Optimizing Parallel Recursive Datalog Evaluation on Multicore Machines

Jiacheng Wu
wu-jc18@mails.tsinghua.edu.cn
Tsinghua University, Beijing

Jin Wang*
jinwang@cs.ucla.edu
University of California, Los Angeles

Carlo Zaniolo
zaniolo@cs.ucla.edu
University of California, Los Angeles

ABSTRACT

Over the past years, there has been a resurgence of interest in Datalog due to its superior ability of expressing applications that require recursive computations. However, in addition to expressive power, supporting analytical tasks with ever-increasing volume of data requires high performance and scalability. In this paper, we present DCDatalog, an in-memory Datalog engine specifically designed for modern shared-memory multicore architectures. Our key contribution is a novel system architecture that supports a wide scope of Datalog applications with a light-weight coordination scheme during parallel evaluation. To this end, we propose a dynamic scheduling strategy that can generate the parallel execution plan on-the-fly while reducing concurrent accesses to the shared memory. Experimental results on several large datasets show that our system significantly outperforms existing parallel Datalog engines and also scales well with increasing amount of data.

CCS CONCEPTS

• **Information systems → Relational parallel and distributed DBMSs**.

KEYWORDS

Datalog, Multicore Machine, Query Processing, Efficiency

ACM Reference Format:
Jiacheng Wu, Jin Wang, and Carlo Zaniolo. 2022. Optimizing Parallel Recursive Datalog Evaluation on Multicore Machines. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517853

1 INTRODUCTION

In the past years, there is a resurgence of Datalog due to its ability to specify declarative data-intensive applications that execute efficiently over different systems and architectures. The recent theoretical advances [32, 55] enable the usage of aggregates in recursions, and this leads to considerable improvements in the expressive power of Datalog. As a result, Datalog has been widely adopted to express complicated recursive queries in many domains [13], such as artificial intelligence [12], graph analysis [2], knowledge reasoning [9], declarative network [30] and many others.

With the ever-growing scale of data analysis tasks, a high level of performance and scalability becomes critical for Datalog systems. In response to this need, many parallel Datalog engines have been developed by researchers from both academia and industry with the expectation that SQL and other query languages will also benefit from these advances. Based on which environment they are deployed, these Datalog engines can be divided into two categories: shared-memory [16, 40, 52] and shared-nothing [41, 43, 46] ones. These approaches implement the idea of parallel bottom-up evaluation [21] by splitting the tables into disjoint partitions via discriminating functions, such as hashing, where each partition is then mapped to one of the parallel workers. After each iteration, workers coordinate with each other to exchange newly generated tuples when necessary. The final result is the union of contributions by all workers. In this way, the entire computation can be divided among all workers and operated in parallel.

Motivated by the emergence of modern commodity machines with massively parallel processors [6], previous studies have demonstrated shared-memory multicore architectures yield superior performance for Datalog applications. However, these studies either (i) underutilize the multicore architecture due to poor parallelism [52]; or (ii) are based on different system architectures [16, 38, 40]. Therefore, while these studies provide highly valuable techniques, mechanisms and execution models, none of them fully exploits the knowledge at hand to maximize parallel execution to the full extent described in this paper.

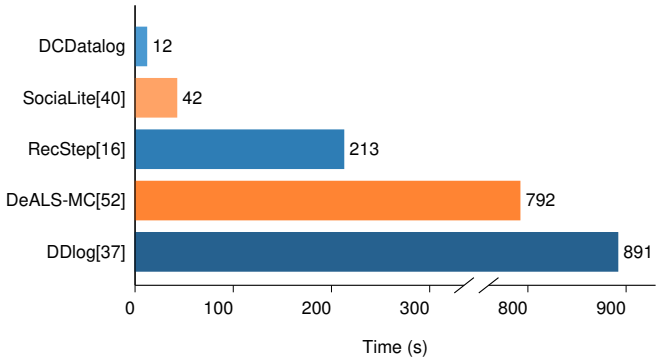


Figure 1: Query Performance of SSSP on LiveJournal Dataset

In this paper, we aim to propose **D**ynamic **C**oordinate **Datalog** (DCDatalog), a parallel engine on shared-memory multicore machines to scale up Datalog evaluation. The key challenge in devising a good plan for the parallel evaluation of recursive Datalog programs consists in providing an efficient mechanism to resolve race

*Corresponding author

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517853

conditions, as required to ensure the atomicity of concurrent update operations. The state-of-the-art solution presented in [52] solves this problem by (i) identifying a family of lock-free programs and (ii) forcing global coordination after each iteration in the parallel evaluation plan. The first issue of this approach is the limited range of programs that are lock-free or can be turned into lock-free ones by simple rules rewriting. As a result, this approach fails to support more powerful applications, such as those that require aggregates in recursion. Secondly, the above approach incurs costly coordination overhead and poor scalability since the faster workers are forced to wait idly until the slowest worker reaches the coordination point. Witnessing this problem, we propose a new system architecture that removes the limitation of both lock-free and decomposable programs, thus boosting the parallel evaluation performance of Datalog applications. We deal with race conditions by leveraging a light-weight scheme that eliminates the need to perform global coordination among workers. This is made possible by the **D**ynamic **W**eight-based **S**trategy (DWS) proposed in this paper. Instead of blocking the faster workers, DWS reduces the straggler using effective local checkers controlled by a simple weight-based mechanism. Since such weights are calculated on-the-fly, the workers can dynamically decide on whether to perform idle waiting or proceed to the next iteration. This relaxation significantly reduces the overall execution time and improves the parallelism of evaluation plans. For example, Figure 1 clearly illustrates that, thanks to its better coordination strategy, DCDatalog considerably outperforms its competitors on the Single Source Shortest Path (SSSP) query.

To optimize the implementation of the above strategy, we devise a suite of query planning techniques that map a Datalog program to a family of specialized relational operators describing the behavior of parallel executions. In this way, we can combine the query optimization techniques of relational DBMS with our dynamic coordination strategy in a shared-memory multicore environment. Since computing aggregate-in-recursion represents the bottleneck for many complex recursive queries, we also propose several optimizations to accelerate this process.

The contributions of the paper can be summarized as follows:

1. We study the problem of improving the parallel Datalog evaluation in shared-memory environments by proposing a new coordination strategy as well as a prototype system. Our DWS strategy (Section 4.2) significantly improves parallelisms by eliminating the requirement of global coordination among all workers.
2. We develop a Datalog engine DCDatalog in shared-memory environments by implementing the DWS approach. To further improve its performance, we also develop some optimizations in query planning (Section 5) and system implementation (Section 6).
3. We conduct extensive experiments with five widely used recursive Datalog programs on both synthetic and real world datasets. The experimental results demonstrate that our DCDatalog engine provides across-board performance gain and outperforms existing Datalog engines by one to two orders of magnitude.

The rest of this paper is organized as follows: We provide the necessary backgrounds in Section 2 and introduce the overall system architecture in Section 3. We propose the dynamic coordination strategy in Section 4. We present the query planning techniques to implement above strategy in Section 5. We introduce the optimizations for system implementation in Section 6. We show the experimental results in Section 7. We survey the related work in Section 8. Finally the conclusion is made in Section 9.

2 PRELIMINARY

2.1 Datalog

A Datalog program P consists of a finite set of rules operating on sets of facts described by database-like schemas. A rule r has the form $h \leftarrow r_1, r_2, ..., r_n$, where h is the head of rule, $r_1, r_2, ..., r_n$ is the body and each comma separating atoms in the body represents the logical conjunction (AND). The rule head h and each r_i are atoms having form $p(t_1, t_2, ..., t_k)$, where p is the predicate and $t_1, t_2, ..., t_k$ are terms which can be variables or constants. On occasions, We use the terms predicate, table and relation interchangeably. A rule defines a logical implication: if all predicates in the body are true, then so is the head h. There are two kinds of relations: (i) the base tables are defined by tables in the *EDB* (extensional database) and (ii) the derived tables are defined by the heads of rules and form the *IDB* (intentional database). To further illustrate the key concepts and terminology of Datalog consider the following program:

Query 1 - Transitive Closure (TC)

$$\begin{aligned} \mathrm{r}_{1,1}: &\quad tc(X,Y) \leftarrow arc(X,Y) \\ \mathrm{r}_{1,2}: &\quad tc(X,Y) \leftarrow tc(X,Z), arc(Z,Y) \end{aligned}$$

Query 1 derives the IDB relation *tc* from the EDB table *arc* representing the edges of a graph. Since the predicate *tc* is contained in both the head and body of rule $\mathrm{r}_{1,2}$, *tc* is a *recursive table* and $\mathrm{r}_{1,2}$ is a *recursive rule*. *tc* is also the head predicate for $\mathrm{r}_{1,1}$ which is non-recursive and therefore provides the *base rule* in the evaluation of our query. Since at most one recursive predicate is included in the body of any of its rule, Query 1 represents a case of *linear recursion*; the term *non-linear recursion* denotes instead the case where some rules contain multiple recursive goals.

The process of query evaluation first initializes *tc* using $\mathrm{r}_{1,1}$, and then uses $\mathrm{r}_{1,2}$ to recursively produce new *tc* facts with join operation between the table containing the *tc* atoms generated in previous iterations with the *arc* relation. The mapping of the head tables from the body tables can be described as an operator, known as the Immediate Consequence Operator (ICO), that can be expressed using relational algebra (RA). While non-recursive queries can be implemented directly using the RA expression of their ICO, the iterative procedure of Definition 2.1 is used to derive the least solution of the fixpoint equation $I = T(I)$ which defines the formal semantics for the recursive queries expressed by *positive Datalog programs*, i.e., programs without negation or aggregates.

Definition 2.1. For a program P with Immediate Consequence Operator (ICO) T, the operational semantics of P is defined by $T^{\uparrow\omega}(\emptyset)$ where: ω denotes the first infinite ordinal, $T^{\uparrow 0}(\emptyset) = \emptyset$ and $T^{\uparrow n+1}(\emptyset) = T(T^{\uparrow n}(\emptyset))$. Then $T^{\uparrow\omega}(\emptyset)$ denotes the union of $T^{\uparrow n}(\emptyset)$ for every n. A recursive program whose iteration converges to the final value in a finite number of steps reaches its fixpoint at the first integer $n+1$, where $T^{\uparrow n+1}(\emptyset) = T^{\uparrow n}(\emptyset)$.

The fixpoint computation described above is then optimized into the (SN) evaluation [3]. SN performs a differential fixpoint

computation of a Datalog program by first applying its base rules and then applying a delta version of its recursive rules till the fixpoint is reached. The core idea of SN is that the delta rules will use *delta relations*, which contain the new atoms produced in the previous iteration step, to avoid the duplicate generation of previously generated atoms.

2.2 Parallel Evaluation of Datalog Programs

There are two kinds of parallel execution architectures: shared-memory and shared-nothing. In this paper, we focus on the shared-memory architecture, where both the base and recursive relations are stored in the main memory that can be directly accessed by all processors, as it is in fact the case for most modern multicore machines. We will use the terms processor, thread and worker interchangeably if there is no ambiguity in the context. When different processors visit the same memory cells, race conditions occur when one or more of the processors performs a write operation. In this case, a lock mechanism is required to ensure the atomicity of operations. The examples of Datalog engines under this architecture include DeALS-MC [52], Souffle [27] and RecStep [16]. On the other hand, in a shared-nothing architecture, the data is distributed into different computation nodes. The nodes in a cluster use the message passing mechanism to exchange information with each other via additional network communication. The examples include BigDatalog [43], Distributed SociaLite [41], Myria [46] and MRA [49].

The state-of-the-art method for evaluating Datalog programs in parallel is the *substitution partitioned parallelization* scheme [21]. This scheme first divides the workload into several disjoint partitions using hash-based discrimination functions, and then assigns each partition to exactly one worker. As the bottom-up SN evaluation is performed by each worker on its assigned data, a worker often needs to use records generated by other workers. Therefore, a *coordination* across workers is required to manage their exchanges. In this paper, we follow the approach used in previous studies and assign partitions according to the join key of each table. Specifically, we use the hash mechanism used in [52] to split the key range into different partitions (as per the function H in Algorithm 1 shown later). Thus the records of both base and recursive tables are allocated into partitions according to the value of their join key. Note that Issues of how to improve the performance of partition schemes are beyond the scope of this paper.

In this paper, we will say that a *local iteration* is executed by a worker when it finishes one iteration of SN evaluation, whereas a *global iteration* is executed when all workers have finished the same number of local iterations. When the delta table becomes empty after a local iteration, then we will say that a *local fixpoint* is reached, whereas the parallel evaluation terminates when the *global fixpoint* is reached.

3 OVERALL FRAMEWORK

In this paper, we aim at developing a Datalog engine which, by eliminating the requirement that programs must be lock-free, will support a wide scope of applications or programs with superior performance and scalability.

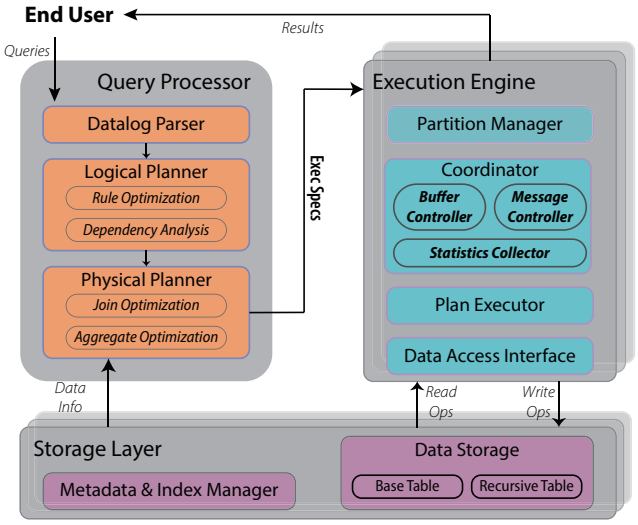


Figure 2: DCDatalog: The Overall Architecture

To archive better performance and scalability for our novel engine, we propose a dynamic strategy DWS to control the coordination process during parallel execution. Instead of requiring workers to request information from all others after a global iteration, each worker just sends the newly generated delta table to the memory space owned by other workers after its local iteration. In this way, we eliminate the requirement of global coordination, and thus significantly save the time of idle waiting. More details will be shown in Section 4.2.

To efficiently implement DWS, we developed the prototype Datalog system DCDatalog, with the overall architecture shown in Figure 2. Apart from the classical and simple recursions, DCDatalog could support programs containing queries with non-linear or mutual recursions. One limitation is that it cannot support programs with the negation in recursions, which is still an open problem yet. Moreover, DCDatalog consists of the following components:

Query Processor The Query Processor provides the functionality of analyzing and planning for the input Datalog programs. It consists of three steps: (i) The *Datalog Parser* compiles the input Datalog program and generates its Predicated Connected Graph (PCG) [8], which is implemented with the data structure of *AND/OR Tree*; (ii) The *Logical Planner* maps the PCG into relational operators to form the logical plan; (iii) The *Physical Planner* further generates the physical plan to be executed in parallel. The details will be illustrated in Section 5.

Execution Engine The Execution Engine aims at providing efficient and scalable execution for the operators in the physical plan. It consists of three components: *Partition Manager*, *Coordinator* and *Buffer and Message Controller*. As detailed in Section 6, we use light-weight *atomic operations* instead of locks to deal with race conditions among workers, and various optimizations affecting the computation of aggregates in recursion are also applied at this point.

Storage Layer The Storage Layer provides the index and storage functions for the base and recursive relations during the SN evaluation process. In this paper, we utilize the storage engine of the

DeALS system [44] enhanced with the B^+-Tree index implemented by ourselves following the idea of the previous studies [52]. Alternatively, other relational DBMSes could be also used as the storage and index engines. Further improvements in this module are also possible but beyond the scope of this paper.

4 DYNAMIC COORDINATION STRATEGY

In this section, we present the dynamic coordination strategy used in the shared-memory multicore architecture. In Section 4.1 below, we start with introducing existing mechanisms for parallel execution coordination. Then in Section 4.2, we propose our new DWS strategy designed to optimize performance. Finally, in Section 4.3, we discuss the support of more complex recursive queries.

4.1 Parallel Execution Mechanism

We first introduce how the SN evaluation has been parallelized in previous studies such as [52] in Algorithm 1. In these studies, the key range is split into disjoint partitions by a predefined hash function H (line 2). If there are m partitions $P_1, ..., P_m$ and n workers $W_1, ..., W_n$ ($m = n$), all workers will run in parallel, each dealing with its own partition. It is trivial to extend the proposed techniques to the case where $m \geq n$ To accelerate evaluation, a separate hash index is built for each partition of the base relation (line 3). Then all workers become active and execute the parallel computation of SN as follows: each worker W_i first initializes its recursive table R_i according to the base rule (line 8) and build a B^+-Tree index on R_i (line 9). Next, a local iteration of semi-naive evaluation is processed and the new delta relation $\delta R_i'$ is generated (line 11). After all workers finish a global iteration, they coordinate with each other by exchanging the newly generated tuples according to H, before proceeding to the next iteration (line 13). At this point, we also need to update indices on recursive tables and perform deduplication. Thus, If $\delta R_i'$ is empty, then the local fixpoint has been reached, and W_i is set to inactive (line 15). But if the δR_i of some inactive worker W_i becomes non-empty after coordinating with other workers, then the worker will become active again (line 17). The parallel evaluation terminates when all workers become inactive (line 19), denoting that the global fixpoint is reached. The final result is equal to the union of recursive tables on all workers (line 20). We refer to this simple approach described above as Global.

Query 2 Connected Component (CC)

$$\begin{aligned} r_{2,1}: \quad & cc2(Y, min\langle Y\rangle) \leftarrow arc(Y, \_). \\ r_{2,2}: \quad & cc2(Y, min\langle Z\rangle) \leftarrow cc2(X, Z), arc(X, Y). \\ r_{2,3}: \quad & cc(Y, min\langle Z\rangle) \leftarrow cc2(Y, Z). \end{aligned}$$

Example 4.1. We evaluate the *CC* program on the graph shown in Figure 3(a). The execution process of Global (Algorithm 1) is displayed in Figure 3(b)(1). The workers W_2, W_3 are slower than W_1 since they are associated with more edges. In the first global iteration, workers W_1, W_2 and W_3 take 5,8 and 8 time units, respectively. Under Global, when W_1 finishes running, it is blocked as it waits for other workers to finish the global iteration. Once W_2 and W_3 have finished their work, they coordinate and exchange their newly generated tuples with each other and W_1. At that point, W_2 and W_3 complete their connected component with vertex 4. Thus, they realize it with vertex 1 after four global iterations. In conclusion, Global takes a total of 128 time units.

We can see that in (Algorithm 1) idle waiting might happen before coordination (line 13). This is due to the requirement that all workers should wait until the current global iteration is finished. For complex queries that are not lock-free, coordination among all workers can result in serious race conditions, which will involve significant overhead. To address this problem, before presenting DWS, we will first discuss a simple improvement that extends the shared-nothing evaluation method recently proposed in [14]. This method uses the Stale-Synchronous Parallel (SSP) model which was previously proposed for distributed machine learning [11, 26]. The core idea of this approach is that the constraints now imposed on local iterations can be relaxed as follows: *Instead of waiting after conducting just one local iteration, we allow all workers to continue executing at most s local iterations before stopping to wait for the current global iteration to be finished.* Thus the intuition behind it consists in having workers spend more time performing actual computation rather than waiting for straddlers to finish. The benefits can be maximized by carefully tuning the hyper-parameter s.

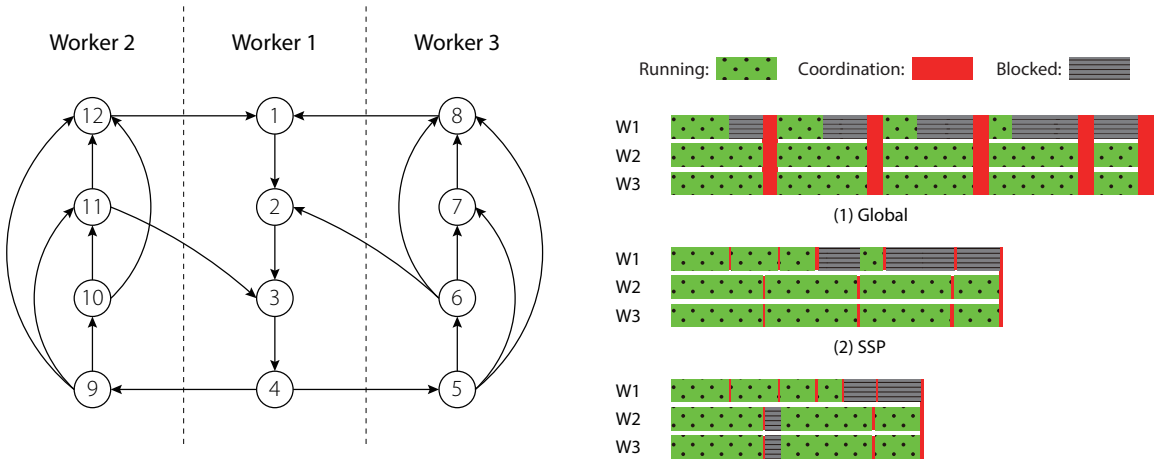
Example 4.2. Let's look back to the previous example, assuming that $s = 1$ in SSP. As depicted in Figure 3(b)(2), W_1 is not blocked by W_2 and W_3 in the first three local iterations as it can proceed one iteration ahead of them under SSP. After that, due to the constraint $s = 1$, W_1 needs to wait until W_2 and W_3 finish their 2^{nd} local iteration. In this example, the coordination only takes one time unit as it only requires the information dispatched by one worker. Thus SSP finishes in 88 time units, about 40% faster than Global.

Furthermore, to alleviate the overhead brought by race conditions, we can split the main memory space into units with finer granularity: Each worker is associated with a memory segment $\mathcal{M}_i$ to hold the delta relation newly generated by other workers whose key falls in the range of W_i under H. The memory space for storing newly generated tuples from worker W_j is denoted as $\mathcal{M}_i^j$. In this way, when performing coordination among workers, the race condition will happen just in a buffer $\mathcal{M}_i$ rather than the whole memory space. We further propose several optimizations in the aspect of system implementation, which will be detailed in Section 6.1.

4.2 The DWS Approach

Although SSP strategy is quite effective, limitations remain. In fact, it is very difficult to set a proper value for s. Actually, as only one s value will be used during the whole evaluation, it is unlikely that it will remain the best for all workers during the whole SN evaluation. Moreover, the approach still requires coordination after each global iteration, causing additional overhead due to race conditions.

To address above concerns, we propose the **D**ynamic **W**eight-based **S**trategy (DWS) to further improve coordination during parallel SN evaluation. In DWS, we eliminate the requirement of global coordination, and instead, once a worker completes its local iteration, we let it decide whether to proceed to the next iteration as described next. During the evaluation by a worker W_i, the evaluation time per iteration depends on the cardinality of its delta table, and when this cardinality is small, W_i is likely to complete



(a) A Toy Graph

(b) Global, SSP and DWS

Figure 3: Execution Time under Different Coordination Strategies

Algorithm 1: Parallel Evaluation (B, H)

Input: B: The base table, H: The hash function for partition
Output: $\mathcal{R}$: All results in the recursive table

```
1  begin
2      Split the key range into disjoint partitions with H;
3      Construct Index for the each partition of B on the partition
       key;
4      while True do
5          foreach worker W_i do
6              // Run in parallel
7              if In the first iteration then
8                  Initialize R_i and δR_i with the base rule;
9                  Construct an index on R_i;
10             if W_i is active then
11                 Conduct one local iteration of evaluation with
                   δR_i and generate δR'_i;
12             // Wait until global iteration is finished
13             Coordinate with all workers to update δR_i
               according to H, update the index for R_i;
14             if δR_i = ∅ then
15                 Mark W_i as inactive;
16             else
17                 Mark W_i as active;
18         if All workers are inactive then
19             Terminate the evaluation;
20     return R as the union of all workers;
21 end
```

faster than other workers. In this case, W_i should wait and collect more tuples from slower workers. Otherwise, W_i should allocate the newly generated tuples to the message buffers M_j^i of other workers W_j ($j \neq i$) and move on to the next iteration. To implement this strategy, we need the following two parameters for each worker i: the time τ_i that W_i should wait before proceeding to the next iteration and a threshold ω_i such that W_i will proceed to the next iteration if the cardinality of delta table is larger than it. As described later, the values of these parameters can be automatically calculated at each iteration. Thus the need for manual tuning is avoided and a better coordination could be achieved. Moreover, the risk of race conditions on memory buffers will be reduced since each worker updates its memory buffers with the delta atoms in an asynchronous manner.

Algorithm 2 describes the behavior of DWS in each iteration. In a way similar to Algorithm 1, it first initializes the recursive table R_i and the delta table δR_i from base rules. If W_i is active, it will collect newly generated tuples from the memory buffer M_i^j, remove tuples already contained in R_i and merge the remaining tuples into the delta table (line 4). Then the algorithm makes a decision according to the cardinality of the newly generated delta table. If the cardinality is smaller than ω_i, the algorithm must wait at most τ_i time units and collect more tuples from other workers before continuing to computing δR_i (line 6). Using the deadlock-avoidance policy that is common in many systems, W_i resumes its processing after a predefined timeout (line 8). If the cardinality is zero, then the local fixpoint is reached and W_i becomes inactive (line 10). Otherwise, the evaluation proceeds to the next iteration by (i) updating the two parameters with a weight-based mechanism (line 12), (ii) performing one iteration of evaluation with the collected delta table δR_i (line 13), (iii) sending the tuples of newly generated delta table $\delta R_i'$ to buffers of other workers and preparing for the next iteration (line 14-17).

Example 4.3. As shown in Figure 3(b)(3), W_1 is never blocked; thus it can quickly propagate the connected component with vertex 1 to other workers. Besides reducing unnecessary computation in SSP, W_2 and W_3 only need to wait a short while to obtain the newly generated tuples produced by W_1 in its second local iteration. As a result of these improvements, DWS, with little additional waiting time included, requires 67 time units: this is about half the time of Global and represents a solid improvement over SSP.

Algorithm 2: Execution of DWS on worker W_i

```
1  begin
2      //Replace line 10 to 18 in Algorithm 1, all workers are
       active in the beginning
3      if W_i is active then
4          δR_i ← (∪_j M_i^j − R_i) ∪ δR_i;
5      while 0 < |δR_i| < ω_i do
6          W_i waits for τ_i time and collect more tuples;
7          if W_i is active due to timeout then
8              break;
9      if |δR_i| = 0 then
10         Make W_i as inactive;
11     else
12         Update ω_i and τ_i;
13         Conduct one local iteration of evaluation with δR_i,
           generate δR'_i;
14         if ∃ tuple R ∈ δR'_i associated with W_j then
15             Update M_j^i and make W_j as active;
16         R_i ← R_i ∪ δR_i;
17         Update the index for R_i, δR_i ← ∅;
18 end
```

A remaining issue is to dynamically adjust the values of parameters ω_i and τ_i for each worker W_i. To address this issue, we utilize the *Queueing Theory*[1] to model the behavior of each worker and adjust such parameters on-the-fly from some basic statistics. The Queueing Theory initially aims at studying the arrivals and departures of elements waiting in a queue. To represent such arrivals and departures, Queueing Theory focuses on the mean arrival rate λ, and the mean service rate μ. In the DWS process, tuples arrive to the worker from message buffers M_j^i and wait in a queue for workers to perform computation on them. For the problem at hand, λ can be defined as the average frequency with which tuples arrive at the worker, and μ can be defined as the reciprocal of the average computation time that a tuple spends on a worker [2]. Then the variants for the arrival time and service time distributions will be denoted as σ_a^2 and σ_s^2, respectively.

Following this route, all message buffers M_i^j in worker i are responsible for maintaining the arrival statistics λ_j and $\sigma_{a,j}$ periodically and incrementally from worker j. To update ω_i and τ_i, we first compute λ and σ_a^2 for worker W_i based on the queueing model G/G/1 [10] that provides the following Equation (1):

$$\lambda = \left(\frac{\sum_j |\mathsf{M}_i^j| \cdot (\lambda_j)^{-1}}{\sum_j |\mathsf{M}_i^j|}\right)^{-1};\ \sigma_a^2 = \frac{\sum_j |\mathsf{M}_i^j| \cdot \left(\sigma_{a,j}^2 + (\lambda_j)^{-2}\right)}{\sum_j |\mathsf{M}_i^j|} - (\lambda)^{-2} \tag{1}$$

Observe that each worker can easily maintain its μ and σ_s^2 statistics. Then, the mean number of tuples in the queue L_q can be estimated with the Kingman's formula [28] as shown in Equation (2), which is known for its good accuracy and is widely used in real-life applications. Thus, with $C_a^2 = \lambda^2 \cdot \sigma_a^2$, $C_s^2 = \mu^2 \cdot \sigma_s^2$ and ρ computed as λ/μ, we have:

$$L_q \simeq \frac{\rho^2(C_a^2 + C_s^2)}{2(1-\rho)} \tag{2}$$

Having derived L_q, we can next estimate ω_i as the expected mean queue length L_q, given that L_q represents the most common queue length when the queues are stable. Thus, in the best case scenario, when none of the generated tuples is in R_i, it is feasible for workers to wait and collect those tuples in the delta relation δR_i. Also, the τ_i can be estimated as $L_q/\lambda = \omega_i/\lambda$, which is also the mean waiting time in the queue since λ is the arrival rate.

Finally, we can make the following observation from a qualitative analysis for our algorithms under certain simplified conditions, i.e., when one worker is slow due to unbalanced workloads:

Worst Case Analysis: When a program reaches the fixpoint, the number of iterations run in the slowest worker of DWS is no larger than that of Global. Specifically, if the slowest worker receive a new tuple in iteration k in the Global approach, then the slowest workers would have seen this new tuple before (or no after) iteration k in DWS since other workers at least run k iterations. Therefore, in DWS, the slowest worker could use fewer iterations to reach the fixpoint. Considering that the overall running time of the program is dominated by the time of the slowest worker, our approach would work well in practice.

4.3 Support for Queries with Complex Recursion

In this section, we show that our proposed techniques can also supports Datalog programs with non-linear and mutual recursion.

As an example of non-linear recursion, let us consider *Query 3* below, with multiple recursive tables *path* in the body of the rules:

Query 3 All Pairs Shortest Path (APSP)

$$\begin{aligned}
&\mathrm{r}_{3,1}: path(A, B, min\langle D\rangle) \leftarrow warc(A, B, D).\\
&\mathrm{r}_{3,2}: path(A, B, min\langle D\rangle) \leftarrow path(A, \underline{C}, D_1), path(\underline{C}, B, D_2),\\
&\qquad\qquad\qquad\qquad\qquad\quad D = D_1 + D_2.\\
&\mathrm{r}_{3,3}: apsp(A, B, min\langle D\rangle) \leftarrow path(A, B, D).
\end{aligned}$$

To deal with non-linear recursion in $\mathrm{r}_{3,2}$, we partition the first recursive table *path* by attribute B and the second one by attribute A. At this point, for each tuple $(A, B, min\langle D\rangle)$, we route it to and replicate it in the two partitions $H(A)$ and $H(B)$. In this way, we can successfully collect two parts of the table with same join keys and then perform the join. This can be easily implemented by maintaining two R_i tables and modifying line 13-14 in Algorithm 2 to allocate each newly generated tuple to two partitions. In this process, at most twice the number of tuples in table *path* are allocated among workers in each iteration.

In the case of mutual recursion, we have that two or more rules belonging to different predicates refer to each other (e.g., $A \leftarrow B, B \leftarrow A$). We will use *Query 4* below to illustrate how our techniques handle this situation.

Query 4 Who will attend the party (Attend)

$$\begin{aligned}
&\mathrm{r}_{4,1}: attend(\underline{X}) \leftarrow organizer(\underline{X}).\\
&\mathrm{r}_{4,2}: cnt(\underline{Y}, count\langle X\rangle) \leftarrow attend(\underline{X}), friend(Y, \underline{X}).\\
&\mathrm{r}_{4,3}: attend(\underline{X}) \leftarrow cnt(\underline{X}, N), N \geq 3.
\end{aligned}$$

[1] https://en.wikipedia.org/wiki/Queueing_theory

[2] λ_i and μ_i can be different for different workers W_i.

The mutual recursion here involves tables *attend* and *cnt*. For $r_{4,2}$, once *attend* is obtained, it is joined with *friend* and the *cnt* tuples so obtained are assigned to the corresponding partitions specified by the group-by attribute Y. After deriving the aggregate *cnt*, we apply the condition $N \geq 3$ in $r_{4,3}$ and obtain the new *attend* table for next iterations. We just need to add this step in line 13 of Algorithm 2.

5 QUERY PLANNING

Before query execution, the Query Processor compiles the Datalog program into an AND/OR tree to identify the recursion, and to translate it into a *Logical Plan* (Section 5.1). The *Physical Plan* generated from this specifies how DWS is executed by a set of physical operators. These operators are different from and complement those of relational DBMS, since here they also include parallel execution primitives, such as coordinating with other threads and accessing the memory buffer (Section 5.2).

5.1 Logical Plan

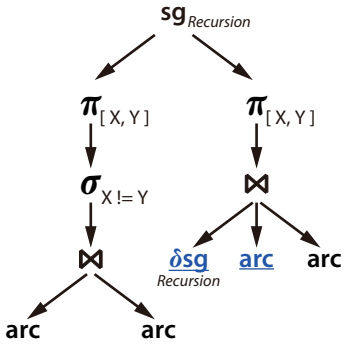


Figure 4: Illustration of the Logical Plan: SG

As shown in the previous study [52], the bottom-up parallel evaluation starts from the AND/OR tree of a Datalog program. To further analyze and evaluate the program, the query processor first maps the AND/OR tree into a logical plan. In our DCDatalog engine, a logical plan is a Directed Acyclic Graph, where nodes denote tables and relational operators, and edges denote the data flow. To identify the recursive predicates in the program, we mark them with special tags. The basic operators in our logical plan are similar to those of relational DBMS: Projection, Join and Selection. For join operators, their join keys are specified in the logical plan and will be used for further processing to be discussed later. Figure 4 shows the logical plan for *Query 5*. The left branch in Figure 4 displays the logical plan of exit rules $r_{5,1}$; the right branch displays the relational operators implementing the recursive rule $r_{5,2}$ at each iteration.

Query 5 Same Generation (SG)

$$r_{5,1}: \quad sg(X,Y) \leftarrow arc(P,X), arc(P,Y), X != Y.$$
$$r_{5,2}: \quad sg(X,Y) \leftarrow arc(A,X), sg(A,B), arc(B,Y).$$

In the process of logical planning, greedy optimization steps, such as pushing down the selection operator to leaf nodes, are applied automatically. Then, an optimization step that is specific to parallel SN evaluation is applied by reordering the tables in the body of recursive rules. For instance, assume that in the recursive rule below P is a recursive table, and b_i and c_i denote different base tables:

$$P(X_1, X_2, ...) \leftarrow b_1, b_2, ..., P(Y_1, Y_2, ...), c_1, c_2, ...$$

The standard left-to-right processing order will not be used for this rule: instead we will always use the recursive relation as the leftmost table in the join. Thus the above rule is reordered and processed as:

$$P(X_1, X_2, ...) \leftarrow P(Y_1, Y_2, ...), b_1, b_2, ..., c_1, c_2, ...$$

The reason for this optimization is that the physical plan performs a nested-loop-join without any further reordering, and by having the recursive table in the outer-loop we can take advantage of the indexes built on the base tables, as discussed in Section 5.2. For example, in the logical plan for SG shown in Figure 4, we can see that δsg is the first child of the join operator in the right subtree. Thus, this optimization will exchange the positions of δsg and *arc*.

5.2 Physical Operators and Plan

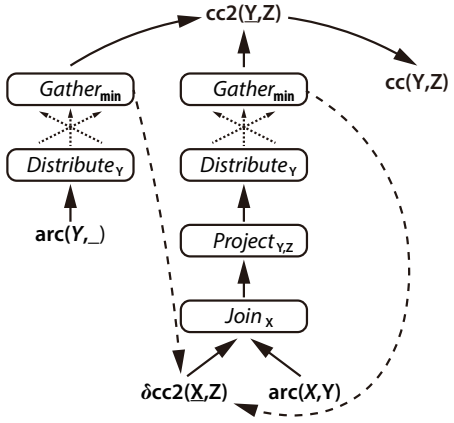


Figure 5: Illustration of Physical Plan: CC

Unlike in relational DBMS where the physical plan is directly generated from a logical plan, the query processor of DCDatalog involves the specification of additional for parallel execution activities before the generation of physical plans. As a result, the physical operators might be required to manage the coordination among different workers. Next we describe the physical operators as follows:

5.2.1 Join. In a Datalog program, the conjunctive queries are implemented by the natural join operation. For recursive queries, we only use binary join in the physical plan, and support multi-way join by series of binary join operations. Indeed binary join operations involved in recursion can be supported very efficiently with the help of tags specified in the logical plans.

As discussed in Section 4, we build a B^+ Tree index on the partition key of all relations. Thus we take advantage of such indices to perform the joins, given that base relations remain unchanged during the evaluation. Similarly to relational DBMS, the actual join operation in DCDatalog will be either implemented as a hash join,

or as an index join, or as a nested-loop join. To select the best join method, we use the following simple optimization heuristic that has proven quite effective: when the recursive rules contain two or more base tables with the same join keys, we perform a hash join; otherwise, if the index is built on the join key, we perform index join. If neither of these two methods is applicable, then a nested-loop join is performed by default.

5.2.2 Gather. This operator, which is essential for parallel SN evaluation, is executed by each worker evaluating the branch specified by lines 12 to 18 in Algorithm 2. Its functionality is to collect tuples newly produced by other workers and update the index structure on the recursive table. Then the delta table to be used in the next iteration of SN is generated.

5.2.3 Distribute. The Distribute operator is called by a worker when the join operation during a local iteration of the SN evaluation is completed. Specifically, the operator will split the join results into disjoint partitions according to the same pre-defined hash mechanism. Then the partitions are sent to the memory buffer of different workers accordingly. In the process described by Algorithm 2, Distribute is executed right before the application of Gather.

For example, Figure 5 illustrates thee physical plan for *Query 2* (Connected Component). The figure shows that the recursive table *cc2* is partitioned by its first attribute, and the base table *arc* should construct an index on its first attribute due to the join key. The dashed arrows here denote coordination among different workers, which are controlled by the Distribute and Gather operators here. Specifically, the Distribute operator distributes the results of SN evaluation to other workers; while the Gather operator merges the results into $cc2$ and generates $\delta cc2$ for each worker.

We next describe this physical plan in more details. First, the base table arc is sent to the Distribute and Gather operators to initialize the recursive table $cc2$ along with the delta table $\delta cc2$. Then δcc will be joined with arc. In this example, an index join algorithm is used since there is an index built on the join key. Next, the join results are dispatched to different workers by the Distribute and Gather operators. Then the Gather operator merges the aggregated results into $cc2$ and generates the new deltas $\delta cc2$. This procedure is repeated until the fixpoint is reached. Finally, table cc will be taken directly from the recursive table $cc2$.

Aggregate operators, such as min or count, specified in the head of rules can also be supported naturally in this framework, since their implementation becomes part of the Gather operator, and the Distribute operators also perform some partial aggregation to expedite the overall evaluation.

6 SYSTEM IMPLEMENTATION AND OPTIMIZATIONS

In this section, we propose several techniques to improve the overall performance. We will first discuss the system implementation issues regarding parallel execution (Section 6.1) and then propose various optimizations used for aggregates and the SN computation (Section 6.2).

6.1 Implementation of Parallel Execution

In previous studies, the lock mechanism was used to deal with race conditions occurring in some complex queries. Specifically, when worker W_i needs to update the buffer memory belonging to W_j ($i \neq j$), the operating system will apply a lock to protect the complex information describing the critical section, and this requires many system calls. As a result, the overall performance will suffer from locks with coarse granularity and the degree of parallelism can be significantly harmed.

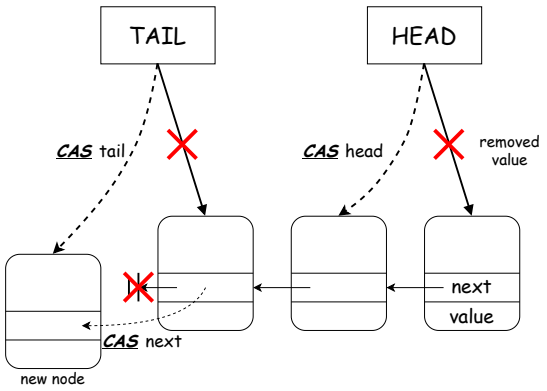


Figure 6: SPSC queue operations

In DCDatalog, we address this issue by a light-weight data structure that dovetails with the DWS coordination strategy. As discussed in Section 4, DWS alleviates idle waiting by relaxing the constraint of global coordination among workers. Next, we will show that this strategy enables us to adopt a finer granularity mechanism that uses atomic operations instead of locks to handle race conditions. During the parallel evaluation using DWS, a worker W_i that wants to update the memory buffer of W_j only needs to append information to the memory space M_j^i; then W_j can collect all contents from $\mathcal{M}_j$ in one operation. In the above process, the race condition happens only between two workers. This enabled us to design the Single Producer Single Consumer (SPSC) Queue to implement it as a light-weight data structure. In fact, the SPSC Queue is a ring array whose head and tail are maintained via atomic operations. The detailed procedures are shown in Figure 6.

As discussed in Section 4, the parallel evaluation terminates when the global fixpoint is reached. With the help of the SPSC Queue, we can detect this event in an efficient way by checking whether: (i) all workers are inactive; and (ii) all the memory buffers $\mathcal{M}_i$ are empty. To validate (i), we maintain a global variable that counts the number of inactive workers. To ensure that all buffers are empty, we maintain one global variable to record the total number of tuples that have been produced and sent into the buffer of all workers; for each worker, we maintain a local variable to count the number of tuples that have been processed by the worker. When the global count is equal to the sum of local ones, then all tuples have been processed and the buffers are empty.

6.2 Optimization Techniques

6.2.1 Improving Aggregates in Recursion. One of the most expensive operations during the evaluation is computing aggregates in

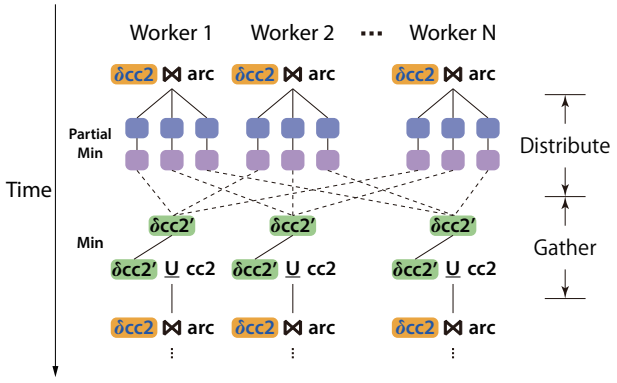


Figure 7: The Computation of Aggregates

recursion. The process of parallel computation of aggregates is shown in Figure 7. It requires to first compute partial aggregate results on each worker and then merge them with existing results. In this process, a linear scan on the deduplicated recursive table of each worker is required. To reduce the memory and computation overhead in this process, we include essential information regarding aggregates besides join keys in the index structure. Specifically, for `min` and `max` aggregates, the key is the attribute on which the aggregate is applied. For the aggregates `count` and `sum`, we instead need to build two index structures: one on the group-by key, the other on the attribute value that is incrementally being computed. In this way, the aggregates can be computed by traversing the index instead of conducting a linear scan on the workers.

Example 6.1. We use the example in Figure 7 to illustrate the computation of aggregates in the *CC* program, whose physical plan is displayed in Figure 5. After executing the Join and Project operators, the intermediate results for $\Pi_{Y,Z}(\delta cc2(X,Z) \bowtie arc(X,Y))$ are produced and split into different partitions (indigo square) based on Y. The partial aggregation is calculated on each partition by sorting the above intermediate results in ascending order based on Y and Z. Then results of partial aggregation (violet square) are obtained by keeping only the minimum Z for each Y and sent it to other workers.

Once the worker finishes collecting the partial aggregated result $\delta cc2$', the Gather operator merges such results into the recursive relation and updates the index. Say that the partial results $\delta cc2$' collected from three workers for worker W_1 are $\{[1,2],[2,4]\}$, $\{[1,1],[2,1]\}$ and $\{[1,2],[2,3]\}$, respectively, and that the recursive relation *cc2* contains $\{[1,4],[2,3]\}$ before merging. Then, we first retrieve $[1,2]$ from the B^+ tree index on *cc2* and find the tuple $[1,4]$ with same key 1 but a larger value 4. Thus, relation *cc2* and its index will be updated to $\{[1,2],[2,3]\}$. Meanwhile, we also add $[1,2]$ into the delta relation. For tuple $[2,4]$, its value is larger than that in *cc2* ($[2,3]$) and thus it will be ignored. We repeat the above procedure, update *cc2* along with its index and generate $\delta cc2$ for the next iteration. In this way, the cost for calculating aggregation can be reduced by looking up keys in the index.

For instance, to help compute `min` in *CC* (*Query 2*), we build an index on table *cc2* with Y as the key. To help compute `sum` in *PageRank* (*Query 6*), we build two indices on table *rank*: The first index is utilized to find and update the partial value, where $\langle X, Y\rangle$ is the key and K is the value. The second index is utilized for updating the aggregated value, where X is the key and $\sum K$ on X is the value.

Query 6 PageRank (PR)

$$
\begin{aligned}
&r_{6,1}: && rank(X, sum\langle(X,I)\rangle) && \leftarrow && matrix(X,\_,\_),\\
& && && && I = (1-\alpha)/VNUM.\\
&r_{6,2}: && rank(X, sum\langle(Y,K)\rangle) && \leftarrow && rank(Y,C),\\
& && && && matrix(Y,X,D),\\
& && && && K = \alpha * (C/D).\\
&r_{6,3}: && results(X,V) && \leftarrow && rank(X,V).
\end{aligned}
$$

6.2.2 Using Cache to Optimize Existence Checking. At each iteration of the SN evaluation, set union and set difference operations are performed to update the recursive tables and the indices associated with them. An essential operation that must be performed during this process is checking whether a key exists in the recursive table, which is realized by checking the B^+ Tree index. Given a tuple in the newly generated delta table, if its key is already presented in the index, then we ignore the tuple. Otherwise, we will insert it into the recursive table and the index. To reduce the overhead of checking the indexes which requires logarithm time, we utilize an additional cache structure for each worker. Then when checking the tuples, we first look up the cache in constant time. If the key is already there, we ignore the tuple; otherwise, we proceed to check the index. Note that this cache structure does not involve much memory overhead, especially for aggregates. Actually, the cache only needs to maintain its group-by key and the corresponding aggregations. That means we only need to keep an array of these pairs, where, typically, their total number is just a small percentage of the number of key values in the dataset.

7 EVALUATION

7.1 Experiment Setup

Table 1: Statistics of Datasets

Name	# Vertices	# Edges	Size
LIVEJOURNAL	4,847,572	68,993,773	527 MB
ORKUT	3,072,441	117,185,083	895 MB
ARABIC	22,744,080	639,999,458	4.8 GB
TWITTER	41,652,231	1,468,365,182	11 GB

7.1.1 Benchmark Programs and Datasets. To evaluate our proposed DCDatalog engine, we conduct experiments using five Datalog programs which were widely used in previous studies. The first two are *Same Generation (SG)* (*Query 5*) and *Delivery* (*Query 8*) [1] that were popular examples for deductive database; the other three express the following popular graph algorithms: *Connected Component* (*Query 2*), *PageRank* (*Query 6*) and *Single Source Shortest Path* (*Query 7*).

Query 7 Single Source Shortest Path (SSSP)

$$
\begin{aligned}
&r_{7,1}: && sp(To, min\langle C\rangle) && \leftarrow && To = start, C = 0.\\
&r_{7,2}: && sp(To2, min\langle C\rangle) && \leftarrow && sp(To1, C1),\\
& && && && warc(To1, To2, C2),\\
& && && && C = C1 + C2.\\
&r_{7,3}: && results(To, min\langle C\rangle) && \leftarrow && sp(To, C).
\end{aligned}
$$

Query 8 BoM – Delivery

$$
\begin{array}{lrcl}
r_{8,1}: & delivery(P, max\langle D\rangle) & \leftarrow & basic(P, D). \\
r_{8,2}: & delivery(P, max\langle D\rangle) & \leftarrow & assbl(P, S), \\
 & & & delivery(S, D). \\
r_{8,3}: & results(P, max\langle D\rangle) & \leftarrow & delivery(P, D).
\end{array}
$$

We evaluate all three graph queries on four real world datasets Livejournal, Orkut, Arabic and Twitter, whose detailed statistics are shown in Table 1. Moreover, we evaluate the first two queries on the synthetic datasets used in previous studies [16, 24, 43]: Tree-11 is a tree of height 11, where the degree of each non-leaf vertex is a random number between 2 and 6. G-10K is a 10,000-vertex random graph[3] generated by randomly connecting vertices so that the probability that a pair of nodes is directly connected by an edge is 0.001. The Rmat-n graphs are generated by the RMAT graph generator, which has n vertices and $10 \times n$ directed edges. The N-n are trees with n vertices, which are generated in different levels following [24]: each tree node has randomly 5 to 10 children, and each child has a 20% to 60% chance of becoming a leaf.

7.1.2 Baseline Systems. We used the following Datalog engines designed for shared-memory multicore architectures as the baseline for our work: SociaLite [40], DeALS-MC [52], DDlog [37], Souffle [38] equipped with the concurrent index [27] and RecStep [16]. For these systems, we obtained the source code of DeALS-MC and SociaLite from the original authors, while the source code for Souffle [4], DDlog [5], and RecStep [6] is publicly available.

The rationale for focusing on those Datalog systems, excluding a few others from our comparisons, is based on the following considerations. We exclude the shared-nothing based systems [43, 46, 49] as they have different problem settings. It was shown in [52] that the single-node based DeALS [44] and LogicBlox [7] cannot outperform DeALS-MC. Furthermore, we did not consider specialized non-Datalog systems such as graph systems.

We use the end-to-end query execution time as the metric for evaluation. Since in this paper we focus on in-memory computation, we exclude the time of loading data from disk for all the systems (which is rather trivial for DCDatalog). We run all the experiments 5 times and report the average results. If a system cannot finish within 10 hours under a particular setting, we regard that as timeout.

7.1.3 Environment. We implement the DCDatalog engine with C++. We run the experiments of all the systems on a server with four AMD Opteron 6376 CPUs (8 physical cores per CPU, 2 hyper-thread per core), 256GB memory (configured into eight NUMA regions) and 1 TB hard disk. The operating system is Ubuntu Linux 14.04 LTS and the compiler is GCC 9.0 with O3 flag.

7.2 End-to-end Query Time Comparison

The comparison with existing Datalog engines produces the results shown in Table 2. For the two recursive queries *SG* and *Delivery*, we find that DCDatalog achieves 3 to 100 times performance gain over the baselines. For example, for the *SG* query on G-10K dataset, the times for Souffle, RecStep, DeALS-MC, SociaLite and DDlog are 194.09, 458.41, 76.18, 4762.25 and 285.78 seconds, respectively. Meanwhile, DCDatalog takes only 15.95 seconds. The superior performance of DCDatalog comes from the comprehensive optimizations made in all components of the system. A separate issue is that some of the language constructs of DCDatalog are not supported in other systems. For instance, Souffle does not allow aggregates in recursion, and thus it must use a stratified query that results in very poor performance for the *Delivery* query. On the other hand, the performance improvements achieved for DeALS-MC must be credited directly to the efficient implementation of DWS provided by DCDatalog which thus reduces idle waiting significantly. The performance of SociaLite queries underscores that the system was optimized for social network applications rather than general-purpose Datalog queries. For RecStep, the source code released by the author does not include the claimed PBME optimizations in [16]. Therefore, we just report the results we obtained from their currently released version, which are likely to be worse than those reported in the original paper. Also, DCDatalog significantly outperforms DDlog, for a number of reasons including the fact that the latter does not provide a good strategy for optimizing parallel execution and also it does not optimize memory usage, whereby it is prone to OOM problems.

If we now study the results produce by the three graph algorithms *CC*, *SSSP* and *PageRank*, we detect trends that are similar to those observed in the two recursive queries we just discussed. Many baseline systems, such as DeALS-MC and RecStep, cannot support *PageRank* because they do not allow the use of the `sum` aggregate in recursion. Also, Souffle runs out of memory on all graph queries because the equivalent stratified queries involve too many intermediate results. Compared with other baseline systems, DCDatalog has both great expressive power and performance because it relaxes the constraint on lock-free programs and uses a light-weight scheme to deal with race conditions that often happen in during the evaluation of more complex programs that use aggregates in recursion. For instance, for the *SSSP* query on Orkut dataset, the times for RecStep, DeALS-MC, SociaLite and DDlog are 88.01, 361.71, 36.84, 611.01 seconds, respectively; while that for DCDatalog is 8.60 seconds.

In passing, we also report our findings on memory usage. For instance, the peak memory usage of DCDatalog for *CC* query on Livejournal, Orkut, Arabic, Twitter datasets is 2.50, 3.45, 17.68, 45.95 GB, respectively. In fact, the memory is just logically partitioned across different workers, and thus the partitioned memory is not exclusive to the worker itself physically. As a result, the memory usage of DCDatalog is within a reasonable range.

We also conduct the additional experiment for Query 3 *APSP*, which is a typical non-linear query. Here we ignore the comparisons with DeALS-MC and RecStep since they do not support queries with non-linear recursions, while Souffle runs out of memory for the same reasons as those we previously discussed for other queries with aggregates. From the results in Table 3, we see that DCDatalog still outperforms other systems under most settings. For example, on the Rmat-256 dataset, the time for SociaLite and DDlog is, respectively, 69.69 and 111.738 seconds while that of DCDatalog is only 0.47 seconds. We believe that the main reason for such dramatic difference is that other systems typically broadcast the new tuples in relation *path* to all other partitions, while DCDatalog

[3] http://www.cse.psu.edu/ kxm85/software/GTgraph
[4] https://souffle-lang.github.io/
[5] https://github.com/vmware/differential-datalog
[6] https://github.com/Hacker0912/RecStep

Table 2: Comparison with State-of-the-art Systems (seconds): OOM means out of memory; NS means the system does not support the corresponding query; TO means timeout

Query	Dataset	DCDatalog	SociaLite	DeALS-MC	Souffle	RecStep	DDlog
SG	Tree-11	40.37	30687.42	71.99	1438.98	OOM	OOM
	G-10K	15.95	4762.25	76.18	194.09	458.41	285.78
	Rmat-10K	12.02	5013.76	80.11	143.46	512.48	184.57
	Rmat-20K	54.33	21048.49	299.16	664.65	2378.16	728.15
	Rmat-40K	231.56	TO	1358.42	2879.03	OOM	OOM
Delivery	N-40M	3.27	233.71	NS	88.06	40.26	163.03
	N-80M	5.07	854.73	NS	167.67	71.71	313.24
	N-160M	11.01	2332.05	NS	369.81	154.13	741.26
	N-300M	18.37	8170.65	NS	729.52	334.43	OOM
CC	Livejournal	8.44	31.70	319.88	OOM	55.12	556.90
	Orkut	11.02	40.91	379.30	OOM	49.41	942.60
	Arabic	50.31	184.55	OOM	OOM	495.54	OOM
	Twitter	77.22	TO	OOM	OOM	637.51	OOM
SSSP	Livejournal	11.82	42.36	791.83	OOM	212.50	891.49
	Orkut	8.60	36.84	361.71	OOM	88.01	611.01
	Arabic	9.83	61.69	OOM	OOM	113.96	OOM
	Twitter	23.79	TO	OOM	OOM	178.24	OOM
PageRank	Livejournal	112.29	12339.52	NS	OOM	NS	2295.93
	Orkut	45.45	4770.41	NS	OOM	NS	1672.18
	Arabic	202.81	TO	NS	OOM	NS	OOM
	Twitter	2008.95	TO	NS	OOM	NS	OOM

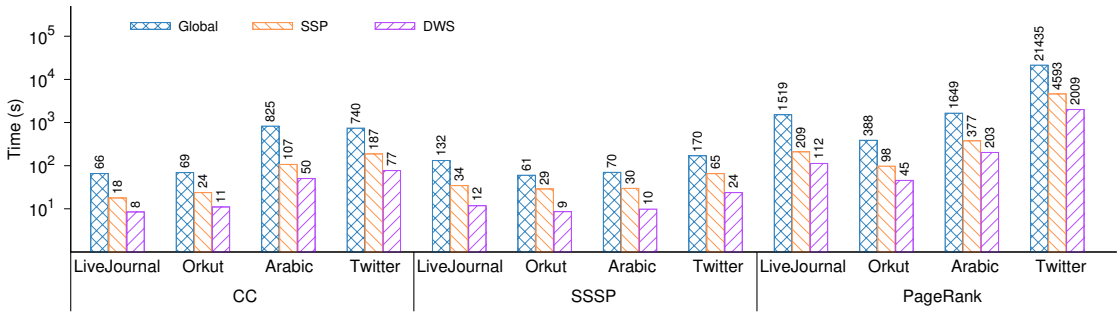


Figure 8: Effect of Different Coordination Strategies

Table 3: Results for the APSP Query

APSP	DCDatalog	SociaLite	DDlog
Rmat-256	0.47	68.69	111.74
Rmat-512	1.35	2517.42	1560.47
Rmat-1K	5.99	OOM	OOM
Rmat-2K	80.13	OOM	OOM
Rmat-4K	317.02	OOM	OOM

only needs to replicate and route newly generated tuples to two partitions, thus reducing both communication and computation costs. Moreover, the fact that the results of *APSP* nearly contain all possible pairs of input vertices generate memory and computation burdens for SociaLite and DDlog when broadcasting new tuples. The other comprehensive optimization techniques of DCDatalog are also contributing to its much better performance.

7.3 Results of Ablation Study

Table 4: Effect of Optimization Techniques

	CC		*SSSP*	
	w/o	w/	w/o	w/
Livejournal	16.11	8.44	29.50	11.82
Orkut	25.41	11.02	23.03	8.60
Arabic	105.64	50.31	18.32	9.83
Twitter	224.81	77.22	58.03	23.79

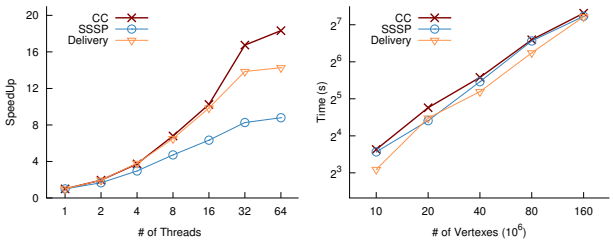
In this Section, we report the results of ablation studies conducted to evaluate the effect of each proposed technique.

Parallel Coordination Strategy To test the effect of different coordination strategies, we consider the following three methods: Global is the method that requires a coordination after each global

iteration; SSP simply extends the techniques proposed in [11, 14] which allows fast workers to proceed at most s iterations. Details of above two methods have been covered in Section 4.1. DWS is our proposed dynamic coordination strategy proposed in Section 4.2. In this experiment, we set the value of s to 5 empirically, since this produces the best performance under most settings. The results of different coordination strategies are shown in Figure 8. We see that DWS achieves the best performance under all settings. For example, for the *SSSP* query on LIVEJOURNAL dataset, the time for Global, SSP and DWS is 131.68, 34.45 and 11.82 seconds, respectively. SSP performs worse than DWS because it relies on a predefined threshold s to avoid the idle waiting of faster workers, which fails to reflect the characteristics of different iterations during the evaluation. Global has the worst performance as it suffers from the idle waiting involved in the parallel evaluation. Note that Global uses the same coordination strategy as DeALS-MC but is equipped with the better implementation techniques introduced in Section 6. Therefore, its general performance is better than DeALS-MC due to the benefits of our designs.

Effect of Optimizing Implementation Next, we evaluate the optimizations proposed in Section 6.2 by eliminating the proposed techniques(w/o optimization) and comparing with the fully optimized DCDatalog engine (w/ optimization). Due to space limitations, we only report the results of *CC* and *SSSP* here. We observe from Table 4 that a 1.86 to 2.91 times performance gain was achieved by applying the optimizations. This result confirms the merits of the proposed coordination strategy and implementation features which enabled DCDatalog to achieve top state-of-the-art performance.

7.4 Scalability



(a) Varying # Threads (b) Varying Data Size

Figure 9: Scalability

We first conduct experiments on scaling-up the number of workers (threads). We vary the number of threads from 1 to 64 and evaluate the *CC*, *SSSP* and *Delivery* programs on datasets LIVEJOURNAL, ARABIC, and N-300M. The results are shown in Figure 9(a). We observe that DCDatalog scales well using up to 32 threads. After that, the speedup tends to stabilize because the number of physical cores on the machine is 32 and they are fully utilized. The *SSSP* query achieves smaller speedups since its evaluation starts from one vertex and the speed-ups of parallelism materialize only after many vertices are reached.

Finally we increase the dataset size and observe how our system scales over large volumes of data. Our synthetic graph datasets RMAT-n have size ranging from 10M to 160M obtained by varying the number of vertices in the graph. Figure 9(b) shows the results of *CC*, *SSSP* and *Delivery* programs. We observe that the time increases proportionally to the size of the dataset. For example, the execution time of *CC* is 12.39, 27.08, 47.76, 96.61, 158.82 seconds on the synthetic graph with 10M, 20M, 40M, 80M, 160M vertices, respectively. These results suggest that DCDatalog can potentially deal with ever larger datasets on modern multicore machines.

8 RELATED WORK

8.1 Datalog Language and Evaluation

Supporting aggregates in recursive Datalog programs is an old and difficult problem which has been the topic of much previous research work. Earlier studies tried to reach this goal by providing formal semantics for recursive Datalog programs with unstratified aggregates [17, 18, 34]. In particular, Ross et al. [36] used semantics based on specialized lattices to express the four aggregates, while Ganguly et al. [19] sought to optimize programs with extrema. Mazuran et al. [32] proposed the monotonic aggregates and proved that they can be used freely in recursion. More recently, Zaniolo et al. [55] introduced the Pre-mappability(PREM) and PCC [54] properties under which programs using aggregates in recursion are equivalent to those aggregate-stratified ones. This led to the design and implementation of RASQL, an extension of SQL that supports extrema aggregates in recursion [24, 48].

There is a long stream of studies about supporting parallel evaluation of recursive Datalog programs. Wolfson et al. [50] identified the decomposable programs which can be evaluated in parallel without communication and duplicated computation. The parallel SN evaluation fixpoint was proposed in [20] for message passing. Seib et al. [39] provided the Generalized Pivoting to distribute the workload of a Datalog program for parallel execution and Ganguly et al. [21] proposed the substitution partitioned parallelization scheme. Shaw et al. [42] and Afrati et al. [4, 5] studied how to support Datalog evaluation under MapReduce. Motik et al. [33] focused on the specific problem of RDF data. All above studies focused on interesting theoretical results rather than concrete system implementations.

8.2 Datalog Systems and Applications

Many efforts focused on designing and implementing efficient engines for Datalog evaluation. LogicBlox [7] designed the Datalog engine using ideas and techniques inspired by relational DBMS. DEALS [44] implemented the idea of monotonic aggregation to efficiently support aggregates in recursion. There are also various special-purpose systems that use Datalog-like interfaces due their ability of expressing succinctly a wide spectrum of applications, such as knowledge reasoning [9], data mining [29], machine learning [47] and data center management [56].

To deal with large-scale analytical queries, another major line of studies focuses on Datalog engines in shared-nothing environment. For instance, Distributed SociaLite [41] extended its single node version [40] to distributed setting with message passing to communicate. Myria [46] proposed an asynchronous approach for Datalog evaluation. BigDatalog [43] developed the Datalog engine on top of Apache Spark. MRA [49] proposed program analysis and query engine for synchronization in distributed environment.

Among shared-memory systems, DeALS-MC [52] implements and optimizes the idea of *substitution partitioned parallelization* for lock-free programs. However, it has certain limitations of performance due to its coordination strategy, which has been detailed in Section 2.2. Souffle [27, 38] is a Datalog engine designed with concurrent B-Tree indexes. It does not support aggregates in recursion and thus cannot express many advanced analytical queries. RecStep [16] is a parallel Datalog engine implemented on top a parallel relational database system named QuickStep [35], which is responsible to support the parallel execution of Datalog programs. The RecStep engine itself did not propose techniques for improving the parallel evaluation which is the focus of this paper.

8.3 Parallel Query Evaluation

In the past years, many distributed big data platforms have been developed to cope with the ever-increasing volume of data collections. For distributed big data platforms, a critical bottleneck is the synchronization mechanism over all workers. The Bulk Synchronous Parallel (BSP) model is the most popular one for distributed computation. Under BSP, iterative computation is separated into super steps, and messages from one super step are only accessible in the step that immediately follows it. It has been adopted by both general purpose systems like Apache Spark [53] and graph processing systems, such as Pregel [31] and GraphX [23]. BSP has been adopted by both general purpose systems like Apache Spark [53] and graph processing systems, such as Pregel [31] and GraphX [23]. To alleviate the overhead of synchronization of BSP, other systems, including GraphLab [22] and Giraph++ [45], adopted the Asynchronous Processing (AP) model. Some follow-up studies [11, 15, 25, 51] investigated trade-offs between AP and BSP and proposed new synchronization techniques, which can reduce both the cost of global synchronization and communication overheads. All above strategies are designed for applications of graph analysis or machine learning in the shared-nothing environment, which is not the focus of our study. Investigating possible extensions of these techniques to our problem represents an interesting direction for future work.

9 CONCLUSION

In this paper, we introduce DCDatalog, a parallel Datalog engine for shared-memory multicore architectures. DCDatalog is equipped with a light-weight scheme to resolve race conditions in parallel execution, thus enabling more efficient evaluation for a broad range of Datalog applications. We propose a novel dynamic coordination strategy to overcome the limitations of existing approaches for parallel Datalog evaluation. The proposed strategy significantly reduces the idle waiting time and brings additional benefits to recursive queries. Furthermore, we proposed and implemented several query planning and optimization techniques to improve the performance of recursive Datalog programs. Experimental results on several real datasets demonstrate the superior efficiency and scalability of DCDatalog compared with other alternatives. For the future work, we will investigate how SQL and other query languages supporting recursion could also benefit from these advances.

REFERENCES

[1] [n.d.]. Recursion Example: Bill Of Materials. https://www.ibm.com/support/knowledgecenter/en/SS6NHC/com.ibm.swg.im.dashdb.sql.ref.doc/doc/r0059242.html.

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *SIGMOD*. 431–446.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[4] Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. 2011. Map-reduce extensions and recursive queries. In *EDBT*. 1–8.

[5] Foto N. Afrati and Jeffrey D. Ullman. 2012. Transitive closure and recursive Datalog implemented on clusters. In *EDBT*. 132–143.

[6] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony I. T. Rowstron. 2013. Scale-up vs scale-out for Hadoop: time to rethink?. In *SOCC*. 20:1–20:13.

[7] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*. 1371–1382.

[8] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. 2003. The Deductive Database System LDL++. *TPLP* 3, 1 (2003), 61–94.

[9] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *PVLDB* 11, 9 (2018), 975–987.

[10] Robert B Cooper. 1981. Queueing theory. In *Proceedings of the ACM'81 conference*. 119–122.

[11] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX ATC*. 37–48.

[12] Adnan Darwiche. 2020. Three Modern Roles for Logic in AI. In *PODS*. 229–243.

[13] Ariyam Das, Youfu Li, Jin Wang, Mingda Li, and Carlo Zaniolo. 2019. BigData Applications from Graph Analytics to Machine Learning by Aggregates in Recursion. In *ICLP*. 273–279.

[14] Ariyam Das and Carlo Zaniolo. 2019. A Case for Stale Synchronous Distributed Model for Declarative Recursive Computation. *TPLP* 19, 5-6 (2019), 1056–1072.

[15] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. 2018. Adaptive Asynchronous Parallelization of Graph Algorithms. In *SIGMOD*. 1141–1156.

[16] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. *PVLDB* 12, 6 (2019), 695–708.

[17] Filippo Furfaro, Sergio Greco, Sumit Ganguly, and Carlo Zaniolo. 2002. Pushing extrema aggregates to optimize logic queries. *Inf. Syst.* 27, 5 (2002), 321–343.

[18] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. 1991. Minimum and Maximum Predicates in Logic Programming. In *PODS*. 154–163.

[19] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. 1995. Extrema Predicates in Deductive Databases. *J. Comput. Syst. Sci.* 51, 2 (1995), 244–259.

[20] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. 1990. A Framework for the Parallel Processing of Datalog Queries. In *SIGMOD*. 143–152.

[21] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. 1992. Parallel Bottom-Up Processing of Datalog Queries. *J. Log. Program.* 14, 1&2 (1992), 101–126.

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*. 17–30.

[23] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*. 599–613.

[24] Jiaqi Gu, Yugo Watanabe, William Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. 2019. RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark. In *SIGMOD*. 467–484.

[25] Minyang Han and Khuzaima Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *PVLDB* 8, 9 (2015), 950–961.

[26] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*. 1223–1231.

[27] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. A specialized B-tree for concurrent datalog evaluation. In *PPoPP*. 327–339.

[28] J. F. C. Kingman. 1961. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society* 57, 4 (1961), 902–904.

[29] Youfu Li, Jin Wang, Mingda Li, Ariyam Das, Jiaqi Gu, and Carlo Zaniolo. 2021. KDDLog: Performance and Scalability in Knowledge Discovery by Declarative Queries with Aggregates. In *ICDE*. 1260–1271.

[30] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative networking: language, execution and optimization. In *SIGMOD*. 97–108.

[31] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146.

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

[33] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. 2014. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *AAAI*. 129–137.

[34] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. The Magic of Duplicates and Aggregates. In *VLDB*. 264–277.

[35] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *PVLDB* 11, 6 (2018), 663–676.

[36] Kenneth A. Ross and Yehoshua Sagiv. 1992. Monotonic Aggregation in Deductive Databases. In *PODS*. 114–126.

[37] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *LPNMR*, Vol. 2368. 56–67.

[38] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *CC*. 196–206.

[39] Jürgen Seib and Georg Lausen. 1991. Parallelizing Datalog Programs by Generalized Pivoting. In *PODS*. 241–251.

[40] Jiwon Seo, Stephen Guo, and Monica S. Lam. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*. 278–289.

[41] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis. *PVLDB* 6, 14 (2013), 1906–1917.

[42] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. 2012. Optimizing Large-Scale Semi-Naïve Datalog Evaluation in Hadoop. In *Datalog in Academia and Industry*. 165–176.

[43] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *SIGMOD*. 1135–1149.

[44] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *ICDE*. 867–878.

[45] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB* 7, 3 (2013), 193–204.

[46] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. 2015. Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines. *PVLDB* 8, 12 (2015), 1542–1553.

[47] Jin Wang, Jiacheng Wu, Mingda Li, Jiaqi Gu, Ariyam Das, and Carlo Zaniolo. 2021. Formal semantics and high performance in declarative machine learning using Datalog. *VLDB J.* 30, 5 (2021), 859–881.

[48] Jin Wang, Guorui Xiao, Jiaqi Gu, Jiacheng Wu, and Carlo Zaniolo. 2020. RASQL: A Powerful Language and its System for Big Data Applications. In *SIGMOD*. 2673–2676.

[49] Qiange Wang, Yanfeng Zhang, Hao Wang, Liang Geng, Rubao Lee, Xiaodong Zhang, and Ge Yu. 2020. Automating Incremental and Asynchronous Evaluation for Recursive Aggregate Data Processing. In *SIGMOD*. 2439–2454.

[50] Ouri Wolfson and Abraham Silberschatz. 1988. Distributed Processing of Logic Programs. In *SIGMOD*. 329–336.

[51] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *PPoPP*. 194–204.

[52] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. 2017. Scaling up the performance of more powerful Datalog systems on multicore machines. *VLDB J.* 26, 2 (2017), 229–248.

[53] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28.

[54] Carlo Zaniolo, Ariyam Das, Jiaqi Gu, Youfu Li, Mingda Li, and Jin Wang. 2019. Monotonic Properties of Completed Aggregates in Recursive Queries. *CoRR* abs/1910.08888 (2019).

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

[56] Qizhen Zhang, Akash Acharya, Hongzhi Chen, Simran Arora, Ang Chen, Vincent Liu, and Boon Thau Loo. 2019. Optimizing Declarative Graph Queries at Large Scale. In *SIGMOD*. 1411–1428.