasets are roughly the same. The corresponding graphs representing the input for these three datasets are relatively sparse and the total size of the data (input and intermediate results) during evaluation is small, and the cores/threads are underutilized; thus, when the data increases, the stale threads will take over the extra processing, and runtime will not increase. With the increasing size of datasets (4 to 7), we observe a similar trend as seen in Figure 9a since the size of input data as well as that of the produced intermediate results increases. All cores are fully utilized, so more data will cause increase in runtime.

6.3.2   Comparison With Other Systems

In this section, we report experimental results over our benchmarks for several other Datalog systems and Graspan. For each Datalog program and dataset shown in the comparison results, we run the evaluation four times (with

the exception of BDDBDDDB, since its runtime is substantially longer than all other systems across the workloads), we discard the first run and report the average time of the last three runs. For each system, we report the total execution time, including the time to load data from the disk and write data back to the disk. For BigDatalog, since its evaluation is *parameter workload dependent* based on the available resources provided (e.g., memory), and its performance depends on both of the supplied parameters, datasets and the programs, we tried different combinations of parameters (e.g., different join types) and report the best runtime numbers. For comparison purposes, we also display the results of BigDatalog that runs on the full cluster from [25] (Distributed-BigDatalog in Figure 10, 12, 13, which has 15 worker nodes with 120 CPU cores and 450GB memory in total.) As we will see next, the experiments show that our system can efficiently evaluate Datalog programs for both large-scale graph analytics and program analyses, by being able to efficiently utilize the available resources on a single node equipped with powerful modern hardware (multi-core processors and large memory). Specific runtime numbers are not shown in Figure 10, 15a due to the space limit.

TC and SG Experiments. For TC and SG, our system uses PBME as discussed in Section 5. Since the `Gn-p` graphs are very dense, in each iteration intermediate results of large sizes are produced. Hence, the original QuickStep operators run out of memory due to the high materialization cost and demand for memory. By using a *bit-matrix* data structure, the evaluation naturally fuses the join and deduplication into a single operation, avoiding the materialization cost and heavy memory usage. Our system is the only one that completes the evaluation for TC and SG on all `Gn-p` graphs (the runtime bar is not shown if the system fails to finish the evaluation due to OOM or timeout ($> 10h$) is observed). The evaluation time of all four systems is shown
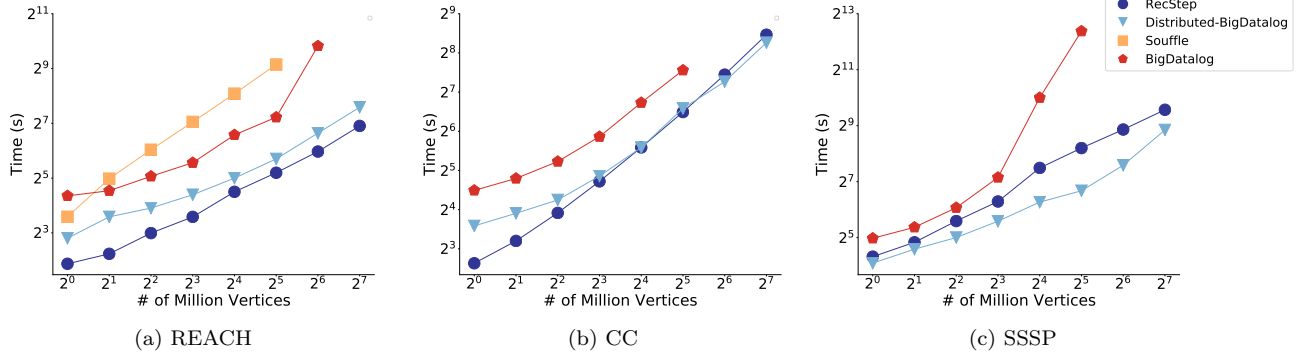
(a) REACH   (b) CC   (c) SSSP

Figure 12: Performance Comparison on `RMAT` Graphs: The x-axis represents `RMAT-1M` to `RMAT-128M`.

in Figure 10. Figure 11 shows the memory consumption of each system as percentage over the total memory given .

For TC, except for Distributed-BigDatalog, our system outperforms all other systems on all `Gn-k` graphs (Distributed-BigDatalog is only slightly faster than RecStep on `G20K`, `G40K` and `G80K`). For `G5K`, `G10K`, `G10K-0.01`, and `G10K-0.1`, our system even outperforms Distributed-BigDatalog, which uses 120 cores and 450GB memory. For graphs that have more vertices, Distributed-BigDatalog slightly outperforms our system due to the additional CPU cores and memory it uses for evaluation.

Due to the use of BDDs, BDDBDDB can only efficiently evaluate graph analytics expressed in Datalog when the graph has a relatively small number of vertices and when the proper variable ordering is given.[6] When the evaluation violates either of these two conditions, BDDBDDB is a few orders of magnitude slower than other systems as shown in graphs `G5K`, `G10K`, `G10K-0.01`, `G10K-0.1`. For graphs `G20K`, `G40K`, `G80K`, BDDBDDB runs out of time ($> 10h$). Souffle runs out of memory when evaluating TC on `G80K`.

Compared to TC, the evaluation of SG is more memory demanding and computationally expensive as observed in Figure 10b and 11. Except for our system, all other systems either use up the memory before the completion of the evaluation of SG or run into timeout ($> 15h$) on some of the `Gn-k` graphs. Unlike TC, we observe that RecStep on SG evaluation is much more sensitive to the graph density (e.g., `G10K`, `G10K-0.01`, `G10K-0.1`).

Experiments of Other Graph Analytics Besides TC and SG, we also perform experiments running REACH, CC and SSSP on both the `RMAT` graphs and the real world graphs (Table 3), comparing the execution time and memory consumption (on `livejournal`) of our system with Souffle and BigDatalog (Figure 13, 14). Since Soufflle does not support recursive aggregation (which shows in CC and SSSP), we only show the execution time results of our system and Big-Datalog for CC and SSSP. BDDBDDB is excluded, since the number of vertices of all the graphs is too large.

As mentioned in [25], the size of the intermediate results produced during the evaluation of REACH, CC, SSSP is $O(m)$, $O(dm)$ and $O(nm)$, where n is the number of vertices, m is the number of edges and d is the diameter of the graph. For convenience of comparison, we follow the way

in which [25] presents the experimental results: for REACH and SSSP, we report the average run time over ten randomly selected vertices. We only consider an evaluation complete if the system is able to finish the evaluation on *all ten vertices* for REACH and SSSP, otherwise, the evaluation is seen as *failed* (due to OOM). The corresponding point of failed evaluation is not reported in Figure 12 (on `RMAT`) and is shown as *Out of Memory* in Figure 13 (on real world graphs).

Besides Distributed-BigDatalog, RecStep is the only system that completes the evaluation for REACH, CC, SSSP on all `RMAT` graphs and real world graphs, and is 3-6X faster than other systems using scale-up approach on all the workloads that other systems manage to finish (as shown in Figures 12 and 13); compared to Distributed-BigDatalog, RecStep shows comparable performance using far less computational resources. Both BigDatalog and Souffle fail to finish evaluating some of the workloads due to OOM. As shown in Figure 12, the execution time of our system increases nearly proportionally to the increasing size of the dataset on all three graph analytics tasks. In contrast, Souffle's parallel behavior is workload dependent though it efficiently evaluates dataflow and points-to analysis (Fig 15b, 15c), it does not fully utilize all the CPU cores when evaluating REACH (Fig13a, 12a) and Andersen's analysis (Fig 15a) . The CPU utilization of different systems evaluating Andersen's analysis, CSPA is visualized in Figure 16.

Program Analysis. We perform experiments on Andersen's analysis using the synthetic datasets (generated based on a real-world dataset). Besides, we also conduct experiments comparing the execution time of CSPA and CSDA on the real system programs in [27]. Nonlinear recursive rules are commonly observed in Datalog programs for program analysis, and the results help us understand the behavior of our system and other systems when evaluating Datalog programs with(out) involving complex recursive rules.

For Andersen's analysis, the number of variables (the size of active domains of each EDB relation) increases from `dataset 1` to `dataset 7`. Our system outperforms all other systems on every dataset. The performance of BDDBDDB is comparable to other systems when the number of variables being considered is small (`dataset 1` and `dataset 2`), but the runtime increases a lot when the number of variables grows, due to its large overhead of building the BDD. BigDatalog outperforms Souffle on large datasets, since Souffle does not parallelize the computation as effectively.

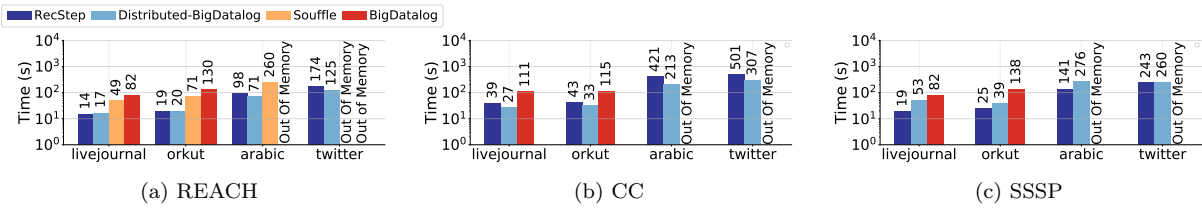All three systems significantly outperform Graspan on both CSPA and CSDA, as shown in Figure 15b and Figure

---

[6]The size of BDD is highly sensitive to the variable ordering used in the binary encoding; finding the best ordering is NP-complete. We let BDDBDDB pick the ordering.

(a) REACH   (b) CC   (c) SSSP

Figure 13: Performance Comparison on Real-World Graphs.

(a) REACH   (b) CC   (c) SSSP

Figure 14: Memory Consumption on `livejournal`.

(a) AA on 7 synthetic datasets   (b) CSDA on systems programs   (c) CSPA on systems programs

Figure 15: Performance Comparison on Program Analyses

(a) AA on `dataset 5`   (b) CSPA on `linux`   (c) CSPA on `httpd`

Figure 16: CPU Utilization on Program Analyses

15c. Since BigDatalog does not support mutual recursion, it is not present in Figure 15c). The inefficiency of Graspan is mainly due to its frequent use of sorting, coordination during the computation and relatively poor utilization of multi-cores for parallel computation.

CSDA is the only Datalog program on which RecStep is outperformed by Souffle and BigDatalog. The reasons are two-fold. First, the evaluation of CSDA on all three datasets needs many iterations (~ 1000), and thus the overhead of triggering each query accumulates. There is also an additional overhead from the `analyze` calls and the materialization cost of the intermediate results. Compared to this overhead, the cost of the actual computation is much smaller. The second reason is that the rules in CSDA are simple and linear. Since the input data and the intermediate results produced in each iteration is small in size, the RDBMS cannot fully utilize the available cores.

In contrast, CSPA has more rules and involves nonlinear recursion, producing large Δ and intermediate results at each iteration. This enables RecStep to exploit both data-level and multiquery-level parallelism. Figure 15c shows the

evaluation time for CSPA: while Souffle slightly outperforms our system on the `httpd` dataset, RecStep outperforms Souffle and Graspan on the other two datasets.

7. CONCLUSION

In this paper, we started with the observation that Datalog engines do not translate across domains. We experimentally evaluated the advantages and disadvantages of existing techniques, and compared them with our own baseline, a general-purpose, parallel, in-memory Datalog solver (RecStep) built upon a RDBMS. We presented the necessary optimizations and guidelines to achieve efficiency, and demonstrated that RecStep is scalable, applicable to a range of application domains, and is competitive with highly optimized and specialized Datalog solvers.

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[2] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *EDBT '11*, pages 1–8, 2011.

[3] F. N. Afrati and J. D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *EDBT '12*, pages 132–143, 2012.

[4] L. O. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, University of Cophenhagen, 1994.

[5] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis. 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*, pages 25–30. ACM, 2017.

[6] `https://www.cloudlab.us/`, 2018.

[7] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, pages 207–224, 2003.

[8] Z. Fan, J. Zhu, Z. Zhang, A. Albarghouthi, P. Koutris, and J. Patel. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. *arXiv e-prints*, page arXiv:1812.03975, Dec. 2018.

[9] T. J. Green, M. Aref, and G. Karvounarakis. Logicblox, platform and language: A tutorial. In *Datalog in Academia and Industry*, pages 1–8. Springer, 2012.

[10] `http://www.cse.psu.edu/~kxm85/software/GTgraph`.

[11] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the myria big data management service. In *SIGMOD '14*, pages 881–884, 2014.

[12] M. Han and K. Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015.

[13] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[14] A. Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. In *FGCS*, pages 915–925, 1992.

[15] M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246, 2002.

[16] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 97–108, 2006.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[18] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *AAAI '14*, pages 129–137, 2014.

[19] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.

[20] T. W. Reps. Program analysis via graph reachability. In J. Maluszynski, editor, *ILPS*, pages 5–19. MIT Press, 1997.

[21] 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*, CC 2016, pages 196–206, New York, NY, USA, 2016. ACM.

[22] J. Seo, S. Guo, and M. S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 278–289, 2013.

[23] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013.

[24] M. Shaw, P. Koutris, B. Howe, and D. Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In *Datalog 2.0*, pages 165–176, 2012.

[25] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD '16*, pages 1135–1149, 2016.

[26] J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

[27] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 389–404, New York, NY, USA, 2017. ACM.

[28] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 97–118, Berlin, Heidelberg, 2005. Springer-Verlag.

[29] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In W. Pugh and C. Chambers, editors, *PLDI*, pages 131–144. ACM, 2004.

[30] C. Zaniolo, M. Yang, M. Interlandi, A. Das, A. Shkapsky, and T. Condie. Declarative bigdata

algorithms via aggregates and relational database dependencies. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018.*, 2018.