A Scheduling Approach to Incremental Maintenance of Datalog Programs

Shikha Singh*, Sergey Madaminov†, Michael A. Bender†, Michael Ferdman†,
Ryan Johnson¶, Benjamin Moseley‡, Hung Ngo‖, Dung Nguyen§,
Soeren Olesen§, Kurt Stirewalt‖, and Geoffrey Washburn§.
*Williams College, Williamstown, MA 01267 USA shikha@cs.williams.edu
†Stony Brook University, Stony Brook, NY 11794-2424 USA
{smadaminov, bender, mferdman}@cs.stonybrook.edu
‡Carnegie Mellon University, Pittsburgh, PA 15213 USA moseleyb@andrew.cmu.edu
§Infor, Inc. {dung.nguyen, soeren.olesen, geoffrey.washburn}@infor.com
¶Amazon, Inc., frj@amazon.com
‖Relational AI {hung.ngo, kurt.stirewalt}@relational.ai

***Abstract*—In this paper, we study the problem of incremental maintenance of Datalog programs and model it as a scheduling problem on DAGs. We design provably good time- and memory-efficient scheduling algorithms for (re)executing a Datalog program where some (but not necessarily all) of the inputs have changed. We prove that our schedulers, called LevelBased and LevelBased with lookahead, have asymptotically improved running time and space efficiency when compared with benchmark algorithms used in production at LogicBlox.**

The main result of the paper is a hybrid scheduler, which combines LevelBased with the production LogicBlox scheduler (or any other heuristic scheduler). The hybrid scheduler achieves strong worst-case guarantees and robustness without losing out on the best-case behavior of the production LogicBlox scheduler. Our experiments show that the hybrid scheduler results in similar or improved total execution times compared to LogicBlox scheduler, while consistently reducing the scheduling overhead—by as much as 50% on some datasets. This hybrid scheme requires little to no overhead but provides predictability and reliability, which are crucial in a commercial application such as LogicBlox.

***Index Terms*—Datalog programs, incremental maintenance, DAG scheduling, parallel task scheduling, databases, incremental computing, LogicBlox.**

I. Introduction

One of the universal problems in computer science is how to update a computation efficiently when the input changes in some way. Specifically, the challenge is to determine how to avoid redoing those parts of the computation that have not been affected by the modified input. This problem, often referred to as ***incremental computing***, has been studied in various guises in the field of programming languages [12], [16], [24], [32], [35], systems [9], [10], algorithms [13], [15], [17], [26], [29], and databases [5], [8], [22], [33].

In this paper, we study an instance of incremental computing in databases, specifically, in the context of Datalog programs. Datalog [23] is a widely-used declarative programming language based on logic programming. It allows for expressing recursive dependencies, and it streamlines the implementations of complex queries [21]. Datalog is implemented in modern databases systems such as Semmle [1], Soufflé [2], and LogicBlox [20]. In this paper, we focus on LogicBlox, a commercial Datalog implementation, which uses incremental computation to support a suite of data mining and machine learning tools for retail.

Efficient incremental computing can improve the performance of user queries substantially in these database systems. Queries in Datalog-based system are answered by checking them against the stored dataset of all facts that can be derived from the Datalog rules. When the base data is updated or the rule definitions change, these derived facts need to be updated as well to answer the queries consistently. Provably-efficient guarantees for incremental computing have crucial commercial value for LogicBlox as its customer base relies on the ability to issue updates to the database with the expectation that its queries can still be answered quickly [7].

The algorithmic challenge involved in incrementally maintaining Datalog programs arises from the dynamic nature of their execution—Datalog programs contain recursive declarations of rules and changing a few base predicates has a cascading effect down the dataflow graph. Moreover, it is unclear what parts of the graph are affected since changes to the inputs to the rules may or may not affect their output. The output of these rules in turn determine the input to other rules, and so on.

Thus, to maintain these programs incrementally, we need to determine: (a) what parts of the dataflow graph have been affected, and (b) in what order should the affected rules be updated to avoid repeated computation. The materialization of the recursive rules of a Data-

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00093

log program is represented as a directed acyclic graph (DAG).

The nodes of the DAG are tasks that may need to be executed, and the edges encode dependencies between tasks. Changes to the program dirty/activate the source nodes (tasks with no input from other tasks). The source nodes are then rerun and each output that changes activates those children that depend on it, and so on, recursively. An activated node should only be rerun once all its inputs are up-to-date, because we do not want to run a node more than once (to avoid unnecessary computation).

The size of these DAGs can be enormous and so avoiding recomputation can significantly conserve resources. For example, Figure 1 shows a DAG used in production that is so large that if it were printed at a resolution of 300 DPI would be a mile long.

To address the problem of incrementally maintaining these DAGs the engineers at LogicBlox have designed a scheduler that determines which tasks from the underlying DAG need to be rerun after an update. While the scheduler does well on most workloads, it does not have strong worst-case guarantees. In particular, there exist pathological DAGs based on real traces, on which the LogicBlox scheduler takes too long—making scheduling the bottleneck cost, restricting performance.

Inspired by the practical concerns faced by LogicBlox, in this paper we give a theoretical framework to study the incremental maintenance problems of Datalog programs. Furthermore, we design a scheduler that has provable guarantees on its memory and running time.

Before we describe our main contributions, we explain the difference between scheduling DAGs resulting out of Datalog programs and standard DAG scheduling.

Datalog DAG scheduling versus traditional DAG scheduling. A key feature of the problem is the non-local precedence relationships between tasks, which dynamically change over time. In the Datalog scheduling problem, the readiness of a task cannot be determined locally. Rather, it requires computing ancestor relationships that dynamically change over time. Due to the size of the DAGs, computing these relationships in a way that is memory and time efficient is a central algorithmic challenge. In contrast, in most precedence-constrained scheduling problems the scheduler can easily determine the nodes that are ready to be executed.

This algorithmic issue that arises in the model of incremental maintenance of Datalog programs presents a new scheduling challenge distinct from most precedence constrained models (e.g., parallel job scheduling [18], [19], [27], job-shop scheduling [6], [30], assembly line scheduling [25], [34]), etc. See [28] for reference.

Results. In this paper, we give time- and memory-efficient scheduling algorithms in support of incremental maintenance of Datalog programs.

We start by formalizing a theoretical model for the problem of maintaining Datalog programs. We then design a provably good scheduler, the LevelBased scheduler, to (re)execute a Datalog program where some (but not necessarily all) of the inputs have changed. We prove that the LevelBased scheduler has asymptotically better worst-case memory efficiency and running time over the LogicBlox scheduler.

We test the empirical performance of the two schedulers on workloads seen in production at LogicBlox. We show that the LogicBlox scheduler does have good typical performance but it suffers on worst-case instances. In contrast, the LevelBased scheduler outperforms the LogicBlox scheduler on worst-case instances, and its typical performance is reasonable, though dominated by the LogicBlox scheduler.

As our main result, we present a hybrid scheduling scheme that achieves the best of both worlds. In particular, in Section V, we prove that the LevelBased scheduler can be effectively combined with *any* existing heuristic like the LogicBlox scheduler, to achieve strong worst-case guarantees, while retaining their best-case performance on typical workloads.

The experiments back up the theoretical result and show that the hybrid scheduler results in similar or improved total execution times compared with LogicBlox scheduler, while consistently reducing the scheduling overhead—by as much as 50% on some datasets. We emphasize that the hybrid scheme requires little to no overhead (the LevelBased scheduler is lightweight) but provides predictability and reliability, which are crucial in a commercial application such as LogicBlox.

II. Formalization of Incremental Maintenance of Datalog Programs

In this section we formalize the model of incremental maintenance of Datalog Programs and introduce terminology and notation used in the rest of the paper.

A. Datalog Programs and the Activate Graph

A Datalog program can be expressed as a directed-acyclic-graph (DAG) $G = (V, E)$ of precedence-constrained vertices (predicate nodes). A vertex in V is a subcomputation that (1) evaluates the predicate based on input to the subcomputation and (2) returns an output that is used by predicates later in the DAG. Each edge $(u, v) \in E$ corresponds to output from u's computation being input to v's computation. Thus, the incoming and outgoing edges determine the flow of data through the graph and explain why there are precedence constraints. To simplify exposition, we V and E to denote $|V|$ and $|E|$ in the asymptotic notation.

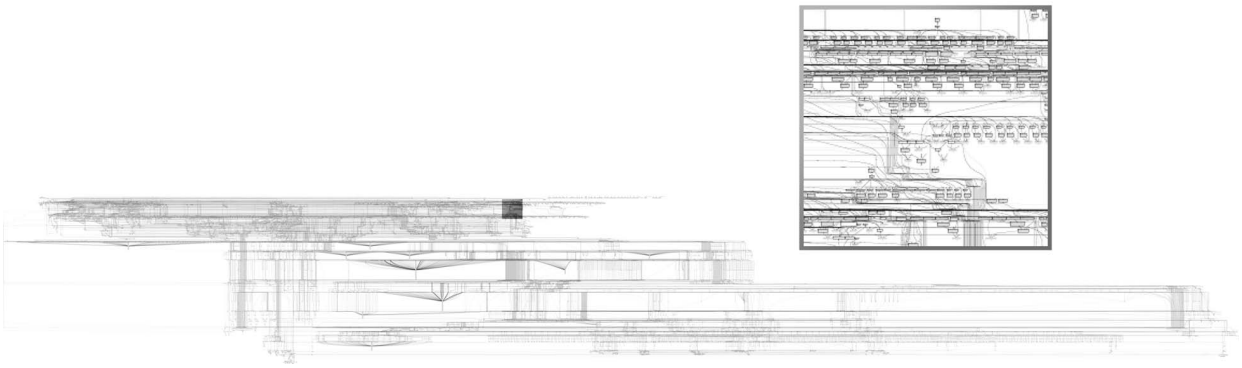

Figure 1: An example of a computational DAG representing dataset #1 in Table I. The DAG consists of 64,910 vertices (predicate nodes) and 101,327 edges (dependencies). There are 20,134 nodes that correspond to tasks that can be activated and the remaining nodes are predicate nodes used to collect inputs and outputs for the nodes. The scheduling starts with updates to five initial tasks, and these changes cascade and result in the activation of 532 descendants, out of 1680 total descendants. Notice that most of the descendants do not need to be recomputed. The challenge is determining which do and their dependency relationship.

Vertices in V with indegree 0 are called ***source nodes***. These nodes represent the data of the database. Over time the database's data is updated, which means the outputs of the source nodes change. These changes can propagate through some (but often not all) nodes of the graph. That is, just because an input to a predicate changes, does not mean that the predicate's output changes.

A node u is said to be ***activated*** if the input to u changes. The **active graph** $H = (W, F)$, where $W \subseteq V$ is set of the activated nodes and $F \subseteq E$ is the set of all the edges (u, v) such that u sends new output to v.

The active graph H comprises those nodes that need to be recomputed and is dynamically revealed over time as the nodes are executed and their outputs are updated. Notice that H is not the induced subgraph $G[W]$. For example, consider active nodes u and v such that (u, v) is an edge in G. It is not necessary that (u, v) is an edge in H, as the output from u to v may not change.

What makes this optimization problem so unusual is that the precedence relationship between nodes in H can be radically different from their relationship in G. In particular, activated nodes u and v may be independent in H but there may be a path from u to v in G.

When a node u is first activated, it may not be ready to run. The constraint is that repeated executions of u are disallowed: we cannot kick off an execution of u until we know that it won't need to be reexecuted, that is, until the changes to all of u's inputs are revealed. Thus, the node u is ***ready to run*** only when all of its activated ancestors have been executed.

B. Formal Results: Guarantees for the Scheduler

In this paper, we design a scheduling algorithm that determines: (a) which predicate nodes need to be recomputed as a result of changes to the source nodes, and (b) how to assign these nodes to processors.

To formally define the guarantees we establish more terminology. Given a DAG, a ***path*** between two distinct nodes is a sequence of edges connecting them. Given a DAG, each node has a ***level***, which is the maximum length (number of nodes minus one) of any path from any source node to that node. Source nodes are defined to have level 0. Throughout the paper, we will use level to refer to the original graph G.

Our scheduler is efficient: the running time of the scheduler is independent of the number of edges in G— it only depends on the number of active nodes $n = |W|$ (not the total number of nodes $|V$—) and the ***number of levels*** L in the original DAG G. In particular, the rescheduling cost is $O(n + L)$.

The resulting schedule is also efficient. Let $\boldsymbol{w}$ be the ***total work*** of all the activated nodes and $\boldsymbol{P}$ be the ***number of cores*** available.

For the case of unit-length or fully parallelizable jobs, the makespan of the resulting schedule is $O(w/P + L)$. In the common case when the computation is ***work dominated***, that is, when $w/P \geq L$, the makespan of our schedule is a 2-approximation. This is the case that we want to optimize for in multithreaded programs. For arbitrary jobs, the makespan of our schedule is $O(w/P+C)$, where C is the ***critical path*** of the DAG G. Again, the resulting schedule is efficient in the common case when the computation is work-dominated.

Moreover, our scheduler can be used in conjunction with the LogicBlox scheduler, to get the best of both worlds. The LogicBlox scheduler has no performance guarantees because it explicitly checks for all precedence constraints during runtime. It is feasible to interleave

steps of our scheduler and theirs. This hybrid scheduler inherits both our performance guarantees and the good cases for the LogicBlox scheduler. That is, the cases where their scheduler does well, it still does. For the cases where it does badly, our scheduler dominates and comes to the rescue. Thus, adding our new scheduler only results in performance improvements.

The scheduler is also *memory efficient*. The scheduler does not require a sophisticated data structure. Rather, the scheduler only needs to store one number for each node in G, the node's level in the original DAG.

C. Comparison to Baseline Solutions

In this section, we discuss the state-of-the-art solutions, related work and natural strategies for solving the incremental maintenance problem studied in this paper.

LogicBlox scheduler. The current approach used in LogicBlox is to precompute and store *all ancestors* of nodes in the underlying DAG. These ancestor relationships are heuristically maintained using a data structure called an ***interval list*** [4], [31]. The interval list is, usually but not always, compact way of encoding ancestor relationships using intervals, generated based on DFS traversals of the DAG. In the worst case, storing all ancestor relationships using interval lists takes $O(V^2)$ space, but the average case seen in practice is often much better.

The LogicBlox scheduler can be computationally and memory intensive. To see why, consider the following actions performed by the scheduler. When the queue of tasks that are ready to run is empty, the scheduler must locate more ready work (tasks with no active ancestors). To do so, the scheduler scans the queue of active tasks. For each one, it checks the interval-list data structure to see if the task is ready to run. If so, it is added to the queue of ready work.

Let's analyze the worst case. First, the space required to store ancestor relations can be large—quadratic in the number of nodes, V^2, in the original DAG. Second, the worst-case running time of the scheduler is cubic in the number of active nodes, i.e., $O(n^3)$. This is because in order to check whether a task is ready to run, it may need to check whether any of the $O(n)$ active nodes are its ancestors. An interval-list query is constant time in the best case and $O(n)$ time in the worst case. We may need to perform ancestor queries $O(n)$ times to find a node that is ready to run, i.e., without active ancestors.

Brute-force signal propagation. Next, we describe another natural baseline scheduler, which is asymptotically slower because its running time depends on the size of the original DAG (i.e., V and E) rather than just the active nodes (i.e. n). The scheduler starts by reexecuting the source nodes. Depending on their output, it propagates appropriate signals (such as "no change to output," or "this is the new output") to all their children. Only when a node receives a signal from all its parents, is it marked ready to run or inactive. After a task is rerun or marked as inactive it sends a message to all its children.

The running time of this scheduler is $O(V + E)$ since the signals get propagated through the entire DAG. Thus, the number of messages is large, regardless of the number of active nodes. Moreover, under this schedule tasks that are ready to run may have to wait a long time before the scheduler discovers that they can be safely scheduled, because a node has to wait for all ancestors to send a signal rather than just the active ancestors.

Our approach compared to baseline solutions. On the one hand, the Logicblox scheduler does a lot of potentially unnecessary precomputations which allows it to make faster scheduling decisions during run time (decisions that only depend on the set of active nodes and thus have guarantees independent of the underlying DAG). On the other hand, brute-force signal propagation does no precomputation at all but spends a lot of time making scheduling decisions during run time by sending potentially unnecessary messages throughout the DAG.

In this paper, we present a level-based scheduler which strikes a balance between the two extremes. In particular, we do some precomputation ($O(V+E)$ time and $O(V)$ space) to determine the levels of the tasks. During runtime, the scheduler uses the levels of the tasks to figure out when an active tasks is ready to run. Unlike signal propagation, the running time of our scheduler does not depend on the size of the original DAG.

III. Level Scheduling Algorithm

In this section, we describe our LevelBased scheduling algorithm and establish its performance guarantees.

The crux of the algorithm is determining which tasks are ***ready to run***, from among all the tasks that are currently active. A ready-to-run task is an active task all of whose ancestors were already activated and recomputed or will never be activated. Once the algorithm has identified ready-to-run tasks, it still needs to assign the tasks to processors, and this is done greedily.

LevelBased scheduling algorithm. Before the algorithm runs, there is a precomputation step: we precompute levels for each node of G, where the level of a node u is the maximum number of edges along any path from any source node to u.

At time t during the execution, let ℓ be the lowest level among nodes in the active set. Then any task at level ℓ is ready to run. The scheduler picks these ready-to-run tasks in a greedy level-based manner as follows:

- When a processor is idle, it removes and processes any task from level ℓ.

- If all processors are idle and level ℓ is empty, then ℓ is incremented.

The LevelBased algorithm identifies tasks that are ready to be scheduled. The criteria for identifying a ready tasks is that all tasks with lower levels have been completed. The method is oblivious to how those tasks were completed and, therefore, LevelBased can be run along side any scheduling algorithm, as explained in Section V.

Extending the algorithm. The fundamental limitation of LevelBased scheduling approach is that it does not proceed further down the levels until it finishes processing all the tasks on a current level. However, this limitation is not an issue when tasks exhibit internal parallelism, as is the case in practical workloads, such as those seen at LogicBlox. Yet, to address that problem we introduce a look-ahead heuristic, LevelBased with LookAhead (LBL), that searches for more tasks to process on next level and simulations show that it yields good performance gains in case of tasks without internal parallelism.

While current industrial solutions demonstrate good performance, the amount of data is growing at a fast pace, making databases larger. In the next section, we prove that LevelBased maintains good theoretical-performance and scalability guarantees as opposed to the LogicBlox scheduler. Moreover in Section V, we advocate that the LevelBased scheduling approach is orthogonal to the existing heuristics—it can be used in coordination with them to provide space and makespan guarantees on worst-case instances, without losing out on the performance on the typical case.

IV. LevelBased Analysis

This section gives an analysis of LevelBased scheduler on different workloads. We prove different guarantees of the scheduler based on the size and parallelizability of the tasks in the input DAG.

The following lemma shows how ready-to-run tasks can be identified based on their levels and will be used throughout the analysis.

Lemma 1. *Fix a time and consider the set of ready-to-run tasks (which includes tasks that are currently running). Suppose that the lowest level among these tasks is ℓ. Then any other active task with level ℓ is ready to run.*

Proof. Consider an active task x at level ℓ. Suppose x cannot be added to the ready-to-run set. This means that there is some node y that has been activated such that level$(y) <$ level(x), y has not been run, but is ready-to-run. This contradicts the definition of x. □

Theorem 2. *On any arbitrary workload, the LevelBased scheduler has the following guarantees.*

- *The total precomputation time and space is $O(V + E)$ and $O(V)$ respectively.*
- *The running time and space of the scheduler is $O(n + L)$ and $O(n)$ respectively.*

Proof. The precomputation time of the LevelBased scheduler is the time required to compute the levels of all the nodes in the DAG. The levels can be computed using depth-first search in $O(V+E)$ time. The precomputation space is the space required for each node to store its level, i.e., $O(V)$. Given a sequence of active and ready-to-run tasks and their levels, the scheduler does the following at each level (a) finds the lowest level of ready-to-run tasks, and (b) marks active tasks ready-to-run based on their level. This proceeds for at most $\sum_{\ell \in L} |\{v \mid v \text{ is an active node at level } \ell\}| = O(n+L)$ steps, as each node has a unique level. As each step costs $O(1)$, the scheduler runs in $O(n + L)$ time. Marking active tasks ready-to-run requires $O(n)$ space. □

Unit-length tasks. We first analyze the case when all tasks have unit size.

Lemma 3. *When tasks have unit-length, the makespan guaranteed by the LevelBased is at most $w/P + L$.*

Proof. At any time step t, the LevelBased scheduler is in one of the following two cases: (a) all P processors are busy executing tasks, or (b) some processor is idle. Case (a) can happen at most w/P times before all the work is done.

Case (b) can happen L times (recall that the tasks are unit length). This is because, by Lemma 1, all tasks at the lowest level ℓ among ready-to-run tasks can be safely executed. Thus, if a processor is idle, all work at level ℓ must be done, which is when ℓ increments, and ℓ can increment at most L times.

Thus, the makespan is at most $w/P + L$. □

Fully parallelizable tasks. In this section, we bound the performance of the LevelBased scheduling algorithm when tasks can have arbitrary sizes, but are guaranteed to be fully parallelizable. To do so, the analysis of the unit-time tasks is generalized.

The specific model used to model such tasks is the Directed-Acyclic-Graph (DAG) model of computation [11]. In this model, each task u is a parallel program modeled by a DAG D_u. Each DAG D_u has a set of ***subtasks*** V_u that are unit time pieces of work to be scheduled. The edges represent precedence constraints. A task can be scheduled when all of its predecessors have been completed.

First, we prove the performance of our scheduler when the DAGs of each task are fully parallelizable. That is, there are no precedence constraints and any number of tasks in D_u can be scheduled simultaneously. Later, we prove gaurantees on arbitrary DAGs.

Definition 4. *We define different types of span:*

- ***level span*** $\mathbf{L}$*: the maximum number of levels in* G*.*
- ***task span*** $\boldsymbol{S^T}$*: the span inside of a task (which depends on how parallelizable that task is).*
- ***realized span*** $\boldsymbol{S}$*: the optimal execution time of* H *on an infinite number of processors.*

The next lemma gives the guarantees of the level scheduling algorithm for the case of arbitrary-length fully-parallelizable tasks.

Lemma 5. *Suppose that all tasks have arbitrary length and are fully parallelizable. Then, the LevelBased scheduler guarantees makespan at most* $w/P + L$*.*

Proof. The proof is similar to that of Lemma 3. At any time step t, we are in one of the following two cases: (a) all P processors are busy executing tasks, or (b) some processor is idle. Case (a) can happen at most w/P times before all the work is done.

Suppose we are in case (b) at any time t, that is, at time t some processors are idle waiting for a task on level ℓ to finish. Then, because the jobs are fully parallelizable, we are guaranteed to move on level $\ell + 1$ at time step $t + 1$. This can happen at most L times. Thus, the makespan is at most $w/P + L$. □

Arbitrary-length jobs. We now consider the case where tasks have arbitrary length that are not necessarily parallelizable. As before, we assume that each task is represented as a DAG. Now the DAGs for the tasks can be arbitrary; we define their level span as follows.

Definition 6. ***Span at level i****, denoted* S_i*, is the maximum task span among all tasks at level* i*, where* $1 \leq i \leq L$*.*

Lemma 7. *Suppose all the tasks have arbitrary length and parallelism, then the LevelBased scheduler guarantees makespan at most* $(w/P + \sum_{i=1}^{L} S_i)$*.*

Proof. The proof is similar to that of Lemma 3 and Lemma 5. At any time step t, we are in one of the following two cases: (a) all P processors are busy executing tasks, or (b) some processor is idle. Case (a) can happen at most w/P times before all the work is done. Suppose we are in case (b) at any time t, that is, at time t some processors are idle waiting for a task on level ℓ to finish. Then in S_i time steps, we are guaranteed to move on level $\ell+1$. This can happen at most L times. Thus, the makespan is at most $(w/P + \sum_{i=1}^{L} S_i)$. □

Remark 8. *Lemma 7 implies that the LevelBased scheduler performs well on arbitrary tasks that is achieves a makepsan of* $O(w/P)$*, as long as the first term dominates, that is,* $P \leq w/(\sum_{i=1}^{L} S_i)$*.*

Tight examples. We show that the analysis of the makespan produced by our algorithm is tight in all cases.

For unit-length and fully-parallelizable tasks, it is well-known that a makespan of $(w/P + L)$ is tight [14] for list scheduling algorithms. For jobs that are arbitrary length and not fully parallelizable, we give an example that match the bounds of the LevelBased algorithm.

Theorem 9. *The analysis of the makespan of the LevelBased algorithm is tight. For the case where the jobs have arbitrary size and parallelism, there is an example where the makespan is* $\Omega(w/P + \sum_{i=1}^{L} S_i)$*.*

Proof. We show that if the jobs are not fully parallelizable and have arbitrary length, the LevelBased algorithm can have poor behavior in situations like the one depicted in Figure 2. In the depicted example, let $M = \max_{v \in V} S_v^T$, where S_v^T is the task span of a task v. We will show that the algorithm's makespan can be as poor as $\Theta(ML)$, while the optimal solution's makespan is bounded by $\Theta(M+L)$. The example assumes $M \leq P$, the number of processors.

In the example there are two types of tasks. There are L tasks $j_1, j_2, j_3, \ldots j_L$. Each task j_i cannot be executed until task j_{i-1} is completed and these are unit-length tasks with no parallelism. Additionally there is a task k_i for $i = 2, \ldots, L$. The task k_i cannot be executed until j_{i-1} is completed. The task k_i has work and span $L - i + 1$. See Figure 2 for the description of this graph.

An optimal scheduler would execute the jobs in the order $j_1, k_2, j_2, k_3, j_3, k_4 \ldots, j_L, k_L$. Each time a task k_i becomes free, a processor begins working on the task until it completes. Since there are $L - 1 \leq P$ such tasks, there is always a processor available. The overall running time is $\Theta(M + L)$.

The level-based scheduling algorithm gets stuck at each level running the long job before processing any job at the next level. Indeed, it first schedules job j_1. Then in the next step it will schedule j_2 and k_2 to completion. Next jobs j_3 and k_3 to completion and so forth for each level. The overall running time will be bounded by $\Theta(ML)$.

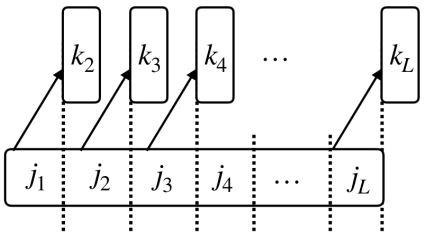

Figure 2: Tight example for the case when jobs are arbitrary length and not fully parallelizable.

The total work in this example is $L+\sum_{i=2}^{L} L-i+1 = \Theta(L^2)$ and $S_i = L - i + 1$ for all $i \geq 2$. In this case $M = L$ and the algorithm has makespan $\sum_{i=2}^{L} L - i + 1 = \Theta(L^2)$. On the other hand, the optimal schedule completes everything at time L. □

V. A Meta Scheduling Algorithm

In this section we show how the LevelBased scheduling algorithm can be combined with any other algorithm. In particular, given a scheduling algorithm A we design a new meta-algorithm A' that: (1) does not overload memory (including the memory used in precomputation), even if A overloads memory, and (2) gives worst-case bounds on makespan for all instances, even if A does not. Thus, the point is to remove all of the bad instances of A in terms of both the makespan and the space consumption while retaining A's good behavior.

Theorem 10. *Let A be any scheduler whose makespan, on a given instance, is T_a, and whose total space consumption is S_a. Let T_b be the makespan of the LevelBased scheduler on the same instance. Suppose we are given a budget $O(\zeta)$, where $\zeta = \Omega(V)$ for the total memory usage. Then, there exists a scheduler A', where*

- *A' uses at most $O(\zeta)$ memory.*
- *The makespan of A' on the instance is at most $2\min\{T_a, T_b\}$ if A uses less than ζ space and at most $2T_b$ otherwise.*

Proof. Algorithm A' uses $P/2$ processors to simulate A and the other $P/2$ processors to simulate LevelBased. Both algorithms are run independently of each other, which means that a task may be executed twice. If the memory consumption of A reaches $\zeta/2$ at any point, A' stops running A, and continues with LevelBased, using all of the processors. Algorithm A' finishes once either A or LevelBased finishes.

Algorithm A' uses at most ζ total memory because its simulation of A uses at most $\zeta/2$ memory, and its simulation of LevelBased uses at most $O(V)$ memory. Since, $\zeta = \Omega(V)$, the total memory usage is $O(\zeta)$. Since A' uses half the number of processes, the makespan of A and LevelBased increases by a factor of 2. Since A' finishes when either subscheduler finishes, its makespan is $2\min\{T_a, T_b\}$. □

Theorem 10 assumes that there is memory budget on the total memory consumption of the hybrid algorithm A'. Corollary 11 says that even without a memory budget, we can design a hybrid algorithm A' that removes all instances of A that have a bad makespan, without increasing the memory overhead by more than $O(V)$.

Corollary 11. *Let A be any scheduler whose makespan, on a given instance, is T_a. Let T_b be the makespan of the LevelBased scheduler on the same instance. Then, there exists a scheduler A', where*

- *The makespan of A' on the instance is at most $2\min\{T_a, T_b\}$.*
- *A' uses at most $O(V)$ additional memory over A.*

Worst-case guarantees for heuristics. Often in practice, heuristics are used because they typically result in a schedule with small makespan. However, these heuristics often can have large makespan in the worst case. Taking advantage of the LevelBased algorithm and running it alongside the heuristic ensures strong worst-case guarantees on the makespan without loosing performance on the typical case.

In Theorem 10, the given heuristic A and LevelBased algorithm were run independently of each other on half the processors because it is needed to prove instance-optimal guarantees of A'.

In practice, both algorithms (heuristic A and LevelBased) should be run in parallel, as they cooperatively identify ready-to-run tasks. In particular, the algorithms share a ready-to-run queue and add tasks to it as soon as they find more ready work. Thus, on a good instance for A, the performance of the hybrid algorithm will still be good. However, on a bad instance for A, where A may be stuck trying to find ready-to-run tasks, running LevelBased to identify ready-to-run tasks will ensure worst-case guarantees on its makespan.

VI. Experimental Evaluation

In this section, we establish the empirical efficiency of our schedulers. We show that the hybrid scheduling scheme (which combines the production LogicBlox scheduler with the LevelBased algorithm) results in similar or improved total execution times, while consistently reducing the scheduling overhead, by as much as 50% on some datasets.

In fact, in the process of implementing and evaluating the hybrid scheduler, we even managed to design a synthetic instance, on which the hybrid scheduler was performing 100x faster than the LogicBlox scheduler. This led the authors at LogicBlox to investigate inefficiencies

in their scheduler. They found that on this dataset, their scheduler was performing unnecessary work to find ready-to-run tasks. They were able to modify it and have it match the hybrid performance on the dataset (thus, improving a fundamental issue with their scheduler).

The bottom line is that LevelBased can be used not only to improve the scheduling performance, but also to identify scheduling anomalies and inefficiencies in any other scheduler for Datalog programs.

A. Experimental setup.

Datasets. We perform our empirical evaluation on job traces supplied by LogicBlox presented in Table I. The traces contain information about the structure of the scheduling DAG, supplemented by information about each task, such as the task processing time. All job traces except #11 are real proprietary datasets seen at production at LogicBlox. (Jobtrace #11 is a synthetic example generated by us to be released publicly.)

Simulator. To evaluate the performance of the LevelBased scheduler, we implemented a `C++` scheduling simulator using the *Boost Graph Library*] [3]. The simulator reconstructs the DAG from a job trace, attaching meta-information, such as its processing time, to each task. The simulator then runs the scheduler simulation on the reconstructed DAG and outputs the makespan.

Implementation. We describe how LevelBased scheduler is implemented. During the precomputation phase, LevelBased computes levels for each node of the computation DAG, as follows. Initially $\ell = 0$:

- All nodes with no incoming edges (i.e., indegree zero) get assigned level ℓ.
- Delete in-degree-zero nodes. Increment ℓ and recurse.

The scheduler maintains the lowest level among nodes in the active set. It schedules all jobs on that level (in a greedy manner) before proceeding to the next level.

B. Scheduling Algorithms

Baseline algorithm: LogicBlox. We compare our scheduling algorithms against the LogicBlox scheduler used in production by LogicBlox.

Before proceeding to the scheduling, the LogicBlox scheduler starts with the preprocessing phase. During that phase, the list of ancestors for each node in the original graph is computed. These ancestors are stored in an interval-list data structure. During the scheduling phase, the interval list is used to resolve which nodes are safe to run. This is done by iterating over the queue of the active nodes and checking ancestors from the interval list. If none of the nodes in the active queue are safe to run, then the LogicBlox scheduler waits for one of the currently running jobs to finish.

LevelBased with LookAhead. We also investigate an optimized version of LevelBased, which we call *LevelBased with LookAhead* (LBL). In the LevelBased with LookAhead algorithm, we run the LevelBased algorithm and whenever a processor is idle we recursively check for additional jobs to run by iterating through the activated nodes. In particular, when a processor is free, we run a breadth-first search to check if a candidate node from the next k levels is not a descendant of either running nodes or nodes that are yet to be run.

The algorithm takes a parameter k specifying the number of levels to look ahead. The worst-case running time of the LBL algorithm is $O(n^2)$, but it performs much better when there are a small number of nodes per level. This is fortuitous because when there are a large number of nodes per level, then the LevelBased scheduler performs essentially optimally.

Hybrid Scheduler. This algorithm combines the production LogicBlox scheduler with the LevelBased algorithm. The hybrid algorithm runs both schedulers in parallel, with a shared ready-to-run queue. Both schedulers independently identify ready-to-run tasks and add them to the shared queue.

This ensures that the hybrid scheduler inherits the good performance of the LogicBlox scheduler on typical instances, along with the strong worst-case guarantees of the LevelBased scheduler. In particular, it avoids the worst-case behavior of the LogicBlox scheduler, e.g. job trace #6, where the scheduler spends unnecessary time looking for more ready-work, when a simple LevelBased approach is sufficient for achieving the same makespan.

C. Empirical Evaluation

Empirical evaluation of LevelBased and LevelBased with LookAhead. The total makespan (which includes the scheduling overhead) of the production LogicBlox scheduler, LevelBased and LevelBased with LookAhead is presented in Table II. The execution time of LogicBlox scheduler was reported by LogicBlox engineers. All of the traces were simulated to run with eight processors. All three scheduler incur negligible scheduling overhead on these workloads (job traces #1-#5).

The performance gains from using the look-ahead heuristic depend on the depth of the look ahead. The results in Table II confirm that, if we increase the depth k of the look-ahead and check 15 or more levels, then the performance of the algorithms become nearly identical. Fifteen is a small number of levels as typically workloads have at least hundreds of levels (see Table I)

Empirical evaluation of the Hybrid scheduler. In Table III, we compare the total makespan and scheduling overhead of the LogicBlox, LevelBased and Hybrid

Table I: Details of workload traces from LogicBlox.

Job traces	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11
No. nodes	64910	64903	29185	64507	1719	379500	35283	35283	65541	65541	465127
No. edges	101327	101319	41506	100779	2430	557702	50511	50511	102219	102219	465158
No. initial tasks	5	16	76	26	6	125544	76	9	10	16	131104
No. active jobs	532	1936	560	1342	296	126979	645	177	111	1936	132162
No. levels	171	171	149	171	39	11	198	198	171	171	5

Table II: The total makespan of the LBLscheduler when the parameter k is varied as compared to the LogicBlox scheduler. All schedulers incur negligible scheduling overhead on these job traces.

Job trace	LogicBlox scheduler	LevelBased	LBL($k = 5$)	LBL($k = 10$)	LBL($k = 15$)	LBL($k = 20$)
Job trace #1	26.5 s	57.74 s	36.72 s	33.09 s	31.25 s	30.99 s
Job trace #2	9736 s	20,979.3 s	11,906.9 s	9,846.16 s	9,866.64 s	9,860.42 s
Job trace #3	187 s	448.40 s	299.34 s	285.91 s	230.22 s	229.34 s
Job trace #4	303 s	866.66 s	576.49 s	490.15 s	444.67 s	426.22 s
Job trace #5	23 s	29.32 s	24.52 s	24.52 s	24.52 s	24.52 s

Table III: This table shows the makespan and scheduling overhead (in this order) for the LogicBlox, LevelBased and Hybrid schedulers. The LogicBlox and Hybrid numbers are from production, as reported by the authors at LogicBlox, while the LevelBased number are from our simulation.

Job trace	LogicBlox Scheduler	LevelBased	Hybrid Scheduler
Job trace #6	(33.24 s, 21.69 s)	(0.49 s, 0.027 s)	(21.93 s, 10.89 s)
Job trace #7	(155.77 s, 0.109 s)	(348.35 s, 0.038 ms)	(187.08 s, 0.077 s)
Job trace #8	(28.69 s, 0.022 s)	(28.29 s, 0.009 ms)	(25.52 s, 0.020 s)
Job trace #9	(0.048 s, 0.0107 s)	(0.037 s, 0.013 ms)	(0.041 s, 0.009 s)
Job trace #10	(9,893.29 s, 0.327 s)	(20,897.9 s, 0.159 ms)	(10,123.74 s, 0.289 s)
Job trace #11	(688.38 s, 21.03 s)	(694.24 s, 0.042 s)	(630.01 s, 7.47 s)

algorithms. The experiments for the LogicBlox and the Hybrid schedulers were done in production and the numbers were reported by the authors at LogicBlox, while the LevelBased numbers are from simulation. The makespan of the schedulers is the total time taken to run all tasks and includes the scheduling overhead, but not any pre-processing cost.

Our results show that the hybrid algorithm produces similar or improved makespan compared to the LogicBlox scheduler. In particular, (for all except job trace #7), the makepan of the hybrid scheduler is within 2% of the LogicBlox scheduler, and is often much better. The real gains of the hybrid scheduler are in the scheduling overhead. In particular, the hybrid scheduler improves the scheduling overhead for all dataset in Table III.

The notable improvements in scheduling overhead are in job traces #6 and #11, where the execution DAGs are shallow, i.e., have a small number of levels. In particular, on job trace #6, the hybrid scheduler reduces the scheduling overhead of the LogicBlox scheduler by 50%. The speedup of the Hybrid scheduler arises due to the nature of the DAG and how each algorithm looks for ready work. The LogicBlox scheduler wastes time by performing many dependency checks (ancestor queries) to find the ready-to-run tasks. This results in worst-case performance to find active work. On the other hand, the Hybrid algorithm interleaves these checks with the LevelBased algorithm, which identifies the ready-to-run tasks in minimal time and is able to keep the processors saturated. Thus, the hybrid scheduler achieves strong worst-case guarantees and robustness without losing out on the best-case behavior of the LogicBlox scheduler.

VII. Conclusions

We study the incremental maintenance of Datalog programs and model it as a scheduling problem on DAGs. Datalog DAGs presents new challenges, not addressed in prior DAG scheduling literature, because the readiness of tasks cannot be determined locally, and instead depends on the dynamic execution of tasks at runtime. This aspect of Datalog programs makes the problem algorithmically interesting and practically challenging.

We give a provably time- and memory-efficient LevelBased scheduler for Datalog programs, and show that it provides asymptotically better guarantees than the benchmark algorithm used in production by LogicBlox. Our experiments show that our scheduler significantly outperforms the industry-grade LogicBlox scheduler on a dataset faced by LogicBlox at production.

Through our hybrid scheduler, we advocate the use of our scheduler alongside or heuristic, to provide strong worst-case guarantees and robustness without losing out on the best-case behavior of the heuristic.

ACKNOWLEDGMENT

We would like to thank Todd J. Green for his contributions and helpful discussions during the early stages of this project. This research was supported in part by NSF grants CRII-1947789, CCF-1725543, CCF-1452904, CSR-1763680, CCF-1716252, CCF-1617618, CNS-1938709, CCF-1830711, CCF-1733873, CCF-1733873, CCF-1845146, and by Sandia National Laboratories. B. Moseley was also supported in part by a Google Research Award, a Bosch junior faculty chair and an Infor faculty award.

REFERENCES

[1] Semmle Inc. https://semmle.com/. Accessed: December 20, 2018.

[2] Soufflé: Logic defined static analysis. http://souffle-lang.github.io. Accessed: December 20, 2018.

[3] *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[4] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 253–262, 1989.

[5] M. Alvarez-Picallo, A. Eyers-Taylor, M. P. Jones, and C.-H. L. Ong. Fixing incremental computation: Derivatives of fixpoints, and the recursive semantics of datalog. In *Programming Languages and Systems (ESOP)*, pages 525–552, 2019.

[6] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

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

[8] L. Baekgaard and L. Mark. Incremental computation of nested relational query expressions. *ACM Transactions on Database Systems (TODS)*, 20(2):111–148, June 1995.

[9] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-accelerated incremental storage and computation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[10] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, 2011.

[11] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, Sept. 1999.

[12] M. Carlsson. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 26–35, 2002.

[13] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, Sept. 1992.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[15] C. Demetrescu, D. Eppstein, Z. Galil, and G. F. Italiano. *Algorithms and Theory of Computation Handbook*, chapter Dynamic Graph Algorithms. Chapman & Hall/CRC, 2010.

[16] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 407–426, 2011.

[17] W. Fan, C. Hu, and C. Tian. Incremental graph computations: doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 155–169, 2017.

[18] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 1–34, 1997.

[19] E. Frachtenberg, D. G. Feitelson, J. Fernandez, and F. Petrini. Parallel job scheduling under dynamic workloads. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 208–227, 2003.

[20] T. J. Green, M. Aref, and G. Karvounarakis. LogicBlox, platform and language: a tutorial. In *Datalog in Academia and Industry*, pages 1–8. 2012.

[21] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[22] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, number 2, pages 328–339, 1995.

[23] M. Gyssens and L. V. S. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 106–115, 1997.

[24] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 156–166, 2014.

[25] M. T. Kaufman. An almost-optimal algorithm for the assembly line scheduling problem. *IEEE Transactions on Computers*, C-23(11):1169–1174, Nov. 1974.

[26] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 81–89, 1999.

[27] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with MapReduce: a survey. *ACM SIGMOD Record*, 40(4):11–20, Jan. 2012.

[28] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., 2004.

[29] Y. A. Liu. *Incremental Computation: A Semantics-based Systematic Transformational Approach*. PhD thesis, Cornell University, Jan. 1996.

[30] A. S. Manne. On the job-shop scheduling problem. *Operations Research*, 8(2):219–223, 1960.

[31] E. Nuutila. Efficient transitive closure computation in large digraphs. *Acta Polytechnica Scandinavica: Mathematics and Computing in Engineering*, 74:1–124, July 1995.

[32] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 315–328, 1989.

[33] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, Sept. 1991.

[34] İ. Sabuncuoğlu, Y. Gocgun, and E. Erel. Backtracking and exchange of information: methods to enhance a beam search algorithm for assembly line scheduling. *European Journal of Operational Research*, 186(3):915–930, May 2008.

[35] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled prolog: beyond pure logic programs. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 215–229, 2006.