Distributed Graph Analytics with Datalog Queries in Flink

Muhammad Imran, Gábor E. Gévay, and Volker Markl

Technische Universität Berlin, Germany

Abstract. Large-scale, parallel graph processing has been in demand over the past decade. Succinct program structure and efficient execution are among the essential requirements of graph processing frameworks. In this paper, we present *Cog*, which executes Datalog programs on the Apache Flink distributed dataflow system. We chose Datalog for its compact program structure and Flink for its efficiency. We implemented a parallel *semi-naive evaluation* algorithm exploiting Flink's *delta iteration* to propagate only the tuples that need to be further processed to the subsequent iterations. Flink's delta iteration feature reduces the overhead present in acyclic dataflow systems, such as Spark, when evaluating recursive queries, hence making it more efficient. We demonstrated in our experiments that Cog outperformed BigDatalog, the state-of-the-art distributed Datalog evaluation system, in most of the tests.

Keywords: Datalog · Recursive Queries · Graph Processing · Cyclic Dataflows.

1 Introduction

Graphs can represent numerous real-world problems. With the advancement of the web and the vast number of its users, *efficiently* processing massive graphs is becoming essential. Efficiency can be achieved by scaling out computations to a cluster and thereby reducing computation times. Existing state-of-the-art systems that choose Datalog as their language, such as BigDatalog [15] and Myria [17], suffer either from significant scheduling overhead or shuffling overhead to perform each iteration of a graph computation.

From the users' perspective, having concise programming constructs that are easy to learn is also an essential factor to consider. Existing large-scale graph processing systems, such as Gelly [21] of Flink [5], or GraphX [9] of Spark [18], do not provide *conciseness* and require significant effort to perform even simple analytics. These systems are complex and verbose due to their APIs being embedded in general-purpose languages, such as Java or Scala. In contrast, Datalog offers more conciseness [10], i.e., shorter programs, and therefore makes it easier to implement graph-analytics or artificial intelligence algorithms.

This paper presents *Cog*, which is a Flink-based evaluation system of positive Datalog programs that do not contain aggregates. The core feature of Cog is the efficient evaluation of Datalog's recursive queries by exploiting Flink's

native delta iterations [7]. Flink is particularly suitable to evaluate Datalog programs because of its ability to evaluate iterative algorithms efficiently by cyclic dataflows. From relational queries (i.e., join, union, recursive queries) to graph processing algorithms (e.g., transitive closure) can conveniently be implemented in Cog, and executed on a cluster in a scalable way.

Contributions. We made the following contributions:

- We created logical plans for Datalog programs to be executed on a distributed dataflow engine. The logical plan also includes an explicit representation of recursive queries.
- We implemented a Datalog query execution engine that exploits Flink's *delta iteration* feature, which we found to be particularly well-suited for the classic *semi-naive* Datalog evaluation algorithm.
- We experimentally confirmed that evaluating recursive queries of Datalog using Flink's delta iteration performs better than the Spark-based BigDatalog [15] system, which is the state of the art in scalable Datalog execution.

2 Preliminaries

We will now briefly review Datalog and Apache Flink. We will also show why Flink's delta iteration is suitable to evaluate recursive Datalog programs efficiently.

2.1 Datalog

Datalog [6] is a rule-based query language. Each rule is expressed as a *function-free* horn clause, such as h :- $b_1, ..., b_n$, where h is the *head* predicate of the rule, and each b_i is a *body* predicate separated by a comma "," which represents the logical AND ($\wedge$). A predicate is also known as a *relation*. A *fact* is a tuple in a relation. A Datalog rule is *recursive* if the head predicate of a rule also appears in the body of the rule. After evaluating all body predicates, the produced facts are assigned to the head predicate of the rule. A relation that comes into existence as a result of a rule execution is called an *intensional database (IDB)*. A stored relation is called an *extensional database (EDB)*. The transitive closure (TC) program in Datalog is given in Listing 1 as an example. In the example, the predicate `arc` is an EDB, whereas the predicate `tc` is an IDB. The rule r_2 is a recursive rule as it has the predicate `tc` in its head and body. A *join* is created between `tc` and `arc` predicates in rule r_2, and the resulting facts are assigned to the head predicate `tc`.

```
r1: tc(X, Y) :- arc(X,Y).
r2: tc(X, Y) :- tc(X,Z),arc(Z,Y).
```

Listing 1. Transitive Closure (TC) program in Datalog.

2.2 Apache Flink

Apache Flink [5] is a distributed dataflow system. While nowadays Flink is mostly known for efficient stream processing, it initially focused on iterative dataflows in batch computations [7]. In this paper, we rely only on its batch-processing capabilities for translating Datalog programs to iterative dataflows.

Flink's batch API is centered around the `DataSet` class, which represents a scalable collection of tuples. `DataSet`s offer numerous data processing operators (such as map, filter, join), which create new `DataSet`s. From a Flink program written using `DataSet` operators, Flink creates a *dataflow job*, a directed graph where nodes represent data processing operators and edges represent data transfers. Flink executes these dataflow jobs in a scalable way, by parallelizing the execution of each dataflow node on the available worker machines in a cluster. Flink executes all operators *lazily*, i.e., the operator is first only added to the dataflow job as a node, and then later executed as part of the dataflow job execution. The dataflow job is executed when Flink encounters an *action* operator (such as counting the elements in a `DataSet`, or printing its elements), or when the user explicitly triggers the execution of the dataflow job that was built up so far. Flink provides libraries and APIs to perform relational querying, graph processing [21], and machine learning.

Iteration APIs. Flink supports two types of iterations: *bulk* and *delta*. Bulk iterations are general-purpose iterations, where the result of each iteration is a completely new *solution set* computed from the previous iteration's solution set [7]. On the other hand, delta iterations are a form of incremental iterations, which is suitable for iterative algorithms with sparse computational dependencies, i.e., where each iteration's result differs only partially from the previous iteration. In the context of Datalog evaluation, the semantics of delta iteration matches well wih the principles of the classic *semi-naive* evaluation algorithm [3], thus making it suitable for recursive Datalog program executions: applying a recursive Datalog rule once often adds only a small number of tuples compared to the total result size.

Iteration Execution in Cyclic Dataflow Jobs. Flink executes iterative programs written using the above iteration APIs in a single, *cyclic* dataflow job, i.e., where an iteration's result is fed back as the next iteration's input through a backwards dataflow edge. This is in contrast to many other dataflow systems, such as Apache Spark [18], which execute iterative programs as a series of acyclic dataflow jobs. Flink's cyclic dataflows are more efficient for several reasons:

– Having a single dataflow job for all iterations avoids the inherent overhead of launching a dataflow job on a cluster of machines. The main overhead of launching a job is the (centralized) scheduling of the constituent tasks of the job to a large number of machines.

– Operator lifespans can be extended to all iterations. (Whereas in Spark, new operators are launched for each iteration.) This enables Flink to naturally perform two optimizations:
  - In the case of a delta iteration, Flink can keep the solution set in the state of an operator that spans all iterations. Thereby, the solution set does not need to be newly rebuilt for each iteration, and instead small changes can be efficiently accommodated by just modifying the existing operator state.
  - Loop-invariant datasets, i.e., datasets that are reused without changes in each iteration (e.g., `arc` in Listing 1), can be more efficiently handled. For example, when one input of an equi-join is a loop-invariant dataset, the join operator can build a hash table of the loop-invariant input only once, and just probe that same hash table at every iteration.

3 Cog

In this section, we discuss Cog, our system that executes Datalog programs on Flink. We implemented positive Datalog without aggregation. Cog takes a Datalog input program, parses it, converts the parsed program to an intermediate representation, creates and optimizes a logical plan, and finally creates a Flink plan for execution. Listing 2 shows an example for writing Datalog programs in Cog.

```
DatalogEnvironment datalogEnv = DatalogEnvironment.create(flinkEnv);
String transitiveClos =
       "tc(X,Y) :- graph(X,Y).\n" +
       "tc(X,Y) :- tc(X,Z),graph(Z,Y).\n";
String query = "tc(X,Y)?";
// Read input data using standard Flink operators:
DataSet<Tuple2<Integer, Integer>> inputGraph = ...;
// Register it for use in Datalog queries:
datalogEnv.registerDataSet("graph", inputGraph);
// Execute the query:
DataSet<Tuple2<Integer, Integer>> result =
  datalogEnv.executeQuery(transitiveClos, query);
// The result is a standard Flink DataSet that we can further process:
System.out.println(result.count());
```

Listing 2. An implementation of Transitive Closure (TC) program in Cog.

3.1 Query Representation and Planning

Query Representation. A parsed Datalog program is represented in the form of a *predicate connection graph (PCG)*. PCG for a deductive database system was introduced in [2]. Figure 1 shows the PCG for the `TC` query as an example. A PCG is an annotated AND/OR tree, i.e., it has alternating levels of AND

and OR nodes. The AND nodes represent head predicates of rules, and the OR nodes represent body predicates of rules. The root and the leaves are always OR nodes. The root of the tree represents the query predicate.

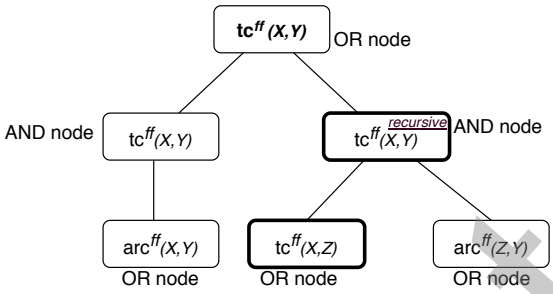

Fig. 1. Predicate Connection Graph (PCG) for Transitive Closure (TC) query.

Logical Plan. We used the algebra module of Apache Calcite [4] to create and optimize logical plans. Calcite provides numerous operators (such as join, project, union) to represent a query algebra. To evaluate recursive Datalog queries, the *repeat union* operator is an important one. The repeat union operator has two child nodes: *seed* and *iterative*. The seed node represents facts generated by non-recursive rule(s), whereas the iterative node represents facts generated by the recursive rule(s). The semantics of the repeat union operator are as follows: it first evaluates the seed node, whose result will be the input to the first iteration; then, it repeatedly evaluates the iterative node, using the previous iteration's result as input. The evaluation terminates when the result does not change between two iterations. Figure 2 shows the logical plan created for the `TC` program given in Listing 1. The Calcite-based logical plans are then transformed into Flink's own logical plans and then to Flink's `DataSet`-based plan. During these transformations, standard relational optimizations are also performed.

Flink Plan. The optimized logical plans are translated into Flink's `DataSet`-based plans. We utilized existing Flink `DataSet` operators for scans, joins, unions, filters, and projections. However, we implemented a translation from the *repeat union* and *transient table scan* operators to Flink `DataSet` operators to enable the execution of recursive queries, which we discuss in the next subsection.

3.2 Semi-Naive Evaluation in the Flink DataSet API

Semi-naive evaluation [3] is an efficient way to evaluate Datalog programs. With this technique, each iteration processes only the tuples that were produced by

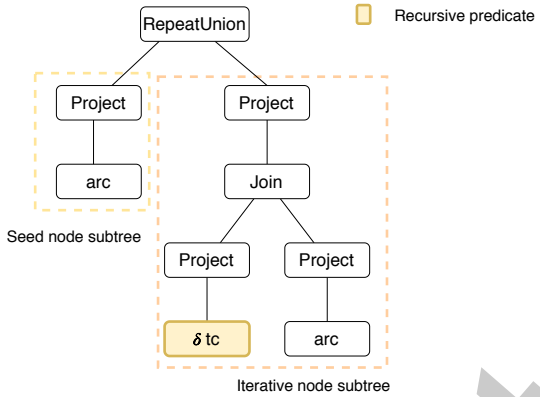

Fig. 2. Cog logical plan for Transitive Closure (TC) query.

the previous iteration, and thus redundant work is eliminated. The final result is obtained by the union of the results produced by each iteration. Algorithm 1 shows the pseudocode of semi-naive evaluation. In the algorithm, *seed* represents the non-recursive rule(s) (e.g., r_1 in `TC`), whereas *recursive* represents one execution of the recursive rule(s) (e.g., r_2 in `TC`). `W` represents the *differential* that is calculated in each iteration, and `S` stores the final result at the end.

Algorithm 1

1: **function** SEMI-NAIVE(*seed*, *recursive*)
2: $S \leftarrow seed$
3: $W \leftarrow seed$
4: **while** $W \neq \emptyset$ **do**
5: $D \leftarrow recursive(W) - S$
6: $W \leftarrow D$
7: $S \leftarrow S \cup D$

Algorithm 2

1: **function** FLINK-DELTA(S, W, u, δ, key)
2:
3:
4: **while** $W \neq \emptyset$ **do**
5: $D \leftarrow u(S, W)$
6: $W \leftarrow \delta(D, S, W)$
7: $S = S \dot{\cup} D$

Compare Algorithm 1 with Algorithm 2, which shows the general template of a Flink Delta Iteration. There is an initial solution set (S), and an initial workset (W), and then each iteration first computes a differential (D), which is to be merged into the solution set (Line 7), and also computes the workset for the next iteration. Note that the merging into the solution set is denoted by $\dot{\cup}$, which means that elements that not yet appear in the solution set should be added, and elements which have the same key as an element already in the solution set should override the old element: $S \dot{\cup} D = D \cup \{s \in S : \neg\exists d \in D | key(d) = key(s)\}$. We can see that with the following mapping, a Flink Delta Iteration performs exactly the semi-naive evaluation of a Datalog query:
$S = seed; W = seed; u(S, W) = recursive(W) - S; \delta(D, S, W) = D; key(x) = x$.
Note that by choosing the key to be the entire tuple, we make the $\dot{\cup}$ behave as a standard union.

When translating from Cog logical plans, the semi-naive evaluation is implemented to translate the *repeat union* operator to `DataSet` operators. Listing 3

presents this translation. We use a CoGroup operation to compute which of the tuples created in this iteration are not already in the solution set. We also use this CoGroup operation to eliminate duplicates. The work set propagates the differential to the next iteration. The solution set accumulates the output of all iterations. The work set and the solution set are always kept in memory for efficiency. Note that all the created Flink operators are evaluated *lazily* upon the call of a sink operator. Figure 3 shows the Flink plan for the `TC` query as an example. The sync task is a special operator inserted by Flink, which waits for all operators in the iteration body to perform one iteration, and then signals to the Flink runtime that the next iteration can start.

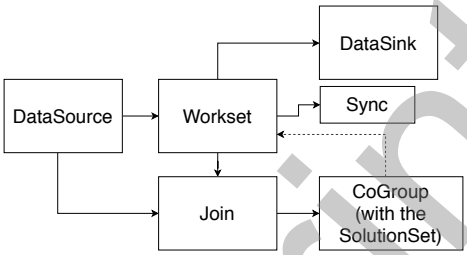

Fig. 3. The Flink plan for Transitive Closure (TC) query. Some operators are omitted/combined for clarity. Note that across all the iterations the Join operator keeps the hash table that it built for the `arc` dataset.


```
// Evaluate seed node (non-recursive rules).
val seedDs = seed.translateToDatasetPlan(tableEnv, queryConfig)
val workSet: DataSet[Row] = seedDs
val solutionSet: DataSet[Row] = seedDs
// Define delta iteration.
val iteration = solutionSet.iterateDelta(
              workSet, // initial workset
              Int.MaxValue, // max number of iterations
              seedDs.allFields) // the key is composed of all the fields
// Register the work set as a temporary table to the Flink catalog
// so that it can be used by the iterative node.
updateCatalog(tableEnv, iteration.getWorkset, "workset-temp-table")
// Translate the subtree of the iterative node (recursive rules).
// The subtree contains a Transient Table Scan operator, which is the
// representation of the recursive reference (shown in yellow in Fig. 2.).
// We added a rule (not shown here) to translate this as a reference to the
// "workset-temp-table", i.e., the DataSet representing the workset.
val iterativeDs =
     iterativeSubplan.translateToDatasetPlan(tableEnv, queryConfig)
// Compute the difference between the newly produced tuples
// and the solution set by a CoGroup operation. Flink will probe
```

```
// the hash table that it stores for the solution set throughout
// the job execution.
val delta = iterativeDs
    .coGroup(iteration.getSolutionSet)
    .where("*").equalTo("*") // all fields are included in the key
    .with(new DeduplicatingMinusCoGroupFunction[Row]())
// At the end of each iteration, the delta is used both as
// the set of tuples to be added to the solution set (1st argument),
// and the next workset (2nd argument).
val result = iteration.closeWith(delta, delta)
```

Listing 3. An implementation of the classic semi-naive Datalog evaluation algorithm in Flink. We mapped the algorithm to just a few standard Flink API calls.

4 Experiments

4.1 Experimental Setup

Hardware and Software Environment. We performed our experiments on a cluster of 8 nodes. Each worker node runs Fedora 28 as operating system, has an IBM PowerPC 48-core CPU, and 62GB memory. We allocated 48GB memory to the Flink and Spark worker processes. The nodes of the cluster are connected with Gigabit Ethernet. We implemented Cog on the current snapshot version of Flink (on the top of commit 8f8e358).

Benchmark Programs. Thus far, Cog supports positive Datalog programs, including recursive queries, without aggregation. We chose the following queries for benchmarking:

- **Transitive Closure (TC):** Finds all pairs of vertices in a graph that are connected by some path. Listing 1 shows `TC` in Datalog.
- **Same Generation (SG):** Two nodes are in the Same Generation (SG) if and only if they are at the same distance from another node in the graph. Listing 4 shows `SG` program in Datalog. The program finds all pairs that are in the 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).
```

Listing 4. Same Generation (SG) program in Datalog.

- **Single-Source Reachability:** Finds all vertices connected by some path to a given source vertex. Listing 5 shows the `Reachability` program in Datalog.

```
r1: reach(X,Y):- arc(X,Y), X=source.
r2: reach(X,Y):- reach(X,Z), arc(Z,Y).
```

Listing 5. `Reachability` program in Datalog.

Datasets. We used synthetic graph datasets to evaluate and benchmark our system. These datasets are `Tree11`, `Grid150`, and `g10K`. The same datasets are also used by Shkapsky et al. [15] for benchmark comparison of BigDatalog with Myria [17] and Distributed SociaLite [14] systems. Table 1 shows the properties of the datasets. These graphs have specific structural properties: `Tree11` has 11 levels, `Grid150` is a grid of 151 by 151, and the `G10K` graphs are 10k-vertex random graphs in which each randomly-chosen pair of vertices is connected with probability 0.001. The last three columns of Table 1 show the output size produced with these datasets by the benchmark queries. For the `Reachability` program, we used graph datasets generated with R-MAT [22] synthetic graph generator with probabilities $a = 0.45, b = 0.25, c = 0.15, d = 0.15$. For all the datasets, we calculated `Reachability` from vertex 977.

Table 1. Input- and output sizes, and the number of iterations (in parenthesis)

Name	**Vertices**	**Edges**	**TC**	**SG**	**Reachability**
Tree11	71,391	71,390	805,001 (11)	2,086,271,974 (11)	-
Grid150	22,801	45,300	131,675,775 (299)	2,295,050 (149)	-
G10K	10,000	100,185	100,000,000 (6)	100,000,000 (3)	-
R-MAT-1M	1 mill.	10 mill.	-	-	523,967 (4)
R-MAT-2M	2 mill.	20 mill.	-	-	1,047,937 (4)
R-MAT-4M	4 mill.	40 mill.	-	-	2,095,865 (4)
R-MAT-8M	8 mill.	80 mill.	-	-	4,191,735 (4)
R-MAT-16M	16 mill.	160 mill.	-	-	8,383,418 (5)
R-MAT-32M	32 mill.	320 mill.	-	-	16,767,026 (5)

4.2 Results

Now we will discuss the results of our experiments of `TC`, `SG`, and `Reachability` programs. We ran these programs in our system, and another state-of-the-art distributed Datalog system, namely BigDatalog [15]. As BigDatalog demonstrated its efficiency compared to other distributed Datalog system (such as Myria [17] and Distributed SociaLite [14]), our purpose of benchmark comparison is to show how Cog performs w.r.t. BigDatalog. Figure 4 and Figure 5 show the benchmark comparison of Cog and BigDatalog. We report the *median* values in Figure 4 and Figure 5.

TC. We used the query shown in Listing 1 for calculating `TC`. Cog outperformed BigDatalog for all the graphs. Notably, Cog showed 3x better performance than BigDatalog for `Tree11` and `Grid150` graphs. BigDatalog suffers from the overhead of scheduling caused by the large number of iterations, whereas no such overhead is present in Cog as it performs iterative programs in a cyclic dataflow job [8]. However, this overhead is negligible when there is only a small number of iterations (see Table 1). Cog can suffer performance loss due to data spilling during the CoGroup operation with the solution set. The performance

loss of Cog is visible in the case of `G10K`. With default settings, BigDatalog always crashed due to running out of memory as it was caching resilient distributed datasets (RDDs) in memory and clearing lineage in order to avoid stack overflow from long lineages. For `TC` queries, we disabled such caching RDDs to avoid crashes.

SG. We used Listing 4 for calculating `SG`. We found that Cog is 2x faster than BigDatalog for `Grid150` and `G10K` graphs, despite RDD caching to memory was enabled for BigDatalog. Though `SG` program produces a small number of output rows when `Grid150` is used as input, however, it is clear from the result of `G10K` that the scheduling overhead is not the only factor for slow execution speed. Cog suffered performance loss when executing `SG` on `Tree11` dataset. The reason for the inefficiency was the fact that the CoGroup operation with the solution set gets slower when the number of records stored in the solution set increases.

Single-Source Reachability. We used Listing 5 for calculating `Reachability` from a single vertex. Figure 5 shows the benchmark results of Cog and BigDatalog for the `Reachability` program. Cog outpaced BigDatalog in all the graph instances we used to evaluate the `Reachability`. The difference in performance between Cog and BigDatalog gets more prominent with the increase in the size of the datasets. When running `Reachability` on BigDatalog with default configuration (e.g., broadcast join), we saw an increase of approximately 1.5x on each 2x increase in the size of the graphs. Though the overhead of scheduling did not increase (i.e., the number of iterations for 1M, 2M, 4M datasets was 4). With default settings, BigDatalog crashed for all the datasets of sizes greater than 4M. We discovered that BigDatalog uses a broadcast join by default, which broadcasts the entire graph to all the worker nodes. We believe that this was the reason for the crash, and we changed the configuration to use a repartition join instead. This performed slightly faster and was able to process all the instances of our datasets. The running time growth for Cog on all the datasets was small and steady. Cog was 3.4x faster for the largest dataset we tested.

5 Related Work

Distributed Dataflow Systems. Flink [5] is a modern dataflow system (initially named Stratosphere [1]) for general-purpose data processing, that employs the incremental iteration model (specifically, delta iterations) [7]. Spark [18] is a scalable, fault-tolerant distributed in-memory dataflow engine suitable for batch processing. Spark, in contrast to Flink, presents a considerable scheduling overhead when used for iterative jobs. Each iteration is scheduled as a new job and performs transformations on the cached RDDs. Naiad [12] is a system based on the timely dataflow computational model that supports structured loops for streaming. The iteration mechanism in Naiad is similar to that in Flink. Therefore, it would be possible to implement semi-naive Datalog execution also on Naiad, similarly to how we implemented it for Cog. The Differential Datalog [13] system goes in this direction, but it supports only single-machine execution.

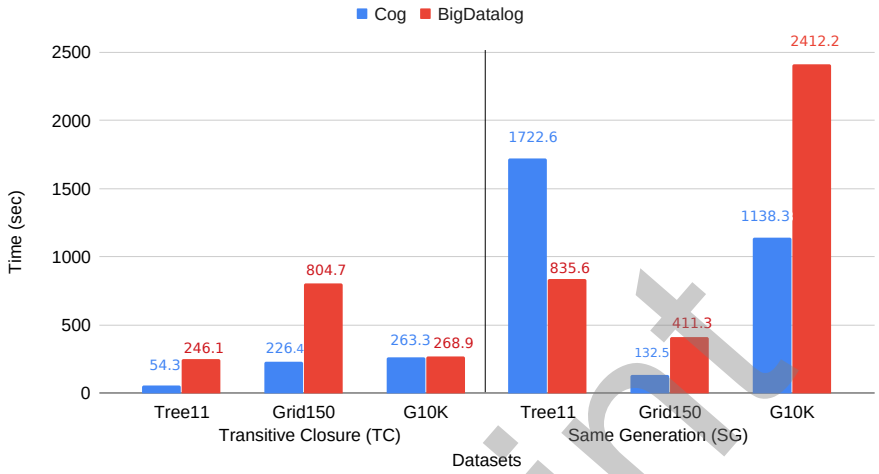

Fig. 4. Evaluation result comparison using `TC` and `SG` queries.

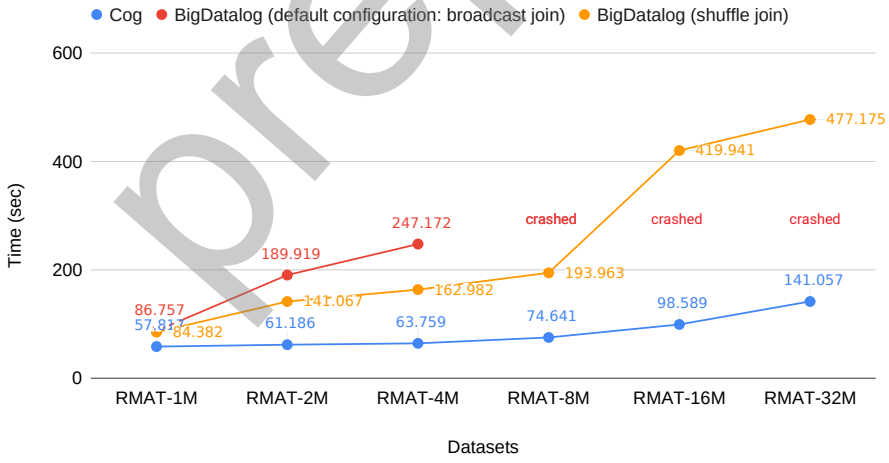

Fig. 5. Evaluation result comparison using `Reachability` query.

Pregel-like Graph Processing Systems. The think-like-a-vertex paradigm for graph processing (based on Bulk Synchronous Parallel (BSP) model [16]) was introduced by Pregel [11], and is used in many large-scale graph processing systems, such as Giraph [19] and GraphX [9]. Contrary to Datalog, the think-like-a-vertex paradigm provides a stateful model for computation and communication, whereas Datalog queries are more declarative.

Datalog Evaluation in Distributed Systems. Several systems implemented Datalog to be executed on a cluster of machines. BigDatalog [15] implemented positive Datalog with recursion, non-monotonic aggregations, and aggregation in recursion with monotonic aggregates on Spark. BigDatalog uses a number of clever tricks to overcome some of the limitations of Spark in the area of iterative computations. It optimizes the jobs by clearing RDD lineage without checkpointing the RDD to disk (and just checkpointing to memory instead). It added scheduler-aware recursion by adding a specialized Spark stage (*FixPointStage*) for recursive queries to avoid the job launching overhead. Furthermore, reusing Spark tasks within a *FixPointStage* eliminates the cost of task scheduling and task creation; however, task reuse can only happen on so-called *decomposable* Datalog programs, and only if the joins can be implemented by broadcasting instead of repartitioning, which is not the case for large graphs. BigDatalog added specialized SetRDD and AggregateRDD to enable efficient evaluation of recursive queries. BigDatalog also pays special attention to joins with loop-invariant inputs. It avoids repartitioning the static input of the join, as well as rebuilding the join's hash table at every iteration step. However, it does not ensure co-location of the join tasks with the corresponding cached build-side blocks, and thus cannot always avoid a network transfer of the build-side.

When implementing Cog, we did not need to perform any of the above optimizations, as Flink has built-in support for efficient iterations with cyclic dataflow jobs. Having cyclic dataflow jobs means that most of the issues that the above optimizations are solving either do not even come up (per-iteration job-launching overhead and task-scheduling overhead), or already have simple solutions by keeping operator states across iterations (loop-invariant join inputs, incremental updates to the solution set). Thus, our view is that relying on Flink's native iterations being implemented as a single, cyclic dataflow job is a more natural way to evaluate Datalog efficiently.

Distributed SociaLite [14], is a system developed for social network analysis that implemented Datalog with recursive monotone aggregate functions using a delta stepping method and gives the ability to programmers to specify data distribution. It uses message passing mechanism for communication among workers. It shows weaknesses in loading datasets (base relations) and poor shuffling performance on large datasets [15]. Myria [17] is a distributed execution engine that implemented Datalog with recursive monotonic aggregation function in a share-nothing engine and supports synchronous and asynchronous iterative models. Myria, however, suffers from shuffling overhead when running large datasets and becomes unstable (it often runs out of memory) [15].

6 Conclusion and Future Work

In this paper, we presented Cog, which is a Datalog language implementation for batch processing tasks on Apache Flink. The main advantages of Cog over other systems from a user perspective are its efficiency and conciseness. Cog executes recursive queries of Datalog as a single, cyclic dataflow job, thus avoiding scheduling overhead that is present in acyclic dataflows. In our experiments, we showed that Cog outperformed BigDatalog, a state-of-the-art large-scale Datalog system, in most of the test cases. The source code and the latest updates of Cog are available at [20].

Future Work. For a potential future work, an implementation of negation, non-monotonic aggregations, and aggregation in recursion for Datalog can be added to the system. Datalog for Flink stream processing tasks can also be implemented to facilitate analytics on real-time datasets. We believe that Datalog's implementation for Flink stream processing API could surpass Cog's efficiency because a Flink streaming job would not need a synchronization barrier after each iteration. Another future direction is to add support to Flink for recursive SQL queries, which are similar to recursive Datalog queries. Cog already performs the groundwork for this by translating the recursive logical plans to the Flink `DataSet` API.

7 Acknowledgment

This work was funded by the German Ministry for Education and Research as BIFOLD (01IS18025A and 01IS18037A).

References

1. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The stratosphere platform for big data analytics. The VLDB Journal **23**(6), 939–964 (Dec 2014). https://doi.org/10.1007/s00778-014-0357-y
2. Arni, F., Ong, K., Tsur, S., Wang, H., Zaniolo, C.: The deductive database system LDL++. Theory Pract. Log. Program. **3**(1), 61–94 (Jan 2003). https://doi.org/10.1017/S1471068402001515
3. Bancilhon, F.: Naive evaluation of recursively defined relations. In: On Knowledge Base Management Systems, pp. 165–178. Springer (1986)
4. Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M.J., Lemire, D.: Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources. In: Proceedings of the 2018 International Conference on Management of Data. pp. 221–230 (2018)
5. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36**(4) (2015)

6. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE transactions on knowledge and data engineering **1**(1), 146–166 (1989)
7. Ewen, S., Tzoumas, K., Kaufmann, M., Markl, V.: Spinning fast iterative data flows. Proc. VLDB Endow. **5**(11), 1268–1279 (Jul 2012). https://doi.org/10.14778/2350229.2350245
8. Gévay, G.E., Rabl, T., Breß, S., Madai-Tahy, L., Markl, V.: Labyrinth: Compiling imperative control flow to parallel dataflows. arXiv preprint arXiv:1809.06845 (2018)
9. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: Graph processing in a distributed dataflow framework. In: 11th USENIX Symposium on Operating Systems Design and Implementation OSDI 14). pp. 599–613 (2014)
10. Hajiyev, E., Verbaere, M., de Moor, O.: Codequest: Scalable source code queries with datalog. In: Proceedings of the 20th European Conference on Object-Oriented Programming. p. 2–27. ECOOP'06, Springer-Verlag, Berlin, Heidelberg (2006)
11. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146 (2010)
12. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 439–455 (2013)
13. Ryzhyk, L., Budiu, M.: Differential Datalog. In: Datalog 2.0 – 3rd International Workshop on the Resurgence of Datalog in Academia and Industry. CEUR-WS (2019)
14. Seo, J., Park, J., Shin, J., Lam, M.S.: Distributed SociaLite: A Datalog-based language for large-scale graph analysis. Proceedings of the VLDB Endowment **6**(14), 1906–1917 (2013)
15. Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., Zaniolo, C.: Big data analytics with Datalog queries on Spark. In: SIGMOD. pp. 1135–1149 (2016)
16. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33**(8), 103–111 (1990)
17. Wang, J., Balazinska, M., Halperin, D.: Asynchronous and fault-tolerant recursive Datalog evaluation in shared-nothing engines. Proceedings of the VLDB Endowment **8**(12), 1542–1553 (2015)
18. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I., et al.: Spark: Cluster computing with working sets. HotCloud **10**(10-10), 95 (2010)
19. Apache Giraph. http://giraph.apache.org/, [Online; Accessed 12 Apr. 2020]
20. Cog. https://github.com/imran-4/cog, [Online; Accessed 12 Apr. 2020]
21. Gelly: Flink Graph API. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/gelly/, [Online; Accessed 12 Apr. 2020]
22. GTGraph. http://www.cse.psu.edu/~kxm85/software/GTgraph/, [Online; Accessed 12 Apr. 2020]