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

MOHAN YANG, ALEXANDER SHKAPSKY, CARLO ZANIOLO
University of California, Los Angeles

submitted 29 April 2015; accepted 5 June 2015

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 (**?**; **?**; **?**; **?**) and strategies to minimize the amount of communication required (**?**; **?**; **?**). 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 (**?**; **?**). 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 (**?**), 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 **??**. We describe how $\mathcal{DeALS}$ tries to find a lock-free evaluation plan in Section **??**. An overview of $\mathcal{DeALS}$ is presented in Section **??**. We report experimental results in Section **??**. Related work is discussed in Section **??**. The paper concludes in Section **??**.

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`.

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

The bottom-up evaluation of this program works as follows. The exit rule r**??**.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 r**??**.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] r**??**.2 is repeatedly evaluated until `tc` does not change between two successive evaluations of r**??**.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 rules which support its goal as described in Section **??**). 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 (**?**).

[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.

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 $r\texttt{??}.1$ 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

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

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, such as concatenation ($||$) where $h(x_1, \cdots, x_t)$ becomes $g(x_1)||\cdots||g(x_t)$ (**?**). 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 values 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:

$$r_i^j : p_{i,0} \texttt{<-}\ p_{i,1}, \cdots, p_{i,m_i}, h(p_{i,t_i}[\overline{X_i}]) = j, \tag{3}$$

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

$$\begin{aligned} &r\textbf{??}.1 : \texttt{tc(X, Y)} \texttt{<-}\ \texttt{arc(X, Y)}, h(\texttt{X}) = j. \\ &r\textbf{??}.2 : \texttt{tc(X, Y)} \texttt{<-}\ \texttt{tc(X, Z)}, \texttt{arc(Z, Y)}, h(\texttt{X}) = j. \end{aligned} \tag{4}$$

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

Example 2
The program in (**??**) shows an adorned version of the program in Example **??**. The predicates in $r??.2$ are reordered to reflect the order in which they are evaluated. In the evaluation of $r??.2$, 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.

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

Example 3
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 (**?**).

$$\begin{aligned} &\texttt{cntfriends}(\texttt{Y}, \texttt{mcount<X>}) \texttt{ <- } \texttt{friend}(\texttt{X},\texttt{Y}), \texttt{attend}(\texttt{X}). \\ &\texttt{attend}(\texttt{X}) \texttt{ <- } \texttt{organizer}(\texttt{X}). \\ &\texttt{attend}(\texttt{Y}) \texttt{ <- } \texttt{cntfriends}(\texttt{Y},\texttt{N}), \texttt{N} \geq 3. \end{aligned} \tag{6}$$

The program in (**??**) shows a lock-free evaluation plan. Each relation is partitioned by its first column except `friend` is partitioned by its second column. 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` which can be efficiently supported by maintaining an index on `attend` during program evaluation. Thus, the j-th worker only writes to the j-th partition of `cntfriends` and `attend`, and no lock is needed.

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

3 Parallel Evaluation of AND/OR Trees

In DeALS, a program is represented as an AND/OR tree (**?**). 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 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 an R-node descendant or a W-node descendant.

Example 4
A non-linear formulation of the transitive closure program is shown as follows.

$$\begin{aligned} &\texttt{tc(X, Y) <- arc(X, Y).} \\ &\texttt{tc(X, Y) <- tc(X, Z), tc(Z, Y).} \end{aligned} \tag{8}$$

The corresponding AND/OR tree is shown in Fig. **??**. 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.

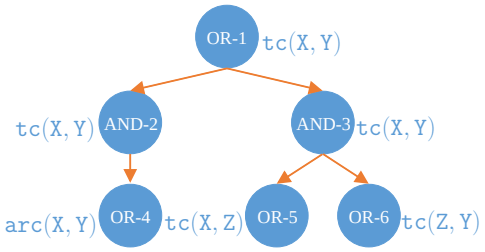

Fig. 1. AND/OR tree of `tc` expressed by the program in (**??**).

In $\mathcal{DeALS}$, the j-th worker 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.

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 performs a depth-first traversal on the AND/OR tree that simulates the actual evaluation to check each read or write operation performed by the j-th worker. For each node encountered during the traversal:

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

(1) if it is an entry node, set it as the current entry node; if it reads from a derived relation, for each W-node that is an ancestor of the current node,[7] determine if the j-th worker only writes to the j-th partition of $R(p_w)$ by checking if $p_e[\overline{X_j}] = p_w[\overline{X_k}]$[8], 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;
(2) 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 current 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.

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 satisfied equations of Type (1) and Type (2). DeALS finds an optimal solution by enumerating all possible assignments, which is feasible for most recursive programs studied in the literature since the number of variables is very small. 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 (**?**). 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.

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.

[7] The set of ancestor W-nodes of an entry node can be efficiently obtained using a stack during the traversal (**?**). The details are omitted due to space constraints.
[8] The tuple denoted by $p[\overline{X}]$ is treated as a multiset of arguments when involved in equality checking.

Example 5
The DSE for the AND/OR tree in Fig. **??** is shown is shown below.

$$\begin{aligned}
&\texttt{arc}(\texttt{X},\texttt{Y})[\overline{X_1}] = \texttt{tc}(\texttt{X},\texttt{Y})[\overline{X_2}] \\
&\texttt{tc}(\texttt{X},\texttt{Z})[\overline{X_2}] = \texttt{tc}(\texttt{X},\texttt{Y})[\overline{X_2}] \\
&\overline{X_2} \subseteq \{1\} \\
&\texttt{tc}(\texttt{X},\texttt{Z})[\overline{X_2}] = \texttt{tc}(\texttt{Z},\texttt{Y})[\overline{X_2}] \\
&\emptyset \subsetneq \overline{X_1} \subseteq \{1,2\} \\
&\emptyset \subsetneq \overline{X_2} \subseteq \{1,2\}
\end{aligned} \tag{9}$$

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 an 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 and generates a corresponding `C++` program. We implemented the database objects (index and storage), base classes for each kind of node in the AND/OR tree and common functions. 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. The thread implementation provided by the Microsoft Windows runtime library is used on a Windows machine to evaluate the query in parallel, while Pthreads is used on a Linux machine.

5 Experimental Results

In this section, we report some experimental results on evaluating both non-recursive and recursive programs. The test machine has 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. All execution times are calculated by taking the average of five runs of the same experiment.

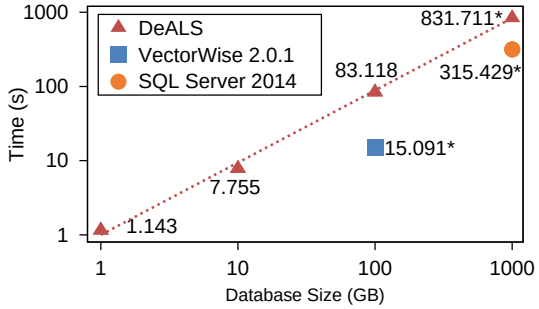

Fig. 2. Total query evaluation time for 22 queries in the TPC-H benchmark. A predicted query evaluation time on our test machine is marked with a * in the figure.

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 (**?**).[9] *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 12.43, 19.48, 23.63, respectively (evaluation time for all the 22 queries using one processor divides the time using 60 processors). Fig. **??** shows the total query evaluation time for all the 22 queries. The last point 831.711s 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 compare *DeALS* with 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,[10] while it takes Microsoft SQL Server 2014 Enterprise Edition 138s on a database of size 1TB.[11] Note that the SPECint_rate2006 of our test machine is 1050 (the larger the more powerful), while the values are 695 and 2400 for the other two machines. Fig. **??** shows the predicted time of both commercial RDBMSs on our test machine if the evaluation time is inversely proportional to the SPECint_rate2006 of the machine. *DeALS* is within six times slower comparing with these two highly optimized commercial systems (both system settings are also optimized for the benchmark).

Exp-II: Recursive programs. We test the performance of *DeALS* on three classical recursive queries — `tc` (program in (**??**)), `attend` (program in Example **??**) and `sg` (same generation) as shown below:

$$\begin{aligned} &\texttt{sg}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{anc}(\texttt{P},\texttt{X}), \texttt{anc}(\texttt{P},\texttt{Y}), \texttt{X} \neq \texttt{Y}. \\ &\texttt{sg}(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{anc}(\texttt{A},\texttt{X}), \texttt{sg}(\texttt{A},\texttt{B}), \texttt{anc}(\texttt{B},\texttt{Y}). \end{aligned} \tag{10}$$

The test datasets contain synthetic graphs and real world graphs.

Synthetic Graphs

[9] 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`.

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

[11] TPC-H Result on Cisco UCS C460 M4 Server, `http://www.tpc.org/3311`.

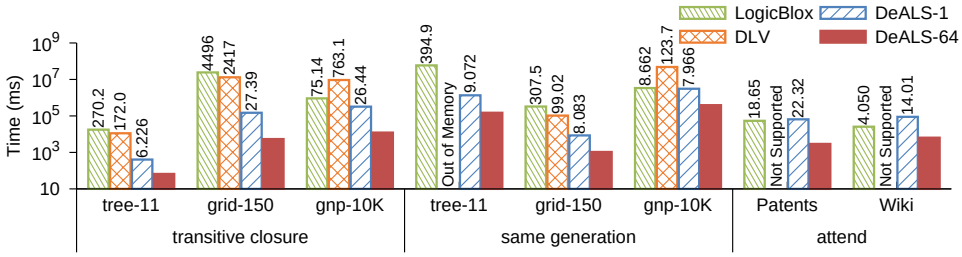

Fig. 3. Query evaluation time. The numbers above the bars for LogicBlox (DLV, DeALS-1) show the speedup of DeALS-64 over LogicBlox (DLV, DeALS-1).

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 random graph (Erdős-Rényi model) of 10,000 vertices generated by connecting vertices randomly such that the average out-degree of a vertex is 10.

Real World Graphs

1) patent is the US patent citation graph[12]. 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, and each edge represents an appearance of an entity in another entity's infobox. It has 3,165,181 vertices and 23,190,820 edges.

The experiments on `tc` (`sg`) evaluation use each of these synthetic graphs as `arc` (`anc`). The experiments on `attend` evaluation use each real world graphs as `friend`, while `organizer` contains all the vertices in the graph whose in-degrees are zero.

We compare $\mathcal{DeALS}$ with LogicBlox 4.1.9 (**?**) and DLV (**?**), 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. 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 on the evaluation of `sg` on tree-11. The version of DLV[13] 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. Fig. **??** compares the evaluation time of three systems on these recursive queries. Bars for LogicBlox show the evaluation time of LogicBlox using 64 processors.[14] Bars for DLV show the evaluation time of DLV using one processor.[15] Bars for DeALS-1 and DeALS-64 show the evaluation time of $\mathcal{DeALS}$ using one processor and 64 processors,

[12] Patent citation network, `https://snap.stanford.edu/data/cit-Patents.html`.
[13] DLV with Recursive Aggregates, downloaded from `http://www.dbai.tuwien.ac.at/proj/dlv/dlvRecAggr/dl-recagg-snapshot-2007-04-14.zip`.
[14] In our experiments, LogicBlox 4.1.9 does not utilize all the processors all the time.
[15] The single-processor version of DLV is 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.

respectively. When $\mathcal{DeALS}$ uses only one processor, it always outperforms DLV on these queries and datasets, and it outperforms or equally performs LogicBlox. $\mathcal{DeALS}$ always outperforms LogicBlox when it uses 64 processors. It 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 achieves limited speedup for `tc` on `tree-11` since evaluation time is dominated by barrier synchronization overhead (**?**).

6 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 (**?**; **?**; **?**; **?**; **?**; **?**; **?**). 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 (**?**; **?**; **?**) 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.

7 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. $\mathcal{DeALS}$ 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 (**?**). 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.

Acknowledgment

This work was supported by NSF grants IIS 1218471 and IIS 1118107. We would like to thank the reviewers and Matteo Interlandi for their comments. We thank

LogicBlox especially Martin Bravenboer, Dung Nguyen, and Yannis Smaragdakis, for their assistance with the LogicBlox comparison.