TPLP **19** (5–6): 1056–1072, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000358

A Case for Stale Synchronous Distributed Model for Declarative Recursive Computation

ARIYAM DAS and CARLO ZANIOLO
Department of Computer Science, University of California, Los Angeles, USA
(*e-mail:* {ariyam, zaniolo}@cs.ucla.edu)

submitted 31 July 2019; accepted 31 July 2019

Abstract

A large class of traditional graph and data mining algorithms can be concisely expressed in Datalog, and other Logic-based languages, once aggregates are allowed in recursion. In fact, for most BigData algorithms, the difficult semantic issues raised by the use of non-monotonic aggregates in recursion are solved by *Pre-Mappability* ($\mathcal{P}reM$), a property that assures that for a program with aggregates in recursion there is an equivalent aggregate-stratified program. In this paper we show that, by bringing together the formal abstract semantics of stratified programs with the efficient operational one of unstratified programs, $\mathcal{P}reM$ can also facilitate and improve their parallel execution. We prove that $\mathcal{P}reM$-optimized lock-free and decomposable parallel semi-naive evaluations produce the same results as the single executor programs. Therefore, $\mathcal{P}reM$ can be assimilated into the data-parallel computation plans of different distributed systems, irrespective of whether these follow bulk synchronous parallel (BSP) or asynchronous computing models. In addition, we show that non-linear recursive queries can be evaluated using a hybrid stale synchronous parallel (SSP) model on distributed environments. After providing a formal correctness proof for the recursive query evaluation with $\mathcal{P}reM$ under this relaxed synchronization model, we present experimental evidence of its benefits.

KEYWORDS: Datalog, Deductive Databases, Recursive Query, Stale Synchronous Parallel Model, Bulk Synchronous Parallel Model, Parallel and Distributed Computing

1 Introduction

The growing interest in Datalog-based declarative systems like *LogicBlox* (Aref et al. 2015), *BigDatalog* (Shkapsky et al. 2016), *SociaLite* (Seo et al. 2013), *BigDatalog-MC* (Yang et al. 2017) and *Myria* (Wang et al. 2015) has brought together important advances on two fronts: (i) Firstly, Datalog, with support for aggregates in recursion (Mazuran et al. 2013), has sufficient power to express succinctly declarative applications ranging from complex graph queries to advanced data mining tasks, such as frequent pattern mining and decision tree induction (Condie et al. 2018). (ii) Secondly, modern architectures supporting in-memory parallel and distributed computing can deliver scalability and performance for this new generation of Datalog systems.

For example *BigDatalog* (bulk synchronous parallel processing on shared-nothing architecture), *BigDatalog-MC* (lock-free parallel processing on shared-memory multicore architecture), *Myria* (asynchronous processing on shared-nothing architecture) spearheaded the system-level scheduling, planning and optimization for different parallel computing models. This line of work was quite successful for Datalog, and also for recursive

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

SQL queries that have borrowed this technology (Gu et al. 2019). Indeed, our recent general-purpose Datalog systems surpassed commercial graph systems like GraphX on many classical graph queries in terms of performance and scalability (Shkapsky et al. 2016).

Much of the theoretical groundwork contributing to the success of these parallel Datalog systems was laid out in the 90s. For example, in their foundation work (Ganguly et al. 1992) investigated parallel *coordination-free* (asynchronous) bottom-up evaluations of simple linear recursive programs (without any aggregates). In fact, many recent works have pushed this idea forward under the broader umbrella of *CALM conjecture* (Consistency And Logical Monotonicity) (Ameloot et al. 2013) which establishes that monotonic Datalog (Datalog without negation or aggregates) programs can be computed in an *eventually consistent, coordination-free* manner (Ameloot 2014), (Ameloot et al. 2015). This line of work led to the asynchronous data-parallel (for *Myria*) and lock-free evaluation plans for many of the aforementioned systems (e.g. *BigDatalog-MC*). Simultaneously, another branch of research about 'parallel correctness' for simple non-recursive conjunctive queries (Ameloot et al. 2017) focused on optimal data distribution policies for repartitioning the initial data under Massively Parallel Communication model (MPC). However, notably, this theoretical groundwork left out programs using aggregates in recursion, for which the existence of a formal semantics could not be guaranteed. But, this situation has changed recently because of the introduction of the notion of *Pre-Mappability*[1] ($\mathcal{P}reM$) (Zaniolo et al. 2017) that has made possible the use of aggregates in recursion to express efficiently a large range of applications (Condie et al. 2018). A key aspect of this line of work has been the use of non-monotonic aggregates and pre-mappable constraints inside recursion, while preserving the formal declarative semantics of aggregate-stratified programs, thanks to the notion of $\mathcal{P}reM$ that guarantees their equivalence. Unlike more complex non-monotonic semantics, stratification is a syntactic condition that is easily checked by users (and compilers), who know that the presence of a formal declarative semantics guarantees the portability of their applications over multiple platforms. Furthermore, evidence is mounting that a higher potential for parallelism is also gained under $\mathcal{P}reM$. Naturally, we would like to examine the applicability of $\mathcal{P}reM$ under a parallel and distributed setting and analyze its potential gains using the rich models of parallelism previously proposed for Datalog and other logic systems.

In this paper, therefore, we begin by examining how $\mathcal{P}reM$ interacts under a parallel setting, and address the question of whether it can be incorporated into the parallel evaluation plans on shared-memory and shared-nothing architectures. Furthermore, the current crop of Datalog systems supporting aggregates in recursion have only explored Bulk Synchronous Parallel (BSP) and asynchronous distributed computing models. However, the new emerging paradigm of Stale Synchronous Parallel (SSP) processing model (Cui et al. 2014) has shown to speed up big data analytics and machine learning algorithm execution on distributed environments (Lee et al. 2014), (Ho et al. 2013) with *bounded staleness*. SSP processing allows each worker in a distributed setting to see and use another worker's obsolete (stale) intermediate solution, which is out-of-date only by

[1] In our initial work (Zaniolo et al. 2017), we interchangeably used the term *Pre-Applicability*. However, in our follow-up works (Condie et al. 2018), (Zaniolo et al. 2018), we consistently used the term *Pre-Mappability* since the latter was deemed more appropriate in the context of 'pre-mapping' aggregates and constraints to recursive rules.

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

a limited (bounded) number of epochs. On the contrary, in a BSP model every worker coordinates at the end of each round of computation and sees each others' current intermediate results. This relaxation of the synchronization barrier in a SSP model can reduce idle waiting of the workers (time spent waiting to synchronize), particularly when one or more workers (stragglers) lag behind others in terms of computation. Thus, in this paper, we also explore if declarative recursive computation can be executed under the loose consistency model of SSP processing and if it has the same convergence as that under a BSP processing framework. To our surprise, we find *PreM* dovetails excellently with SSP model for a class of non-linear recursive queries with aggregates, which are not embarrassingly parallel and still require some coordination between the workers to reach eventual consistency (Interlandi and Tanca 2018). Thus, the contributions of this paper can be summarized as follows:

- We show that *PreM* is applicable to parallel bottom-up semi-naive evaluation plan, terminating at the same minimal fixpoint as the corresponding single executor based sequential execution.
- We further show how recursive query evaluation with *PreM* can operate effectively under a SSP distributed model.
- Finally, we discuss the merits and demerits of a SSP model with initial empirical results on some recursive query examples, thus opening up an interesting direction for future research.

2 An Overview of *PreM*

This section provides a brief overview about *PreM* and some of its properties (Zaniolo et al. 2016), (Zaniolo et al. 2018). Consider the Datalog query in Example 1 that computes the shortest path between all pairs of vertices in a graph, given by the relation `arc(X, Y, D)`, where `D` is the distance between source node `X` and destination node `Y`. The $\texttt{min}\langle\texttt{D}\rangle$ syntax in our example indicates *min* aggregate on the cost variable `D`, while `(X, Y)` refer to the group-by arguments. This head notation for aggregates directly follows from SQL-2 syntax, where cost argument for the aggregate consists of one variable and group-by arguments can have zero or more variables. Rules $r_{1.3}$ in the example shows that the aggregate *min* is computed at a stratum higher than the recursive rule ($r_{1.2}$).

Example 1. *All Pairs Shortest Path*

$r_{1.1}$: $\texttt{path}(\texttt{X},\texttt{Y},\texttt{D})$ <- $\texttt{arc}(\texttt{X},\texttt{Y},\texttt{D})$.

$r_{1.2}$: $\texttt{path}(\texttt{X},\texttt{Y},\texttt{D})$ <- $\texttt{path}(\texttt{X},\texttt{Z},\texttt{Dxz}), \texttt{arc}(\texttt{Z},\texttt{Y},\texttt{Dzy}), \texttt{D} = \texttt{Dxz} + \texttt{Dzy}$.

$r_{1.3}$: $\texttt{shortestpath}(\texttt{X},\texttt{Y},\texttt{min}\langle\texttt{D}\rangle)$ <- $\texttt{path}(\texttt{X},\texttt{Y},\texttt{D})$.

Incidentally, $r_{1.3}$ can also be expressed with stratified negation as shown in rules $r_{1.4}$ and $r_{1.5}$. This guarantees that the program has a perfect-model semantics, although an iterated fixpoint computation of it can be very inefficient and even non-terminating in presence of cycles.

$r_{1.4}$: $\texttt{shortestpath}(\texttt{X},\texttt{Y},\texttt{D})$ <- $\texttt{path}(\texttt{X},\texttt{Y},\texttt{D}), \neg\texttt{betterpath}(\texttt{X},\texttt{Y},\texttt{D})$.

$r_{1.5}$: $\texttt{betterpath}(\texttt{X},\texttt{Y},\texttt{D})$ <- $\texttt{path}(\texttt{X},\texttt{Y},\texttt{D}),\texttt{path}(\texttt{X},\texttt{Y},\texttt{Dxy}), \texttt{Dxy} < \texttt{D}$.

PreM **Application.** The aforementioned inefficiency can be mitigated with *PreM*, if the *min* aggregate can be pushed inside the fixpoint computation, as shown in rules $r_{2.1}$ and

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

$r_{2.2}$. The following program under $\mathcal{P}reM$ has a stable model semantics and (Condie et al. 2018) showed that this transformation is indeed equivalence-preserving with an assured convergence to a minimal fixpoint within a finite number of iterations. In other words, without $\mathcal{P}reM$ the shortest path in our example (according to rule $r_{1.3}$) is given by the subset of the minimal model (computed from rules $r_{1.1}, r_{1.2}$) obtained after removing `path` atoms that did not satisfy the *min* cost constraint for a given source-destination pair. However, with $\mathcal{P}reM$, the transfer of *min* cost constraint inside recursion results in an optimized program, where the fixpoint computation is performed more efficiently, eventually achieving the same shortest path values (as those produced in the perfect model of the earlier program) by simply copying the atoms from `path` under the name `shortestpath` (rule $r_{2.3}$) after the least fixpoint computation terminates.

$r_{2.1}$: $\texttt{path}(\texttt{X}, \texttt{Y}, \texttt{min}\langle\texttt{D}\rangle)$ <- $\texttt{arc}(\texttt{X}, \texttt{Y}, \texttt{D})$.

$r_{2.2}$: $\texttt{path}(\texttt{X}, \texttt{Y}, \texttt{min}\langle\texttt{D}\rangle)$ <- $\texttt{path}(\texttt{X}, \texttt{Z}, \texttt{Dxz}), \texttt{arc}(\texttt{Z}, \texttt{Y}, \texttt{Dzy}), \texttt{D} = \texttt{Dxz} + \texttt{Dzy}$.

$r_{2.3}$: $\texttt{shortestpath}(\texttt{X}, \texttt{Y}, \texttt{D})$ <- $\texttt{path}(\texttt{X}, \texttt{Y}, \texttt{D})$.

Formal Definition of $\mathcal{P}reM$. For a given Datalog program, let P be the rules defining a (set of mutually) recursive predicate(s) and T be the corresponding *Immediate Consequence Operator* (ICO) defined over P. Then, a constraint γ is said to be $\mathcal{P}reM$ to T (and to P) when, for every interpretation I of the program, we have $\gamma(T(I)) = \gamma(T(\gamma(I)))$.

In Example 1, the final rule $r_{1.3}$ imposes the constraint $\gamma = (\texttt{X}, \texttt{Y}, \texttt{min}\langle\texttt{D}\rangle)$ on $I = \texttt{path}(\texttt{X}, \texttt{Y}, \texttt{D})$ (representing all possible paths) to eventually yield the shortest path between all pairs of nodes. Thus, the aggregate-stratified program defined by rules $r_{1.1} - r_{1.3}$ is equivalent to $\gamma(T(I))$ in the definition of $\mathcal{P}reM$. On the other hand, with *min* aggregate pushed inside recursion, recursive rules $r_{2.1} - r_{2.2}$ represent $\gamma(T(\gamma(I)))$.

$\mathcal{P}reM$ Properties. We now discuss some important results about $\mathcal{P}reM$ from (Zaniolo et al. 2017). We refer interested readers to our paper (Zaniolo et al. 2017) for the detailed proofs. Let T_γ denote the *constrained immediate consequence operator*, where constraint γ is applied after the ICO T, i.e., $T_\gamma(I) = \gamma(T(I))$. The following results hold when γ is $\mathcal{P}reM$ to a positive program P with ICO T:

1. If $I = T(I)$ is a fixpoint for T, then $I' = \gamma(I)$ is a fixpoint for $T_\gamma(I)$, i.e., $I' = T_\gamma(I')$.
2. For some integer n, if $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$, then $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ is a minimal fixpoint for T_γ and $T_\gamma^{\uparrow n}(\emptyset) = \gamma(T^{\uparrow\omega}(\emptyset))$, where $T^{\uparrow\omega} = \bigcup_{n \geq 1} T^{\uparrow n}$

$\mathcal{P}reM$ Provability. We can verify if $\mathcal{P}reM$ holds for a recursive rule by explicitly validating $\gamma(T(I)) = \gamma(T(\gamma(I)))$, i.e., $T_\gamma(I) = T_\gamma(\gamma(I))$ at every iteration of the fixpoint computation. To simplify, this would indicate that we can verify if the *min* constraint can be pushed inside recursion in rule $r_{2.2}$ by inserting an additional goal *is_min* in the body of the rule as follows:

$r'_{2.2}$: $\texttt{path}(\texttt{X}, \texttt{Y}, \texttt{min}\langle\texttt{D}\rangle)$ <- $\texttt{path}(\texttt{X}, \texttt{Z}, \texttt{Dxz}), \textbf{is_min}((\texttt{X}, \texttt{Z}), \texttt{Dxz}), \texttt{arc}(\texttt{Z}, \texttt{Y}, \texttt{Dzy}), \texttt{D} = \texttt{Dxz} + \texttt{Dzy}$.

This additional goal in the body pre-applies the constraint γ on I, followed by the application of T_γ operator, i.e., it expresses $T_\gamma(\gamma(I))$. Note, the *is_min* constraint is satisfied by `Dxz`, if it is the minimum value seen yet in the fixpoint computation for the source-destination pair `(X, Z)`. It is also evident that any other distance value between `(X, Z)`, which violates the *is_min* constraint, will also not satisfy the *min* aggregate at

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

the head of the rule, since the additional goal minimizes the sum `D` for each `Dzy`. Thus, this new goal in the body does not alter the ICO mapping defined by the original recursive rule, thereby proving γ is $\mathcal{P}reM$ in this example program. More broadly speaking, these additional goals can be formally defined as "half functional dependencies", borrowing the terminology from classical database theory of Functional and Multi-Valued Dependencies (FDs and MVDs). We next present the formal definition of *half FD* from (Zaniolo et al. 2018), which will be used later for our proofs.

Definition 1. *(Half Functional Dependency). Let $R(\Omega)$ be a relation on a set of attributes Ω, $X \subset \Omega$ and $A \in \Omega - X$. Considering the domain of A to be totally ordered, a tuple $t \in R$ is said to satisfy the min-constraint is_min$((X), A)$ (denoted as $X \overset{min}{\rightharpoonup} A$), when R contains no tuple with the same X-value and a smaller A-value. Similarly, a tuple $t \in R$ satisfies a max-constraint is_max$((X), A)$ (denoted as $X \overset{max}{\rightharpoonup} A$) if R has no tuple with the same X-value and a larger A-value.*

For any *min* or *max* constraint to be $\mathcal{P}reM$ to a positive program P, the corresponding half FD should hold for the relational view of the relevant recursive predicate across every interpretation I of P, where a relational view for predicate q is defined as $R_q = \{(x_1, ..., x_n) | q(x_1, ..., x_n) \in I\}$ for a given I. (Zaniolo et al. 2018) provides generic templates, based on Functional and Multi-valued Dependencies, for identifying constraints that satisfy $\mathcal{P}reM$.

$\mathcal{P}reM$ with Semi-Naive Evaluation. A naive fixpoint computation trivially generates new atoms from the entire set of atoms available at the end of the last fixpoint iteration. Semi-naive evaluation improves over this naive fixpoint computation with the aid of the following enhancements:

1. At every iteration, track *only* the new atoms produced.
2. Rules are re-written into their differential versions, so that only new atoms are produced and old atoms are never generated redundantly.
3. Ensure step (2) does not generate any duplicate atoms.

For programs where $\mathcal{P}reM$ can be applied, steps (1) and (2) remain identical. However, step (3) is extended so that (i) new atoms produced may not be retained, if they do not satisfy the constraint γ and (ii) existing atoms may get updated and thereafter tracked for the next iteration. For example, new atoms produced from rule $r_{2.2}$ are added to the working set and tracked only if a new source-destination (`X,Y`) path is discovered. On the other hand, if the new `path` atom, thus produced, has a smaller distance than the one in the working set, then the distance of the existing `path` atom is updated to satisfy the *min*-constraint. However, if new `path` atoms are generated, which have larger distances, then they are simply ignored. This understanding of $\mathcal{P}reM$ for semi-naive evaluation leads to a case for SSP model, where significant communication can be saved by condensing multiple updates into one. This is discussed in detail later in Section 5.

3 An Overview of Parallel Bottom-Up Evaluation

One of the early foundational works that established a standard technique to parallelize bottom-up evaluation of *linear recursive* queries was presented in (Ganguly et al. 1992). The authors proposed a *substitution partitioned parallelization* scheme, where the set

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

of possible ground substitutions, i.e., the base (extensional database) and derived relation (intensional database) atoms in the Datalog program are disjointedly partitioned, using a hash-based discriminating function, so that each partition of possible ground substitutions is mapped to exactly one of the parallel workers. The entire computation is then divided among all the workers, operating in parallel, where each worker only processes the partition of ground substitutions mapped to it during the bottom-up semi-naive evaluation. Since, each worker operates on a distinct non-overlapping partition of ground substitutions, no two workers perform the same or redundant computation, i.e., this scheme is *non-redundant*. Formally, if $v(r)$ is a non-repetitive sequence of variables appearing in the body of rule r and $\mathcal{W}$ denotes a finite set of parallel workers, then $h : v(r) \longrightarrow \mathcal{W}$ is a discriminating hash function that divides the workload by assigning the ground substitution and corresponding processing to exactly one worker. The workers can send and receive information (ground instances from partially computed derived relations) to and from other workers to finish the assigned computation tasks. Ganguly et al. summarized the correctness of this parallelization scheme with the following result:

Correctness of Partitioned Parallelization Scheme. Let P be a recursive Datalog program to be executed over $\mathcal{W}$ workers. Under the partitioned parallelization scheme, let Q_i be the program to be executed at worker i and let $Q = \bigcup_{1 \leq i \leq \mathcal{W}} Q_i$. Then, for every interpretation, the least model of the recursive relation in Q is identical to the least model obtained from the sequential execution of P.

Note, the above parallelization strategy did not involve aggregates in recursion. But, nevertheless it was of significant consequence, since the scheme has been extended to derive *lock-free parallel* plans for shared-memory architectures as well as *sharded data parallel decomposable* plans for shared-nothing distributed architectures to parallelize bottom-up semi-naive evaluation of Datalog programs. We discuss them next with examples.

Shared-Memory Architecture. A trivial hash-based partitioning, as described above, can often lead to conflicts between different workers on a shared-memory architecture[2]. This can be prevented with the implementation of classical locks to resolve read-write conflicts. However, recently, (Yang et al. 2017) proposed a hash partitioning strategy based on *discriminating sets* that allows *lock-free* parallel evaluation of a broad class of generic queries including non-linear queries. We illustrate this with our running all pairs shortest path example.

Assume the relations `arc, path` and `shortestpath` from example 1 (rules $r_{1.1} - r_{1.3}$) are partitioned by the first column[3] (i.e., the source vertex), using a hash function h that maps the source vertex to an integer between 1 to $\mathcal{W}$, latter denoting the number of workers. Now, a worker i can execute the following program in parallel:

$$r_{3.1} : \texttt{path(X,Y,D)} \texttt{ <- } \texttt{arc(X,Y,D)}, \texttt{h(X)} = \texttt{i}.$$

$$r_{3.2} : \texttt{path(X,Y,D)} \texttt{ <- } \texttt{path(X,Z,Dxz)}, \texttt{arc(Z,Y,Dzy)}, \texttt{D} = \texttt{Dxz} + \texttt{Dzy}, \texttt{h(X)} = \texttt{i}.$$

$$r_{3.3} : \texttt{shortestpath(X,Y,min}\langle\texttt{D}\rangle\texttt{)} \texttt{ <- } \texttt{path(X,Y,D)}, \texttt{h(X)} = \texttt{i}.$$

[2] For example, two distinct workers may update a `path` atom for the same `(X,Y)` pair in rule $r_{1.2}$, if the hashing is done based on the ground instances of the sequence `{X,Z,Dxz,Z,Y,Dzy}` or even on the sequence `{X,Z,Y}`.

[3] The first attribute forms a *discriminating set* that is used for partitioning.

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

1. The i^{th} worker executes rule $r_{3.1}$ by reading from the i^{th} partition of arc.
2. Once all the workers finish step (1), the i^{th} worker begins semi-naive evaluation with rule $r_{3.2}$, where it reads from the i^{th} partition of `path`, joins with the corresponding atoms from the `arc` relation, which is shared across all the workers, and then writes new atoms into the same i^{th} partition of `path`.
3. Once all the workers finish step (2), the semi-naive evaluation proceeds to the next iteration and repeats step (2) till the least fixpoint is reached.
4. In the final step, the i^{th} worker computes the `shortestpath` for the i^{th} partition.
5. All the `shortestpath` data pooled across the workers produce the final query result.

It is easy to observe that the above parallel execution does not require any locks, since each worker is writing to exactly one partition and no two workers are writing to the same partition. We formally define the lock-free parallel bottom-up evaluation scheme next.

Definition 2. *(Lock-free Parallel Bottom-up Evaluation). Let P be a recursive Datalog program to be executed over $\mathcal{W}$ workers and let T be the corresponding ICO for the sequential execution of P. Under the lock-free parallel plan executed over $\mathcal{W}$ workers, let Q_i be the program to be executed at worker i, producing an interpretation I_i of the recursive predicate with the corresponding ICO T_i. Then, for every input of base relations, we have, $T_i^{\uparrow\omega}(\emptyset) \bigcap T_j^{\uparrow\omega}(\emptyset) = \emptyset$ for $1 \leq i, j \leq \mathcal{W}$, $i \neq j$. It also follows from the correctness of partitioned parallelization scheme that $\bigcup_{1 \leq i \leq \mathcal{W}} T_i^{\uparrow\omega}(\emptyset) = T^{\uparrow\omega}(\emptyset)$.*

The underlying strategy of a lock-free parallel plan to use disjointed data partitions have also been adopted to execute data-parallel distributed bottom-up evaluations, as explained next.

Shared-Nothing Architecture. Distributed systems like *BigDatalog* (Shkapsky et al. 2016) also divide the entire dataset into disjointed data shards in an identical manner as the lock-free partitioning technique described above. Each data shard resides in the memory of a worker and this partitioning scheme reduces the data shuffling required across different workers (Shkapsky et al. 2016). In the context of shared-nothing architecture, this sharding scheme and subsequent distributed bottom-up evaluation is termed as a *decomposable plan* (Shkapsky et al. 2016), (Gu et al. 2019). In the rest of this paper, we will use the term 'lock-free parallel plan' in the context of shared-memory architecture and 'parallel decomposable plan' in the context of distributed environment for clarity.

Distributed systems like *BigDatalog* and *SociaLite* (Seo et al. 2013) perform the fixpoint computation under BSP model with synchronized iterations. However, note that, if each node caches the `arc` relation, then each node can operate independently without any *co-ordination* or *synchronization* with other nodes (i.e., step 3 listed before in the lock-free evaluation plan becomes unnecessary). The *Myria* system follows this asynchronous computing model for the query evaluation. Interestingly, (Ganguly et al. 1992) showed that only a subclass of linear recursive queries[4] can be executed in a *co-ordination free* manner or asynchronously. Thus, for a large class of non-linear and even many linear recursive queries (e.g. same generation query (Ganguly et al. 1992)), BSP computing model has been the only viable option.

[4] The dataflow graph corresponding to the linear recursive query must have a cycle.

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

4 Parallel Evaluation with $\mathcal{P}reM$

In this section, we now examine if $\mathcal{P}reM$ can be easily integrated into the *lock-free parallel* and *parallel decomposable bottom-up evaluation* plans that have been widely adopted across shared-memory and shared-nothing architectures for a broad range of generic queries. We next provide some interesting theoretical results.

Lemma 1. Let $R(\Omega)$ be a relation defined over a set of attributes Ω, where $X \subset \Omega$ and $A \in \Omega - X$. For a subset S of X ($S \subseteq X$), if R is divided into k disjoint subsets $R_1, R_2, ..., R_k$ using a hash function $h : S \rightarrow k$ such that R_i is defined as $R_i = \{e | e \in R \wedge h(e\,[S]) = i\}$, then a tuple $t \in R$ satisfying $X \overset{min}{\rightharpoonup} A$ (or, $X \overset{max}{\rightharpoonup} A$) will also satisfy $X \overset{min}{\rightharpoonup} A$ (or, $X \overset{max}{\rightharpoonup} A$ respectively) over R_i and vice versa, where $h(t\,[S]) = i$.

Proof. This follows directly from the fact that since $S \subseteq X$, for any two tuples $t_1, t_2 \in R$, if $t_1\,[X] = t_2\,[X]$, then $t_1\,[S] = t_2\,[S]$, i.e., any two tuples with the same X-value will be mapped into the same partition, decided by their common S-value. Since, all tuples with the same X-value belong to a single partition, any tuple $t \in R_i$ will satisfy $X \overset{min}{\rightharpoonup} A$ (or, $X \overset{max}{\rightharpoonup} A$) over both R and R_i.

Theorem 1. *Let P be a recursive Datalog program, T be its corresponding ICO and let the constraint γ be $\mathcal{P}reM$ to T and P, resulting in the constrained ICO T_γ. Let P be executed over $\mathcal{W}$ workers under a lock-free parallel (or parallel decomposable) bottom-up evaluation plan, where Q_i is the program executed at worker i and T_i be the corresponding ICO defined over Q_i. If the group-by arguments used for the γ constraint also contain the discriminating set used for partitioning in the lock-free parallel (or parallel decomposable) plan, then:*

1. *γ is also $\mathcal{P}reM$ to T_i and Q_i, for $1 \leq i \leq \mathcal{W}$.*
2. *For some integer n, if $T_\gamma^{\uparrow n}(\emptyset)$ is the minimal fixpoint for T_γ, then $T_\gamma^{\uparrow n}(\emptyset) = \bigcup_{1 \leq i \leq \mathcal{W}} T_{i_\gamma}^{\uparrow n}(\emptyset)$, where T_{i_γ} denotes the constrained ICO with respect to T_i.*

Proof. The proof for (i) follows trivially from lemma 1 and the $\mathcal{P}reM$ provability technique discussed earlier in Section 2.

Since, γ is $\mathcal{P}reM$ to T and P, $T_\gamma^{\uparrow n}(\emptyset) = \gamma(T^{\uparrow \omega}(\emptyset))$ according to the properties of $\mathcal{P}reM$. Similarly, since for $1 \leq i \leq \mathcal{W}$, γ_i is $\mathcal{P}reM$ to T_i and Q_i (from (i) of theorem 1), $T_{i_\gamma}^{\uparrow n_i}(\emptyset) = \gamma(T_i^{\uparrow \omega}(\emptyset))$, for some integer n_i, where $T_{i_\gamma}^{\uparrow n_i}(\emptyset) = T_{i_\gamma}^{\uparrow (n_i+1)}(\emptyset)$ is the minimal fixpoint for T_{i_γ}. Thus, $\bigcup_{1 \leq i \leq \mathcal{W}} T_{i_\gamma}^{\uparrow n_i}(\emptyset) = \bigcup_{1 \leq i \leq \mathcal{W}} \gamma(T_i^{\uparrow \omega}(\emptyset))$. Now, γ constraints are also trivially $\mathcal{P}reM$ to union over disjoint sets (Zaniolo et al. 2017), i.e., $\gamma(\bigcup_{1 \leq i \leq \mathcal{W}} T_i^{\uparrow \omega}(\emptyset)) = \bigcup_{1 \leq i \leq \mathcal{W}} \gamma(T_i^{\uparrow \omega}(\emptyset))$. Also recall from the definition of lock-free parallel (or parallel decomposable) plan that $\bigcup_{1 \leq i \leq \mathcal{W}} T_i^{\uparrow \omega}(\emptyset) = T^{\uparrow \omega}(\emptyset)$. Combining these aforementioned equalities, we get,

$$T_\gamma^{\uparrow n}(\emptyset) = \gamma(T^{\uparrow \omega}(\emptyset)) = \gamma(\bigcup_{1 \leq i \leq \mathcal{W}} T_i^{\uparrow \omega}(\emptyset)) = \bigcup_{1 \leq i \leq \mathcal{W}} \gamma(T_i^{\uparrow \omega}(\emptyset)) = \bigcup_{1 \leq i \leq \mathcal{W}} T_{i_\gamma}^{\uparrow n_i}(\emptyset).$$

Since, $T_{i_\gamma}^{\uparrow n_i}(\emptyset)$ is the minimal fixpoint with respect to T_{i_γ}, for $n > n_i$, $T_{i_\gamma}^{\uparrow n}(\emptyset) = T_{i_\gamma}^{\uparrow n_i}(\emptyset)$. Therefore, $T_\gamma^{\uparrow n}(\emptyset) = \bigcup_{1 \leq i \leq \mathcal{W}} T_{i_\gamma}^{\uparrow n}(\emptyset)$.

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

Thus, following theorem 1, we can push the *min* constraint within the parallel recursive plan expressed by rules $r_{3.1} - r_{3.3}$ and rewrite them for worker i as follows:

$r_{4.1}$: path(X, Y, min⟨D⟩) <- arc(X, Y, D), h(X) = i.

$r_{4.2}$: path(X, Y, min⟨D⟩) <- path(X, Z, Dxz), arc(Z, Y, Dzy), D = Dxz + Dzy, h(X) = i.

$r_{4.3}$: shortestpath(X, Y, D) <- path(X, Y, D), h(X) = i.

Thus, we observe that pre-mappable constraints can be also easily pushed inside parallel lock-free (or parallel decomposable) evaluation plans of recursive queries to yield the same minimal fixpoint, yet making them computationally more efficient and safe. Thus, $\mathcal{PreM}$ can be easily incorporated into the parallel computation plans (equivalent to rules $r_{4.1} - r_{4.3}$) of different systems like *BigDatalog-MC*, *BigDatalog* and *Myria*, irrespective of whether they use (1) shared-memory or shared-nothing architecture, or (2) they follow BSP or asynchronous computing models.

5 A Case for Relaxed Synchronization

We now consider a non-linear query, which is equivalent to the linear all pairs shortest path program with the application of $\mathcal{PreM}$ (rules $r_{2.1} - r_{2.3}$). Since this is a non-linear query (rules $r_{5.1} - r_{5.3}$), this program *cannot* be executed in a *coordination-free* manner or *asynchronously* following the technique described in (Ganguly et al. 1992).

$r_{5.1}$: path(X, Y, min⟨D⟩) <- arc(X, Y, D).

$r_{5.2}$: path(X, Y, min⟨D⟩) <- path(X, Z, Dxz), path(Z, Y, Dzy), D = Dxz + Dzy.

$r_{5.3}$: shortestpath(X, Y, D) <- path(X, Y, D).

However, as shown in (Yang et al. 2017), a simple query rewriting technique can produce an equivalent parallel decomposable evaluation plan for this non-linear query. Rules $r_{6.1} - r_{6.4}$ show the equivalent decomposable program, which can be executed by worker i on a distributed system following a bulk synchronous parallel model. In this following decomposable evaluation plan, there is a mandatory synchronization step (rule $r_{6.3}$), where each worker i (operating on the i^{th} partition) copies the new atoms or updates in path produced during the semi-naive evaluation from rule $r_{6.2}$ to $\texttt{path}^{(1)}$ and the new $\texttt{path}^{(1)}$ is then sent to other workers so that they can use it in the evaluation of rule $r_{6.2}$ in the next iteration.

$r_{6.1}$: path(X, Y, min⟨D⟩) <- arc(X, Y, D), h(X) = i.

$r_{6.2}$: path(X, Y, min⟨D⟩) <- path(X, Z, Dxz), $\texttt{path}^{(1)}$(Z, Y, Dzy), D = Dxz + Dzy, h(X) = i.

$r_{6.3}$: $\texttt{path}^{(1)}$(X, Y, min⟨D⟩) <- path(X, Y, D), h(X) = i.

$r_{6.4}$: shortestpath(X, Y, D) <- path(X, Y, D), h(X) = i.

In a bulk synchronous distributed computing model, the communication between the workers in each iteration can be considerably more expensive than the local computation performed by each worker due to the bottleneck of network bandwidth. We now investigate if we can relax this synchronization constraint at every iteration.

Under a *stale synchronous parallel* (SSP) model, a worker i can use an obsolete or stale version of $\texttt{path}^{(1)}$ that omits some recent updates, produced by other workers, for its

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

local computation. In particular, a worker using $\texttt{path}^{(1)}$ at iteration c will be able to use all the atoms and updates generated from iteration 0 to $c-s-1$, $s \geq 0$ is a user-specified threshold for controlling the staleness. In addition, the worker's stale $\texttt{path}^{(1)}$ may have atoms or updates from iteration beyond $c-s-1$, i.e., from iteration $c-s$ to $c-1$ (although this is not guaranteed). The intuition behind this is that in a SSP model, a worker for its local computation should be able to see and use its own updates at every iteration, in addition to seeing and using as many updates as possible from other workers, with the constraint that any updates older than a given age are not missed. This is the *bounded staleness* constraint (Cipar et al. 2013). This leads to two advantages:

1. Workers spend more time performing actual computation, rather than idle waiting for other workers to finish. This can be very helpful, when there are straggling workers present, which lag behind others in an iteration. In fact in distributed computing, stragglers present an acute problem since they can occur for several reasons like hardware differences (Krevat et al. 2011), system failures (Ananthanarayanan et al. 2010), skewed data distribution or even from software management issues and program interruptions caused from garbage collections or operating system noise, etc. (Beckman et al. 2006).
2. Secondly, workers can end up communicating less than under a BSP model. This is primarily because under $\mathcal{P}reM$, each worker can condense several updates computed from different local iterations into a single update before eventually sending it to other workers.

We illustrate the above advantages through an example. Figure 1 shows a toy graph which is distributed across two workers: (i) all edges incident on nodes 1-4 are available on *worker 0*, (ii) and the rest of the edges reside on *worker 1*. Now consider the shortest path between nodes 4 and 8, given by the path 4-3-2-1-5-6-7-8, which spans across 7 hops. The parallel program defined by rules $r_{6.1} - r_{6.4}$ with BSP processing would require at least three synchronized iterations to reach to the least fixpoint by semi-naive evaluation. Now consider *worker 1* to be a straggling node that lags behind *worker 0* during the computation because of hardware differences. Thus, *worker 0* spends significant time idle waiting for *worker 1* to complete, as shown in Figure 2a. But in this example, the shortest path between nodes 4 and 8 changes because of two aspects: (1) the shortest path between nodes 4 and 1 changes and (2) the shortest path between nodes 5 and 8 changes. Both of these computations can be done independently on the two workers and *worker 0* needs to know the eventual shortest path between nodes 5 and 8 calculated by *worker 1* and vice versa. It is important to note that this will only work if each worker can use the most recent *local* updates (newest atoms) generated by itself. In other words, *worker 0* should be able to see the changes of the shortest path between node 4 and node 1 in every iteration (which is generated locally) and use a stale (obsolete) knowledge about the shortest path between nodes 5 and 8 (as sent by *worker 1* earlier). This stale synchronization model is summarized in Figure 2b.

In this same example, note how the minimum cost for the path between node 1 and node 4 (computed by *worker 0*) changes in every iteration: (i) in the first iteration, the minimum cost was 10 given by the edge between node 1 and node 4, (ii) in the next iteration, the minimum cost drops to 7 given by the path 1-3-4 and (iii) in the third iteration the final minimum cost of 5 is given by the sequence 1-2-3-4. In a BSP model,

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

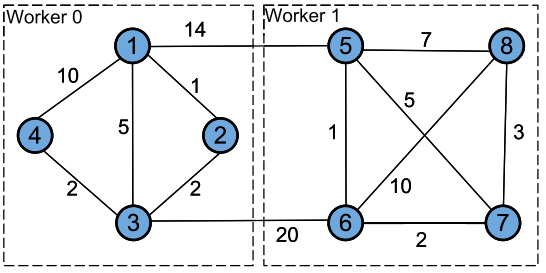


Fig. 1. A toy graph distributed across two workers.

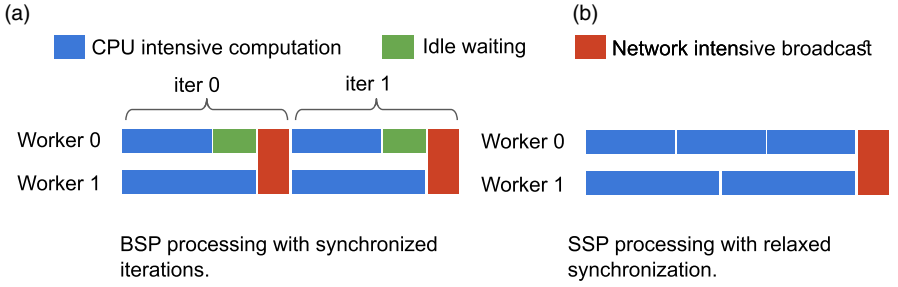


Fig. 2. BSP vs. SSP model for evaluating *all pairs shortest path* query on two workers.

each of this update generated in every iteration needs to be communicated to all the remaining workers. However, in a SSP model due to the advantage of this staleness, these multiple updates from different local iterations can be condensed into one most recent update, which is then sent to other workers. In other words, SSP with $\mathcal{P}reM$ may skip sending some updates to remote workers, thus saving communication time.

Figure 3 formally presents the SSP processing based bottom-up evaluation plan for the non-linear all pairs shortest path example given by rules $r_{6.1} - r_{6.4}$. If the evaluation is executed over a distributed system of $\mathcal{W}$ workers, Figure 3 depicts the execution plan for a worker i. A coordinator marks the completion of the overall evaluation process by individually tracking the termination of each of the worker's task. For simplicity and clarity, we have used the naive fixpoint computation to describe the evaluation plan instead of using the optimized differential fixpoint algorithm. The γ used in the Figure 3 denotes the *min* constraint. Step (3) in this evaluation plan shows how worker i uses stale knowledge from other workers j (denoted by $\texttt{path}_j^{(r')}$) during the recursive rule evaluation, shown by step (4). It is also important to note that in step (4), each worker i is also using the most recent atoms generated by itself (denoted by $\texttt{path}_i^{(r)}$) for the evaluation. The condition in step (6) allows each local computation on worker i to reach local fixpoint or move further by at least $\mathcal{T}$ iterations. Thus, each worker i can condense multiple updates generated within these $\mathcal{T}$ iterations due to $\mathcal{P}reM$ into a single update. Finally, step (9) ensures that if any worker falls beyond the user-defined *staleness bound*, then other workers wait for it to catch up within the desired staleness level before starting their local computations again. We next present some theoretical and empirical results about the SSP model based bottom-up evaluation.

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

1: $\texttt{path}_\texttt{i}^{(0)}(\texttt{X},\texttt{Y},\texttt{D}) := \{(\texttt{X},\texttt{Y},\texttt{D})|\texttt{arc}(\texttt{X},\texttt{Y},\texttt{D})\}$, $\texttt{path}_\texttt{j}^{(0)}(\texttt{X},\texttt{Y},\texttt{D}) := \emptyset\ \forall i \neq j$, $\texttt{r} = 0$, $\texttt{s} = 0$
2: **repeat**
3: $\quad \texttt{path}_\texttt{j}^{(\texttt{r}')}(\texttt{X},\texttt{Y},\texttt{D})$:=Last received $\texttt{path}_\texttt{j}$ by worker $\texttt{i}, \forall i \neq j$.
4: $\quad \texttt{path}_\texttt{i}^{(\texttt{r}+1)}(\texttt{X},\texttt{Y},\texttt{D}) := \gamma\Big(\Big(\bigcup_{i\neq j} \texttt{path}_\texttt{i}^{(\texttt{r})}(\texttt{X},\texttt{Z},\texttt{Dxz}) \bowtie \texttt{path}_\texttt{j}^{(\texttt{r}')}(\texttt{Z},\texttt{Y},\texttt{Dzy})\Big) \bigcup \Big(\texttt{path}_\texttt{i}^{(\texttt{r})}(\texttt{X},\texttt{Z},\texttt{Dxz}) \bowtie \texttt{path}_\texttt{i}^{(\texttt{r})}(\texttt{Z},\texttt{Y},\texttt{Dzy})\Big)\Big)$
5: $\quad \texttt{r} := \texttt{r} + 1$, $\texttt{s} := \texttt{s} + 1$
6: **until** $s < \mathcal{T}$ and $\texttt{path}_\texttt{i}^{(\texttt{r})} \neq \texttt{path}_\texttt{i}^{(\texttt{r}-1)}$
7: $\texttt{s} := 0$
8: Send $\texttt{path}_\texttt{i}^{(\texttt{r})}$ to other workers.
9: **if** for any worker $j, r - r' >$ staleness bound **then**
10: $\quad$ Wait for a new update from worker j before continuing
11: **end if**
12: **if** $\texttt{path}_\texttt{i}^{(\texttt{r})} \neq \texttt{path}_\texttt{i}^{(\texttt{r}-1)}$ or a new update has been received from worker j **then**
13: $\quad$ repeat from Step (2)
14: **else**
15: $\quad$ Send a `finish` message to coordinator.
16: $\quad$ **if** any new update is received from worker j **then**
17: $\quad\quad$ Send a `resume` message to coordinator.
18: $\quad\quad$ Repeat from step (2).
19: $\quad$ **end if**
20: **end if**

Fig. 3. SSP based bottom-up evaluation plan executed by worker i for computing all pairs shortest path.

6 Bottom-up Evaluation with SSP Processing

Under the SSP model, a recursive query evaluation with $\mathcal{P}reM$ constraints has the following theoretical guarantees:

Theorem 2. *Let P be a recursive Datalog program with ICO T and let the constraint γ be $\mathcal{P}reM$ to T and P. Let P have a parallel decomposable evaluation plan that can be executed over $\mathcal{W}$ workers, where Q_i is the program executed at worker i and T_i is the corresponding ICO defined over Q_i. If γ is also $\mathcal{P}reM$ to T_i and Q_i for $1 \leq i \leq \mathcal{W}$, then:*

1. *The SSP processing yields the same minimal fixpoint of $\gamma(T^{\uparrow\omega}(\emptyset))$, as would have been obtained with BSP processing.*
2. *If any worker i under BSP processing requires r rounds of synchronization, then under SSP processing i would require $\leq r$ rounds to reach the minimal fixpoint, where r rounds of synchronization in SSP model means every worker has sent at least r updates.*

Proof. The proof is provided in Appendix A of (Das and Zaniolo 2019).

6.1 SSP Evaluation of Queries without $\mathcal{P}reM$ Constraint

We now consider the parallel decomposable plan of a transitive closure query, which does not contain any aggregates in recursion. We use the same non-linear recursive example from (Yang et al. 2017), given by rules $r_{7.1} - r_{7.3}$, which shows the program executed by

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

worker i. Note, in this example every worker i eventually has to compute and send to other workers all `tc` atoms of the form `(X, Y)`, where `h(X)` $= i$. Without $\mathcal{P}reM$, a worker i does not update its existing `tc` atoms. In fact, during semi-naive evaluation of this query, at any time, only new unique atoms are appended to `tc`. Thus, a SSP evaluation for the transitive closure query (without $\mathcal{P}reM$) does not save any communication cost as compared to a BSP model. However, as shown in our experimental results next, the SSP model can still mitigate the influence of stragglers.

$$r_{7.1} : \texttt{tc(X,Y)} \texttt{ <- } \texttt{arc(X,Y)}, \texttt{h(X)} = \texttt{i}.$$
$$r_{7.2} : \texttt{tc(X,Y)} \texttt{ <- } \texttt{tc(X,Z)}, \texttt{tc}^{(1)}\texttt{(Z,Y)}, \texttt{h(X)} = \texttt{i}.$$
$$r_{7.3} : \texttt{tc}^{(1)}\texttt{(X,Y)} \texttt{ <- } \texttt{tc(X,Y)}, \texttt{h(X)} = \texttt{i}.$$

6.2 Experimental Results

Setup. We conduct our experiments on a 12 node cluster, where each node, running on Ubuntu 14.04 LTS, has an Intel i7-4770 CPU (3.40GHz, 4 cores) with 32GB memory and a 1 TB 7200 RPM hard drive. The compute nodes are connected with 1Gbit network. Following the standard practices established in (Shkapsky et al. 2016), (Yang et al. 2017), we execute the distributed bottom-up semi-naive evaluation using an AND/OR tree based implementation in Java on each node. Each node executes one application thread per core. We evaluate both the non-linear all pairs shortest path and transitive closure queries on a subset of the real world *orkut* social network data[5].

Inducing Stragglers. In order to study the influence of straggling nodes in a declarative recursive computation, we induce stragglers in our implementation following the strategy described in (Cui et al. 2014). In particular, each of the nodes in our setup can be disrupted independently by a CPU-intensive background process that kicks in following a Poisson distribution and consumes at least half of the CPU resources.

Analysis. In this section, we empirically analyze the merits and demerits of a SSP model over a BSP model, by examining the following questions: (1) How does a SSP model compare to a BSP model when queries contain $\mathcal{P}reM$ constraints and aggregates in recursion? (2) How do these two processing paradigms compare when $\mathcal{P}reM$ cannot be applied? (3) And, how do the overall performances in the above scenarios change in presence and absence of stragglers? Table 1 captures the first case with the all pairs shortest path query (where $\mathcal{P}reM$ is applicable), while Table 2 presents the second case with the transitive closure query, which do not contain any aggregates or $\mathcal{P}reM$ constraints in recursion. For each of these two cases, as shown in the tables, we experimented with two different staleness values for a SSP model, both under the presence and absence of induced stragglers. Notably, a SSP model with bounded staleness (alternatively also called 'slack' and indicated by s in the tables) set as zero reduces to a BSP model. Tables 1 and 2 capture the average execution time for the query at hand under different configurations over five runs. This *run time* can be divided into two components— (1) *average computation time*, which is the average time spent by the workers performing semi-naive evaluation for the

[5] http://snap.stanford.edu/data/com-Orkut.html

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

recursive computation, and (2) *average waiting time*, which is the average time spent by the workers waiting to receive a new update to resume computation. Tables 1 and 2 show the run time break down for the two aforementioned cases (with and without $\mathcal{P}reM$ respectively).

From Tables 1 and 2, it is evident that BSP processing requires the least compute time irrespective of straggling nodes. This is also intuitively true because the total recursive computation involved in a BSP based distributed semi-naive evaluation is similar to that of a single executor based sequential execution and as such a BSP model should require the least computational effort to reach the minimal fixpoint. On the other hand, a SSP model may perform many local computations optimistically with obsolete data using relaxed synchronization barriers, which can become redundant later on. As shown in the tables, average compute time indeed increases with higher slack indicating that a substantial amount of the work becomes unnecessary. However, as seen from both the tables, SSP plays a major rule in reducing the average wait time. This is trivially true, since in SSP processing, any worker can move ahead with local computations using stale knowledge, instead of waiting for global synchronization as required in BSP. However, note the reduction in average wait time under SSP model in Table 1 (with $\mathcal{P}reM$) is more significant than in Table 2 (without $\mathcal{P}reM$). This can be attributed to the fact that $\mathcal{P}reM$ with semi-naive evaluation (Section 2) under SSP model can batch multiple updates together before sending them, thereby saving communication cost. However, for the transitive closure query (without $\mathcal{P}reM$), the overall updates sent in BSP and SSP models are similar (since no aggregates are used, semi-naive evaluation only produces new atoms, never updates existing ones). Thus, in the latter case (Table 2), the wait times between BSP and SSP models are comparable when there are no induced stragglers, whereas the wait time in SSP is marginally better than BSP when stragglers are present. Notably, inducing stragglers obviously increases the average wait time all throughout as compared to a no straggler situation. The compute time also increases marginally in presence of stragglers, primarily because the straggling nodes take longer time to finish its computations.

Thus, to summarize based on the run times in the two tables, we see that in absence of stragglers, the SSP model can reduce the run time of the shortest path query (with $\mathcal{P}reM$ constraint) by nearly 30%. However, the same is not true for the transitive closure query, which do not have any $\mathcal{P}reM$ constraint. Hence, a BSP model would suffice if there are no stragglers and the query does not contain any $\mathcal{P}reM$ constraint. However, in presence of stragglers or $\mathcal{P}reM$ constraints, SSP model turns out to be a better alternative than BSP model, as it can lead to a execution time reduction of as high as 40% for the shortest path query and nearly 7% for the transitive closure query. Finally, it is also worth noting from the results that too much of a slack can also increase the query latency. Thus, a moderate amount of slack should be used in practice.

7 Conclusion

$\mathcal{P}reM$ facilitates and extends the use of aggregates in recursion, and this enables a wide spectrum of graph and data mining algorithms to be expressed efficiently in declarative languages. In this paper, we explored various improvements to scalability via paralled execution with $\mathcal{P}reM$. In fact, $\mathcal{P}reM$ can be easily integrated with most of the

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

Table 1. *Comparing BSP vs. SSP model for all pairs shortest path query containing aggregates in recursion (with $\mathcal{P}reM$).*

Time consumption	No stragglers (time in sec)			With stragglers (time in sec)		
	BSP (s=0)	SSP (s=3)	SSP (s=6)	BSP (s=0)	SSP (s=3)	SSP (s=6)
Avg. compute time	**2224**	2443	3038	**2664**	2749	3435
Avg. wait time	1679	**302**	**408**	2786	**485**	**704**
Run time	3903	**2745**	**3446**	5450	**3234**	**4139**

Table 2. *Comparing BSP vs. SSP model for transitive closure query containing* no *aggregates in recursion (without $\mathcal{P}reM$).*

Time consumption	No stragglers (time in sec)			With stragglers (time in sec)		
	BSP (s=0)	SSP (s=3)	SSP (s=6)	BSP (s=0)	SSP (s=3)	SSP (s=6)
Avg. compute time	**682**	762	879	**754**	827	921
Avg. wait time	367	**345**	**334**	618	**456**	**412**
Run time	**1049**	1107	1213	1372	**1283**	1431

current generation Datalog engines like *BigDatalog*, *Myria*, *BigDatalog-MC*, *SociaLite*, *LogicBlox*, irrespective of their architecture differences and varying synchronization constraints. Moreover, in this paper, we have shown that $\mathcal{P}reM$ brings additional benefits to the parallel evaluation of recursive queries. For that, we established the necessary theoretical framework that allows bottom-up recursive computations to be carried out over stale synchronous parallel model—in addition to the synchronous or completely asynchronous computing models studied in the past. These theoretical developments lead us to the conclusion, confirmed by initial experiments, that the parallel execution of non-linear queries with $\mathcal{P}reM$ constraints can be expedited with a stale synchronous parallel (SSP) model. This model is also useful in the absence of $\mathcal{P}reM$ constraints, where bounded staleness may not reduce communications, but it nevertheless mitigates the impact of stragglers. Initial experiments performed on a real-world dataset confirm the theoretical results, and are quite promising, paving the way toward future research in many interesting areas, where declarative recursive computation under SSP processing can be quite advantageous. For example, declarative advanced stream reasoning systems (Das et al. 2018), supporting aggregates in recursion, can adopt distributed SSP model to query evolving graph data, especially when one portion of the network changes more rapidly as compared to others. SSP models under such scenario offer the flexibility to batch multiple network updates together, thereby reducing the communication costs effectively.

Finally, it is important to note that the methodologies developed here can also be applied to other declarative logic based systems beyond Datalog, like in SQL-based query engines (Gu et al. 2019), which also use semi-naive evaluation for recursive computation. In addition, the SSP processing paradigm can also be adopted in many state-of-the-art graph-centric platforms such as Pregel (Malewicz et al. 2010) and GraphLab (Low et al. 2012). These modern graph engines use a vertex-centric computing model (Yan et al.

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

2015), which enforces a strong consistency requirement among its model variables under the "Gather-Apply-Scatter" abstraction. Consequently, this makes the synchronization cost for these graph frameworks similar to that of standard BSP systems. Thus, for many distributed graph computation problems involving aggregators (like shortest path queries), SSP model, as demonstrated in this paper, can be quite useful for these graph based platforms.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068419000358.

References

Ameloot, T. J. 2014. Declarative networking: Recent theoretical work on coordination, correctness, and declarative semantics. *SIGMOD Rec. 43,* 2, 5–16.

Ameloot, T. J., Geck, G., Ketsman, B., Neven, F., and Schwentick, T. 2017. Parallel-correctness and transferability for conjunctive queries. *J. ACM 64,* 5, 36:1–36:38.

Ameloot, T. J., Ketsman, B., Neven, F., and Zinn, D. 2015. Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the calm-conjecture. *ACM Trans. Database Syst. 40,* 4, 21:1–21:45.

Ameloot, T. J., Neven, F., and Van Den Bussche, J. 2013. Relational transducers for declarative networking. *J. ACM 60,* 2, 15:1–15:38.

Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. 2010. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. 265–278.

Aref, M., ten Cate, B., Green, T. J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. L., and Washburn, G. 2015. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1371–1382.

Beckman, P., Iskra, K., Yoshii, K., and Coghlan, S. 2006. The influence of operating systems on the performance of collective operations at extreme scale. In *2006 IEEE International Conference on Cluster Computing*. 1–12.

Cipar, J., Ho, Q., Kim, J. K., Lee, S., Ganger, G. R., Gibson, G., Keeton, K., and Xing, E. 2013. Solving the straggler problem with bounded staleness. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*. HotOS'13. 22–22.

Condie, T., Das, A., Interlandi, M., Shkapsky, A., Yang, M., and Zaniolo, C. 2018. Scaling-up reasoning and advanced analytics on bigdata. *TPLP 18,* 5-6, 806–845.

Cui, H., Cipar, J., Ho, Q., Kim, J. K., Lee, S., Kumar, A., Wei, J., Dai, W., Ganger, G. R., Gibbons, P. B., Gibson, G. A., and Xing, E. P. 2014. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*. 37–48.

Das, A., Gandhi, S. M., and Zaniolo, C. 2018. Astro: A datalog system for advanced stream reasoning. In *CIKM'18*. 1863–1866.

Das, A. and Zaniolo, C. 2019. A case for stale synchronous distributed model for declarative recursive computation. *CoRR abs/1907.10278*.

Ganguly, S., Silberschatz, A., and Tsur, S. 1992. Parallel bottom-up processing of datalog queries. *J. Log. Program. 14,* 1-2, 101–126.

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press

Gu, J., Watanabe, Y., Mazza, W., Shkapsky, A., Yang, M., Ding, L., and Zaniolo, C. 2019. Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark. In *SIGMOD'19*.

Ho, Q., Cipar, J., Cui, H., Kim, J. K., Lee, S., Gibbons, P. B., Gibson, G. A., Ganger, G. R., and Xing, E. P. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*. 1223–1231.

Interlandi, M. and Tanca, L. 2018. A datalog-based computational model for coordination-free, data-parallel systems. *Theory and Practice of Logic Programming 18,* 5-6, 874–927.

Krevat, E., Tucek, J., and Ganger, G. R. 2011. Disks are like snowflakes: No two are alike. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. HotOS'13. 14–14.

Lee, S., Kim, J. K., Zheng, X., Ho, Q., Gibson, G. A., and Xing, E. P. 2014. On model parallelization and scheduling strategies for distributed machine learning. In *NIPS*. 2834–2842.

Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. 2012. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5,* 8, 716–727.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. 2010. Pregel: A system for large-scale graph processing. In *SIGMOD'10*. 135–146.

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

Seo, J., Park, J., Shin, J., and Lam, M. S. 2013. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow. 6,* 14, 1906–1917.

Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., and Zaniolo, C. 2016. Big data analytics with datalog queries on spark. In *SIGMOD*. ACM, New York, NY, USA, 1135–1149.

Wang, J., Balazinska, M., and Halperin, D. 2015. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *Proc. VLDB Endow. 8,* 12, 1542–1553.

Yan, D., Cheng, J., Lu, Y., and Ng, W. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*. 1307–1317.

Yang, M., Shkapsky, A., and Zaniolo, C. 2017. Scaling up the performance of more powerful datalog systems on multicore machines. *VLDB J. 26,* 2, 229–248.

Zaniolo, C., Yang, M., Das, A., and Interlandi, M. 2016. The magic of pushing extrema into recursion: Simple, powerful datalog programs. In *AMW*.

Zaniolo, C., Yang, M., Interlandi, M., Das, A., Shkapsky, A., and Condie, T. 2017. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *TPLP 17,* 5-6, 1048–1065.

Zaniolo, C., Yang, M., Interlandi, M., Das, A., Shkapsky, A., and Condie, T. 2018. Declarative bigdata algorithms via aggregates and relational database dependencies. In *AMW*.

https://doi.org/10.1017/S1471068419000358 Published online by Cambridge University Press