Big Data Analytics with Datalog Queries on Spark

Alexander Shkapsky Mohan Yang Matteo Interlandi Hsuan Chiu
Tyson Condie Carlo Zaniolo
University of California, Los Angeles
{shkapsky, yang, minterlandi, cherylautumn, tcondie, zaniolo}@cs.ucla.edu

ABSTRACT

There is great interest in exploiting the opportunity provided by cloud computing platforms for large-scale analytics. Among these platforms, Apache Spark is growing in popularity for machine learning and graph analytics. Developing efficient complex analytics in Spark requires deep understanding of both the algorithm at hand and the Spark API or subsystem APIs (e.g., Spark SQL, GraphX). Our BigDatalog system addresses the problem by providing concise declarative specification of complex queries amenable to efficient evaluation. Towards this goal, we propose compilation and optimization techniques that tackle the important problem of efficiently supporting recursion in Spark. We perform an experimental comparison with other state-of-the-art large-scale DATALOG systems and verify the efficacy of our techniques and effectiveness of Spark in supporting DATALOG-based analytics.

Keywords

Datalog, Recursive Queries, Monotonic Aggregates, Spark

1. INTRODUCTION

Over the past decade, the demand for analytics has driven both researchers and industry to build cluster-based data analysis systems. Initially, the focus was on batch analysis and both research and industry proposed systems [3, 20, 26, 35] and languages [32, 51] supporting this endeavour. Recently, demand has exploded for analytics over graphs and networks. This has led researchers to refocus on providing scalable systems for machine learning and graph analytics. Among these systems is Apache Spark [4, 62], which is attracting a great deal of interest as a general platform for large-scale analytics, particularly because of its support for in-memory iterative analytics.

Some might think as a system designed for iterative applications, Spark would also be well suited for recursive applications such as shortest paths computations, and link and graph structure analysis. However this ignores three decades worth of recursive query evaluation and optimization techniques. Spark's support of recursion through iteration is inefficient: in an iterative Spark application, a new job is submitted for every iteration and thus the system has only limited visibility over an application's entire execution. From a programming perspective, the development of efficient recursive applications in Spark requires the programmer to have (1) deep understanding of the algorithm being implemented, (2) extensive knowledge of the Spark API, and (3) mastery of Spark internals. Nevertheless, Spark is a promising system for recursive applications because it provides many features essential for recursive evaluation, including dataset caching and low task startup costs. Along those lines, to examine how Spark can be made to efficiently support recursive applications we implement a recursive query language on Spark. Specifically, we implement DATALOG, a well known recursive query language. New interest has recently re-emerged around DATALOG for a wide spectrum of knowledge-oriented applications including distributed programming [25], AI [27], and distributed data management [64], as well as analytics on single-node systems [47, 49]. The fact that DATALOG is also well suited to declaratively support large-scale analytics was recently recognized by [48, 54].

In this paper we present BigDatalog, a full DATALOG language implementation on Apache Spark developed under the ***D****eductive* ***A****pplication* ***L****anguage* ***S****ystem* (*DeALS*) project [7] at UCLA. The *DeALS* project seeks to (1) design a unified logical language that enables the concise and declarative expression of analytics [41, 42], and (2) provide a system that optimizes execution over diverse platforms including sequential implementations [49], multi-core machines [59], and clusters (with BigDatalog). BigDatalog supports relational algebra, aggregation, and recursion, as well as a host of declarative optimizations. It also exploits semantic extensions for programs with *aggregation in recursion* [41, 42]. As a result, the Spark programmer can now implement complex analytics pipelines of relational, graph and machine learning tasks in a single language, instead of stitching together programs written in different APIs, i.e., Spark SQL [16], GraphX [33] and MLlib. Furthermore, BigDatalog employs techniques to identify and evaluate recursive programs that are *decomposable* and can be evaluated without communication [46, 57], leading to efficient distributed evaluations.

Motivating Example. As an example of the performance improvement that BigDatalog achieves for the evaluation of recursive queries consider Figure 1. This figure shows the execution time required to compute a 100 million vertex pair transitive closure of a graph using a highly optimized hand-

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915229

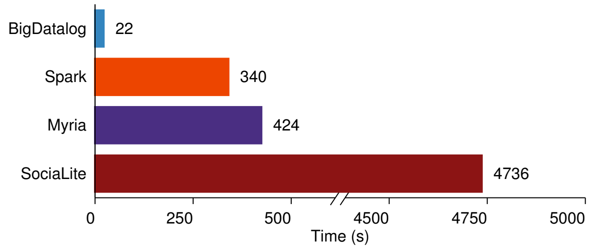


Figure 1: Example Recursive Query Performance.

written Spark program versus the BigDatalog version (cluster specs. in Section 6). This example shows how BigDatalog is both considerably better than its host framework and also performant *w.r.t.* large-scale DATALOG systems, namely, Myria [54] and SociaLite [48]. This orders of magnitude speed-up is achieved by employing the efficient evaluation techniques and optimizations of DATALOG in Spark.

Contributions. We make the following contributions:

- We design and implement the BigDatalog compiler. We show how BigDatalog programs are compiled into recursive physical plans for Spark.
- We present a parallel evaluation technique for distributed DATALOG evaluation on Spark. We introduce recursion operators and data structures to efficiently implement the technique in Spark.
- We propose physical planning and scheduler optimizations for recursive queries in Spark, including techniques to evaluate decomposable programs.
- We present distributed monotonic aggregates, and accompanying evaluation technique and data structures to support DATALOG programs with aggregates on Spark.
- We provide experimental evidence that a generic declarative system can compete with a special-purpose graph system.

Spark as a runtime for BigDatalog. In addition to its popularity and healthy ecosystem, Spark is a general data processing system and provides a SQL API; therefore, it is conducive to supporting a DATALOG compiler and DATALOG evaluation. Moreover, Spark's large and active user community helps to ease engineering effort. BigDatalog both benefits from and is limited by Spark's generality and overall system design principles. These tradeoffs will be discussed throughout the paper. Lastly, BigDatalog is designed for general analytical workloads, and although we will focus much of the discussion and experiments on graph queries and recursive program evaluation, we do not claim that Spark is the best platform for graph workloads in general. In fact, BigDatalog can also be backed by other general dataflow systems, including Naiad [44] and Hyracks [20], and many of the optimization techniques presented in this paper will also apply. This is the benefit of employing a declarative language, and comparing and analyzing the performance tradeoffs between different backend systems and/or computation models on various workloads would be an interesting future work.

Outline. In Section 2, we review DATALOG and Spark. Section 3 introduces BigDatalog and the distributed evaluation technique used to evaluate BigDatalog programs, and shows how BigDatalog programs are compiled into physical plans for execution on Spark. Section 4 presents evaluation, physical plan and job scheduling optimizations. Section 5 describes our aggregate design and implementation. Section 6 presents experimental results, including comparisons of BigDatalog with other large-scale DATALOG systems. Section 7 reviews related works. Section 8 presents conclusions and plans for future work.

2. PRELIMINARIES

In this section, we first provide background on DATALOG and then briefly review Spark. We will then tie the two together with an example and discuss challenges for using Spark as a DATALOG runtime.

2.1 Datalog

A DATALOG program is a finite set of rules. A *rule* r has the form $h \leftarrow b_1, \ldots, b_n$, where h is the *head* of r and $b_1, \ldots, b_n$ is the *body*. h and each b_i are *literals* with the form $p_i(t_1, \ldots, t_j)$ where p_i is a *predicate* and $t_1, \ldots, t_j$ are *terms* which can be *constants*, *variables* or *functions*. We say r is a rule of predicate h, unless it has an empty body, and then it is a *fact*. The comma separating literals in a body is a logical conjunction (AND). A successful assignment of all variables in the body produces a fact for the head's predicate. Predicates are also considered *relations* and WLOG throughout the paper we will use the terms predicate and relation interchangeably. A *query* indicates the desired predicate to evaluate. As a convention, predicate and function names begin with lower case letters, and variable names begin with upper case letters.

Datalog by Example. We continue our review of DATALOG with the *Transitive Closure* (TC) program. Program 1 recursively produces all pairs of vertices that are connected by some path in a graph.

PROGRAM 1. *Transitive Closure*

r1. tc(X, Y) ← arc(X, Y).
r2. tc(X, Y) ← tc(X, Z), arc(Z, Y).

Program 1 is explained as follows. *r1* is an *exit rule* because it serves as a base case of the recursion. In *r1*, the arc predicate represents the edges of the graph – arc is a *base relation*[1]. *r1* produces a tc fact for each arc fact. *r2* is a *recursive rule* since it has the tc predicate in both its head and body. *r2* will recursively produce tc facts from the conjunction[2] of previously produced tc facts and arc facts. The query to evaluate TC is of the form tc(X,Y). Lastly, this program uses a *linear* recursion in *r2*, since there is a single recursive predicate literal, whereas a *non-linear* recursion will have multiple recursive literals in its body. The number of iterations required to evaluate Program 1 is, in the worst case, equal to the longest simple path in the graph.

2.2 Datalog Evaluation

DATALOG is a declarative language and therefore rules are independent of the operators used to implement them (e.g., type of join used). Furthermore, rules are independent of the particular evaluation order and technique used as long as the monotonic *w.r.t.* set-containment[3], and least fixpoint,

[1] For readers not familiar with DATALOG, base relations can be seen as tables stored in a relational DBMS.

[2] In relational terminology, tc is joined with arc on Z – which is positionally interpreted as where the second argument of tc equals the first argument of arc.

[3] With set-containment monotonicity, evaluation only grows a predicate's set of facts.

semantics of DATALOG is maintained. Lastly, the order of literals in a rule body provides no semantic meaning and most implementations, including this work, evaluate literals in a left-to-right fashion.

A *naïve* evaluation of Program 1 will execute *r1* and then repeatedly evaluate *r2*, joining `arc` facts with already discovered `tc` facts in each iteration, until no new facts are produced – a *fixpoint* has been reached. This approach will inefficiently re-produce known facts in every iteration. We can instead use the well-known *Semi-Naïve evaluation* (SN) [18] which is efficient and produces no duplicates. Both naïve and SN evaluations are *bottom-up* evaluation techniques, which start from the initial database and perform a repeated application of the rules until a fixpoint is reached. Although SN is a centralized evaluation method, since it serves as the basis for the evaluation method used in this paper (cf. Section 3.3) we walk through an application of SN using Program 1 as our target. To enable SN, a (symbolic) rewriting [63] is applied to the rules of the original program to produce a new recursive rule that maintains program correctness. In the specific case of Program 1, the new rule only evaluates facts of the recursive predicate (`tc`) produced during the previous iteration (indicated with δ) and has the form $\mathtt{tc}(\mathtt{X},\mathtt{Y}) \leftarrow \delta\mathtt{tc}(\mathtt{X},\mathtt{Z}),\ \mathtt{arc}(\mathtt{Z},\mathtt{Y})$.

Algorithm 1 Semi-Naïve Evaluation of Program 1

```
1: $\delta\mathtt{tc} := \mathtt{arc}(\mathtt{X},\mathtt{Y})$
2: $\mathtt{tc} := \delta\mathtt{tc}$
3: do
4:     $\delta\mathtt{tc}' := \pi_{\mathtt{X},\mathtt{Y}}(\delta\mathtt{tc}(\mathtt{X},\mathtt{Z}) \bowtie \mathtt{arc}(\mathtt{Z},\mathtt{Y})) - \mathtt{tc}$
5:     $\mathtt{tc} := \mathtt{tc} \cup \delta\mathtt{tc}'$
6:     $\delta\mathtt{tc} := \delta\mathtt{tc}'$
7: while ($\delta\mathtt{tc} \neq \varnothing$)
8: return tc
```

Algorithm 1 shows the SN evaluation for Program 1. Note that the rules have been converted to a relational operator form (lines 1,4). In Algorithm 1 `tc` is the set of all facts produced for the recursive predicate and $\delta\mathtt{tc}$ ($\delta\mathtt{tc}'$) is the set of facts produced for `tc` during the previous (current) iteration. In SN, exit rules are evaluated first. The facts of `arc` become the initial set of facts for both $\delta\mathtt{tc}$ (line 1) and `tc` (line 2). Then, SN iterates until a fixpoint is reached (line 7). Each iteration begins by joining $\delta\mathtt{tc}$ with `arc` and projecting $\mathtt{X},\mathtt{Y}$ terms to produce candidate `tc` facts (line 4). These facts are then set-differenced with `tc` to eliminate duplicates and produce $\delta\mathtt{tc}'$ (line 4), which is unioned into `tc` (line 5) and becomes $\delta\mathtt{tc}$ (line 6).

2.3 Apache Spark

Spark provides a language-integrated Scala API enabling the expression of programs as *dataflows of transformations* (e.g., `map`, `filter`) on *Resilient Distributed Datasets* (RDD) [62]. An RDD is a distributed shared memory abstraction representing a partitioned dataset; RDDs are immutable, and transformations are coarse-grained and thus apply to all items in the RDD to produce a new RDD. Spark executes transformations lazily: a *job* is submitted for execution only when *actions* such as `count` or `reduce` are called by the user's program. Once a job is submitted, the scheduler groups transformations that can be pipelined (e.g., `map` over a `join`) into a single *stage*. The stages composing a dataflow are executed *synchronously* in a topological order: a stage will not be scheduled until all stages it is dependent upon have finished successfully. Between stages, Spark *shuffles* the dataset to *repartition* it among the nodes of the cluster. When a stage can be run, the scheduler creates a set of *tasks* (i.e., execution units) consisting of one task for each input *partition*, and launches the tasks on *worker nodes*.

```
1  var tc = sc.parallelize(graph, numPartitions)
2  val arcs = tc.map(x => (x._2, x._1)).cache()
3  var deltaTC = tc
4  do {
5    deltaTC = deltaTC.join(arcs)
6      .map(x => (x._2._2, x._2._1))
7     .subtract(tc).distinct().cache()
8    tc = tc.union(deltaTC).cache()
9  } while (deltaTC.count() > 0)
10 tc
```

Figure 2: Semi-naïve TC Spark Program.

RDDs can be explicitly cached by the programmer in memory or on disk at workers. Fault tolerance is provided by recomputing the sequence of transformations for the missing partition(s). Spark has libraries for structured data processing (Spark SQL), stream processing (Spark Streaming), machine learning (MLlib), and graph processing (GraphX).

Spark SQL. Spark's structured data and relational processing module, supports a subset of SQL. Spark SQL provides logical and physical relational operators. Spark SQL physical operators use a pipelined iterator model and are implemented as functions applied over the iterator from an upstream operator. The Catalyst framework [16] supports the compilation and optimization of Spark SQL programs into physical plans. In this work, we use and extend Spark SQL operators. We also propose BigDatalog operators that are implemented using the Catalyst framework so BigDatalog can use Catalyst planning features on recursive plans.

Iterative Spark Programs. Spark iterative applications are implemented by having a *driver program* iterate over a sequence of transformations terminated by an action. Each iteration is a new job that operates on cached RDD(s) produced by the previous iteration. Iteration terminates after a user-defined number of iterations or based on a user-defined predicate that determines when convergence is reached. Examples of algorithms supported by this approach include PageRank, logistic regression, and the semi-naïve transitive closure shown in Figure 2, and explained as follows. After some initial setup including distributing the graph among nodes of the cluster (line 1) and preparing the edges of the graph for joins (line 2), the program enters a `do-while` loop. It will iterate by executing a new job for each `count` action, until an iteration produces no new results (line 9). In each iteration, the program will join facts from the previous iteration (`deltaTC`) with `arcs` (line 5), project the pair of vertices (line 6), and eliminate duplicates (line 7). The set of all previously produced pairs is then combined with the newly produced pairs (line 8). Reused RDDs are cached.

Note the simplicity of the DATALOG program in Program 1 compared to the Spark program in Figure 2. Spark requires the programmer (1) be familiar with semi-naïve evaluation, (2) directly express a dataflow's physical plan composed of properly ordered operations and (3) handle memory management (RDD caching) to obtain better performance. Instead, BigDatalog enables high-level specification amenable to optimizations and rescues the programmer from extensive coding, debugging and maintenance effort.

Challenges for Datalog on Spark. The three main challenges we face with implementing DATALOG on Spark are:

1. Acyclic Plans: Supporting compilation, optimization and evaluation of DATALOG programs on Spark requires features not currently supported. A recursive, rule-based syntax requires a different compiler front-end than Spark SQL language queries. Spark SQL lacks recursion operators, operators are designed for acyclic use, and the Catalyst optimizer is targeted for non-recursive plans.

2. Scheduling: Spark's synchronous stage-based scheduler issues tasks for a stage only after all tasks of the previous stages have completed. For (monotonic) DATALOG programs, like the ones studied in this paper, this can be seen as unnecessary coordination because monotonic DATALOG programs are eventually consistent [15, 34].

3. RDD Immutability & Memory Utilization: An iteration of recursion will produce a new RDD to represent the updated recursive relation. This RDD will contain both new facts and all the facts produced in earlier iterations, which are already contained in earlier RDDs. If poorly managed, recursive applications on Spark can experience memory utilization problems.

3. BIGDATALOG

BigDatalog programs are expressed as DATALOG rules, then compiled, optimized and executed on Spark. BigDatalog will manage the persistence of datasets and make partitioning decisions. BigDatalog supports recursion, non-monotonic aggregation (min, max, sum, count, average) and aggregation in recursion with monotonic aggregates (Section 5).

3.1 Benchmark Programs

In this paper, we focus on monotonic (positive) programs which include classical recursive queries from the literature as well as aggregate queries, some of which are long studied (e.g., shortest paths) and others studied more recently (connected components) [48, 54].

Classical Recursive Queries

- **Transitive Closure** (TC)
- **Same Generation** (SG) identifies pairs of vertices where both are the same number of hops from a common ancestor.
- **Reachability** (REACH) produces all nodes connected by some path to a given source node.

Aggregation in Recursion Queries

- **Single-Source Shortest Paths** (SSSP) computes the length of the shortest path from a source vertex to each vertex it is connected to.
- **Connected Components** (CC) identifies connected components in the graph.

3.2 BigDatalog API By Example

The program snippet shown in Figure 3 computes the size of the transitive closure of the graph using the BigDatalog API for Spark. In a driver program, the user first gets a `BigDatalogContext` (line 1), which wraps the `SparkContext` (`sc`) – the entry point for writing and executing Spark programs. The user then specifies a database schema definition for base relations and program rules (lines 2-4). Lines 3-4 implement TC from Program 1. The database definition and rules are given to the BigDatalog compiler which loads the database schema into a relation catalog (line 5). Next, the data source (e.g., local or HDFS file path, or RDD) for the `arc` relation is provided (line 6). Then, the query to evaluate is given to the `BigDatalogContext` (line 7) which compiles it and returns an execution plan used to evaluate the query. As with other Spark programs, evaluation is lazy – the query is evaluated when `count` is executed (line 8).

```
1  val bdCtx = new BigDatalogContext(sc)
2  val program = "database({arc(X:Integer, Y:Integer})."
3    + "tc(X,Y) <- arc(X,Y)."
4    + "tc(X,Y) <- tc(X,Z), arc(Z,Y)."
5  bdCtx.datalog(program)
6  bdCtx.datasource("arc", filePath)
7  val tc = bdCtx.query("tc(X,Y).")
8  val tcSize = tc.count()
```

Figure 3: BigDatalog Program for Spark.

3.3 Parallel Semi-naïve Evaluation on Spark

BigDatalog programs are evaluated using a parallel version of SN we call *Parallel Semi-naïve evaluation* (PSN). PSN is an execution framework for a recursive predicate and it is implemented using RDD transformations. Since Spark evaluates synchronously, PSN will evaluate one iteration at a time, where an iteration will not begin until all tasks from the previous iteration have completed.

The two types of rules for a recursive predicate – the exit rules and recursive rules – are compiled into separate *physical plans* (plans) which are then used in the PSN evaluator. Physical plans are composed of Spark SQL and BigDatalog operators that produce RDDs. The exit rules plan is first evaluated once, and then the recursive rules plan is repeatedly evaluated until a fixpoint is reached. Note that as with SN, PSN will also evaluate symbolically rewritten rules (e.g., $\mathtt{tc}(\mathtt{X},\mathtt{Y}) \leftarrow \delta\mathtt{tc}(\mathtt{X},\mathtt{Z}),\ \mathtt{arc}(\mathtt{Z},\mathtt{Y}).$).

Algorithm 2 is the psuedocode for the PSN evaluator. The **exitRulesPlan** (line 1) and **recursiveRulesPlan** (line 5) are plans for the exit rules and recursive rules, respectively. We use `toRDD`() (lines 1,5) to produce the RDD for the plan. Each iteration produces two new RDDs – an RDD for the new results produced during the iteration (`delta`) and an RDD for all results produced thus far for the predicate (`all`). The `updateCatalog` (lines 3,8) stores new `all` and `delta` RDDs into a catalog for plans to access. The exit rule plan is evaluated first. The result is de-duplicated (`distinct`) (line 1) to produce the initial `delta` and `all` RDDs (line 2), which are used to evaluate the first iteration of the recursion. Each iteration is a new job executed by `count` (line 9). First, the **recursiveRulesPlan** is evaluated using the `delta` RDD from the previous iteration. This will produce an RDD that

Algorithm 2 PSN Evaluator with RDDs

```
 1: delta = exitRulesPlan.toRDD().distinct()
 2: all = delta
 3: updateCatalog(all, delta)
 4: do
 5:     delta = recursiveRulesPlan.toRDD()
 6:         .subtract(all).distinct()
 7:     all = all.union(delta)
 8:     updateCatalog(all, delta)
 9: while (delta.count() > 0)
10: return all
```

is set-differenced (`subtract`) with the `all` RDD (line 6) and de-duplicated to produce a new `delta` RDD. With lazy evaluation, the union of `all` and `delta` (line 7) from the previous iteration is evaluated prior to its use in `subtract` (line 6).

We have implemented PSN to cache RDDs that will be reused, namely `all` and `delta`, but we omit this from Algorithm 2 to simplify its presentation. Lastly, in cases of mutual recursion, when two or more rules belonging to different predicates reference each other (e.g., A ← B, B ← A), one predicate will "drive" the recursion with PSN and the other recursive predicate(s) will be an operator in the driver's recursive rules plan.

3.4 Compilation and Planning

For BigDatalog we have extended the *DeALS* compiler [49, 50], which was originally designed for sequential program evaluation, and we optimized and re-targeted it for parallel, distributed bottom-up evaluation of DATALOG programs. The input for the compiler is a database schema definition, a set of rules and a query. From this, the compiler creates a logical plan for the program, which is optimized using database techniques such as projection pruning. The logical plan for a non-recursive BigDatalog query is mapped into a Spark SQL plan and executed accordingly. Logical plans for recursive queries are converted to BigDatalog physical plans.

3.4.1 Logical Plans

Here, we use Program 1 (TC) to describe how the compiler produces a logical plan. Given the query `tc(X,Y)`, the program is first compiled into a Predicate Connection Graph (PCG) to identify the exit rules (*r1*) and recursive rules (*r2*) of the `tc` recursive predicate. The PCG is a type of AND/OR tree where OR nodes represent predicate occurrences in rule bodies and AND nodes represent rule heads [17]. From the PCG, the logical query plan is produced by mapping it into a tree of relational and recursion (i.e., fixpoint) operators. A recursion operator has two child logical (sub)plans: one plan for the predicate's exit rules and the other for the predicate's recursive rules. Figure 4(a) is the logical plan produced by the BigDatalog compiler for Program 1. The left side of Figure 4(a) is the exit rules plan with only the `arc` relation, representing *r1*. The right side of Figure 4(a) is the recursive rules plan made up of relational operators to produce one iteration of *r2*.

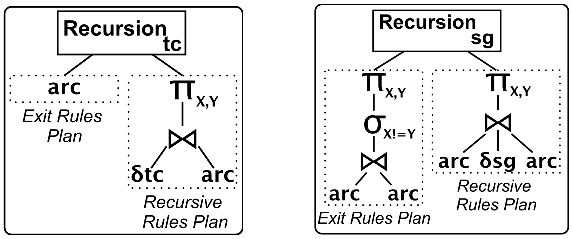
PROGRAM 2. *Same Generation*

r1. sg(X, Y) ← arc(P, X), arc(P, Y), X != Y.
r2. sg(X, Y) ← arc(A, X), sg(A, B), arc(B, Y).

Consider Program 2, the same generation (SG) program. The exit rule *r1* produces all `X,Y` pairs with the same parents (i.e. siblings) and the recursive rule *r2* produces new `X,Y` pairs where both `X` and `Y` have parents of the same generation. For PSN, *r2* is (symbolically) rewritten as sg(X, Y) ← arc(A, X), δsg(A, B), arc(B, Y). The left side of Figure 4(b) is the exit rules plan with a self-join of `arc` to find siblings. The right side of Figure 4(b) is the recursive rules plan which includes a three-way join of δsg and arc.

3.4.2 Physical Plans

BigDatalog translates logical plans into physical plans comprised of Spark SQL and BigDatalog physical operators. Like Spark SQL operators, BigDatalog operators use the Spark SQL `Row` type. Most logical-to-physical operator mapping is straightforward, however *recursion*, *join* and *shuffle* operators require discussion.



(a) TC (b) SG

Figure 4: BigDatalog Logical Plans.

Recursion Operators. The *Recursion Operator* (RO) is a special driver operator that runs on the master and executes PSN, i.e., the psuedocode from Algorithm 2. An RO has two child physical (sub)plans, the *Exit Rules Plan* (ERP) and the *Recursive Rules Plan* (RRP). A *Recursive Relation* operator represents a recursive predicate in the RRP and produces the recursive relation when evaluated (i.e., the plan version of a recursive predicate body literal).

Join Operators. BigDatalog uses binary hash join operators. We convert a multi-way join from logical plans into a hierarchy of binary join operators, in a left-to-right fashion. In a linear recursion, where only one join input is a recursive relation, the non-recursive input is loaded into lookup tables and the recursive relation is streamed. For instance, from the logical plan for TC in Figure 4(a), the RRP will have a join where δtc is streamed and arc is loaded into lookup tables. To help explain our approach for non-linear recursions, we use the following non-linear program. In this program, *r2* creates new tc facts of the form (X, Y) by joining tc facts of the form (X, Z) with tc facts of the form (Z, Y).

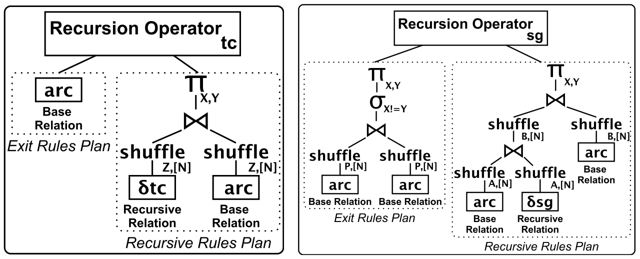
PROGRAM 3. *Non-Linear Transitive Closure*

r1. tc(X, Y) ← arc(X, Y).
r2. tc(X, Y) ← tc(X, Z), tc(Z, Y).

In Program 3, *r2* will be (symbolically) rewritten for SN, as tc(X, Y) ← δtc(X, Z), tc(Z, Y). Since both inputs to the join are recursive relations, δtc will be loaded into lookup tables and tc will be streamed. We choose this approach because loading the smaller of the two into lookup tables is less expensive and after a few iterations, tc is likely to be much larger than δtc.

Shuffle Operators. After mapping the logical operators into physical operators, the last step to produce a physical plan for execution is to add *shuffle operators* for distributed evaluation. Shuffle operators are used to repartition the dataset when there is a mismatch between an operator's required input partitioning and a child operator's output partitioning. For example, a shuffle operator is needed to repartition an input to a join if the input is not partitioned on the join keys. We use a Catalyst feature to analyze the physical plan and add shuffle operators where needed. We use hash partitioning and a static number of partitions throughout evaluation. Future work is to investigate dynamically adjusting the number of partitions during evaluation.

Example Plans. The physical plans produced for Program 1 (TC) and Program 2 (SG) are displayed in Figure 5(a) and 5(b), respectively. Using Figure 5(a) as our point of



(a) TC (b) SG

Figure 5: BigDatalog Physical Plans.

reference, we explain how our operators from above are used in plans. The root of the plan is the RO for the tc recursive predicate. In the RRP, δtc is a Recursive Relation and when evaluated will produce tc's facts from the previous iteration. Both inputs to the binary hash join are shuffled. The subscript Z, [N] indicates the partitioning key is the Z argument (from the rule), and there will be N partitions. Here Z is the join argument so that tuples of arc and δtc having the same key will be co-located on the same worker.

4. OPTIMIZATIONS

This section presents optimizations to improve the performance of BigDatalog programs. Details on the datasets used in experiments in this section can be found in Section 6.

4.1 Optimizing PSN

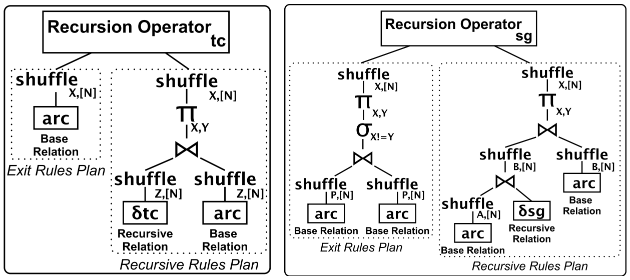
As shown with Algorithm 2, PSN can be implemented with RDDs and transformations such as subtract, distinct and union. However, using standard RDD transformations is inefficient because each iteration the results of the recursive rules are set-differenced with the entire recursive relation (line 6 in Algorithm 2), which is growing each iteration, and thus expensive data structures must be created each iteration. We propose instead to use the *SetRDD*, a specialized RDD for storing distinct Rows and tailored for set operations needed for PSN. Each partition of a SetRDD is a set data structure.

Monotonicity and RDD Immutability. We can apply an optimization enabled by DATALOG set-containment semantics to efficiently produce a new SetRDD from the union transformation. Although an RDD is intended to be immutable, we make SetRDD mutable under the union operation. The union mutates the set data structure of each SetRDD partition and outputs a new SetRDD comprised of these same set data structures. If a task performing union fails and must be re-executed, this approach will not lead to incorrect results because union is monotonic and facts can be added only once. This design saves system memory because only one set exists per partition across all iterations.

Table 1 displays the results of evaluating TC and SG with both PSN and PSN with SetRDD. PSN with SetRDD outperforms PSN significantly in all cases.

Table 1: PSN vs. PSN with SetRDD Performance

Time (s)	TC			SG		
	Tree17	Grid150	G10K	Tree11	Grid150	G10K
PSN	244	OOM	208	OOM	230	1129
PSN with SetRDD	41	134	20	59	61	130



(a) TC (b) SG

Figure 6: PSN with SetRDD Physical Plans.

4.2 Partitioning

The initial version of PSN used RDD transformations (e.g., distinct, subtract) that performed the necessary shuffling operations. That approach was sufficient to produce a correct result, but could be inefficient to evaluate. Now, SetRDD's diff and union transformations are designed to require properly partitioned input (i.e., they will not shuffle). Therefore, none of the transformations used in PSN will repartition inputs so shuffle operators need to be placed into ERP and RRP to produce properly partitioned output for PSN transformations. This approach allows for a simplified and generalized PSN evaluator and brings the insertion of shuffle operators to the workflow under the control of the BigDatalog compiler. With full control over shuffle operator placement, (i.e., communication decisions), BigDatalog can produce very efficient evaluations.

Earlier DATALOG research showed a good *partitioning strategy* (i.e., the arguments on which to partition) for a recursive predicate was important for efficient parallel evaluation [23, 30, 31, 56]. In general, we seek a partitioning strategy that limits shuffling. The default partitioning strategy employed by BigDatalog is to partition the recursive predicate on the *first argument*. Now we can produce plans for PSN that will terminate with a shuffle operator if the output of the plan does not match the partitioning strategy of the predicate. Figure 6(a) is the plan for Program 1 for PSN with SetRDD. With the recursive predicate (tc) partitioned on the first argument notice how both the ERP and RRP terminate with a shuffle operator.

User-Defined Partitioning. In the plan in Figure 6(a) δtc requires shuffling prior to the join since it is not partitioned on the join key (Z) because the default partitioning is the first argument (X). However, if the second argument were instead made the default, the inefficiency with Figure 6(a) would be resolved but then other programs such as SG in Figure 6(b) would suffer (δsg would require a shuffle prior to the join). Therefore, to support programs where the default partitioning will lead to inefficient execution, BigDatalog allows the user to define a recursive predicate's partitioning via a configuration option. For instance, by overriding the default partitioning and making tc's second argument the partitioning strategy, the shuffle for δtc before the join in Figure 6(a) will not be inserted to the plan. Table 2 shows the results of TC evaluated with the plan in Figure 6(a) versus the same plan, but using the second argument as tc's partitioning strategy. In fact, on all graphs from Table 6, the plan using the second argument matched or outperformed the other.

Table 2: Comparison of TC with Different Partitioning

Time (s)	Tree17	Grid250	G10K
1st Argument	41	370	20
2nd Argument	26	265	19

Table 3: Join Optimizations for Linear Recursion

Time (s)	TC		SG	
	Tree17	Grid250	Tree11	Grid250
Shuffle join no caching	26	265	59	107
Shuffle join caching	17	196	56	81
Broadcast join	53	197	45	54

4.3 Join Optimizations for Linear Recursion

Input Caching. Since we use a static number of partitions and because non-recursive inputs do not change during evaluation, for a *shuffle join* implementing a linear recursion, the non-recursive join input can be cached. This can lead to significant performance improvement since input partitions no longer have to be shuffled and loaded into lookup tables prior to the join each iteration. Table 3 shows the improved performance of caching. For TC on Tree17, the time for shuffling and loading lookup tables each iteration is significant even though there are only 17 iterations. Grid250 also benefits from caching because although the dataset is smaller, evaluation requires 500 iterations.

Broadcast Joins. For linear recursions, instead of shuffle joins, each partition of a recursive relation can be joined with an entire relation (*broadcast join*[4]). For both types of joins, the non-recursive input is loaded into a lookup table. For a broadcast join, the cost of loading the entire relation into a lookup table is amortized over the recursion because the lookup table is cached and then reused every iteration.

Figure 7 shows an RRP for Program 2 (SG) where the three-way join from the logical plan (Figure 4(b)) has been converted to a two-level broadcast join. In the event that a broadcast relation is used multiple times in a plan, as in Figure 7, BigDatalog will broadcast it once and share it among all broadcast join operators joining the relation.

Table 3 shows the results of using broadcast joins compared to shuffle joins for TC and SG. SG benefits on both graphs from using broadcast joins because three shuffles are eliminated from the plan and these graphs require minimal broadcast time. However, broadcast joins proved inefficient for Tree17 for TC — the job to load and broadcast the lookup table takes as long as the entire execution using shuffle joins. Nevertheless, broadcast join is the default join operator for linear recursion, and shuffle join can be selected via configuration setting.

4.4 Decomposable Programs

Previous research on parallel evaluation of Datalog programs determined some programs are *decomposable* and thus evaluable in parallel without redundancy (a fact is only produced once) and without processor communication or synchronization [57]. Techniques for evaluating decomposable programs are appealing for BigDatalog because data-parallel systems like Spark can scale to large numbers of cpu cores. Furthermore, mitigating the cost of synchronization and shuffling can lead to significant execution time speedup. However, even if a program is decomposable, the system still needs to be able to produce physical plans to evaluate it as such. We consider a BigDatalog physical plan decomposable if RRP has no shuffle operators.

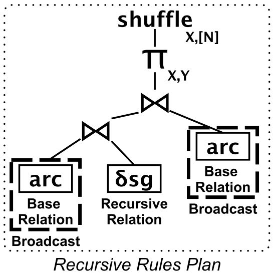


Figure 7: SG with Broadcast Joins.

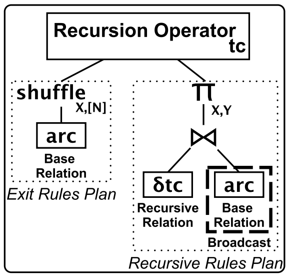


Figure 8: Decomposable TC Plan.

Program 1 (linear TC) is a decomposable program [57] however, its physical plan shown in Figure 6(a) has shuffle operators in RRP. BigDatalog will produce a decomposable physical plan for Program 1 by partitioning tc on the first argument and using a broadcast join. The partitioning strategy (first argument) divides the recursive relation so each partition can be evaluated independently and without shuffling, and the broadcast join allows each partition of the recursive relation to join with the *entire* arc base relation. Figure 8 is the decomposable physical plan for Program 1. Since we do not pre-partition base relations, the ERP has a shuffle operator to repartition the arc base relation into N partitions by arc's first argument X. Table 4 displays the execution times using the shuffle join plan and the decomposable plan (Figure 8). With the exception of Tree17, the decomposable plan greatly outperforms.

Table 4: Shuffle vs. Decomposable TC Plans

Time (s)	Tree17	Grid250	G10K	G10K-0.01	G20K
Shuffle	26	265	19	121	101
Decomposable	49	55	7	22	19

Identifying Decomposable Programs. BigDatalog identifies decomposable programs via syntactic analysis of program rules using techniques presented in the *generalized pivoting* work [46]. The authors of [46] show that the existence of a *generalized pivot set* (GPS) for a program is a sufficient condition for decomposablility and present techniques to identify GPS in arbitrary Datalog programs. We have implemented the techniques described in [46] to determine the GPS for BigDatalog programs. When a BigDatalog program is submitted to the compiler, the compiler will apply the generalized pivoting solver to determine if the program's recursive predicates have GPS. If they indeed do, we now have a partitioning strategy and in conjunction with broadcast joins we efficiently evaluate the program with these settings. For example, Program 1 has a GPS which says to partition the tc predicate on its first argument.

Note that this technique is enabled by using Datalog and allows BigDatalog to analyze the program at the logical level. The Spark API alone is unable to provide this support since programs are written in terms of physical operations.

4.5 Job Optimizations

Lineage. Since RDDs produced during an iteration are input for the next iteration, RDD lineage can grow long for

[4] Broadcast joins are supported with Spark's broadcast variable infrastructure.

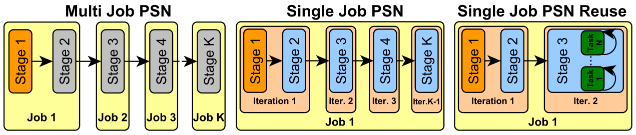


Figure 9: Program 1 (TC) Scheduling Options. ShuffleMapStages are orange; ResultStages are gray; FixpointStages are blue. Job 0 broadcasts the `arc` base relation.

recursive programs. Since lineage is inspected frequently during execution, for long running recursions we found this can result in a stack overflow. The standard solution is to checkpoint the RDD which clears the lineage after the RDD is written to disk. To optimize this, we implement a technique for cached RDDs that will clear lineage, but does not checkpoint. We sacrifice some degree of fault tolerance in favor of execution time performance, although this technique can still utilize cache replication. Otherwise, we leverage the standard fault tolerance mechanisms provided by Spark.

Scheduler-Aware Recursion. With PSN as shown in Algorithm 2, the scheduler is unaware that subsequent iterations could be required and therefore is unable to optimize recursive execution. To address this, we investigate pushing the recursion into the scheduler so recursive queries are supported as *Single-Job PSN*. We extend the Spark scheduler to use a special stage for recursion (*FixpointStage*) and support a *fixpoint job*, which is different from normal jobs in that 1) each iteration, the scheduler evaluates a new RDD over the previous iteration's results and 2) the scheduler will issue iterations until evaluation of the RDD results in an empty RDD indicating a fixpoint has been reached. We now refer to the original PSN (job per iteration) as *Multi-Job PSN*. To support checkpointing an iteration in Single-Job PSN, checkpointing is also pushed into the scheduler.

Optimizing Single-Job PSN. With Single-Job PSN, the scheduler is now aware that multiple iterations could be required. If a program is partitioned such that it does not require shuffling in the recursion, the scheduler will not create stages with shuffle operators. When the scheduler detects this situation, it configures the stage's tasks to iterate on workers and execute the same RDD until a fixpoint is reached. To support reusing the same RDD, the RDD partitions in the local cache from the previous iteration are overwritten with the RDD partitions produced during the current iteration. We call this *Single-Job PSN Reuse*. This approach eliminates the cost of scheduling and task creation for subsequent iterations. Figure 9 depicts the three different scheduling approaches for Program 1 (TC) evaluated with the plan from Figure 8.

Table 5 displays results of the execution times of TC and SG using the Multi-Job PSN, Single-Job PSN and Single-Job PSN Reuse. For datasets that require many iterations, such as `Grid250`, the performance improvement is substantial.

Table 5: Comparison of PSN Job Strategies

Time (s)	TC		SG	
	Tree17	Grid250	Tree11	Grid250
Multi-Job PSN	51	111	53	75
Single-Job PSN	49	55	53	53
Single-Job PSN Reuse	45	26	N/A	N/A

Note that this scheduler optimization is used on decomposable programs. Being able to identify a decomposable program (i.e., generalized pivoting) is independent from this optimization and thus this can be used in general to evaluate a decomposable plan, not just by BigDatalog.

5. AGGREGATES

BigDatalog supports non-monotonic aggregates (e.g., traditional SQL aggregates) `min`, `max`, `sum`, `count`, `avg`. As an example, consider Program 4, the BigDatalog program which counts the triangles in a graph, an important program in network analysis. In this non-recursive program, *r1* performs self-joins of `arc` to produce triangle occurrences which are then counted by *r2*. Lastly, note that although this program is expressed as two DATALOG rules, this program is a 50+ line GraphX program.

PROGRAM 4. *Triangle Counting*

$r1.\, \mathtt{triangles(X,Y,Z)} \leftarrow \mathtt{arc(X,Y)}, \mathtt{X} < \mathtt{Y}, \mathtt{arc(Y,Z)}, \mathtt{Y} < \mathtt{Z}, \mathtt{arc(Z,X)}.$
$r2.\, \mathtt{count\_triangles(count\langle\_\rangle)} \leftarrow \mathtt{triangles(X,Y,Z)}.$

However, non-monotonic aggregates cannot be used in recursion. Researchers have recently proposed aggregates that are monotonic w.r.t. set containment, the same monotonicity used by standard DATALOG, meaning these aggregates can be used in recursive rules and evaluated using techniques such as SN and magic sets [41, 42]. We have presented a sequential version of these aggregates in [49], whereas in this paper, we present a distributed version of the aggregates.

BigDatalog supports four monotonic aggregates - `mmin`, `mmax`, `mcount`, `msum`. The declarative semantics allows the aggregates inside the recursion so long as monotonicity w.r.t. set containment is maintained. Therefore, during evaluation the monotonic aggregates can produce new higher (`mmax`, `mcount`, `msum`) or lower (`mmin`) values with each input fact and thus an outer non-monotonic aggregate (`min` or `max`) is necessary to produce only the final value. An example of this can be seen in Program 5, the single-source shortest paths program (SSSP). Note, BigDatalog uses aggregates functions in rule heads with the non-aggregate arguments as the grouping arguments.

PROGRAM 5. *Single-Source Shortest Paths*

$r1.\, \mathtt{sssp2(Y,mmin\langle D\rangle)} \leftarrow \mathtt{Y} = 1, \mathtt{D} = 0.$
$r2.\, \mathtt{sssp2(Y,mmin\langle D\rangle)} \leftarrow \mathtt{sssp2(X,D1)}, \mathtt{arc(X,Y,D2)}, \mathtt{D} = \mathtt{D1} + \mathtt{D2}.$
$r3.\, \mathtt{sssp(X,min\langle D\rangle)} \leftarrow \mathtt{sssp2(X,D)}.$

The SSSP program computes the length of the shortest path from a source vertex to all vertices it is connected to. This program uses a `mmin` monotonic aggregate. Here the `arc` predicate in *r2* denotes edges of the graph (`X`, `Y`) with edge cost `D2`. *r1* seeds the recursion with starting vertex 1. Then, *r2* will recursively produce all new minimum cost paths to a node `Y` though node `X`. Lastly, *r3* produces only the minimum cost path for each node `X`, however in our actual implementation, we do not have to evaluate *r3* since at the completion of the recursion, `sssp2`'s relation will contain the shortest path from 1 to each vertex.

Evaluation and Implementation. Programs with monotonic aggregates in recursive rules are evaluated with an aggregate version of PSN we call *Parallel Semi-naive - Aggregate* (PSN-A). Compared with PSN, PSN-A is a simpler evaluator. Since new facts are only produced when a greater (`mmax`, `mcount`, `msum`) or lesser (`mmin`) value than the previous value for the (aggregate) group is produced, de-duplication is unnecessary. Furthermore, the union is unnecessary because new results are added to the aggregate relation during aggregate evaluation. We implement PSN-A in an aggregate version of an RO. Also, we use a specialized RDD called an *AggregateSetRDD*, in which each partition is a key value map where each entry represents a unique group and its current value. Caching AggregateSetRDD avoids the expense of reloading key value maps each iteration for aggregate. Additionally, since the aggregate functions are monotonic, as with SetRDD's `union` operation, AggregateSetRDD is mutable under aggregate evaluation. AggregateSetRDD will reference the same maps as its creator. Should a task fail during evaluation, any changes to the aggregate partition will not result in incorrect results since a value can only be updated if it is higher (`mmax`, `mcount`, `msum`) or lower (`mmin`) than previously computed values.

6. EXPERIMENTS

Experimental Setup. Our experiments are conducted on a 16 node cluster. Each node runs Ubuntu 14.04 LTS and has an Intel i7-4770 CPU (3.40GHz, 4 core/8 thread), 32GB memory and a 1 TB 7200 RPM hard drive. Nodes of the cluster are connected with 1Gbit network. Our implementation is in Spark 1.4.0 and uses Hadoop 1.0.4.

Table 6: Parameters of Synthetic Graphs

Name	**Vertices**	**Edges**	TC	SG
`Tree11`	71,391	71,390	805,001	2,086,271,974
`Tree17`	13,766,856	13,766,855	237,977,708	————
`Grid150`	22,801	45,300	131,675,775	2,295,050
`Grid250`	63,001	125,500	1,000,140,875	10,541,750
`G5K`	5,000	24,973	24,606,562	24,611,547
`G10K`	10,000	100,185	100,000,000	100,000,000
`G10K-0.01`	10,000	999,720	100,000,000	100,000,000
`G10K-0.1`	10,000	9,999,550	100,000,000	100,000,000
`G20K`	20,000	399,810	400,000,000	400,000,000
`G40K`	40,000	1,598,714	1,600,000,000	1,600,000,000
`G80K`	80,000	6,399,376	6,400,000,000	6,400,000,000

Table 7: Parameters of Real World Graphs

Name	**Vertices**	**Edges**	**Source**
`livejournal`	4,847,572	68,993,773	[9, 58]
`orkut`	3,072,441	117,185,083	[10, 58]
`arabic`	22,744,080	639,999,458	[5, 19]
`twitter`	41,652,231	1,468,365,182	[36]

Datasets. Table 6 shows the synthetic graphs used for TC and SG experiments. We use these graphs to understand how BigDatalog evaluates TC and SG on graphs exhibiting specific structural properties. `Tree11` and `Tree17` are trees of height 11 and 17 respectively, and the degree of a non-leaf vertex is a random number between 2 and 6. `Grid150` is a 151 by 151 grid while `Grid250` is a 251 by 251 grid. The Gn-p graphs are n-vertex random graphs (Erdős-Rényi model) generated by randomly connecting vertices so that each pair is connected with probability p. Gn-p graph names omitting p use default probability 0.001. Note that although these graphs appear small in terms of number of vertices and edges, TC and SG are capable of producing result sets many orders of magnitude larger than the input dataset, as shown by the last two columns in Table 6.

We perform experiments on REACH (Program 7), CC (Program 8) and SSSP (Program 5) using both real world and synthetic graphs. The real world graphs are displayed in Table 7. The synthetic graphs, `RMAT`-n for $n \in$ {1M, 2M, 4M, 8M, 16M, 32M, 64M, 128M}, are generated by the RMAT graph generator [8] with parameters (a, b, c) = (0.45, 0.25, 0.15). `RMAT`-n has n vertices and $10n$ directed edges with uniform integer weights range from $[0, 100)$.

6.1 Benchmark Comparison

In this section, we report experimental results over the benchmark programs of Section 3.1. We compare BigDatalog with other distributed DATALOG systems, namely *Myria* [54] and *SociaLite* [48], and with options available in the Spark stack. The purpose of this comparison is two-fold. Firstly, it shows how the enhancements and optimizations proposed in this paper enable Spark to serve as an efficient runtime for DATALOG. Secondly, it shows how our BigDatalog implementation performs *w.r.t.* to other DATALOG systems.

For each system, one machine was dedicated as the master and each of the 15 worker nodes was allowed 30 GB RAM and 8 CPU cores (120 total cores). Myria was configured with one instance of Myria and PostgreSQL per node, since each node has one disk, which was confirmed as appropriate by an author of [54]. For Spark programs and BigDatalog, we evaluate with one partition per available CPU core. BigDatalog uses Single-Job PSN with SetRDD.

6.1.1 TC *and* SG *Experiments*

For TC, BigDatalog uses Program 1 with the decomposed plan from Figure 8. We use the Program 1 equivalent in Myria and Socialite, and a hand-optimized semi-naïve program written in the Spark API which is implemented to minimize shuffling. For SG, BigDatalog uses Program 2 with the plan with broadcast joins whose RRP is shown in Figure 7. We use the Program 2 equivalent in Myria and Socialite, and we also implement a hand-optimized semi-naïve program in the Spark API that attempts to minimize shuffling.

BigDatalog is the only system that finishes the evaluation for TC and SG on all graphs in Table 6, except SG on `Tree17` since the size of the result is larger than the total disk space of the cluster. Figure 10 shows the evaluation time for all four systems, while the results for graphs that only BigDatalog is capable of handling are not displayed. We now go into details for each program.

TC. BigDatalog has the fastest execution time on six of the seven graphs for TC; on four of the graphs it outperforms the other systems by an order of magnitude. The BigDatalog plan only performs an initial shuffle of the dataset, and then evaluates the recursion without shuffling, and proves very efficient. In the case of `Grid150`, which is the smallest graphs used in this experiment, in terms of both edges and transitive closure size, Myria outperforms BigDatalog. This is explained as the evaluation requires many iterations (300), where each iteration performs very little work, and therefore the overhead of scheduling in BigDatalog takes a significant portion of execution time. Note however that if BigDatalog instead evaluates `Grid150` with Single-Job PSN Reuse (Section 4.5), the execution time drops to thirteen seconds.

The Spark program is also affected by the overhead of scheduling on `Grid150` and `Grid250`, which requires 300 and 500 iterations, respectively, but also suffers memory utilization issues related to dataset caching and therefore runs out of memory. For the remaining five graphs, the Spark program is slower compared with BigDatalog due to the over-

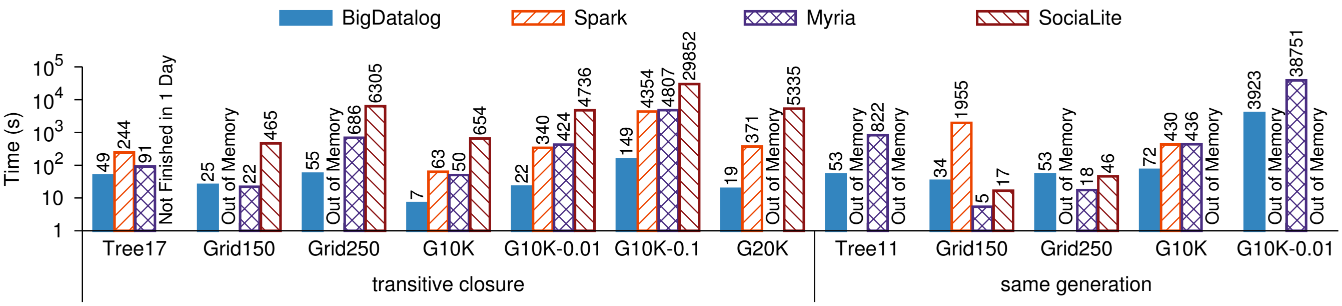


Figure 10: System Comparison using TC and SG.

head of shuffling. The same amount of data is also transmitted via shuffling or message passing for both Myria and SociaLite, but their performance is less stable compared with Spark. More specifically, Myria runs out of memory on G20K and SociaLite is always more than 10X slower. We believe this is, in part, because the implementation of their communication subsystem is less robust compared to Spark's.

SG. BigDatalog outperforms the other systems on the three graphs with the largest result sets for SG. Although evaluation of Grid150 and Grid250 produces the two smallest intermediate results and two smallest final results of our test graphs, because there is little work to be performed each iteration, Myria and SociaLite outperform BigDatalog on both of these graphs due to the overhead of scheduling in BigDatalog and differences in the generated physical plan. For instance, recall the RRP for SG in Figure 7. The two joins can generate a massive amount of intermediate duplicate results, however de-duplication occurs after the shuffle (in PSN). Therefore, to prevent a large amount of disk writes for the shuffle, we place a distinct operator into the plan immediately before the shuffle, much like a map-side combiner. However, this has a negative impact on execution time for the smaller graphs (Grid150 and Grid250), but allows BigDatalog to support larger graphs. This optimization, along with Spark's robust shuffling implementation, helps to explain why BigDatalog is faster than Myria and SociaLite on Tree11, G10K and G10K-0.01.

For SG, BigDatalog outperforms the handwritten Spark programs on all graphs tested. Unlike with TC, the handwritten Spark program finishes the evaluation on Grid150 as the amount of data it caches in memory for SG is much less than it does for TC. However it is over 50X slower compared with BigDatalog since BigDatalog only requires a single shuffle per iteration, whereas the Spark program has to shuffle between nested-loop joins. The handwritten Spark program runs out of memory on three graphs due to dataset caching.

6.1.2 REACH, CC *and* SSSP *Experiments*

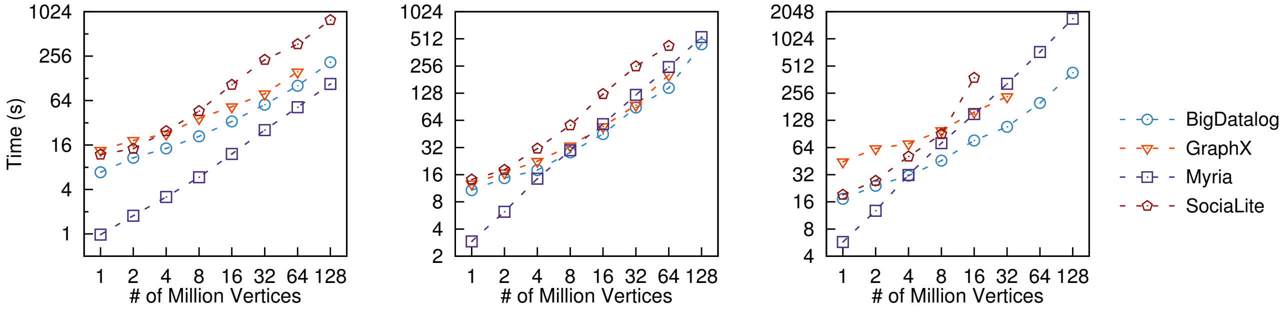
We perform experiments comparing the execution time of BigDatalog for REACH, CC and SSSP programs with Myria, SociaLite and GraphX programs on both the RMAT graphs and the real world graphs of Table 7. For these experiments, we use programs for GraphX [33], Spark's graph processing module that implements Pregel [40], instead of handwritten Spark programs. GraphX outperforms native Spark on these types of programs [33], which we validated in our experimental environment. Lastly, these results also help us understand how BigDatalog scales on different programs as the graph sizes increase. Further scaling experiments are reported in Appendix B.

Let n, m and d be the number of vertices, number of edges, and diameter of a graph; the number of intermediate results produced during evaluation is $O(m)$, $O(dm)$ and $O(nm)$ for REACH, CC and SSSP, respectively. Figures 11 and 12 show the experimental results for the RMAT graphs and the real world graphs, respectively. For each system, we report the total time of evaluation starting from loading the data from persistent storage, i.e., from PostgreSQL for Myria and from HDFS for the remaining three systems, until the evaluation completes. For CC, each point represents the average evaluation time on the test graph over five runs. For REACH and SSSP, each point represents the average time over ten randomly selected vertices, run five times each. A point is not reported in a figure if a system runs out of memory for the experiment for all vertices. In general we noticed that for all three programs on our test graphs, SociaLite spends most of the time on the loading and initialization of base relations, and its implementation expects a fast network connection to load large datasets efficiently, as suggested by an author of [48]. Lastly, although both systems evaluate on Spark, BigDatalog requires storing less auxiliary data in memory than GraphX. We will now detail the results of each program.

REACH. The REACH program finds all vertices connected by some path to a given source vertex using a simple linear recursion. REACH can be found in Appendix A. Figure 11(a) shows that Myria performs the best on all graph instances for REACH. Although Myria significantly outperforms on the smaller graphs, as the graph size increases, and thus the amount of communication required increases, BigDatalog is able to narrow the gap in performance. Similar behavior appears also in Figure 12. On all test graphs, BigDatalog outperforms GraphX for REACH.

CC. The connected component program is depicted in Program 8 of Appendix A. It uses a label propagation approach for determining the lowest vertex id a vertex is connected to, thus establishing membership in a component. This program is interesting because it uses a monotonic mmin aggregate in recursion. The Myria and SociaLite programs are expressed similarly and we use the connected components program packaged with the GraphX distribution.

For the RMAT graphs, as the amount of communication increases, BigDatalog outperforms Myria starting from RMAT-8M for CC. BigDatalog is roughly 20% faster than GraphX for the RMAT graphs. SociaLite exhibits poor relative performance due to slow dataset loading times. For the real world graphs, we observed that BigDatalog outperforms Myria and Socialite on all four graphs, and outperforms GraphX on three graphs, the exception being arabic.



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

Figure 11: Performance comparison on RMAT Graphs. The x-axis represents test graphs from RMAT-1M to RMAT-128M.

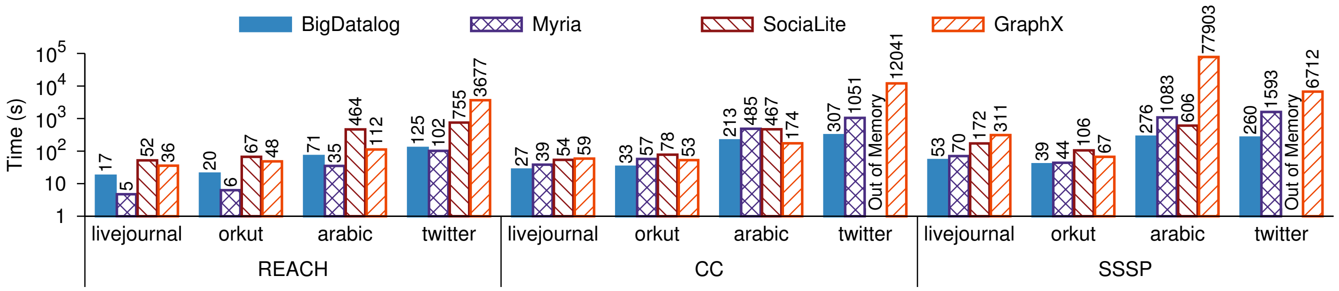


Figure 12: Performance comparison of REACH, CC and SSSP on real world graphs.

SSSP. This program was introduced in Program 5 and uses a mmin aggregate and a linear recursion. The Myria and SociaLite programs are expressed similarly. We have implemented SSSP in GraphX. As shown in Figure 11(c) and 12, BigDatalog outperforms all the systems on both synthetic and real world graph, except for Myria on the two smallest RMAT instances (1M, 2M). On all test graphs, BigDatalog outperforms GraphX.

6.2 Complex Data Analytics

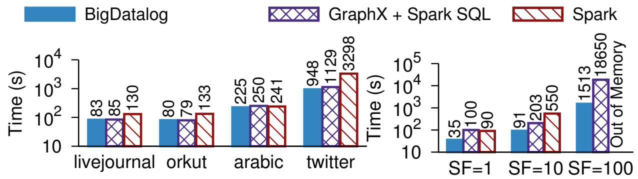
In this section, we report experimental results on two data analytics programs, where each represents a complex data-processing pipeline of mixed graph and relational workloads. For each program, we compare BigDatalog against (1) a regular Spark implementation; and (2) an implementation that uses a mixture of GraphX (for graph computations) and Spark SQL (for the relational part of the queries). The later is the implementation that an expert programmer will design to be able to exploit system-specific optimizations.

People You May Know (PYMK) is a feature of LinkedIn that helps members to grow their network by recommending other people to connect with [6]. One important component used in PYMK is based on the idea that a member is likely to know the people that share many common connections with him/her. This is also known as *triangle closing*. Program 9 in Appendix A contains our implementation of PYMK. Specifically, we count for a given member X, the number of common connections with each member Y that is not already connected to X. We then display to X the members with top-k ranked count values together with their basic information. The member information are stored in a 200GB table produced using the PigMix dataset generator [11]. The query evaluation is depicted in Figure 13(a), where we used each of the four real world graphs in Table 7 as the member connection graph. This query is not recursive, therefore BigDatalog generates a plan that is the same as the regular Spark SQL plan. As the experiments show, the performance of BigDatalog is competitive to that of the GraphX/Spark SQL implementation of the query.

Multi-Level Marketing Network Bonus Calculation (MLM). Many companies use a multi-level marketing model to sell a variety of products [1]. The MLM query, as shown in Program 10 of Appendix A, computes the net profit of a company embracing a Multi-Level Marketing model after paying bonuses to members. A bonus is distributed to each member based on his/her personal sales, and the sales of each member of the network he/she directly/indirectly sponsored. The database contains the following tables:

- sponsor(M, NM) stores the sponsorship information. A new member NM is sponsored by a member M that is already part of the marketing network;
- sales(M, S, P) stores the transaction records, where member M sold some products for S dollars, and the gross profit of the transaction is P;
- schedule(LS, RS, BP) stores the bonus schedule, which is used to determine the bonus for making a sales of S dollars, i.e., the bonus is BP × S if S $\in$ [LS, RS).

We generate tables under a setting akin to the TPC-H benchmark [13]: for a given scale factor (SF), sponsors contains a forest of ten random recursive trees [12] with 150K × SF vertices in total, and sales contains 1.5M × SF



(a) PYMK (b) MLM

Figure 13: Comparison of complex data analytics programs.

records. schedule always contains 12 records. Figure 13(b) shows the experimental results on scale factors 1, 10 and 100; BigDatalog is consistently at least 2 times faster than Spark and GraphX/Spark SQL.

7. RELATED WORKS

Datalog Implementations. The Myria [54] runtime supports DATALOG evaluation using a pipelined, parallel, distributed execution engine that evaluates a graph of operators. Datasets are sharded and stored in PostgreSQL instances at worker nodes. SociaLite [48] is a DATALOG language implementation for social network analysis. SociaLite programs are evaluated by parallel workers that use message passing to communicate. Both SociaLite and Myria support monotonic aggregation inside recursion using aggregate semantics based on the lattice-semantics of Ross and Sagiv [45]. This semantics was shown to be not general and difficult to use in practice [53]. Furthermore, although operational semantics of their monotonic aggregate programs is provided, no declarative semantics is given. Instead, BigDatalog bases its monotonic aggregate (operational and declarative) semantics on works [41, 42] that use monotonic *w.r.t.* set-containment semantics, and therefore maintain the least fixpoint semantics of DATALOG.

Parallel Datalog Evaluation and Languages. Previous research on parallel evaluation of DATALOG programs determined that some programs are evaluable in parallel without redundancy and without processor communication or synchronization [57]. Such programs are called decomposable. BigDatalog identifies decomposable programs via syntactic analysis of program rules using the generalized pivoting work [46]. To our knowledge BigDatalog is the only current DATALOG system providing such a feature.

Many works produced over twenty years ago focused on parallelization of bottom-up evaluation of DATALOG programs, however they were largely of a theoretical nature. For instance [52] proposed a message passing framework for parallel evaluation of logic programs. Techniques to partition program evaluation efficiently among processors [56], the tradeoff between redundant evaluation and communication [30, 31] and classifying how certain types of DATALOG programs can be evaluated [23] were also studied. A parallel semi-naïve fixpoint has been proposed for message passing [56] that includes a step for sending and receiving tuples from other processors during computation. The PSN used in this work applies the same program over different partitions and shuffle operators in place of processor communication.

Among the distributed DATALOG languages, it is noteworthy to mention *OverLog* [24, 38], used in the P2 system to express overlay networks, and *NDlog* [37] for declarative networking. The *Bloom*^{L} [25] distributed programming language uses various monotonic lattices, also based on the semantics of [45], to identify program elements not requiring coordination. [21] showed how XY-stratified DATALOG can support computational models for large-scale machine learning, although no full DATALOG language implementation on a large-scale system was provided. Recent works on recursive query evaluation showed efficient versions of transitive closure for multi-core [60] and distributed [14] settings, however these works did not address how to convert arbitrary programs to these desirable evaluation forms.

Systems for Large Scale Data Analysis. Spark [62] has recently gained much attentions as a general platforms for large-scale analytics. The Spark stack provides APIs for relational queries [16], graph analytics [33], stream processing and machine learning. DryadLINQ [61], REX [43], and SCOPE [66] provide high level languages for data analysis and support iteration. Extended MapReduce system designs providing API support include Haloop [22], PrIter [65], and Twister [28]. Incremental iterations were integrated into Stratosphere to support iterative algorithms with sparse computational dependencies [29]. ScalOps [55] supports a loop construct to include iteration in a recursive query plan executed on Hyracks [20], a distributed dataflow engine. Naiad [44] uses a time-based dataflow computational model to support iterative workflows and incremental updates. Distributed systems providing a vertex-centric API for graph analytics workloads include Pregel [40], Giraph [2] and GraphLab [39].

8. CONCLUSION AND FUTURE WORK

In this paper, we presented BigDatalog, a DATALOG language implementation on Apache Spark. Using our system Spark programmers can now benefit from using a declarative, recursive language to implement their distributed algorithms, while maintaining the efficiency of highly optimized programs. On our large test graph instances BigDatalog outperforms other state-of-the-art DATALOG systems on the majority of our tests. Moreover, our experimental results confirmed that among Spark-based systems BigDatalog outperforms both GraphX and native Spark for recursive queries.

Addressing our Challenges. We addressed the challenges for using Spark as a DATALOG runtime as outlined in Section 2.3 as follows: now with BigDatalog, recursive queries are compiled and optimized for efficient evaluation on Spark, which was verified by our experimental results (Challenge 1). BigDatalog is able to identify and produce physical plans for evaluating decomposable programs. In addition, we propose a new type of job for recursive programs to allow the scheduler greater control over iterations (Challenge 2). Lastly, we propose specialized RDDs (SetRDD/AggregateSetRDD) that utilize DATALOG semantics to support memory-efficient recursive evaluation (Challenge 3).

Future Work. In the course of this research we have identified several opportunities for exciting new directions. One first direction is to extend BigDatalog to support XY-DATALOG and realize the vision of [21] to use DATALOG to support complex machine learning analytics such as logistic regression over a massively parallel system. Another area is to investigate system extensions for provenance and fault tolerance enabled by monotonic DATALOG constructs.

For the latest updates on the BigDatalog project, see [7].

9. ACKNOWLEDGEMENTS

This work was supported by NSF grants IIS-1218471, IIS-1302698 and CNS-1351047, U54EB020404 awarded by the NIBIB through funds provided by the NIH BD2K initiative, and generous gifts from IBM Research, Symantec and Intel. We thank our reviewers for their thoughtful comments and insights. We very much appreciate the time and effort they put forth to make this a better paper. We thank Jingjing Wang and Jiwon Seo for their assistance with the experimental comparison.

10. REFERENCES

[1] Amway Business Reference Guide. https://www.amway.com/en/ResourceCenterDocuments/Visitor/ops-amw-gde-v-en--BusinessReferenceGuide.pdf.

[2] Apache Giraph. http://giraph.apache.org.

[3] Apache Hadoop. http://hadoop.apache.org.

[4] Apache Spark. http://spark.apache.org.

[5] arabic-2005 network. http://law.di.unimi.it/webdata/arabic-2005/.

[6] Big Data Ecosystem at LinkedIn. http://www.slideshare.net/mitultiwari/big-data-ecosystem-at-linkedin-keynote-talk-at-big-data-innovators-gathering-at-www-2015. Keynote talk at Big Data Innovators Gathering at WWW 2015.

[7] Deductive Application Language System (DeALS). http://wis.cs.ucla.edu/deals/.

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

[9] LiveJournal social network. http://snap.stanford.edu/data/com-LiveJournal.html.

[10] Orkut social network. http://snap.stanford.edu/data/com-Orkut.html.

[11] PigMix. https://cwiki.apache.org/confluence/display/PIG/PigMix.

[12] Recursive tree. https://en.wikipedia.org/wiki/Recursive_tree.

[13] TPC-H. http://www.tpc.org/tpch/.

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

[15] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *PODS*, pages 283–292, 2011.

[16] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[17] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo. The deductive database system ldl++. *TPLP*, 3(1):61–94, 2003.

[18] F. Bancilhon. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems*, pages 165–178. Springer-Verlag, 1986.

[19] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[20] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[21] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[22] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The haloop approach to large-scale iterative data analysis. *The VLDB Journal*, 21(2):169–190, 2012.

[23] S. Cohen and O. Wolfson. Why a single parallelization strategy is not enough in knowledge bases. In *PODS*, pages 200–216, 1989.

[24] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *PVLDB*, 1(1):1153–1165, Aug. 2008.

[25] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SoCC*, 2012.

[26] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[27] J. Eisner and N. W. Filardo. Dyna: Extending datalog for modern ai. In *Datalog Reloaded*, pages 181–220, 2010.

[28] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.

[29] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[30] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of datalog queries. In *SIGMOD*, pages 143–152, 1990.

[31] S. Ganguly, A. Silberschatz, and S. Tsur. Parallel bottom-up processing of datalog queries. *The Journal of Logic Programming*, 14(1):101–126, 1992.

[32] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2), 2009.

[33] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[34] M. Interlandi and L. Tanca. On the CALM principle for BSP computation. In *AMW*, 2015.

[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[36] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.

[37] 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 *SIGMOD*, pages 97–108. ACM, 2006.

[38] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, pages 75–90. ACM, 2005.

[39] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[40] 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 *SIGMOD*, pages 135–146, 2010.

[41] M. Mazuran, E. Serra, and C. Zaniolo. A declarative

extension of horn clauses, and its significance for datalog and its applications. *TPLP*, 13(4-5):609–623, 2013.

[42] M. Mazuran, E. Serra, and C. Zaniolo. Extending the power of datalog recursion. *The VLDB Journal*, 22(4):471–493, 2013.

[43] S. R. Mihaylov, Z. G. Ives, and S. Guha. Rex: Recursive, delta-based data-centric computation. *PVLDB*, 5(11):1280–1291, July 2012.

[44] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[45] K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *PODS*, pages 114–126, 1992.

[46] J. Seib and G. Lausen. Parallelizing datalog programs by generalized pivoting. In *PODS*, pages 241–251, 1991.

[47] J. Seo, S. Guo, and M. S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, 2013.

[48] 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.

[49] A. Shkapsky, M. Yang, and C. Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. In *ICDE*, pages 867–878, 2015.

[50] A. Shkapsky, K. Zeng, and C. Zaniolo. Graph queries in a next-generation datalog system. *PVLDB*, 6(12):1258–1261, 2013.

[51] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[52] A. Van Gelder. A message passing framework for logical query evaluation. In *SIGMOD*, pages 155–165, 1986.

[53] A. Van Gelder. Foundations of aggregation in deductive databases. In *DOOD*, pages 13–34, 1993.

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

[55] M. Weimer, T. Condie, and R. Ramakrishnan. Machine learning in scalops, a higher order cloud computing language. In *BigLearn*, December 2011.

[56] O. Wolfson and A. Ozeri. A new paradigm for parallel and distributed rule-processing. In *SIGMOD*, pages 133–142, 1990.

[57] O. Wolfson and A. Silberschatz. Distributed processing of logic programs. In *SIGMOD*, pages 329–336, 1988.

[58] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[59] M. Yang, A. Shkapsky, and C. Zaniolo. Parallel bottom-up evaluation of logic programs: DeALS on shared-memory multicore machines. In *Technical Communications of ICLP*, 2015.

[60] M. Yang and C. Zaniolo. Main memory evaluation of recursive queries on multicore machines. In *IEEE Big Data*, pages 251–260, 2014.

[61] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[62] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[63] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.

[64] V. Zaychik Moffitt, J. Stoyanovich, S. Abiteboul, and G. Miklau. Collaborative access control in webdamlog. In *SIGMOD*, pages 197–211, 2015.

[65] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritized iterative computations. In *SOCC*, pages 13:1–13:14, 2011.

[66] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. Scope: Parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, Oct. 2012.

APPENDIX

A. ADDITIONAL QUERIES

PROGRAM 6. *Who will attend the party?*

```
r1. cntComing(Y, mcount⟨X⟩) ← attend(X), friend(Y, X).
r2. attend(X) ← organizer(X).
r3. attend(X) ← cntComing(X, N), N ≥ 3.
```

Program 6 is the ATTEND program which identifies people who will attend a party if they are an organizer or at least three of their friends are attending. A version of this program was originally proposed in [45]. This program is explained as follows. The `friend(Y,X)` predicate instance represents that person `Y` is a friend of `X`. *r1* uses the `mcount` monotonic aggregate inside the recursion to count the number of friends `X` that each person `Y` knows who is going to the party. *r2* says that organizers attend the party. *r3* determines `X` is going to the party if they have at least three friends already attending.

There are two attributes of the ATTEND program worth noting. Firstly, this program has a mutual recursion between the `attend` and `cntComing` recursive predicates. The query to evaluate the program will be of the form `attend(X)` so BigDatalog will make the `attend` predicate the RO, and thus the driver of the recursion, and `cntComing` will be a non-driver recursion operator in `attend`'s RRP. Second, the comparison `N` $\geq$ `3` in *r3* provides an example of a monotonic arithmetic and monotonic boolean expression[5], the only type of expressions allowed on the result of a monotonic aggregate. If instead of $\geq$, equality was used, this comparison would only be true at three and then become false at higher counts and thus introduce non-monotonicity. □

PROGRAM 7. *Reachability*

```
r1. reach(Y) ← Y = $ID.
r2. reach(Y) ← reach(X), arc(X, Y).
```

[5] Once a monotonic boolean expression evaluates to true, it stays true for evaluations on subsequent values.

The reachability (REACH) program identifies all nodes reachable from the given source node $ID. It has a linear recursion and a unary (single argument) recursive predicate (reach). *r1* initializes the recursion from $ID. Then, in *r2* previously computed reach facts are joined to arc to find new vertices reachable from $ID. □

Program 8. *Connected Components*

```
r1. cc2(X, mmin⟨X⟩) ← arc(X, _).
r2. cc2(Y, mmin⟨Z⟩) ← cc2(X, Z), arc(X, Y).
r3. cc(X, min⟨Y⟩) ← cc2(X, Y).
```

The connected components (CC) program is used to identify the connected components of a graph. This program works by initially assigning the node's id to itself (*r1*), and then propagating a new lower node id for any edge the node is connected to. *r3* is necessary to select only the minimum node id Y for each X found in cc2. □

Program 9. *People You May Know*

```
r1. uarc(X, Y) ← arc(X, Y).
r2. uarc(Y, X) ← arc(X, Y).
r3. cnt(Y, Z, count⟨X⟩)) ←uarc(X, Y), uarc(X, Z), Y!= Z, ∼uarc(Y, Z).
r4. pymk(X, W9, topk⟨10, Z⟩) ← cnt(X, $ID, Z), pages(X, W2, . . . , W9).
```

The people you may know (PYMK) program is for helping members to grow their network by recommending other people to connect with. Let arc(X, Y) be the member connection graph. We assume the input graph to be undirected, but arc only keeps pairs (X, Y) that satisfy X < Y in order to save space. *r1* and *r2* construct the full undirected graph from arc, and *r3* counts the number of shared connections between each pair (Y, Z) such that Y and Z are not directly connected by an edge in the graph. In BigDatalog syntax, Y!= Z means Y is not equal to Z, and ∼uarc(Y, Z) means that tuple (Y, Z) is not in uarc. For a given member $ID, *r4* first finds all the tuples (X, $ID, Z) in cnt, i.e., X is a candidate member to recommend to $ID, and Z is the number of shared connections between X and $ID; then finds the user information for each candidate member X, which is stored in the last column of pages. Finally, *r4* returns the top 10 candidate members (together with their infos) by number of shared connections. In BigDatalog, topk⟨X, Y⟩ is a special "aggregate" function returning the top X tuples ordered by the Y-term. □

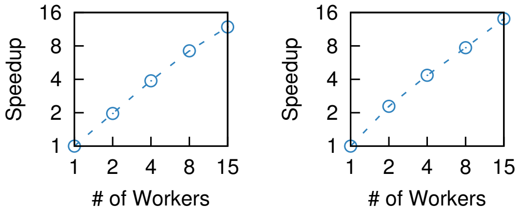
Program 10. *Multi-Level Marketing Network Bonus Calculation*

```
r1. networkTC(M, M) ← sponsor(M, _).
r2. networkTC(M, M) ← sponsor(_, M).
r3. networkTC(M, M2) ← networkTC(M, M1), sponsor(M1, M2).
r4. memberTotalSales(M, sum⟨S⟩) ← networkTC(M, NM),
                                   memberSales(NM, S).
r5. memberBonusSelf(M, B) ← memberSales(M, ST),
                  memberTotalSales(M, S), schedule(LS, RS, BP),
                  S >= LS, S < RS, B = ST ∗ BP.
r6. memberBonusFrontline(M, sum⟨B⟩) ← sponsor(M, NM),
                  memberTotalSales(NM, S), schedule(LS, RS, BP),
                  S >= LS, S < RS, B = S ∗ BP.
r7. bonus(sum⟨B⟩) ← memberBonusSelf(M, B1),
                   memberBonusFrontline(M, B2), B = B1 + B2.
r8. grossProfit(sum⟨P⟩) ← sales(_, _, P).
r9. netProfit(NP) ← grossProfit(P), bonus(B), NP = P − B.
```

The multi-level marketing network bonus calculation (MLM) program computes the net profit of a company embracing a multi-level marketing model, after paying bonuses to members. sponsor is a directed acyclic graph that represents the sponsorship relation. *r1*, *r2*, and *r3* computes the transitive closure of sponsor, such that networkTC contains all the tuples (M, NM) where M directly/indirectly sponsors NM. For each member M, *r4* computes the total sales for the network where the members in the network are directly/indirectly sponsored by M. The bonus of each member M consists of the following two parts: (1) the bonus from his/her own personal sales (*r5*); and (2) the bonus derived from the sales of each NM directly sponsored by M (*r6*). Bonuses are computed by multiplying the related sales figures with a factor BP determined by the schedule. Finally, *r7* computes the total bonus paid by the company, *r8* computes the gross profit of the company, and *r9* computes the net profit for the company. □

B. MORE SCALING EXPERIMENTS

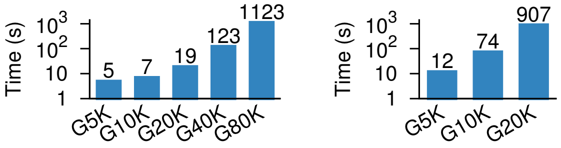
Here we report additional experimental results of how BigDatalog scales over different cluster and dataset sizes.



(a) TC on G20K (b) SG on G10K

Figure 14: Scaling-out Cluster Size.

Scaling-out. In this set of experiments we use the largest G*n*-*p* graphs that could be evaluated on all cluster sizes. Figure 14(a) shows the speedup for TC on G20K as the number of workers increases from one to 15 (all with one master) *w.r.t.* using only one worker, and Figure 14(b) shows the same experiment run for SG with G10K. Both figures show a linear speedup, with the speedup of using 15 workers is 12X and 14X for TC and SG, respectively.



(a) TC (b) SG

Figure 15: Scaling-up on Random Graphs.

Scaling-up. We use the full cluster to see how BigDatalog scales over graphs of increasing sizes for TC and SG. We use G*n*-*p* graphs from Table 6. For TC in Figure15(a) the smaller G5K and G10K graphs take roughly the same time to evaluate. From G20K, we observe the increase in runtime matches the number of intermediate facts produced, which should be viewed as the work the system must perform, rather than the increases in the size of the transitive closure. For example from G40K to G80K, the size of the transitive closure increases 4X, while the size of the intermediate results increases over 10X. We observe a similar result with SG. For example from G10K to G20K we observe a 4X increase in the size of the SG result set, but a nearly 16X increase in intermediate facts produced.