Automating Incremental and Asynchronous Evaluation for Recursive Aggregate Data Processing

Qiange Wang[§], Yanfeng Zhang[§], Hao Wang[*], Liang Geng[§], Rubao Lee[*], Xiaodong Zhang[*], Ge Yu[§]
[§]Northeastern University, China, [*]The Ohio State University
wangqiange@stumail.neu.edu.cn,{zhangyf,yuge}@mail.neu.edu.cn
wang.2721@osu.edu,{liru,zhang}@cse.ohio-state.edu

ABSTRACT

In database and large-scale data analytics, recursive aggregate processing plays an important role, which is generally implemented under a framework of incremental computing and executed synchronously and/or asynchronously. We identify three barriers in existing recursive aggregate data processing. First, the processing scope is largely limited to monotonic programs. Second, checking on conditions for monotonicity and correctness for async processing is sophisticated and manually done. Third, execution engines may be suboptimal due to separation of sync and async execution.

In this paper, we lay an analytical foundation for conditions to check if a recursive aggregate program that is monotonic or even non-monotonic can be executed incrementally and asynchronously with its correct result. We design and implement a condition verification tool that can automatically check if a given program satisfies the conditions. We further propose a unified sync-async engine to execute these programs for high performance. To integrate all these effective methods together, we have developed a distributed Datalog system, called PowerLog. Our evaluation shows that PowerLog can outperform three representative Datalog systems on both monotonic and non-monotonic recursive programs.

CCS CONCEPTS

• **Information systems → Data management systems**; • **Computing methodologies → Distributed computing methodologies**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389712

KEYWORDS

recursive programs; aggregate operations; monotonic sequences; asynchronous execution; graph processing; Datalog

ACM Reference Format:
Qiange Wang, Yanfeng Zhang, Hao Wang, Liang Geng, Rubao Lee, Xiaodong Zhang and Ge Yu. 2020. Automating Incremental and Asynchronous Evaluation for Recursive Aggregate Data Processing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389712

1 INTRODUCTION

Large-scale recursive algorithms, including PageRank, Single Source Shortest Path (SSSP), and many others, play important roles in database and big data analytics. Recently, new development has re-emerged around Datalog [2, 6, 24, 35, 46, 49, 51, 59] for expressing such algorithms, due to its high-level declarative semantics and support for recursive programs. `Program 1` shows an example of using Datalog for SSSP. Programmers can use two *rules* of Datalog to express the algorithm: rule `r1` initializes the distance to the source node (with ID 1) as 0; and rule `r2` recursively declares a path from the source to node Y with length $d_x + d_{xy}$, if there is a path from the source to node X with length d_x and an edge from X to Y with length d_{xy}. The shortest path to Y is the shortest one of all possible paths from the source to Y, as expressed by the aggregate operation `min`. This type of recursive programs that have an aggregate operation in the recursion are called *recursive aggregate programs* (to be formally defined in Section 2.1). This example shows that Datalog only needs 2 rules for SSSP. In contrast, the graph processing systems [13, 15, 27, 28, 53, 68] require tens of lines of code, and other programming languages, such as Java, need more than one hundred of lines.

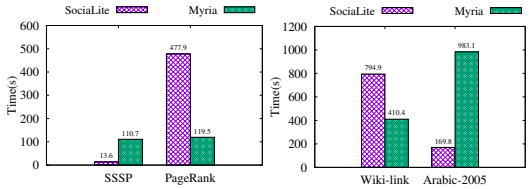
```
Program 1. Single Source Shortest Path
r1. sssp(X,d)          :- X=1,d=0.
r2. sssp(Y,min[d_y])   :- sssp(X,d_x),edge(X,Y,d_xy),
                          d_y = d_x + d_xy.
```

In Datalog, *semi-naive evaluation* [8, 24] is critical for efficient execution of Datalog rules. In each iteration, semi-naive evaluation performs the computation on the incremental (delta) result from the previous iteration and feeds the newly computed result to the next iteration. In this way, it can avoid redundant computation to achieve high performance. A number of Datalog systems [24, 29, 30, 46, 49, 50, 59] support semi-naive evaluation for recursive aggregate programs, when certain conditions are satisfied. We use a simplified term, called *monotonic programs*, to refer to these satisfiable programs. The results from a recursive sequence generated by a monotonic program are either monotonically increasing or monotonically decreasing; otherwise, it is a *non-monotonic program*. Existing systems only support semi-naive evaluation for monotonic programs, and none of the systems can verify these conditions automatically, so that users have to manually check the conditions for each program. It requires high human efforts and is also tedious and error-prone.

Meanwhile, there exist many non-monotonic programs, e.g., the original PageRank algorithm shown in `Program 2`. Instead of incrementally adding new ranking scores, the PageRank algorithm replaces old ranking scores with new scores in each iteration, making the scores changed non-monotonically. Semi-naive evaluation cannot be used for these non-monotonic programs. Previous studies [29, 30, 49] show that users can explicitly write some non-monotonic programs into monotonic ones and then use semi-naive evaluation. For example, users can write the delta-based PageRank algorithm [13, 68] that iteratively and incrementally accumulates non-negative ranking scores (as `Program 2.b` in Section 3.3). However, under what conditions a non-monotonic program can be written into a monotonic one that has the same result as that of the original non-monotonic program is unclear. Since existing systems cannot identify those convertible non-monotonic programs, they fall back to *naive evaluation*[1] in the execution by creating an additional `rank` table to perform the join operation in each iteration, which is computationally expensive for large datasets.

```
Program 2. PageRank (Declarative + Imperative)
r1. degree(X,count[Y]) :- edge(X,Y).
r2. rank(0,X,r)        :- node(X),r = 0.
r3. rank(i+1,Y,sum[r_y]):- node(Y),r_y = 0.15;
                        :- rank(i,X,r_x),
                        edge(X,Y),degree(X,d),
                        r_y = 0.85 · r_x/d.
```

In data processing, there are two execution modes, namely *synchronous execution* and *asynchronous execution*. Sync execution requires a barrier before next iteration, while async execution does not. Some Datalog systems adopt sync execution, e.g., SociaLite [46] and BigDatalog [49], while the others use the async, e.g., Myria [59]. However, executing some recursive aggregate programs asynchronously could result in wrong results. The correctness of incremental and asynchronous evaluation is not guaranteed in existing systems. Moreover, existing studies [12, 61] show that neither sync nor async execution consistently outperforms the other. Figure 1(a) shows that on LiveJournal dataset[26], SociaLite outperforms Myria on SSSP, but it loses on PageRank. In contrast, for SSSP in Figure 1(b), SociaLite beats Myria on Arabic-2005 dataset[5], but loses on Wiki-link dataset[60]. The performance dynamics and uncertainty come from over-controlled synchronization and under-controlled asynchronization. Between them, there must be a "properly-controlled" execution mode in the "sweet point" to achieve the best performance.



(a) LiveJournal **(b) SSSP**

Figure 1: Performance comparison of SociaLite (sync) and Myria (async) on variable algorithms and datasets.

In this paper, we lay an analytical foundation for correct incremental and asynchronous evaluation: we propose *monotonic recursive aggregate (MRA) evaluation*, a type of semi-naive evaluation but dedicated for recursive aggregate programs, and prove that when the MRA conditions are satisfied, a recursive aggregate program that is monotonic or even non-monotonic can be executed incrementally and asynchronously with its correct result. We design and implement a condition verification tool that can automatically check if a given program satisfies the MRA conditions by leveraging Satisfiability Modulo Theories (SMT) solver Z3 [63]. We further design a unified sync-async engine to execute these Datalog programs. We observe that the communication frequency can effectively serve as a control knob to adjust the level of asynchronization. We then develop an adaptive buffer method to adjust the amount of stored updates and control the communication frequency, leading to a properly-controlled execution. To integrate all these effective methods together, we have developed a new Datalog system, called PowerLog. Figure 2 shows its high-level structure. Our major contributions include:

- **An analytical foundation for correctness of incremental and asynchronous evaluation:** We prove that a recursive aggregate program that reaches a fixpoint can be correctly executed with incremental and asynchronous evaluation when the MRA conditions are satisfied. This

[1] As these PageRank-like algorithms are not defined in a declarative way, strictly speaking, naive evaluation cannot be used. But existing systems can follow the way of naive evaluation in the execution.

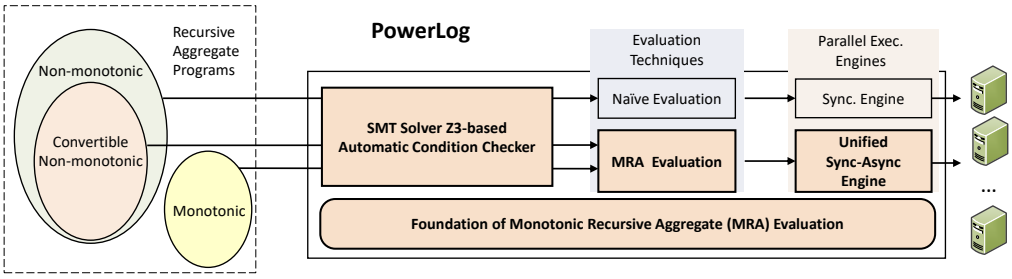


Figure 2: The overall structure of PowerLog. A recursive aggregate program is first processed by Automatic Condition Checker. If the MRA conditions are satisfied, the program will be executed with MRA evaluation on Unified Sync-Async Engine. Otherwise, it will be executed with naive evaluation on sync execution engine.

not only guarantees the correctness of incremental and asynchronous evaluation but also enlarges its scope to execute those non-monotonic but convertible programs.

- **An automatic condition check tool:** We leverage SMT solver Z3 to develop a condition check tool that can automatically check if the MRA conditions are satisfied for a given recursive aggregate program.
- **A unified sync-async engine:** We design and implement a distributed execution engine that can execute a recursive aggregate program in an adaptively sync-async way to achieve best performance.

In our experiments, PowerLog identifies that twelve widely-used recursive aggregate programs can pass the condition check and be executed with incremental and asynchronous evaluation but two programs cannot. We then compare PowerLog to three existing and representative Datalog systems (SociaLite, Myria, and BigDatalog) on six recursive aggregate algorithms, including CC, SSSP, PageRank, Adsorption, Katz Metric, and Belief Propagation, with a set of real-world datasets. On a 17-node Aliyun cluster, PowerLog can achieve 1.1x to 188.3x speedups over other Datalog systems.

2 PRELIMINARIES

2.1 Recursive Aggregate Datalog Programs

Datalog is best known for expressing recursive queries due to its high-level declarative semantics and its support for recursion. In a Datalog program, if a predicate appears in the head and a body, this rule is identified as a *recursive rule*. If an aggregation, e.g., `min` and `max`, appears in the head of a recursive rule, this program is a **recursive aggregate program**. We define a recursive aggregate program as follows.

$$\begin{aligned} R(k_1,\ldots,k_n,g(y_0,y_1,\ldots,y_l)) :& -T_1(k_1,\ldots,k_n,y_1);\ldots;\\ & :-T_m(k_1,\ldots,k_n,y_m);\\ & :-R(k_1,\ldots,k_n,x),\\ & \quad P_1(k_1,\ldots,k_n,c_1),\ldots,\\ & \quad P_l(k_1,\ldots,k_n,c_l),\\ & \quad y_0 = f(x,c_1,\ldots,c_m). \end{aligned} \tag{1}$$

The predicate on the left side of ':-' is the *rule head*, where predicate R has an aggregate function $g(\cdot)$ with input variables y and zero or several group-by arguments $k_1,\ldots,k_n$ for $g(\cdot)$. On the right side of ':-', there are multiple rule bodies T separated by semicolons to provide aggregation inputs y. Each T may include multiple predicates P separated by commas that provide parameters to compute y, e.g., $y_0 = f(x,c_1,\ldots,c_m)$. In the rule bodies, there is one and only one predicate with the same name as the head predicate R, which makes it be a recursive aggregate rule. In this paper, we only allow one predicate to have R in the rule bodies[2].

2.2 Naive and Semi-Naive Evaluations

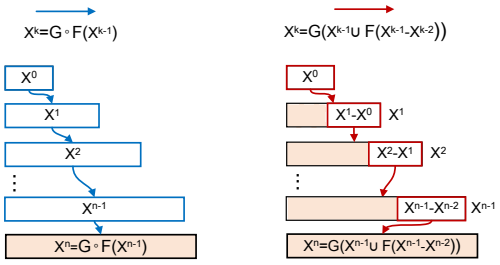
Two major evaluation techniques exist in Datalog systems.

Naive Evaluation:. Given an input set X^0, an aggregate G, and the rest of rules F, naive evaluation performs:

$$\begin{aligned} Y^{k-1} &= F(X^{k-1})\\ X^{k} &= G(Y^{k-1}). \end{aligned} \tag{2}$$

Usually, F is applied on a set of records, and G is a group-by aggregation to be applied on a set of key-specified records, i.e., X is a set and Y is a multiset. Let $(G\circ F)^k$ denote k applications of $(G\circ F)$. If the recursive program terminates after n iterations, its result is denoted as $(G\circ F)^n(X^0)$. We call the evaluation terminates at a fixpoint [1], i.e., F does not produce new facts after n iterations and we have $Y^n = Y^{n+1}$. There also exist some algorithms (e.g., PageRank) that have a mathematical limit. In order to guarantee their termination, we extend the syntax of Datalog for programmers to customize termination criteria at the user level. That is, the program will terminate after a number of iterations when the difference between the results of two consecutive aggregations is sufficiently small, i.e., $|\Delta X^{n+1}| = |X^{n+1} - X^n| < \epsilon$, where ϵ is user specified. Furthermore, we also define a termination number of iterations at the system level to stop the iterative processing after the iteration limit. With this two-level termination criteria, the iterative processing of a

[2]We focus on the direct recursion and linear Datalog programs in this paper, and leave the mutual recursion and the non-linear cases in a future work.



(a) Naïve Evaluation (b) Semi-Naïve Evaluation

Figure 3: Naive vs. Semi-Naive Evaluation.

given Datalog program starts from an initial value, and each step of the processing is derived from the previous step and is hoped to converge toward to a fixpoint step by step.

Semi-Naive Evaluation. It avoids redundant computation by incrementally computing the existing results and can be formalized as follows.

$$\begin{aligned} X^k &= G(X^{k-1} \cup F(\Delta X^{k-1})) \\ \Delta X^k &= X^k - X^{k-1}, \end{aligned} \tag{3}$$

where '−' is set subtraction and $\Delta X^0 = X^0$. The difference between naive and semi-naive evaluation is shown in Figure 3. Figure 3 (a) depicts naive evaluation. In each iteration, X^k is fully computed for producing X^{k+1}. In semi-naive evaluation of Figure 3 (b), only the incremental result ΔX^k between every two iterations is computed by F. Existing systems [24, 29, 50, 59] show the effectiveness with semi-naive evaluation.

2.3 Monotonic/Non-Monotonic Programs

For a recursive aggregate program with aggregate operation G and non-aggregate operation F, during the execution (with naive or semi-naive evaluation), we have a series of $X^0, \ldots, X^n$, where X^k $(\forall k, 0 \leq k \leq n)$ is a set of key-value pairs $\{\langle K, V\rangle, \ldots\}$ with unique keys, which are group-by arguments $k_1, \ldots, k_n$ for $g(\cdot)$ in the definition of recursive aggregate program of Section 2.1. We define a partial order on $\{X^0, \ldots, X^n\}$: $X^i \sqsubseteq X^j$ if and only if $\forall \langle K, V\rangle \in X^i, \exists \langle K', V'\rangle \in X^j, \langle K, V\rangle \sqsubseteq \langle K', V'\rangle$, where $\langle K, V\rangle \sqsubseteq \langle K', V'\rangle$ if and only if $K = K', V \sqsubseteq V'$. With respect to the partial order, if $X^{k-1} \sqsubseteq X^k$ holds for any k, such a program defined by G and F is a **monotonic program**.

A monotonic program can be executed with semi-naive evaluation for high performance. However, checking on monotonic conditions in existing systems [24, 29, 49, 58, 65] are sophisticated and manually done by users. Myria [58] requires to check that aggregate G is defined on a finite set and commutative, associative, and bag-monotonic, and non-aggregate F is monotonic with respect to the partial order defined above and distributive. DeALS [50] introduces monotonic aggregates for some aggregates, e.g., `mmin` for `min` and `mmax` for `max`. In the implementation of a monotonic aggregate, DeALS keeps intermediate results in a built-in buffer and computes the output based on intermediate results and results produced in the current iteration. For example, for `min`, if the minimum of the produced values $[d]$ is smaller than the smallest value r in the buffer, it returns `min([`d`])`; otherwise, it returns r, i.e., `return min(`r`,min([`d`]))`. In this way, DeALS can make the value sequence of each key changed monotonically and then use semi-naive evaluation. However, programmers need to manually rewrite programs by using the monotonic aggregates and guarantee the correctness of the rewritten programs.

PageRank of `Program 2` is non-monotonic because its iteratively updated score on each vertex does not monotonically increase or decrease. However, it can be manually rewritten into a monotonic program [13, 68] when using the delta-based algorithm. Such convertible non-monotonic programs cannot be executed with semi-naive evaluation in existing systems. In this paper, we aim to extend the scope of semi-naive evaluation and asynchronous execution to include such convertible non-monotonic programs. We also aim to provide a condition verification tool to automatically check if a given program satisfy the conditions to use incremental and asynchronous evaluation. In the implementation, we adopt the built-in buffer method to build our execution engine. Besides `min`, for other aggregates, including `max`, `sum`, and `count`, we have their runtime semantics as `return max(`r`,max([`d`]))`, `return sum(`r`,sum([`d`]))`, and `return sum(`r`,count[`d`])`, where `r` are previously computed results.

3 ANALYTICAL FOUNDATION

In this section, we formally describe monotonic recursive aggregate (MRA) evaluation and discuss under what conditions MRA evaluation can be applied. Finally, we present our automatic condition verification mechanism.

3.1 MRA Evaluation

We formally describe MRA evaluation as follows.

$$\begin{aligned} \Delta X^k &= G \circ F'(\Delta X^{k-1}) \\ X^k &= G(X^{k-1} \cup \Delta X^k), \end{aligned} \tag{4}$$

where F' is a new non-aggregate operation different from F, and ΔX^1 is the execution start point that consists of initial key-value pairs. $G(X^{k-1} \cup \Delta X^k)$ is to apply the group-by aggregate operator on the previously computed result X^{k-1} and the newly computed delta result ΔX^k. Different from the recursions defined in Equations (2) and (3), Equation (4) directly computes the delta result ΔX^k based on the previously computed ΔX^{k-1}. The result after k recursions is

$$X^k = G(X^0 \cup \bigcup_{i=0}^{k-1} (G \circ F')^i(\Delta X^1)). \tag{5}$$

There are two types of terminations of MRA evaluation. Let S be a set of bags of kv-pairs, i.e., $\forall k, Y^k \in S$, where Y^k is a bag of kv-pairs as defined in Equation (2). If S is a finite set, there must be a finite n such that $Y^n = Y^{n+1}$ and Y^n is the fixpoint bag [1, 59]. Accordingly, we have the fixpoint X^n after the aggregation on Y^n. Program SSSP is such a case. In addition, some algorithms do not reach a fixpoint in a finite n, but they have a mathematical limit. For example, the computation of PageRank on a primitive stochastic matrix converges to the unique dominant eigenvector [25], which is the fixpoint of PageRank. This type of algorithms will terminate after a number of iterations when the difference between the results of two consecutive aggregations is sufficiently small, i.e., $|\Delta X^{n+1}| = |X^{n+1} - X^n| < \epsilon$, where ϵ is algorithm specific. To support this type of termination, we extend the Datalog syntax to allow users to specify the termination conditions (as shown in the programs of Section 6.1).

3.2 Conditions for MRA Evaluation

We propose the following theorem to guarantee the correctness of MRA evaluation for recursive aggregate programs.

Theorem 1. ***(Conditions for MRA Evaluation)*** *Given a recursive aggregate program with aggregate operation G and non-aggregate operation F starting from X^0, if F can be decomposed into a new F' and a constant set C, i.e., $G \circ F(X) = G(F'(X) \cup C)$, and F' and G have the following properties:*

- *1. $G(X \cup Y) = G(Y \cup X)$ and $G(X \cup Y) = G(G(X) \cup Y)$,*
- *2. $G \circ F' \circ G(X) = G \circ F'(X)$,*

MRA evaluation in Equation (4) can produce the same result as that of naive evaluation in Equation (2), where MRA evaluation is initialized as ΔX^1 such that $X^1 = G \circ F(X^0) = G(\Delta X^1 \cup X^0)$.

Proof. We use mathematical induction to prove it. In base case for $k = 1$, we have $X^1 = G(X^0 \cup \Delta X^1) = G \circ F(X^0)$ according to the definition of initial start of MRA evaluation. Assume that the following statement is true for k:

$$X^k = G \circ F(X^{k-1}) = G((G \circ F')^{k-1}(\Delta X^1) \cup X^{k-1}). \tag{6}$$

For $k + 1$, we have

$$\begin{aligned}
X^{k+1} &= G \circ F(X^k) = G(F'(X^k) \cup C) && \text{(6.a)}\\
&= G(F' \circ G((G \circ F')^{k-1}(\Delta X^1) \cup X^{k-1}) \cup C) && \text{(6.b)}\\
&= G(F'((G \circ F')^{k-1}(\Delta X^1) \cup X^{k-1}) \cup C) && \text{(6.c)}\\
&= G(F'(G \circ F')^{k-1}(\Delta X^1) \cup F'(X^{k-1}) \cup C) && \text{(6.d)}\\
&= G((G \circ F')^k(\Delta X^1) \cup G(F'(X^{k-1}) \cup C)) && \text{(6.e)}\\
&= G((G \circ F')^k(\Delta X^1) \cup X^k) && \text{(6.f)}
\end{aligned}$$

Line (6.a) is true because of the decomposability of $F(X)$, i.e., $G \circ F(X) = G(F'(X) \cup C)$. By substituting X^k with Equation (6), we have Line (6.b). Line (6.c) is true when applying Property 1, Property 2, and Property 1: $G(F' \circ G(X) \cup C) = G(G \circ F' \circ G(X) \cup C) = G(G \circ F'(X) \cup C) = G(F'(X) \cup C)$. Line (6.d) is true because of the distributive property of $F'(X)$, i.e., $F'(X1 \cup X2) = F'(X1) \cup F'(X2)$, which is the native property of non-aggregate operation. Line(6.e) is true because of Property 1. Line (6.f) is true because of the decomposability of $F(X)$ and Equation (2), i.e., $X^k = G \circ F(X^{k-1})$. Therefore, the statement is also true for $k + 1$. Proved. □

The insights into the properties can be explained as follows. First, Property 1 indicates that G is commutative and associative. The commutative property implies that we can change the order of the operands in the aggregation and the associative property allows us to aggregate "partial" results first after applying F'. Second, if Property 2 holds, we can convert the recursion with aggregation to a recursion without aggregation by repeatedly applying this property, e.g., $G \circ F' \circ G \circ F' \circ G(X)$ is equal to $G \circ F' \circ F'(X)$. In other words, we can move G out of the recursion of F'. A recursive program without aggregation and negation ($G \circ F' \circ F'(X)$) can be executed with semi-naive evaluation under certain conditions [1], so that its equivalent program ($G \circ F' \circ G \circ F' \circ G(X)$) can also be executed with semi-naive evaluation[3]. The properties also guarantee the correctness of async execution. After G is out of the recursion, each F' in the recursive sequence can be asynchronously executed as only G needs to synchronize the input. We will prove it in Section 4. A set of monotonic and non-monotonic programs satisfy the properties. We show two examples here and more others in Section 6.1.

Single Source Shortest Path. As shown in `Program 1`, SSSP has the non-aggregate operation F that is applied on vertex x to propagate the candidate shortest distance $d_x + d_{xy}$ to its neighbor y, where d_x is the current shortest distance from source s to x and d_{xy} is the weight of the edge between x and y. The aggregate operation G groups all the distances by destination vertex ID and performs `min` aggregation to update the shortest distance to vertex y. First, Property 1 holds for `min` because of $min(X, Y) = min(Y, X)$ and $min(X, Y) = min(min(X), Y)$. Second, F does not contain a constant part and F' is F; and G and F' satisfy Property 2, as the shortest distances are the same in the following two cases: (1) making expansion and then making aggregation ($G \circ F'(X)$), i.e., $min_y(d_x + d_{xy})$; and (2) making aggregation, then making expansion, and making aggregation again ($G \circ F' \circ G(X)$), i.e., $min_y(min_x(d_x) + d_{xy})$. We will discuss how to set the initial ΔX^1 in Section 3.3.

PageRank. As shown in `Program 2`, each vertex computes and sends the partial ranking score ($r_y = 0.85 \cdot r_x/d$) to its neighbours and also sends a constant value 0.15 to itself. G groups all scores by destination vertex ID y and performs `sum`

[3]When the monotonic G is moved out of the recursion of F', we need Property 1 to guarantee the correctness of aggregation results.

```
(declare-const d Int)
(define-fun g ( (a Real) (b Real) ) Real
  ( + a b ) )
(define-fun f ( (a Real) ) Real
  (/ (* a 0.85) d ) )
(assert (> d 0))
(assert (
    not (forall ( (x1 Real) (y1 Real) (x2 Real) (y2 Real))
        (= (g (f (g x1 y1) ) (f (g x2 y2) ) )
           (g (g (g (f x1)  (f y1) ) (f x2) ) (f y2) ) ) )
) )
(check-sat)
```

Figure 4: The Z3 code for automatic verification of $G \circ F' \circ G(X) = G \circ F'(X)$ for PageRank.

aggregation. First, sum satisfies Property 1 as $sum(X, Y) = sum(Y, X)$ and $sum(X, Y) = sum(sum(X), Y)$. Second, F can be decomposed into F' that is "$0.85 \cdot r_x/d$" and a constant part C that is "0.15". Assuming vertex x has two values r_{x1} and r_{x2} before sending to y, r_{x1} and r_{x2} can be used to calculate the partial ranking scores first, i.e., $0.85 \cdot r_{x1}/d$ and $0.85 \cdot r_{x2}/d$, or be aggregated first before calculating the partial ranking score, i.e., $0.85 \cdot sum(r_{x1}, r_{x2})/d$. On the receiver y where sum is applied, these two cases have the same result, i.e., $sum(0.85 \cdot r_{x1}/d, 0.85 \cdot r_{x2}/d) = sum(0.85 \cdot sum(r_{x1}, r_{x2})/d)$. This property also holds for variable numbers of input values.

Notice that although the original PageRank of Program 2 is non-monotonic, we can execute it incrementally with MRA evaluation because the MRA conditions are satisfied. Our system can convert it to its equivalent incremental program [13, 68] automatically and transparently to users. For ease of understanding, we show the equivalent in Program 2.b, where the ranking score of each vertex is monotonically increasing since it is the aggregation result of its previous score (i.e., $r_y = r$ in r3) and the scores from its neighbours.

Program 2.b PageRank (Incremental)

```
r1. degree(X,count[Y]) :- edge(X,Y).
r2. rank(0,X,r)        :- node(X),r = 0.15.
r3. rank(i+1,Y,sum[r_y]) :- rank(i,Y,r),r_y = r;
                        :- rank(i,X,r_x),
                        edge(X,Y),degree(X,d),
                        r_y = 0.85 · r_x/d.
```

3.3 Automating Condition Verification

It is cumbersome and error-prone to manually check the MRA conditions of Theorem 1. To turn our theory into reality, we design and implement an automatic condition verification tool with the help of Satisfiability Modulo Theories (SMT) solver Z3 [63]. We automatically extract G and F' (in Section 5.1) and address the following two problems.

Verifying Properties of G and F' with Z3. We use SMT solver Z3 to automatically verify the properties of G and F'. An SMT instance is a formula where some functions and constants are defined with constraints. SMT can determine if such a formula is satisfiable, i.e., if there is an assignment of proper values to its uninterpreted function and constant symbols to make the formula to be true. Z3 asserts a formula and may return "satisfiable", "non-satisfiable", or "unknown". Notice that an SMT solver cannot judge "whether a formula H is always true?" but only answers "whether a formula H is satisfiable?". To verify a property that should be always true, we use double negation to convert "H is always true" into "NOT H is not satisfiable". If H is always true, "NOT H" is always false and cannot have any satisfiable assignment, and the assertion will return "unsat". We provide the Z3 code template to verify the properties of G and F'. The algorithm-specific G and F' will be automatically filled into the template. The verification is automatically done. Figure 4 shows the Z3 code for PageRank that can automatically verify the formula $G \circ F' \circ G(X) = G \circ F'(X)$. If "NOT" $G \circ F' \circ G(X) = G \circ F'(X)$ returns "unsat", $G \circ F' \circ G(X) = G \circ F'(X)$ is always true. We skip how to verify Property 1 due to the page limit.

Determining Initial Values. It is necessary to automatically determine the initial value ΔX^1 to satisfy $X^1 = G(\Delta X^1 \cup X^0)$. We need to find the inverse operation G^- such that $\Delta X^1 = G^-(X^1, X^0)$ to compute ΔX^1. We have predefined G^- for the typical aggregate operations. For example, when G is min, G^- is still min; while, when G is sum, G^- is the pair-wise subtraction. With G^-, we can obtain ΔX^1 from X^1 and X^0 by applying G^-. For SSSP, we get $\Delta X^1 = X^1$, because $\Delta X^1 = min(X^1, X^0) = d_{sx} = min(d_{sx}) = min(d_s + d_{sx}) = G \circ F(X^0) = X^1$, where s is the source and d_s is 0. The formula indicates that the shortest distances from source s to vertices X at the first recursion (i.e., X^1) are the weights of edges from s to each x (i.e., ΔX^1). In practice, the initialization can be enforced after calculating X^1 with rule r2. This process is handled by our system automatically. Section 5.1 will introduce more details of the automation.

4 CORRECTNESS OF ASYNC EXECUTION

In a distributed computing environment, input kv-pairs of aggregate operation G and non-aggregate operation F' are partitioned across multiple workers. Due to the existence of G, in each iteration, the whole result of applying F' must be ready before G can be applied in synchronous (sync) execution. This forms a strict operation sequence of $G \circ F'$. We can formalize it with the bulk-synchronous parallel (BSP) model [55], where F' can be executed independently on kv-pairs and G is the synchronous point. Equation (5) shows the result of MRA evaluation in sync execution. We denote ΔY^1 as the result of applying F' on ΔX^1, i.e., $\Delta Y^1 = F'(\Delta X^1)$; and we rewrite Equation (5) with $(n+1)$ iterations and have the result of **Sync MRA Evaluation** as follows:

$$R^n_{syn} = G\Big(X^0 \cup \Delta X^1 \cup \bigcup_{i=0}^{n-1} (G \circ F')^i (G(\Delta Y^1))\Big). \tag{7}$$

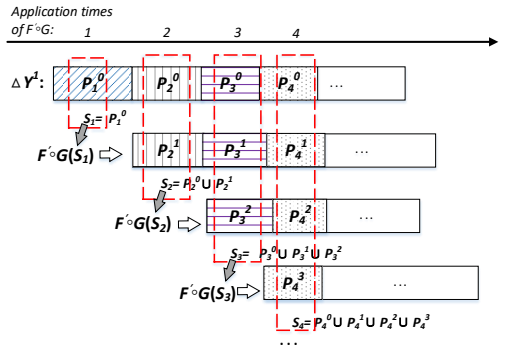


Figure 5: An example of async execution. $\Delta Y^1 = F'(\Delta X^1)$ is the start point of async execution. A horizontal box $F' \circ G(S_t)$ represents the output of the tth-time applying $F' \circ G$, and $F' \circ G(S_t)$ is partitioned into multiple disjoint subsets P that will be used in the following $F' \circ G$. A vertical box S_t represents the input of the tth-time applying $F' \circ G$, and S_t is a union of subsets P from previous results of applying $F' \circ G$, i.e., $S_t = P_t^0 \cup P_t^1 \cup \ldots \cup P_t^{t-1}$.

In contrast, asynchronous (async) execution allows a part of results of applying F' to be aggregated and processed earlier than others within an iteration or even across iterations. Translating a recursive aggregate program into a query plan for async execution has been studied [59]. However, the correctness of results is not guaranteed. Figure 5 shows an example of async execution. It starts from ΔY^1 and performs $F' \circ G$ iteratively. When applying $F' \circ G$ in the first time, the input is denoted as S_1 and the output is denoted as $F' \circ G(S_1)$. In async execution, S_1 is P_1^0 that means the input of the first time applying $F' \circ G$ is a subset of applying $F' \circ G$ zero time, i.e., a subset of the initial ΔY^1. Similarly, the subsequent $F' \circ G$ can take a union of the subsets from one or more previous steps as the input. We formalize async execution as follows.

Definition 2. **(Async MRA Evaluation)** From a start ΔY^1, where $\Delta Y^1 = F'(\Delta X^1)$ and ΔX^1 is the start point of MRA evaluation, we define the result of async MRA evaluation after the mth-time applying $F' \circ G$ as:

$$R_{asy}^m = G\Big(X^0 \cup \Delta X^1 \cup \Delta Y^1 \cup F' \circ G(S_1) \cup \ldots F' \circ G(S_m)\Big), \quad (8)$$

where

$$S_{t+1} = \bigcup_{i=0}^{t} P_{t+1}^i$$
$$\bigcup_{i=t+1}^{m} P_i^t = \begin{cases} F' \circ G(S_t) & t > 0 \\ \Delta Y^1 & t = 0. \end{cases} \quad (9)$$

In the definition, S_{t+1} is the input of the $(t+1)$th-time applying $F' \circ G$, and P_{t+1}^i is a subset of output from the ith-time applying $F' \circ G$ and is used as a part of S_{t+1}. This means in async execution, the input of applying the $(t+1)$th-time $F' \circ G$ includes the output from applying the 0th-time to the tth-time $F' \circ G$. Equation (9) also shows that for $t > 0$ there must be an m, such that $F' \circ G(S_t) = \bigcup_{i=t+1}^{m} P_i^t$. It means that all results of the tth-time applying $F' \circ G$ are used up at the mth-time applying $F' \circ G$; and for $t = 0$, data used up at the mth-time applying $F' \circ G$ is from ΔY^1. For any t and i, where $i > t$, P_i^t can be an empty set, i.e., applying $F' \circ G$ at the ith step does not need the output of the tth step.

Theorem 3. ***((Correctness of Async MRA Evaluation)*** *For a recursive program with F' and G and the initial start ΔX^1 and ΔY^1, where $\Delta Y^1 = F'(\Delta X^1)$, if sync MRA evaluation of the program reaches a fixpoint in a finite n and async MRA evaluation reaches the same fixpoint in an m, R_{syn}^n in Equation (7) is equal to R_{asy}^m in Equation (8), when the properties of Theorem 1 hold.*

Proof. In order to simplify the discussion, we move $X^0 \cup \Delta X^1$ out of both Equations (7) and (8). In async execution starting from ΔY^1, after m times applying $F' \circ G$, where m is described in Definition 2, we have:

$$R_{asy}^m = G\Big(\Delta Y^1 \cup F' \circ G(S_1) \cup \ldots F' \circ G(S_m)\Big) \quad (10)$$
$$= G\big(\Delta Y^1 \cup F'(S_1) \cup F'(S_2) \cup \ldots F'(S_m)\big) \quad (10.a)$$
$$= G\big(\Delta Y^1 \cup F'(\cup_{k=0}^{0} S_1^k) \cup F'(\cup_{k=0}^{1} S_2^k) \cup \ldots F'(\cup_{k=0}^{m-1} S_m^k)\big) \quad (10.b)$$
$$= G\big(\Delta Y^1 \cup F'(\cup_{t=1}^{m} S_t^0) \cup F'(\cup_{t=2}^{m} S_t^1) \cup \ldots F'(\cup_{t=m}^{m} S_t^{m-1})\big) \quad (10.c)$$
$$= G\big(\Delta Y^1 \cup \bigcup_{i=1}^{n-1} F'^i(\Delta Y^1) \cup S_{RES}\big) \quad (10.d)$$
$$= G\big(G(\Delta Y^1 \cup \bigcup_{i=1}^{n-1} (G \circ F')^i(G(\Delta Y^1))) \cup S_{RES}\big) \quad (10.e)$$
$$= G\big(R_{syn}^n \cup S_{RES}\big) = R_{syn}^n \quad (10.f)$$

To get Line (10.a), we first apply Property (1) on Line (10) and get $G(\Delta Y^1 \cup G \circ F' \circ G(S_1) \cup \ldots G \circ F' \circ G(S_m))$, then apply Property (2) to get $G(\Delta Y^1 \cup G \circ F'(S_1) \cup \ldots G \circ F'(S_m))$, and apply Property (1) again to remove G inside the braces and get Line (10.a). This step shows that the properties can help to transform the recursion of $F' \circ G$ into the recursion of F' (both having a G after the recursion). On the recursion of F', we prove that async execution can get the same result as that of sync execution by regrouping results of async execution on which F' has been applied the same times.

To get Line (10.b), we first introduce a symbol $P_{t+1}^{k,i}$ to decompose P_{t+1}^i of Equation (9), i.e., $P_{t+1}^i = \bigcup_{k=0}^{i} P_{t+1}^{k,i}$. This means for each k, $P_{t+1}^{k,i}$ is a subset of P_{t+1}^i on which F' has been applied k times. For $i > 0$, $P_{t+1}^{0,i}$ is $\emptyset$ as F' has been applied at least once. We also introduce a symbol S_{t+1}^k to denote the union of the subsets of S_{t+1} where F' has been applied exactly k times, and we have $S_{t+1}^k = \bigcup_{i=k}^{t} P_{t+1}^{k,i}$. As F' can be applied on input kv-pairs independently, from

Equation (9) and for the recursion of F', we have:

$$S_{t+1} = \bigcup_{i=0}^{t} \left(\bigcup_{k=0}^{i} P_{t+1}^{k,i} \right) = \bigcup_{k=0}^{t} \left(\bigcup_{i=k}^{t} P_{t+1}^{k,i} \right) = \bigcup_{k=0}^{t} S_{t+1}^{k} \quad (11)$$

And then from Equation (11), we have Line (10.b).

Line (10.c) is true: because F' is distributive on Union, we can unfold unions S_t^k having the same t inside each F' at Line (10.b), and regroup to unions S_t^k that have the same k. Then, we get Line (10.c).

To get Line (10.d), we first denote $\cup_{t=k+1}^{m} S_t^k$ to be the union of the input subsets of applying the $(k+1)$th-time to the mth-time F' and on these subsets F' has been applied exactly k times. For any k ($k < m$), we have:

$$F'^{k}(\Delta Y^1) = \bigcup_{t=k+1}^{m} S_t^k = \bigcup_{t=k+1}^{m} \bigcup_{i=k}^{t-1} P_t^{k,i}. \quad (12)$$

The rationale of Equation (12) is as follows. Based on the definition in Equation (9), there exists an m such that ΔY^1 will be used up at the mth-time applying F'. As ΔY^1 is an intermediate result of sync execution, i.e., $\Delta Y^1 = F'(\Delta X^1)$, we use ΔY^1 to bridge the results of sync and async execution: the union of data on which F' has been applied k times in async execution is the result of sync execution after applying F' on ΔY^1 k times. In async execution, such data is from different subsets $P_t^{k,i}$. We use mathematical induction to prove Equation (12). For $k = 0$, we have:

$$\bigcup_{t=1}^{m} \bigcup_{i=0}^{t-1} P_t^{0,i} = \bigcup_{t=1}^{m} P_t^{0,0} \cup \bigcup_{t=1}^{m} \bigcup_{i=1}^{t-1} P_t^{0,i} = \bigcup_{t=1}^{m} P_t^{0,0} \cup \emptyset \quad (13)$$

$$= \bigcup_{t=1}^{m} P_t^0 = \Delta Y^1 \quad (13.a)$$

Line (13) is true, because of the definition that $P_t^{k,i}$ is $\emptyset$ when $k = 0$ and $i > 0$. Line (13.a) is true because of Equation (9).

For $k > 1$, if there exists an m, such that $F'^{k}(\Delta Y^1) = \bigcup_{t=k+1}^{m} \bigcup_{i=k}^{t-1} P_t^{k,i}$. For $k+1$, we apply F' on both side of Equation (12) and have $F'^{k+1}(\Delta Y^1) = F'(\bigcup_{t=k+1}^{m} \bigcup_{i=k}^{t-1} P_t^{k,i})$. The right side $F'(\bigcup_{t=k+1}^{m} \bigcup_{i=k}^{t-1} P_t^{k,i})$ is to apply one more F' on all data that has been applied k times F'. Based on Definition 2, there must exist an m' such that all data applied $k+1$ times F' is used up. That means the data generated by the $(k+1)$th-time F' is the union of the subsets of the input from applying the $(k+2)$th-time to the (m')th-time F'. Then we have $F'^{k+1}(\Delta Y^1) = F'(\bigcup_{t=k+1}^{m} \bigcup_{i=k}^{t-1} P_t^{k,i}) = \cup_{t=k+2}^{m'} \cup_{i=k+1}^{t-1} P_t^{k+1,i}$. Equation (12) holds for $k+1$.

Since m is not necessarily equal to n (usually m of async execution is larger than n of sync execution), we denote S_{RES} as the data on which F' is applied more than $n-1$ times. Then, we can get Line (10.d) by applying Equation (12) to Line (10.c) with S_{RES}.

Line (10.e) is true when applying $G \circ F' \circ G(X) = G \circ F'(X)$ and $G(X \cup Y) = G(G(X) \cup Y)$ on Line (10.d).

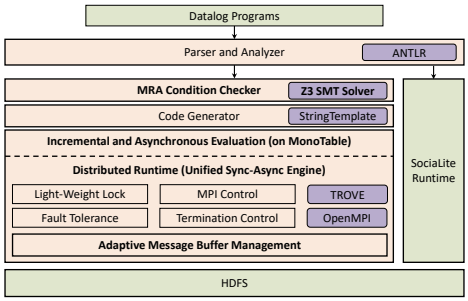


Figure 6: System overview of PowerLog.

Line (10.f) is true. Because of the fixpoint at n of sync execution, we have $R_{syn}^n = R_{syn}^m = G(R_{syn}^n \bigcup \cup_{k=n}^{m}(F^k(\Delta Y^1)))$ for any $m \geq n$. As in async execution, $S_{RES} \subseteq \cup_{k=n}^{m}(F^k(\Delta Y^1))$. Thus, based on the definition of monotonic in Section 2.3, we have $R_{syn}^n \sqsubseteq G(R_{syn}^n \cup S_{RES}) \sqsubseteq G(R_{syn}^n \bigcup \cup_{k=n}^{m}(F^k(\Delta Y^1)))$. That is $R_{syn}^n \sqsubseteq R_{asy}^m \sqsubseteq R_{syn}^m$. Thus, $R_{syn}^n = R_{asy}^m$. Proved. □

Theorem 3 can be extended to the case if sync MRA evaluation does not reach a fixpoint in a finite n but has a mathematical limit. Similar to the proof of Theorem 3, when async MRA evaluation reaches the same limit, we have $R_{syn}^{\infty} = G(R_{syn}^{\infty} \cup S_{RES}) = R_{asy}^{\infty}$. We also point out that sync execution is a special case of async execution with the restriction $S_1 = \Delta Y^1$ and $\forall t \geq 1, S_{t+1} = F' \circ G(S_t)$. That means the whole S_t is aggregated together by the tth-time applying G.

5 THE POWERLOG SYSTEM

Figure 6 shows the structure of PowerLog that is developed based on SociaLite open-source software [24] and has the following major modules.

5.1 Automatic Condition Checker

PowerLog has a parser to convert a Datalog program into an abstract syntax tree (AST) by using ANTLR [3], and uses an analyzer to traverse the AST to perform syntactic and semantic analysis, to identify the recursive rule, and to extract the aggregate operation G, the non-aggregate operation F', and the constant set C. As shown in Section 2.1, the rule head of a recursive program includes G, and each rule body logically contains multiple table predicates for join operation and multiple expression predicates that could include F' and C. In PageRank of `Program 2`, F contains the table predicates "`rank(i,X,`r_x`),edge(X,Y),degree(X,`d`)`", the expression predicate "$r_y = 0.85 \cdot r_x/d$" that includes F', and the constant predicate "$r_y = 0.15$" that includes C. By analyzing the rules, PowerLog can identify a recursive aggregate program and then extract G, F' and C.

PowerLog takes two steps to check if a recursive aggregate program satisfies the MRA conditions. Each step checks one property of Theorem 1. In PowerLog, we have defined five common aggregate operators `min`, `max`, `sum`, `count`, and `mean`

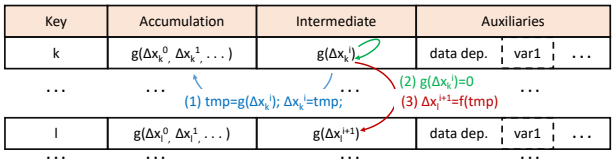


Figure 7: MonoTable data structure and its update.

in Z3 SMT solver. The first four operators are commutative and associative, while the last one `mean` does not. The details of how to define an operator in Z3 are referred to [64].

PowerLog then checks if G and F' can satisfy Property 2 $G \circ F' \circ G(X) = G \circ F'(X)$. In Z3, we define f as a single-input-single-output function and define g, i.e., `min`, `max`, `sum`, and `count`, as a double-input-single-output function (binary operator). Because g can naturally take any number of inputs, in order to generalize the expression of g with variable inputs, we use the binary aggregate operators in Z3 code. We have $g(x_0, x_1, x_2, \ldots, x_{n-1}) = g(g(\ldots g(g(x_0, x_1), x_2)\ldots), x_{n-1})$, as the associative property $G(X \cup Y) = G(G(X) \cup Y)$ holds.

As shown in Figure 4 of Section 3.3, in the Z3 code, we express $G \circ F' \circ G(X)$ with g and f as $g(f(g(x_1, y_1)), f(g(x_2, y_2)))$ and express $G \circ F'(X)$ as $g(g(g(f(x_1), f(y_1)), f(x_2)), f(y_2))$, where x and y are "for all" real numbers. We let Z3 check if $G \circ F' \circ G(X)$ is equal to $G \circ F'(X)$ by using the Z3 assertion with double negation. If Z3 returns "unsat" for "NOT" $G \circ F' \circ G(X) = G \circ F'(X)$, $G \circ F' \circ G(X)$ is always equal to $G \circ F'(X)$. Notice that g and f are automatically extracted by PowerLog and used in the Z3 assertion. If these properties hold, PowerLog executes the program with MRA evaluation.

5.2 Implementation of MRA Evaluation

PowerLog implements MRA evaluation by manipulating a distributed mutable in-memory table, called *MonoTable*. The table maintains the states of the recursive computation.

Each tuple of the table includes an accumulated aggregation result x and a delta result Δx. Variable x_i is accumulated by $x_i = g(x_i, \Delta x_i)$, and Δx_i is computed by $\Delta x_i = g(f(\Delta x_{j1}), f(\Delta x_{j2}), \ldots)$, where $(\Delta x_{j1}, \Delta x_{j2}, \ldots)$ are delta results of other tuples dependent with tuple i. Figure 7 shows the design of MonoTable. The `Key` column indexes the table. The `Accumulation` column maintains the result x. The `Intermediate` column stores the intermediate aggregated delta results, i.e., $g(\Delta x_k)$. The `Auxiliaries` columns store the joined results of non-recursive predicates in the recursive rule body and other constant values of each tuple. The table is initialized with the method described in Section 3.3.

There are three steps to manipulate MonoTable as shown in Figure 7. First, an intermediate entry $g(\Delta x_k)$ is fetched into a local variable, i.e., variable `tmp` in the figure, and then the local variable is used to update the final result x in the accumulation entry at the same row. The superscript of Δx_k indicates the time point when Δx_k is generated. Second, the intermediate entry is reset to the identity element so that a delta result will not be aggregated multiple times. Third, the fetched value in variable `tmp` is computed by f and the result is used to update intermediate entries at other rows. As this step has cross-row operations, communication between workers may be needed. Notice that there are concurrent read and write on an entry, the atomic operations are needed: the first and second steps need an atomic exchange and the third step needs an atomic implementation of the aggregation.

5.3 Unified Sync-Async Engine

The basis of our method comes from two insights into asynchronous processing in a distributed environment. First, the frequency of async message passing determines the level of asynchronization for each worker. More frequent message passing activities lead more data that are frequent updates among distributed nodes, which raises the level of async computing. A high level async computing may help the computation to reach the convergence at almost zero coordination cost. In contrast, less frequent message passing activities can accumulate data updates in high granularity, which reflect a reduced level of asynchronization. An extreme case is the maximum reduced level of asynchronization that is equivalent to sync computing: no workers pass updated data until each of them finishes its computation. Another phenomenon in async computing is called "stale synchronous parallel (SSP)" [18], where distributed workers are allowed to read stale or non-updated data. SSP may lower the quality of async computing by increasing the amount of computing but delaying the convergence. Adjusting the frequency of async message passing activities is the key to adaptively establish a "properly-controlled" mechanism to minimize the synchronization overhead and to minimize the access to the stale data in order to achieve optimal performance.

Second, all the programs in PowerLog can be correctly processed in async mode with our automatic checking mechanism. In addition, in a distributed environment, synchronization overhead is the most expensive, compared with other types of overheads. Considering these two facts, our method is in a framework of async computing.

Meanwhile, sync execution is still necessary, especially to check the termination condition. Workers in async computing do not have any global information about the computation progress and determine the time to stop. For algorithms where the global termination check is demonstrated to be more efficient, e.g., PageRank, sync execution is needed in the execution engine to collect the local computation states and to make a global termination decision.

Based on the above considerations, we design and implement a unified sync-async engine as shown in Figure 8. The

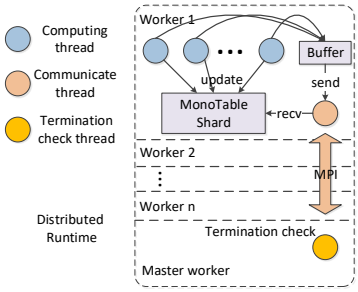


Figure 8: Structure of the sync-async engine.

engine contains a number of workers and one master. Each worker includes several compute threads to update the local shard of MonoTable and has a dedicated thread for the communication among workers. The master globally and periodically checks the termination condition.

The key of the engine is a method to adaptively adjust the frequency of message passing in each worker independently. Specifically, we adaptively adjust the message size and the time interval between two consequent message-passing activities for the following reasons. A larger message size and higher interval time, the lower frequency will be, and so is the chance of stale processing in each node. For a slow updating worker, message passing size should also be reduced accordingly for quality control. We will show that the adaptive method is effective and overhead free because no global information is collected.

Each worker node in an N-node cluster maintains $(N-1)$ message buffers for the rest of the workers. Initially, a message-passing size $\beta(i,j)$ and a message-passing interval τ are predefined for all worker nodes. During the execution, each worker makes its own adaptive adjustment based on the following principle. If the updates (denoted as $|B(i,j)|$) are accumulated in a fast pace (in the time window ΔT), the message size $\beta(i,j)$ is enlarged; and if the updates are in a slow pace, the message size $\beta(i,j)$ is reduced. Specifically, when $|B(i,j)|/\Delta T > r \cdot \beta(i,j)/\tau$ in the fast pace or $|B(i,j)|/\Delta T < \frac{1}{r} \cdot \beta(i,j)/\tau$ in the slow pace, $\beta(i,j)$ is updated to be $\beta(i,j) = \alpha \cdot \tau \cdot |B(i,j)|/\Delta T$, where α is a damping factor fixed to 0.8 and r is a configurable parameter set to 2 in our experiments. From the perspective of system management, this is a tradeoff between batch and stream processing.

5.4 Other Optimizations

PowerLog uses TROVE [54] for container operations, OpenMPI [38] for message passing, ProtoStuff [41] for serialization and deserialization, and HDFS to store data and checkpoint intermediates. Here we introduce the termination check and the optimizations for MRA evaluation as follows.

Termination Check. In MRA evaluation, the results of a program are monotonically increasing or decreasing. MRA evaluation is terminated when either reaching a fixpoint or the changes of consecutive results are sufficiently small. A dedicated thread on each worker evaluates the aggregation result of accumulation column entries of MonoTable. These threads report local aggregation results to the master that will merge local results and determine whether to stop the evaluation by checking the difference between two consecutive global aggregation results. Notice that async execution does not have a central coordination mechanism, thus in our unified sync-async engine, we check the termination condition periodically.

Optimizations for MRA Evaluation. We have optimized MRA evaluation for the `sum` aggregation. In `sum`, the number of delta results produced in each iteration is fixed; and delta results to be accumulated become smaller over iterations. We observe that the effect of delta results on performance is distinct. Those with larger values are more important for the convergence [67]. Therefore, we make the following optimization efforts. First, delta results are distinguished. Second, the important ones, whose values are larger than a configurable threshold, are accumulated to the final results and their computed values after applying f are sent to neighbors immediately. Third, the less important delta results are contained and accumulated in the local cache before they are used. These optimizations can reduce the number of communications between workers and also reduce the amount of computations, e.g., the non-aggregate operation.

6 EVALUATION

6.1 MRA Evaluation Satisfiable Programs

Some non-monotonic programs can satisfy the MRA conditions and be executed with MRA evaluation, without changing their semantics, e.g., PageRank. There also exist core non-monotonic programs that cannot pass the MRA condition check, e.g., GCN-Forward [22], and evaluating these programs with MRA evaluation will lead to incorrect results. We investigate more widely-used programs in this section. Besides SSSP and PageRank, we have found ten recursive aggregate programs that can pass the MRA condition check of PowerLog but other two that cannot. Table 1 summarizes these programs, where "`MRA sat.`" indicates if a program passes the check.

Connected Component (CC) is to find the connected components in a graph. Its Datalog expression in `Program 3` is based on the label propagation approach [42], where rule `r1` initializes the component of a vertex with its own vertex ID and rule `r2` recursively updates the component of each vertex with the minimum ID propagated by its in-neighbors. In this program, the aggregate operation `min` is commutative and associative (Property 1). F is an identity function and does not contain a constant part (F' is F). Thus, G and F' hold Property 2, as SSSP algorithm.

Table 1: Twelve recursive aggregate programs that can be executed with MRA evaluation in PowerLog.

Program	MRA sat.	Aggregator	Program	MRA sat.	Aggregator
SSSP [24]	yes	min	**PageRank** [39]	yes	sum
CC [24]	yes	min	**Adsorption** [7]	yes	sum
Katz metric [21]	yes	sum	**Belief Propagation** [40]	yes	sum
Computing Paths in DAG [50]	yes	count	Cost [50]	yes	sum
Viterbi Algorithm [50]	yes	max	SimRank [20]	yes	sum
Lowest Common Ancestor [44]	yes	min	APSP [50]	yes	min
CommNet [52]	no	sum	GCN-Forward [22]	no	sum

Program 3. Connected Components

```
r1. cc(X,X)       :- edge(X,_).
r2. cc(Y,min[v]):- cc(X,v),edge(X,Y);
```

Adsorption is a graph label propagation algorithm. The algorithm shown in Program 4 is based on the Markov Process [7]. Rule r1 gives the initial label distribution I for each vertex. In rule r2, L reserves the intermediate label distribution in recursion. The label distribution consists of two parts. First, each vertex computes and propagates its current distribution $a1 = 0.7 * a * w * p$ to its out-neighbours as their partial distributions. Second, each vertex sends the initial weighted distribution $a1 = i * p2$ to itself. The aggregation *sum* aggregates these two parts as the new distribution. Other symbols can be treated as constant variables, e.g., p_i and p_c are two constant weight tables associated with vertices.

As shown in Program 4, we extend Datalog syntax to include a termination condition in the recursive rule body with a pair of braces, i.e., $\{sum[\Delta a] < 0.001\}$. This condition terminates the execution of Adsorption when the difference of accumulated results in a time interval is smaller than 0.001. For different algorithms, users can specify different conditions. For example, one can add $\{sum[\Delta r_x] < 0.001\}$ to Program 2 to specify the terminate condition for PageRank.

Adsorption satisfies the MRA conditions. First, sum satisfies Property 1. Second, F can be decomposed into F' as $0.7 * a * w * p$ and the constant part $i * p2$. Considering vertex x has two values a_{x1} and a_{x2} before sending to its neighbour y, no matter computing the distributions first or aggregating them first, these two cases have the same result when G is on the receiver y. i.e., $sum(0.7 * a_{x1} * w * p, 0.7 * a_{x2} * w * p) = sum(0.7 * sum(a_{x1}, a_{x2}) * w * p)$. Property 2 also holds.

Program 4. Adsorption

```
r1. I(x,i)              :- node(x), i=1.
r2. L(0,x,l)            :- node(x), l=0.
r3. L(j+1,y,sum[a1])):-I(y,i),p_i(y,p2),a1 = i * p2;
                       L(j,x,a),A(x,y,w),p_c(x,p),
                       a1 = 0.7 * a * w * p;
                       {sum[Δa] < 0.001}.
```

Katz Metric is a proximity measure in a network [21]. Rule r1 of Program 5 sets the source and its initial metric score. In rule $r2$, the metric score of a vertex is computed by iteratively aggregating the metric scores from its in-neighbors. In this program, G is *sum*, F' is $0.1 * k$, and C is j. The MRA conditions are satisfied, as sum satisfies Property 1 and G and F' satisfy Property 2 due to the similar reason to that for Adsorption.

Program 5. Katz Metric

```
r1. I(X,k)              :- X=0,k = 10000.
r2. K(i+1,y,sum[k1]) :- I(y,j),k1 = j;
                        K(i,x,k),edge(x,y),k1 = 0.1 * k;
                        {sum[Δk] < 0.001}.
```

Belief Propagation (BP) [23] is a message-passing algorithm that performs inference on graphical models. In Program 6, variable B is a vector including the to-be-returned beliefs. Variable E is an input weighted network with vector I for initial beliefs. Variable H is a vector of coupling scores. In this program, F does not have a constant part. F' is $0.8 * w * b * h$ and G operation is *sum*. The MRA conditions are satisfied in BP as Adsorption.

Program 6. Belief Propagation

```
r1. B(0,v,c,b)          :- I(v,c,b).
r2. B(j+1,t,c2,sum[b1]) :- B(j,s,c1,b),
                           E(s,t,w),H(c1,c2,h),
                           b1 = 0.8 * w * b * h;
                           {sum[Δb] < 0.0001}.
```

GCN-Forward is the forward progress of Graph Convolutional Network (GCN) [22]. Program 7 shows its imperative expression. Variable g is the embedding vector on each vertex. Variable A represents a normalized weighted adjacency table that stores the normalized weight w on edges. Data set Para stores the learnable parameter matrix. Function *relu* resets negative values to zero as {return (x>0)?x:0;}. In rule r1, F' is $relu(g * p) \cdot w$, where $*$ is matrix multiplication and $\cdot$ is scalar multiplication, and G is sum. Because we have 1 for $sum(relu(sum(-1, 2)), relu(sum(1, -2)))$ but have 3 for $sum(relu(-1), relu(2), relu(1), relu(-2))$, Property 2 of Theorem 1 does not hold. We skip the details of CommNet [52] that cannot pass the check due to the page limit.

Program 7. GCN-Forward

```
r1. GCN(j+1,Y,sum[g1]) :- GCN(j,X,g),
                          A(X,Y,w),Para(p),
                          g1 = relu(g * p) · w.
```

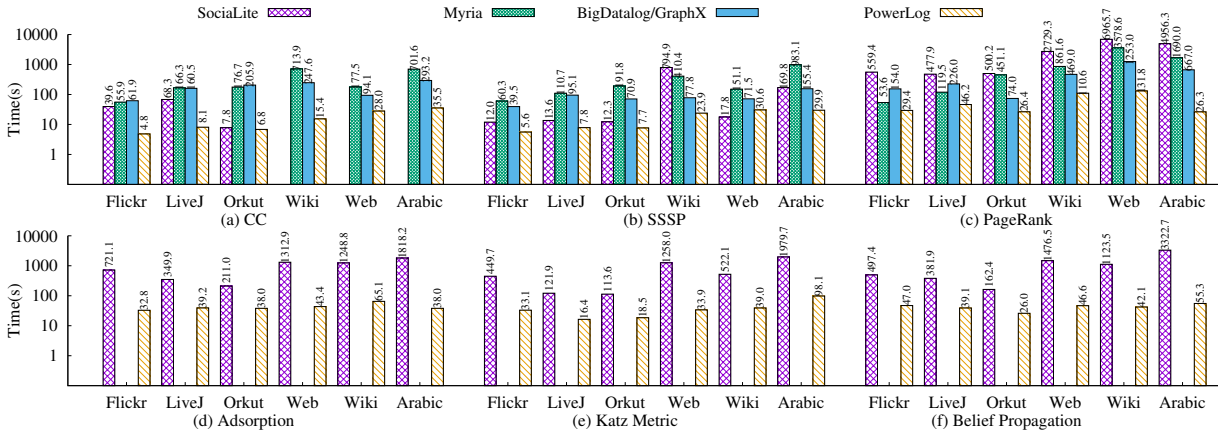


Figure 9: Performance comparisons of PowerLog with existing Datalog systems on 6 algorithms and 6 datasets.

Table 2: Dataset Description

Dataset	Vertices	Links	Abbreviation
Flickr [33]	2,302,925	33,140,017	Flickr
LiveJournal [26]	4,847,571	68,475,391	LiveJ
Orkut [34]	3,072,441	117,184,899	Orkut
ClueWeb09 [10]	20,000,000	243,063,334	Web
Wiki-link [60]	12,150,976	378,142,420	Wiki
Arabic-2005 [5]	22,744,080	639,999,458	Arabic

6.2 Experimental Setups

Cluster Setups. We conduct the evaluation on an Aliyun ECS cluster with 17 nodes. Each node is an "ecs.r5.xlarge" instance that has 4 vCPUs and 32GB memory with Ubuntu 16.04 LTS OS. The network bandwidth between nodes is 1.5 Gbps/s. A node is dedicated as the master and the others are configured as workers. Each worker has 4 parallel threads. Apache Hadoop 2.6.4 is used as the distributed storage system. The graph datasets in Table 2 are also used.

Existing Datalog Systems. We compare PowerLog with SociaLite [46], Myria [59], and BigDatalog [49]. All these three systems support semi-naive evaluation but only for monotonic programs. SociaLite and BigDatalog use the sync execution while Myria uses the async one. We get their latest open-source versions from github at [45, 48, 58], respectively.

6.3 Overall Performance

Because SociaLite, Myria, and BigDatalog use different data loading methods, we exclude the data loading time from the execution time. We report the results with the mean of three runs under their best configurations, e.g., we follow the BigDatalog setups [49] that use the Single-Job PSN with SetRDD and 64 partitions. As Adsorption, Katz Metric, and Belief Propagation are not supported by Myria and BigDatalog, we only compare to SociaLite on these three programs.

Figure 9(a) shows the execution time of CC. All systems can use the incremental evaluation on CC. The performance differences come from different execution engines. Because CC of SociaLite runs out of memory on Wiki-link, ClueWeb09 and Arabic-2005 datasets, we do not show the corresponding results. PowerLog consistently outperforms others and can achieve **1.1x** to **46.4x** speedups. Figure 9(b) shows the execution time of SSSP. All systems can also use the incremental evaluation. PowerLog outperforms others in almost all cases and can achieve **1.6x** to **33.2x** speedups. On ClueWeb09 dataset, SociaLite is 1.7x faster than PowerLog. This is because ClueWeb09 dataset has a small graph diameter that can be optimized by the delta stepping method [31], which is used in SociaLite only. Figure 9(c) shows the execution time of PageRank. Different from CC and SSSP, the original PageRank is non-monotonic and semi-naive evaluation cannot be used. SociaLite and Myria use naive evaluation. BigDatalog does not support PageRank. We use GraphX [15] to substitute BigDatalog since both are built upon Apache Spark. As PageRank can pass the condition check, PowerLog uses MRA evaluation with the sync-async engine and can achieve **1.8x** to **188.3x** speedups.

Figures 9(d)-(f) compare PowerLog and SociaLite on Adsorption, Katz Metric, Belief Propagation[4]. Because these programs are non-monotonic (similar as PageRank), semi-naive evaluation cannot be used. SociaLite uses naive evaluation with an additional join in each iteration. PowerLog can use MRA evaluation and achieve **5.6x** to **47.8x**, **6.1x** to **37.1x**, and **6.2x** to **60.1x** speedups, respectively.

6.4 Performance Gain Analysis

The performance gain of PowerLog comes from the incremental evaluation (i.e., MRA evaluation) and the high efficiency of our unified sync-async execution. We quantify the

[4]We simplify its implementation for large graph datasets by abstracting vertex-pairs into vertices.

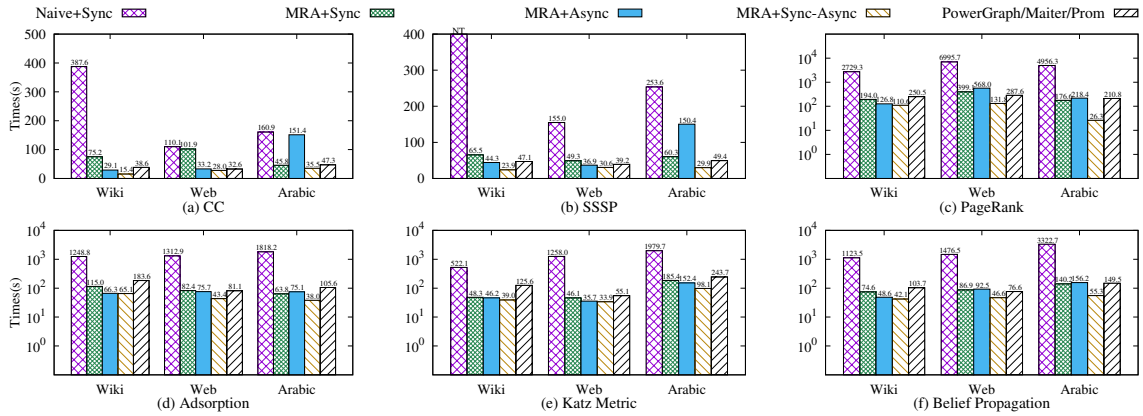


Figure 10: Performance gain of PowerLog from MRA evaluation and sync-async execution.

gain from MRA evaluation and the gain from unified sync-async execution separately. We first use naive evaluation with sync execution and then combine MRA evaluation with sync, async, and unified sync-async execution.

Figure 10(a) shows the results of CC, where MRA evaluation can achieve **1.1x** to **5.2x** speedups over naive evaluation (both with sync). Async execution can further improve the performance on Wiki-link and ClueWeb09 datasets. However, it cannot outperform sync execution on Arabic-2005 dataset. In contrast, sync-async execution can outperform both sync and async execution and get **3.9x** to **25.2x** speedups over naive evaluation with sync execution. Figure 10(b) shows the results of SSSP. MRA evaluation achieves **3.1x** to **4.1x** speedups over naive evaluation (both with sync); and the speedups can be further increased by sync-async execution, leading to **5.1x** to **8.5x** speedups over naive evaluation. Figure 10(c) shows the results of PageRank. MRA evaluation with unified sync-async execution can get the most speedups, leading to **24.7x** to **188.3x** speedups over naive evaluation (with sync). On Arabic dataset, MRA evaluation has 28.1x speedup over naive evaluation (both with sync); and sync-async execution can further achieve 6.7x speedup over sync execution (both with MRA evaluation), leading to the overall 188.3x speedup over naive evaluation with sync execution that is what SociaLite does for PageRank of `Program 2`.

Figures 10(d)-(e) show the performance gains from MRA evaluation and unified sync-async execution for Adsorption, Katz Metric, and Belief Propagation. These figures show that MRA evaluation can significantly reduce the execution time from naive evaluation. However, which execution mode (sync or async) can deliver better performance is not certain. With our sync-async execution, MRA evaluation can get best performance on all algorithms and datasets, leading to **19.2x** to **47.8x**, **13.4x** to **37.1x**, and **26.7x** to **60.1x** speedups over naive evaluation with sync execution in Figures 10(d)-(e).

We further evaluate PowerLog by comparing it with existing graph processing systems that support the incremental computation [11, 32]. As such a system that supports the incremental computation for all evaluated algorithms does not exist, we use PowerGraph [14] for CC and SSSP, Maiter [68] for PageRank, Adsorption, and Katz Metric, and Prom [62] for Belief Propagation. Besides the incremental computation, PowerGraph can use either sync or async execution, and we use its best performance results in evaluation. Maiter and Prom use async execution. As shown in Figures 10(a)-(e), with the incremental computation, these graph processing systems have better performance than that of naive evaluation with sync execution by PowerLog and can get comparable performance with either "MRA+Sync" or "MRA+Async" of PowerLog. For all cases, "MRA+Sync-Async" of PowerLog can outperform others due to the effectiveness of the unified sync-async engine. Furthermore, using these graph processing systems, as well as the Datalog systems [46, 50, 59], users have to manually check and rewrite programs to use the incremental computation.

6.5 Comparing Other Execution Engines

We further evaluate our unified sync-async engine with the Adaptive Asynchronous Parallel (AAP) model proposed in Grape+ system [12]. Grape+ is a parallel graph processing system that makes the execution mode switching among SP (sync parallel), AP (async parallel), and SSP (stale synchronous parallel) on each worker. Different modes exist not only at different stages of the execution but also among different workers in the same stage. To achieve such a complex hybrid execution, Grape+ uses a block-based computation, where each worker decides its own execution mode by analyzing the sizes of in-messages. The network thread communicates with others via a fix-sized buffer.

Our unified sync-async engine can make a timely adjustment of the levels of sync and async to improve performance. The major difference with AAP is that our adjustment is based on valid data generated on each worker, instead of in-messages from others. Each worker decides its own levels of sync and async and controls the communication frequency

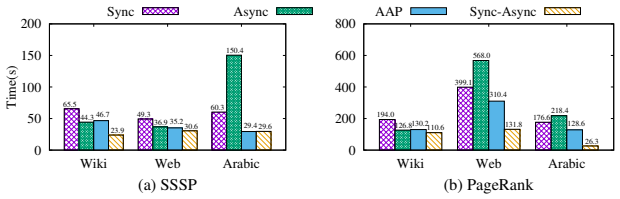


Figure 11: Comparing sync-async with AAP [12]

to affect others, but does not depend on the received data size. In addition, our approach is vertex-based and uses dynamic message sizes to adjust sync and async in a finer granularity.

Since AAP is not released yet, we follow the paper to implement AAP and integrate it with semi-naive evaluation in our execution engine. We evaluate SSSP and PageRank with four modes, i.e., sync, async, AAP, and our unified sync-async. Figure 11(a) shows the performance of SSSP. In this figure, AAP has the comparable performance with our sync-async engine on Arabic-2005 dataset; but ours outperforms AAP on other two datasets. For PageRank in Figure 11(b), AAP outperforms both sync and async modes on ClueWeb09 and Arabic-2005 datasets, and has comparable performance with async on Wiki-link dataset. On all datasets, our sync-async engine shows the best performance.

7 RELATED WORK

Monotonic Aggregates in Recursive Queries. Enabling aggregates in recursive queries is long-term desired [36, 37, 43]. But even the recent SQL standard disallows the use of aggregates in recursion. Ross and Sagiv [43] observe that particular monotonic aggregates can be used in recursive queries. Recent studies on the formalization of monotonic aggregates [29, 30, 65] are conducive. A class of recursive aggregate programs can be written to monotonic programs. However, due to the absence of an analytical foundation, these studies can only write programs with monotonic aggregates in a manually case-by-case manner. REX [32] supports the incremental computation with user-specified termination conditions and explicitly-defined delta operations for pipelined queries. But it does not guarantee the correctness of user-defined delta computations. The conditions proposed in this paper can help REX achieve this goal.

RaSQL [16] is a recursive-aggregate-SQL system for big data analytics. RaSQL and PowerLog share the same goal of leveraging the incremental evaluation for high performance. RaSQL summarizes the Pre-Mappability (PreM) conditions. A recursive aggregate program satisfying PreM is identified as being monotonic and can be executed with semi-naive evaluation. PowerLog goes further to enable the incremental computation for some non-monotonic programs, e.g., PageRank, thus significantly enlarges its application scope of incremental computation. More importantly, the condition check is automated in PowerLog. This method is also applicable to verify the conditions of using other parallel techniques [19].

Datalog Systems. Besides the three representative Datalog systems used in our experiments, there exist several other Datalog systems. DeALS [50, 51] is a deductive database system built on Datalog language. It supports multiple execution platforms, such as multi-core machines and clusters. Datalography [35] can transform Datalog programs to graph processing programs on Giraph [4] so that Datalog programs can be efficiently evaluated in a distributed computing environment. GraphRex [66] provides a Datalog-like declarative interface and a series of optimizations for graph analysis.

Async Execution in Graph Processing. Many graph processing systems support async execution [9, 17, 27, 47, 53, 56, 68]. Recent studies show that neither sync nor async execution can consistently outperform the other. Besides Grape+ [12], several sync-async hybrid systems have emerged. PowerSwitch [61] performs an automatic switch between sync and async on workers. SEP-Graph [57] selects the shortest execution paths of graph algorithms on GPU, considering sync and async with different communication and vertex traversal schemes. Most of these systems are performance-centric without considering the correctness of async execution.

8 CONCLUSION

We have designed and implemented PowerLog, which consists of three components in both theory and system building. First, we lay an analytical foundation for conditions to determine whether a monotonic or even a non-monotonic program can be correctly executed in an incremental and asynchronous way. Second, we develop a condition verification tool to automatically check if a program satisfies the conditions. Finally, we build a unified sync-async processing engine to minimize execution times for the programs that satisfy the conditions. Compared with three representative Datalog systems in large-scale experiments with many workloads, we show the effectiveness of PowerLog. We believe the methodology and analytical foundation in this paper are applicable to a broad scope of incremental computing for graph processing and neural network systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their encouraging and constructive comments and suggestions. The work is supported by the National Key R&D Program of China under grants 2018YFB1003404, the U.S. National Science Foundation under grants CCF-1629403, IIS-1718450, CCF-2005884, the National Natural Science Foundation of China under grants 61672141, U1811261, and the Fundamental Research Funds for the Central Universities under grants N181605017, N181604016.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu (Eds.). 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in Time and Space. In *Datalog 2011*. 262–281.

[3] ANTLR 2020. ANother Tool for Language Recognition. http://www.antlr.org/

[4] Apache Giraph 2020. Iterative graph processing system. http://giraph.apache.org

[5] Arabic-2005 2020. Arabic-2005 Network. http://law.di.unimi.it/webdata/arabic-2005/

[6] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '15)*. 1371–1382.

[7] Shumeet Baluja, Rohan Seth, D. Sivakumar, Yushi Jing, Jay Yagnik, Shankar Kumar, Deepak Ravichandran, and Mohamed Aly. 2008. Video Suggestion and Discovery for Youtube: Taking Random Walks Through the View Graph. In *Proceedings of the 17th International Conference on World Wide Web (WWW '08)*. ACM, New York, NY, USA, 895–904.

[8] François Bancilhon. 1986. *Naive Evaluation of Recursively Defined Relations*.

[9] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 235–248.

[10] ClueWeb09 2020. The ClueWeb09 Dataset. https://lemurproject.org/clueweb09/

[11] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental Graph Computations: Doable and Undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 155–169.

[12] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. 2018. Adaptive Asynchronous Parallelization of Graph Algorithms. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '18)*. 1141–1156.

[13] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing Sequential Graph Computations. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '17)*. 495–510.

[14] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, USA, 17–30.

[15] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 599–613.

[16] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. 2019. RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 467–484.

[17] Minyang Han and Khuzaima Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Systems. *Proceedings of the VLDB Endowment* 8, 9 (May 2015), 950–961.

[18] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS '13)*. 1223–1231.

[19] Kaixi Hou, Hao Wang, Wu-chun Feng, Jeffrey S. Vetter, and Seyong Lee. 2018. Highly Efficient Compensation-based Parallelism for Wavefront Loops on GPUs. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '18)*. 276–285.

[20] Glen Jeh and Jennifer Widom. 2002. SimRank: A Measure of Structural-Context Similarity. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining(KDD '02)*. 538–543.

[21] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (01 Mar 1953), 39–43.

[22] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR '17)*.

[23] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory* 47, 2 (Feb 2001), 498–519.

[24] Monica S. Lam, Stephen Guo, and Jiwon Seo. 2013. SociaLite: Datalog Extensions for Efficient Social Network Analysis. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE '13)*. 278–289.

[25] Amy N. Langville and Carl D. Meyer. 2004. Deeper Inside PageRank. *INTERNET MATHEMATICS* 1 (2004), 2004.

[26] Livejournal 2020. KONECT Livejournal so network dataset. http://konect.uni-koblenz.de/networks/soc-LiveJournal1

[27] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (April 2012), 716–727.

[28] Grzegorz Malewicz, Matthew H. Austern, Aart J. C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '10)*. 135–146.

[29] Mirjana Mazuran, Edoardo Serra, and Zaniolo Carlo. 2013. Extending the power of datalog recursion. *VLDB Journal* 22, 4 (2013), 471–493.

[30] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. 2013. A declarative extension of horn clauses, and its significance for datalog and its applications. *Theory and Practice of Logic Programming (TPLP '13)* 13, 4-5 (2013), 609–623.

[31] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (Oct. 2003), 114–152.

[32] Svilen R. Mihaylov, Zachary G. Ives, and Sudipto Guha. 2012. REX: Recursive, Delta-Based Data-Centric Computation. *Proceedings of the VLDB Endowment* 5, 11 (July 2012), 1280–1291.

[33] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2008. Growth of the Flickr Social Network. In *SIGCOMM Workshop on Social Networks (WoSN '08)*.

[34] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proceedings of the ACM/USENIX Internet Measurement Conference (IMC '07)*. San Diego, CA.

[35] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. 2016. Datalography: Scaling datalog graph analytics on graph processing systems. In *Proceedings of the IEEE International Conference on Big Data (BigData '16)*. 56–65.

[36] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. The Magic of Duplicates and Aggregates. In *Proceedings of the International Conference on Very Large Databases (VLDB '90)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 264–277.

[37] Sumlt Oanguly and Sergio Greco. 1991. Minimum and Maximum Predicates in Logic Programming. In *Proceedings of the 1991 ACM Symposium on Principles of Database Systems (PODS '91)*.

[38] OpenMPI 2020. A High Performance Message Passing Library. https://www.open-mpi.org/

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. The PageRank Citation Ranking: Bringing Order to the Web. *Stanford Digital Libraries Working Paper* 9, 1 (1998), 1–14.

[40] Judea Pearl. 1982. Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '82)*.

[41] Protostuff 2020. Java serialization library, proto compiler, code generator. https://github.com/protostuff/protostuff

[42] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[43] Kenneth A. Ross and Yehoshua Sagiv. 1992. Monotonic Aggregation in Deductive Databases. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS '92)*. 114–126.

[44] Baruch Schieber and Uzi Vishkin. 1988. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.* 17, 11 (1988), 1145–1152.

[45] Jiwon Seo. 2016. SociaLite: Query Language For Large-Scale Graph Analysis. https://github.com/socialite-lang/socialite

[46] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis. *Proceedings of the VLDB Endowment* 6, 14 (Sept. 2013), 1906–1917.

[47] Xuanhua Shi, Junling Liang, Sheng Di, Bingsheng He, Hai Jin, Lu Lu, Zhixiang Wang, Xuan Luo, and Jianlong Zhong. 2015. Optimization of Asynchronous Graph Processing on GPU with Hybrid Coloring Model. In *Proceedings of the ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '15)*. 271–272.

[48] Alexander Shkapsky. 2016. BigDatalog on Spark. https://github.com/ashkapsky/BigDatalog

[49] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1135–1149.

[50] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '15)*. 867–878.

[51] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. 2013. Graph Queries in a Next-generation Datalog System. *Proceedings of the VLDB Endow.* 6, 12 (Aug. 2013), 1258–1261.

[52] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. 2016. Learning Multiagent Communication with Backpropagation. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS '16)*. Curran Associates Inc., USA, 2252–2260.

[53] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *Proceedings of the VLDB Endowment* 7, 3 (Nov. 2013), 193–204.

[54] TROVE 2020. High performance collections for Java. http://trove.starlight-systems.com/

[55] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111.

[56] Guozhang Wang, Wenlei Xie, Alan Demers, and Johannes Gehrke. 2013. Asynchronous Large-Scale Graph Processing Made Easy. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR '13)*.

[57] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: Finding Shortest Execution Paths for Graph Processing Under a Hybrid Framework on GPU. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 38–52.

[58] Jingjing Wang. 2017. Myria: A Scalable Analytics-As-A-Service Platform Based on Relational Algebra. https://github.com/uwescience/myria

[59] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. 2015. Asynchronous and Fault-tolerant Recursive Datalog Evaluation in Shared-nothing Engines. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1542–1553.

[60] Wiki 2020. Wiki links english dataset. http://konect.uni-koblenz.de/networks/wikipedia_link_en

[61] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *Proceedings of the ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '15)*. 194–204.

[62] Jiangtao Yin and Lixin Gao. 2014. Scalable Distributed Belief Propagation with Prioritized Block Updates. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management (CIKM '14)*. ACM, New York, NY, USA, 1209–1218.

[63] Z3 2020. SMT Solver. https://github.com/Z3Prover/z3

[64] Z3-guide 2020. Getting Started with Z3: A Guide. https://rise4fun.com/z3/tutorial

[65] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. 2017. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory and Practice of Logic Programming* 17, 5-6 (2017), 1048–1065.

[66] Qizhen Zhang, Akash Acharya, Hongzhi Chen, Simran Arora, Ang Chen, Vincent Liu, and Boon Thau Loo. 2019. Optimizing Declarative Graph Queries at Large Scale. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 1411–1428.

[67] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2011. PrIter: A Distributed Framework for Prioritized Iterative Computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*. 1–14.

[68] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2014. Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems* 25, 8 (Aug. 2014), 2091–2100.