Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines

Jingjing Wang, Magdalena Balazinska, and Daniel Halperin
Dept. of Computer Science & Engineering, University of Washington
{jwang,magda,dhalperi}@cs.washington.edu

ABSTRACT

We present a new approach for data analytics with iterations. Users express their analysis in Datalog with bag-monotonic aggregate operators, which enables the expression of computations from a broad variety of application domains. Queries are translated into query plans that can execute in shared-nothing engines, are incremental, and support a variety of iterative models (synchronous, asynchronous, different processing priorities) and failure-handling techniques. The plans require only small extensions to an existing shared-nothing engine, making the approach easily implementable. We implement the approach in the Myria big-data management system and use our implementation to empirically study the performance characteristics of different combinations of iterative models, failure handling methods, and applications. Our evaluation uses workloads from a variety of application domains. We find that no single method outperforms others but rather that application properties must drive the selection of the iterative query execution model.

1. INTRODUCTION

Whether in industry or in the sciences, users today need to analyze large datasets. Astronomers, for example, work with simulations of the universe that produce hundreds of terabytes of data [34]. They similarly work with telescope images from sky surveys such as the SDSS [42] or the upcoming LSST [32] that require the analysis of tens of terabytes of data. Other examples include the need to process large-scale outputs from genome sequencers and other high-throughput sensors or devices, the analysis of click streams, or social networks data.

Users need to perform a variety of complex analyses on this data including traditional relational algebra operations but also machine learning, linear algebra, and various domain-specific cleaning and transformation steps. One distinguishing feature of modern analytics is that it often requires *iterative* computations [14]. Graph analytics is the prime example of the need for efficient iterative processing: shortest path, reachability, connected components, and hubs and authorities are all iterative algorithms. The need for iterative computations extends beyond graphs, though. With astronomy simulations, for example, one type of analysis traces the evolution of galaxies in the universe by iteratively following the simulated particles over time.

In response to this need, many existing data processing engines have been extended with support for iterative computations [14, 19, 50]. New systems that are built today for data analytics all include some method to execute iterative queries [49]; And several systems have been developed specifically for iterative computations [21, 35, 36, 38, 39].

Despite this interest, existing solutions for iterative query processing have significant limitations: most parallel data processing engines support only synchronous iterations [9, 14, 33, 35, 49, 50], in which all machines must complete one iteration before the next begins. Such global synchronization barriers between iterations are undesirable because they cause faster machines to wait for the stragglers, slowing down query evaluation. Some engines support parallel and asynchronous iterative processing but are specialized for graphs [31, 38, 39, 46]. Many general-purpose iterative query processing systems do not support declarative queries but instead require users to specify query plans directly [21, 36, 46]. Finally, some systems generate code [38, 39] and thus do not provide a complete and general-purpose data management system. Datalog engines exist but are either single-node [4] or use MapReduce as backend [22] which supports only synchronous iterations.

Based on our experience working with big data users in domain sciences through the University of Washington eScience Institute [43], we argue that none of the above approaches is sufficient. Modern engines should provide the following set of capabilities for iterative computations: First, query evaluation should be performed in parallel in shared-nothing clusters to ensure scalability. Second, iterative query evaluation should be incremental, so that each iteration only computes a change to the final solution rather than recomputing the entire result each time. Third, the system should provide a variety of iteration models (synchronous, asynchronous, prioritized) because, as we will show, different problems necessitate different evaluation strategies. Fourth, engines need to support a broad variety of application domains, not only graphs. Finally, users should be able to specify their computation using a declarative query language for ease of use. The above features should be provided in an efficient and fault-tolerant manner.

In this paper, we develop such a new query evaluation approach and implement it in the Myria big data management system and service [5, 25]. Our query evaluation method sup-

Proceedings of the VLDB Endowment, Vol. 8, No. 12

ports *incremental*, *asynchronous* or *synchronous* evaluation of iterative queries in a *shared-nothing* cluster. In our system, users specify their queries *declaratively in a subset of Datalog with aggregation*. Importantly, this approach requires only *small extensions* to a shared-nothing query processing system, making it well suited for implementation in other engines. In addition to developing this query evaluation method, we empirically evaluate how small changes in the query evaluation strategy can lead to large performance differences for certain applications. We characterize when each approach leads to the lowest query run time for different types of queries.

An important problem with iterative query processing is failure handling. Given the scale of today's data analysis queries and the increasing use of preemptable compute resources such as Amazon Spot instances [1], modern query engines must include failure-handling techniques. Most query engines that provide iterative processing capabilities select a failure-handling method in an ad-hoc fashion. The interplay between iterative processing method, iterative query, and failure handling method is poorly understood. We address this problem by empirically studying this interplay.

An important overall contribution of our work is to answer the question: *How should an existing shared-nothing query engine be extended to efficiently support powerful iterative computations that are also easy for users to express?* We address this question through the following specific contributions:

- We develop a query-plan based approach for efficient, incremental processing of iterative computations in a parallel, relational engine. Our approach supports iterative computations with aggregations and with multiple output results. It supports both synchronous and asynchronous query evaluation, and different processing priorities. Our approach is easily implementable as it only requires a small set of extensions to an existing shared-nothing engine (Sections 3.2 and 3.3).
- We enable users to specify declarative queries in a subset of Datalog with aggregation extended from prior work [39] and present an algorithm to automatically convert these programs into our new query plans (Section 3.1).
- We empirically evaluate different query evaluation methods on iterative applications from the astronomy, social networks, and bibliometrics domains and on real data from these domains. We characterize when and why each query evaluation method yields the lowest query run time (Section 6.2).
- We study the interplay of iterative query execution methods, iterative queries, and fault-tolerance techniques. We evaluate which failure-handling method yields the fastest recovery in the presence of failures while imposing a small overhead without failures (Section 4 and Section 6.3).

Exploring a broad set of applications and techniques, we find that no single iterative execution strategy outperforms all others; rather, application properties must drive method selection. The wrong choice of execution technique can increase CPU utilization and network bandwidth by a factor of 6×. Memory usage can increase by a factor of 2 ×.

2. BACKGROUND

We briefly review how to express iterative queries with aggregation in Datalog and present existing query evaluation methods for iterative queries in shared-nothing systems.

2.1 Iterative Queries in Datalog

```
Edges(time,gid1,gid2, :- Particles(pid,gid1,time),
 $Count(*))              Particles(pid,gid2,time+1) (1)

Galaxies(1,gid)       :- GalaxiesOfInterest(gid)    (2)

Galaxies(time+1,gid2) :- Galaxies(time,gid1),
                         Edges(time,gid1,gid2,c),
                         c >= threshold             (3)
```

Figure 1: Datalog program for the GalaxyEvolution application. The inputs are relation Particles(pid,gid,time), which contains the output of an astrophysical simulation, and a set of galaxies of interest GalaxiesOfInterest(gid) at time=1.

Iterative queries are naturally expressed in Datalog as this language supports recursion. As an example, Figure 1 shows an iterative Datalog program called `GalaxyEvolution`. The program computes the history of a set of galaxies in an astrophysical simulation. The history of a galaxy is the set of past galaxies that merged over time to form the galaxy of interest at present day. The input table, `Particles(pid,gid,time)`, holds the simulation output as a set of particles, where `pid` is a unique particle identifier, and `gid` is the identifier of the galaxy that the particle belongs to at time, `time`. The `gid` values are unique only within their timesteps, `time`, but a particle retains the same `pid` throughout the simulation.

The program takes the form of a set of rules. Each rule has a head, an implication symbol `:-`, and a body. The first rule computes relation `Edges`, which contains the number of shared particles for every pair of galaxies at adjacent timesteps. The second and the third rules compute `Galaxies`, which is the set of earlier galaxies that have merged to form the galaxies of interest. The second rule states that a galaxy from `GalaxiesOfInterest` at present day (*i.e.*, `time=1`) is a part of the ancestry. The third rule states that a galaxy `(time+1, gid2)` is also part of the ancestry if an adjacent galaxy `(time, gid1)` is part of the ancestry, and the number of shared particles between the two galaxies is above the threshold, `threshold`. This last rule is recursive because the relation Galaxies appears both in the head and in the body of the rule.

In Datalog programs, base data is referred to as Extensional Database predicates (EDBs) (*e.g.*, `Particles`). Derived relations are Intensional Database predicates (IDBs) (*e.g.*, `Edges` and `Galaxies`). EDBs occur only in bodies, while IDBs appear in the heads of the rules and can also appear in the bodies.

Without aggregations and without negations (*i.e.*, in the case of conjunctive queries), the result of a Datalog program is its least fixpoint, which can be computed by repeatedly evaluating the rules in any order until no new facts (*i.e.*, tuples) are found.

When a positive Datalog program includes aggregations, as in our example, it is evaluated in a stratified manner, which means that the program is evaluated one subset of rules at the time. More specifically, the aggregate function is evaluated only after all predicates that form the right hand-side of the corresponding rule have been fully evaluated (*i.e.*, have reached their fixpoint). In our example, the condition holds trivially since the rule with the `$Count` aggregate has only an EDB in its body rather than one or more IDBs. Similarly, the rules that use the aggregate value are blocked until the aggregate has been computed. In the example, rules (2) and (3) are not evaluated until rule (1), which contains the aggregate function `$Count`, has been evaluated.

In this paper, we focus on positive Datalog programs (no negated predicates) with aggregate functions that only occur

in IDBs. Given an IDB with a set of grouping attributes v and aggregate functions f, the semantics are those of group by aggregation: For all tuples that satisfy the body of the rule, the rule evaluates the value of each function f for each unique combination of values of v.

2.2 Shared-Nothing Iterative Processing

There exist three common approaches to evaluating iterative query plans in shared-nothing systems:

- *Bulk synchronous*: Each iteration computes a completely new result from the result of the previous iteration. A synchronization barrier separates each iteration.
- *Incremental synchronous*: Each iteration computes a new result, then compares it with the result of the previous iteration and feeds only the delta to the next iteration. This approach corresponds to semi-naïve Datalog evaluation.
- *Incremental asynchronous*: New facts are continuously discovered without coordination between operator partitions and without any synchronization barriers.

Consider `GalaxyEvolution` from Figure 1. During the evaluation of rules (2) and (3), with bulk synchronous execution, each iteration i takes all the galaxies reachable from `GalaxiesOfInterest` within $i-1$ timesteps as input, then joins them with `Edges` to get all the galaxies reachable within i timesteps. With the incremental synchronous evaluation, the input of iteration i is only the set of galaxies that are reachable with no less than $i-1$ timesteps. The output contains only newly-discovered reachable galaxies. With asynchronous execution, operator partitions continuously discover and communicate reachable galaxies without synchronization barriers.

2.3 Problem Statement

The recursive query plans that we introduce (Section 3.2) support all three execution models. The key questions that we answer in the paper are (1) What class of positive Datalog queries with aggregation can be evaluated recursively and asynchronously in a shared-nothing systems, (*i.e.*, the aggregate function can be evaluated inside the iteration) (2) What query plans to use to evaluate such queries and how to execute these plans to obtain the best performance? And (3) which fault-tolerance method imposes a low overhead yet achieves fast recovery from failures in these iterative plans.

3. ASYNCHRONOUS EVALUATION OF DATALOG WITH AGGREGATION

In this section, we first introduce an extended class of aggregate functions that can recursively be evaluated in Datalog programs (Section 3.1). We then show how to generate query plans for the asynchronous and parallel evaluation of Datalog programs with this extended class of recursive aggregates (Section 3.2). Finally, we discuss optimizations that can significantly affect query performance and example applications that illustrate these performance trade-offs (Section 3.3).

3.1 Recursive Bag-Monotonic Aggregation

As described in Section 2.1, the non-recursive method to evaluate a Datalog program with aggregation consists in evaluating an aggregate function only after all its input IDBs have converged to their fixpoints. Furthermore, if an IDB with an aggregate function is used in the body of another rule, the evaluation of that rule also blocks until the aggregate function is evaluated.

In some cases, recursively evaluating these aggregates can speed-up convergence by pruning unnecessary partial results early. In contrast, the blocking strategy must evaluate each IDB in full, which may yield worse performance as we show in Section 6.2. The SociaLite work [39] has shown that a small class of aggregates, *meet* aggregates, can be evaluated recursively. These aggregate functions are associative, commutative, and idempotent binary operations defined on a domain S. These operations, denoted with $\wedge$, induce a partial order $\preceq$ on S, which is defined as: $\forall x, y \in S, x \preceq y$ if and only if $x \wedge y = x$. The result of the function on any two elements is the greatest lower bound with respect to this partial order. `$Min` and `$Max` are two examples of meet operations. Furthermore, if a Datalog program comprises a meet aggregate function defined on a finite set, and the rest of the program is monotonic with respect to the partial order induced by the aggregate, then the iterative evaluation of the rules converges to the greatest fixpoint. Finally, these programs can be evaluated incrementally [39] and asynchronously [38].

As in SociaLite, we support meet aggregates, which already enables us to express a subset of our target applications (see Section 3.3) . We observe, however, that many applications require aggregates other than meet aggregates, yet can still benefit from the recursive evaluation of those aggregates. Typical examples of these aggregates are `$Count` and `$Sum`. These aggregates are commonly used in analytical applications yet are not meet aggregates because they are not idempotent.

GalaxyEvolution is one example application that can benefit from the recursive evaluation of a `$Count` aggregate, which is illustrated by the Datalog program from Figure 2. This program computes the same result as the one in Figure 1 but uses different rules, which involve recursive aggregates. Here, the `Edges` IDB depends on the recursively defined `Galaxies` IDB. As a result, `Edges` and its `$Count` aggregate must be recursively evaluated as well. We show in Section 6.2 that the recursive version of the application can significantly reduce the run time because it avoids the computation of unnecessary tuples in `Edges`. The clustering algorithm DBSCAN [20] is another example application that can benefit from the recursive evaluation of a `$Count` aggregate (used during the density estimates).

The above examples and other similar examples that we encountered while working with scientists at the University of Washington motivate us to extend the notion of recursive aggregates to a broader class of aggregate functions: We show that it is possible to recursively evaluate aggregate functions that are commutative, associative, and *bag-monotonic* (but not necessarily idempotent). Examples of bag-monotonic aggregates include `$Count`, `$Sum`, which are not idempotent, and also include `$Min` and `$Max`. We start with a definition of a bag-monotonic aggregate:

DEFINITION 3.1. *Let S be a set of bags of tuples, and $x, y \in S$. A partial order $\preceq$ on S is defined as: $x \preceq y$ if and only if $x \subseteq_{bag} y$ [1]. An aggregate function $a : S \rightarrow V$ is* `bag-monotonic` *with respect to a partial order $\preceq$ defined on V, if for any $x, y \in S$ such that $x \preceq y$, we have $a(x) \preceq a(y)$.*

[1] $\subseteq_{bag}$ is bag containment, $x \subseteq_{bag} y$ if and only if for any tuple t that appears n times in x, t also appears m times in y, and $n \leq m$.

We can now define a Datalog program with a recursive, `bag-monotonic` aggregate. Let $g : T \to W$ be the function that takes a bag of tuples $R \in T$, does a group by on the set of grouping attributes, then applies the aggregate function a to each group k to generate a set of tuples $U = \{(k, v) \dots\}, v \in V, U \in W$. The partial order on T is defined as: $R_1 \preceq R_2$ if and only if $R_1 \subseteq_{bag} R_2$. The partial order on W is its Hoare order: $U_1 \preceq U_2$ if and only if $\forall (k, v) \in U_1, \exists (k', v') \in U_2$, $(k, v) \preceq (k', v')$, where $(k, v) \preceq (k', v')$ if and only if $k = k', v \preceq v'$. We refer to the rest of the program as $f : W \to T$, and require f to be monotonic with respect to the order defined on T and distributive. The whole program is then defined as the recursive application of the function $(f \circ g)$. Starting with a set of empty bags of tuples R_0, for each $i \geq 0$, we have:

$$U_{i+1} = g(R_i), R_{i+1} = f(U_{i+1}).$$

We illustrate the definition using Figure 2. In this program, the `$Count` aggregate computes the number of particles shared between any pair of galaxies, `(gid1,gid2)`, at adjacent timesteps. As the rules are evaluated, more particles can be found to satisfy the body of rule (3). As a result, the bag of particles that serves as input to `$Count` grows. Each bag is a superset of the previous bag, thus only causes `$Count` to be computed on supersets of the previous inputs. Since `$Count` is bag-monotonic, the output of the aggregate only increases. Furthermore, once a new pair of galaxies is discovered, no new tuples can cause the pair to be removed. Notice that if the condition in rule (2) was changed to `c < threshold`, the program would no longer be monotonic with respect to the partial order defined on the input to the aggregate operator.

Finally, we use a similar approach as in SociaLite to show that the naïve evaluation of such a program converges to a fixpoint, and that its semi-naïve evaluation is equivalent to the naïve evaluation. Since R_0 consists of sets of empty bags, we have $R_0 \preceq (f \circ g)(R_0)$. Using mathematical induction, if T is a finite set, then there must exist a finite n such that

$$R_0 \preceq (f \circ g)(R_0) \preceq \cdots \preceq (f \circ g)^n(R_0) = (f \circ g)^{n+1}(R_0),$$

where $(f \circ g)^n(R_0)$ is the greatest lower bound of the program.

The process of semi-naïve evaluation is the following. Starting from $R'_0 = R_0$, for each $i >= 0$, we have:

$$\begin{aligned} U'_{i+1} &= g(R'_i), \\ \Delta_{i+1} &= U'_{i+1} - U'_i, \\ R'_{i+1} &= R'_i \cup f(\Delta_{i+1}). \end{aligned}$$

We use mathematical induction to show that $U'_i = U_i$.

Basis: $U'_1 = g(R'_0) = g(R_0) = U_1$.

Inductive: assuming $U'_k = U_k$ for all $k \leq i$, then we have:

$$\begin{aligned} U'_{i+1} &= g(R'_i) = g(R'_{i-1} \cup f(\Delta_i)) \\ &= \dots \\ &= g(f(\Delta_1) \cup f(\Delta_2) \cup \cdots \cup f(\Delta_i)) \\ &= g(f(\Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_i)) \\ &= g(f(U'_i)) = g(f(U_i)) = U_{i+1}. \end{aligned}$$

3.2 Parallel and Asynchronous Evaluation

In this section, we show how to translate a Datalog program with bag-monotonic recursive aggregates into a query plan that can be executed *asynchronously* and *incrementally*. Our approach can be implemented in a broad class of big data management systems with only small extensions that we present in this section. Our approach then enables the asynchronous evaluation of even complex query plans with multiple recursive IDBs. The systems that we target are shared-nothing, dataflow, analytical engines, in which operators and data are horizontally partitioned across worker processes, and data can be pipelined from one operator to the next without going to disk and without synchronization barriers. Examples of such systems include Flink [2, 21], Dryad [27], parallel database systems [3, 23], and also our own system, Myria [25]. Spark [49] could also benefit from our approach if it was extended with pipelined data shuffling, while MapReduce [18] serves as a counterexample. Note that these specific systems already have their own approach to iterative processing (see Section 7). We use them as examples of the class of systems to which our approach also applies.

```
Galaxies(1,gid)       :- GaliesOfInterest(gid)      (1)
Galaxies(time+1,gid2) :- Galaxies(time,gid1),
                         Edges(time,gid1,gid2,c),
                         c >= threshold             (2)
Edges(time,gid1,gid2, :- Galaxies(time,gid1),
 $Count(*))              Particles(pid,gid1,time),
                         Particles(pid,gid2,time+1) (3)
```

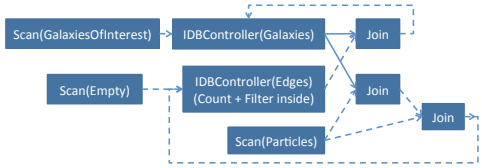


Figure 2: Datalog program for the GalaxyEvolution application using a recursive aggregate (top). Query plan for this application (bottom). Dashed lines indicate shuffling over the network. Note that: 1. we push the selection into the IDBController as an optimization, 2. since the Edges IDB does not have an initial input, we link a Scan, which reads an empty relation, to initialize the IDBController for Edges.

3.2.1 Recursive Query Plans

The incremental evaluation of recursive Datalog programs with bag-monotonic aggregate operators requires query plans that perform several functions: (1) At each iteration, the query plan must compute new facts based on the incremental changes to the states of the recursive IDBs since the last iteration. In the example from Figure 2, each iteration discovers new `Galaxies` tuples and new qualifying pairs of `Particles` tuples that join together. (2) The query plan must then update the state of the recursive IDBs based on the new facts. This state update may require the computation of the aggregate functions if present. (3) The query plan must compute and output the changes to the IDB states since the last iteration, such as the newly computed `Edges` and `Galaxies` tuples. (4) The plan must detect whether all IDBs have reached a fixpoint. We propose to use regular relational operators for (1). We encapsulate functions (2) and (3) into a new operator that we call IDBController. We introduce a second operator, the TerminationController, to check (4).

As shown in Figure 2, an IDBController has two input children. One child is the initial state of the IDB, which is not recursive. For `Galaxies`, this input is the relation `GalaxiesOfInterest`, although it can also be empty, as for `Edges`. The other child is the recursive input for the new tuples that are generated during the iterations.

We let the IDBController perform the group-by and aggregation within itself to avoid generating and moving unnecessary tuples between operators. Instead, the IDBController computes

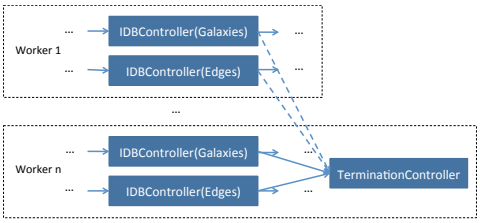


Figure 3: GalaxyEvolution query plan with IDBControllers and an TerminationController. Other operators are omitted.

directly the aggregated state. Additionally, while the IDBController accumulates the complete IDB state, it outputs only changes to that state in order to support incremental evaluation.

An IDBController has two execution modes: synchronous and asynchronous. In synchronous mode, the IDBController first fully consumes its initial input, initializes the state of its IDB, and outputs that state. Second, it accumulates tuples from the recursive input until the end of one iteration. It then updates the state of its IDB and outputs changes to that state for the next iteration. In asynchronous mode, the IDBController consumes input tuples on either input as they become available. For each input tuple, it updates the state of the corresponding group, and outputs the new aggregate value if it has changed. We explore the performance implications of each execution mode in Section 6.2. In the rest of this subsection, we focus on the asynchronous mode.

There is one TerminationController for each iterative computation. This operator collects periodic messages from all IDBControllers indicating whether the operators produced any new tuples since the last message. We further discuss the details of the TerminationController and fixpoint detection in Section 3.2.2.

Figure 3 illustrates the positions of the two operators in a query plan by showing the query plan produced for GalaxyEvolution. In all other figures, we omit the TerminationController (and the parallelism) when showing query plans.

Given these two special operators, we are now able to translate a positive Datalog program into an asynchronous recursive query plan, as shown in Algorithm 1. Briefly, the procedure translates the head of each rule into an IDBController, the body of each rule into a relational query plan. It then connects the appropriate inputs and outputs. As an example, Figure 2 shows the generated query plan for GalaxyEvolution, where the two IDBControllers recursively depend on each other.

3.2.2 *Lightweight Termination Check*

Evaluating a recursive query asynchronously raises an important issue: how to decide if the program has terminated. Since there are multiple IDBs being evaluated on multiple machines, and messages are sent through the network with possible delays, we need a protocol to guarantee a correct termination.

To detect termination, we propose a lightweight protocol that only requires small extensions to operators. First, we extend all operators with the ability to propagate a special message called *end-of-iteration* (EOI), based on the following three rules. Later in this section, we show how these messages serve as markers to determine that the fixpoint has been reached.

1. An IDBController generates the first EOI when its initial input has been fully consumed, and all the following EOIs when it receives an EOI from its iterative input.
2. An operator which is not an IDBController generates an EOI when it has received at least one EOI from each of its children since the last time it generated an EOI.
3. The internal design of an operator needs to ensure that, after it generates an EOI, it does not generate more tuples until it has fetched some new tuples from its children.

Algorithm 1 Translate a Datalog program into an asynchronous recursive query plan

```
1.  function GeneratePlan(P)
2.      Input: Datalog program P
3.      R ← Set of all rules in P
4.      X ← Set of all IDBs in P
5.      Y ← Set of all EDBs in P
6.      Instantiate an TerminationController E
7.      for (x ∈ X) do
8.          Instantiate an IDBController C_x with empty inputs
9.          Connect control output of C_x to input of E
10.     for (y ∈ Y) do
11.         Instantiate a Scan S_y
12.     for (r ∈ R with h ∈ X as head) do
13.         Translate the body of r into a relational query plan Q_r
14.         for (x ∈ X that appears in the body of r) do
15.             Connect output of C_x to Q_r as input
16.         for (y ∈ Y that appears in the body of r) do
17.             Connect output of S_y to Q_r as input
18.         if (body of r contains only EDBs) then
19.             Union the output of Q_r with the initial input of C_h
20.         else
21.             Union the output of Q_r with the recursive input of C_h
22.     for (x ∈ X) do
23.         if (x has nothing on its initial input) then
24.             Instantiate and connect a Scan(∅) to its initial input
25.         if (x has nothing on its recursive input) then
26.             Instantiate and connect a Scan(∅) to its recursive input
```

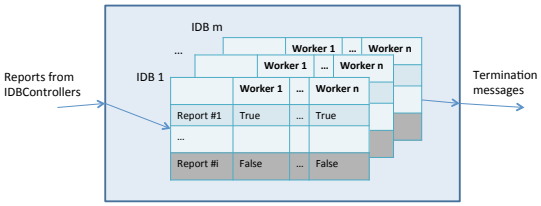


Figure 4: The internal state of an TerminationController

Additionally, an IDBController sends a message to the TerminationController every time it generates an EOI. Each message is a triple (R, α, b), where b is a boolean value indicating whether the IDBController for relation R on worker α generated any new tuples since the last message. The TerminationController maintains one relation for each IDB with one column per worker. Whenever it receives a message from a worker, the TerminationController appends the boolean value to column α in table R as shown in Figure 4.

When the TerminationController finds a *full-false row*, in which which all attribute values in all tables are `false`, the TerminationController signals that the query has completed. To ensure that the above condition correctly identifies the termination of the query, we need to prove two lemmas:

LEMMA 3.1. *When the query terminates, there exists a row n such that any row i with $i \geq n$ is a full-false row.*

PROOF. The above lemma follows directly from the rules in the protocol: operators output an EOI in response to receiving EOIs on all their children and there is no termination condition for this process. Starting from some time t, if the query will not generate any more data, then all the following messages must be `false`. □

LEMMA 3.2. *If row k is a full-false row, then any row i, $i > k$, is also a full-false row.*

PROOF. We prove the lemma by contradiction. Consider that operators produce EOIs with increasing sequence numbers (in our protocol, EOIs need not be numbered). We use $\text{EOI}_{t,A}$ to denote the t-th EOI produced by IDBController A. A outputs $\text{EOI}_{0,A}$ after consuming its entire initial input. Because an operator only propagates an EOI after receiving *at least one* EOI from each of its children, A will be able to produce $\text{EOI}_{1,A}$ only after all IDBControllers B that produce data consumed by A have produced their $\text{EOI}_{0,B}$. Furthermore, we use s_1 to denote the largest EOI sequence number that A has received as input before generating $\text{EOI}_{1,A}$, then $s_1 \geq 0$. [2] By induction, A outputs $\text{EOI}_{i+1,A}$ only after any recursively connected IDBController B has produced $\text{EOI}_{s_{i+1},B}$, and $s_{i+1} \geq i$.

Consider the case where k is a *full-false row* but there exist some `true` cells following row k. Consider t_a, the earliest tuple that was generated among all the `true` cells after row k. By rule 3 in the protocol, t_a was generated in response to some other tuple t_b and, by definition of t_a, t_b must belong to a cell before row k. On the other hand, since $\text{EOI}_{s_k,B}$ was generated before $\text{EOI}_{k,A}$ based on the above induction and t_a goes after $\text{EOI}_{k,A}$, then t_b must go after $\text{EOI}_{s_k,B}$. Since $s_k \geq k-1$ and k is a *full-false row*, we know t_b must live in a cell after row k. Thus the lemma is proven by contradiction. □

3.3 Execution-Time Optimizations

Iterative query processing with aggregation is amenable to several execution time optimizations. Importantly, as we show in Section 6.2, selecting different execution strategies for the same query plan can significantly impact performance.

The first optimization is the decision to execute a query either *synchronously or asynchronously*. We use `sync` and `async` to denote these two execution modes respectively. We find that it is not the case that the latter always outperforms the former for applications that tolerate asynchrony.

Asynchronous processing has the benefit of resilience against uneven load distribution because workers process data without synchronization barriers. This benefit, however, can be offset by a larger amount of intermediate result tuples generated during execution. We demonstrate experimentally that, each combination of iterative application and execution strategy (`sync` or `async`), can generate a different number of intermediate result tuples, which significantly affects performance. The `sync` and `async` execution modes are supported by the IDBControllers as described in Section 3.2.1. The system sets the execution mode when initializing these operators for a query.

A second query execution strategy choice that can yield dramatically different numbers of intermediate result tuples is the *join pull order*. Binary operators, such as joins, can consume their input data in several ways. One approach is to fully consume one of the inputs before fetching any data from the other one. Alternatively, a join can pull from both children with or without preference if it is a symmetric operator such as a symmetric hash-join. We observe that, when one child of a join is an EDB and the other one is a recursive IDB, the join pull order can significantly affect the number of intermediate result tuples. More precisely, we comparatively evaluate three execution strategies:

[2] To see why A can receive $\text{EOI}_{s_1,B}$ with $s_1 \geq 0$ before generating $\text{EOI}_{1,A}$, consider the case where B generates data consumed by A but is itself independent from A.

```
CC(x,x)          :-  Edges(x,_)              (1)
CC(y,$Min(v))    :-  CC(x,v), Edges(x,y)     (2)
                 :-  CC(y,v)                 (3)
```

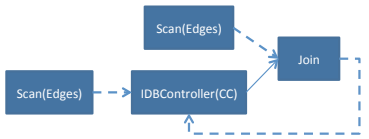


Figure 5: Datalog query (top) and recursive query plan (bottom) for connected components. The input EDB Edges(x,y) contains follower-followee edges.

- `build EDB`: The join operator first consumes all data from one input and builds a hash-table in memory. It then streams the other input and probes the hash table. Note that it is only possible to fully consume the input that is not recursive.
- `pull IDB`: The join only consumes data from its EDB child if no data is available on the recursive IDB input.
- `pull EDB`: Opposite to `pull IDB`, the join only consumes its IDB child if no data is available on the EDB child.
- `pull alter`: The join pulls alternatively from the two children without favoring one over the other.

The presence and impact of the intermediate result tuples depend both on the execution strategy and the application. In GalaxyEvolution (Figure 2), the number of intermediate result tuples is the same independent of the execution strategy. However, for other applications, the number of intermediate tuples varies when the execution strategy changes. To better illustrate this point, we consider two additional applications.

Consider the connected components application shown in Figure 5, which computes the connected components in a graph. In this application, rule (1) initializes the connected components: each vertex starts as its own connected component with its identifier. Rules (2) and (3) recursively compute the connected components: For all combinations of facts that satisfy the bodies, the aggregate function `$Min(v)` keeps and propagates only the current minimal component ID v for each vertex y. The evaluation of `$Min(v)` is inter-twined with the discovery of new facts, where intermediate result tuples are generated until convergence. This is true in both the synchronous and asynchronous modes, but the number of intermediate result tuples varies when the join pull order changes in the asynchronous mode. Intuitively, if the join between `Edges` and the newly updated component values from `CC` favors the recursive input (`pull IDB` execution method), then it prioritizes the propagation of values closer to convergence, which ultimately reduces the number of intermediate result tuples.

We also consider another application from the bibliometrics domain. The application computes the least common ancestor (LCA) for pairs of publications in a citations graph. An ancestor a of a paper p is any paper that is transitively cited by p, and the LCA a of two papers p_1 and p_2 is the least ancestor of both p_1 and p_2. Ancestor order is defined by the triple: (depth, year, paper_id). Figure 6 shows the Datalog and query plan for this application. The IDB `Ancestor` uses the aggregate function `$Min` to keep the length of the shortest path between two papers. In the synchronous mode, each iteration i only generates new pairs of papers with shortest path lengths equal to i. Once such a tuple is emitted, it will never be replaced by another tuple, which means there will be no unnecessary intermediate result

```
Ancestor(b,a,1)                    :- Cite(b,a), b<seed     (1)
Ancestor(p,a,$Min(depth+1))        :- Ancestor(p,b,depth),
                                      Cite(b,a)             (2)
LCA(p1,p2,$Min(greater(            :- Ancestor(p1,a,d1),
 d1,d2),year,a))                      Ancestor(p2,a,d2),
                                      Paper(a,year),p1<p2   (3)
```

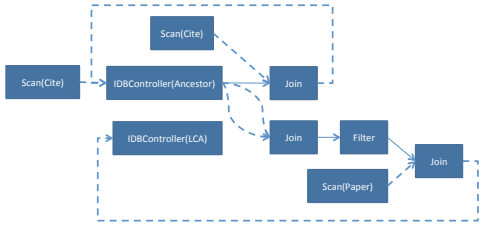


Figure 6: Datalog query (top) and recursive query plan (bottom) for LCA. The inputs are two EDBs: Paper(a,year) and Cite(b,a). The computation produces two outputs: Ancestor(b,a,depth) and LCA(p1,p2,depth,year,a).

tuples in this mode. In the asynchronous mode, however, a tuple of `Ancestor` may be replaced by another tuple with a smaller path length, which leads to a larger number of intermediate result tuples. The number of intermediate result tuples grows even larger in rule (3) with a self-join on `Ancestor`.

We evaluate the performance implications of these different execution alternatives on these three applications in Section 6.2.

4. FAILURE HANDLING

Several techniques exist to handle failures during the execution of iterative queries. The simplest approach is to restart the entire computation. The most well-known alternatives include data materialization at synchronization boundaries [14, 18, 47], checkpointing the state of the entire computation either synchronously [33,36] or asynchronously [31], and restarting using lineage tracking and periodic checkpoints [40, 49].

An important goal of our work is to develop synchronous and asynchronous iterative query processing methods that are simple to add to an existing shared-nothing engine. For this reason, we focus on failure handling methods that can be implemented by simply inserting failure-handling operators into query plans rather than modifying all operators (and queues) with the ability to checkpoint and recover state. We do not develop a new failure-handling method. Instead, we study fault-tolerance methods in the context of iterative query plans.

Similar to MapReduce [18] and Hadoop [47], we focus on fault-tolerance methods that buffer data on the producer side of data shuffling operators in the query plan. When a shuffle consumer worker fails, the shuffle producer resends the buffered data to a newly scheduled instance of the failed worker. Unlike MapReduce, but similar to the River system [11], the upstream backup methods developed for stream processing engines [26] and also used with shared-nothing database management systems [44], we buffer the data *in-memory and without blocking*. Figure 7 illustrates the approach. Buffers can spill to disk but we did not find that necessary in the applications that we used in the experiments.

For failure detection, we use simple heartbeat messages from the workers to the master, but we could also use more sophisticated cluster configuration methods [45].

During normal computation, each worker buffers its outgoing messages to other workers in memory in these shuffle operator buffers. If a worker fails, the master starts a new worker process and reschedules the failed query fragment on that process.

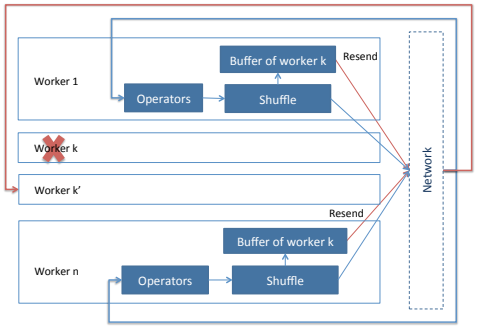


Figure 7: Fault-tolerance through data buffering in shuffle producer operators. When worker k fails, a new worker is re-scheduled and all other workers re-send their buffered data.

The approach could also recover the failed fragments in parallel using multiple workers to speed-up recovery [40, 49]. The newly scheduled fragments process the iterative query from the beginning, while all other workers resend their buffered data.

The above failure handling methods are known. Our contribution is to study how amenable iterative computations are to optimizations that are possible for these buffer-based fault-tolerance methods. We study the following optimizations:

- `Append Buffer`: Buffer all data in FIFO queues with no optimization.
- `Aggregate Buffer`: The idea is equivalent to using a MapReduce combiner. In the case when the data being buffered is part of an IDB with aggregation, the data can be partially aggregated at the sender.
- `Prioritized Buffer`: Prior work [50] has shown that prioritizing the execution of specific tuples can speed-up convergence of iterative computations. For example, in the case of Connected Components, prioritizing tuples with the lowest component IDs can help to propagate these lower values faster. The idea of the prioritized buffer is to support such prioritization during failure recovery be re-ordering tuples in the buffer based on an application-specific priority function. For Connected Components, the sort order is increasing on component ID.

We empirically compare the overhead and recovery time of the above failure-handling methods in Section 6.3.

5. IMPLEMENTATION

We implement our approach in the Myria [5,25] data management system. Myria's query execution layer, called MyriaX, is a shared-nothing distributed engine, where there is one master node and multiple worker nodes. As in HadoopDB [7], datasets ingested into Myria are sharded into PostgreSQL databases local to each node. MyriaX can read from other sources but we use PostgreSQL in our experiments. Once data is read out of PostgreSQL it is processed entirely in memory.

MyriaX is a relational engine. Query plans comprise relational algebra operators that are partitioned across workers. To distribute data across operator partitions, we use hash-partitioning and insert data shuffling operators to perform data re-distribution when necessary. Within each worker, query execution is pull-based: each operator produces output by pulling data from its children and returning it to its parent. Communication between workers is push-based: producer operators aggressively push data to consumers, with backpressure-based

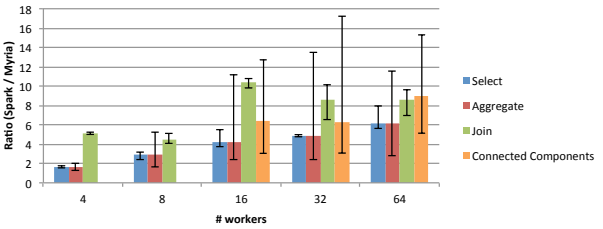
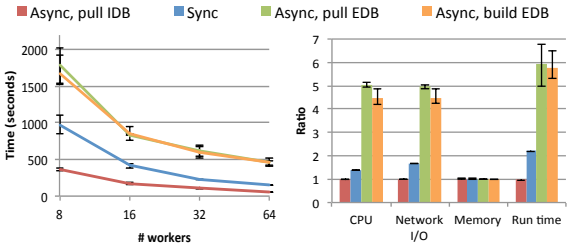


Figure 8: A comparison of Spark and MyriaX on four queries: Select (`R(x,y) :- Twitter(x,y), x < 5000000`), aggregate (`R(x,sum(y)) :- Twitter(x,y)`), join (`R(x,z) :- Twitter(x,y), Twitter(y,z)`), and connected components. MyriaX completes these queries $1.5\times$ to $10\times$ faster on average than Spark. For connected components, Spark (using GraphX) runs out of memory for small cluster sizes and large data.



(a) Query run times of different cluster sizes and execution models

(b) Relative resource consumption of different execution models with 32 workers. (`Async, pull IDB`) serves as reference.

Figure 9: Connected Components: query run times and resource consumption.

flow control used to balance the rates of data production and consumption while keeping the dataflow pipeline full. MyriaX processes tuples in batches to remove function call and network protocol overheads.

To justify our choice of the Myria engine for the implementation and evaluation of recursive query plans, we compare Myria's basic query execution performance to Spark [49], a state-of-the-art engine that includes support for synchronous iterative computations.

Figure 8 shows the results [3, 4]. Each bar shows the ratio of query execution time of Spark over Myria. Selection, aggregation and connected components are running on top of the full Twitter [29] dataset, which contains approximately 41 million vertices and 1.5 billion edges of the "follower, followee" relationships. For join, we use a subset of Twitter, which contains 60 thousand vertices and 1.5 million edges, because Spark could not produce results when a larger subset was used due to large memory usage. The join result has around 400 billion tuples. In all cases, Myria outperforms Spark with also a smaller variance in query execution times. These experiments illustrate that MyriaX achieves state of the art performance on standard queries, including iterative queries and is thus a good platform for the study of the performance differences between recursive query plan execution methods presented in this paper.

6. EVALUATION

In this section, we evaluate the performance of our recursive query plans. The experiments address the following: (1) *Does asynchronous query evaluation always lead to the fastest run times?*, (2) *Do the variants of asynchronous evaluation (Section 3.3) matter? When does each variant lead to the fastest query run time and why?*, (3) *Which fault-tolerance approach yields the best trade-off between run time overhead and failure recovery time?*

We evaluate our techniques using three applications and datasets:

- Connected Components (Figure 5): We compute the connected components on a subset of the Twitter graph [29], which contains 21 million vertices and 776 million edges. To study how the graph degree distribution may affect results, we also compute the connected components for three synthetic graphs generated using Snap [30]. These graphs are power-law graphs obtained by varying the exponent of the power law distribution. Each graph has 21 million vertices and either 192 (dense), 81 (medium), or 20 (sparse) million edges.
- GalaxyEvolution (Figure 2): We compute galactic merger graphs on an astronomy simulation [6] that is 80GB in size with 27 timesteps.
- LCA (Figure 6): We determine the least common ancestor in a real bibliometrics dataset obtained from a UW collaborator containing 2 million papers and 8 million citations.

6.1 Experimental Method

We run all experiments using our Myria prototype implementation (Section 5) in a 16-node shared-nothing cluster interconnected by 10 Gbps Ethernet. Each machine has four Intel Xeon CPU E5-2430L 2.00GHz processors with 6 cores, 64GB DDR3 RAM and four 7200rpm hard drives. We vary cluster size using 8 or 16 machines with 1 to 4 worker processes each.

In each experiment, we measure the run time and resource consumption of each query while it executes until convergence. We report the query run time, total CPU time across all workers, total network I/O (number of tuples sent), and the *maximum* memory consumption for the entire query (number of tuples in operator states and buffers). All queries are executed five times, and we report the average values along with min/max values as error bars. Unless stated otherwise, we use round-robin partitioning for the base data (EDBs), hash-based partitioning for the operators in the query plans, present resource consumption results for the 32-worker cluster configuration (other sizes exhibit the same trends), and normalize the figures to the resource consumption of the best evaluation strategy to enable comparison across resource types.

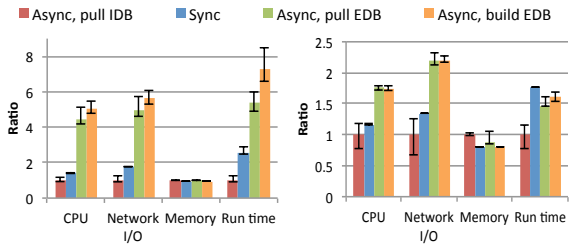
6.2 Execution Model

In this section, we evaluate the performance of the various execution models described in Section 3.3. In particular, we evaluate synchronous versus asynchronous execution strategies and, for asynchronous execution, compare how EDB-first, IDB-first, or balanced pull orders affect convergence.

6.2.1 Connected Components

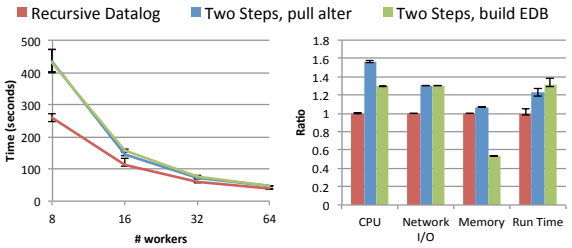
Figure 9 shows the query run time and resource consumption results for Connected Components on Twitter: we find that using asynchronous execution and preferring new IDB results (`(async, pull IDB)`) yields both the fastest query run time and the lowest overall resource utilization. Synchronous iteration is about a factor of 2 slower. This result is expected, as the

[3] Results are generated in memory, not materialized to disk.
[4] Here we use round-robin partitioning for all datasets in Myria to make it a fair comparison with Spark and HDFS.



(a) Dense (b) Sparse

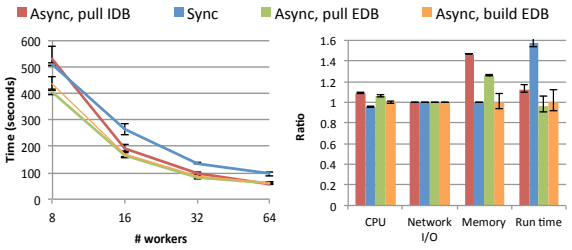
Figure 10: Connected Components on synthetic datasets: relative resource consumption of different execution models with 32 workers. `(Async, pull IDB)` serves as reference.



(a) Query run times of two-step and recursive Datalog.

(b) Relative resource consumption of two-step and recursive Datalog with 32 workers. Recursive Datalog serves as reference.

Figure 11: GalaxyEvolution: query run times and resource consumption, two-step versus recursive Datalog.



(a) Query run times of different cluster sizes and execution models.

(b) Relative resource consumption of different execution models with 32 workers. `(Async, build EDB)` serves as reference.

Figure 12: GalaxyEvolution: query run times and resource consumption.

benefits of asynchronous execution for Connected Components have been widely reported [35, 38].

However, we surprisingly find that *synchronous iteration is significantly faster* than `(async, pull EDB)` and `(async, build EDB)`. Asynchronous models range from 2× faster to 6× slower than synchronous, depending on how they propagate data. For this query, pulling from the IDB as much as possible helps to propagate small component IDs faster across the network to `CC` on remote nodes. This helps reduce the amount of intermediate result tuples significantly and thus achieves faster convergence and lower resource consumption. In contrast, the strategies that prefer to load the EDB into memory generate many intermediate tuples that are later replaced; `build EDB` is slightly faster than `pull EDB` because it only builds a single hash table, saving some computation.

The synchronous model achieves a middle ground between the asynchronous strategies. It is slower than `(async, pull IDB)` because of the global barrier between each iteration step. However, it is able to aggregate away intermediate results at the barriers, and this reduction in redundant work dominates the EDB strategies. The sizes of intermediate results are immediately visible among all four techniques when considering the network I/O: `(async, pull IDB)` shuffles fewer than one quarter of the tuples of the other asynchronous methods.

Figure 10 shows the results on the synthetic datasets. We only show the resource consumption of the dense and the sparse datasets because the pattern of the medium dataset falls between them. The dense dataset yields similar results to the Twitter dataset. In contrast, for the sparse dataset, `sync` becomes the slowest strategy in terms of query run time. This is caused by the long tail of the sparse graph. However, for other types of resource consumption, `sync` still sits in between the two asynchronous strategies, which is similar to the Twitter dataset.

To our knowledge, even though Connected Components is such a well-studied problem, no prior report has illustrated the subtleties in how strongly the choice of execution strategy affects distributed system performance.

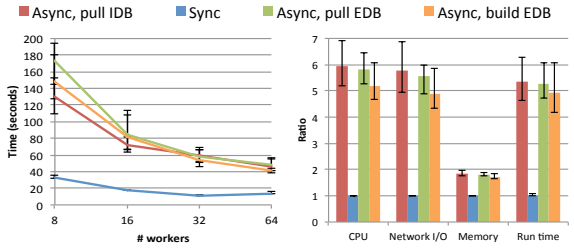
6.2.2 *GalaxyEvolution*

An important benefit of our approach is its focus on general-purpose Datalog programs as opposed to focusing only on processing graphs as in the case of SociaLite [38, 39] and several other engines [31, 46]. In GalaxyEvolution, the input data is a relation tracking particles through galaxies over time, and the goal is to compute the historical merger graph for a set of galaxies of interest at present day. As we described above, there are two ways to query the data. One approach Figure 1 first computes a full `Edges` relation for the galaxies in the simulation, in essence a full graph of galaxies, then extracts the sub-graphs reachable from the galaxies of interest. The alternate approach (Figure 2) represents the entire computation directly using recursive Datalog with our novel support for bag-monotonic aggregation.

In this experiment, we randomly select one percent of the present day galaxies as the groups of interest. We compare our novel recursive Datalog plan (the choice of strategy does not matter, for reasons we discuss below) to the two-step approach using two execution models: `pull alter` for the join in step 1 or `build EDB` for building the EDB hash table first since they have different memory consumption on hash tables. As the results in Figure 11 show, using recursive Datalog leads to a 25% faster total run time and a lower resource utilization except for memory. The performance and network I/O gains are explained because we avoid the computation of unnecessary `Edges`. The higher memory utilization comes from having both recursive join operators active at the same time in one query rather than computing them one at a time in two separate steps.

Next, we focus on the recursive Datalog approach. Figure 12 shows the performance of different execution methods. To emphasize the differences between these models, we change the selectivity of the GalaxyEvolution query by using all galaxies in the groups of interest and also lowering the `threshold` to ensure that the bottleneck of the query is not the disk I/O.

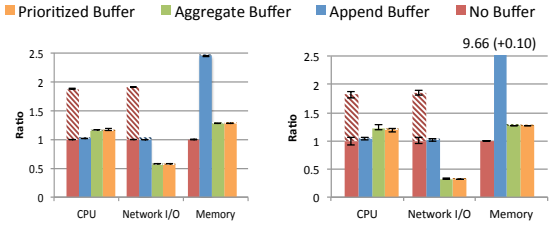
As we can see, `(async, pull EDB)` and `(async, build EDB)` yield the lowest run time. The latter does so with less memory because it never builds a hash table for the IDB input. These results are in contrast with Connected Components, where `(async, pull IDB)` is the most efficient execution model. The key reason is that there are no invalid intermediate



(a) Query run times of different cluster sizes and execution models.

(b) Relative resource consumption of different execution models with 32 workers. `(Async, pull IDB)` serves as reference.

Figure 13: LCA: query run times and resource consumption.



(a) Pull from IDB

(b) Pull from EDB

Figure 14: Connected Components: relative resource consumption of different buffers with 32 workers. Filled bars: no failure, patterned bars: overhead to recover from a failure. `No buffer`**, no failure serves as reference. In (b), the memory consumption of the Append Buffer reaches** 9.66 **without failures and** 9.76 **with failure. It is truncated in the figure.**

results as GalaxyEvolution converges to a fixpoint; each step learns new facts about the next timestamp. Hence, prioritizing the IDB does not help to converge faster, and instead the symmetric joins employed by `pull IDB` and `pull EDB` spend time updating two hash tables, though `pull EDB` finishes one child and switches to a single-sided join earlier. The `sync` uses the same amount of resources as the best `async` methods but it is slower because of the global barrier.

6.2.3 *Least Common Ancestor*

Figure 13 shows the performance results for the LCA application. As the Datalog program is essentially stratified by depth, the synchronous execution model always finds ancestors at the lowest depth and has no unnecessary intermediate result tuples in the `Ancestor` relation. In contrast, the asynchronous model generates many such intermediate result tuples in `Ancestor`, and even more such tuples are generated when `Ancestor` is joined with itself to compute the `LCA` relation. Because of the large number of intermediate result tuples, and their quadratic impact on result size, all `async` strategies yield much worse performance than `sync`. In general, the three `async` models have similar resource consumption numbers when varying the cluster size, although in this figure, `(async, build EDB)` slightly outperforms the other methods.

6.2.4 *Summary*

In summary, our experiments show the following trends:

- *Asynchronous query evaluation does not always lead to the fastest query run times.* For Connected Components `async` only works well combined with the right execution strategy (join pull order), and for stratified applications like LCA, asynchronous query evaluation performs unnecessary work that synchronous evaluation can avoid.
- *The variants of asynchronous evaluation (Section 3.3) have a big impact on query run time.* Our study of Connected Components shows, for the first time, that variants can *significantly* affect performance. The system should favor propagating newly-generated IDB tuples only when it will not generate many intermediate result tuples; otherwise using single-sided joins with fewer hash tables saves computation.

6.3 Failure Handling

In this subsection, we evaluate the failure-handling approaches described in Section 4 on the same three applications. For each query, we first evaluate the resource consumption overhead of fault-tolerance in the absence of failures. Then, we kill one worker during the query execution, and compare the total amount of resources used to process the query using either query restart (`No Buffer`) or one of the three buffer-based methods described in Section 4. We measure resource consumption instead of run time as it measures the total overhead on the cluster independent of how the recovery tasks are scheduled. We show the results when killing one worker approximately 70% of the time into the query execution. Similar patterns, though with somewhat different values, emerge when killing a worker earlier during the query execution.

We perform all experiments twice. First, we randomly partition EDBs. These EDBs must be shuffled during the query execution. Second, we hash-partition EDBs before the query execution such that they only need to be read locally. The difference between the two approaches lies only in the number of shuffle operators and in-memory buffers: when EDBs are hash-partitioned, shuffles after scans are not needed, which saves in-memory buffers. We find that all the trends are *identical* for both scenarios. The overheads are uniformly somewhat larger when an extra shuffle operator is added. We thus only show the results with the hash-partitioned EDBs.

Figure 14 shows the fault-tolerance overheads of the `(async, pull IDB)` or `(async, pull EDB)` execution methods. Each bar represents the ratio of resource utilization of one buffer type compared with execution without any failure handling. The filled portions at the bottom show the ratios in the absence of failures, while the portions with diagonal stripes on top show the additional overhead to recover from a failure.

In the absence of failures, maintaining buffers in shuffle operators adds overhead. Basic Append buffers add significant memory overhead, especially for `(async, pull EDB)` as it generates more intermediate tuples. The Aggregate and Prioritized buffers dramatically cut memory and network I/O overheads while only minimally increasing CPU overheads. These two methods have even lower network I/O than no buffering due to the data aggregation they perform before shuffling.

In the case of failure, all three buffer-based methods incurred *negligible* overhead due to failure recovery. Because this overhead is negligible, the extra work of sorting tuples in the buffer to prioritize the execution is unnecessary. As a result, the Aggregate Buffer delivers the best trade-off in terms of total resource consumption with or without failures.

Figure 15 shows the results for the GalaxyEvolution application. Aggregate Buffer and Prioritized Buffer are not applicable to this application since the only aggregate function is `$Count`. Similar to Connected Components, using Append Buffer consumes more memory during normal execution, but saves CPU

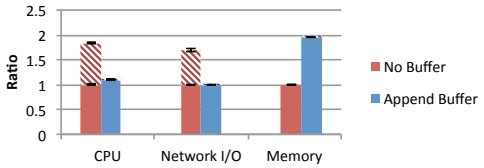
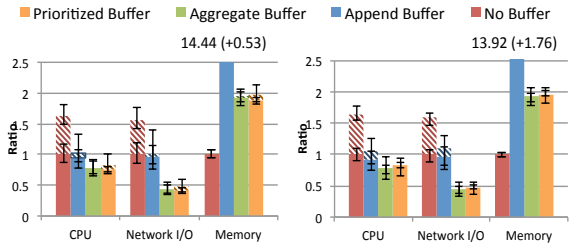


Figure 15: GalaxyEvolution: relative resource consumption of different buffers with 32 workers. Filled bars: no failure, patterned bars: overhead to recover from a failure. `No buffer`**, no failure serves as reference.**



(a) Pull from IDB (b) Pull from EDB

Figure 16: LCA: relative resource consumption of different buffers with 32 workers. Filled bars: no failure, patterned bars: overhead to recover from a failure. `No buffer`**, no failure serves as reference. The memory consumption of the Append Buffer reaches** 14.44 **without failures and** 14.97 **with failure in (a), and** 13.92 **without failures and** 15.68 **with failure in (b). They are truncated in the figure.**

time and network bandwidth in case of failures. Once again, the total resource consumption is nearly identical for an execution without failure or one with one failed worker.

Finally, Figure 16 shows the results for LCA. The trends are the same as for Connected Components. An interesting effect is that the early aggregation in the recovery buffers reduces total CPU consumption even in the absence of failures. Importantly, total resource utilization in the presence of a failure is, once again, nearly identical to the resource utilization without failure for all three buffer-based methods showing that prioritization during recovery is not necessary.

In summary, a lightweight buffer-based method for failure handling yields only a small increase in CPU utilization in the absence of failures, yet can recover from failures with negligible added CPU cost. The memory overhead of data buffering can be large but extending the buffers with early aggregates dramatically cuts these costs, which stay within 2X in all three applications tested. Interestingly, the failure-handling extensions reduce network I/O even without failures.

7. RELATED WORK

Adaptive Query Processing. The Eddy query processing mechanism [12] dynamically reorders operators in a query plan. It detects places where the reordering can happen, and routes tuples iteratively through operators based on costs. In contrast, our work focuses on query plans that are static but have loops.

Iterative MapReduce. MapReduce [18] and Hadoop [47] are known to be inefficient for iterative applications and several systems have been developed to address this limitation including HaLoop [14] and Twister [19]. Besides supporting iterations, PrIter [50] also provides the ability to prioritize the execution of subsets of data. OptIQ [37] uses program analysis to detect loop-variant data and evaluate it incrementally. [8] observes that recursive tasks only deliver output at the end and thus increases the cost of fault-tolerance. In contrast to our work, systems that extend Hadoop can only support synchronous iterations.

Synchronous-only systems. Beyond MapReduce, multiple systems have been designed for iterative applications and have introduced their own programming models. Some of them focus on *graph applications*, while others have more general programming models. In Pregel [33], a program consists of iterations, and in each iteration vertices can receive messages from the previous iteration, update states, and send messages out. Pregelix [15], which is built on top of Hyracks [13], is similar to Pregel but also supports both in-memory and out-of-core workloads efficiently. GraphX [24] is a graph processing framework built on top of Spark [49], a distributed in-memory dataflow engine. REX [35] provides a programming model that focuses on delivering deltas across iterations. All these systems focus on synchronous iterative computations.

Systems that also support asynchronous iterations. These systems generally have programming models that are based on message passing between units, such as graph vertices. They typically provide a set of low-level interfaces for users to implement their own applications. Some of these systems are specialized for graph processing, such as GraphLab [31] and Grace [46], while others are more general. Stratosphere [21] focuses on incremental iteration evaluation with different granularities: *superstep* for a full iteration and *microstep* for a single tuple. Naiad [36] proposes a general-purpose dataflow framework that supports nested loops. epiC [28] adopts the Actor-like programming model to encapsulate various parallel processing models into one system. To choose between a synchronous and asynchronous model, PowerSwitch [48] does the first comparison and comes up with a cost model to guide the switch between the two models. In contrast, our approach generates query plans from Datalog programs and these plan require only small changes to an existing shared-nothing system.

Datalog Evaluation Systems. Several systems focus on evaluating Datalog (with extensions) or equivalent high-level declarative languages. LogicBlox [4] is a single-machine commercial system that focuses on Datalog evaluation. In contrast, we focus on a shared-nothing implementation. GLog [22] provides a language similar to Datalog with extensions, then translates such a program into MapReduce jobs, which support only synchronous iterations. SociaLite [38, 39] is a distributed system that evaluates Datalog programs with meet aggregate functions. Asynchronous execution is supported within each *epoch* but not for the entire program. More importantly, the implementation is based on code-generation instead of having a general-purpose query engine. DeAL [41] is a single-machine Datalog evaluation system with support for aggregate functions such as Min/Max and Sum/Count. CALM [10] is a set of principles that connect distributed consistency with logical monotonicity, which leads to the Bloom [10] language. Bloom helps users identify unnecessary coordination. Bloom^L [16] is an extension to Bloom [10] with lattices, but without support for asynchronous evaluation on them.

Iterations in Scientific Workflows The Orbit [17] operator provides support for the iterative processing of workflow fragments. In contrast, our work focuses on Datalog programs.

Failure handling. Discussed in Section 4.

8. CONCLUSION AND FUTURE WORK

This paper developed an approach for large-scale iterative data analytics that combines the benefits of many existing systems: Users express their analysis in recursive Datalog with aggregation, which simplifies the expression of analytics from a variety of application domains. The system executes the analysis using parallel query plans that require only small extensions to an existing shared-nothing engine yet deliver the full power of incremental synchronous and asynchronous query evaluation even for query plans with multiple recursive IDBs. Finally, we empirically evaluate when different variants of query execution and failure handling methods deliver the fastest query run time for different applications. We find that no single method outperforms others. An important area of future work is thus to develop a cost-based optimizer to select the least-cost plan for each application including choosing between synchronous and asynchronous execution and selecting the pull order for each join operator. An alternate approach is to explore an Eddy [12] style executor that dynamically changes these choices based on the observed query performance.

9. ACKNOWLEDGMENTS

This project is supported in part by the National Science Foundation through grant IIS-1247469, the Intel Science and Technology Center for Big Data, a gift from EMC, an award from the Gordon and Betty Moore Foundation and the Alfred P Sloan Foundation, the Washington Research Foundation Fund for Innovation in Data-Intensive Discovery, and the UW eScience Institute. The astronomy simulation dataset was graciously supplied by T. Quinn, F. Governato, and S. Loebman of the UW Dept. of Astronomy. The bibliometrics dataset was provided by J. West of the UW iSchool.

10. REFERENCES

[1] Amazon EC2 spot instances. `http://aws.amazon.com/ec2/purchasing-options/spot-instances/`.
[2] Apache flink. `http://flink.apache.org/`.
[3] Greenplum. `http://pivotal.io/big-data/pivotal-greenplum-database`.
[4] LogicBlox inc. `http://www.logicblox.com/`.
[5] Myria: Big Data as a Service. `http://myria.cs.washington.edu/`.
[6] SDSS SkyServer DR7. `http://skyserver.sdss.org/dr7`.
[7] A. Abouzeid et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.
[8] F. N. Afrati et al. Mapreduce extensions and recursive queries. In *EDBT*, 2011.
[9] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. In *VLDB*, 2014.
[10] P. Alvaro et al. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, 2011.
[11] R. H. Arpaci-Dusseau et al. Cluster I/O with river: Making the fast case common. In *IOPADS*, 1999.
[12] R. Avnur et al. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.
[13] V. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.
[14] Y. Bu et al. HaLoop: Efficient iterative data processing on large clusters. In *VLDB*, 2010.
[15] Y. Bu et al. Pregelix: Big(ger) graph analytics on a dataflow engine. In *VLDB*, 2014.
[16] N. Conway et al. Logic and lattices for distributed programming. In *SoCC*, 2012.
[17] D. E. M. de Oliveira et al. Orbit: Efficient processing of iterations. In *SBBD*, 2013.
[18] J. Dean et al. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[19] J. Ekanayake et al. Twister: A runtime for iterative MapReduce. In *HPDC*, 2010.
[20] M. Ester et al. A density-based algorithm for discovering clusters in large spatial databases with noise. AAAI Press, 1996.
[21] S. Ewen et al. Spinning fast iterative data flows. In *VLDB*, 2012.
[22] J. Gao et al. GLog: A high level graph analysis system using MapReduce. In *ICDE*, 2014.
[23] A. Ghazal et al. Adaptive optimizations of recursive queries in teradata. In *SIGMOD*, 2012.
[24] J. E. Gonzalez et al. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.
[25] D. Halperin et al. Demo of the Myria big data management service. In *SIGMOD*, 2014.
[26] J. Hwang et al. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.
[27] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.
[28] D. Jiang et al. epiC: An extensible and scalable system for processing big data. In *VLDB*, 2014.
[29] H. Kwak et al. What is Twitter, a social network or a news media? In *WWW*, 2010.
[30] J. Leskovec et al. Snap.py: SNAP for Python, a general purpose network analysis and graph mining tool in Python. `http://snap.stanford.edu/snappy`, 2014.
[31] Y. Low et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *VLDB*, 2012.
[32] Large Synoptic Survey Telescope. `http://www.lsst.org/`.
[33] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.
[34] H. Menon et al. Adaptive Techniques for Clustered N-Body Cosmological Simulations. *ArXiv e-prints*, Sept. 2014.
[35] S. Mihaylov et al. REX: Recursive, delta-based data-centric computation. In *VLDB*, 2012.
[36] D. G. Murray et al. Naiad: A timely dataflow system. In *SOSP*, 2013.
[37] M. Onizuka et al. Optimization for iterative queries on MapReduce. In *VLDB*, 2013.
[38] J. Seo et al. Distributed SociaLite: A Datalog-based language for large-scale graph analysis. In *VLDB*, 2013.
[39] J. Seo et al. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*, 2013.
[40] Y. Shen et al. Fast failure recovery in distributed graph processing systems. In *VLDB*, 2014.
[41] A. Shkapsky et al. Graph queries in a next-generation datalog system. In *VLDB*, 2013.
[42] Sloan Digital Sky Survey. `http://cas.sdss.org/`.
[43] University of Washington eScience Institute. `http://escience.washington.edu/`.
[44] P. Upadhyaya et al. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD*, 2011.
[45] W. Vogels et al. The design and architecture of the Microsoft Cluster Service - a practical approach to high-availability and scalability. In *FTCS*, 1998.
[46] G. Wang et al. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.
[47] T. White. *Hadoop: The Definitive Guide*. 2009.
[48] C. Xie et al. SYNC or ASYNC: Time to fuse for distributed graph-parallel computation. In *PPoPP*, 2015.
[49] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[50] Y. Zhang et al. PrIter: a distributed framework for prioritized iterative computations. In *SoCC*, 2011.