Parallel Bottom-Up Evaluation of Logic Programs: DeALS on Shared-Memory Multicore Machines

MOHAN YANG, ALEXANDER SHKAPSKY, CARLO ZANIOLO

Technical Report #150003
UCLA CS Department
E-mail: {yang, shkapsky, zaniolo}@cs.ucla.edu

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Delivering superior expressive power over RDBMS, while maintaining competitive performance, has represented the main goal and technical challenge for deductive database research since its inception forty years ago. Significant progress toward this ambitious goal is being achieved by the DeALS system through the parallel bottom-up evaluation of logic programs, including recursive programs with monotonic aggregates, on a shared-memory multicore machine.

In DeALS, a program is represented as an AND/OR tree, where the parallel evaluation instantiates multiple copies of the same AND/OR tree that access the tables in the database concurrently. Synchronization methods such as locks are used to ensure the correctness of the evaluation. We describe a technique which finds an efficient hash partitioning strategy of the tables that minimizes the use of locks during the evaluation. Experimental results demonstrate the effectiveness of the proposed technique — DeALS achieves competitive performance on non-recursive programs compared with commercial RDBMSs and superior performance on recursive programs compared with other existing systems.

KEYWORDS: parallel, bottom-up evaluation, Datalog, multicore, AND/OR tree

1 Introduction

There has been much research on improving the performance of Datalog systems through parallel bottom-up evaluation. Previous studies focused on the message passing model where processors communicate with each other by exchanging messages. This includes both strategies for programs that can be evaluated without any communication (Wolfson and Silberschatz 1988; Wolfson 1988; Cohen and Wolfson 1989; Seib and Lausen 1991) and strategies to minimize the amount of communication required (Ganguly et al. 1992; Zhang et al. 1995; Ganguly et al. 1995). In this paper we instead assume the shared-memory model, where the data is stored in shared-memory that can be directly accessed by all processors, as is supported by most modern multicore machines, rather than explicit exchange of messages through a shared segment of memory as studied in (Wolfson and Silberschatz 1988; Ganguly et al. 1992). The shared data may be modified by multiple processors

concurrently, and synchronization methods such as locks are used to ensure the correctness of the evaluation.

The key to achieving a good performance in the shared-memory model is to use as little synchronization as possible in the program evaluation. In this paper, we present the technique used by the ***De**ductive **A**pplication **L**anguage **S**ystem* ($\mathcal{DeALS}$)[1] under development at UCLA, which extends the $\mathcal{LDL}$++ technology (Arni et al. 2003), to support the parallel bottom-up evaluation of Datalog programs on shared-memory multicore machines. The proposed technique produces efficient evaluation plans for both non-recursive programs and recursive programs. $\mathcal{DeALS}$ delivers competitive performance on the non-recursive queries of the TPC-H benchmark[2], compared with the state of the art RDBMSs such as Vectorwise[3] and SQL Server[4], and superior performance on recursive programs compared with other existing systems.

The rest of this paper is organized as follows. We introduce the concept of lock-free programs in Section 2. We describe how $\mathcal{DeALS}$ tries to find a lock-free evaluation plan in Section 3. An overview of $\mathcal{DeALS}$ is presented in Section 4. We report experimental results in Section 5. We present a sufficient condition for a program to be a lock-free program and a rewriting from an arbitrary program to a lock-free program in Section 6, and some discussion about lock-free programs in Section 7. Related work is discussed in Section 8. The paper concludes in Section 9.

2 Lock-free Programs

Let `arc` be a base relation that represents the edges of a directed graph. The transitive closure `tc` of `arc` is a derived relation that contains all the pairs $(\mathtt{X}, \mathtt{Y})$ where there is a path from `X` to `Y` in the graph. The following program is used to compute `tc`.

$$\begin{aligned} &r1.1 : \mathtt{tc}(\mathtt{X}, \mathtt{Y}) \texttt{ <- } \mathtt{arc}(\mathtt{X}, \mathtt{Y}). \\ &r1.2 : \mathtt{tc}(\mathtt{X}, \mathtt{Y}) \texttt{ <- } \mathtt{tc}(\mathtt{X}, \mathtt{Z}), \mathtt{arc}(\mathtt{Z}, \mathtt{Y}). \end{aligned} \tag{1}$$

The bottom-up evaluation of this program works as follows. The exit rule $r1.1$ is evaluated first. A tuple $(\mathtt{X}, \mathtt{Y})$ is added to `tc` for each tuple $(\mathtt{X}, \mathtt{Y})$ in `arc`. Then the left-linear recursive rule $r1.2$ is evaluated. For each tuple $(\mathtt{X}, \mathtt{Z})$ in `tc`, a tuple $(\mathtt{X}, \mathtt{Y})$ is added to `tc` for each tuple of the form $(\mathtt{Z}, \mathtt{Y})$ in `arc`.[5] $r1.2$ is repeatedly evaluated until `tc` does not change between two successive evaluations of $r1.2$.

Now we describe how to parallelize the bottom-up evaluation on a shared-memory machine with n processors. The parallel evaluation strategy presented in this paper uses the same workflow as the sequential bottom-up evaluation to ensure the correctness of evaluation, while the parallelism is achieved through the parallel evaluation of each single rule (including the parallel pipelined evaluation of all the

[1] Deductive Application Language System, `http://wis.cs.ucla.edu/deals/`.
[2] TPC-H, `http://www.tpc.org/tpch/`.
[3] Vectorwise, `http://www.actian.com/`.
[4] SQL Server 2014, `http://www.microsoft.com/en-us/server-cloud/products/sql-server/`.
[5] Naive evaluation is used here for simplicity. The technique described in this paper also applies to the semi-naive evaluation.

rules which support its goal as described in Section 3). As shown in our experiments, our strategy is able to achieve a reasonable speedup for a data intensive application to the point that we do not need to explore rules level and components level parallelism (Perri et al. 2013).

We divide each relation into n partitions and we use the relation name with a superscript i to denote the i-th partition of the relation. For each partition, we use a lock to ensure the atomicity of each write operation if multiple write operations can occur concurrently. Let h be a hash function that maps a vertex to an integer between 1 to n. Both `arc` and `tc` are partitioned by the first column, i.e., $h(\texttt{X}) = i$ for each $(\texttt{X}, \texttt{Y})$ in $\texttt{arc}^i$ and $h(\texttt{X}) = i$ for each $(\texttt{X}, \texttt{Y})$ in $\texttt{tc}^i$. Assuming that there are one coordinator and n workers, the parallel evaluation proceeds as follows.

(1) The i-th worker evaluates $r1.1$ by adding a tuple $(\texttt{X}, \texttt{Y})$ to `tc` for each tuple $(\texttt{X}, \texttt{Y})$ in $\texttt{arc}^i$.
(2) Once all workers finish Step (1), the coordinator notifies each worker to start Step (3).
(3) For each tuple $(\texttt{X}, \texttt{Z})$ in $\texttt{tc}^i$, the i-th worker looks for tuples in the form $(\texttt{Z}, \texttt{Y})$ in `arc` and adds a tuple $(\texttt{X}, \texttt{Y})$ to `tc`.
(4) Once all workers finish Step (3), the coordinator checks if the evaluation for `tc` is completed. If so, the computation terminates; otherwise the computation starts from Step (3).

In Step (1) and Step (3), each worker performs its task on one processor while the coordinator waits. Step (2) and Step (4) serve as synchronization barriers.

In Step (1), the i-th worker only writes to $\texttt{tc}^i$ since every tuple $(\texttt{X}, \texttt{Y})$ it adds to `tc` is taken from $\texttt{arc}^i$ where $h(\texttt{X}) = i$. In Step (3), the i-th worker reads from $\texttt{tc}^i$ and only writes to $\texttt{tc}^i$ since every tuple $(\texttt{X}, \texttt{Y})$ it adds to `tc` is derived from a tuple $(\texttt{X}, \texttt{Z})$ in $\texttt{tc}^i$ and a tuple $(\texttt{Z}, \texttt{Y})$ in `arc` where $h(\texttt{X}) = i$. There is no need for locks since $\texttt{tc}^i$ is only accessed by the i-th worker during evaluation. The above evaluation plan is called a *lock-free plan* where no lock is needed, and a program which has a lock-free plan is called a *lock-free program*.

A key factor that enables a lock-free plan is the partitioning strategy. If we keep the current partitioning for `arc` but instead partition `tc` by its second column, then every worker could write to $\texttt{tc}^i$ in Step (3), and an *exclusive lock* (x-lock) is needed to ensure every write operation to $\texttt{tc}^i$ is atomic. For some programs, it is possible that a worker reads from a partition of a relation while the same partition is being modified by another worker. In this case, a *readers-writer lock* (rw-lock) is needed to ensure that every write operation is atomic, read operations will not get a tuple that is partially updated, and concurrent read operations are allowed when the partition is not being modified by other workers.

Another key factor is the selection of hash functions. It is possible to get a lock-free plan even if we use different hash functions for `arc` and `tc`. However, in this paper we focus on the case where every relation is partitioned using the same hash function h defined as

$$h(x_1, \cdots, x_t) = \sum_{i=1}^{t} g(x_i) \bmod n, \tag{2}$$

where the input to h is a tuple of any arity t, g is a hash function with a range no less than n, and $\sum$ can be replaced with any commutative function. Using a commutative function to combine the hash values of h allows for more parallelism than an arbitrary function.

Example 1
Consider the following program where `p` is partitioned by its first and second columns and q is partitioned by its first column.

$$\texttt{p(X, Y, Z) <- p(Y, X, W), q(W, Z).} \tag{3}$$

If the i-th worker reads from the i-th partition of `p` where $(g(\texttt{Y}) + g(\texttt{X})) \bmod n = i$, then the i-th worker only writes to the i-th partition of `p` since $(g(\texttt{X})+g(\texttt{Y})) \bmod n = (g(\texttt{Y}) + g(\texttt{X})) \bmod n = i$. Thus, this is a lock-free plan. However, if we replace $\sum$ with a noncommutative function, such as concatenation ($||$) where $h(x_1, \cdots, x_t)$ becomes $g(x_1)|| \cdots ||g(x_t)$, the same plan is not lock free.

Next, we present a systematic approach to find a partitioning strategy that generates a lock-free plan for an arbitrary program when such a plan exists.

Discriminating Sets and Parallel Program Evaluation. Consider the evaluation of a stratifiable program P that consists of l rules $r_1, \cdots, r_l$, where each rule r_i is in the form $p_{i,0}$ <- $p_{i,1}, \cdots, p_{i,m_i}$. Here $p_{i,0}$ is a predicate that serves as the head of r_i, m_i is the number of predicates in the body of r_i, and each $p_{i,j}$ is a predicate in the body for $j = 1, \cdots, m_i$. Assume that a predicate p is associated with a relation of the same name and arity as the predicate, and we use $R(p)$ to denote the relation that stores all tuples corresponding to facts about p in memory. A *discriminating set* of a (non-nullary) relation R is a non-empty subset of columns in R. Given a discriminating set of a relation, we divide the relation into n partitions by the hash value of the columns that belong to the discriminating set.

Let P_j be the program to be executed by the j-th worker. For each rule r_i, we add the following rule in P_j:

$$r_i^j : p_{i,0} \texttt{ <- } p_{i,1}, \cdots, p_{i,m_i}, h(p_{i,t_i}[\overline{X_i}]) = j, \tag{4}$$

where p_{i,t_i} is a predicate selected from the m_i predicates in the body for a t_i between 1 to m_i, $\overline{X_i}$ is a selected discriminating set of $R(p_{i,t_i})$, $p_{i,t_i}[\overline{X_i}]$ denotes a tuple of arity $|\overline{X_i}|$ by retrieving the arguments in p_{i,t_i} whose positions belong to $\overline{X_i}$, and the last predicate $h(p_{i,t_i}[\overline{X_i}]) = j$ means a tuple from the j-th partition of $R(p_{i,t_i})$ is accessed in every successful derivation of r_i^j. We select the same p_{i,t_i} and $\overline{X_i}$ for $j = 1, \cdots, n$. We also select a discriminating set, denoted by $X(R)$, for each derived relation R. These discriminating sets can be arbitrarily selected as long as there is a unique discriminating set for each derived relation. We might have selected several different discriminating sets of the same base relation which correspond to different ways of partitioning the relation. This relation is preprocessed before the evaluation so that it can be efficiently accessed for every partitioning.

In general, the parallel evaluation proceeds as follows.

(1) The coordinator determines the first rule to be evaluated, say r_i, and instructs the workers to start Step (2).

(2) All n workers become active with the j-th worker evaluating r_i^j.
(3) After all workers finish, the coordinator checks if the evaluation for P is completed. If not, it determines the next rule to be evaluated and Step (2) repeats.

The parallel evaluation scheme described above is a special case of the *substitution partitioned scheme* (Ganguly et al. 1992). Our scheme improves it by removing the *sending*, *receiving* and *final pooling* steps since each worker has full access to all the relations in the shared-memory model.

Example 2
The corresponding P_j in the lock-free plan for the program in (1) is shown as follows.

$$\begin{aligned} &r5.1 : \texttt{tc}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{arc}(\texttt{X},\texttt{Y}), h(\texttt{X}) = j. \\ &r5.2 : \texttt{tc}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{tc}(\texttt{X},\texttt{Z}), \texttt{arc}(\texttt{Z},\texttt{Y}), h(\texttt{X}) = j. \end{aligned} \tag{5}$$

For each worker, there are still many different ways to evaluate $r5.2$ since the order of the three predicates in the body is immaterial. We force the same bottom-up evaluation plan upon every worker by using the same *bound/free adornments* (Ullman 1985) for every P_j. A bound/free adornment of a predicate p is a string of `b`'s and `f`'s whose length equals the arity of p. A `b` (An `f`) in the i-th position means the i-th argument in p is bound (free).

Example 3
The program in (6) shows an adorned version of the program in Example 2. The predicates in $r6.2$ are reordered to reflect the order in which they are evaluated (we always reorder the predicates for an adorned program in the rest of this paper). In the evaluation of $r6.2$, the j-th worker evaluates $\texttt{tc}^{\texttt{ff}}(\texttt{X},\texttt{Z}), h(\texttt{X}) = j$ by reading from the j-th partition of `tc`. Then it evaluates $\texttt{arc}^{\texttt{bf}}(\texttt{Z},\texttt{Y})$ where `Z` is bound. So it finds a tuple $(\texttt{Z},\texttt{Y})$ from `arc` with the given `Z`, and adds $(\texttt{X},\texttt{Y})$ to `tc` if it succeeds.

$$\begin{aligned} &r6.1 : \texttt{tc}^{\texttt{ff}}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{arc}^{\texttt{ff}}(\texttt{X},\texttt{Y}), h(\texttt{X}) = j. \\ &r6.2 : \texttt{tc}^{\texttt{ff}}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{tc}^{\texttt{ff}}(\texttt{X},\texttt{Z}), h(\texttt{X}) = j, \texttt{arc}^{\texttt{bf}}(\texttt{Z},\texttt{Y}). \end{aligned} \tag{6}$$

Example 4
The `attend` program below finds all the people who will attend the party. A person will attend the party if he/she is an organizer, or he/she has at least three friends who will attend the party. Here, `mcount` is the monotonic and continuous count aggregate (Shkapsky et al. 2015).

$$\begin{aligned} &\texttt{cntfriends}(\texttt{Y}, \texttt{mcount<X>}) \texttt{ <- } \texttt{friend}(\texttt{X},\texttt{Y}), \texttt{attend}(\texttt{X}). \\ &\texttt{attend}(\texttt{X}) \texttt{ <- } \texttt{organizer}(\texttt{X}). \\ &\texttt{attend}(\texttt{Y}) \texttt{ <- } \texttt{cntfriends}(\texttt{Y},\texttt{N}), \texttt{N} \geq 3. \end{aligned} \tag{7}$$

The program in (8) shows a lock-free evaluation plan. In each iteration, the j-th worker scans through the j-th partition of `friend`, and checks if there is a specific `X` in `attend`. This evaluation plan requires no lock on `attend` and `cntfriends` but

requires maintaining an index on `attend` during program evaluation.

$$\begin{aligned}
&\texttt{cntfriends}^{\texttt{ff}}(\texttt{Y}, \texttt{mcount<X>}) \texttt{ <- } \texttt{friend}^{\texttt{ff}}(\texttt{X}, \texttt{Y}), h(\texttt{Y}) = j, \texttt{attend}^{\texttt{b}}(\texttt{X}).\\
&\texttt{attend}^{\texttt{f}}(\texttt{X}) \texttt{ <- } \texttt{organizer}^{\texttt{f}}(\texttt{X}), h(\texttt{X}) = j.\\
&\texttt{attend}^{\texttt{f}}(\texttt{Y}) \texttt{ <- } \texttt{cntfriends}^{\texttt{ff}}(\texttt{Y}, \texttt{N}), h(\texttt{Y}) = j, \texttt{N} \geq 3.
\end{aligned} \tag{8}$$

3 Parallel Evaluation of AND/OR Trees

In $\mathcal{DeALS}$, a program is represented as an AND/OR tree (Arni et al. 2003). An OR node represents a predicate and an AND node represents the head of a rule. The root is an OR node. The children of an OR node and an AND node are AND nodes and OR nodes, respectively. There are specialized OR nodes, such as the *R-node* that reads from a base relation or a derived relation and the *W-node* that writes to a derived relation. We use a pipelined evaluation which only materializes relations when it is necessary — if 1) it is the root; or 2) it is an aggregation; or 3) the optimizer determines that the cost of computing the tuples when they are needed is much higher than the cost of materializing the relation in memory. [6]

A program is transformed into an AND/OR tree such that the root represents the query and the children of each AND node follow the same order as the corresponding predicates in the corresponding rule. The children of an AND node are evaluated in the same order as the user provides in the program. The AND/OR tree is then adorned for bottom-up evaluation. An OR node is an *entry node* if 1) it is a leaf, and 2) it is the first R-node among its siblings, and 3) each of its ancestor OR node does not have a left-sibling (i.e., a sibling that appears before the current node) that has a W-node descendant.

Example 5
A non-linear formulation of the transitive closure program is shown as follows.

$$\begin{aligned}
&\texttt{tc}(\texttt{X}, \texttt{Y}) \texttt{ <- } \texttt{arc}(\texttt{X}, \texttt{Y}).\\
&\texttt{tc}(\texttt{X}, \texttt{Y}) \texttt{ <- } \texttt{tc}(\texttt{X}, \texttt{Z}), \texttt{tc}(\texttt{Z}, \texttt{Y}).
\end{aligned} \tag{9}$$

The corresponding AND/OR tree is shown in Fig. 1. The text description inside a node indicates the type and ID of the node, e.g., "OR-1" indicates the root is an OR node with ID 1. OR-4, OR-5 and OR-6 are R-nodes and OR-1 is a W-node. OR-4 and OR-5 are entry nodes in this program.

In $\mathcal{DeALS}$, the j-th worker always evaluates an entry node by instantiating variables with constants from the j-th partition of the corresponding relation. For the remaining R-nodes, each worker has full access to all partitions of the corresponding relations. This strategy ensures that the evaluation is divided into n disjoint parts; otherwise if each worker evaluates an entry node by instantiating variables with constants from all partitions of the corresponding relation, redundant work will be performed.

[6] Currently this is done manually where a user can force the materialization of a relation by an annotation in the program.

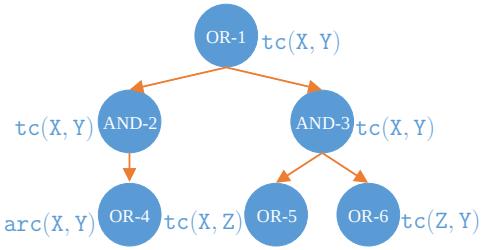

Fig. 1. AND/OR tree of `tc` expressed by the program in (9).

Determining the Discriminating Sets. Now we describe the *read/write* analysis on an adorned AND/OR tree that determines the type of lock needed for each derived relation. The analysis initializes an empty stack, and performs a depth-first traversal on the AND/OR tree. For each node encountered during the traversal:

(1) if it is a W-node, add it to the stack when entering the node, and remove it from the stack when leaving the node;
(2) if it is an entry node, set it as the current entry node; if it reads from a derived relation, for each W-node in the stack, determine if the j-th worker only writes to the j-th partition of $R(p_w)$ by checking if $p_e[\overline{X_j}] = p_w[\overline{X_k}]$[7], where p_e and p_w are the predicates associated with the entry node and W-node, respectively, and $\overline{X_j}$ and $\overline{X_k}$ are the corresponding discriminating sets;
(3) if it is an R-node that reads from a derived relation, determine if the j-th worker only reads from the j-th partition of $R(p_r)$ by checking if $\overline{X_k} \subseteq \overline{B}$ and $p_e[\overline{X_j}] = p_r[\overline{X_k}]$, where p_e and p_r are the predicates associated with the entry node and R-node, respectively, $\overline{X_j}$ and $\overline{X_k}$ are the corresponding discriminating sets, and $\overline{B}$ is the set of positions for bound arguments in the R-node.

In Case (2), the W-nodes in the stack are processed from the last element to the first element until a W-node that is evaluated in the *materialized mode* is encountered. A node evaluated in the materialized mode represents the boundary of a stratum — the evaluation will not move on to any other nodes above it until the evaluation on this node and the nodes below it is finished.

We use the following *discriminating set equations* (DSE) obtained through a read/write analysis to find the best possible discriminating sets for a given program. A DSE consists of three types of equations:

(1) $p_e[\overline{X_j}] = p_w[\overline{X_k}]$ for each entry node in Case (2) of the read/write analysis;
(2) $\overline{X_k} \subseteq \overline{B}$, $p_e[\overline{X_j}] = p_r[\overline{X_k}]$ for each R-node in Case (3) of the read/write analysis;
(3) $\emptyset \subsetneq \overline{X} \subseteq \{1, \cdots, arity(R)\}$ for each $\overline{X}$ appearing in the above two types of equations, where $arity(R)$ is the arity of the relation R associated with $\overline{X}$.

The target is to find an optimal solution to the DSE which is an assignment to the variables that satisfies all the equations of Type (3) and maximizes the number of

[7] The tuple denoted by $p[\overline{X}]$ is treated as a multiset of arguments when involved in equality checking.

satisfied equations of Type (1) and Type (2). The optimal solution that minimizes $\sum_i |\overline{X_i}|$ is selected if there are multiple optimal solutions.

DSE is very similar to the idea of *generalized pivoting* where a system of equations is obtained from the rules and an exact solution is required (Seib and Lausen 1991). But it is different from generalized pivoting in two aspects: 1) DSE is obtained through the read/write analysis on the AND/OR tree since the pipelined evaluation on the AND/OR tree might evaluate multiple rules at the same time where the equations obtained from each single rule cannot capture all the constraints; 2) an exact solution is not required since we want to obtain the best possible evaluation plan even when the program cannot be evaluated without any communication under the message passing model. A DSE can be re-expressed as a system of linear equations with coefficients in $\{-1, 0, 1\}$ (cf. Appendix A). Finding a solution that satisfies the maximum number of equations is $\mathcal{NP}$-hard since there is a reduction from Exact 3-sets Cover to this problem (similarly to Theorem 1 in (Amaldi and Kann 1995)). It is always easy to find an optimal solution for a non-recursive program. For a recursive program, $\mathcal{DeALS}$ finds an optimal solution by enumerating all possible assignments. This is feasible when the number of variables is not large (below 20). Most recursive programs studied in the literature, including transitive closure, bill of materials and same generation, belong to this case. Heuristic search methods for constraint satisfaction problems can be used if there are too many variables.

Finally, the discriminating sets are determined as follows.

(1) Obtain the DSE by performing a read/write analysis on the AND/OR tree.
(2) Find an optimal solution to the DSE.
(3) Determine the type of lock for each derived relation by a read/write analysis on the AND/OR tree with the selected discriminating sets.

Example 6
The DSE for the AND/OR tree in Fig. 1 is shown below.

$$\begin{aligned}
&\texttt{arc}(\texttt{X},\texttt{Y})[\overline{X_1}] = \texttt{tc}(\texttt{X},\texttt{Y})[\overline{X_2}] \\
&\texttt{tc}(\texttt{X},\texttt{Z})[\overline{X_2}] = \texttt{tc}(\texttt{X},\texttt{Y})[\overline{X_2}] \\
&\overline{X_2} \subseteq \{1\} \\
&\texttt{tc}(\texttt{X},\texttt{Z})[\overline{X_2}] = \texttt{tc}(\texttt{Z},\texttt{Y})[\overline{X_2}] \\
&\emptyset \subsetneq \overline{X_1} \subseteq \{1,2\} \\
&\emptyset \subsetneq \overline{X_2} \subseteq \{1,2\}
\end{aligned} \tag{10}$$

The optimal solution is $\overline{X_1} = \overline{X_2} = \{1\}$ that only violates the fourth equation. The result of a read/write analysis determines that a rw-lock is needed for `tc` in the evaluation of the non-linear recursive rule.

We assume the program does not contain aggregates and arithmetic expressions in the above procedure. If the program does contain these constructs, the same procedure is applicable if we ignore all the arguments which are either aggregates or arithmetic expressions. The only exception is that the evaluation of a W-node always requires locks if it contains an aggregate with no *group by* arguments.

4 $\mathcal{DeALS}$

$\mathcal{DeALS}$ is a Datalog system under development at UCLA. It has a Java-based compiler and a sequential Java-based interpreter that allows users to develop and debug their applications. The new parallel evaluation module targeted for shared-memory multicore machines is implemented as a separate module in the system — the compiler compiles a program into an AND/OR tree and then the parallel module determines the parallel evaluation plan using the technique presented in this paper ($\sim$ 700 lines Java) and generates a corresponding `C++` program ($\sim$ 4000 lines Java). We implemented the database objects (index and storage), base classes for each kind of node in the AND/OR tree and common functions in about 6000 lines `C++`. The generated program contains the definition of tuples and relations, and the actual implementation of the AND/OR tree based on the base classes. It is compiled into the final executable by invoking the Visual C++ Compiler that comes with Visual Studio 2013 (v120) on a Windows machine or GCC 4.9.2 on a Linux machine. We plan to take advantage of modern compiler technologies like LLVM (Lattner 2008) to reduce the compilation latencies from several seconds to around 10ms in the next version of $\mathcal{DeALS}$.

5 Experimental Results

In this section, we report some experimental results on both non-recursive and recursive programs. Each system is configured to use all the available CPUs and memory on the test machine. All execution times are calculated by taking the average of five runs of the same experiment.

Exp-I: Non-recursive programs — TPC-H benchmark. The benchmark contains 22 (non-recursive) SQL queries over a database of eight tables. The data types involved in the queries are integer, decimal, string and date. We implemented all the 22 queries following the query plans described in (Dees and Sanders 2013).[8] We tested the performance of $\mathcal{DeALS}$ on a test machine with four AMD Opteron 6376 CPUs (16 cores per CPU) and 256GB memory (configured into eight NUMA regions). The operating system is Ubuntu Linux 12.04 LTS. $\mathcal{DeALS}$ is able to correctly evaluate all the 22 queries on databases of size 1GB, 10GB and 100GB on the test machine, with a speedup of 11.49, 23.05, 27.32, respectively (evaluation time for all the 22 queries using one processor divides the time using 64 processors). Fig. 2 shows the total query evaluation time for all the 22 queries. The last point 667.974s shows the predicted time on a database of size 1TB if the evaluation time scales linearly w.r.t. the size of the database. We also show the current single machine world record for the benchmark on databases of size 100GB and 1TB. VectorWise 2.0.1 evaluates all the queries in 22.8s on a database of size 100GB,[9] while it takes

[8] COUNT(DISTINCT) is replaced with COUNT in `query16`. ORDER BY and LIMIT are ignored in our program. The evaluation time will not change significantly if we add these constructs since most queries return very few results except `query3` and `query10`.

[9] TPC-H Result on Dell PowerEdge R720, `http://www.tpc.org/3282`.

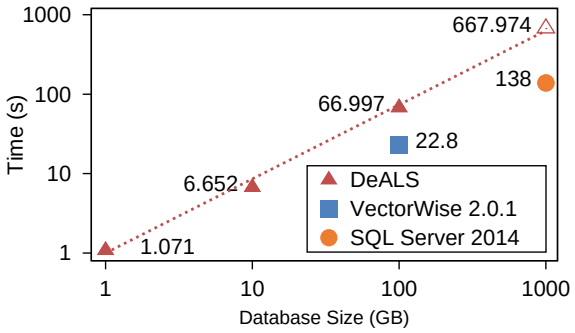

Fig. 2. Total query evaluation time for 22 queries in the TPC-H benchmark.

Microsoft SQL Server 2014 Enterprise Edition 138s on a database of size 1TB.[10] $\mathcal{DeALS}$ is within five times slower comparing with these two highly optimized commercial systems (both system settings are also optimized for the benchmark). Note that the SPECint_rate2006 of our test machine is only 527 (the larger the more powerful), while the values are 695 and 2400 for the other two machines. So the real performance gap between $\mathcal{DeALS}$ and the state of the art RDBMS is even closer on the TPC-H benchmark.

Exp-II: Non-recursive programs — graph queries. We compare $\mathcal{DeALS}$ with LogicBlox (Aref et al. 2015) and SQL Server on evaluating three graph queries — `3clique` (find the number of three cliques in the graph), `4clique` (find the number of four cliques) and `4cycle` (find the number of cycles of length four) on the Pokec social network graph.[11] The database contains only one table `arc(X, Y)` where a primary index on `(X, Y)` and a secondary index on `Y` are built. The queries are listed in Appendix B. We used a Microsoft Azure D4 Linux virtual machine instance running Ubuntu Linux 14.04 LTS with an Intel Xeon E5-2660 CPU (8 hyperthreads), 28GB memory and 400GB SSD to run $\mathcal{DeALS}$ and LogicBlox 4.1.9. The experiments on SQL Server were run on a Microsoft Azure D4 SQL Server instance (same hardware as a D4 Linux virtual machine) running Microsoft SQL Server 2014 Enterprise Edition. The speedups of $\mathcal{DeALS}$ on these three queries are 6.48, 5.96, 6.20, respectively (evaluation time using one processor divides the time using eight processors). Fig. 3 compares the query evaluation time of these systems. $\mathcal{DeALS}$ is faster than SQL Server on all three queries, while LogicBlox is faster than $\mathcal{DeALS}$ on two queries which is mainly the result of a worst-case optimal join algorithm called Leapfrog Triejoin (Veldhuizen 2014) used by LogicBlox. It is interesting to note that LogicBlox demonstrates much better performance on the same set of queries comparing with graph databases such as Virtuoso and Neo4j as reported in Table 6 of (Nguyen et al. 2015) on a machine with similar hardware. Thus, $\mathcal{DeALS}$ achieves competitive performance on these graph queries comparing with other existing systems with declarative languages.

[10] TPC-H Result on Cisco UCS C460 M4 Server, `http://www.tpc.org/3311`.
[11] Pokec social network, `http://snap.stanford.edu/data/soc-pokec.html`. We use the graph as a directed graph for `4cycle`, and as a undirected graph for `3clique` and `4clique`.

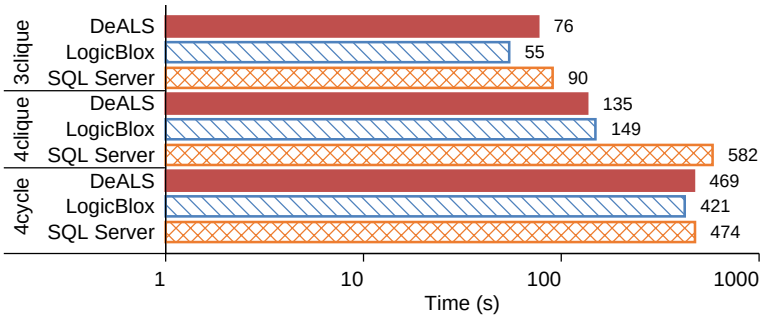

Fig. 3. Query evaluation time of graph queries on the Pokec social network graph.

Exp-III: Recursive programs. Finally, we test the performance of $\mathcal{DeALS}$ on three classical recursive queries — `tc` (transitive closure), `sg` (same generation), `attend` (program in Example 4). The queries and test datasets are listed in Appendix B and Appendix C, respectively. We compare $\mathcal{DeALS}$ with LogicBlox 4.1.9 and DLV (Leone et al. 2006), which are two systems that represent the state of the art in evaluating logic programs in the areas of deductive database and disjunctive logic programming, respectively, on the same machine used in **Exp-I**. Both $\mathcal{DeALS}$ and LogicBlox support all three queries and evaluate them correctly on the test datasets. DLV runs out of memory on our test machine with 256GB memory on the evaluation of `sg` on tree-11. The version of DLV[12] that supports aggregates in recursion is a 32-bit executable which fails on the evaluation of `attend` on both patent and wiki as it does not support more than 4GB memory required by evaluation. DLV takes 11.28s, 13,127s, 9,272s for `tc` on tree-11, grid-150, gnp-10K and 104.9s, 48,038s for `sg` on grid-150, gnp-10K,[13] which is much slower than $\mathcal{DeALS}$ using one processor. Fig. 4 compares the evaluation time of $\mathcal{DeALS}$ and LogicBlox on these recursive queries. Bars for DeALS-1 and DeALS-64 show the evaluation time of $\mathcal{DeALS}$ using one processor and 64 processors, respectively. Bars for LogicBlox show the evaluation time of LogicBlox using 64 processors.[14] $\mathcal{DeALS}$ outperforms or equally performs LogicBlox on these queries and datasets when it uses only one processor, while it always outperforms LogicBlox when it uses 64 processors. $\mathcal{DeALS}$ achieves a greater speedup (the speedup of DeALS-64 over DeALS-1) for `tc` and `attend` than `sg` since no lock is used in `tc` and `attend`, while `sg` suffers from lock contention. It is important to note that the evaluation of `tc` and `attend` still requires synchronization where the coordinator determines the next rule to be evaluated after all workers finish, and the synchronization time increases as the number of processors increases. In the extreme case where the synchronization time

[12] DLV with Recursive Aggregates, downloaded from `http://www.dbai.tuwien.ac.at/proj/dlv/dlvRecAggr/dl-recagg-snapshot-2007-04-14.zip`.

[13] The times are collected from a single-processor version of DLV (downloaded from `http://www.dlvsystem.com/files/dlv.x86-64-linux-elf-static.bin`). Although a parallel version (`http://www.mat.unical.it/ricca/downloads/parallelground10.zip`) is available, it is either much slower than the single-processor version or it fails since it is a 32-bit executable that does not support more than 4GB memory required by evaluation.

[14] In our experiments, LogicBlox 4.1.9 does not utilize all the processors all the time.

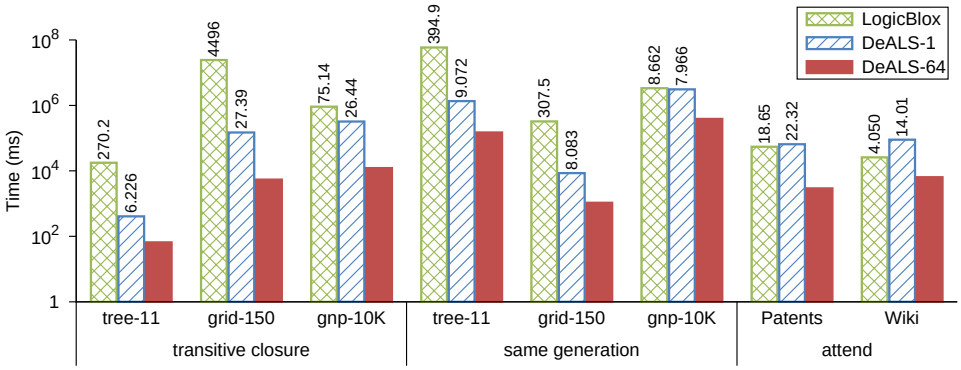

Fig. 4. Query evaluation time of recursive queries. The numbers above the bars for LogicBlox (DeALS-1) show the speedup of DeALS-64 over LogicBlox (DeALS-1).

dominates the evaluation time, $\mathcal{DeALS}$ achieves very limited speedup — e.g., the evaluation of `tc` on `tree-11`.

6 How to Obtain a Lock-free Program

The experimental results in the previous section show that $\mathcal{DeALS}$ is able to achieve a good speedup using multiple processors in the evaluation of a lock-free program. In this section, we present a sufficient condition for a program to have a lock-free evaluation plan and a rewriting from an arbitrary program to a lock-free program. We assume the program does not contain aggregates and arithmetic expressions.

A Sufficient Condition. In the evaluation of the program in (6), the j-th worker only reads from and writes to the j-th partition of `tc`. It finishes the evaluation when there are no new tuples in the j-th partition of `tc`. Thus, the coordinator is not necessary in the evaluation. Such a program is a *coordination-free* program given that every worker has full access to all the base relations. If the program is evaluated in the message passing model, no worker needs to communicate with any other workers during the evaluation. This is a very nice property in the message passing model since communication is expensive in the model. Theorem 5.1 (Ganguly et al. 1992) says that there exists a coordination-free evaluation plan for a linear single rule program if the corresponding dataflow graph contains a cycle. Now we show that an arbitrary linear program has a lock-free evaluation plan if the corresponding dataflow graphs satisfy a similar condition. We say a multi-rule program is a linear program, if the head predicate of a rule appears at most once in the rule body for every rule.

Definition 1

For a recursive predicate `p`, let $\mathtt{p}(X_1, \cdots, X_m)$ be the head of a linear recursive rule r and $\mathtt{p}(Y_1, \cdots, Y_m)$ be the occurrence of `p` in the body of r. The *dataflow graph* of $(r, \mathtt{p})$ is a directed graph $G = (V, E)$, where:

- $V \subseteq \{1, \cdots, m\}$ and $i \in V$ iff $\exists j \in \{1, \cdots, m\}$ such that $Y_i = X_j$.
- There is an edge from i to j iff $Y_i = X_j$.

The following theorem states a sufficient condition for a program to have a lock-free evaluation plan.

Theorem 1
Given a program where each recursive predicate is defined by a set of linear recursive rules. For every recursive predicate, if the dataflow graphs corresponding to all related recursive rules contain the same cycle, then there exists a lock-free parallel evaluation plan.

Proof
The proof is a constructive one that is similar to Appendix C in (Ganguly et al. 1992). For a recursive predicate $\mathtt{p}(X_1, \cdots, X_m)$, assume all related recursive rules contain the same cycle $a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_t \rightarrow a_1$. Let the discriminating set of $\mathtt{p}$ be $\{a_1, a_2, \cdots, a_t\}$. For each recursive rule, use the recursive predicate as the first occurrence in the body of the rule during the adornment. Then we can prove that the evaluation for $\mathtt{p}$ is lock-free. □

Example 7
Consider the following program where $\mathtt{p}$ and $\mathtt{q}$ are the recursive predicates. The recursive rules that define $\mathtt{p}$ are $r11.3$ and $r11.4$. The edge set for the dataflow graph of $(r11.3, \mathtt{p})$ is $\{(2,2),(3,3)\}$. There is an edge from 2 to 2 since $\mathtt{Y}$ is the second argument in $\mathtt{p}(\mathtt{W},\mathtt{Y},\mathtt{Z})$ and $\mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z})$. Similarly for the edge from 3 to 3. The edge set for the dataflow graph of $(r11.4, \mathtt{p})$ is $\{(1,3),(2,2),(3,1)\}$. Both graphs contain a self-loop from 2 to 2. The recursive rule that defines $\mathtt{q}$ is $r11.5$. The edge set for the dataflow graph of $(r11.5, \mathtt{q})$ is $\{(1,1)\}$. It contains a self-loop from 1 to 1. Thus, the program has a lock-free evaluation plan.

$$
\begin{aligned}
&r11.1 : \mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \texttt{<-} \mathtt{b}_1(\mathtt{X},\mathtt{Y},\mathtt{Z}).\\
&r11.2 : \mathtt{q}(\mathtt{X},\mathtt{Y}) \texttt{<-} \mathtt{b}_2(\mathtt{X},\mathtt{Y}).\\
&r11.3 : \mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \texttt{<-} \mathtt{q}(\mathtt{X},\mathtt{W}), \mathtt{p}(\mathtt{W},\mathtt{Y},\mathtt{Z}).\\
&r11.4 : \mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \texttt{<-} \mathtt{b}_3(\mathtt{X}), \mathtt{p}(\mathtt{Z},\mathtt{Y},\mathtt{X}).\\
&r11.5 : \mathtt{q}(\mathtt{X},\mathtt{Z}) \texttt{<-} \mathtt{q}(\mathtt{X},\mathtt{W}), \mathtt{p}(\mathtt{W},\_,\mathtt{Z}).
\end{aligned}
\tag{11}
$$

Let the discriminating sets of $\mathtt{p}$ and $\mathtt{q}$ be $\{2\}$ and $\{1\}$, respectively. The program below represents a lock-free evaluation plan.

$$
\begin{aligned}
&\mathtt{p}^{\mathtt{fff}}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \texttt{<-} \mathtt{b}_1^{\mathtt{fff}}(\mathtt{X},\mathtt{Y},\mathtt{Z}), h(\mathtt{Y}) = j.\\
&\mathtt{q}^{\mathtt{ff}}(\mathtt{X},\mathtt{Y}) \texttt{<-} \mathtt{b}_2^{\mathtt{ff}}(\mathtt{X},\mathtt{Y}), h(\mathtt{X}) = j.\\
&\mathtt{p}^{\mathtt{fff}}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \texttt{<-} \mathtt{p}^{\mathtt{fff}}(\mathtt{W},\mathtt{Y},\mathtt{Z}), h(\mathtt{Y}) = j, \mathtt{q}^{\mathtt{fb}}(\mathtt{X},\mathtt{W}).\\
&\mathtt{p}^{\mathtt{fff}}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \texttt{<-} \mathtt{p}^{\mathtt{fff}}(\mathtt{Z},\mathtt{Y},\mathtt{X}), h(\mathtt{Y}) = j, \mathtt{b}_3^{\mathtt{b}}(\mathtt{X}).\\
&\mathtt{q}^{\mathtt{ff}}(\mathtt{X},\mathtt{Z}) \texttt{<-} \mathtt{q}^{\mathtt{ff}}(\mathtt{X},\mathtt{W}), h(\mathtt{X}) = j, \mathtt{p}^{\mathtt{bff}}(\mathtt{W},\_,\mathtt{Z}).
\end{aligned}
\tag{12}
$$

Rewriting a Program into a Lock-free Program. Given an arbitrary program, we can always rewrite it into a lock-free program. The rewriting replaces an occurrence of a recursive predicate $\mathtt{p}$ with a predicate $\mathtt{p}^{(i)}$ that caches the state of $\mathtt{p}$ at the end of the previous iteration. The corresponding relation is repartitioned

and stored in $\mathtt{p}^{(i)}$. It is then used like a base relation during the evaluation of the rule. The rewriting produces lock-free programs at the cost of extra space and additional relation repartitioning operations (which can be implemented as in-memory shuffles). The rewriting consists of two steps: 1) from a non-linear program to a linear program; and 2) from a linear program to a lock-free program.

Step 1. For each recursive predicate p and each non-linear recursive rule r that can be canonically represented as

$$\mathtt{p}(X_1, \cdots, X_m) \texttt{ <- } \mathtt{p}(Y_{1,1}, \cdots, Y_{1,m}), \cdots, \mathtt{p}(Y_{t,1}, \cdots, Y_{t,m}), b_1, \cdots, b_s, \tag{13}$$

where each $\mathtt{p}(Y_{i,1}, \cdots, Y_{i,m})$ represents an occurrence of p in the rule body, and each b_j is a non-p predicate. We replace r with the following rules:

$$\begin{aligned} \mathtt{p}(X_1, \cdots, X_m) \texttt{ <- } & \mathtt{p}(Y_{1,1}, \cdots, Y_{1,m}), \mathtt{p}^{(1)}(Y_{t,1}, \cdots, Y_{t,m}), \cdots, \\ & \mathtt{p}^{(t-1)}(Y_{t,1}, \cdots, Y_{t,m}), b_1, \cdots, b_s. \\ \mathtt{p}^{(1)}(X_1, \cdots, X_m) \texttt{ <- } & \mathtt{p}(X_1, \cdots, X_m). \\ & \vdots \\ \mathtt{p}^{(t-1)}(X_1, \cdots, X_m) \texttt{ <- } & \mathtt{p}(X_1, \cdots, X_m). \end{aligned} \tag{14}$$

The program becomes a linear program after rewriting. Now we test if the new program satisfies the condition in Theorem 1. If not, we continue to Step 2.

Example 8
The non-linear formulation of the transitive closure program in Example 5 can be rewritten into the following program by replacing the the second occurrence of tc with $\mathtt{tc}^{(1)}$ in the non-linear rule.

$$\begin{aligned} & r15.1 : \mathtt{tc}(\mathtt{X}, \mathtt{Y}) \texttt{ <- } \mathtt{arc}(\mathtt{X}, \mathtt{Y}). \\ & r15.2 : \mathtt{tc}(\mathtt{X}, \mathtt{Y}) \texttt{ <- } \mathtt{tc}(\mathtt{X}, \mathtt{Z}), \mathtt{tc}^{(1)}(\mathtt{Z}, \mathtt{Y}). \\ & r15.3 : \mathtt{tc}^{(1)}(\mathtt{X}, \mathtt{Y}) \texttt{ <- } \mathtt{tc}(\mathtt{X}, \mathtt{Y}). \end{aligned} \tag{15}$$

Now the dataflow graph of $(r15.2, \mathtt{tc})$ contains a self-loop from 1 to 1. Thus, the program has a lock-free evaluation plan.

Step 2. For a recursive predicate, if the dataflow graphs corresponding to all related recursive rules contain the same cycle, we keep all the rules about this predicate. For each of the remaining recursive predicate p and each linear recursive rule r that can be canonically represented as

$$\mathtt{p}(X_1, \cdots, X_m) \texttt{ <- } \mathtt{p}(Y_1, \cdots, Y_m), b_1, \cdots, b_s, \tag{16}$$

where $\mathtt{p}(Y_1, \cdots, Y_m)$ represents the occurrence of p in the rule body, and each b_j is a non-p predicate. We replace r with the following two rules:

$$\begin{aligned} & \mathtt{p}(X_1, \cdots, X_m) \texttt{ <- } \mathtt{p}^{(1)}(Y_1, \cdots, Y_m), b_1, \cdots, b_s. \\ & \mathtt{p}^{(1)}(X_1, \cdots, X_m) \texttt{ <- } \mathtt{p}(X_1, \cdots, X_m). \end{aligned} \tag{17}$$

The new program is still a recursive program. But in each iteration, no workers will write to a relation while reading from the same relation. Let the discriminating set

of p be {1}. Use a predicate that contains variable X_1 as the first predicate in the body of the rule during the adornment. The adorned program is lock-free.

Example 9
The sg program below which finds all the (X, Y) pairs that are of the same generation is not a lock-free program.

$$\begin{aligned} &r18.1 : \texttt{sg}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{anc}(\texttt{A},\texttt{X}), \texttt{anc}(\texttt{A},\texttt{Y}), \texttt{X} \neq \texttt{Y}. \\ &r18.2 : \texttt{sg}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{anc}(\texttt{A},\texttt{X}), \texttt{sg}(\texttt{A},\texttt{B}), \texttt{anc}(\texttt{B},\texttt{Y}). \end{aligned} \tag{18}$$

The program in (19) shows a lock-free program obtained by replacing the occurrence of sg in $r18.2$ with $\texttt{sg}^{(1)}$.

$$\begin{aligned} &\texttt{sg}^{\texttt{ff}}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{anc}^{\texttt{ff}}(\texttt{A},\texttt{X}), h(\texttt{X}) = j, \texttt{anc}^{\texttt{bf}}(\texttt{A},\texttt{Y}), \texttt{X} \neq \texttt{Y}. \\ &\texttt{sg}^{\texttt{ff}}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{anc}^{\texttt{ff}}(\texttt{A},\texttt{X}), h(\texttt{X}) = j, \texttt{sg}^{(1)\texttt{bf}}(\texttt{A},\texttt{B}), \texttt{anc}^{\texttt{bf}}(\texttt{B},\texttt{Y}). \\ &\texttt{sg}^{(1)\texttt{ff}}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{sg}^{\texttt{ff}}(\texttt{X},\texttt{Y}), h(\texttt{X}) = j. \end{aligned} \tag{19}$$

7 Discussion

Although we can rewrite many programs into lock-free programs, a lock-free plan may not always be the best plan in terms of evaluation time. We show this with the following two examples.

Example 10
Consider the following program, where X is the primary key in q and h(Z) only takes four values.

$$\texttt{p}(\texttt{Z}, \texttt{count<Y>}) \texttt{ <- } \texttt{q}(\texttt{X},\texttt{Y},\texttt{Z}). \tag{20}$$

The evaluation is lock-free if the discriminating sets of p and q are {1} and {3}, respectively. However, if the number of available processors is greater than four, then only four processors will be busy during evaluation since there are four non-empty partitions. A better plan is to choose the discriminating set of q as {1}, i.e., partitioning q by its X argument. During evaluation, each worker performs the aggregation locally and the coordinator merges these partial values to produce the final aggregation result after all the workers finish.

Example 11
The program in (21) shows an alternative plan for the attend program in (7) that requires locks. In each iteration, the j-th worker scans the j-th partition of attend, and checks for a tuple (X, Y) in friend. If so, it acquires the x-lock on the h(Y)-th partition of cntfriends, increases the corresponding counter by 1, and releases the x-lock. This plan requires x-locks on cntfriends and an index on friend that can be built before program evaluation.

$$\begin{aligned} &\texttt{cntfriends}^{\texttt{ff}}(\texttt{Y}, \texttt{mcount<X>}) \texttt{ <- } \texttt{attend}^{\texttt{f}}(\texttt{X}), h(\texttt{X}) = j, \texttt{friend}^{\texttt{bf}}(\texttt{X},\texttt{Y}). \\ &\texttt{attend}^{\texttt{f}}(\texttt{X}) \texttt{ <- } \texttt{organizer}^{\texttt{f}}(\texttt{X}), h(\texttt{X}) = j. \\ &\texttt{attend}^{\texttt{f}}(\texttt{Y}) \texttt{ <- } \texttt{cntfriends}^{\texttt{ff}}(\texttt{Y},\texttt{N}), h(\texttt{Y}) = j, \texttt{N} \geq 3. \end{aligned} \tag{21}$$

Assume the index lookup operation takes the same constant time C in both programs, and the total evaluation time is dominated by the index lookup time. The number of index lookup operations in the program in (21) is bounded by $3|\texttt{friend}|$, while the number of index lookup operations in the program in (8) is $|\texttt{friend}| \times T$ where T is the number of iterations in the semi-naive evaluation. If T equals 12 and the number of processors n equals 2, the program in (21) is a better plan than the program in (8) since $3|\texttt{friend}| \times C/1 < |\texttt{friend}| \times T \times C/2$ even if the program in (8) achieves linear speedup (2x) while the program in (21) achieves no speedup (1x) due to heavy lock contention.

$\mathcal{DeALS}$ leaves the choice of how to evaluate the program to the educated users. A user can force the system to partition a relation by a certain columns by specifying a discriminating set for a predicate in the program. $\mathcal{DeALS}$ tries to find the best parallel evaluation plan for that program where the predicates in every rule are evaluated in the same order as they are provided in the program.

8 Related Work

The parallel evaluation strategy proposed in this paper uses a simple hash-based data partitioning strategy. Various data partitioning strategies for parallel bottom-up evaluation have been studied in (Wolfson and Silberschatz 1988; Wolfson 1988; Cohen and Wolfson 1989; Seib and Lausen 1991; Ganguly et al. 1992; Zhang et al. 1995; Ganguly et al. 1995). These studies assume a message passing model and focus on minimizing the amount of message exchange, whereas our study considers a shared-memory model where no message exchange is needed during the evaluation; we demonstrate the effectiveness of our technique with a real Datalog system implementation while previous studies focus on the theoretical aspect. The settings of (Raschid and Su 1986; Hulin 1989; Bell et al. 1991) are more similar to ours, where strategies for top-down evaluation are proposed. These strategies are complementary to ours since we focus on the bottom-up evaluation. Yet another related work is our ongoing research which aims to provide a distributed evaluation engine for $\mathcal{DeALS}$. However, the study presented in this paper focus on the case where all the base relations and derived relations fit into the memory of a single machine.

9 Conclusion

In this paper, we presented the technique used in $\mathcal{DeALS}$ for parallel bottom-up evaluation on shared-memory multicore machines. The technique is simple and applicable to a wide range of non-recursive and recursive programs. A somewhat surprising result is that $\mathcal{DeALS}$ is able to achieve competitive performance on non-recursive programs compared with RDBMSs and superior performance on recursive programs compared with other existing systems, by adding a parallel evaluation module based on this technique. However, there is still a clear performance gap between $\mathcal{DeALS}$ and the hand written optimal programs, such as the SSC12 algorithm for transitive closure (Yang and Zaniolo 2014). We are working on reducing

the gap by further optimizing the performance of the generated program. Another ongoing work is to provide a parallel evaluation module targeted for distributed environment that is able to solve problems when the dataset does not fit into the memory of a single machine. We are also working on optimizing the support for and the performance of new applications requiring data mining and graph analysis.

References

Amaldi, E. and Kann, V. 1995. The complexity and approximability of finding maximum feasible subsystems of linear relations. *Theoretical Computer Science 147,* 1, 181–210.

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

Arni, F., Ong, K., Tsur, S., Wang, H., and Zaniolo, C. 2003. The deductive database system LDL++. *TPLP 3,* 1, 61–94.

Bell, D. A., Shao, J., and Hull, M. E. C. 1991. A pipelined strategy for processing recursive queries in parallel. *Data & Knowledge Engineering 6,* 5, 367–391.

Cohen, S. and Wolfson, O. 1989. Why a single parallelization strategy is not enough in knowledge bases. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, 200–216.

Dees, J. and Sanders, P. 2013. Efficient many-core query execution in main memory column-stores. In *Proceedings of the 29th International Conference on Data Engineering (ICDE 2013)*. IEEE, 350–361.

Ganguly, S., Silberschatz, A., and Tsur, S. 1992. Parallel bottom-up processing of datalog queries. *The Journal of Logic Programming 14,* 1, 101–126.

Ganguly, S., Silberschatz, A., and Tsur, S. 1995. Mapping datalog program execution to networks of processors. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 7,* 3, 351–361.

Hulin, G. 1989. Parallel processing of recursive queries in distributed architectures. In *Proceedings of the 15th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 87–96.

Lattner, C. 2008. Llvm and clang: Next generation compiler technology. In *The BSD Conference*. 1–2.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2006. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL) 7,* 3, 499–562.

Nguyen, D., Aref, M., Bravenboer, M., Kollias, G., Ngo, H. Q., Ré, C., and Rudra, A. 2015. Join processing for graph patterns: An old dog with new tricks. *arXiv preprint arXiv:1503.04169*.

Perri, S., Ricca, F., and Sirianni, M. 2013. Parallel instantiation of asp programs: techniques and experiments. *Theory and Practice of Logic Programming 13,* 02, 253–278.

Raschid, L. and Su, S. Y. W. 1986. A parallel processing strategy for evaluating recursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 412–419.

Seib, J. and Lausen, G. 1991. Parallelizing datalog programs by generalized pivoting. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, 241–251.

Shkapsky, A., Yang, M., and Zaniolo, C. 2015. Optimizing recursive queries with monotonic aggregates in deals. In *Proceedings of the 31st International Conference on Data Engineering (ICDE 2015)*. IEEE.

Ullman, J. D. 1985. Implementation of logical query languages for databases. *ACM Transactions on Database Systems (TODS) 10,* 3, 289–321.

Veldhuizen, T. L. 2014. Triejoin: A simple, worst-case optimal join algorithm. In *Proceedings of the 17th International Conference on Database Theory (ICDT 2014)*. 96–106.

Wolfson, O. 1988. Sharing the load of logic-program evaluation. In *Proceedings of the First International Symposium on Databases in Parallel and Distributed Systems*. IEEE Computer Society Press, 46–55.

Wolfson, O. and Silberschatz, A. 1988. Distributed processing of logic programs. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. ACM, 329–336.

Yang, M. and Zaniolo, C. 2014. Main memory evaluation of recursive queries on multicore machines. In *2014 IEEE International Conference on Big Data (IEEE BigData 2014)*. IEEE, 251–260.

Zhang, W., Wang, K., and Chau, S.-C. 1995. Data partition and parallel evaluation of datalog programs. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 7,* 1, 163–176.

Appendix A Rewriting Discriminating Sets Equations

We describe how to re-express a DSE as a system of linear equations with coefficients in $\{-1, 0, 1\}$ using Equation (10) as an example. Equation (10) can be re-expressed as follows if we replace $\overline{X_1}$ with $(x_{1,1}, x_{1,2})$ and $\overline{X_2}$ with $(x_{2,1}, x_{2,2})$, where $x_{i,j} = 1$ iff $\overline{X_i}$ contains j.

$$\begin{aligned}
&\{x_{1,1}\mathtt{X}, x_{1,2}\mathtt{Y}\} = \{x_{2,1}\mathtt{X}, x_{2,2}\mathtt{Y}\}\\
&\{x_{2,1}\mathtt{X}, x_{2,2}\mathtt{Z}\} = \{x_{2,1}\mathtt{X}, x_{2,2}\mathtt{Y}\}\\
&x_{2,2} = 0\\
&\{x_{2,1}\mathtt{X}, x_{2,2}\mathtt{Z}\} = \{x_{2,1}\mathtt{Z}, x_{2,2}\mathtt{Y}\}\\
&0 \leq x_{1,1} \leq 1, 0 \leq x_{1,2} \leq 1, x_{1,1} + x_{1,2} > 0\\
&0 \leq x_{2,1} \leq 1, 0 \leq x_{2,2} \leq 1, x_{2,1} + x_{2,2} > 0
\end{aligned} \tag{A1}$$

Equation (A1) is equivalent to the following Equation (A2), which is a system of linear equations with coefficients in $\{-1, 0, 1\}$ if we remove the equations that are trivial.

$$\begin{aligned}
&x_{1,1} - x_{2,1} = 0, x_{1,2} - x_{2,2} = 0\\
&x_{2,1} - x_{2,1} = 0, -x_{2,2} = 0, x_{2,2} = 0\\
&x_{2,2} = 0\\
&x_{2,1} = 0, -x_{2,2} = 0, -x_{2,1} + x_{2,2} = 0\\
&x_{i,j} = 0, i \in \{1, 2\}, j \in \{1, 2\}\\
&x_{i,j} = 1, i \in \{1, 2\}, j \in \{1, 2\}\\
&x_{i,1} + x_{i,2} = 1, i \in \{1, 2\}\\
&x_{i,1} + x_{i,2} = 2, i \in \{1, 2\}
\end{aligned} \tag{A2}$$

Appendix B $\mathcal{DeAL}$ Queries used in the Experiments

```
count_3clique(count<_>) <- arc(X, Y), X < Y, arc(Y, Z), Y < Z, arc(Z, X).

count_4cycle(count<_>) <- arc(X, Y), X < Y, arc(Y, Z), Y < Z, arc(Z, W), Z < W, arc(W, X).

count_4clique(count<_>) <- arc(X, Y), X < Y, arc(Y, Z), Y < Z, arc(Z, X), arc(Z, W),
                           Z < W, arc(X, W), arc(Y, W).

tc(X, Y) <- arc(X, Y).
tc(X, Y) <- tc(X, Z), arc(Z, Y).

sg(X, Y) <- anc(P, X), anc(P, Y), X ~= Y.
sg(X, Y) <- anc(A, X), sg(A, B), anc(B, Y).

cntfriends(Y, mcount<X>) <- friend(Y, X), attend(X).
attend(X) <- organizer(X).
attend(Y) <- cntfriends(Y, N), N >= 3.
```

Appendix C Description of Datasets

Synthetic Graphs.

1) tree-11 is a randomly generated tree of depth 11 where the out degree of a non-leaf vertex is a random number between 2 to 6.
2) grid-150 is a 151×151 square grid.
3) gnp-10K is a $G(n, p)$ graph (Erdős-Rényi model) of 10,000 vertices generated by connecting vertices randomly such that the average out-degree of a vertex is 10.

The experiments on `tc` evaluation use each of these synthetic graphs as `arc`, where the experiments on `sg` evaluation use each of these graphs as `anc`.

Real World Graphs.

1) patent is the US patent citation graph[15]. Each vertex represents a patent, and each edge represents a citation between two patents. It has 3,774,769 vertices and 16,518,948 edges.
2) wiki is the Wikipedia knowledge graph. Each vertex represents an entity in the Wikipedia. If an entity appears in the infobox of another entity, there is a directed edge between the two corresponding vertices. It has 3,165,181 vertices and 23,190,820 edges.

[15] Patent citation network, `https://snap.stanford.edu/data/cit-Patents.html`.

The experiments on `attend` evaluation use each of these graphs as `friend`, while `organizer` contains all the vertices in the graph whose in-degree are zero.