Title
Declarative Languages and Scalable Systems for Graph Analytics and Knowledge Discovery

Author
Yang, Mohan

Publication Date
2017

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Declarative Languages and Scalable Systems for Graph Analytics and Knowledge Discovery

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Mohan Yang

2017

ABSTRACT OF THE DISSERTATION

Declarative Languages and Scalable Systems for
Graph Analytics and Knowledge Discovery

by

Mohan Yang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2017
Professor Carlo Zaniolo, Chair

The growing importance of data science applications has motivated great research interest in powerful languages and scalable systems for supporting advanced analytics on massive data sets. Languages such as R and Scala are used to develop advanced analytical applications that are not supported by SQL, the traditional query language used for decades to search the database and analyze its data. An interesting research question that arises in this scenario is whether it is possible to design an efficient query language that simplifies the writing of advanced analytical applications and provides a unified environment for their development and deployment on multiple platforms, including massively parallel ones. In this thesis, we provide a positive answer to this question by demonstrating extensions of the logic-based query language Datalog and their implementation techniques to enable (i) scalable support for graph analytics and knowledge discovery applications, and (ii) portability between multicore machines and clusters.

A first set of extensions discussed in this thesis is based on monotonic aggregates and led to the implementation of our Deductive Application Language (DeAL) system which (i) achieves superior performance for graph analytics applications compared with other Datalog systems on multicore machines, and (ii) outperforms other distributed Datalog systems, as well as both GraphX and native Apache Spark. We then tackle the difficult problem of supporting knowledge discovery applications, by introducing non-monotonic extensions to support generic user-defined aggregates, for which we provide a formal logic-based semantics. The Knowledge Discovery in Datalog

(KDDlog) language so derived can express efficiently both descriptive analytics, such as rollups and data cubes, and predictive analytics, such as association rule mining, classification, regression analysis, and cluster analysis.

The dissertation of Mohan Yang is approved.

Junghoo Cho

Tyson Condie

Vwani P. Roychowdhury

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2017

To my parents and wife

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

I am thankful to my advisor, Carlo Zaniolo, who guided me throughout my Ph.D. study. He is a great researcher who showed me how to real research, and a great advisor who always provided me with advice and support whenever I needed. I am really fortunate to work with and learn from him during my years at UCLA. I own special thanks to Haixun Wang, who was my mentor at Microsoft Research Asia, and introduced me to Carlo. I would also like to thank my committee, Junghoo Cho, Tyson Condie, and Vwani P. Roychowdhury, for their time and help over the years.

I want to thank my beloved wife, Yifan, who has always been understanding and supportive. She shared all my happiness and sadness. I could not have gone so far without her company. I am grateful to my parents for their unconditional love. I spent very little time with them since I went to study in Wuhan in 2003. But their care and support has always been with me no matter how far away I am. I also want to thank my whole family for their help and understanding.

I would like to thank all my friends and fellow students at UCLA for their friendship and collaboration. This list includes but is not limited to Muhao Chen, Xuelu Chen, Hsuan Chiu, Ariyam Das, Shi Gao, Jiaqi Gu, Cheng-Kang Hsieh, Matteo Interlandi, Nikolay P. Laptev, Yuchen Liu, Giuseppe M. Mazzeo, Hamid Mousavi, Alexander Shkapsky, Dong Wang, Jin Wang, Zhiyang Wang, and Kai Zeng. I would also like to thank our department staffs, especially Charlie Fritzius and Peter Schultze, for their help in setting up and managing servers that I used throughout my Ph.D. study. Finally, but not lastly, I would like to thank all the people I worked with during the two wonderful summers at Microsoft Research, including Kaushik Chakrabarti, Surajit Chaudhuri, Tao Cheng, Bolin Ding, and Manoj Syamala.

VITA

2009	Research Intern, Microsoft Research Asia, Beijing, China.
2010	Bachelor of Engineering in Computer Science, Shanghai Jiao Tong University, Shanghai, China.
2010-2011	Software Engineer, fang.com, Beijing, China.
2012	Research Intern, Microsoft Research, Redmond, WA.
2012	Teaching Assistant, Computer Science Department, UCLA.
2013	Research Intern, Microsoft Research, Redmond, WA.
2014	Master of Science in Computer Science, UCLA.
2013-2016	Graduate Student Researcher, Computer Science Department, UCLA.

PUBLICATIONS

Mohan Yang, Alexander Shkapsky, Carlo Zaniolo. Scaling up the Performance of more Powerful Datalog Systems on Multicore Machines. *The VLDB Journal*, 2016.

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

Carlo Zaniolo, Mohan Yang, Ariyam Das, Matteo Interlandi. The Magic of Pushing Extrema into Recursion: Simple and Powerful Datalog Programs. In *AMW*, 2016.

Alexander Shkapsky, Mohan Yang, Carlo Zaniolo. Optimizing Recursive Queries with Monotonic Aggregates in DeALS. In *ICDE*, pp. 867–878. IEEE, 2015.

Mohan Yang, Alexander Shkapsky, Carlo Zaniolo. Parallel Bottom-Up Evaluation of Logic Programs: DeALS on Shared-Memory Multicore Machines. In *Technical Communications of ICLP*, 2015.

Mohan Yang, Bolin Ding, Surajit Chaudhuri, Kaushik Chakrabarti. Finding Patterns in a Knowledge Base using Keywords to Compose Table Answers. *PVLDB*, 7(14):1809–1820, 2014.

Mohan Yang, Carlo Zaniolo. Main Memory Evaluation of Recursive Queries on Multicore Machines. In *IEEE BigData*, pp. 251–260. IEEE, 2014.

Kai Zeng, Mohan Yang, Barzan Mozafari, Carlo Zaniolo. Complex Pattern Matching in Complex Structures: the XSeq Approach. In *ICDE*, pp. 1328–1331. IEEE, 2013.

CHAPTER 1

Introduction

The growing importance of big data applications is promoting a burgeoning research interest in parallel systems and algorithms needed to achieve scalable performance on various parallel platforms, including multicore machines and clusters. A wide spectrum of systems are being proposed for the development of advanced analytics algorithms on these parallel platforms. Among them, we should mention MapReduce [DG08], DryadLINQ [YIF08], Scope [CJL08], Pig [ORS08], Hive [TSJ09], Pregel [MAB10], Hyracks [BCG11], GraphLab [LBG12], Spark [ZCD12], and Flink [fli]. While these systems feature many differences in the techniques they use, they all seek to provide levels of *scalability*, *usability*, and *portability* that can compete with those of DBMSs, which for traditional applications provided (i) scalability via parallelization, (ii) usability via declarative query languages, and (iii) portability via compliance with SQL standards.

However, extending SQL and DBMSs to provide efficient support for advanced analytics, such as graph applications, and knowledge discovery and data mining (KDD) algorithms, proved exceedingly difficult. In particular, the difficulty of integrating into IBM DB2 the Apriori algorithm for frequent itemset mining was documented in a much cited paper [STA98]. Instead of using SQL, most advanced analytics tasks are currently supported in DBMSs through vendor-specific built-in functions implemented in low-level languages such as C and Java. A similar approach is also adopted by most big data analytics systems mentioned above, where they usually provide (i) a high-level language, such as SQL, to express traditional data intensive applications that can be optimized and executed on different platforms, and (ii) several subsystems where each has (a) a set

of APIs that can be used with low-level languages, such as R and Scala, to develop programs for a specific type of applications, and (b) a library of functions that implement popular applications of the specific type using the subsystem APIs. For example, Spark supports SQL queries through a subsystem called Spark SQL [AXL15], while graph computations and KDD applications are supported through GraphX [GXD14] and MLlib [MBY16], respectively.

Despite the huge success achieved by the big data analytics systems, especially Spark, for providing scalable support for diverse types of applications within one system, the following two limitations remain:

1. **Limited usability**. The knowledge about each subsystem and the corresponding programming model is required to develop complex applications that involve several types of analytics. Thus, such applications can only be developed by very experienced users. However, even for these users, it is still a non-trivial task to pick the most suitable subsystem for a specific application at hand, i.e., there is no clear rule to decide which subsystem to use when one application can be supported by several different subsystems. On the other hand, the domain experts, who frequently lack the time or the skills required to develop/modify the programs written in low-level languages, are bound by the limited functionalities provided by the system libraries.

2. **Missed optimization opportunities**. Although each subsystem still performs its subsystem-specific optimizations, different subsystems are normally not aware of the existence of each other. As a result, the underlying system does not have a global view about complex applications that involve the use of several subsystems, thus cannot find the execution plans that minimize the total time spent on different subsystems.

An interesting research question that arises in this scenario is whether it is possible to design a high-level declarative language that can simplify the development of diverse analytical applications via a unified environment that facilitates their development and scalable deployment on multiple platforms. In this thesis, we provide a positive answer to this question by demonstrating extensions of the logic-based query language Datalog that enable (i) efficient and scalable support for graph

analytics and KDD applications, and (ii) portability between multicore machines and clusters. More specifically, we have developed the following languages and systems:

Deductive Application Language (DeAL) System. Our first set of extensions based on *monotonic aggregates*, which (i) are recently introduced for use in recursive rules [MSZ13a, MSZ13b], and (ii) are critical in expressing many graph applications, lead to the implementation of our DeAL system. A key component in Datalog program evaluation is the efficient support for recursion, and our experimental study on finding an efficient and scalable parallel evaluation strategy for basic recursive queries was presented in [YZ14a]. Then, the resulting evaluation strategy was generalized and used in the DeAL system for in-memory parallel query evaluation on multicore machines [YSZ15, YSZ16]. We designed and implemented a system that delivers (i) competitive performance on the non-recursive queries of the TPC-H benchmark [tpca], compared with the state-of-the-art DBMSs such as Vectorwise [vec] and SQL Server [sql], and (ii) superior performance on recursive queries compared with Datalog systems on multicore machines, namely CLINGO [GKK14], DLV [LPF06], LogicBlox [ACG15], and SociaLite [SPS13].

Finally, our BigDatalog system that focused on the distributed evaluation of the same set of queries on clusters was presented in [SYI16]. In fact, BigDatalog (i) outperforms or performs as well as other distributed Datalog systems, including SociaLite and Myria [WBH15], in speed and scalability, and (ii) often outperforms many applications written in native Spark from which BigDatalog is developed, and GraphX [GXD14] which is the dedicated graph processing subsystem on Spark.

Knowledge Discovery in Datalog (KDDlog) Language. We extend DeAL to tackle the difficult problem of supporting KDD applications that lead to the design of our KDDlog language [YZ16]. The language is built on top of a query optimization technique called *constrained least fixpoint* that allows the use of non-monotonic constructs in the iterative computation of recursive rules. We introduce the notion of *genericity* for KDD algorithms, i.e., algorithms that can operate on tables having arbitrary number of columns, and we propose constructs that achieve genericity by supporting verticalized representations for tables. While vertical representations are sufficient in

the implementation of simple KDD tasks, such as naive Bayes classifiers, more compact representations are needed for advanced KDD tasks. Thus we introduce more efficient representations, called *prefix tables*, for rollups and data cubes. Then we show how these new constructs that are designed for descriptive analytics can be used to express efficiently predictive analytics, including association rule mining, classification, regression analysis, and cluster analysis, thus leading to a unified support for both descriptive analytics and predictive analytics. Finally, the language supports user-defined aggregate functions which enable an open environment where we can export KDD algorithms implemented in KDDlog and import algorithms implemented in other languages.

The rest of this thesis is organized as follows. In Chapter 2, we present our experimental study on finding the most efficient transitive closure query evaluation algorithm on multicore machines. Then we describe techniques used in our DeAL system to support efficient in-memory parallel bottom-up evaluation of Datalog programs on multicore machines in Chapter 3, followed by an overview of our BigDatalog system for the distributed evaluation of Datalog programs on clusters in Chapter 4. After that, we present the KDDlog language that supports KDD applications in a declarative language in Chapter 5. Finally, we conclude and discuss future work in Chapter 6.

CHAPTER 2

Transitive Closure Queries on Multicore Machines

In this chapter, we present an experimental study on finding efficient main memory recursive query evaluation algorithms on modern multicore machines.

2.1 Introduction

As we have moved from the simple applications originally supported by MapReduce to more advanced applications requiring iteration and/or recursion, it is now clear that classical algorithms designed for traditional architectures should be re-evaluated and re-designed for these new massively parallel systems. For instance, several recent studies [ABC11, AU12, SKH12] have focused on the implementation of transitive closure (TC) in clusters, and Afrati et al. [ABC11] showed that a relatively obscure algorithm called SMART [Ioa86, VB86] outperforms other algorithms on this problem.

However, algorithms that deliver optimal performance on clusters are hardly optimal on multicore machines, and vice versa: in the rest of the chapter we demonstrate this point by an in-depth experimental study of various transitive closure algorithms on multicore machines. Thus, we will first show that many other algorithms are significantly better than SMART, and then propose a hybrid algorithm that achieves the best performance by combining two existing algorithms.

The novel performance findings presented here are hardly surprising in view of the fact that the most previous studies date back to the late 1980 s and early 1990 s [VK88, AJ88, WZB92, CCH93],

and there has been much progress in multicore systems since then. Moreover, we assume here that all our data resides in main memory, whereas past studies on recursive query evaluation [War62, War75, Ioa86, VB86, AJ87, AJ88, IR88, VK88, AJ90, Jak91, KIC92, WZB92, CCH93, DR94] often assumed a database-oriented environment with data residing on secondary storage, whereby query evaluation algorithms were designed to reduce I/O costs rather than in-memory evaluation costs.

We begin with a study of the TC query evaluation using the semi-naive fixpoint computation (denoted by SEMINAIVE) and then SMART. Then we study SSC algorithms that decompose the TC computation into disjoint computations, each computing the closure from a single vertex using linear recursive rules. The two SSC algorithms are SSC1 based on SEMINAIVE, and SSC2 based on the breadth-first search. We also propose a new hybrid SSC algorithm, called SSC12, which integrates the merits of SSC1 and SSC2. We perform an experimental evaluation of our implementations focusing on memory utilization, speed, and scalability.

2.2 Transitive Closure Algorithms

2.2.1 Linear TC Rules and Semi-Naive Evaluation

Let $\texttt{arc}(\texttt{X}, \texttt{Y})$ be a relation that represents the edges of a directed graph, i.e., there is a directed edge from x to y if and only if $\texttt{arc}(\texttt{x}, \texttt{y})$ is a fact. The transitive closure (TC) of arc is a relation $\texttt{tc}(\texttt{X}, \texttt{Y})$ such that tc contains all pairs $(\texttt{X}, \texttt{Y})$ that X can reach Y via a path in the graph. A linear version of TC is given by the rules below.

$$\texttt{tc}(\texttt{X}, \texttt{X}) \texttt{<-} \texttt{arc}(\texttt{X}, \_).$$
$$\texttt{tc}(\texttt{X}, \texttt{Y}) \texttt{<-} \texttt{tc}(\texttt{X}, \texttt{Z}), \texttt{arc}(\texttt{Z}, \texttt{Y}).$$

The first rule is an exit rule[1]. It adds a tuple $(\texttt{X}, \texttt{X})$ to tc for each source vertex X. The second rule is a left-linear recursive rule. For every tuple $(\texttt{X}, \texttt{Z})$ that is already in tc, it expands the path represented by $(\texttt{X}, \texttt{Z})$ with one more edge, then adds the new tuple to tc. The relation tc can be computed by iteratively performing this operation until no more new tuples can be added to it. This

[1] A more common way to write the exit rule is $\texttt{tc}(\texttt{X}, \texttt{Y}) \texttt{<-} \texttt{arc}(\texttt{X}, \texttt{Y})$. The exit rule here ensures that tc always contains the tuple $(\texttt{X}, \texttt{X})$ for each source vertex X.

procedure is called the *naive evaluation*. This simple procedure may involve redundant derivations since the same path may be generated in several iterations.

SEMINAIVE **Algorithm**

Semi-naive evaluation [ZCF97] is an optimized variant of naive evaluation. The idea is to use only the new tuples derived in the previous iteration to derive the tuples in the current iteration. The pseudocode is shown in Figure 2.1.

1: $\texttt{tc} := \{(\texttt{X},\texttt{X})|\texttt{arc}(\texttt{X},\_)\}$, $\Delta\texttt{tc} := \{(\texttt{X},\texttt{X})|\texttt{arc}(\texttt{X},\_)\}$
2: **repeat**
3: $\quad\delta\texttt{tc} := \pi_{\texttt{X},\texttt{Y}}\big(\Delta\texttt{tc}(\texttt{X},\texttt{Z}) \bowtie \texttt{arc}(\texttt{Z},\texttt{Y})\big)$
4: $\quad\Delta\texttt{tc} := \delta\texttt{tc} - \texttt{tc}$
5: $\quad\texttt{tc} := \texttt{tc} \cup \Delta\texttt{tc}$
6: **until** $\Delta\texttt{tc} = \emptyset$

Figure 2.1: SEMINAIVE algorithm for computing tc.

SEMINAIVE evaluates as follows. First, all source vertices in arc are added into tc and Δtc, i.e., tc and Δtc contain tuples for all paths of length 0. Then, in the i-th iteration, the initial Δtc contains tuples for paths of length $i-1$. δtc contains tuples corresponding to paths of length i derived from extending paths of length $i-1$ with one more edge. However, some tuples may already be present in tc. It is not necessary to do derivations on these tuples since it will only derive tuples that are already derived. So the new Δtc excludes these tuples that are already in tc. All tuples in the new Δtc are merged into tc. The algorithm iterates until Δtc becomes empty.

The number of iterations required by SEMINAIVE equals the length of the longest simple path in the graph. The maximal value is $n-1$ for a graph of n vertices. Thus, the algorithm could take $O(n)$ iterations to terminate.

2.2.2 Nonlinear TC Rules and SMART Algorithm

The number of iterations required by SEMINAIVE is large when the length of the longest simple path is very long. However the use of quadratic recursive rules, as shown below, doubles the length of the paths at each iteration thus reducing the number of iterations required to their logarithm.

$$\texttt{tc}'(\texttt{X},\texttt{X}) \texttt{ <- } \texttt{arc}(\texttt{X},\_).$$

$$\texttt{tc}'(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{arc}(\texttt{X},\texttt{Y}).$$

$$\texttt{tc}'(\texttt{X},\texttt{Y}) \texttt{ <- } \texttt{tc}'(\texttt{X},\texttt{Z}), \texttt{tc}'(\texttt{Z},\texttt{Y}).$$

The SMART algorithm [Ioa86, VB86] optimizes the computation of these rules by avoiding the generation of the same path multiple times. The pseudocode for SMART is as follows:

1: $\texttt{tc}' := \{(\texttt{X},\texttt{X}) | \texttt{arc}(\texttt{X},\_)\}$, $\delta\texttt{tc}' := \texttt{arc}$
2: **repeat**
3: $\quad \Delta\texttt{tc}' := \pi_{\texttt{X},\texttt{Y}}\big(\delta\texttt{tc}'(\texttt{X},\texttt{Z}) \bowtie \texttt{tc}'(\texttt{Z},\texttt{Y})\big)$
4: $\quad \texttt{tc}' := \texttt{tc}' \cup \Delta\texttt{tc}'$
5: $\quad \delta\texttt{tc}' := \pi_{\texttt{X},\texttt{Y}}\big(\delta\texttt{tc}'(\texttt{X},\texttt{Z}) \bowtie \delta\texttt{tc}'(\texttt{Z},\texttt{Y})\big) - \texttt{tc}'$
6: **until** $\delta\texttt{tc}' = \emptyset$

Figure 2.2: SMART algorithm for computing $\texttt{tc}'$.

At the beginning of iteration i, $\texttt{tc}'$ contains all tuples corresponding to paths of length at most $2^{i-1} - 1$, and $\delta\texttt{tc}'$ contains all tuples corresponding to paths of length exactly 2^{i-1} that are not in $\texttt{tc}'$. This condition holds in the first iteration as $\texttt{tc}'$ is set to the set of tuples corresponding to paths of length $2^0 - 1 = 0$ in line 1, and $\delta\texttt{tc}'$ is set to the set of tuples corresponding to paths of length $2^0 = 1$ in line 1. In line 3, all tuples corresponding to paths of length between 2^{i-1} and $2^i - 1$ are derived by joining $\delta\texttt{tc}'$ and $\texttt{tc}'$. These tuples are merged into $\texttt{tc}'$ in line 4. Now $\texttt{tc}'$ contains all tuples corresponding to paths of at most $2^i - 1$. In line 5, $\delta\texttt{tc}'$ is joined with itself to derive all tuples corresponding to paths of length exactly 2^i. Tuples that are already in $\texttt{tc}'$ are excluded from the new $\delta\texttt{tc}'$. So the condition still holds for iteration $i + 1$. The algorithm iterates until $\delta\texttt{tc}'$ becomes empty. In an n-vertex graph, there is no simple paths of length n. Thus, $\delta\texttt{tc}'$ is empty at the end of iteration $\lceil \log n \rceil$, or the algorithm terminates in $O(\log n)$ iterations.

2.2.3 Single-Source Closure Algorithms: SSC1 and SSC2

The previous algorithms compute tuples from different source vertices at the same time, whereas a more parsimonious usage of memory can be achieved by computing the paths that originate from one source vertex one at the time [Jak91].

In Datalog, we can express the optimization by replacing each goal $\mathtt{tc}(\mathtt{X}, \mathtt{Y})$ with the one-column predicate $\mathtt{tc}''$—for each value $\mathtt{x}$ that satisfies $\mathtt{arc}(\mathtt{X}, \_)$:

$$\begin{aligned} &\mathtt{tc}''(\mathtt{x}). \\ &\mathtt{tc}''(\mathtt{Y}) \text{ <- } \mathtt{tc}''(\mathtt{Z}), \mathtt{arc}(\mathtt{Z}, \mathtt{Y}). \end{aligned} \tag{2.1}$$

The closure under the operation defined by the rules in Equation (2.1) contains all vertices reachable from the source vertex $\mathtt{x}$. We call it a single-source closure (SSC). An SSC algorithm computes TC by computing the SSC for every source vertex in the graph. It decomposes the original computation into disjoint computations based on the source vertex.

The SSC1 Algorithm

A straightforward way to compute the SSC of a source vertex $\mathtt{x}$ is to apply SEMINAIVE to the rules in Equation (2.1). The algorithm, named as SSC1, is shown in Figure 2.3. For each vertex $\mathtt{Z}$ in $\Delta\mathtt{tc}''$, it finds all the $\mathtt{Y}$ that satisfies $\mathtt{arc}(\mathtt{Z}, \mathtt{Y})$, and adds $\mathtt{Y}$ to $\delta\mathtt{tc}''$. All the vertices that are already in $\mathtt{tc}''$ are excluded from the new $\Delta\mathtt{tc}''$, and the remaining vertices are added to $\mathtt{tc}''$. When the evaluation terminates, $\mathtt{tc}''$ contains all the vertices in the SSC of $\mathtt{x}$. We compute the TC by repeating SSC1 on all source vertices.

```
1: tc'' := {x}, Δtc'' := {x}
2: repeat
3:    δtc'' := π_Y(Δtc''(Z) ⋈ arc(Z, Y))
4:    Δtc'' := δtc'' − tc''
5:    tc'' := tc'' ∪ Δtc''
6: until Δtc'' = ∅
```

Figure 2.3: SSC1 algorithm for computing $\mathtt{tc}''$.

SSC1 performs exactly the same (logical) computation as SEMINAIVE does. The only difference is the computation is partitioned based on the source vertex. The effect of this is similar to hashing. For example, the set difference in line 4 of Figure 2.1 is replaced by many set differences in line 4 of Figure 2.3 which is equivalent to a hash-based set difference where the hash function simply returns the source vertex of a tuple. As we will see, SSC1 normally outperforms

SEMINAIVE which is slower because of the overhead of hashing and related operations.

The SSC2 Algorithm

The SSC of x is represented as a set in SSC1. An alternative representation is a Boolean array of size n where the i-th element represents whether x can reach the vertex i in the graph[2]. This array representation converts SSC1 to the SSC2 algorithm shown in Figure 2.4. The algorithm essentially performs a breadth-first search starting from x. d, $\Delta\mathtt{tc}''$, and $\delta\mathtt{tc}''$ are three arrays of size n which are reused throughout the evaluation for all source vertices. Initially, all elements in d are set to **false** except $d[\mathtt{x}]$. In each iteration, $\Delta\mathtt{tc}''$ and $\delta\mathtt{tc}''$ contain the vertices derived in the last iteration and the current iteration, respectively. For each vertex Z in $\Delta\mathtt{tc}''$, edges starting from Z are explored to derive new vertices. When a new vertex Y is derived, we check if Y is already in $\mathtt{tc}''$ by testing if $d[\mathtt{Y}]$ is **true**. If not, we set $d[\mathtt{Y}]$ to **true**, and then add Y to $\delta\mathtt{tc}''$. The check in line 8 replaces the set difference in line 4 of Figure 2.3, while the operation of setting $d[\mathtt{Y}]$ to **true** replaces the union in line 5 of Figure 2.3. When the algorithm terminates, the actual SSC is constructed by collecting all vertices Y where $d[\mathtt{Y}]$ is **true**.

```
 1: set each element in d[] to false
 2: d[x] := true, Δtc''[0] := x, L := 1
 3: repeat
 4:    l := 0
 5:    for i := 0 to L − 1 do
 6:       Z := Δtc''[i]
 7:       for each edge (Z, Y) in arc do
 8:          if d[Y] = false then
 9:             d[Y] := true, δtc''[l] := Y, l := l + 1
10:    Δtc'' := δtc'', L := l
11: until L = 0
```

Figure 2.4: SSC2 algorithm for computing $\mathtt{tc}''$.

The array representation allows SSC2 to replace the set operations (insert, set difference, and union) with array accesses. This optimization reduces the time of computation (lines 2–11 in Figure 2.4) at the expense of additional time on array initialization (line 1) which is proportional

[2]We assume each vertex is encoded as an integer ranging from 0 to $n-1$ in an n-vertex graph.

to n. If n is very large, but very few edges are explored during the computation, the time spent on array initialization may be longer than the computation. In this case, SSC2 is slower than SSC1. On the other hand, if the algorithm explores many edges during the computation, the advantage of the array representation becomes clear, and SSC2 becomes faster than SSC1.

2.2.4 An Adaptive Single-Source Algorithm: SSC12

The performance of the previous two SSC algorithms varies on different source vertices. We propose a hybrid SSC algorithm, named as SSC12, which is a trade-off between SSC1 and SSC2. Evaluation starts with SSC1, and converts to SSC2 when the algorithm predicts that the time would be shorter if it converts to SSC2.

If SSC2 is faster than SSC1 on a source vertex, the optimal conversion point is the beginning of the evaluation. But the prediction is difficult without computing the SSC. To control the conversion of the hybrid algorithm, we use a heuristic algorithm as shown in Figure 2.5. $\delta\mathtt{tc}''$ and $\Delta\mathtt{tc}''$ are represented as sets in SSC1. Assume the cost of set insert and delete is the same. Let $|adj(\mathtt{Z})|$ be the number of $\mathtt{Y}$ that satisfies $\mathtt{arc}(\mathtt{Z},\mathtt{Y})$. The number of set operations performed to compute $\delta\mathtt{tc}''$ and $\Delta\mathtt{tc}''$ are $C_\delta = \sum_{\mathtt{Z}\in\Delta\mathtt{tc}''} |adj(\mathtt{Z})| + |\mathtt{tc}''|$ and $C_\Delta = |\mathtt{tc}''| + |\Delta\mathtt{tc}''|$, respectively, which are simple to compute. The algorithm chooses to convert if $C_\delta > n/\alpha$ or $C_\Delta > n/\beta$, where α and β are parameters that control the timing of conversion. It degenerates to SSC1 (resp., SSC2) if $\alpha = \beta = 0$ (resp., $\alpha = \beta = \infty$).

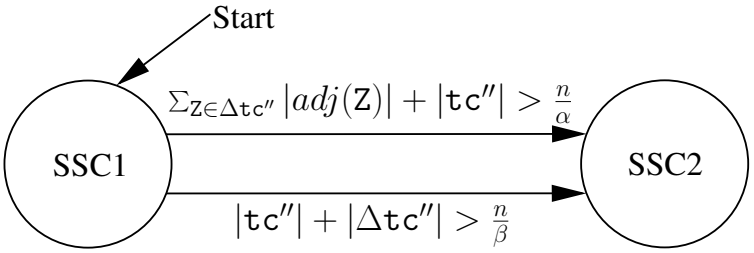

Figure 2.5: Control algorithm for SSC12 algorithm.

A sufficiently small α ensures that the time becomes shorter if the algorithm chooses to convert since SSC2 is expected to be faster than SSC1 for a large C_δ. But the algorithm may have to execute SSC1 for a long time to obtain such a decision. Thus, the effect of conversion on speedup the computation is diminished as the time of conversion may be too late. On the other hand, large

α may lead to wrong predictions which slow down the evaluation. The same dilemma applies to β. We describe how to tune these two parameters in Section 2.4.2.

2.3 Algorithm Implementation and Memory Usage

We will next describe in more details the implementation of these algorithms which we then compare in terms of memory utilization, speed, and scalability. As discussed in the introduction, the result of this comparison will help programmers implementing recursive applications, and it is actually critical for a compiler optimizing the execution of TC Datalog program on multicore machines. In the previous section, we have seen how a simple rewriting can be used to redirect the execution of linear rules from SEMINAIVE to SSC1, which can in turn be recast as SSC2 and SSC12 (by the compiler using different memory structures). Rewriting rules that transform linear recursive rules into nonlinear rules and then these to SEMINAIVE and SMART respectively are also available. Thus all these algorithms represent achievable targets for a parallel Datalog compiler, which will then select the optimal one for the system at hand. Afrati et al. [ABC11] have shown that SMART is optimal for clusters, and in the rest of the chapter we seek to resolve the optimality question for multicore machines.

2.3.1 Main Memory Representation

The different algorithms achieve their best performance with different representations. Thus SMART performs well when `arc` is represented as a collection of tuples, but the performance of SEMINAIVE and the SSC algorithms improves significantly when an *adjacency list* representation is used for `arc`. The time required for structuring the input data into an adjacency list and building an index for it is included in the total time reported in our experiments (but it is small and only accounts for less than 2% of the total time).

Adjacency List Index

The operation of deriving new tuples in SEMINAIVE can be simply implemented as a nested loop join between Δ`tc` and `arc`. But it requires a full scan on `arc` for each tuple in Δ`tc`. Note that `arc` does not change during the evaluation. A better strategy is to build an index on `arc` so that all tuples

with a specific source vertex can be accessed directly. The index is an *adjacency list* representation of the graph represented by `arc` where each vertex `X` is associated with an unordered list $adj(\texttt{X})$ describing the set of neighbors of this vertex. It is built by scanning `arc` twice:

1. The first scan counts how many neighbors each vertex has. These values are stored in an array of size n.

2. A contiguous memory space is allocated for the adjacency list. The starting position within the allocated space for the unordered list associated with each vertex is determined by computing the prefix-sum of the array obtained in Step (1).

3. During the second scan, the destination vertex of each tuple is stored in the list associated with the source vertex.

Now, for each tuple $(\texttt{X}, \texttt{Z})$ in $\Delta\texttt{tc}$, the algorithm retrieves from the index the list $adj(\texttt{Z})$ that contains all neighbors of `Z`. For each vertex $\texttt{Y} \in adj(\texttt{Z})$, it generates a new tuple $(\texttt{X}, \texttt{Y})$. There is no redundant accesses on `arc`. This index is also used in SSC1, SSC2, and SSC12.

2.3.2 Implementation of SEMINAIVE and SMART

Figure 2.1 and Figure 2.2 contain three basic relational algebra operators: join, union and set difference. We implemented both algorithms using hash-based parallel relational algebra operators. We discuss possible alternative implementations in Section 2.4.5.

The join operator in line 3 of Figure 2.2 joins two large relations. Hash join is an efficient main memory join algorithm for this scenario. The standard hash join algorithm builds a hash table using the tuples in the smaller of the two input relations, and probes the hash table to find possible matches for each tuple in the remaining input relation. If the hash table is very huge, the random memory access pattern in building and probing of the hash table may lead to many cache misses. Shatdal et al. [SKN94] proposed to partition both relations into smaller relations to reduce the cache misses. Manegold et al. [MBK02] introduced the multi-pass radix partitioning to reduce the translation lookaside buffer misses during the partitioning. Kim et al. [KKL09] presented the radix

join algorithm on multi-core machines, which is a parallel implementation of the radix partitioning based join algorithm. In this study, we implement the join operator using the radix join algorithm.

The complete radix join algorithm is shown in Figure 2.6. Given two input relations R and S, a hash function $h : \{0,1\}^* \rightarrow \{0,1\}^N$, both relations are partitioned into $k = 2^B$ partitions R_i and S_i with respect to the hash function h. The partitioning is achieved in multiple passes where the i-th pass uses B_i bits from the hash value of a tuple, starting from the least significant bit ($\sum B_i = B$). For example, assume that the hash function h maps a tuple to a 32-bit integer, $B_1 = B_2 = 7$, then the two hash functions in Figure 2.6 are $h_1 = h \,\&\, 7F$ and $h_2 = h \,\&\, 3F80$. The first pass partitions R into $2^{B_1} = 128$ partitions by looking at bits 0–6 in the hash value, and the second pass partitions each of the 128 partitions in the first pass into $2^{B_2} = 128$ partitions by looking at bits 7–13 in the hash value. Similar for relation S. When the partitioning finishes, the two input relations R and S become 2^B pairs of relations R_i and S_i. For each R_i and S_i pair, a hash table is built using tuples in R_i, and is then probed for each tuple in S_i. If either R_i or S_i is empty, this pair can be ignored since the join result of these two relations is empty.

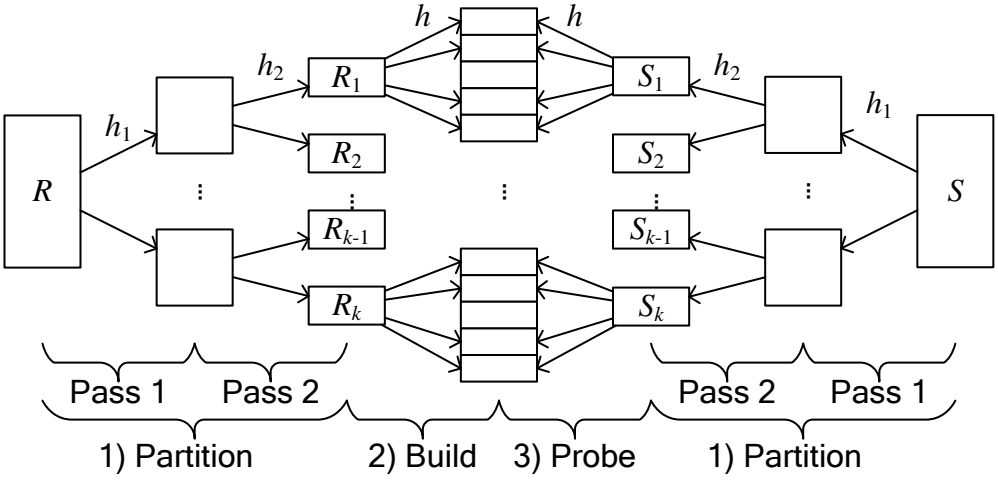

Figure 2.6: Radix join algorithm (based on Figure 4 in [BTA13]).

The first pass of partitioning in the radix join algorithm can be parallelized as follows: 1) equally assign the relation to each thread, and each thread scans its assigned tuples once to count how many tuples belong to a partition; 2) compute the starting offset of each partition in the final location for each thread; 3) each thread scans its assigned tuples again, copies each tuple to its

target partition. The partitions resulted from the first pass of partitioning are inserted into a queue. Each thread takes a partition from the queue and performs the second pass of partitioning on it concurrently. The remaining passes of partitioning can also be parallelized in this way. Finally, all R_i, S_i pairs are inserted into a queue, where each thread pops a pair and joins the two relations until it is empty.

The union operator in line 4 of Figure 2.2 represents the operation of merging the newly derived tuples in Δ`tc` into `tc`. This operation does not simply append Δ`tc` to the end of `tc`. Instead, it also performs deduplication to ensure the algorithm does not perform redundant derivations. An ideal way to achieve both at the same time is to use hash tables. We implemented the union operator using the same techniques used by the radix join to optimize main memory performance. We first partition both input relations into smaller pairs of relations R_i and S_i, and then build a hash table using all tuples in R_i and S_i for each pair of relation. The output contains all tuples in the hash table. If only one of R_i and S_i is empty, this pair should still be processed since the union of these two relations is not empty.

The set difference operator in line 5 of Figure 2.2 represents the operation of removing previous derived tuples in `tc` from δ`tc`. This operation also requires deduplication to ensure no extra derivations. We also use hash tables and techniques from the radix join to implement the set difference operator. Both input relations are partitioned into smaller pairs of relations R_i and S_i. A hash table is built using tuples in R_i, and is then probed for each tuple in S_i. Each matched tuple is removed from the hash table. All remaining tuples in the hash table are collected as the final result. If R_i is empty, this pair can be ignored since empty set minus another set is always empty.

In Figure 2.1, the join operator is implemented as an index nested loop join as described in lines 4–5. It is parallelized by equally assigning Δ`tc` to each thread. The set difference between δ`tc` and `tc` requires deduplication. It is implemented as described in the previous paragraph. After the set difference, Δ`tc` contains neither duplicate tuples nor tuples in `tc`. Thus, the union operator is simply implemented as appending Δ`tc` to the end of `tc`.

Our implementations use the Pthreads library. Each (working) thread is assigned to a unique physical core on a multicore machine, and performs all the computation on its assigned core. In

addition to the working threads, there is a control thread that coordinates the computation. In each iteration, 1) the control thread assigns the job to each working thread; 2) all working threads perform the computation; 3) the control thread allocates some memory, revokes some memory, assigns the job; 4) each working thread copies the tuples from its local buffer to its assigned location; 5) the control thread decides if the termination condition is satisfied, if not starts the next iteration. The execution time is divided into three parts: 1) *control time*, the time spent by the control thread; 2) *computing time*, when all working threads perform some computation; 3) *copying time*, when each working thread copies tuples to its assigned location.

2.3.3 Implementation of SSC Algorithms

Our implementation of different SSC algorithms uses different data structures. In SSC1, we use hash tables to represent $\texttt{tc}''$, $\Delta\texttt{tc}''$, and $\delta\texttt{tc}''$. In SSC2, we use arrays to represent d, $\Delta\texttt{tc}''$, and $\delta\texttt{tc}''$. In SSC12, we use both hash tables and arrays.

An SSC algorithm can be easily parallelized on a shared memory machine as follows: 1) add all source vertices into a queue; 2) each thread removes a vertex from the queue, and computes the SSC of this vertex; 3) repeat until the queue is empty. The queue is implemented as an array with a counter. Each element in the array is a vertex. The initial value of the counter is zero. A thread gets the index of the next vertex by calling the `gcc` atomic memory access function `__sync_fetch_and_add`. Calling the function requires an implicit synchronization between threads that are fetching the counter value at the same time. But the time spent on this synchronization is negligible compared to the control time in SEMINAIVE and SMART.

NUMA Aware Optimization. On non-uniform memory access (NUMA) hardware, the memory is configured into several NUMA regions and each region is attached to a unique CPU as its local memory. A CPU accesses its local memory faster than non-local memory. If the relation `arc` is resident on one CPU's local memory, the threads running on this CPU access the relation faster than the threads running on other CPUs. This unbalanced access speed slows down the computation when we use multiple threads running on different CPUs. If the relation fits in one NUMA region, we can duplicate the relation on each NUMA region such that each thread accesses

the (copy of) relation that is resident on the NUMA region attached to the CPU the thread is running on. SEMINAIVE, SSC1, SSC2, and SSC12 adopt this optimization since our experiments are performed on a NUMA machine. The adjacency list index is duplicated on each NUMA region. We discuss the impact of this optimization in Section 2.4.4.1. SMART does not adopt this optimization since `arc` is only used in the first iteration and the time spent on this iteration is very short comparing to the total time.

2.3.4 Memory Requirements

The main factors determining memory usage are as follows:

n	number of vertices in the graph
m	number of edges in the graph
m_c	number of tuples in the TC
p	number of threads used by an algorithm
b	number of bits to store a vertex

Now, Table 2.1 summarises the memory requirement of each implementation. It is clear to see the advantage of the SSC algorithms on the memory requirement. They use at most half (resp., one third) of the memory required by SEMINAIVE (resp., SMART) if $m \ll m_c$ (i.e., the TC contains much more tuples than `arc` does). Moreover, the SSC algorithms do not need to access the tuples in the TC. If the TC does not fit in the memory, an SSC algorithm can still executes as long as `arc` fits in the memory and the remaining memory is sufficient to hold the auxiliary data structures used by the algorithm. Instead of storing a newly derived tuple in the main memory, the algorithm now appends it to the end of a file on the disk.

Table 2.1: Memory requirements of implementations.

Algorithm	Memory Requirement (bits)
SEMINAIVE	$\geq \mathbf{4bm_c + 2bm}$
SMART	$> \mathbf{6bm_c}$
SSC1	$\leq \mathbf{2bm_c + bm + 6bpn}$
SSC2	$\mathbf{2bm_c + bm + (3b + 1)pn}$
SSC12	$\leq \mathbf{2bm_c + bm + (9b + 1)pn}$

2.4 Experimental Evaluation

All experiments are run on a multicore machine with four AMD Opteron 6376 CPUs and 256 GB memory (configured into 8 NUMA regions). Each CPU has 16 cores organized as follows: 1) each core has its own 16 KB L1 cache; 2) two cores share a 2 MB L2 cache; 3) eight cores share a 6 MB L3 cache, and have direct access to 32 GB memory. The operating system is Ubuntu Linux 12.04 LTS and the compiler is `gcc` 4.6.3 using `-O3` optimization. Execution time is calculated by taking the average of five runs of the same experiment[3]. Execution time is measured as the number of CPU cycles elapsed from start to finish for computing TC.

In the rest of this section, we first describe the topology of the test graphs, and then describe how the two parameters of SSC12 are determined using some test graphs. Finally, we present the experimental results on serial and parallel execution of the compared algorithms.

2.4.1 Topology of Test Graphs

In our experiments, we encode each vertex in an n-vertex graph as a random 32-bit integer ranging from 0 to $n-1$. Each edge is represented as a pair of 32-bit integers. The edges are shuffled into a random order and stored in a file. Recall that the graph has m edges. We say the graph is sparse if m is much less than n^2 (e.g., $m = cn$ or $m = cn \log n$ where c is a small constant); otherwise, we say it is dense (e.g., $m = cn^2$ or $m = cn^{\frac{3}{2}}$ where c is a small constant). The TC is also a directed graph. Similarly, we say the TC is sparse or dense based on whether the number of tuples in the TC is much less than n^2 or not. We evaluated the algorithms on synthetic graphs of four different topologies and four real-world graphs as shown in Table 2.2.

Synthetic Graphs.

1) Tree. A tree has a root whose in degree is zero. There is exactly one path between a vertex and each of its descendent vertex. We say a vertex v is a *level-l vertex* if there are l edges on the path from the root to v. The *depth of the tree* is the maximum value of l for all vertices in the tree. Given

[3]We are not reporting the maximal/minimum execution time since the corresponding line and the average line are almost coincident in the figures.

Table 2.2: Parameters of test graphs.

Name	Type	Vertices	Edges	TC Size
tree-11	tree	71,391	71,390	876,392
tree-17	tree	13,766,856	13,766,855	251,744,564
grid-150	grid	22,801	45,300	131,698,576
grid-250	grid	63,001	125,500	1,000,203,876
sf-100K	scale-free	100,002	350,604	96,157,950
gnp-0.001	$G(n,p)$	10,000	100,185	100,000,000
gnp-0.01	$G(n,p)$	10,000	999,720	100,000,000
gnp-0.1	$G(n,p)$	10,000	9,999,550	100,000,000
gnp-0.5	$G(n,p)$	10,000	49,986,806	100,000,000
patent	real-world	3,774,769	16,518,948	5,833,193,395
wiki	real-world	1,675,063	2,505,046	8,643,588,110
road	real-world	3,598,623	8,778,114	7,719,381,925
stanford	real-world	281,904	2,312,497	40,044,147,167

a positive integer d, we randomly generate a tree of depth d such that the out degree of a non-leaf vertex is a random number between 2 to 6. We use tree-d to denote such a tree of depth d. It is a sparse graph with at most $6n$ edges ($d = \Theta(\log n)$). The number of tuples in the TC is

$$\begin{aligned} & \sum_{i=0}^{d} |\text{descendent vertices of all level-}d\text{ vertices}| \\ \leq & \sum_{i=0}^{d} |\text{vertices in the tree}| \\ = & (d+1)n \\ = & \Theta(n \log n), \end{aligned}$$

i.e., the number of tuples in the TC is $O(n \log n)$. In this case, we have a sparse directed graph whose TC is also sparse. tree-11 and tree-17 are two trees used in the experiment.

2) Grid. We use grid-d to denote a $(d+1) \times (d+1)$ square grid of $(d+1)^2$ vertices. The vertices can be labeled as $(0,0), (0,1), \ldots, (0,d)$, $(1,0), \ldots, (d,d)$. For each vertex (x,y) ($x \neq d, y \neq d$), there is a directed edge from it to vertex $(x+1,y)$ and vertex $(x,y+1)$, respectively. Each vertex (d,y) ($y \neq d$) is connected to vertex $(d,y+1)$, and each vertex (x,d) ($x \neq d$) is connected to vertex $(x+1,d)$. The out degree of vertex (d,d) is zero. Let $n = (d+1)^2$, then a square grid of n vertices has $2d(d+1) = \Theta(n)$ edges. The number of tuples in the TC is

$$
\begin{aligned}
& \sum_{x=0}^{d}\sum_{y=0}^{d} |\text{vertices reachable from } (x,y)| \\
= & \sum_{x=0}^{d}\sum_{y=0}^{d} (d+1-x)(d+1-y) \\
= & \left[\frac{(d+1)(d+2)}{2}\right]^2 \\
= & \Theta(n^2).
\end{aligned}
$$

A square grid is a sparse directed graph but whose TC is dense. grid-150 and grid-250 are two grids used in the experiment.

3) Scale-free. The degree distribution of a scale-free graph follows a power law distribution, i.e., the probability of vertex whose degree is k is proportional to $k^{-\gamma}$ where γ is a parameter. Many real-world graphs, including the World Wide Web link graph, the paper citation graph and the protein-protein interaction graph, are conjectured to be scale-free graphs [CSN09]. We use the scale-free graph generator in the GraphStream library [gra] to generate a scale-free graph sf-100K. (The edges generated from the generator are undirected, but we use them as directed edges.) The TC of the graph contains about 96 million tuples. It is neither as dense as the TC of a grid ($0.25n^2$), nor as sparse as the TC of a tree ($n \log n$).

4) $G(n,p)$. An n-vertex $G(n,p)$ graph (Erdős–Rényi model) is generated by connecting vertices randomly such that each pair of vertices are connected with probability p (the graph can have self-loops). The expected number of edges is n^2p. We use gnp-p to denote such a random graph of 10000 vertices with parameter p. gnp-0.001, gnp-0.01, gnp-0.1 and gnp-0.5 are four $G(n,p)$ graphs used in the experiment. gnp-0.001 is a very sparse graph as each vertex has only ten edges on average, while gnp-0.5 is a very dense graph as every vertex is directly connected to half of all vertices. The TC of each $G(n,p)$ graph has 100 million edges, i.e., it is a complete graph (the densest graph).

Real-World Graphs.

1) patent is the US patent citation graph [LKF05]. Each vertex represents a patent, and each edge

represents a citation between two patents. The graph is a directed acyclic graph.

2) wiki is a subgraph of the Wikipedia knowledge graph [wik] (the graph was extracted in [YD-C14]). Each vertex in the knowledge graph 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. wiki contains 20% of edges and the related vertices from the knowledge graph.

3) road is the eastern USA road network [roa]. Each directed edge represents a road between two points in the road network. The graph has a tree structure where the root is a strongly connected component (SCC) consisting of 2141 vertices (about 0.06% of all vertices). All the paths point toward the root.

4) stanford is the Stanford Web graph [LLD09]. Each directed edge represents a hyperlink between two pages under the stanford.edu domain in 2002. The largest SCC in the graph contains about half of the vertices.

Memory Utilization of Algorithms. Table 2.3 shows the memory utilization of each algorithm on the test graphs. An X mark indicates that an algorithm is not applicable to a graph because the computation requires more memory than the machine has. The memory utilization is usually higher when an algorithm uses more threads. But adjusting the number of threads does not affect whether an algorithm is applicable to a graph in our experiments. Thus, each value in the table represents a typical memory utilization of an algorithm on a test graph (using the *optimal number of threads*, see Section 2.4.4.2). Two values are reported for each SSC algorithm—the memory utilization of computing and storing TC in memory, and the memory utilization of computing TC in memory while storing it to disk. For graphs that fit in memory but whose TC cannot fit in memory, the storing-to-disk option allows an SSC algorithm to compute TC without loosing the speed of in-memory computing. For example, the TC of stanford cannot fit in memory, we can still use the SSC algorithms in the storing-to-disk mode as shown by the last row of Table 2.3. Moreover, when the query only needs to compute some aggregates on each source vertex, an SSC algorithm can optimize the evaluation by computing the aggregates on each SSC without storing the whole TC. However, there is no such simple optimization for SEMINAIVE and SMART.

Table 2.3: Memory utilization of algorithms on test graphs (unit GB).

	SEMINAIVE	SMART	SSC1	SSC2	SSC12
tree-11	0.06	0.08	0.02/0.02	0.04/0.04	0.02/0.02
tree-17	6.83	8.77	5.57/3.70	8.79/6.91	5.68/3.80
grid-150	3.75	22.39	1.00/0.02	0.99/0.01	1.00/0.03
sf-100K	2.51	21.96	0.76/0.05	0.77/0.06	0.79/0.08
gnp-0.001	10.95	X	0.76/0.02	0.77/0.01	0.79/0.03
gnp-0.01	97.90	X	0.81/0.07	0.79/0.06	0.81/0.07
gnp-0.1	X	X	1.24/0.50	1.22/0.49	1.24/0.50
gnp-0.5	X	X	3.18/2.44	3.16/2.42	3.18/2.44
grid-250	X	X	7.52/0.07	7.48/0.04	7.50/0.05
patent	X	X	44.99/0.37	45.92/0.85	45.96/0.87
wiki	X	X	64.76/1.54	65.24/2.45	65.26/2.52
road	X	X	58.45/0.94	59.50/1.99	58.45/0.94
stanford	X	X	X/0.50	X/0.23	X/0.26

The algorithms can be ordered as follows based on their memory utilization: SMART > SEMINAIVE > SSC1, SSC2, SSC12. The SSC algorithms always use the least memory which is consistent with our analysis in Section 2.3.4. They are applicable to all test graphs, while SEMINAIVE and SMART are only applicable to some test graphs. Although the TCs of all test graphs can fit in memory (except stanford), SEMINAIVE and SMART are not applicable to some test graphs since the intermediate result (δtc in Figure 2.1 and Δtc$'$ in Figure 2.2) may be extremely large before deduplication. Moreover, Δtc$'$ is usually larger than the corresponding δtcs since Δtc$'$ in the i-th iteration equals the union of δtcs from iteration 2^{i-1} to $2^i - 1$. Thus, SEMINAIVE is applicable to two more graphs than SMART.

The SSC algorithms have a significant advantage over SEMINAIVE and SMART in terms of memory utilization—the memory utilization of SEMINAIVE and SMART is 1.2× to 120× of SSC12, or 1.8× to 1400× if SSC12 uses the storing-to-disk mode. In the remaining of this section, we compare these algorithms focusing on speed and scalability. We only show the result of an algorithm on a test graph if the algorithm is applicable to the test graph.

2.4.2 Tuning the Parameters of SSC12

Now we determine the values of α and β in SSC12. We reduce the search space by forcing both parameters to be powers of two. α is set to a pessimistic value: we find the α which is the

smallest power of two such that for $k = n/\alpha$ random integers in the range of $[0, n-1]$, the time of initializing array d and setting the values corresponding to the k integers to **true** is smaller than that of inserting the k integers into δ`tc`$''$. This value is pessimistic since it guarantees that the time becomes shorter if the algorithm chooses to switch. We select $\alpha = 1/8$ as it is the value found by the above procedure for $n = 10^6$.

We tune β by executing SSC12 on four synthetic test graphs, namely tree-17, grid-250, sf-100K, and gnp-0.01. We select these graphs since each graph represents a medium workload such that the execution time is neither too short nor too long. The algorithm uses 16 threads on tree-17, 64 threads on the remaining three graphs as these values are optimal in Figure 2.14. Figure 2.7 shows the execution time for different β. The execution time on tree-17 decreases as β increases, while the execution on the remaining three graphs increases as β increases. We select $\beta = 1/128$ as a trade-off between these two cases.

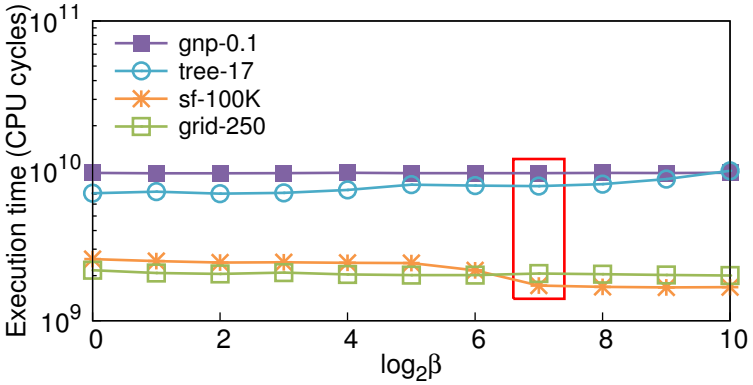

Figure 2.7: Execution time of SSC12 for different β.

The meaning of this parameter setting is: SSC12 chooses to switch if the number of tuples in the SSC is greater than $n/128$ or the number of derivations in an iteration is greater than $n/8$. The setting derived from these four graphs proved effective at delivering good SSC12 performance throughout the wide range of test graphs tested in our experiments.

2.4.3 Serial Execution Performance

We first study the serial execution performance of the compared algorithms. Figure 2.8 shows the serial execution time of each algorithm. The computation of SSC2 on tree-17 did not finish in

one day. We use a value of 10^{13} in the figure to indicate this.

Effectiveness of SSC algorithms. Overall, the SSC algorithms achieve the shortest serial execution time on all the test graphs reported in Figure 2.8: SSC2 is the fastest on grid-150, sf-100K and gnp-0.001, and SSC12 is the fastest on the remaining graphs. We say a source vertex is a type I (resp., II) vertex if SSC1 is much faster (resp., slower) than SSC2, otherwise it is a type III vertex. Since most vertices in trees (e.g., tree-11 and tree-17) are type I vertices, SSC2 performs poorly on trees, while SSC1 is much faster than SSC2 on trees. On the other hand, most vertices in grid-150 are type II vertices, thus SSC2 is much faster than SSC1. SSC2 is only slightly faster than SSC1 on the remaining graphs since most vertices are type III vertices. The hybrid algorithm, SSC12, is able to select the more efficient algorithm for every source vertex in the graph. The ability to make this intelligent selection enables SSC12 to outperform both SSC1 and SSC2 on many test graphs (e.g., tree-11 and gnp-0.01). It consistently performs well on all the test graphs.

Linear Recursion Algorithms. SEMINAIVE and the SSC algorithms are based on linear recursion. Figure 2.8 shows that SEMINAIVE is always slower than SSC1 on all the test graphs, which is consistent with the analysis in Section 2.2.3. This result shows the effectiveness of the partitioning by source vertex optimization employed by SSC1. Algorithm I.1 [AJ88] and strategy TCr [WZB92] share the same idea with SSC1. But both compute the tuples from a set of source vertices at the same time. They are expected to be slower than SSC1 for the same problem that SEMINAIVE suffers from.

Linear vs. Nonlinear Recursion. Figure 2.8 shows an empirical comparison between the nonlinear recursion based SMART algorithm and the linear recursion based algorithms. SMART is always faster than some linear recursion based algorithms on the four test graphs that it is applicable, which exhibits the advantage of smaller number of iterations during the TC computation. However, the linear recursion based SSC12 algorithm outperforms SMART on all four graphs as it uses more efficient data structures.

2.4.4 Parallel Execution Performance

Now we study the parallel execution performance of the compared algorithms on the test graphs.

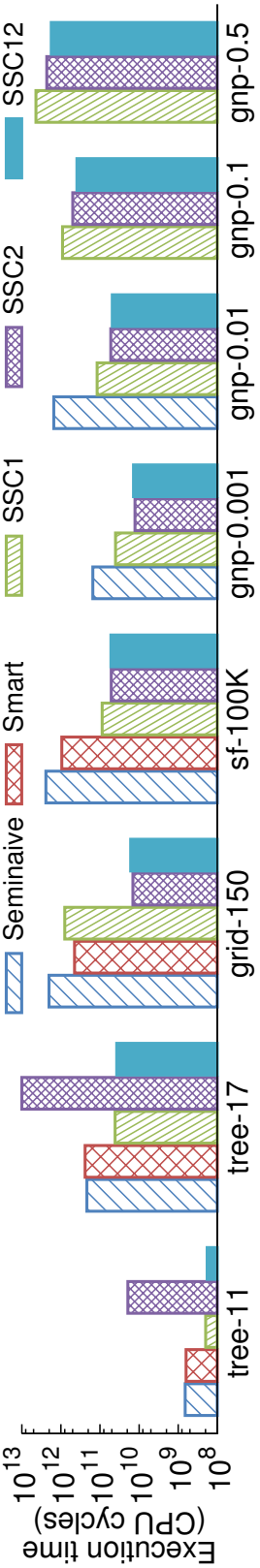

Figure 2.8: Serial execution time of algorithms.

2.4.4.1 Speedup of Algorithms

Figure 2.9 shows the speedup of each algorithm w.r.t. the serial execution. Overall, every algorithm exhibits some speedup as the number of threads increases: SEMINAIVE and SMART show very limited speedup, while the SSC algorithms achieve almost linear speedup. A general rule is that the speedup does not increase linearly with the number of threads, but it becomes progressively less due to the increased overhead of synchronizing more threads. In the extreme cases where the time for synchronization dominates real computation time, we see the overall computation taking a longer time when we increase the number of threads—see, e.g., SEMINAIVE and SMART on tree-11.

We next discuss the speedups of the various algorithms in detail. We use the result of grid-150 (i.e., Figure 2.9(c)) as an example since the same trend is also observed in Figures 2.9(d), 2.9(e), and 2.9(f).

Speedup of SEMINAIVE. The execution time of SEMINAIVE is the sum of computing time, control time, and copying time (defined in Section 2.3.2). Figure 2.10 shows the execution time breakdown of SEMINAIVE on grid-150. The bars for copying time are unnoticeable since copying time accounts for less than 1% of the execution time. As the number of threads increases from 1 to 64, control time remains almost unchanged, while computing time decreases by about two thirds. (Note that computing time increases slightly when the number of threads increases from 16 to 64 as a result of increased overhead of synchronizing more threads.) Control time accounts for more than half of the execution time when 64 threads are used. Although there is speedup in computing time, the unchanged control time limits the overall speedup in SEMINAIVE. This result is very common for graphs that require many iterations while the computation in each iteration is very fast. However, for graphs like gnp-0.001 and gnp-0.01 that SEMINAIVE terminates in a few iterations (8 and 4, respectively) while the computation in each iteration takes a long time, the execution time is still dominated by computing time, and the speedup curve of SEMINAIVE is similar to that of SMART. As we will see next, the speedup of this kind of evaluation is bound by the memory bandwidth.

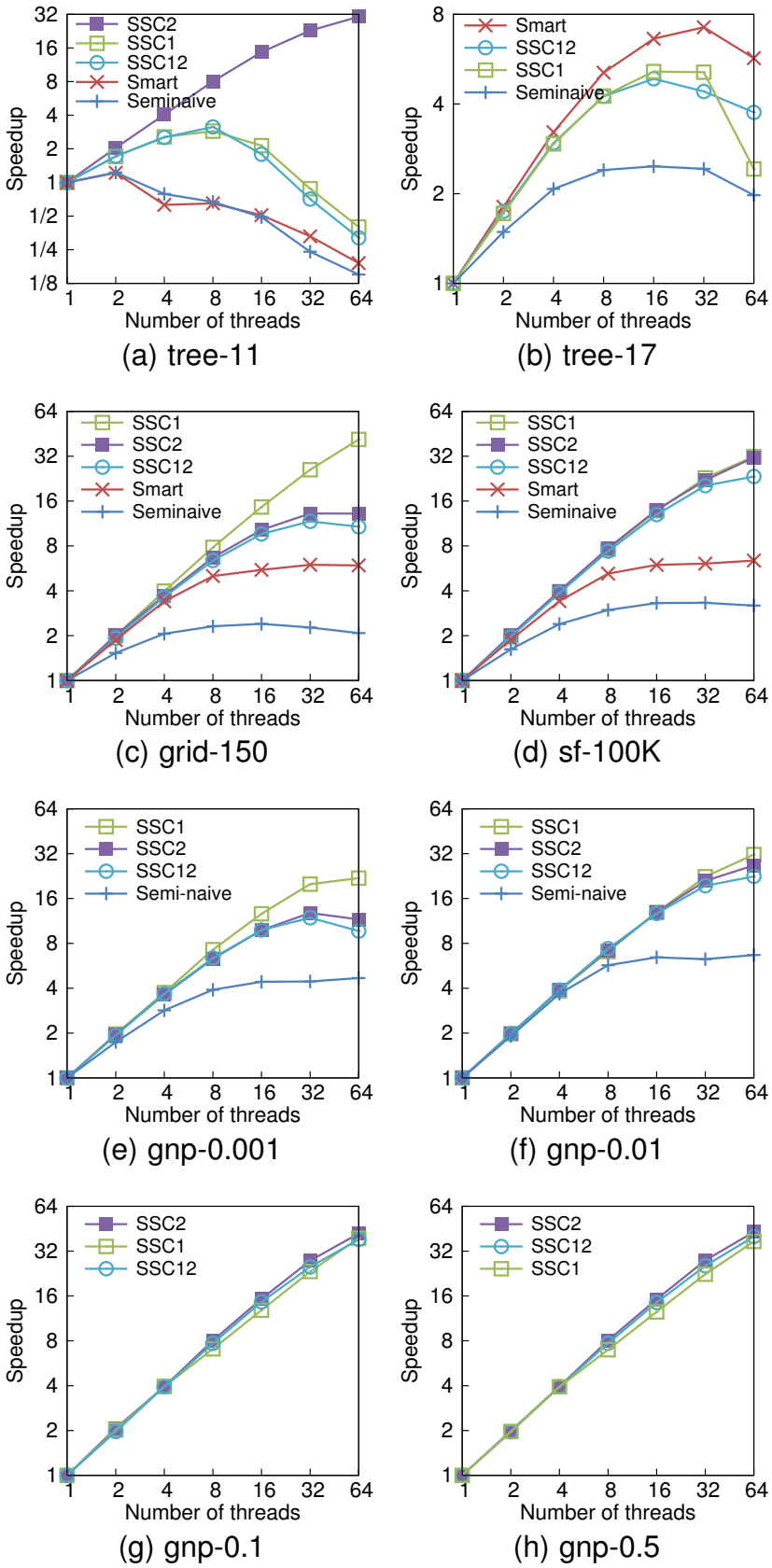

Figure 2.9: Speedup of algorithms (w.r.t. the serial execution) using different number of threads.

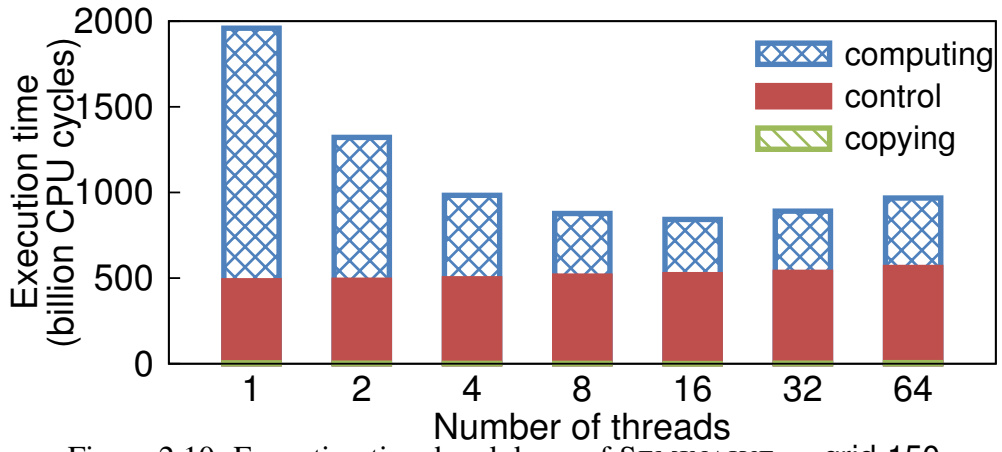

Figure 2.10: Execution time breakdown of SEMINAIVE on grid-150.

Speedup of SMART. Figure 2.11 shows the execution time breakdown of SMART on grid-150. Computing time in SMART scales much better than that in SEMINAIVE, while control time accounts for a smaller percentage in SMART. Thus, SMART scales better than SEMINAIVE on grid-150. However, the speedup of computing time is only 8 when 64 threads are used. We further investigated the time spent on each relational algebra operators in computing time. There are four operators in each iteration, namely join, union, join, and set difference. Figure 2.12 shows the total time spent on each operator. The time spent on the union operator dominates the computing time. A further breakdown of the time spent on the union operator reveals that the overall speedup is bound by the speedup of partition phase. A union operator works in two phases—partition phase and union phase. Both input relations are partitioned into smaller relations in the partition phase. Each thread computes the union of two partitioned relations in the union phase. Figure 2.13 shows the maximal speedup of each phase in each iteration. The speedup of union phase is twice as much as that of partition phase from iteration 5 to 8, while the time spent on these 4 iterations accounts for 97% of the execution time on the union operator. The speedup of partition phase is limited by the memory bandwidth [KKL09]. Thus, the overall speedup of SMART is bound by the memory bandwidth.

Speedup of SSC algorithms. An SSC algorithm is expected to achieve linear speedup since the computation in one thread does not interfere with the computation in other threads. However, its speedup stays below eight as the number of threads increases from 8 to 64 when the NUMA aware

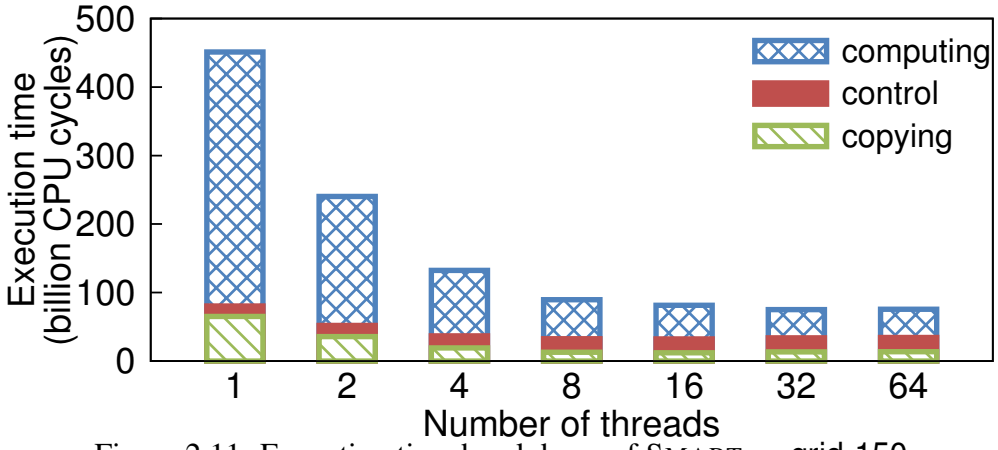

Figure 2.11: Execution time breakdown of SMART on grid-150.

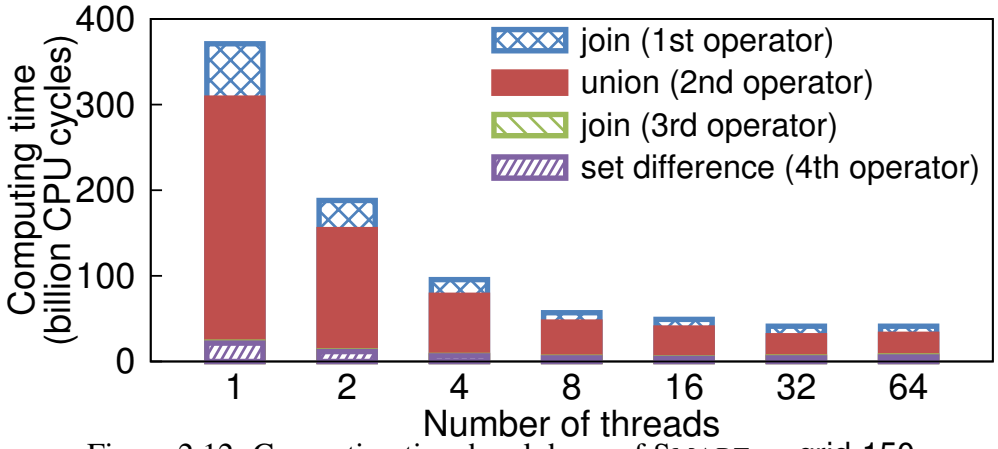

Figure 2.12: Computing time breakdown of SMART on grid-150.

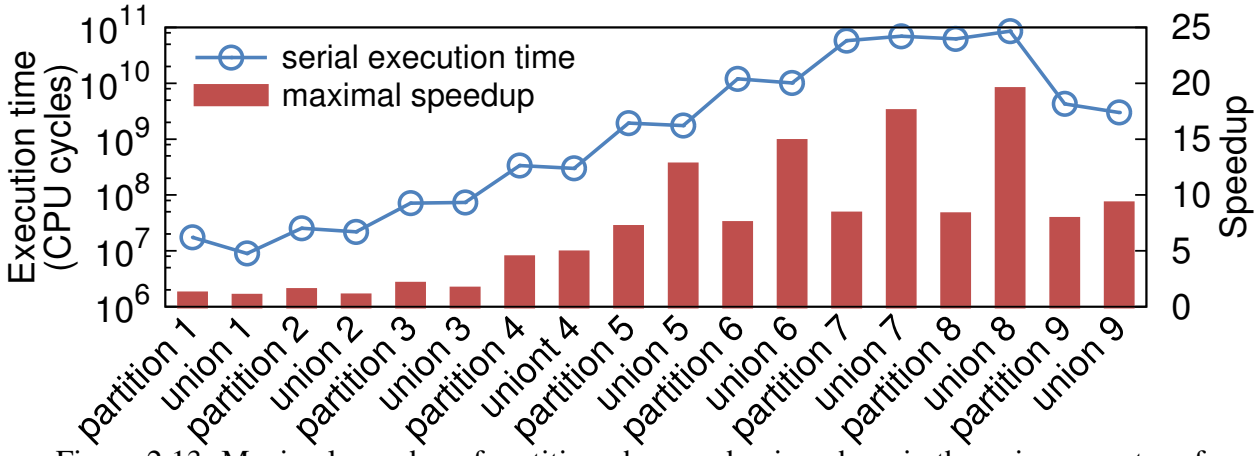

Figure 2.13: Maximal speedup of partition phase and union phase in the union operator of SMART on grid-150. A label "partition i" (resp., "union i") on the x-axis shows the maximal speedup of partition (resp., union) phase in i-th iteration.

optimization described in Section 2.3.3 is not enabled[4]. The main reason for this behavior is due to the limited memory bandwidth. The NUMA region that the shared relation is resident on is the hotspot since all threads request data from it concurrently. When the memory is saturated (i.e., it cannot handle more memory load requests in a time unit), increasing the number of threads does not increase the speedup.

The memory saturation problem is alleviated by the NUMA aware optimization. This optimization 1) distributes the memory load requests to each NUMA region and 2) ensures each thread always accesses data on the local memory. The SSC algorithms exhibit linear speedup as shown in Figure 2.9. But this optimization does not change the fact that the SSC algorithms are memory bandwidth bound. The evaluation of the all-pairs shortest path query described in Section 5 of [YZ14b] accesses more data than that of the TC query does. None of the SSC algorithms scale linearly on the graph that does not fit in the L3 cache as a result of memory saturation.

2.4.4.2 Minimal Execution Time

Finally, we compare all the algorithms in terms of their execution time using multiple threads. The execution time does not always decrease as the number of threads increases. The *optimal number of threads* is the number of threads such that the execution time of an algorithm on a graph is minimal among all other numbers. Figure 2.14 shows the execution time of each algorithm using the optimal number of threads.

SMART is faster than SEMINAIVE on all four graphs that both algorithms are applicable as a result of better speedup, which is consistent with the conclusion of [ABC11] that SMART has advantages over SEMINAIVE on TC computation. However, neither algorithms is the fastest on any test graph. The SSC algorithms achieve the minimal execution time on all the test graphs, while only SSC12 consistently performs well.

Figure 2.14 also shows the minimal execution time of the SSC algorithms on grid-250 and four real-world graphs. These graphs have much larger TCs such that only the SSC algorithms are applicable. SSC1 is faster than SSC2 on patent and road since both graphs have tree struc-

[4]We are referring to the case that the graph does not fit in the L3 cache, otherwise the algorithms scale better.

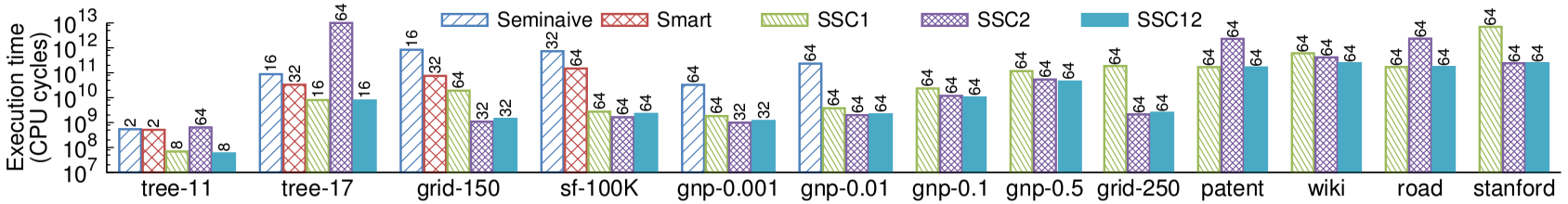

Figure 2.14: Execution time for optimal number of threads.

tures. SSC2 is faster than SSC1 on the remaining three graphs. Nevertheless, SSC12 is the only algorithm that performs well on all five graphs.

Besides the speed, SSC12 is more memory efficient than SEMINAIVE and SMART. These two advantages make SSC12 an ideal choice for main memory parallel TC evaluation.

2.4.5 Comparison with Alternative Implementations

The previous experimental result demonstrates the effectiveness of SSC12 comparing to our implementation of the semi-naive evaluation and the Smart algorithm. There are other choices on the parallel implementation of these algorithms. In this section, we compare our result with those of the alternative implementations. The comparison is organized into three categories as proposed in [CCH93].

Data Parallel. This approach assigns different operators to different threads. SEMINAIVE and SMART adopt this approach. The speedup of SEMINAIVE is limited by the control time. A data parallel implementation of the semi-naive evaluation spends a significant percentage of the execution time on the control time due to the large number of iterations required by the semi-naive evaluation. It only achieves limited speedup as the control time does not decrease when more threads are used. The speedup of SMART is limited by the memory bandwidth since the execution time is dominated by the radix partition. An alternative choice is to implement the algorithm using sort-based operators. Albutiu et al. [AKN12] reported a 4× speedup on their sort-merge join implementation compared with a radix join implementation on datasets of sizes ranging from 50 GB to 400 GB on a 32-core machine. But SSC12 is still the best algorithm even if the Smart algorithm becomes four times faster in our experiment.

Operator Pipeline. This approach assigns different data to different threads. It might achieve good speedup if we can execute many operators at the same time. Both the semi-naive evaluation and the Smart algorithm contain a set difference operator which has `tc` on the right-hand side. The computation for the set difference cannot start until the computation for `tc` is finished. Since `tc` changes in every iteration, both algorithms have to synchronize before the set difference in every iteration. Thus, we can execute at most two (resp., three) operators at the same time in an operator

pipeline implementation of the semi-naive evaluation (resp., the Smart algorithm). Although the monotonic nature of the TC computation guarantees that the computation still terminates with correct result if we proceed asynchronously without synchronizing before the set difference, the implemented algorithm is different from the semi-naive evaluation described in Figure 2.1 and the Smart algorithm described in Figure 2.2, and may perform redundant derivations. We limit our comparison to the faithful implementations of Figure 2.1 and Figure 2.2. Their speedup is no better than that of the data parallel approach.

Combining Approach. A semi-naive evaluation implementation using this approach still suffers from the same problem as SEMINAIVE does since the evaluation requires many iterations. A Smart algorithm implementation that belongs to this category is the one proposed by Afrati et al. [ABC11]. Although it is designed for a distributed environment, it can be adapted to a shared memory machine. The computation involves two kinds of tasks: *join tasks* and *dup-elim tasks*. Given a hash function, each task is responsible for the computation of all tuples with the same hash value. A join task receives a tuple from a dup-elim task and sends the derived tuples to the corresponding dup-elim tasks, while a dup-elim task receives a tuple from a join task and sends the tuple to the corresponding join task if the tuple is not in its local store. If the computation uses p threads, then both $\Delta\texttt{tc}$ and $\delta\texttt{tc}$ are supported by $p/4$ tasks of each kind, while each task is a thread executed on a unique core.

Figure 2.15 compares this implementation, denoted by SMARTA, with SMART and SSC12. SMARTA performs duplicate elimination when a tuple is derived, while SMART waits until all tuples in an iteration are derived. This duplicate elimination strategy reduces the memory requirement of SMARTA on graphs that the TC computation derives a lot of duplicate tuples. SMARTA is applicable to two more graphs than SMART. A disadvantage of this duplicate elimination strategy is its poor cache behaviour. Duplicate elimination in SMARTA is implemented as a hash table lookup. Performing many hash table lookups is expected to be slower than performing a radix partitioning based set difference due to the cache issues discussed in Section 2.3.2. SMART is faster than SMARTA on three graphs among the four graphs that both algorithms are applicable. Besides the cache issues, SMARTA needs special treatment to deal with skewed data, and it cannot fully utilize

all cores due to the load imbalance between different kind of tasks. SSC12 is between 7× to 20000× faster than SMARTA.

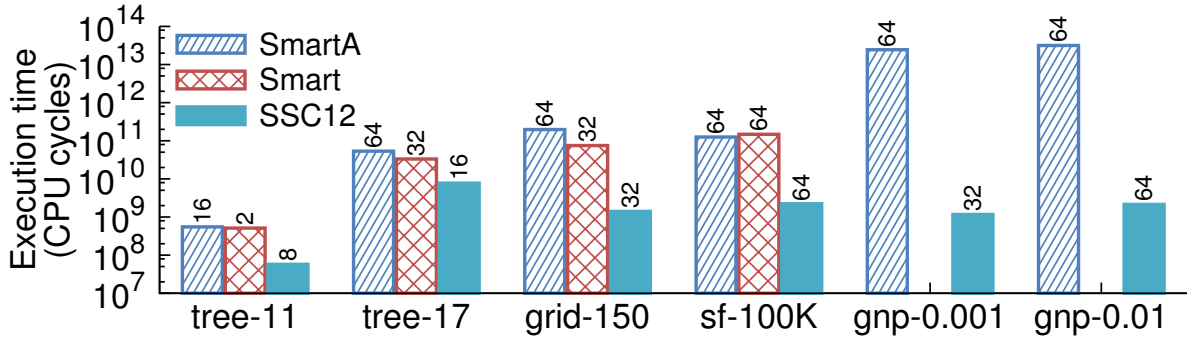

Figure 2.15: Execution time comparison with implementation of Smart algorithm proposed in [ABC11] (denoted by SMARTA).

2.5 Related Work

The TC of a binary relation is a much-studied recursive query. The earliest work dates back to 1962, when Warshall [War62] proposed the Floyd–Warshall algorithm that computes the TC of an n-vertex graph in $\Theta(n^3)$ time. One line of research tries to speed up the computation by exploiting the special property of the problem itself. Warren and Henry [War75] proposed a variant that works faster for sparse graphs in a paging environment. Agrawal and Jagadish [AJ87] studied I/O efficient variants, the Blocked Warshall algorithm and the Blocked Warren algorithm, under the assumption that the memory size is small compared to the result relation size. The I/O cost is further reduced in algorithms based on depth-first search and a marking optimization [IR88, AJ90]. [KIC92] and [DR94] compared I/O costs of TC algorithms using different implementations. Our study compares the serial execution performance of several TC algorithms. But we focus on main memory evaluation, and our implementations use cache conscious algorithms.

Our implementations of parallel TC algorithms are inspired by previous studies on parallel TC computation [VK88, AJ88, WZB92, CCH93]. The idea of implementing SEMINAIVE and SMART using hash-based parallel relational algebra operators is attributed to Valduriez and Khoshafian [VK88]. Agrawal and Jagadish [AJ88] and Wolfson et al. [WZB92] proposed to partition the computation by the source vertices so that each core applies SEMINAIVE on a set of source vertices.

The idea is similar to that of SSC1 except SSC1 applies SEMINAIVE on one source vertex one at the time. Cacace et al. [CCH93] provided a survey on parallel TC algorithms. Previous studies use theoretical models to analyse the performance of algorithms, whereas our study focuses on experimental evaluation. In another experimental study [YZ14a] that includes the parallel Floyd algorithm [AJ88], we showed that the Floyd algorithm achieves competitive performance for small dense graphs. But its memory requirement is impractical for large sparse graphs. Moreover, it is outperformed by SSC12 in the experiments.

2.6 Conclusion

In this chapter, we compared several recursive query evaluation algorithms on a modern multicore machine. A clear conclusion emerging from these experiments is that, for multicore machines, the simple SSC algorithms perform better than other algorithms in terms of speed and significantly better in terms of memory utilization. We thus introduced an algorithm, called SSC12, which combines the strengths of SSC1 and SSC2. However, our experiments also confirmed that performance of SSC12 (and other algorithms) on multicore machines will always be limited by the memory bandwidth bottleneck.

CHAPTER 3

Scaling up Datalog on Multicore Machines

In this chapter, we generalize the techniques described in the previous chapter that focus on transitive closure, and introduce query evaluation techniques for parallelizing arbitrary Datalog programs on multicore machines.

3.1 Introduction

With the first wave of interest in Datalog, much attention was paid on parallelizing its bottom-up evaluation. Various studies were proposed focusing on the message passing model, in which processors communicate with each other by exchanging messages. This includes both strategies for programs that can be evaluated without any communication [WS88, Wol88, CW89, SL91] and strategies to minimize the amount of communication required [GST92, ZWC95, GST95]. Although few system implementations and experimental results were reported on these techniques, these papers contained a number of important ideas which we have refined and extended in the implementation of our **De**ductive **A**pplication **L**anguage (DeAL) system [dea] (hereafter, we use DeALS as an abbreviation for DeAL system).

In this chapter, instead of the message passing model, we assume the shared-memory model, where the data is stored in shared memory that can be directly accessed by all processors, as supported by most modern multicore machines. We present an extensive performance comparison between DeALS and other Datalog systems that adopt the shared-memory model, including CLINGO [GKK14], DLV [LPF06], LogicBlox [ACG15], and SociaLite [SGL13, SPS13]. Figure 3.1

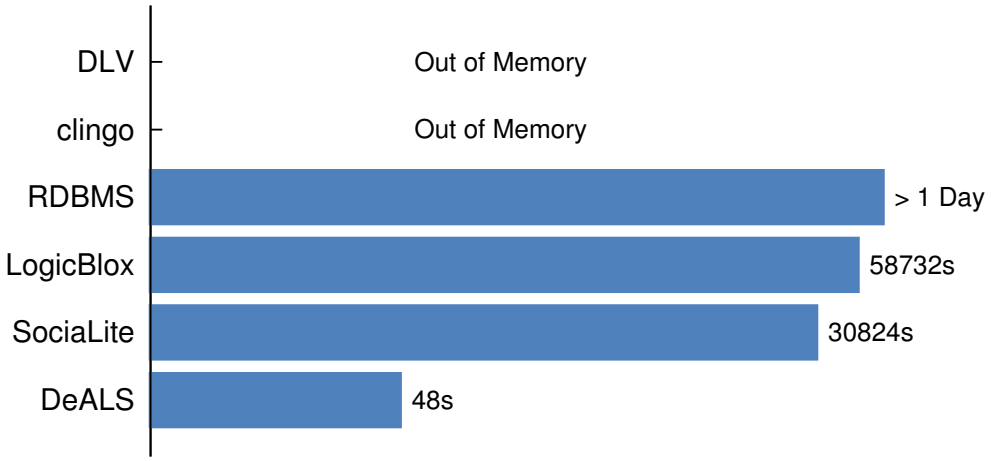

Figure 3.1: Query evaluation time of sg on tree-11.

is an excerpt from the results of Exp III in Section 3.6, showing the evaluation time of the same generation query sg (see Example 3.7) on the small test graph tree-11 for the aforementioned systems. The test graph contains 71 thousand pairs of integers, while the output contains two billion pairs of integers. The evaluation was run on a machine with four AMD Opteron 6376 CPUs (64 cores) and 256 GB memory, and the systems were configured to use all the cores and memory on the machine. Although the total size of input and output is only 1/16 of the memory size in the test machine, both CLINGO and DLV run out of memory during the evaluation. The two systems, LogicBlox and SociaLite, which can finish the evaluation, spend more than 16 hours and 8 hours, respectively. This query can also be expressed by an SQL query with a WHILE loop, which can be evaluated by RDBMSs. We implemented and tested the query on a commercial RDBMS with its memory-optimized database engine, but the evaluation on tree-11 did not finish in one day, and the similar problem occurs in other datasets we tested.[1] These disappointing results, summarized in Figure 3.1, highlight the difficulty of efficiently evaluating recursive Datalog programs on multicore machines. On the other hand, DeALS evaluates sg on tree-11 in 48 seconds—more than 600 times faster than the best evaluation time achieved by other existing systems.

In this chapter, we present the compilation techniques used by DeALS for in-memory parallel evaluation of Datalog programs on shared-memory multicore machines. We focus on one type of

[1] Another way to implement this query is to use the recursive common table expressions. But the approach of using a WHILE loop performs significantly better in our experiments.

parallel evaluation plans in which every table in the database is hash-partitioned and the number of partitions equals the number of available processors. In the parallel evaluation, locks are used to ensure the atomicity of each update operation if multiple update operations on the same partition of a table can occur concurrently, and multiple index lookups are required to achieve the effect of one index lookup in the sequential evaluation when the corresponding index is not partitioned by the key columns. Both the use of locks and additional index lookups introduce overhead in the parallel evaluation w.r.t. sequential evaluation. DeALS finds a parallel plan that correctly evaluates the query while minimizing such overhead. Here we also present a rewriting technique that transforms a locking program into a lock-free program. As a result, we obtain compilation techniques that produce efficient evaluation plans for both non-recursive programs and recursive programs. Therefore, DeALS delivers (i) competitive performance on the non-recursive queries of the TPC-H benchmark [tpca], compared with the state-of-the-art RDBMSs such as Vectorwise [vec] and SQL Server [sql], and (ii) superior performance on recursive programs compared with Datalog systems on multicore machines, namely CLINGO, DLV, LogicBlox, and SociaLite.

The rest of this chapter is organized as follows. We review some background material on monotonic aggregates in Section 3.2. We introduce our parallel evaluation strategy in Section 3.3. We describe how DeALS finds an efficient parallel evaluation plan in Section 3.4 and generates code for evaluation in Section 3.5. We report experimental results in Section 3.6. We present a sufficient condition for a program to be a lock-free program and a rewriting that transforms a locking program into a lock-free program in Section 3.7. Related work is discussed in Section 3.8. The chapter concludes in Section 3.9.

3.2 Monotonic Aggregates

An important contribution of DeALS is that it supports monotonic aggregates in recursion according to the formal semantics elucidated in [MSZ13a, MSZ13b], where basic Datalog techniques, including semi-naive evaluation and magic sets rewriting, were also extended and specialized for these monotonic aggregates. A simple example of the use of these aggregates is provided by the following `attend` program that is based on a similar one proposed by Ross and Sagiv in [RS92].

Example 3.1. *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 we use the "<···>" notation in the rule head, that is used in LDL [CGK90] and Coral [RSS92]. However, instead of the common* count *aggregate that produces only a final count, say K, we use the countinous aggregate* mcount *that produces $1, \ldots, K$, i.e., all the integers starting with 1 and up to K included.*

$$\texttt{cntfriends}(\texttt{Y}, \texttt{mcount}\langle\texttt{X}\rangle) \texttt{<-} \texttt{attend}(\texttt{X}), \texttt{friend}(\texttt{X}, \texttt{Y}).$$
$$\texttt{attend}(\texttt{X}) \texttt{<-} \texttt{organizer}(\texttt{X}).$$
$$\texttt{attend}(\texttt{Y}) \texttt{<-} \texttt{cntfriends}(\texttt{Y}, \texttt{N}), \texttt{N} \geq 3.$$

Unlike the standard count, mcount is monotonic w.r.t. set containment: if applied to a superset of S it produces a set of integers that is a superset of that produced when it is applied to S. Because of this monotonicity property, mcount can be freely used in recursive rules while preserving the least fixpoint semantics of Datalog. On the other hand, count is not monotonic in the lattice of set-containment and its use in recursive rules can produce programs that have no least fixpoint semantics. This problem is illustrated by the following example that recasts in our syntax the example of Ross and Sagiv [RS92].

Example 3.2.

$$\texttt{p(b)}.$$
$$\texttt{q(b)}.$$
$$\texttt{cp(count<X>)} \texttt{<-} \texttt{p(X)}.$$
$$\texttt{cq(count<Y>)} \texttt{<-} \texttt{q(Y)}.$$
$$\texttt{p(a)} \texttt{<-} \texttt{cq(1)}.$$
$$\texttt{q(a)} \texttt{<-} \texttt{cp(1)}.$$

Here we have two minimal models that are also minimal solutions of the fixpoint equations for the immediate-consequence operator of this program—the first is $\{\texttt{p(b)}, \texttt{cp(1)}, \texttt{q(a)}, \texttt{q(b)}, \texttt{cq(2)}\}$*, and the other is* $\{\texttt{p(a)}, \texttt{p(b)}, \texttt{cp(2)}, \texttt{q(b)}, \texttt{cq(1)}\}$*. If instead of* count *we use* mcount *in our program above, we only get the following solution:* $\{\texttt{p(a)}, \texttt{p(b)}, \texttt{cp(1)}, \texttt{cp(2)}, \texttt{q(a)}, \texttt{q(b)}, \texttt{cq(1)}, \texttt{cq(2)}\}$.

This is the correct solution since the monotonic mcount `cp(2)` *also implies* `cp(1)`*, and* `cq(2)` *implies* `cq(1)`.

The same problem also prevents the use of the standard sum in recursion. However, it seems that both [SGL13] and [WBH15] ignore this problem, and allow unrestricted use of count or sum in recursion, e.g., the program in Figure 5 of [SGL13] and the program in Figure 2 of [WBH15]. Although the iterated fixpoint computation produces the correct results for these programs, this is not true for other programs. Therefore, DeALS provides both the standard count and sum, and the monotonic mcount and msum, but only allows the use of mcount and msum in recursive rules. This avoids the semantic problems elucidated in [RS92, Van93], which beset systems such as [SGL13, WBH15] which only support the standard count and sum.

At the same time, however, the DeALS compiler recognizes that in the semi-naive evaluation, mcount (resp., msum) can often be replaced by the more efficient count (resp., sum) in programs that satisfy the conditions discussed in [MSZ13b, SYZ15].[2] This optimization is applicable in most cases of particular interest, including the `attend` example above and other examples used in previous papers on DeALS [MSZ13a, MSZ13b, SZZ13, SYZ15, SYI16], SociaLite [SGL13, SPS13], and Myria [WBH15] that use aggregate in recursion. Therefore, starting from a rigorous abstract semantics, DeALS is able to optimize execution into a concrete semantics that assures superior performance by applying this and other query optimization techniques discussed in the chapter.

3.3 Parallel Datalog Evaluation

In this section, we present a parallelization strategy for Datalog programs on a shared-memory machine with n processors and how it is implemented in DeALS.

[2]The idea of this optimization is that there is no need to consider any value but the maximum value produced by the mcount goals, i.e., the current count values, when certain conditions are satisfied. DeALS uses simple sufficient conditions that can be easily checked by a compiler, including (i) the values produced by the mcount goals are tested against some monotonic Boolean conditions which evaluate to true iff they are true for the max values; or (ii) the values produced by the mcount term are fed to the final extraction rule which disregards all the values but the max ones. Similar conditions apply for msum.

3.3.1 Parallel Bottom-Up Evaluation

Let us recall the transitive closure program and its semi-naive evaluation described in Section 2.2.

Example 3.3. *Bottom-up evaluation of* tc*. Let* arc *be a base relation that represents the edges of a directed graph. The* tc *program below computes the transitive closure of* arc*, which is a derived relation that contains all the pairs* $(\texttt{X}, \texttt{Y})$ *where there is a path from* X *to* Y *in the graph.*

$$r1.\ \texttt{tc}(\texttt{X}, \texttt{Y}) \texttt{<-}\ \texttt{arc}(\texttt{X}, \texttt{Y}).$$
$$r2.\ \texttt{tc}(\texttt{X}, \texttt{Y}) \texttt{<-}\ \texttt{tc}(\texttt{X}, \texttt{Z}), \texttt{arc}(\texttt{Z}, \texttt{Y}).$$

The bottom-up evaluation of this program works as follows. The exit rule $r1$ *is evaluated first. A tuple* $(\texttt{X}, \texttt{Y})$ *is added to* tc *for each tuple* $(\texttt{X}, \texttt{Y})$ *in* arc*. Then the left-linear recursive rule* $r2$ *is evaluated. For each new tuple* $(\texttt{X}, \texttt{Z})$ *in* tc *derived in the previous iteration, a tuple* $(\texttt{X}, \texttt{Y})$ *is added to* tc *for each tuple of the form* $(\texttt{Z}, \texttt{Y})$ *in* arc*. Rule* $r2$ *is repeatedly evaluated until* tc *does not change between two successive evaluations of the rule.*

In the parallel bottom-up 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. Each partition has its own storage for tuples, unique index, and secondary indexes. Assuming that there are one coordinator and n workers, the following Example 3.4 shows a plan for the parallel evaluation of the tc program.

Example 3.4. *Parallel bottom-up evaluation of* tc*. 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* $\texttt{tc}^i$*.*[3] *The parallel evaluation proceeds as follows.*

1. *The* i*-th worker evaluates the exit rule by adding a tuple* $(\texttt{X}, \texttt{Y})$ *to* tc *for each tuple* $(\texttt{X}, \texttt{Y})$ *in* $\texttt{arc}^i$*.*

[3]There are other possible partitioning strategies, and the choice will be discussed later in the section.

2. *Once all workers finish Step (1), the coordinator notifies each worker to start Step (3).*

3. *For each new tuple* $(\texttt{X}, \texttt{Z})$ *in* $\texttt{tc}^i$ *derived in the previous iteration, the* i*-th worker looks for tuples of 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 evaluation terminates; otherwise, the evaluation 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 the above example, the i-th worker only writes to $\texttt{tc}^i$ in Step (1), and it only reads from and writes to $\texttt{tc}^i$ in Step (3). Thus, $\texttt{tc}^i$ is only accessed by the i-th worker. This property does not always hold in every evaluation plan of `tc`. For example, 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 multiple write operations to $\texttt{tc}^i$ can occur concurrently; in this plan, we use a lock to ensure only one write operation to $\texttt{tc}^i$ is allowed at a time—a worker needs to acquire the lock before it writes to $\texttt{tc}^i$, and it releases the lock once the write operation completes.

In general, we use a lock to control the access to a partition if multiple read/write operations can occur concurrently. There are two types of locks: (i) an *exclusive lock* (x-lock) that allows only one operation at a time; and (ii) a *readers–writer lock* (rw-lock) that a) allows only one write operation at a time, b) allows concurrent read operations when no write operation is being performed, and c) disallows any read operation when a write operation is being performed. We use (i) an x-lock if there is no read operation and only multiple write operations can occur concurrently; (ii) an rw-lock if multiple read and write operations can occur concurrently since it allows for more parallelism than an x-lock. A plan that requires locks is called a *locking plan*, while a plan like the one in Example 3.4 where no lock is needed is called a *lock-free plan*. A program which has a lock-free plan is called a *lock-free program*. Conversely, a program which does not admit any lock-free plan is called a *locking program*.

A key factor that enables a lock-free plan 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, we focus on the

case where every relation is partitioned using the same hash function h defined as

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

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. We prefer a commutative function to a non-commutative function as a commutative function allows for more parallelism. It is easy to prove that if a program has a lock-free plan using a non-commutative function, the program has the same lock-free plan using an arbitrary commutative function. However, the inverse of the statement is not true, and this is illustrated by the following example.

Example 3.5. *Consider the following program where* `p` *is partitioned by its first and second columns and* `q` *is partitioned by its first column.*

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

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 non-commutative function, such as concatenation ($||$) where $h(x_1, \ldots, x_t)$ becomes $g(x_1)|| \cdots ||g(x_t)$, the same plan is not lock-free.*

Another key factor is how each relation is partitioned. We specify this for a plan using *discriminating sets*. 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. For each predicate p that corresponds to a base relation or a derived relation, let R be the relation that stores all tuples corresponding to facts about p in memory; we select a discriminating set of R that specifies the partitioning of R used in the evaluation of p. The collection of all the selected discriminating sets uniquely determines how each relation is partitioned. 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.

Example 3.6. *Discriminating sets for the plan in Example 3.4. The discriminating sets for the two occurrences of* `arc` *are both* $\{1\}$. `tc` *is a derived relation, and its discriminating set is* $\{1\}$.

3.3.2 Parallel Evaluation of AND/OR Trees

The internal representation used by DeALS to represent a Datalog program is an AND/OR tree [AOT03]. 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 (resp., AND node) are AND nodes (resp., OR nodes). Each node has a `getTuple` method that calls the `getTuple` methods of its children. Each successful invocation to the method instantiates the variables of one child (resp., all the children) and the parent itself for an OR node (resp., AND node). The program is evaluated by repeatedly applying the `getTuple` method upon its root until it fails. Thus, for an OR node, the execution (i) exhausts the tuples from the first child; (ii) continues to the next child; and (iii) fails when the last child fails. An OR node is an *R-node* if it reads from a base or derived relation with its `getTuple` method, while an OR node is a *W-node* if it writes to a derived relation with its `getTuple` method. An OR node can be both an R-node and a W-node at the same time, i.e., it corresponds to a materialized intermediate relation. The execution of an AND node corresponds to a nested loop, where, for each tuple of its first child, the `getTuple` method of its second child is called, and continues to the next child until it reaches the last child. When the `getTuple` method of a child N fails, the execution backtracks to the previous child $N1$ to invoke its `getTuple` method, and it (i) further backtracks to the child before $N1$ if the invocation fails; or (ii) resets the internal status of N and continues the execution on N.

The pipelined evaluation on an AND/OR tree only materializes relations when it is necessary, i.e., if (i) it is the root; or (ii) the relation needs to be materialized to ensure the correctness, e.g., an aggregate or when semi-naive evaluation is used to compute a recursive goal; or (iii) the cost of computing the tuples when they are needed is much higher than the cost of materializing the

relation in memory.[4] In the last case, a new stratum is added to the evaluation which computes and materializes all the tuples of the relation.

The given program is transformed into an AND/OR tree, where the root represents the query, and the children of each AND node are arranged in the same order as the predicates in the corresponding rule. The AND/OR tree is then adorned for bottom-up evaluation, which produces a *bound/free adornment* [Ull85] for each node. An adornment is a string of `b`'s and `f`'s whose length equals the arity of the corresponding predicate p, where `b` or `f` in the i-th position means the i-th argument in p is bound or free when p is evaluated. For an R-node with an adornment that contains some `b`'s, DeALS tries to build a secondary index on the accessed relation, which covers all the columns that correspond to the positions of `b`'s and facilitates the evaluation of this R-node. An OR node is an *entry node* if (i) it is a leaf, and (ii) it is the first R-node among its siblings, and (iii) none of its ancestor OR nodes has 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 3.7. *Let* $\texttt{anc}(\texttt{X}, \texttt{Y})$ *be a relation describing that* `X` *is a parent of* `Y`*. The* `sg` *program below finds all the* $(\texttt{X}, \texttt{Y})$ *pairs that are of the same generation.*

$$r1.\ \texttt{sg}(\texttt{X}, \texttt{Y}) \texttt{<-}\ \texttt{anc}(\texttt{A}, \texttt{X}), \texttt{anc}(\texttt{A}, \texttt{Y}), \texttt{X} \neq \texttt{Y}.$$
$$r2.\ \texttt{sg}(\texttt{X}, \texttt{Y}) \texttt{<-}\ \texttt{anc}(\texttt{A}, \texttt{X}), \texttt{sg}(\texttt{A}, \texttt{B}), \texttt{anc}(\texttt{B}, \texttt{Y}).$$

*The corresponding adorned AND/OR tree is shown in Figure 3.2, where (i) the text inside a node indicates its type and ID, e.g., "*OR-1*" indicates that the root is an OR node with ID 1, and (ii) the text adjacent to a node shows the corresponding predicate with its adornment. Thus,* OR-4*,* OR-5*,* OR-7*,* OR-8*, and* OR-9 *are R-nodes, and* OR-1 *is a W-node.* OR-4 *and* OR-7 *are entry nodes in this program. Although* OR-5 *is an R-node, it is not an entry node since it is not the first R-node among its siblings. Similarly for* OR-8 *and* OR-9*.*

Now we describe the pipelined evaluation of this AND/OR tree in DeALS. We use $N.\texttt{getTuple}$ *to denote the* `getTuple` *method of a node* N*. The evaluation starts with the exit rule represented by* AND-2 *and its children. For each tuple* $(\texttt{A}, \texttt{X})$ *of* `anc` *retried by* OR-4.`getTuple`*,*

[4] Currently, the user determines when to force the materialization of a relation with an annotation in the program.

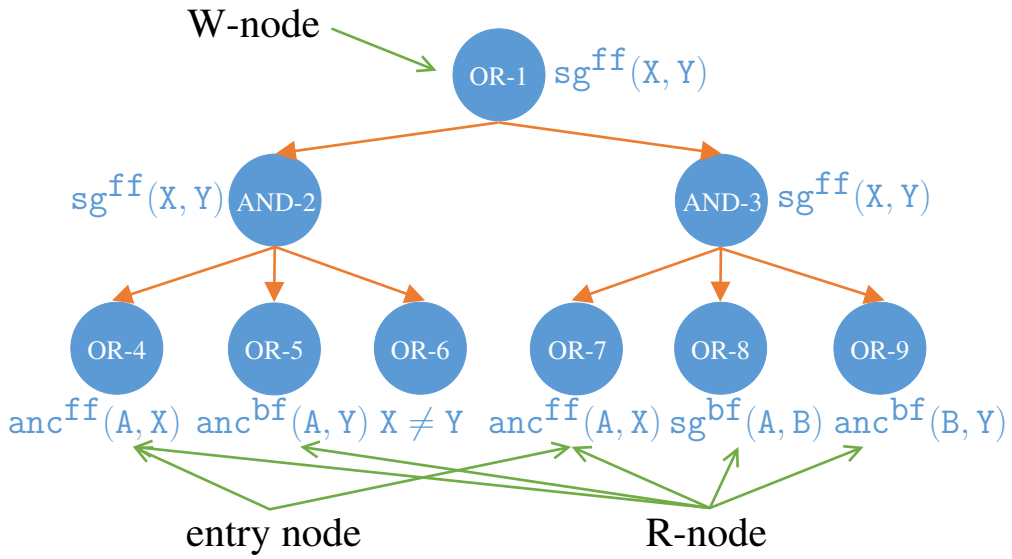

Figure 3.2: AND/OR tree of sg program in Example 3.7.

OR-5.`getTuple` *is invoked to retrieve tuples of the form* $(\mathtt{A},\mathtt{Y})$ *in* `anc` *with a bound* `A`*, which is implemented by looking up a secondary index on the first column of* `anc`*. Then* OR-6.`getTuple` *simply checks whether* $\mathtt{X} \neq \mathtt{Y}$*. If so,* AND-2.`getTuple` *also succeeds, and* OR-1 *adds the derived tuple* $(\mathtt{X},\mathtt{Y})$ *to* `sg`*. When* AND-2.`getTuple` *fails, the evaluation starts a new iteration that evaluates the recursive rule represented by* AND-3 *and its children. For each tuple* $(\mathtt{A},\mathtt{X})$ *of* `anc` *retried by* OR-7.`getTuple`*,* OR-8.`getTuple` *looks for tuples of the form* $(\mathtt{A},\mathtt{B})$ *in* `sg` *through a secondary index on the first column of* `sg` *that indexes all the tuples in* `sg` *derived in the previous iteration. Thus,* OR-9.`getTuple` *looks for tuples of the form* $(\mathtt{B},\mathtt{Y})$ *in* `anc` *through the secondary index on* `anc` *that is also used in* OR-5.`getTuple`*.* OR-1 *adds the derived tuple* $(\mathtt{X},\mathtt{Y})$ *to* `sg` *after each successful invocation to* AND-3.`getTuple`*, and checks whether a fixpoint is reached when* AND-3.`getTuple` *fails. If not, it starts the next iteration on the subtree with root* AND-3.

In the parallel evaluation of an AND/OR tree with one coordinator and n workers, we create n copies of the same AND/OR tree, and assign the i-th copy to the i-th worker. The evaluation is divided into n disjoint parts, where the i-th worker evaluates an entry node by instantiating variables with constants from the i-th partition of the corresponding relation, while it has full access to all partitions of the corresponding relations for the remaining R-nodes. The parallel evaluation ensures the same workflow as the sequential pipelined evaluation by adding synchronization barri-

ers in the nodes that represent recursion. For example, we create a synchronization barrier B, and add it to OR-1 of Figure 3.2 for every copy of the AND/OR tree. Now, the evaluation works as follows.

1. Each worker evaluates the exit rule by calling AND-2.`getTuple` until it fails. A worker waits at B after it finishes.

2. Once all n workers wait at B, the coordinator notifies each worker to start Step (3).

3. Each worker evaluates the recursive rule by calling AND-3.`getTuple` until it fails. A worker waits at B after it finishes.

4. Once all n workers wait at B, the coordinator checks if there are new tuples derived in `sg`. If so, the evaluation continues from Step (3); otherwise, the evaluation terminates.

The parallelism is achieved through the parallel evaluation of each single rule, including the parallel pipelined evaluation of all the rules which support its goals. 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 [PRS13].

3.4 Selecting a Parallel Plan

In this section, we describe how DeALS finds the best discriminating sets to evaluate a program.

3.4.1 Read/Write Analysis

For a given set of discriminating sets, the *Read/Write Analysis* (RWA) on an adorned AND/OR tree determines the actual program evaluation plan, including the type of lock needed for each derived relation, whether an OR node needs to acquire a lock before accessing the corresponding relation, and which partition of the relation an OR node needs to access when it accesses the relation through index lookups. 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 i-th worker. For each node N encountered during the traversal, the following three cases are possible:

Case (1) N is an entry node. In this case, set it as the current entry node; then, for each W-node that is an ancestor of N and is in the same stratum as N, determine whether the i-th worker only writes to the i-th partition of $R(p_w)$. This is done by checking if $p_e[\overline{X_j}] = p_w[\overline{X_k}]$,[5] where p_e and p_w are the predicates associated with N and the W-node, respectively, and $\overline{X_j}$ and $\overline{X_k}$ are the corresponding discriminating sets.

Case (2) N is an R-node that reads from a derived relation. In this case, determine whether the i-th worker only reads from the i-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 N, 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 N.

Case (3) N is an R-node that reads from a base relation through a secondary index. In this case, determine whether the i-th worker only needs to read from one partition of $R(p_r)$ instead of all the partitions by checking if $\overline{X_k} \subseteq \overline{B}$, where p_r is the predicate associated with N, $\overline{X_k}$ is the corresponding discriminating set, and $\overline{B}$ is the set of positions for bound arguments in N.

In Case (1) of the above procedure, the set of ancestor W-nodes of an entry node can be efficiently obtained by using a stack that keeps track of the nodes that are being traversed. The W-nodes in the stack are processed from the last element to the first element until a W-node that is not in the same stratum with the entry node is encountered.

Example 3.8. *RWA on the AND/OR tree in Figure 3.2. Let* $\overline{X_1}$ *be the discriminating set of* `sg`*,* $\overline{X_2}$*,* $\overline{X_3}$*,* $\overline{X_4}$*, and* $\overline{X_5}$ *be the discriminating sets of the four occurrences of* `anc`*. Figure 3.3 shows the AND/OR tree with the discriminating sets. The RWA proceeds as follows.*

1. *The depth-first traversal visits* OR-1*,* AND-2*, and* OR-4*. It sets* OR-4 *as the current entry node, and then checks whether* $\texttt{anc}(\texttt{A},\texttt{X})[\overline{X_2}] = \texttt{sg}(\texttt{X},\texttt{Y})[\overline{X_1}]$ *since* OR-1 *is a W-node and an ancestor of* OR-4*.*

[5]For a predicate p, $R(p)$ denotes the relation that stores all tuples corresponding to facts about p; $p[\overline{X}]$ denotes a tuple of arity $|\overline{X}|$ by retrieving the arguments in p whose positions belong to $\overline{X}$, and it is treated as a multiset of arguments when involved in equality checking.

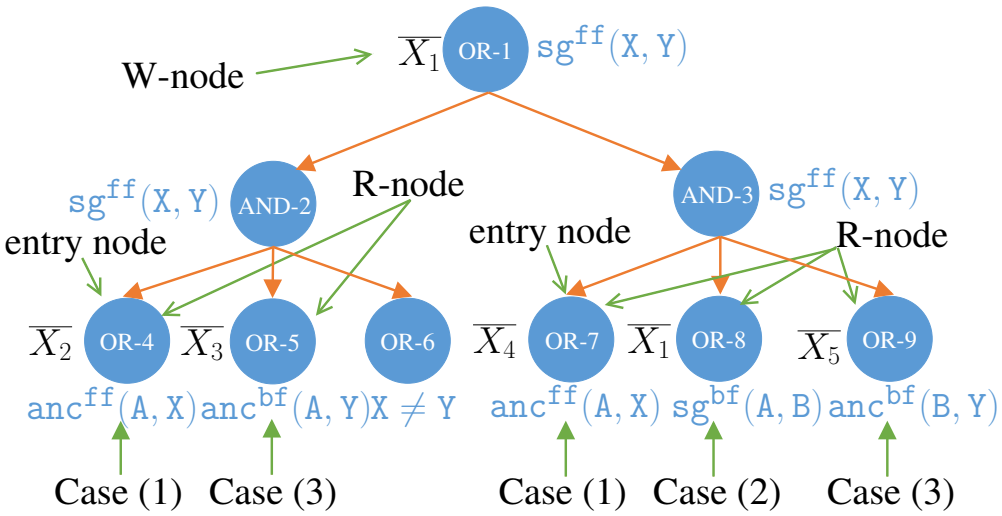

Figure 3.3: RWA on the AND/OR tree of sg program. Each leaf OR node is marked with the case it belongs to in the RWA.

2. *It visits* OR-5. OR-5 *is an R-node that reads from a base relation, and the set of positions for bound arguments in* OR-5 *is* $\{1\}$. *It checks whether* $\overline{X_3} \subseteq \{1\}$.

3. *It visits* OR-6, AND-3, *and* OR-7. *It sets* OR-7 *as the current entry node, and checks whether* $\mathtt{anc}(\mathtt{A},\mathtt{X})[\overline{X_4}] = \mathtt{sg}(\mathtt{X},\mathtt{Y})[\overline{X_1}]$.

4. *It visits* OR-8. OR-8 *is an R-node that reads from a derived relation,* OR-7 *is the current entry node, and the set of positions for bound arguments in* OR-8 *is* $\{1\}$. *Thus it checks whether* $\overline{X_1} \subseteq \{1\}$ *and* $\mathtt{anc}(\mathtt{A},\mathtt{X})[\overline{X_4}] = \mathtt{sg}(\mathtt{A},\mathtt{B})[\overline{X_1}]$.

5. *It visits* OR-9. *It checks whether* $\overline{X_5} \subseteq \{1\}$ *since* OR-9 *satisfies the condition of Case (3).*

In the ideal situation, where all the conditions are satisfied, the i-th worker only accesses the i-th partition of each derived relation, and it only accesses one partition when it performs an index lookup on a relation. As a result, the program can be evaluated in a lock-free way without performing redundant work. A condition of Case (3) can be easily satisfied by picking a non-empty subset of the set of positions for bound arguments in the R-node. But it is not always possible to satisfy all the conditions of Case (1) and Case (2) for an arbitrary program, and each unsatisfied condition means overhead in program evaluation, including the cost of locking and additional index lookups. More specifically, let R be the relation accessed by node N:

- When a condition of Case (1) is not satisfied, we check whether there are some R-nodes that read from R in the current stratum. If so, each partition of R needs an rw-lock, and N and all the R-nodes that read from R acquire a lock when they access R; if not, each partition of R needs an x-lock, and N acquires a lock when it accesses R.

- When a condition of Case (2) is not satisfied, we first check whether N accesses R through index lookups. If so, N performs n index lookups where each accesses one partition of R when $\overline{X_k} \not\subseteq \overline{B}$, while it only performs one index lookup when $\overline{X_k} \subseteq \overline{B}$. Then we check whether there is a W-node that writes to R in the current stratum. If so, each partition of R needs an rw-lock, and both the W-node and all the R-nodes that read from R acquire a lock when they access R.

Example 3.9. *Continuing Example 3.8. The resulting plan is different from the ideal plan in the following aspects when* $\overline{X_1} = \overline{X_2} = \overline{X_4} = \{2\}$, $\overline{X_3} = \overline{X_5} = \{1\}$*:*

- *Each partition of* `sg` *needs an rw-lock since the conditions in Step (1) and Step (3) of Example 3.8 are not satisfied;*

- OR-1 *needs to acquire a w-lock when adding a tuple to* `sg`*;*

- OR-8 *needs to acquire an r-lock before it accesses the index;*

- OR-8 *needs to perform* n *index lookups to achieve the same effect of one index lookup in the ideal plan since the condition* $\overline{X_1} \subseteq \{1\}$ *in Step (4) of Example 3.8 is not satisfied.*

We assume the program does not contain aggregates and arithmetic expressions in the above analysis. If the program contains these constructs, the same analysis is still applicable after 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.

3.4.2 Determining the Discriminating Sets

DeALS determines the best discriminating sets for the given program by solving an optimization problem that minimizes a *cost function*. We use the *cost of program evaluation*, which can be approximated by estimating the cost of an actual evaluation plan. There are four types of constraints in this optimization problem:

(i) $p_e[\overline{X_j}] = p_w[\overline{X_k}]$ for each entry node that belongs to Case (1) in the RWA;

(ii) $\overline{X_k} \subseteq \overline{B}$, $p_e[\overline{X_j}] = p_r[\overline{X_k}]$ for each R-node that belongs to Case (2) in the RWA;

(iii) $\overline{X_k} \subseteq \overline{B}$ for each R-node that belongs to Case (3) in the RWA;

(iv) $\emptyset \subsetneq \overline{X} \subseteq \{1, \ldots, arity(R)\}$ for each $\overline{X}$ appearing in the above three types of constraints, where $arity(R)$ is the arity of the relation R associated with $\overline{X}$.

The constraints of Type (iii) and Type (iv) are *hard constraints* that must be satisfied, while the constraints of Type (i) and Type (ii) are *soft constraints* that can be violated. The set of constraints can be obtained by performing the RWA on the AND/OR tree once.

Example 3.10. *Continuing Example 3.8. The set of constraints for the AND/OR tree in Figure 3.3 is shown below.*

$$\begin{aligned}
&\texttt{anc}(\texttt{A},\texttt{X})[\overline{X_2}] = \texttt{sg}(\texttt{X},\texttt{Y})[\overline{X_1}] \\
&\overline{X_3} \subseteq \{1\} \\
&\texttt{anc}(\texttt{A},\texttt{X})[\overline{X_4}] = \texttt{sg}(\texttt{X},\texttt{Y})[\overline{X_1}] \\
&\overline{X_1} \subseteq \{1\},\ \texttt{anc}(\texttt{A},\texttt{X})[\overline{X_4}] = \texttt{sg}(\texttt{A},\texttt{B})[\overline{X_1}] \\
&\overline{X_5} \subseteq \{1\} \\
&\emptyset \subsetneq \overline{X_i} \subseteq \{1,2\}, i = 1,2,3,4,5.
\end{aligned} \tag{3.1}$$

The best assignment that minimizes the cost function can be found by enumerating all possible assignments. It is feasible to use brute force enumeration for a program with several predicates (less than 20), e.g., transitive closure, bill of materials, and same generation. For a program with

hundreds of predicates, such as the programs in DOOP [BS09], the predicates are stratified, where each stratum typically contains less predicates. Instead of finding the best assignment for the whole program, we can find an assignment whose cost is very close to that of the best assignment by (i) performing brute force enumeration for each subprogram consisting of all the predicates in one stratum, and (ii) combining the results for every stratum. Moreover, in the case that a certain stratum contains too many predicates where the enumeration is still infeasible, local search methods designed for constraint satisfaction problems [SLM92, SKC93, Spe93, SK93] can be used to find a "good" assignment under a time constraint.

The idea of our approach is very similar to that of *generalized pivoting* where a system of equations is obtained from the rules and an exact solution is required [SL91]. But our approach is different from generalized pivoting in two aspects: (i) the set of constraints is obtained through the RWA on the AND/OR tree instead of each single rule, since the pipelined evaluation on the AND/OR tree might evaluate multiple rules at the same time, while generalized pivoting assumes only one rule is evaluated at a time; (ii) 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.

3.4.3 A Simplified Cost Function

Now we describe how the cost of an actual evaluation plan is estimated in DeALS. Note that minimizing the cost of program evaluation is equivalent to minimizing the overhead of program evaluation over the "ideal" plan in which all the constraints are satisfied.

We use `sg` as an example to demonstrate how the cost function is computed. Consider the set of constraints in Example 3.10 when the assignment is $\overline{X_1} = \overline{X_2} = \overline{X_4} = \{2\}$, $\overline{X_3} = \overline{X_5} = \{1\}$, the first, third, and fourth constraints are violated. The differences between the resulting plan and the ideal plan are described in Example 3.9. Thus, the overhead of program evaluation can be expressed as

$$
\begin{aligned}
& C_{\text{w-lock}} \times \#\text{ times OR-1 is evaluated} \\
+\ & nC_{\text{r-lock}} \times \#\text{ times OR-8 is evaluated} \\
+\ & (n-1)C_{\text{lookup}} \times \#\text{ times OR-8 is evaluated},
\end{aligned}
$$

where $C_{\text{w-lock}}$, $C_{\text{r-lock}}$, and C_{lookup} are the costs of acquiring a w-lock, acquiring an r-lock, and performing an index lookup, respectively, and we assume the cost is a constant throughout the evaluation.

It is very difficult to provide an accurate estimation of the number of times a node is evaluated without evaluating the program. In this chapter, we consider a simplified cost function in which each node is evaluated exactly once. Furthermore, we assume a unit cost for all basic operations, including acquiring a lock and performing an index lookup; we also assume that the value of n is two. For each OR node N in the AND/OR tree, its contribution to the overhead of program evaluation is denoted by $c(N)$, where

$$
c(N) = \begin{cases}
3, & \text{if } N \text{ needs to acquire an r-lock before performing an} \\
 & \text{index lookup and condition } \overline{X_k} \subseteq \overline{B} \text{ is violated;} \\
1, & \text{if } N \text{ needs to acquire a lock before accessing the relation;} \\
0, & \text{otherwise.}
\end{cases}
$$

Thus, the optimization problem becomes finding an assignment that minimizes $\sum_N c(N)$, where N iterates over the set of OR nodes in the AND/OR tree.

Example 3.11. *Continuing Example 3.10. Among all the possible assignments of the variables, the minimal value of the cost function is 2, which is obtained under the following two assignments:*

- $\overline{X_1} = \overline{X_3} = \overline{X_4} = \overline{X_5} = \{1\}$, $\overline{X_2} = \{2\}$, *which violates the third constraint;*
- $\overline{X_1} = \overline{X_3} = \overline{X_5} = \{1\}$, $\overline{X_2} = \overline{X_4} = \{2\}$, *which violates the fourth constraint.*

Both OR-1 *and* OR-8 *need rw-locks for both assignments, i.e., the program does not have lock-free plans. Since both assignments have the same cost, the assignment that appears first during the evaluation is picked by DeALS.*

3.4.4 Some Notes on the Cost Function

Besides the cost of program evaluation, the total cost also includes the cost of preprocessing the base relations. For a selection of the discriminating sets, there might be several discriminating sets of the same base relation. Thus, the base relation needs to be partitioned, and the corresponding secondary indexes needs to be built for each different discriminating set. The cost function can be adjusted to include the cost of preprocessing if we want to model the total cost of program evaluation.

So far, we have focused on finding an efficient evaluation plan for a single query. Given a workload with many queries, such as the TPC-H benchmark that contains 22 non-recursive SQL queries, the best plan to evaluate the whole workload is the one that minimizes the total cost for all queries in the workload. However, brute force enumeration becomes infeasible due to the exponential growth of the number of possible plans. We adopt a simple heuristic in DeALS that (i) finds the best plan for each query in the workload; (ii) performs a majority voting to determine the primary partitioning for each base relation, i.e., partition a base relation in a way that is preferred by majority of the queries; (iii) performs preprocessing on each base relation that enables efficient access to it for all the queries in the workload. For example, there are 17 queries in the TPC-H benchmark that access the base relation `lineitem`, and DeALS determines to partition `lineitem` by its first column, second column, and third column for ten queries, six queries, and one query, respectively. Thus, DeALS partitions `lineitem` by its first column, builds a secondary index partitioned by the second column that will be used by six queries, and builds a secondary index partitioned by the third column that will be used by one query. It remains unclear how good is our heuristic w.r.t. the optimal plan, and there are many other strategies for finding a good plan. We leave this investigation as a future work.

3.5 Code Generation

DeALS is a Datalog prototype system developed at UCLA. Its Java-based compiler first compiles a program into an AND/OR tree. Then, the parallel evaluation module targeted for shared-memory

multicore machines determines the parallel evaluation plan using the techniques presented in this chapter and generates a corresponding C++ program. We implemented the database objects (storage and index), 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, including each node's `getTuple` method. In this section, we use the `sg` program as an example to present general designs and implementation techniques that are applicable for both lock-free plans and locking plans, and the techniques for obtaining and implementing lock-free plans will be discussed in Section 3.7.

Figure 3.4 shows the code snippet that defines the tuple and relation for `sg`. Each tuple in `sg` is represented by a struct `sgTuple` that has two fields. The relation `sg` is represented by a class `sgRelation` that extends the base class `Relation` as discussed next; an instance of the class `Relation` holds `nthreads` instances of `TupleArrayStore` that store the tuples, where `nthreads` is the number of threads for program evaluation. We set the value of `nthreads` to n if there are n processors available for the evaluation. Now, besides the fields inherited from `Relation`, `sgRelation` has four additional fields:

1. `uniqueIndexes` is the unique index of `sg` implemented as n B+ trees. Each key is an unsigned 64-bit integer obtained from concatenating the values of two columns in a tuple in `sg`. The leaf nodes in the B+ trees have no values.

2. `secondaryIndexes0` is a secondary index on `sg` implemented as n B+ trees. Each key is an unsigned 32-bit integer obtained from the value of the first column in a tuple in `sg`. The leaf nodes in the B+ trees store the addresses of the actual tuples, i.e., offsets in the `TupleArrayStore`.

3. `pages` points to n instances of `Page`, where each `Page` holds a list of memory blocks of size 1 MB. The i-th instance `pages[i]` is responsible for allocating memory for nodes in `uniqueIndexes[i]` and `secondaryIndexes0[i]` from the last unfilled memory block in its list. However, when the last memory block is full, it allocates a new memory block of size 1 MB from memory, adds the new block to its list, and allocates memory from the new block.

```
struct sgTuple {
    uint32_t col0;
    uint32_t col1;
};

class sgRelation : public Relation<sgTuple> {
public:
    sgRelation(Page** pages, int nthreads);
    ~sgRelation();
    bool add(sgTuple* tuple);
    ...
private:
    Page** pages;
    BPlusTreeK<uint64_t>** uniqueIndexes;
    BPlusTreeKIndex<uint32_t>** secondaryIndexes0;
    RWLock** locks;
};
```

Figure 3.4: Snippet of generated code for sg.

4. locks points to n instances of readers–writer lock RWLock, where locks[i] is associated with the i-th partition of sg.

In the above implementation, sgRelation is divided into n partitions, where each partition has an instance of TupleArrayStore, a unique index, a secondary index, an instance of Page, and an rw-lock. When a tuple is derived during the evaluation, it is added to sg by calling method bool add(sgTuple* tuple) which works as follows:

1. It computes the value of i, which is the partition of sg that tuple belongs to.
2. It acquires locks[i] for write access.
3. It adds tuple to uniqueIndexes[i], which returns false or true depending on whether tuple is already present in uniqueIndexes[i] or not.
4. If the add operation returns false, it releases locks[i] and returns false; otherwise, it continues.
5. It adds tuple to the i-th TupleArrayStore, and then adds tuple together with its offset in the i-th TupleArrayStore to secondaryIndexes0[i].

6. It releases `locks[i]`, and returns `true`.

Thus, `uniqueIndexes` ensures that each tuple appears only once in `sg`.

The generated code also contains definitions of cursors that wrap all the read operations to `sgRelation`, where a caller calls the `getTuple` method of a cursor to get a desired tuple. For example, in the semi-naive evaluation of `sg`, the cursor used by OR-8 retrieves tuples of the form `(A, B)` from `sg` derived in the previous iteration, while the `getTuple` method works as follows:

1. It acquires `locks[i]` for read access, where $i = h(\texttt{A})$.

2. It searches for a list associated with key `A` from `secondaryIndexes0[i]`, and if the index does not contain the key, it releases `locks[i]` and returns `NULL`.

3. For each address $addr$ in the list associated with key `A`, it checks if $addr$ is within the range of tuples derived in the previous iteration, and if so, it fetches the tuple at address $addr$ from the i-th `TupleArrayStore`, releases `locks[i]`, and returns the tuple.

Each cursor also contains a `beginNextStage` method, which prepares the cursor for a new iteration of the semi-naive evaluation, and is called when the evaluation reaches an iteration boundary. For example, the cursor used by OR-8 maintains two arrays, `beginTupleAddress[i]` and `endTupleAddress[i]`, which store the addresses of the first and last tuple in the i-th `TupleArrayStore` in the previous iteration, respectively. The corresponding `beginNextStage` method sets the values for both arrays. Then, the cursor checks if a tuple at address $addr$ is derived in the previous iteration by checking if `beginTupleAddress[i]` $\leq addr \leq$ `endTupleAddress[i]`.

In summary, the `getTuple` method of a node in the AND/OR tree reads from `sgRelation` through a cursor, and writes to it by invoking the `add` method. The `sg` implementation described above is a basic implementation of the semi-naive evaluation in DeALS that is applicable in general. The cursor abstraction allows the system to optimize the execution by plugging in different optimized cursor implementations. For example, given an AND/OR tree like the one in Figure 3.2, where the semi-naive evaluation only accesses the tuples of the recursive relation that are derived

in the previous iteration by searching a secondary index, DeALS optimizes the execution by maintaining two separate secondary indexes that keep track of the tuples in the previous and current iteration, respectively. More details about this optimized implementation are discussed in Section 3.7.3.

Before moving onto the discussion on compilation, we would like to discuss two more optimizations. The first one is to replace B+ trees with hash tables as indexes. Instead of using B+ trees, hash tables are widely used in analytical workloads, as the cost of accessing a hash table is lower than that of a B+ tree that indexes the same data. However, hash tables may not be the most suitable data structures for recursive relations that are frequently updated. We choose to use B+ trees as indexes in DeALS, and our experience shows that this design choice works quite well in general. As we will show in Exp III of Section 3.6, DeALS outperforms SociaLite that uses hash tables for recursive relations. This is largely due to a better memory allocation pattern used by B+ trees. Although we prefer B+ trees to hash tables for recursive relations, it is still beneficial to use hash tables for base relations that remain unchanged throughout the evaluation. However, the improvement is marginal as the evaluation time is usually not dominated by the time spent on fetching tuples from base relations. Thus, DeALS supports both B+ trees and hash tables, and it can be configured to use B+ trees or hash tables for certain relations, while it uses B+ trees as a default setting for indexes.

Another potential important optimization is to use a worst-case optimal join algorithm called the leapfrog triejoin [Vel14]. This is studied but not yet implemented because the current evaluation model used by DeALS assumes the predicates in the body of each rule are accessed following a fixed order, while the leapfrog triejoin algorithm only requires a fixed variable order, but the predicates in the body of a rule might be accessed without a fixed order. For a given program, if the program is always safe and does not trigger any magic sets rewriting for any reordering of the predicates in the body of each rule, the leapfrog triejoin algorithm can be simply plugged into the evaluation, and the techniques presented in this chapter are still applicable. However, the situation becomes messy when magic sets rewritings are applicable for certain orders. There might be several orders that trigger magic sets rewritings, where each leads to a different magic sets rewriting. It

remains unclear how to estimate the cost of the program obtained through each rewriting. Instead of enumerating all possible orders, we can adopt a simple design that decides whether magic sets rewritings are applicable following the left-to-right order provided by the original rules.

Finally, the generated code is compiled into the final executable by invoking the Visual C++ Compiler that comes with Visual Studio 2013 (version 120) 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. We are working on reducing the compilation latencies from several seconds to around 10 ms by taking advantage of compiler technologies such as LLVM [Lat08].

3.6 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. Both DeALS and SociaLite perform code generation for a given query. The execution times for both systems do not include the time spent on code generation and compilation, as these overheads are usually small for long-running queries.

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 22 queries following the query plans described in [DS13].[6] We tested the performance of DeALS on a test machine with four AMD Opteron 6376 CPUs (16 cores per CPU) and 256 GB memory (configured into eight NUMA regions). The operating system is Ubuntu Linux 12.04 LTS.

DeALS is able to correctly evaluate all 22 queries on databases of size 1, 10, and 100 GB on the test machine, while the total query evaluation time for all 22 queries using 64 processors is 1.020,

[6] 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`.

3.998, and 38.230 s, respectively.[7] The speedup of evaluating all 22 queries using 64 processors over using one processor is 5.31, 16.57, and 26.37, respectively. The predicted query evaluation time would be 378.122 s on a database of size 1 TB if the evaluation time of DeALS scales linearly w.r.t. the size of the database.

We also compared DeALS with the current single-machine world record for the benchmark on databases of size 100 GB and 1 TB. VectorWise 2.0.1 evaluates all the queries in 22.8 s on a database of size 100 GB [tpcc], while Microsoft SQL Server 2014 Enterprise Edition takes 138 s to run this benchmark on a database of size 1 TB [tpcb]. Note that the SPECint_rate2006 of our test machine is 1050 [cpuc] (the larger the more powerful), while the values are 695 [cpub] and 2400 [cpua] for the other two machines. The predicted evaluation time would be 15.091 s (315.429 s) for VectorWise 2.0.1 (SQL Server 2014) on a database of size 100 GB (1 TB) on our test machine if the evaluation time is inversely proportional to the SPECint_rate2006 of the machine. Thus, DeALS achieves very competitive performance comparing with these two highly optimized commercial systems on the TPC-H benchmark—it is only 2.6× and 1.2× slower comparing with VectorWise 2.0.1 and SQL Server 2014, respectively.

Exp II: Non-recursive programs—graph queries. We compare DeALS with LogicBlox and a commercial RDBMS 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 [LK14].[8] The database contains only one table $\texttt{arc}(\texttt{X}, \texttt{Y})$ where a unique index on $(\texttt{X}, \texttt{Y})$ and a secondary index on `Y` are built. The queries are listed below:

$$\texttt{3clique}(\texttt{count<\_>}) \texttt{ <- } \texttt{arc}(\texttt{X}, \texttt{Y}), \texttt{X} < \texttt{Y}, \texttt{arc}(\texttt{Y}, \texttt{Z}), \texttt{Y} < \texttt{Z}, \texttt{arc}(\texttt{Z}, \texttt{X}).$$

$$\texttt{4cycle}(\texttt{count<\_>}) \texttt{ <- } \texttt{arc}(\texttt{X}, \texttt{Y}), \texttt{X} < \texttt{Y}, \texttt{arc}(\texttt{Y}, \texttt{Z}), \texttt{Y} < \texttt{Z}, \texttt{arc}(\texttt{Z}, \texttt{W}),$$
$$\texttt{Z} < \texttt{W}, \texttt{arc}(\texttt{W}, \texttt{X}).$$

[7] DeALS is about 2× faster than the version used in [YSZ15] on the TPC-H benchmark by function inline optimization.

[8] We use the graph as a directed graph for `4cycle`, and as a undirected graph for `3clique` and `4clique`.

$$\texttt{4clique(count<\_>)} \texttt{ <- arc(X, Y), X} < \texttt{Y, arc(Y, Z), Y} < \texttt{Z, arc(Z, X),}$$
$$\texttt{arc(Z, W), Z} < \texttt{W, arc(X, W), arc(Y, W).}$$

We used a machine running Ubuntu Linux 14.04 LTS with an Intel Xeon E5-2660 CPU (eight hyperthreads), 28 GB memory, and 400 GB SSD to run DeALS, LogicBlox 4.1.9, and a commercial RDBMS. The speedups of DeALS on these three queries are 6.48, 5.96, and 6.20, respectively (evaluation time using one processor divided by the time using eight processors). Figure 3.5 compares the query evaluation time of the three systems.

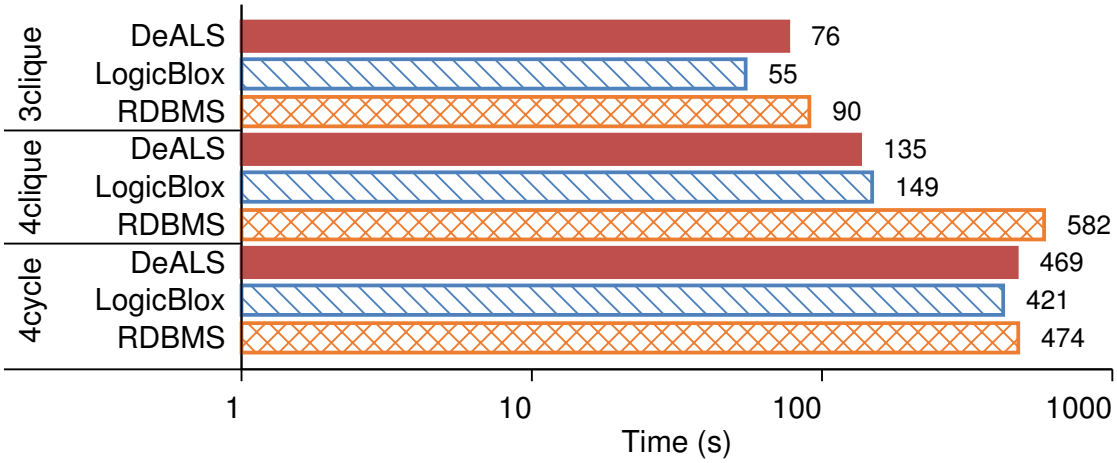

Figure 3.5: Query evaluation time of graph queries on the Pokec social network graph.

DeALS is faster than the commercial RDBMS on all three queries, while LogicBlox is faster than DeALS on two queries which is due to the leapfrog triejoin algorithm 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 [NAB15] on a machine with similar hardware. Thus, DeALS achieves competitive performance on these graph queries comparing with other existing systems with declarative languages.

Exp III: Recursive programs. We test the performance of DeALS on evaluating recursive queries. Here, we report the results of three classical queries—`tc` (program in Example 3.3), `sg` (program in Example 3.7), and `attend` (program in Example 3.1). `tc` and `sg` are two widely-studied queries in the literature, while `attend` is a graph traversal query that uses the monotonic aggregate. We select `attend` as a representative among other graph traversal queries, including reachability, connected

component, and single-source shortest path, that share the same memory access pattern; but it is more "difficult" than other queries since it requires correct and efficient handling of the count aggregate in recursion, and this is indeed a non-trivial problem as we will see that two out of five systems in this comparison do not support this query. The test datasets contain synthetic graphs and real-world graphs as described below. (These graphs are also used in the experimental evaluation reported in Section 2.4.) The parameters of the graphs are shown in Table 3.1.

Synthetic Graphs

- 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.
- grid-150 is a 151×151 square grid.
- 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.

Real-World Graphs

- patent is the US patent citation graph [LK14]. Each vertex represents a patent, and each edge represents a citation between two patents.
- 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.

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

DeALS evaluates all three queries correctly on the test datasets, where it uses a lock-free plan for `tc`, and locking plans for `sg` and `attend`. We compare DeALS with four other Datalog systems, namely LogicBlox, DLV, CLINGO, and SociaLite, on the same multicore machine used in Exp I.

Table 3.1: Parameters of test graphs.

Name	Vertices	Edges	Query	Result Size
tree-11	71,391	71,390	tc	876,392
			sg	2,086,271,974
grid-150	22,801	45,300	tc	131,698,576
			sg	2,295,050
gnp-10K	10,000	100,185	tc	100,000,000
			sg	100,000,000
patent	3,774,769	16,518,948	attend	1,742,094
wiki	3,165,181	23,190,820	attend	1,374,035

For each test graph, the table shows # vertices and # edges in the graph, and # tuples in the output of a query on this graph.

- LogicBlox [ACG15] is a commercial deductive database system. We used LogicBlox 4.1.9 in our experiment. It supports all three queries and evaluates them correctly on the test datasets. Our analysis of its system logs reveals that it evaluates a recursive query using semi-naive evaluation, and each recursive rule is repeatedly evaluated as follows: (1) it derives all the new tuples using the tuples from the delta relation, where the delta relation contains the result of the exit rule in the first iteration; (2) if the head of the rule does not contain any aggregates, it removes all the duplicates from the new tuples; otherwise, it performs the aggregation using all the new tuples; (3) it computes the new delta relation, and if it is not empty, the evaluation continues from Step (1). We also observed that, the time spent on Step (1) and Step (2) typically dominates the evaluation time, and the system uses multiple processors for Step (1), while it uses only one processor for Step (2). The main difference with DeALS is that DeALS does not face the bottleneck of Step (2), and the monotonic aggregates are optimized with the *eager monotonic aggregate semi-naive evaluation* [SYZ15].

- DLV [LPF06] is a system that represents the state of the art in the area of disjunctive logic programming. We used a single-processor version of DLV released on Dec 17, 2012.[9] It correctly evaluates `tc` on all three test graphs, and `sg` on grid-150 and gnp-10K. It ran out of memory on our test machine with 256 GB memory on the evaluation of `sg` on tree-

[9]The single-processor version of DLV is downloaded from [dlvb]. Although a parallel version [dlva] 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 4 GB memory required by evaluation.

11. The version of DLV [dlvc] 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 4 GB memory required by evaluation. The system consists of three main modules: intelligent grounding, model generator, and model checker. For each test query, there is only one answer set, which is the output of the intelligent grounding module, and the remaining two modules simply return the answer set. Thus, the system spends most of the evaluation time on the intelligent grounding module, which computes the answer set using semi-naive evaluation. However, the implementation details are not documented and cannot be inferred from its system logs as little information is logged.

- CLINGO [GKK14] is another state-of-the-art system in disjunctive logic programming. We used CLINGO 4.5.0, and it supports all three queries on all the test datasets, except that it ran out of memory on the evaluation of `sg` on tree-11. The system consists of a sequential grounder that replaces the variables in the program with constants, and a parallel solver that computes the stable models from the output of the grounder. There is only one stable model for each test query, and it is exactly the output of the grounder. Although we used a parallel version of CLINGO, the evaluation time is dominated by the time spent on the grounder that utilizes only one processor. Similar to DLV, the grounder of CLINGO also performs semi-naive evaluation, but its implementation details are not known to the public.

- SociaLite [SGL13] is a graph analysis system that can evaluate Datalog-style queries. We used a parallel version of the system [SPS13] downloaded from [Soc]. It evaluates `tc` and `sg` correctly on the test datasets. However, it does not support `attend` as it only supports the standard count aggregate that cannot be used in recursion. In fact, given a query akin to the one in Example 3.1 where count is used instead of mcount, it produces inconsistent results for multiple runs of the same query on the same dataset. For a given query, the system generates a Java program that performs semi-naive evaluation. It stores a relation of arity two as an array of hash tables, and the array is ranged partitioned into many small partitions. Each partition of a recursive relation is protected by a lock, where a worker always acquires the lock before accessing the data in the partition. It is clear that the generated program in

SociaLite and the generated program in DeALS share many designs. However, as we will show in the experimental results, the subtle differences in designs lead to drastic differences in performance.

Figure 3.6 compares the evaluation time of five systems on these recursive queries. Bars for DLV and DeALS-1 show the evaluation time of DLV and DeALS using one processor, while bars for LogicBlox, clingo, SociaLite, and DeALS-64 show the evaluation time of LogicBlox, CLINGO, SociaLite, and DeALS using 64 processors. In our experiments, both SociaLite and DeALS utilize all the processors most of the time, while LogicBlox does not, and CLINGO utilizes only one processor most of the time.

When DeALS uses only one processor, it always outperforms DLV and CLINGO on these queries and datasets. Here, DeALS is limited to use one processor, while DLV is only capable of using one processor, and CLINGO spends most of the time on the grounder that uses one processor. Although all three systems perform the semi-naive evaluation of the same query on the same dataset, this comparison suggests that DeALS provides a tighter implementation compared with the other two.

Using only one processor, DeALS outperforms or performs as well as LogicBlox and SociaLite, while LogicBlox and SociaLite are allowed to use all 64 processors. Naturally, DeALS always significantly outperforms LogicBlox and SociaLite when they all use 64 processors. The performance gap between LogicBlox and DeALS is largely due to the staged evaluation used by LogicBlox, which stores all the derived new tuples in an intermediate relation, and performs deduplication or aggregation on the intermediate relation. For the evaluation that produces large amount duplicate tuples, such as `tc` on grid-150 and `sg` on tree-11, this strategy incurs a high space overhead, and the time spent on the deduplication, which uses only one processor, dominates the evaluation time. On the other hand, for the evaluation that produces very few duplicate tuples, such as `attend` on Wiki, the performance gap between LogicBlox and DeALS is much smaller.

Next, we discuss the design choices that lead to the performance gap between SociaLite and DeALS. SociaLite uses an array of hash tables with an initial capacity of around 1000 entries for a

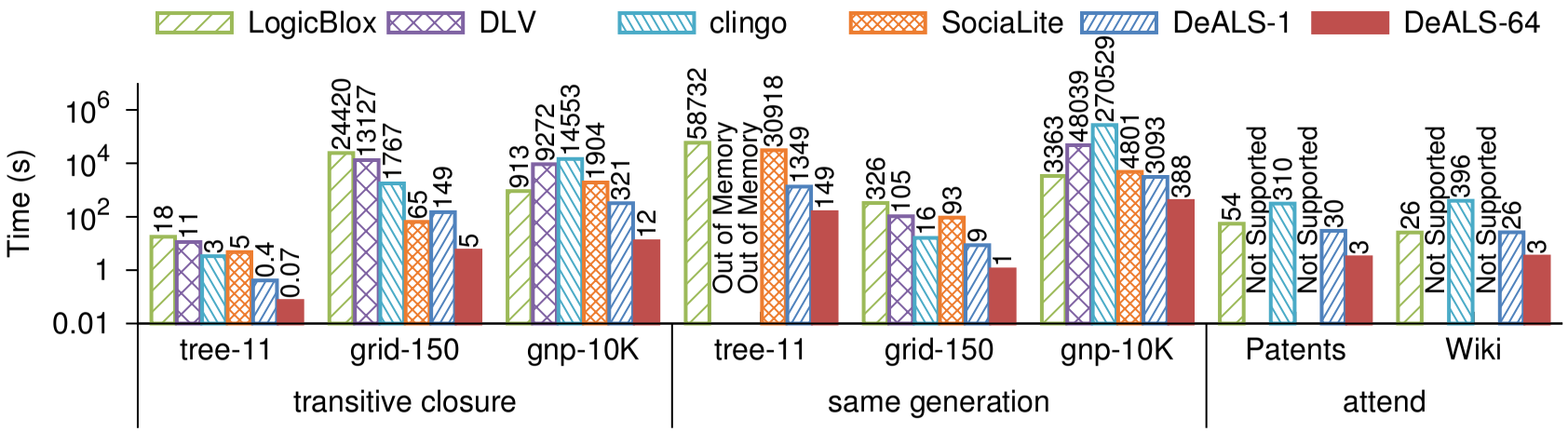

Figure 3.6: Query evaluation time of recursive queries.

derived relation, whereas DeALS uses an append-only structure to store the tuples and a B+ tree to index the tuples. Although the cost of accessing a hash table is lower than that of a B+ tree, the design adopted by DeALS allows a better memory allocation pattern as the relation grows—DeALS always requests for the space of a new node when a B+ tree grows, while SociaLite requests for the space that is twice the space used by the current hash table when a hash table grows. Indeed, the memory allocation pattern of B+ trees can be efficiently fulfilled by sequentially allocating small memory blocks from a large pre-allocated memory block, while the memory allocation pattern of hash tables requires lots of non-trivial work from the memory allocator. Such overhead is amplified when (i) multiple processors try to allocate memory at the same time, or (ii) the system has a high memory footprint. As a result, DeALS performs better than SociaLite, and the performance gap becomes larger for the evaluation of `tc` on gnp-10K and `sg` on tree-11, in which SociaLite uses more than 10 GB memory.

Note that DeALS achieves a greater speedup (i.e., the speedup of DeALS-64 over DeALS-1) for `tc` than `sg` and `attend` since no lock is used in `tc`, while `sg` and `attend` suffer from lock contention. It is important to note that the evaluation of `tc` still requires synchronization where the coordinator determines the next node 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 dominates the evaluation time, DeALS achieves very limited speedup—e.g., the evaluation of `tc` on tree-11.

A final note on this experiment is that DeALS is the only system that can handle all three queries on all the test graphs. The other systems suffer greatly on the evaluation of one or several queries on some test graphs, even though the size of all the test graphs is relatively small compared to the memory size of the test machine. We also performed experiments on larger synthetic graphs that exhibit the same structural properties as the graphs shown in Table 3.1. DeALS remains the only system that evaluates all these queries within a reasonable time.

3.7 Obtaining Lock-Free Programs

The experimental results in the previous section show that DeALS is able to achieve a good speedup using multiple processors in the evaluation of lock-free programs. In this section, we (i) present a sufficient condition for a program to be lock-free, and (ii) show how to rewrite a locking program into a lock-free program. For now, to simplify the discussion, we assume that the program does not contain negation, aggregates, and arithmetic expressions.

3.7.1 A Sufficient Condition

In the parallel evaluation of `tc` presented in Example 3.4, the i-th worker only reads from and writes to the i-th partition of `tc`, and finishes the evaluation when there are no new tuples in the i-th partition of `tc`. Thus, the coordinator is not necessary in the evaluation, i.e., there is no need for synchronization barriers in the nodes that represent recursion in the corresponding AND/OR tree. If the program is evaluated in the message passing model, and every worker has full access to all the base relations, then no worker needs to communicate with any other workers during the evaluation. This is the ideal situation since communication is expensive in the message passing model. Ganguly et al. [GST92] have proved (Theorem 5.1) that *there exists an evaluation plan for a linear single rule program that does not require any communication if the corresponding dataflow graph contains a cycle*. We will next generalize this result to arbitrary *linear programs*. For a multi-rule program, we say it is a *linear program* if the head predicate of a rule appears at most once in the rule body for every rule.

Definition 3.1. *For a recursive predicate* `p`*, let* `p`*(*X_1*, ..., *X_m*) be the head of a linear recursive rule* r *and* `p`*(*Y_1*, ..., *Y_m*) be the occurrence of* `p` *in the body of* r*. The* dataflow graph *of* $(r, \texttt{p})$ *is a directed graph* $G = (V, E)$*, where:*

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

For a program that contains equality predicates, i.e., predicates of the form `X` $=$ `Y` where `X` and

Y are variables or constants, we can remove the equality predicates, and construct the dataflow graph on the rewritten program. However, we can also construct the dataflow graph on the original program, while the obtained results lead to evaluation plans that follow the user's intention. The latter approach is adopted in DeALS.

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

Theorem 3.2. *Let P be a program where each recursive predicate is defined by a set of linear recursive rules. For every recursive predicate* p *in P, if the dataflow graphs corresponding to all the recursive rules with head* p *contain the same cycle, then there exists a lock-free parallel evaluation plan for P.*

Proof. The proof is a constructive one that is similar to Appendix C in [GST92]. For a recursive predicate p(X_1, ..., X_m), assume all the recursive rules with head p contain the same cycle $a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_t \rightarrow a_1$. Let the discriminating set of p be $\{a_1, a_2, \ldots, a_t\}$. For each recursive rule with head p, use the recursive predicate as the first predicate in the body of the rule during the adornment. Then the evaluation for p is lock-free. □

When the condition in the theorem is satisfied, this constructive proof produces a set of discriminating sets that satisfy all the constraints. Thus, we can determine the discriminating sets that lead to a lock-free plan without enumerating all the possible assignments.

Example 3.12. *Consider the following program where* p *is the recursive predicate. The recursive rules that define* p *are $r2$ and $r3$. The edge set for the dataflow graph of $(r2, \mathtt{p})$ is $\{(2,2),(3,3)\}$. There is an edge from 2 to 2 since* Y *is the second argument in* p(W, Y, Z) *and* p(X, Y, Z)*. Similarly for the edge from 3 to 3. The edge set for the dataflow graph of $(r3, \mathtt{p})$ is $\{(1,3), (2,2), (3,1)\}$. Both graphs contain a self-loop from 2 to 2. Thus, the program has a lock-free plan.*

$r1$. p(X, Y, Z) <− $\mathtt{b}_1$(X, Y, Z).
$r2$. p(X, Y, Z) <− $\mathtt{b}_2$(X, W), p(W, Y, Z).
$r3$. p(X, Y, Z) <− $\mathtt{b}_3$(X), p(Z, Y, X).

We use the following adorned program to explain the evaluation of a lock-free plan. We pick the discriminating set of $\mathtt{p}$ *as* $\{2\}$*, which is the same as the one picked in the proof of Theorem 3.2. The discriminating sets of* $\mathtt{b}_1$*,* $\mathtt{b}_2$*, and* $\mathtt{b}_3$ *are* $\{2\}$*,* $\{2\}$*, and* $\{1\}$*, respectively.*

$$r1'.\ \mathtt{p}^{\mathtt{fff}}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \text{ <- } \mathtt{b}_1^{\mathtt{fff}}(\mathtt{X},\mathtt{Y},\mathtt{Z}).$$
$$r2'.\ \mathtt{p}^{\mathtt{fff}}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \text{ <- } \mathtt{p}^{\mathtt{fff}}(\mathtt{W},\mathtt{Y},\mathtt{Z}), \mathtt{b}_2^{\mathtt{fb}}(\mathtt{X},\mathtt{W}).$$
$$r3'.\ \mathtt{p}^{\mathtt{fff}}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \text{ <- } \mathtt{p}^{\mathtt{fff}}(\mathtt{Z},\mathtt{Y},\mathtt{X}), \mathtt{b}_3^{\mathtt{b}}(\mathtt{X}).$$

During the evaluation, the i*-th worker evaluates (i)* $r1'$ *by scanning* $\mathtt{b}_1^i$ *and writing to* $\mathtt{p}^i$*; (ii)* $r2'$ *by scanning* $\mathtt{p}^i$*, looking for tuples* $(\mathtt{X},\mathtt{W})$ *from* $\mathtt{b}_2^{h(\mathtt{W})}$*, and writing to* $\mathtt{p}^i$*; (iii)* $r3'$ *by scanning* $\mathtt{p}^i$*, looking for tuples* $(\mathtt{X})$ *from* $\mathtt{b}_3^{h(\mathtt{X})}$*, and writing to* $\mathtt{p}^i$*.*

3.7.2 Rewriting to Lock-Free Programs

Now we present a rewriting that can transform a locking program into a lock-free program. The rewriting replaces an occurrence of a recursive predicate $\mathtt{p}$ with a predicate $\mathtt{p}^{(j)}$ that caches the state of $\mathtt{p}$ at the end of the previous iteration. The corresponding relation is repartitioned and stored in $\mathtt{p}^{(j)}$. 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 are implemented as in-memory shuffles). The rewriting consists of two steps: (1) from a nonlinear program to a linear program; and (2) from a linear program to a lock-free program.

Step (1). For each recursive predicate $\mathtt{p}$ and each nonlinear recursive rule r that can be canonically represented as

$$\mathtt{p}(X_1,\ldots,X_m) \text{ <- } \mathtt{p}(Y_{1,1},\ldots,Y_{1,m}),\ldots,$$
$$\mathtt{p}(Y_{t,1},\ldots,Y_{t,m}), b_1,\ldots,b_s.$$

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

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

The program becomes a linear program after rewriting. Now we test whether the new program satisfies the condition in Theorem 3.2. If not, we continue to Step (2).

We demonstrate how Step (1) of the rewriting can rewrite a nonlinear program that does not have lock-free plans into a linear program that has a lock-free plan with the example below.

Example 3.13. *A nonlinear formulation of the transitive closure program is shown as follows.*

$$
\begin{aligned}
& r1.\ \mathtt{tc}(\mathtt{X}, \mathtt{Y}) \text{ <- } \mathtt{arc}(\mathtt{X}, \mathtt{Y}). \\
& r2.\ \mathtt{tc}(\mathtt{X}, \mathtt{Y}) \text{ <- } \mathtt{tc}(\mathtt{X}, \mathtt{Z}), \mathtt{tc}(\mathtt{Z}, \mathtt{Y}).
\end{aligned}
$$

$r2$ is a nonlinear rule about tc*. There are two constraints about $r2$ in the RWA on the AND/OR tree:*

$$
\begin{aligned}
& \mathtt{tc}(\mathtt{X}, \mathtt{Z})[\overline{X_1}] = \mathtt{tc}(\mathtt{X}, \mathtt{Y})[\overline{X_1}] \\
& \overline{X_1} \subseteq \{1\},\ \mathtt{tc}(\mathtt{X}, \mathtt{Z})[\overline{X_1}] = \mathtt{tc}(\mathtt{Z}, \mathtt{Y})[\overline{X_1}]
\end{aligned}
$$

where $\overline{X_1}$ is the discriminating set of tc*. The two constraints cannot be satisfied at the same time, i.e., the program does not have lock-free plans. Replacing the second occurrence of* tc *with* $\mathtt{tc}^{(1)}$ *leads to the following program:*

$$
\begin{aligned}
& r1'.\ \mathtt{tc}(\mathtt{X}, \mathtt{Y}) \text{ <- } \mathtt{arc}(\mathtt{X}, \mathtt{Y}). \\
& r2'.\ \mathtt{tc}(\mathtt{X}, \mathtt{Y}) \text{ <- } \mathtt{tc}(\mathtt{X}, \mathtt{Z}), \mathtt{tc}^{(1)}(\mathtt{Z}, \mathtt{Y}). \\
& r3'.\ \mathtt{tc}^{(1)}(\mathtt{X}, \mathtt{Y}) \text{ <- } \mathtt{tc}(\mathtt{X}, \mathtt{Y}).
\end{aligned}
$$

Now the dataflow graph of $(r2', \mathtt{tc})$ contains a self-loop from 1 to 1. Thus, the program has a lock-free plan, in which the discriminating set of tc *is $\{1\}$.*

Step (2). For each recursive predicate q, where the dataflow graphs corresponding to all the recursive rules with head q contain the same cycle, we keep all the rules with head q. For each remaining recursive predicate p and each linear recursive rule r that can be canonically represented as

$$\mathtt{p}(X_1, \ldots, X_m) \text{ <- } \mathtt{p}(Y_1, \ldots, Y_m), b_1, \ldots, b_s.$$

where $\mathtt{p}(Y_1, \ldots, Y_m)$ 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:

$$\mathtt{p}(X_1, \ldots, X_m) \text{ <- } \mathtt{p}^{(1)}(Y_1, \ldots, Y_m), b_1, \ldots, b_s.$$
$$\mathtt{p}^{(1)}(X_1, \ldots, X_m) \text{ <- } \mathtt{p}(X_1, \ldots, X_m).$$

The new program is still a recursive program; however, now in each iteration, it is impossible for a worker to read from a partition of p while the same partition is being modified by another worker, thereby eliminating the need for locks on p.

Our rewriting introduces additional rules and predicates to the original program. It requires extra space to store these additional predicates. The space overhead of our rewriting is characterized by the following theorem.

Theorem 3.3. *Let S be the amount of space required to evaluate a recursive program P. If m is the maximal number of occurrences of a recursive predicate in P, then our rewriting finds a lock-free version of P that can be evaluated with at most $m \times S$ space.*

We have shown that sg does not have lock-free plans in Example 3.8 to Example 3.11. In the following example, we show how to obtain a lock-free plan for sg using Step (2) of the rewriting.

Example 3.14. *The* sg *program in Example 3.7 can be rewritten into the following lock-free program by replacing the occurrence of* sg *in $r2$ with* $\mathtt{sg}^{(1)}$*:*

$$\mathtt{sg}^{\mathtt{ff}}(\mathtt{X}, \mathtt{Y}) \text{ <- } \mathtt{anc}^{\mathtt{ff}}(\mathtt{A}, \mathtt{X}), \mathtt{anc}^{\mathtt{bf}}(\mathtt{A}, \mathtt{Y}), \mathtt{X} \neq \mathtt{Y}.$$
$$\mathtt{sg}^{\mathtt{ff}}(\mathtt{X}, \mathtt{Y}) \text{ <- } \mathtt{anc}^{\mathtt{ff}}(\mathtt{A}, \mathtt{X}), \mathtt{sg}^{(1)\mathtt{bf}}(\mathtt{A}, \mathtt{B}), \mathtt{anc}^{\mathtt{bf}}(\mathtt{B}, \mathtt{Y}).$$
$$\mathtt{sg}^{(1)\mathtt{ff}}(\mathtt{X}, \mathtt{Y}) \text{ <- } \mathtt{sg}^{\mathtt{ff}}(\mathtt{X}, \mathtt{Y}).$$

The new program has a lock-free plan, in which the discriminating sets of $\mathtt{sg}$ *and* $\mathtt{sg}^{(1)}$ *are both* $\{1\}$*, and the discriminating sets of the four occurrences of* $\mathtt{anc}$ *are* $\{2\}$*,* $\{1\}$*,* $\{2\}$*, and* $\{1\}$*, respectively.* $\mathtt{anc}(\mathtt{X},\mathtt{Y})$ *is preprocessed and partitioned in two different ways: in one partitioning, it is partitioned by the second column, and the tuples from one partition can be accessed by a scan; in the other partitioning, it is partitioned by the first column, and the tuples from one partition can be accessed by looking up a secondary index.*

The program evaluation starts from the exit rule where the i*-th worker scans from the* i*-th partition of* $\mathtt{anc}$*, looks for tuples* $(\mathtt{A},\mathtt{Y})$ *from* $\mathtt{anc}$ *with a fixed* $\mathtt{A}$*, and writes to the* i*-th partition of* $\mathtt{sg}$*. In each iteration of the evaluation of the recursive rules, the* i*-th worker copies all the (new) tuples in the* i*-th partition of* $\mathtt{sg}$ *to the* i*-th partition of* $\mathtt{sg}^{(1)}$*; after all the workers finish the copy operation, the* i*-th worker scans from the* i*-th partition of* $\mathtt{anc}$*, looks for tuples* $(\mathtt{A},\mathtt{B})$ *from* $\mathtt{sg}^{(1)}$ *with a fixed* $\mathtt{A}$*, looks for tuples* $(\mathtt{B},\mathtt{Y})$ *from* $\mathtt{anc}$ *with a fixed* $\mathtt{B}$*, and writes to the* i*-th partition of* $\mathtt{sg}$*. Although there is one more synchronization in each iteration, no locks are needed in the evaluation.*

3.7.3 Implementation

The rewriting introduces predicates $\mathtt{p}^{(1)}, \mathtt{p}^{(2)}, \mathtt{p}^{(3)}, \ldots$ that contain the same set of tuples as the original predicate $\mathtt{p}$. In each iteration of the iterative evaluation, the workers first copy the tuples in $\mathtt{p}$ to $\mathtt{p}^{(1)}, \mathtt{p}^{(2)}, \mathtt{p}^{(3)}, \ldots$, and then evaluate the rules that correspond to rules in the original program, where all the workers only read from $\mathtt{p}^{(1)}, \mathtt{p}^{(2)}, \mathtt{p}^{(3)}, \ldots$. Thus, for each $\mathtt{p}^{(\mathtt{j})}$, it is only necessary to allocate spaces for the secondary index that enables efficient access to $\mathtt{p}^{(\mathtt{j})}$ required by the program evaluation, while all $\mathtt{p}^{(\mathtt{j})}$ and $\mathtt{p}$ share only one copy of the actual relation that stores all the tuples. For each $\mathtt{p}^{(\mathtt{j})}$ that has the same discriminating set as $\mathtt{p}$, the i-th worker builds the secondary index of the i-th partition of $\mathtt{p}^{(\mathtt{j})}$ by scanning from the i-th partition of $\mathtt{p}$. The case where the two predicates have different discriminating sets is more complex. The lock-free way of building the secondary indexes is implemented as follows:

1. The coordinator allocates an array of addresses $addr$. Its size is the same as the number of tuples in $\mathtt{p}$ that will be accessed in the current iteration. It also allocates a two-dimensional array $hist$ of size $n \times n$.

2. The i-th worker scans through the i-th partition of $\mathtt{p}$, counts the number of tuples that belong to each partition of $\mathtt{p}^{(\mathtt{j})}$, stores the values in $hist[i]$, and computes the prefix sum of $hist[i]$.

3. The i-th worker allocates an array offset of size n, where

$$\mathit{offset}[k] = \sum_{t=1}^{i-1} hist[t][k] + \sum_{t=i}^{n} hist[t][k-1],$$

   for $k = 1, \ldots, n$. The i-th worker then scans through the i-th partition of $\mathtt{p}$ again. For each tuple t encountered during the scan, let d be the address of t, if t belongs to the k-th partition of $\mathtt{p}^{(\mathtt{j})}$, it sets $addr[\mathit{offset}[k]] = d$ and increments $\mathit{offset}[k]$.

4. The addresses of tuples that belong to the i-th partition of $\mathtt{p}^{(\mathtt{j})}$ are stored in $addr$ between index $\sum_{t=1}^{n} hist[t][i-1]$ and index $\sum_{t=1}^{n} hist[t][i]$. The i-th worker retrieves each tuple based on its address, and adds the tuple to the i-th partition of the corresponding secondary index.

The above mathematical expressions assume the index of an array starts with one, and the element in any undefined index is zero. There is a synchronization barrier after each step in the above procedure. Step (2) and Step (3) perform an in-memory shuffle that collects the addresses of tuples that belong to the same partition of $\mathtt{p}^{(\mathtt{j})}$ in a consecutive space in memory. After the in-memory shuffle, each worker creates one partition of the secondary index without interfering any other workers.

Optimizing Semi-Naive Evaluation. In the semi-naive evaluation, the rewriting that introduces new delta rules can be viewed as a special case of our rewriting. Each occurrence of the delta relation can be viewed as a base relation in the RWA, where we only need to ensure the corresponding discriminating set is a subset of the set of positions for the bound arguments. There is a lock-free semi-naive evaluation plan if all the constraints can be satisfied. We keep two copies of the secondary index of the relation, where one copy indexes the tuples in the old delta relation, and the other copy indexes the tuples in the new delta relation. The i-th worker reads from the index that corresponds to the old delta relation, and updates the index that corresponds to the new delta

relation, when a new tuple is added to the i-th partition of the relation. At the end of each iteration, each worker first switches the pointers that point to these two indexes, and then clears the index that corresponds to the old delta relation.

Example 3.15. *If we apply the above optimization on the evaluation of* sg*, then the fourth constraint in Equation (3.1) is replaced by* $\overline{X_1} \subseteq \{1\}$*. The assignment* $\overline{X_1} = \overline{X_3} = \overline{X_5} = \{1\}$*,* $\overline{X_2} = \overline{X_4} = \{2\}$ *satisfies all the constraints. Thus, there is a lock-free semi-naive evaluation plan for* sg*. The plan is similar to the one in Example 3.14.*

3.7.4 Performance Comparison

The experimental results in Figure 3.6 show that DeALS achieves limited speedup on the evaluation of sg and attend due to the use of locks. However, there are lock-free plans for both queries. We have shown a lock-free plan for sg. The following example shows a lock-free plan for attend.

Example 3.16. *A lock-free plan for* attend*. Consider the following adorned program, where each relation is partitioned by its first column, except for* friend *that is partitioned by its second column.*

$$\texttt{cntfriends}^{\texttt{ff}}(\texttt{Y}, \texttt{mcount}\langle\texttt{X}\rangle) \texttt{<-} \texttt{friend}^{\texttt{ff}}(\texttt{X}, \texttt{Y}), \texttt{attend}^{\texttt{b}}(\texttt{X}).$$

$$\texttt{attend}^{\texttt{f}}(\texttt{X}) \texttt{<-} \texttt{organizer}^{\texttt{f}}(\texttt{X}).$$

$$\texttt{attend}^{\texttt{f}}(\texttt{Y}) \texttt{<-} \texttt{cntfriends}^{\texttt{ff}}(\texttt{Y}, \texttt{N}), \texttt{N} \geq 3.$$

In each iteration, the i-th worker scans through the i-th partition of friend*, and checks whether there is a specific* X *in* attend *that is supported by maintaining an index on the delta relation of* attend*. Thus, the i-th worker only writes to the i-th partition of* cntfriends *and* attend*, and no lock is needed.*

Table 3.2 compares the evaluation time of DeALS on sg and attend using the locking plans and the lock-free plans. For a query evaluation plan and a dataset, it shows the evaluation time using one processor, the evaluation time using 64 processors, and the speedup of using 64 processors over using one processor. The results in the table show that DeALS achieves a much higher speedup using a lock-free plan comparing to a locking plan.

Table 3.2: Query evaluation time of DeALS on sg and attend using different plans.

Query(plan)	Dataset	Time-1 (s)	Time-64 (s)	Speedup
sg(locking)	tree-11	1349.299	148.725	9.072
	grid-150	8.561	1.059	8.083
	gnp-10K	3092.658	388.212	7.966
sg(lock-free)	tree-11	1309.118	48.116	27.207
	grid-150	7.613	0.354	21.505
	gnp-10K	2926.720	79.119	36.991
attend (locking)	Patents	29.820	3.025	9.858
	Wiki	26.122	3.202	8.158
attend (lock-free)	Patents	65.112	2.917	22.319
	Wiki	89.473	6.387	14.009

"Time-1" column and "Time-64" column show the evaluation time (in seconds) using one processor and 64 processors, respectively.

3.7.5 Discussion

The evaluation time using one processor reflects the actual cost of evaluation without the overhead of parallel evaluation. The results in Table 3.2 show that the lock-free plan of sg has a slightly lower cost than the locking plan of sg, while the lock-free plan of attend has a significantly higher cost than the locking plan of attend. For sg, the slight difference in cost is due to the different memory access patterns in the two plans—the write operations to sg are grouped by X in the lock-free plan, while there is no clear ordering on the write operations in the locking plan. Although a lock-free plan produces better memory access patterns, this may not always be the best plan in terms of evaluation time. This is demonstrated by the evaluation of attend, where the evaluation time is dominated by the time spent on index lookups. Let N be the number of tuples in friend. The number of index lookups performed by the locking plan is bounded by $3N$, while the number of index lookups in the lock-free plan is $T \times N$, where T is the number of iterations in the semi-naive evaluation. Thus, the lock-free plan has a much higher cost than the locking plan when the semi-naive evaluation requires more than three iterations.

Besides the cost of evaluation, a lock-free plan might require partitioning a relation by a certain column that only takes a few distinct values, which leads to imbalanced load among processors where most processors are idle during evaluation.

Example 3.17. *Consider the following program, where* X *is the primary key in* q *and* $h(\texttt{Z})$ *only takes four values.*

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

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.*

Therefore, DeALS also provides users with the option of override the system's default optimization plan. In fact, a user can force the system to partition a relation by certain columns by specifying a discriminating set for a predicate in the program. DeALS tries to find the best parallel evaluation plan for the given program, where the predicates in every rule are evaluated in the same order as they are provided in the program.

3.8 Related Work

The parallel evaluation strategy proposed in this chapter uses a simple hash-based data partitioning strategy. Various data partitioning strategies for parallel bottom-up evaluation have been studied in [WS88, Wol88, CW89, SL91, GST92, ZWC95, GST95]. 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 theoretical aspects. Strategies for top-down evaluation in the shared-memory model are proposed in [RS86, Hul89, BSH91], while in this chapter we focus on the bottom-up evaluation.

Another line of related work is new systems that can evaluate Datalog programs on a multi-node cluster [SPS13, WBH15]. The techniques presented in these papers are complementary to

ours since our study focuses on the parallel evaluation on a single machine. Another difference is the treatment of aggregates in recursion which is discussed in Section 3.2. Moreover, these studies focus on the evaluation of recursive queries with aggregates, while our study covers recursive queries with/without aggregates and non-recursive queries. Finally, although these distributed systems have a potential to scale to larger datasets and clusters, DeALS achieves superior performance as a single-machine solution when the datasets fit in the memory of a single machine.

3.9 Conclusion

In this chapter, we presented the compilation techniques used by DeALS for in-memory parallel evaluation of Datalog programs on shared-memory multicore machines. The techniques produce efficient parallel evaluation plans for both non-recursive and recursive programs. As a result, DeALS achieves superior performance on the evaluation of recursive queries compared with other existing systems, while maintaining competitive performance with commercial RDBMSs on non-recursive queries. Moreover, since the speedup of parallel evaluation over sequential evaluation is modest for evaluation plans that use locks, we presented rewriting techniques that transform the given program into a lock-free program.

While DeALS is a very robust prototype, there remain plenty of opportunities for future improvement. Our ongoing work seeks to further optimize the code generation which reduces the performance gap between DeALS and the hand written optimal programs, including (i) employing a vectorized processing model [BZN05] and the techniques presented in [DS13] to improve the performance on non-recursive queries; (ii) implementing the SSC12 algorithm introduced in the previous chapter for transitive closure-like recursive queries; and (iii) providing a worst-case optimal guarantee for joins used by both non-recursive and recursive queries with the leapfrog triejoin algorithm [Vel14]. Another improvement planned for the future is to study techniques that can be integrated into DeALS to improve its performance when skew is present.

CHAPTER 4

Scaling out Datalog on Clusters

In this chapter, we present an overview of the BigDatalog system that focuses on the distributed evaluation of Datalog programs on clusters. More details about the query evaluation and optimization techniques used by BigDatalog are discussed in [SYI16] and [Shk16].

4.1 Introduction

BigDatalog is a **De**ductive **A**pplication **L**anguage (DeAL) implementation on Spark [ZCD12]. It enables Spark programmers to implement complex analytics pipelines of relational, graph, and machine learning tasks in a single declarative language, instead of stitching together programs written in different subsystem APIs, i.e., Spark SQL [AXL15], GraphX [GXD14], and MLlib [MBY16]. Moreover, it employs techniques to identify and evaluate recursive programs that are *decomposable* and can be evaluated without communication [WS88, SL91], leading to efficient distributed evaluations.

4.1.1 Challenges

There are following three main challenges to implement BigDatalog on Spark:

1. **Acyclic Plans**. Supporting compilation, optimization, and evaluation of Datalog programs on Spark requires features not currently supported. In particular, Spark SQL lacks recursion operators, operators are designed for acyclic use, and the Catalyst optimizer is targeted for non-recursive plans.

2. **Scheduling**. Spark's synchronous stage-based scheduler issues tasks for a stage only after all tasks of the previous stages have completed. This can be seen as unnecessary coordination for monotonic Datalog programs because these programs are eventually consistent [ANB11, IT15].

3. **RDD Immutability & Memory Utilization**. An iteration of recursion will produce a new *Resilient Distributed Dataset* (RDD) to represent the updated recursive relation. This RDD will contain both new facts and all the facts produced in earlier iterations, which are already contained in earlier RDDs. If poorly managed, recursive applications on Spark can experience memory utilization problems.

4.1.2 Contributions

We make the following contributions through the design and implementation of our BigDatalog system:

- We show how recursive Datalog programs are compiled into recursive physical plans for Spark.
- We present a parallel evaluation technique for distributed Datalog evaluation on Spark. We introduce recursion operators and data structures to efficiently implement the technique in Spark.
- We propose physical planning and scheduler optimizations for recursive Datalog programs in Spark, including techniques to evaluate decomposable programs.
- We present distributed monotonic aggregates, and accompanying evaluation technique and data structures to support Datalog programs with aggregates on Spark.
- We provide experimental evidence that a generic declarative system can compete with a special-purpose graph system.

4.2 System Overview

4.2.1 Parallel Semi-Naive Evaluation on Spark

BigDatalog evaluates programs using a parallel version of standard semi-naive evaluation we call *parallel semi-naive* (PSN) evaluation. PSN is an execution framework for a recursive predicate and it is implemented using RDD transformations. Since Spark evaluates synchronously, PSN will evaluate one iteration at a time, where an iteration will not begin until all tasks from the previous iteration have completed.

The two types of rules for a recursive predicate, the exit rules and recursive rules, are compiled into separate *physical plans* which are then used in the PSN evaluator. Physical plans are composed of Spark SQL and BigDatalog operators that produce RDDs. The exit rules plan is first evaluated once, and then the recursive rules plan is repeatedly evaluated until a fixpoint is reached.

The psuedocode for the PSN evaluator is shown in Figure 4.1. The `exitRulesPlan` (line 1) and `recursiveRulesPlan` (line 5) are plans for the exit rules and recursive rules, respectively. We use toRDD (lines 1, 5) to produce the RDD for the plan. Each iteration produces two new RDDs—an RDD `delta` for the new results produced during the iteration, and an RDD `all` for all results produced thus far for the predicate. The method updateCatalog (lines 3, 7) stores new `all` and `delta` RDDs into a catalog for plans to access. The exit rule plan is evaluated first. The result is de-duplicated by distinct (line 1) to produce the initial `delta` and `all` RDDs (line 2), which are used to evaluate the first iteration of the recursion. Each iteration is a new job executed by count (line 8). First, the `recursiveRulesPlan` is evaluated using the `delta` RDD from the previous iteration. This will produce an RDD that is set-differenced (subtract) with the `all` RDD (line 5) and de-duplicated to produce a new `delta` RDD. With lazy evaluation, the union of `all` and `delta` (line 6) from the previous iteration is evaluated prior to its use in subtract (line 5).

4.2.2 Evaluation Plans

For a given Datalog program, the BigDatalog compiler first creates a logical plan for the program, and then maps the logical plan into a Spark SQL plan or a BigDatalog physical plan depending

```
1: delta := exitRulesPlan.toRDD().distinct()
2: all := delta
3: updateCatalog(all, delta)
4: repeat
5:    delta := recursiveRulesPlan.toRDD().subtract(all).distinct()
6:    all := all.union(delta)
7:    updateCatalog(all, delta)
8: until delta.count() = 0
9: return all
```

Figure 4.1: PSN evaluator with RDDs.

on whether the program is non-recursive or recursive. We use the transitive closure query `tc` in Example 3.3 to illustrate the process.

Logical Plans. The program is compiled into an AND/OR tree (discussed in Chapter 3.3.2), and then the logical plan is produced by it into a tree of relational and recursion operators. A recursion operator has two child logical (sub)plans: one plan for the predicate's exit rules and the other for the predicate's recursive rules. Figure 4.2(a) is the logical plan produced by the BigDatalog compiler for `tc`. The left side is the exit rules plan with only the `arc` relation, representing the exit rule, while the right side is the recursive rules plan made up of relational operators to produce one iteration of the recursive rule.

Physical Plans. Figure 4.2(b) is a physical plan for the logical plan in Figure 4.2(a). The root of the plan is the *recursion operator* (RO) for the `tc` recursive predicate. It is a special driver operator that runs on the master and executes PSN. An RO has two child physical (sub)plans, the *Exit Rules Plan* (ERP) and the *Recursive Rules Plan* (RRP). In the RRP, δ`tc` is a recursive relation and when evaluated will produce `tc`'s facts from the previous iteration. In each iteration, δ`tc` is joined with `arc` to derive new tuples. BigDatalog uses binary hash join operators, and a multi-way join is converted into a hierarchy of binary join operators in a left-to-right fashion. Here, both inputs to the binary hash join are shuffled. The subscript $_{\texttt{Z}, [\texttt{N}]}$ indicates the partitioning key is the `Z` argument (from the rule), and there will be `N` partitions. Here `Z` is the join argument so that tuples of `arc` and δ`tc` having the same key will be co-located on the same worker.

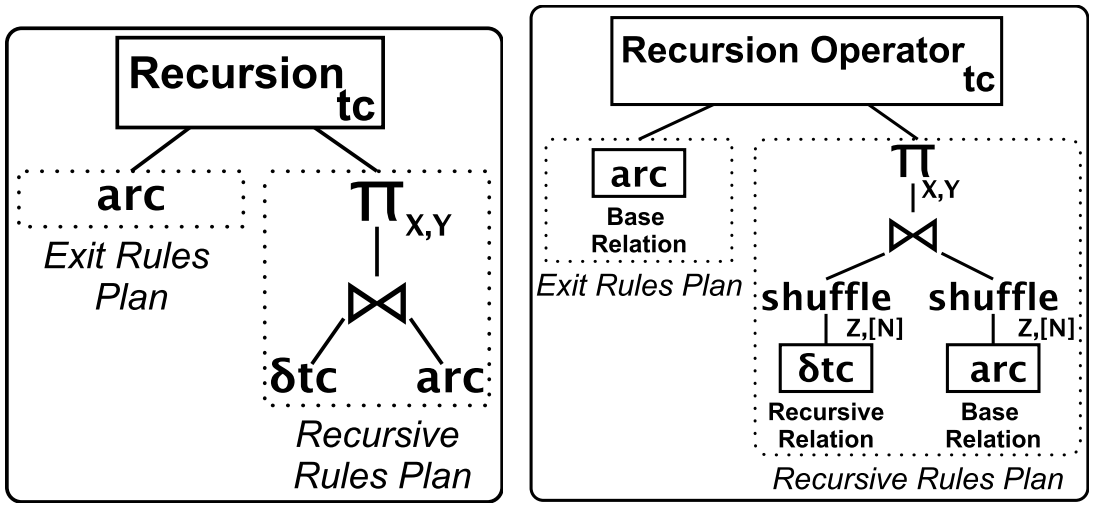

Figure 4.2: BigDatalog plans for `tc`.

4.2.3 Optimizations

BigDatalog employs various optimization techniques to enable efficient and scalable distributed evaluation of Datalog programs, including introducing a specialized RDD called SetRDD tailored for set operations needed for PSN, data partitioning optimization, join optimizations, optimization for decomposable programs, and job optimizations. These optimizations are discussed in details in [SYI16] and [Shk16]. Here, we show the effectiveness of these optimizations with example results from the transitive closure query evaluation. Again, more experiments and analysis are presented in [SYI16] and [Shk16].

Performance Comparison. The experiments are conducted on a 16 node cluster. Each node runs Ubuntu 14.04 LTS and has an Intel i7-4770 CPU (3.40 GHz, 4 core/8 thread), 32 GB memory and a 1 TB 7200 RPM hard drive. The cluster are connected with 1Gbit network. Our implementation is in Spark 1.4.0 and uses Hadoop 1.0.4.

Figure 4.3 shows the execution time required to compute a 100 million vertex pair transitive closure of a graph using a highly optimized handwritten Spark program versus the BigDatalog version. It is clearly to see that BigDatalog is both considerably better than its host framework and also performant w.r.t. other distributed Datalog systems, namely, Myria [WBH15] and SociaLite

[SPS13]. This orders of magnitude speed-up is achieved by employing the efficient evaluation techniques and optimizations of Datalog in Spark.

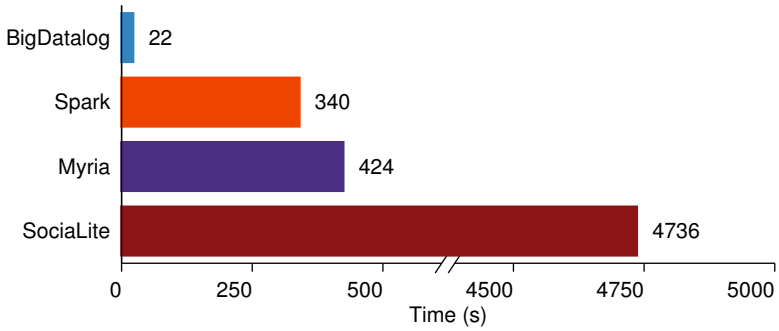

Figure 4.3: Example recursive query performance.

4.3 Conclusion

BigDatalog is a powerful Datalog system that enables Spark programmer to develop complex distributed algorithms in a declarative language, while maintaining the efficiency of highly optimized programs. In fact, BigDatalog can also be backed by other general dataflow systems, including Hyracks [BCG11] and Naiad [MMI13], and many of the optimization techniques that are currently employed by BigDatalog will also apply. This is the benefit of employing a declarative language, and comparing and analyzing the performance tradeoffs between different backend systems and/or computation models on various workloads would be an interesting future work.

An interesting direction for future research is to extend BigDatalog to support XY-stratification [ZCF97, AOT03]. This will realize the vision of [BBC12a] and [BBC12b], in which the authors

1. showed that many machine learning algorithms, such as PageRank and batch gradient descent, can be expressed using XY-stratified Datalog programs that dovetail with the MapReduce paradigm of cluster computing;

2. thus argued that users can greatly benefit from systems that support such feature in the development and deployment of complex machine learning algorithms over massively parallel systems.

In fact, this is the topic of the next chapter, where we present a language extension of DeAL with

the support for XY-stratification. It is interesting to note that this new language also supports monotonic aggregates, which can be used in recursive rules to express asynchronous computations that (i) can not be expressed by XY-stratification[1], and (ii) is also part of the vision of [BBC12a] and [BBC12b] but presented as a future work.

[1]The semantics of monotonic aggregates does not define whether the underlying computation is synchronous or asynchronous. Thus, the compiler is free to choose an implementation that is more efficient. In fact, we have shown in [SYZ15] that an asynchronous version, i.e., eager monotonic aggregate semi-naive evaluation, is usually more efficient than the synchronous version, i.e., monotonic aggregate semi-naive evaluation. Thus, our systems use an asynchronous implementation by default.

CHAPTER 5

Knowledge Discovery in Datalog

In this chapter, we present a new language, called **K**nowledge **D**iscovery in **D**ata**log** (KDDlog), that extends DeAL with support for knowledge discovery and data mining (KDD) applications.

5.1 Introduction

In the previous chapters, we have discussed the progress achieved through the research and development of DeAL and its system implementations. However, major research challenges remain before we can realize the ambitious goals of KDDlog that seeks to support KDD tasks in Datalog. In particular, we must overcome the following two limitations of current Datalog technologies:

1. Limited expressive power, whereby the current Datalog systems might not be able to express complex KDD tasks, or they can only express them in ways that are not amenable to efficient implementation and parallelization. The current state of the art is such that basic SQL and Datalog have demonstrated to be conducive to efficient implementation via optimization and scalability via data-oriented parallelism for both applications that do not require recursion, and those that do not require aggregates in recursion. However, most KDD tasks require aggregates and other non-monotonic constructs in recursion. In spite of the recent progress on monotonic aggregates that can be used in recursion [MSZ13a, MSZ13b], the programs with monotonic aggregates usually do not exhibit the same time and space complexities as that of procedure programs that a user would write for KDD tasks. Thus, there is still a need

for more general and efficient solutions.

2. Lack of genericity whereby programs must be written specifically for the number and types of each column in the table rather than for generic tables. A typical solution is to convert a given table from a row-oriented representation to a column-oriented representation, and write programs for tables using a column-oriented representation. However, this conversion introduces space overhead, and it is not easy for a Datalog system to fully optimize the accesses to the resulting tables for the programs that operate on these tables since the system is not aware of the properties of this representation.

Thus, in order to overcome the first limitation, we adopt an approach called *Constraint Pushing into Recursion (CPR)* that allows the use of non-monotonic constructs in recursion [ZYD16a, ZYD16b], and we demonstrate how it will allow us to express efficiently KDD algorithms. We focus on providing declarative formulations for these algorithms, while leaving the investigation for parallelization as a future work. In order to overcome the second limitation, we introduce constructs that achieve genericity by supporting verticalized views for tables, whereby we can write algorithms that work on tables with arbitrary number of attributes, and the system is then able to optimize the accesses to these tables.

The rest of this chapter is organized as follows. We first introduce CPR and the body aggregate notations in Section 5.2. Then we present our constructs for obtaining columnar views of tables and rollups in Section 5.3. After that, from Section 5.4 to Section 5.8, we show how to express efficiently major KDD algorithms in KDDlog, including data cubes, frequent item set mining, classification, regression analysis, and cluster analysis. We discuss how to support the exportation of KDD algorithms implemented in KDDlog and importation of algorithms implemented in other languages through user-defined aggregate functions in Section 5.9. The chapter concludes in Section 5.10.

5.2 Background

5.2.1 Constraint Pushing into Recursion

The declarative and constructive semantics of a Datalog program P is defined in terms of the *Immediate Consequences Operator* (ICO) for P, denoted by $T_P(I)$, and where I denotes a Herbrand interpretation of P. For basic Datalog, without negation or aggregates, $T_P(I)$ is a monotonic continuous mapping in the lattice of set-containment which the interpretation I belongs to, whereby we have the following well-known properties:

1. A unique minimal (w.r.t. set-containment) solution of the equation $I = T_P(I)$ always exists and it is known as the *least-fixpoint* of T_P *which defines the declarative semantics of* P.

2. The *fixpoint iteration* $T_P^{\uparrow\omega}(\emptyset)$, often called the *naive fixpoint* of T_P, defines the operational semantics of our program. For cases of practical interest, the computation only needs to be performed till the first integer $n+1$ where, $T_P^{n+1}(\emptyset) = T_P^n(\emptyset)$. For positive programs without negation and aggregates, the operational and declarative semantics coincide.

In order to express many practical applications, basic Datalog must be extended to allow the use of negation and aggregates, and in fact most Datalog compilers do so, but require that the programs be stratified w.r.t. negation and aggregates, such as the following program:

Example 5.1. *Shortest path from node* a.

$$r1.\ \texttt{path}(\texttt{Y}, \texttt{Dy}) \texttt{<-}\ \texttt{arc}(\texttt{a}, \texttt{Y}, \texttt{Dy}).$$
$$r2.\ \texttt{path}(\texttt{Y}, \texttt{Dy}) \texttt{<-}\ \texttt{path}(\texttt{X}, \texttt{Dx}), \texttt{arc}(\texttt{X}, \texttt{Y}, \texttt{Dxy}), \texttt{Dy} = \texttt{Dx} + \texttt{Dxy}.$$
$$r3.\ \texttt{spath}(\texttt{Y}, \texttt{min}\langle\texttt{Dy}\rangle) \texttt{<-}\ \texttt{path}(\texttt{Y}, \texttt{Dy}).$$

The syntax $\texttt{min}\langle\texttt{Dy}\rangle$ above illustrates the head notation for aggregates that is used in many Datalog systems, and follows SQL-2 approach of allowing zero, one or more group-by variables for aggregate. Thus, in our example Dy is the aggregate variable (specifically the min variable) and Y is the group-by variable. We will often refer to the min and max variables as *cost variables.*

Stratification can be used to avoid the semantic problems caused by using aggregates in recursion. For instance, in our example spath belongs to a stratum that is above that of path, whereby our program is assured to have a *perfect-model semantics* [Prz88]. The perfect model of a stratified program is unique and can be computed using an *iterated fixpoint computation* [ZCF97], whereby the least fixpoint is computed starting at the bottom stratum and moving up to higher strata. In our example, therefore, all the possible paths will be computed using rules $r1$ and $r2$, before selecting values that are minimal using $r3$. This is the approach used by current Datalog compilers, and it can be very inefficient or even non-terminating when the original graph contains cycles.

In fact, for large classes of programs, the computation can be significantly optimized by simply pushing constraints into the fixpoint computation. In Example 5.1 above, the constraint is imposed by the last rule that, for each point reached, selects the minimal value of its distance from a. This non-monotonic constraint can now be pushed into the recursive rules whereby the rules used by the compiler in the actual implementation become:

Example 5.2. *Optimized shortest path from node* a.

$$r1'.\ \texttt{path}(\texttt{Y}, \texttt{min}\langle\texttt{Dy}\rangle) \texttt{<-}\ \texttt{arc}(\texttt{a}, \texttt{Y}, \texttt{Dy}).$$
$$r2'.\ \texttt{path}(\texttt{Y}, \texttt{min}\langle\texttt{Dy}\rangle) \texttt{<-}\ \texttt{path}(\texttt{X}, \texttt{Dx}), \texttt{arc}(\texttt{X}, \texttt{Y}, \texttt{Dxy}), \texttt{Dy} = \texttt{Dx} + \texttt{Dxy}.$$
$$r3'.\ \texttt{spath}(\texttt{Y}, \texttt{Dy}) \texttt{<-}\ \texttt{path}(\texttt{Y}, \texttt{Dy}).$$

The rules so obtained define the optimized ICO that will be used in the fixpoint iteration to construct path. This optimization is referred to as *Constraint Pushing into Recursion (CPR)* optimization [ZYD16a, ZYD16b].

As illustrated by Example 5.1, the CPR optimization considers programs (or segments of larger programs) consisting of (i) rules defining one or more recursive predicate(s), and (ii) one constraint rule that has one or more of those recursive predicates as its goals with min, max, and possibly other constraints on their variables. Such programs are called *CPR-structured programs*. The CPR optimization removes the constraint goals from the constraint rule to add them to the rules defining the recursive predicates with the expectation that they will be compiled into an execution plan that

realizes the declarative semantics of the original program. In the optimized program, constraints are applied to the rules or facts defining the recursive predicate, and the old constraint rule has now become a *copy rule* that returns the values produced by the recursive predicate. Thus, the transformed program in Example 5.2 shows the min constraint now implanted into the heads of rules $r1'$ and $r2'$, whereas $r3'$ has become a copy rule from `path` to `spath`.

Now, given a CPR-structured program P, let γ and T_ρ denote the constraints in the constraint rule[1] and the recursive ICO, respectively. If the following equality holds for any interpretation I,

$$\gamma(T_\rho(I)) = \gamma(T_\rho(\gamma(I))),$$

then we say that γ *is implantable into* P*'s recursion*, and it is not hard to prove that the CPR-optimized version of P is both sound and complete, i.e., no incorrect result is ever produced by the optimized rules, and any result that is produced by the iterated fixpoint computation of P is also produced by the optimized rules. Thus, the CPR-optimized program (the copy rule is excluded) behaves as a positive program since $T_{\gamma\rho}$ has a unique least fixpoint computable as $T_{\gamma\rho}^{\uparrow\omega}(\emptyset)$.

In practice, it is not necessary to check whether the equality condition holds for every interpretation. In fact, there are other sufficient conditions that can be easily checked through simple syntactic analyses on a given program, and these conditions apply to most examples of practical interest [ZYD16b]. Thus, in the rest of this chapter, we use CPR-optimized programs without providing formal proofs on the correctness of the optimized programs.

5.2.2 Notations for Body Aggregates

We allow the use of aggregates in the body. For example, the rule $r3$ in Example 5.1 is equivalent to the following rule, where a body notation is used for the min aggregate:

$$\texttt{spath(Y, Dy)} \texttt{ <- path(Y, Dy), is\_min((Y), (Dy))}.$$

We also allow the use of aggregates with multiple arguments as in the following:

[1] We use the symbol γ to denote both a constraint, and the application of the constraint itself.

$$\texttt{onearc}(\texttt{X},\texttt{Y},\texttt{D}) \texttt{<-} \texttt{arc}(\texttt{X},\texttt{Y},\texttt{D}), \texttt{is\_min}((\texttt{X}),(\texttt{D},\texttt{Y})).$$

which is a short-hand for:

$$\texttt{minarc}(\texttt{X},\texttt{D}) \texttt{<-} \texttt{arc}(\texttt{X},\texttt{Y},\texttt{D}), \texttt{is\_min}((\texttt{X}),(\texttt{D})).$$
$$\texttt{onearc}(\texttt{X},\texttt{Y},\texttt{D}) \texttt{<-} \texttt{arc}(\texttt{X},\texttt{Y},\texttt{D}), \texttt{minarc}(\texttt{X},\texttt{D}), \texttt{is\_min}((\texttt{X}),(\texttt{Y})).$$

Thus we first select the nodes of min distance, and if there are several of such nodes, we select the one that has the min value (as nodes are numeric or belong to a totally ordered domain).

5.3 Basic Representations and Constructs

Now we start introducing the new constructs as well as the corresponding representations that are added to KDDlog to provide efficient support for KDD applications.

5.3.1 Verticalization

Data mining tool sets support algorithms that are *generic* and can be applied to datasets described by tables with arbitrary number of columns. However, in Datalog, SQL and other database query languages, different queries must be written for tables having different number of columns. To addresses the need for genericity, we describe an approach called *verticalization*, which displays column numbers and values for the tuples. For example, the well-known *PlayTennis* example from [Mit97] and the verticalized representation of its first three tuples are shown in Table 5.1 and Table 5.2, respectively.

We provide the meta-level construct `Val@Col`, i.e., the value `Val` at column `Col`, that turns the original row-oriented representation into a column-oriented one. For example, the rule of Example 5.3 transforms Table 5.1 into the vertical representation shown in Table 5.2, where `train` and `vtrain` are the names of the relations that store all the tuples in Table 5.1 and Table 5.2, respectively.

Example 5.3. *From a row-oriented representation to a column-oriented representation.*

$$\texttt{vtrain}(\texttt{ID},\texttt{Col},\texttt{Val}) \texttt{<-} \texttt{train}(\texttt{ID},\texttt{Val@Col}).$$

Table 5.1: Training examples for the PlayTennis table.

ID	**Outlook** (1)	**Temperature** (2)	**Humidity** (3)	**Wind** (4)	**PlayTennis** (5)
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no

Table 5.2: Vertical view of the tuples in Table 5.1.

ID	**Col**	**Val**
1	1	sunny
1	2	hot
1	3	high
1	4	weak
1	5	no
2	1	sunny
2	2	hot
2	3	high
2	4	strong
2	5	no
3	1	overcast
3	2	hot
3	3	high
3	4	weak
3	5	yes
...	...	...

Table 5.3: Tuples produced by the rule in Example 5.5.

ID	**Col**	**Val**	**PlayTennis**
1	1	sunny	no
1	2	hot	no
1	3	high	no
1	4	weak	no
2	1	sunny	no
2	2	hot	no
2	3	high	no
2	4	strong	no
3	1	overcast	yes
3	2	hot	yes
3	3	high	yes
3	4	weak	yes
...	...	...	...

The inverse transformation is achieved by using the `Val@Col` construct in the head of a rule. For example, the rule of Example 5.4 transforms Table 5.2 back to the corresponding row-oriented representation in Table 5.1.

Example 5.4. *From a column-oriented representation to a row-oriented representation.*

$$\texttt{train}(\texttt{ID}, \texttt{Val@Col}) \texttt{ <- } \texttt{vtrain}(\texttt{ID}, \texttt{Col}, \texttt{Val}).$$

Moreover, we allow extra columns after the `Val@Col` construct to indicate the columns that will be used as labels. For example, the rule of Example 5.5 produces tuples shown in Table 5.3, where for every column of a tuple in Example 5.1 except for the fifth column (i.e., the `PlayTennis` column), it generates a tuple with the tuple ID, column number and value, while the value of the fifth column is displayed in the last column.

Example 5.5. *Obtaining a column-oriented representation with values in the* `PlayTennis` *column as labels.*

$$\begin{aligned}\texttt{vtrain2}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{PlayTennis}) \texttt{ <- } &\texttt{train}(\texttt{ID}, \texttt{Val@Col}, \texttt{Val@5}),\\ &\texttt{PlayTennis} = \texttt{Val@5}.\end{aligned}$$

While the rules above can be used to materialize verticalized tables shown in Table 5.2 and Table 5.3, more often than not, they will be used as virtual views defining some mappings that are then folded into the computation of rollups and data cubes described in the following sections.

In the following, we use k and n to denote the number of columns (excluding the ID column) and the number of tuples in the row-oriented representation, respectively. Then the column-oriented representation `vtrain` contains kn tuples, and there are $(k-1)n$ tuples in `vtrain2`. Assume that every column of a tuple requires the same amount of space to store[2], then we use the product between the number of columns and the number of tuples as the space cost of a relation. For example, the space cost of `train` is $(k+1)n$, and the space costs to materialize `vtrain` and `vtrain2` are $3kn$ and $4(k-1)n$, respectively.

[2]We store each integer or real number using 64 bits, and store a 64-bit pointer for each string, where the pointer points to the actual unique string. In practice, the space required to store these unique strings is relatively small w.r.t. the size of the data set. Thus, it is a fair approximation to assume that every column of a tuple requires the same amount of space to store.

Table 5.4: The count rollup for the example in Table 5.1.

ID	Outlook (1)	Temperature (2)	Humidity (3)	Wind (4)	PlayTennis (5)	count
1	null	null	null	null	null	14
2	overcast	null	null	null	null	4
3	overcast	cool	null	null	null	1
4	overcast	cool	normal	null	null	1
5	overcast	cool	normal	strong	null	1
6	overcast	cool	normal	strong	yes	1
7	overcast	hot	null	null	null	2
8	overcast	hot	high	null	null	1
9	overcast	hot	high	weak	null	1
10	overcast	hot	high	weak	yes	1
11	overcast	mild	null	null	null	1
12	overcast	mild	high	null	null	1
13	overcast	mild	high	strong	null	1
14	overcast	mild	high	strong	yes	1
...	...	...	...	...	...	...

5.3.2 Rollups and Rollup Prefix Tables

Next, we describe a more compact representation that assures low storage and good run time, while preserving intuitive appeal and data independence at the logical level.

A basic SQL-2003 count rollup on our example in Table 5.1 produces the output in Table 5.4 once the result is displayed using ROW_NUMBER, ASC, NULLS FIRST (to save space we only show the first 14 lines among the 51 lines in total).

Now, we see that, in columns 1–5, only the items in the main diagonal hold new information (highlighted in sky blue). The items to left of the diagonal repeat previous values, whereas those to right are nulls. Then we can use a more concise representation such as the one in Table 5.5, where the first four columns contain the same information as an item in the main diagonal does, whereas the last column (PID) specifies the ID of the tuple from where we can find the value of the previous column. We refer to this condensed representation as a *prefix table*. In this particular case it is a *rollup prefix table* for count, and similar representations can be used for other aggregates.

The rollup prefix tables for count are just relational representations for prefix trees, which

Table 5.5: A rollup prefix table for the tuples in Table 5.4.

ID	Col	Val	count	PID
1	1	null	14	1
2	1	overcast	4	1
3	2	cool	1	2
4	3	normal	1	3
5	4	strong	1	4
6	5	yes	1	5
7	2	hot	2	2
8	3	high	1	7
9	4	weak	1	8
10	5	yes	1	9
11	2	mild	1	2
12	3	high	1	11
13	4	strong	1	12
14	5	yes	1	13
...	...	...	...	...

Table 5.6: A logically equivalent rollup prefix table of Table 5.5.

ID	Col	Val	count	PID
1	1	null	14	1
3	1	overcast	4	1
7	2	cool	1	3
7	3	normal	1	7
7	4	strong	1	7
7	5	yes	1	7
3	2	hot	2	3
3	3	high	1	3
3	4	weak	1	3
3	5	yes	1	3
12	2	mild	1	3
12	3	high	1	12
12	4	strong	1	12
12	5	yes	1	12
...	...	...	...	...

proved very valuable in data mining algorithms, such as the famous FP-growth algorithm [H-PY00]. The nodes of rollup prefix tables are identified by integers. In Table 5.5 the integers form a sequence of consecutive integers. However in general, no logical significance needs to be associated to those numbers: the only requirement is that no two nodes are assigned the same identifier, i.e., the combination of (ID, Col, Val, PID) must be unique. In fact, we can reuse the IDs of the original tuples in Table 5.1, and one such example is shown in Table 5.6. An algorithm to construct a rollup prefix table from a verticalized representation $\texttt{vtrain}(\texttt{T}, \texttt{C}, \texttt{V})$ (T, C, and V stand for the ID column, Col column, and Val column, respectively) is presented in the example below.[3]

Example 5.6. *Rollup prefix tables. Given two IDs* T1 *and* T2*, we say that the tuple with ID* T1 *can* represent *the tuple with ID* T2 *(or* T1 *can represent* T2 *for short) in column* C *if both tuples are identical in the first* C *columns.* repr *is a recursive relation that represents* vtrain *in a different format, where each tuple* $(\texttt{T}, \texttt{C}, \texttt{V})$ *in* vtrain *is augmented with one more column* T1 *indicating that* T1 *can represent* T *in column* $\texttt{C} - 1$*, i.e., the parent ID of the current tuple is* T1*. Then a prefix table* rupt *is constructed on top of* repr *in* $r3$*, where among all the tuples with the same parent ID* Ta*,*

[3]Hereafter, a rollup prefix table does not contain the tuple that represents the total number of tuples, i.e., the tuple (1, 1, null, 14, 1) in Table 5.5 and Table 5.6.

and the same value V *in column* C*, the tuple with the minimal ID* T *is selected as a* representative *by the monotonic aggregate* mmin*, and the number of tuples that satisfy this property is tracked by the other monotonic aggregate* mcount*. The recursive computation for* repr *is performed as follows. The exit rule* $r1$ *initializes* repr *by setting that all the tuples have the parent ID 1 in column 1. Then the recursive rule* $r2$ *is repeatedly evaluated until no more tuples can be added to* repr*, where for each tuple* $(\mathtt{T}, \mathtt{C}, \mathtt{V})$ *in* vtrain*, if the parent ID of the tuple with ID* T *is* Ta *in column* $\mathtt{C} - 1$*, and the tuple with ID* T1 *is selected as the current representative, then* T1 *is the parent ID for the tuple with ID* T *in column* C*.*

$$r1.\ \mathtt{repr}(\mathtt{T1}, \mathtt{C}, \mathtt{V}, \mathtt{T}) \text{ <- } \mathtt{vtrain}(\mathtt{T}, \mathtt{C}, \mathtt{V}), \mathtt{C} = 1, \mathtt{T1} = 1.$$

$$r2.\ \mathtt{repr}(\mathtt{T1}, \mathtt{C}, \mathtt{V}, \mathtt{T}) \text{ <- } \mathtt{vtrain}(\mathtt{T}, \mathtt{C}, \mathtt{V}), \mathtt{C1} = \mathtt{C} - 1, \mathtt{repr}(\mathtt{Ta}, \mathtt{C1}, \mathtt{V1}, \mathtt{T}), \mathtt{rupt}(\mathtt{T1}, \mathtt{C1}, \mathtt{V1}, \_, \mathtt{Ta}).$$

$$r3.\ \mathtt{rupt}(\mathsf{mmin}\langle\mathtt{T}\rangle, \mathtt{C}, \mathtt{V}, \mathsf{mcount}\langle\mathtt{T}\rangle, \mathtt{Ta}) \text{ <- } \mathtt{repr}(\mathtt{Ta}, \mathtt{C}, \mathtt{V}, \mathtt{T}).$$

The rule below is the extraction rule that gets the final values of the monotonic aggregates for each unique combination of $(\mathtt{Ta}, \mathtt{C}, \mathtt{V})$*.*

$$r4.\ \mathtt{rupt2}(\mathsf{min}\langle\mathtt{T}\rangle, \mathtt{C}, \mathtt{V}, \mathsf{max}\langle\mathtt{CT}\rangle, \mathtt{Ta}) \text{ <- } \mathtt{rupt}(\mathtt{T}, \mathtt{C}, \mathtt{V}, \mathtt{CT}, \mathtt{Ta}).$$

Note that rupt2 *might contain more tuples than the final rollup prefix table since (i) the continuous monotonic aggregates might produce multiple tuples of the form* $(\mathtt{T}, \mathtt{C}, \mathtt{V}, \mathtt{CT}, \mathtt{Ta})$ *for each unique combination of* $(\mathtt{Ta}, \mathtt{C}, \mathtt{V})$*, and (ii) these* T *values then become the parent IDs of some tuples in* repr*,* rupt*, and* rupt2*. However, it is easy to see that these parent IDs share the same* T *value in* rupt2*. Thus,* $r5$ *is added to keep the minimal parent ID* Ta*, i.e., to keep a unique parent for every node. This rule is not necessary if the tuples in* vtrain *are accessed in increasing order of IDs. This condition is guaranteed since* vtrain *is produced by our built-in verticalization function.*

$$r5.\ \mathtt{rupt3}(\mathtt{T}, \mathtt{C}, \mathtt{V}, \mathtt{CT}, \mathtt{Ta}) \text{ <- } \mathtt{rupt2}(\mathtt{T}, \mathtt{C}, \mathtt{V}, \mathtt{CT}, \mathtt{Ta}), \mathsf{is\_min}((\mathtt{T}, \mathtt{C}, \mathtt{V}), (\mathtt{Ta})).$$

Performance Optimizations. In practice, vtrain is usually a very large relation, and we prefer

access it with sequential scans to index lookups since the cost of indexing is very high. Thus, one full scan to `vtrain` is performed in each iteration of the semi-naive evaluation. The semi-naive evaluation requires $k + 1$ iterations, where the i-th iteration derives new tuples based on tuples of the form $(\_, i, \_)$ in `vtrain`, i.e., it creates the i-th level of the prefix table. The main reason that leads to this behavior is due to the design of the semi-naive evaluation, which only uses tuples derived in the previous iterations to derive new tuples in one iteration. For example, assume that we start the second iteration of the evaluation, where all the tuples in `repr` and `rupt` are in the form of $(\_, 1, \_, \_)$ and $(\_, 1, \_, \_, \_)$, respectively. We read the first tuple $(1, 1, \text{sunny})$ in `vtrain`, but do not derive anything since there is no tuple in `repr` in the form of $(\_, 0, \_, \_)$. Then we read the second tuple $(1, 2, \text{hot})$, find $(1, 1, \text{sunny}, 1)$ in `repr`, find $(1, \text{sunny}, 1, \_, 1)$ in `rupt`, and derive $(1, 2, \text{hot}, 1)$ for `repr`. After that, we move to the third tuple $(1, 3, \text{high})$, but do not derive anything since the tuple $(1, 2, \text{hot}, 1)$ in `repr` cannot be used until the next iteration. We can reduce the number of iterations by allowing the evaluation to use all the tuples derived so far. Such an evaluation strategy is called the *eager semi-naive evaluation*. Thus, with eager semi-naive evaluation, one scan of `vtrain` is performed in the evaluation of *r1*, and two scans of `vtrain` are performed in the evaluation of *r2*—all the tuples are derived during the first scan, and the evaluation terminates when it completes the second scans since it identifies that no new tuple is derived during this scan. We can further eliminate the scan required by the exit rule by replacing *r1* and *r2* with the following rule:

$$
\begin{aligned}
\textit{r6}.\ \texttt{repr}(\texttt{T1}, \texttt{C}, \texttt{V}, \texttt{T}) \texttt{<-}\ & \texttt{vtrain}(\texttt{T}, \texttt{C}, \texttt{V}), \\
& \texttt{if}(\texttt{C} = 1 \\
& \quad \texttt{then}\ \texttt{T1} = 1 \\
& \quad \texttt{else}\ \texttt{C1} = \texttt{C} - 1, \texttt{repr}(\texttt{Ta}, \texttt{C1}, \texttt{V1}, \texttt{T}), \\
& \qquad \texttt{rupt}(\texttt{T1}, \texttt{C1}, \texttt{V1}, \_, \texttt{Ta}).
\end{aligned}
$$

The expression $\texttt{h} \texttt{<-}\ \texttt{if}(\texttt{p}\ \texttt{then}\ \texttt{q}\ \texttt{else}\ \texttt{w})$ is equivalent to the following two rules:

$$
\begin{aligned}
& \texttt{h} \texttt{<-}\ \texttt{p}, \texttt{q}. \\
& \texttt{h} \texttt{<-}\ \neg\texttt{p}, \texttt{w}.
\end{aligned}
$$

Thus, the new rule $r6$ eliminates the exit rule, and only two scans of `vtrain` are required in total.

Space Optimizations. Both `repr` and `rupt` are recursive relations, and are materialized during the evaluation. The purpose to materialize a recursive relation is to avoid redundant computation and ensure the termination of computation. However, the space cost can be very expensive. In Example 5.6, `repr` contains the same number of tuples as `vtrain` does. Thus, the space cost to materialize `repr` is $4kn$. On the other hand, the space cost of to materialize `vtrain` is only $3kn$. Although we try to reduce the space cost by not materializing `vtrain`, the computation produces an intermediate relation that is actually larger than the space we tried to save. For this program, it is not necessary to materialize the full `repr` relation during the evaluation. Let $\texttt{repr}_i$ be the subset of `repr` that contains all the tuples of the form $(\_, i, \_, \_)$. During the i-th ($i > 1$) iteration of the standard semi-naive evaluation, it is easy to see that only tuples in $\texttt{repr}_{i-1}$ will be accessed to derived tuples in $\texttt{repr}_i$. Thus, it is only necessary to keep the tuples in $\texttt{repr}_{i-1}$ and $\texttt{repr}_i$ during the i-th iteration, and the tuples in $\texttt{repr}_{i-2}, \ldots, \texttt{repr}_1$ can be removed. In this way, the space cost to materialize `repr` is reduced from $4kn$ to $4 \times 2n$. Moreover, if the eager semi-naive evaluation is used, the space cost can be further reduced to a constant. At any time of the evaluation, it is only necessary to keep two tuples in `repr`, where one tuple T_1 is used to derive the other tuple T_2, T_1 is removed after T_2 has been derived, and the newly derived tuple T_2 is then immediately used to derive another tuple[4]. Thus, the space cost to materialize `repr` is only $4 \times 2 = 8$ in the eager semi-naive evaluation. We have focused on the space cost to materialize `repr`. The materialization of `rupt` is unavoidable since it represents our final prefix table. Moreover, no extra space is used for `rupt2` since $r4$ is an extraction rule implemented as reading the final values of the monotonic aggregates from `rupt`.

Thus, we have come up with a Datalog program that achieves the same time and space complexity as a fully optimized procedure program for constructing a rollup prefix table. To enhance usability, we provide the following construct:

$$\texttt{myrupt}(\texttt{rollup}\langle\texttt{ID}\rangle, \texttt{Col}, \texttt{Val}, \texttt{count}\langle\texttt{ID}\rangle, \texttt{prev}\langle\texttt{ID}\rangle) \texttt{ <- } \texttt{train}(\texttt{ID}, \texttt{Val@Col}).$$

[4]No tuple from `repr` is used in the derivation when the condition $\texttt{C} = 1$ in `if` is satisfied.

In this built-in construct, the keyword rollup indicates that we are computing a rollup prefix table, rollup⟨ID⟩ denotes the operation that selects a representative among all the tuples with the same combination of (PID, Col, Val), and prev⟨ID⟩ denotes the corresponding operation that decides the PID of a tuple based on the actual implementation of rollup⟨ID⟩. We have used the mmin/min aggregates to implement rollup⟨ID⟩ in Example 5.6. Other constructs, such as mmax/max and choice, could also be used to deliver logically equivalent rollup prefix tables.

Finally, we introduce a self-explanatory horizontal representation, called *compact rollups*, to provide a more intuitive visualization for rollup prefix tables. An example of a compact rollup is shown in Table 5.7. In this representation, each item e that is not under the ID column and is not empty represents a tuple in the rollup prefix table, where the values for ID, Col, Val, count columns are the tuple ID of e, e's column number, e's value, and the number associated with e's value, respectively, and the value of the PID column is determined as follows: if the column number of e equals 1, then PID equals 1; if the item before e is not empty, then PID equals ID; otherwise PID equals the tuple ID of the first non-empty item above e. The compact rollup in Table 5.7 represents the 51 tuples in Table 5.4 (excluding the first tuple that shows the total number of tuples) by only 14 tuples. It is easy to see that this representation displays the same information in a more concise and appealing way to end-users w.r.t. the typical rollup output format displayed in Table 5.4.

5.4 Data Cubes

The representations used in rollup prefix tables and compact rollups are also good for representing data cubes. Example 5.7 shows a program that computes a data cube from a rollup prefix table myrupt, where the data cube also uses a prefix table representation that we refer to as a *cube prefix table*. Note that a cube prefix table is also a prefix tree. To simplify the discussion, we will use terms that are specifically for trees, i.e., we use node, root, and leaf to refer to a specific tuple, and use path to refer to a collection of tuples that forms a path.

Example 5.7. *From a rollup prefix table to a data cube. Predicates* crepr *and* cpt *correspond to* repr *and* rupt *in Example 5.6, respectively. Let* $\texttt{crepr}_i$ *be the set of tuples in* crepr *with the*

Table 5.7: A compact rollup for the example in Table 5.1.

ID	**Outlook**	C1	**Tempe rature**	C2	**Humidity**	C3	**Wind**	C4	**Play Tennis**	C5
1	sunny	5	hot	2	high	2	weak	1	no	1
2							strong	1	no	1
8			mild	2	high	1	weak	1	no	1
11					normal	1	strong	1	yes	1
9			cool	1	normal	1	weak	1	yes	1
3	overcast	4	hot	2	high	1	weak	1	yes	1
13					normal	1	weak	1	yes	1
7			cool	1	normal	1	strong	1	yes	1
12			mild	1	high	1	strong	1	yes	1
4	rain	5	mild	3	high	2	weak	1	yes	1
14							strong	1	no	1
10					normal	1	weak	1	yes	1
5			cool	2	normal	2	weak	1	yes	1
6							strong	1	no	1

value of the first column being i. Similarly for $\texttt{cpt}_i$. $\texttt{cpt}_i$ *stores a subset of the data cube as a prefix tree, where the value of the i-th column is null, and* $\texttt{crepr}_i$ *stores the intermediate results that lead to* $\texttt{cpt}_i$.

Let k be the number of columns in `train`. *The recursive computation of* `crepr` *and* `cpt` *is guided by the predicate* `step`, *where for each (i, j) pair such that $1 \leq i < j \leq k$, for each path from the root to a leaf in* $\texttt{cpt}_i$, *a new path is constructed by replacing the value of the j-th level with null, and the new path is then added to* $\texttt{crepr}_j$ *and* $\texttt{cpt}_j$.

$$r1.\ \texttt{cpt}(0, \texttt{mmin}\langle \texttt{T}\rangle, \texttt{C}, \texttt{V}, \texttt{msum}\langle(\texttt{T}, \texttt{CT}), \texttt{Ta}\rangle) \texttt{<-}\ \texttt{myrupt}(\texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}).$$

$$\begin{aligned} r2.\ \texttt{crepr}(\texttt{X}, \texttt{T2}, \texttt{C}, \texttt{V1}, \texttt{T1}, \texttt{CT}) \texttt{<-}\ & \texttt{step}(\texttt{X}, \texttt{Y}, \texttt{MT}, \_), \texttt{cpt}(\texttt{Y}, \texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}), \texttt{C} >= \texttt{X}, \\ & \texttt{if}(\texttt{C} = \texttt{X} \\ & \quad \texttt{then}\ \texttt{T2} = \texttt{Ta}, \texttt{V1} = \texttt{null} \\ & \quad \texttt{else}\ \texttt{C1} = \texttt{C} - 1, \texttt{Ta1} = \texttt{MT} + \texttt{Ta}, \\ & \qquad \texttt{crepr}(\texttt{X}, \texttt{Ta2}, \texttt{C1}, \texttt{V2}, \texttt{Ta1}, \_), \\ & \qquad \texttt{cpt}(\texttt{X}, \texttt{T2}, \texttt{C1}, \texttt{V2}, \_, \texttt{Ta2}), \texttt{V1} = \texttt{V}), \\ & \texttt{T1} = \texttt{MT} + \texttt{T}. \end{aligned}$$

$$r3.\ \texttt{cpt}(\texttt{X}, \texttt{mmin}\langle \texttt{T}\rangle, \texttt{C}, \texttt{V}, \texttt{msum}\langle(\texttt{T}, \texttt{CT})\rangle, \texttt{Ta}) \texttt{<-}\ \texttt{crepr}(\texttt{X}, \texttt{Ta}, \texttt{C}, \texttt{V}, \texttt{T}, \texttt{CT}).$$

Rule $r4$ extracts the final aggregate values from cpt*, and rule $r5$ obtains the final data cube by projecting out the first column.*

$$r4.\ \texttt{cpt2}(\texttt{X}, \mathsf{min}\langle\texttt{T}\rangle, \texttt{C}, \texttt{V}, \mathsf{max}\langle\texttt{CT}\rangle, \texttt{Ta}) \texttt{<-}\ \texttt{cpt}(\texttt{X}, \texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}).$$
$$r5.\ \texttt{cpt3}(\texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}) \texttt{<-}\ \texttt{cpt2}(\_, \texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}).$$

The rules below construct step *recursively.*

$$r6.\ \texttt{stat}(\mathsf{max}\langle\texttt{T}\rangle, \mathsf{max}\langle\texttt{C}\rangle) \texttt{<-}\ \texttt{vtrain}(\texttt{T}, \texttt{C}, \texttt{V}).$$
$$r7.\ \texttt{step}(1, 0, \texttt{MT}, \texttt{MC}) \texttt{<-}\ \texttt{stat}(\texttt{MT}, \texttt{MC}).$$
$$r8.\ \texttt{step}(\texttt{X1}, \texttt{Y1}, \texttt{MT1}, \texttt{MC}) \texttt{<-}\ \texttt{step}(\texttt{X}, \texttt{Y}, \texttt{MT}, \texttt{MC}),$$
$$\texttt{if}(\texttt{X} = \texttt{Y} + 1$$
$$\texttt{then}\ \texttt{X} < \texttt{MC}, \texttt{X1} = \texttt{X} + 1, \texttt{Y1} = 0, \texttt{MT1} = \texttt{MT} * 2$$
$$\texttt{else}\ \texttt{X1} = \texttt{X}, \texttt{Y1} = \texttt{Y} + 1, \texttt{Y1} < \texttt{X}, \texttt{MT1} = \texttt{MT}).$$

Now we show some intermediate results obtained during the semi-naive evaluation of the above program. These results are displayed as *compact cubes*, which uses a representation that is similar to the one used in compact rollups. We display a value in the ID column in the format of $\texttt{T}(+\texttt{MT})$ indicating that the ID value is $\texttt{T} + \texttt{MT}$, and an item in this row is derived from an item in the row with ID T. In Table 5.8, the items that are marked in sky blue represent nodes in the data cube that are derived in the second iteration of the semi-naive evaluation, and the remaining items are copied from the rollup prefix table during the first iteration where the exit rule $r1$ is evaluated. With all the tuples in Table 5.8, the third iteration of the semi-naive evaluation leads to a compact cube shown in Table 5.9, where the newly derived items are marked in sky blue.

Note that the performance and space optimizations presented for Example 5.6 are all applicable to the program of Example 5.7. We introduce the following construct to denote the computation of a data cube directly from train, where $\mathsf{cube}\langle\texttt{ID}\rangle$ tells the compiler that we are dealing with a data cube.

$$\texttt{mycpt}(\mathsf{cube}\langle\texttt{ID}\rangle, \texttt{Col}, \texttt{Val}, \mathsf{count}\langle\texttt{ID}\rangle, \mathsf{prev}\langle\texttt{ID}\rangle) \texttt{<-}\ \texttt{train}(\texttt{ID}, \texttt{Val@Col}).$$

Table 5.8: The compact cube for the example in Table 5.1 obtained after the 2nd iteration of the semi-naive evaluation of the program in Example 5.7.

ID	**Outlook**	C1	**Temperature**	C2	**Humidity**	C3	**Wind**	C4	**Play Tennis**	C5
1	sunny	5	hot	2	high	2	weak	1	no	1
2							strong	1	no	1
8			mild	2	high	1	weak	1	no	1
11					normal	1	strong	1	yes	1
9			cool	1	normal	1	weak	1	yes	1
3	overcast	4	hot	2	high	1	weak	1	yes	1
13					normal	1	weak	1	yes	1
7			cool	1	normal	1	strong	1	yes	1
12			mild	1	high	1	strong	1	yes	1
4	rain	5	mild	3	high	2	weak	1	yes	1
14							strong	1	no	1
10					normal	1	weak	1	yes	1
5			cool	2	normal	2	weak	1	yes	1
6							strong	1	no	1
1(+14)	null	14	hot	4	high	3	weak	2	no	1
3(+14)									yes	1
2(+14)							strong	1	no	1
13(+14)					normal	1	weak	1	yes	1
4(+14)			mild	6	high	4	weak	2	yes	1
8(+14)									no	1
12(+14)							strong	2	yes	1
14(+14)									no	1
10(+14)					normal	2	weak	1	yes	1
11(+14)							strong	1	yes	1
5(+14)			cool	4	normal	4	weak	2	yes	2
6(+14)							strong	2	no	1
7(+14)									yes	1

Iceberg Cubes. The above cube construct can be used to define *iceberg cubes* [FSG98], which compute aggregate functions over certain set of attributes to find aggregate values above predefined thresholds. For example, an iceberg cube that finds all the nodes with count values no less than a predefined threshold can be defined by the following rule:

$$\texttt{icebergcpt}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{Cnt}, \texttt{PID}) \texttt{ <- } \texttt{mycpt}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{Cnt}, \texttt{PID}),$$
$$\texttt{threshold}(\texttt{T}), \texttt{Cnt} \geq \texttt{T}.$$

Our algorithm computes the iceberg cube by constructing a cube prefix table from the rollup prefix

Table 5.9: The compact cube for the example in Table 5.1 obtained after the 3rd iteration of the semi-naive evaluation of the program in Example 5.7.

ID	**Outlook**	C1	**Tempe rature**	C2	**Humidity**	C3	**Wind**	C4	**Play Tennis**	C5
1	sunny	5	hot	2	high	2	weak	1	no	1
2							strong	1	no	1
8			mild	2	high	1	weak	1	no	1
11					normal	1	strong	1	yes	1
9			cool	1	normal	1	weak	1	yes	1
1(+28)			null	5	high	3	weak	2	no	2
2(+28)							strong	1	no	1
9(+28)					normal	2	weak	1	yes	1
11(+28)							strong	1	yes	1
3	overcast	4	hot	2	high	1	weak	1	yes	1
13					normal	1	weak	1	yes	1
7			cool	1	normal	1	strong	1	yes	1
12			mild	1	high	1	strong	1	yes	1
3(+28)			null	4	high	2	weak	1	yes	1
12(+28)							strong	1	yes	1
7(+28)					normal	2	strong	1	yes	1
13(+28)							weak	1	yes	1
4	rain	5	mild	3	high	2	weak	1	yes	1
14							strong	1	no	1
10					normal	1	weak	1	yes	1
5			cool	2	normal	2	weak	1	yes	1
6							strong	1	no	1
4(+28)			null	5	high	2	weak	1	yes	1
14(+28)							strong	1	no	1
5(+28)					normal	3	weak	2	yes	2
6(+28)							strong	1	no	1
1(+14)	null	14	hot	4	high	3	weak	2	no	1
3(+14)									yes	1
2(+14)							strong	1	no	1
13(+14)					normal	1	weak	1	yes	1
4(+14)			mild	6	high	4	weak	2	yes	1
8(+14)									no	1
12(+14)							strong	2	yes	1
14(+14)									no	1
10(+14)					normal	2	weak	1	yes	1
11(+14)							strong	1	yes	1
5(+14)			cool	4	normal	4	weak	2	yes	2
6(+14)							strong	2	no	1
7(+14)									yes	1

continued on the next page . . .

Table 5.9 (continued)

ID	Outlook C1	Temperature C2	Humidity C3	Wind C4	Play Tennis C5
1(+42)		null 14	high 7	weak 4	no 2
3(+42)					yes 2
2(+42)				strong 3	no 2
12(+42)					yes 1
5(+42)			normal 7	weak 4	yes 4
6(+42)				strong 3	no 1
7(+42)					yes 2

table representation of the data set, and checking if the minimal support constraint is satisfied for each node in the cube immediately after the corresponding tuple is derived.

5.5 Association Rule Mining

This type of tasks focus on discovering associations among items in a large transactional data set. Given a set of items $I = \{i_1, \ldots, i_k\}$, and a list of transactions $D = [t_1, \ldots, t_n]$, where each transaction t_i has a unique transaction ID and contains a subset of items in I, an association rule is a pattern of the following form:

$$X \Rightarrow Y, \text{ where } X, Y \subseteq I \text{ and } X \cap Y = \emptyset,$$

indicating that if the items in X are bought, then the items in Y are also bought. For example, $\{\text{beer}\} \Rightarrow \{\text{diapers}\}$ means if a customer buys beer, he/she also buys diapers.

The *support* of an association rule is the frequency of $X \cup Y$ appearing in D, and the *confidence* is the proportion of the transactions that contains X which also contains Y. Typically, we are looking for association rules that satisfy both a minimum support constraint (min_sup) and a minimum confidence constraint (min_conf). The mining process consists of two steps. We start with finding all the itemsets that are subset of I and appear in at least $min\_sup \times n$ transactions. Then, for each itemset discovered in the first step, we enumerate all possible ways of splitting the set into two subsets X and Y and check whether the corresponding association rules satisfy the minimum confidence constraint. The first step of this mining process is elaborated in more detail

Table 5.10: List of transactions.

ID	Items
1	{I1, I2, I5}
2	{I2, I4}
3	{I2, I3}
4	{I1, I2, I4}
5	{I1, I3}
6	{I2, I3}
7	{I1, I3}
8	{I1, I2, I3, I5}
9	{I1, I2, I3}

Table 5.11: A table representation of transactions in Table 5.10.

ID	I1	I2	I3	I4	I5
1	yes	yes	no	no	yes
2	no	yes	no	yes	no
3	no	yes	yes	no	no
4	yes	yes	no	yes	no
5	yes	no	yes	no	no
6	no	yes	yes	no	no
7	yes	no	yes	no	no
8	yes	yes	yes	no	yes
9	yes	yes	yes	no	no

Table 5.12: A compact rollup for the tuples in Table 5.11.

ID	I1	C1	I2	C2	I3	C3	I4	C4	I5	C5
1	yes	6	yes	4	no	2	no	1	yes	1
4							yes	1	no	1
8					yes	2	no	2	yes	1
9									no	1
5			no	2	yes	2	no	2	no	2
2	no	3	yes	3	no	1	yes	1	no	1
3					yes	2	no	2	no	2

in Example 5.8.

Example 5.8. *Mining frequent itemsets. Given a list of transactions as shown in Table 5.10, we can represent the transactions as a table* `trans` *as shown in Table 5.11, where a "yes"/"no" value under a column indicates whether a transaction contains the corresponding item or not. The rule below converts the table into a compact rollup as shown in Table 5.12:*

$$\texttt{transrupt}(\mathsf{rollup}\langle\texttt{ID}\rangle, \texttt{Col}, \texttt{Val}, \mathsf{count}\langle\texttt{ID}\rangle, \mathsf{prev}\langle\texttt{ID}\rangle) \texttt{ <- } \texttt{trans}(\texttt{ID}, \texttt{Val@Col}).$$

Furthermore, with the cube construct introduced previously, the rule below produces a compact cube as shown in Table 5.13:

$$\texttt{transcpt}(\mathsf{cube}\langle\texttt{ID}\rangle, \texttt{Col}, \texttt{Val}, \mathsf{count}\langle\texttt{ID}\rangle, \mathsf{prev}\langle\texttt{ID}\rangle) \texttt{ <- } \texttt{trans}(\texttt{ID}, \texttt{Val@Col}).$$

Table 5.13: A compact cube for the tuples in Table 5.11.

ID	I1	C1	I2	C2	I3	C3	I4	C4	I5	C5
1	yes	6	yes	4	no	2	no	1	yes	1
4							yes	1	no	1
1(+72)							null	2	yes	1
4(+72)									no	1
8					yes	2	no	2	yes	1
9									no	1
8(+72)							null	2	yes	1
9(+72)									no	1
1(+36)					null	4	no	3	yes	2
9(+36)									no	1
4(+36)							yes	1	no	1
1(+108)							null	4	yes	2
4(+108)									no	2
5			no	2	yes	2	no	2	no	2
5(+72)							null	2	no	2
5(+36)					null	2	no	2	no	2
5(+108)							null	2	no	2
1(+18)			null	6	no	2	no	1	yes	1
4(+18)							yes	1	no	1
1(+90)							null	2	yes	1
4(+90)									no	1
5(+18)					yes	4	no	4	no	3
8(+18)									yes	1
5(+90)							null	4	no	3
8(+90)									yes	1
1(+54)					null	6	no	5	yes	2
5(+54)									no	3
4(+54)							yes	1	no	1
1(+126)							null	6	yes	2
4(+126)									no	4
2	no	3	yes	3	no	1	yes	1	no	1
2(+72)							null	1	no	1
3					yes	2	no	2	no	2
3(+72)							null	2	no	2
2(+36)					null	3	yes	1	no	1
3(+36)							no	2	no	2
2(+108)							null	3	no	3
2(+18)			null	3	no	1	yes	1	no	1
2(+90)							null	1	no	1
3(+18)					yes	2	no	2	no	2
3(+90)							null	2	no	2
2(+54)					null	3	yes	1	no	1
3(+54)							no	2	no	2
2(+126)							null	3	no	3

continued on the next page . . .

Table 5.13 (continued)

ID	I1	C1	I2	C2	I3	C3	I4	C4	I5	C5
1(+9)	null	9	yes	7	no	3	no	1	yes	1
2(+9)							yes	2	no	2
1(+81)							null	3	yes	1
2(+81)									no	2
3(+9)					yes	4	no	4	no	3
8(+9)									yes	1
3(+81)							null	4	no	3
8(+81)									yes	1
1(+45)					null	7	no	5	yes	2
3(+45)									no	3
2(+45)							yes	2	no	2
1(+117)							null	7	yes	2
2(+117)									no	5
5(+9)			no	2	yes	2	no	2	no	2
5(+81)							null	2	no	2
5(+45)					null	2	no	2	no	2
5(+117)							null	2	no	2
1(+27)			null	9	no	3	no	1	yes	1
2(+27)							yes	2	no	2
1(+99)							null	3	yes	1
2(+99)									no	2
3(+27)					yes	6	no	6	no	5
8(+27)									yes	1
3(+99)							null	6	no	5
8(+99)									yes	1
1(+63)					null	9	no	7	yes	2
3(+63)									no	5
2(+63)							yes	2	no	2
1(+135)							null	9	yes	2
2(+135)									no	7

In Table 5.13, each path that starts from the root represents an itemset, and the count value associated with the end node indicates the number of occurrences for this path. However, different paths can represent the same itemset, but not all count values represent the total occurrences of this itemset. The itemsets together with their total occurrences can be extracted with the rules below, where each path that starts from the root, ends at a "yes" node, and does not contain any internal "no" node, is kept (the end nodes of all the paths kept are marked in sky blue in Table 5.13). For example, the path starting from the root ending at the node with ID 80 and i_5 column represents the itemset $\{i_1, i_2, i_3, i_5\}$ with frequency 1.

$$\texttt{itemset}(\texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}, \texttt{C}, \texttt{T}, []) \texttt{<-} \texttt{transcpt}(\texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}), \texttt{V} = \texttt{yes}.$$

$$\begin{aligned}\texttt{itemset}(\texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}, \texttt{C2}, \texttt{Ta1}, \texttt{L1}) \texttt{<-}\ & \texttt{itemset}(\texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}, \texttt{C1}, \texttt{T1}, \texttt{L}), \\ & \texttt{C1} > 0, \texttt{C2} = \texttt{C1} - 1, \\ & \texttt{if}(\texttt{C1} = \texttt{C}\ \texttt{then}\ \texttt{Ta1} = \texttt{Ta}, \texttt{V1} = \texttt{V} \\ & \quad \texttt{else}\ \texttt{transcpt}(\texttt{T1}, \texttt{C1}, \texttt{V1}, \_, \texttt{Ta1}), \texttt{V1} \neq \texttt{no}), \\ & \texttt{if}(\texttt{V1} = \texttt{yes}\ \texttt{then}\ \texttt{L1} = [\texttt{C1}|\texttt{L}]\ \texttt{else}\ \texttt{L1} = \texttt{L}).\end{aligned}$$

$$\texttt{itemset2}(\texttt{L}, \texttt{CT}) \texttt{<-} \texttt{itemset}(\texttt{T}, \texttt{C}, \texttt{V}, \texttt{CT}, \texttt{Ta}, \texttt{C1}, \_, \texttt{L}), \texttt{C1} = 0.$$

Finally, we can find all the itemsets that satisfy the minimum support constraint as follows:

$$\texttt{transcnt}(\texttt{count}\langle\texttt{ID}\rangle) \texttt{<-} \texttt{trans}(\texttt{ID}, \texttt{Val@Col}), \texttt{Col} = 1.$$

$$\begin{aligned}\texttt{freqitemset}(\texttt{L}, \texttt{CT}) \texttt{<-}\ & \texttt{itemset2}(\texttt{L}, \texttt{CT}), \texttt{transcnt}(\texttt{C}), \texttt{threshold}(\texttt{MC}, \texttt{MS}), \\ & \texttt{CT} \geq \texttt{C} * \texttt{MS}.\end{aligned}$$

Performance and Space Optimizations. The space cost of the table representation shown in Table 5.11 is $3kn$, where k is the number of unique items, and n is the number of transactions. Let m be the total number of items in these n transactions. In practice, the average number of items per transaction $c = m/n$ is usually a small number such that $c \ll k$. It is not hard to see that the prefix table representation of the transactions contains m tuples, and its space cost is $5m$. So far, the actual space cost is only $5m$ since the table representation of transactions is not materialized. However, the cost to compute the data cube is prohibitive—it creates at least 2^k tuples in `basketcpt` with at least 2^k derivations, and both the time and space requirements are extremely high for a small value of k. In fact, the computation produces way more than 2^k tuples, e.g., the value of 2^k is only 32 when k equals 5, but there are 215 tuples in `basketcpt` as shown in Table 5.13. On the other hand, only a small fraction of tuples are kept with the rules that derive `itemset2`, e.g., only 34 tuples in Table 5.13 that represent 19 itemsets are kept. The main reason that leads to this gap is due to the representation used to represent transactions and paths, where each transaction or path has all k columns and the value of each column can be yes, no, or null. In this representation, both no and null indicate that a certain item does not appear in an itemset. For many items with null values in

Table 5.13, the corresponding count value of such an item is the same as that of the item with no value that derives it. For example, the item with ID 41 column I4 is derived from the item with ID 5 column I4, and both items have the same count value 2. Since these items with null values do not hold new information w.r.t. the items with no values, the key to optimize the computation and space costs is to use an appropriate representation that can avoid these duplicates.

Our solution is to use a variable-length representation for the transactions and paths as shown in Table 5.14 and Table 5.15, respectively. In the table representation of transactions shown in Table 5.14, a tuple of the form (ID, Col, Val) indicates that the Col-th item in the transaction ID is Val. Then we obtain a rollup prefix table shown in Table 5.15 using a program similar to that of Example 5.6. The prefix tree represented by the current rollup prefix table is different from the one in Table 5.12—instead of having a fixed depth for every leaf, the depth of a leaf varies from 2 to 4. On top of this prefix table, we can compute the data cube using the same computation described in Example 5.7, i.e., in the i-th iteration of the semi-naive evaluation, we replace the value of the i-th column with null for every path derived in the previous iterations. However, the same logical operation is implemented via a different set of physical operations since the underlying representation of paths is different. In the i-th iteration of the semi-naive evaluation, for each path derived in the previous iterations containing a value Ii, i.e., there is a tuple N of the form $(\texttt{X}, \texttt{T}, \texttt{C}, \text{I}i, \texttt{CT}, \texttt{Ta})$ in the current cube prefix table, the following two steps are performed:

1. For each tuple C that represents a child of N, it is in the form of $(\texttt{X}, \texttt{T1}, \texttt{C}+1, \text{I}j, \texttt{CT1}, \texttt{T})$, and we create a new subtree with a new tuple $(i, \texttt{T1} + 2^{i-1}n, \texttt{C}, \text{I}j, \texttt{CT1}, \texttt{Ta} + 2^{i-1}n)$ as root by copying the subtree with C as root.

2. For each tuple S that represents a sibling of N that is in the form of $(\texttt{X}, \texttt{T1}, \texttt{C}, \text{I}j, \texttt{CT1}, \texttt{Ta})$ with $j > i$, we create a new subtree with a new tuple $(i, \texttt{T1} + 2^{i-1}n, \texttt{C}, \text{I}j, \texttt{CT1}, \texttt{Ta} + 2^{i-1}n)$ as root by copying the subtree with S as root.

The monotonic aggregates are responsible for finding the common patterns among these subtrees and keep track of the number of paths that share the same pattern. Note that we have also removed the null values in this representation. Now if we need to reconstruct a complete path, we can

(i) start from an end node, (ii) traverse backward toward the root until we cannot find a tuple of the form $(i, \texttt{T1}, \texttt{C}, \_, \_, \_)$, (iii) recover the next tuple by retrieving the tuple of the form $(\_, \texttt{T1} - 2^{i-1}n, \texttt{C}, \texttt{I}i, \_, \_)$, and (iv) continue the traversal until we reach the root.

Table 5.14: A verticalized table representation of transactions in Table 5.10.

ID	**Col**	**Val**
1	1	I1
1	2	I2
1	3	I5
2	1	I2
2	2	I4
3	1	I2
3	2	I3
4	1	I1

ID	**Col**	**Val**
4	2	I2
4	3	I4
5	1	I1
5	2	I3
6	1	I2
6	2	I3
7	1	I1
7	2	I3

ID	**Col**	**Val**
8	1	I1
8	2	I2
8	3	I3
8	4	I5
9	1	I1
9	2	I2
9	3	I3

Table 5.15: A compact rollup for the tuples in Table 5.14.

ID	**1**	C1	**2**	C2	**3**	C3	**4**	C4
1	I1	6	I2	4	I5	1		
4					I4	1		
8					I3	2	I5	1
5			I3	2				
2	I2	3	I4	1				
3			I3	2				

Moreover, we can further reduce the space cost by skipping the operations in Step (2) when no subtree is created in Step (1) since the subtrees created in Step (2) do not carry any new information w.r.t. existing subtrees in such a situation. The data cube obtained from this computation is shown in Table 5.16. There are only 34 tuples, and we mark 19 tuples in sky blue, where the path starting from the root and ending at a marked node corresponds to an itemset, and the count value associated with the end node indicates the number of occurrences of the itemset. It is clear to see the space saving achieved through this representation w.r.t. the representation used in Table 5.13—the number of tuples in the data cube is reduced to 34 from 215, and only 19 tuples (instead of 34 tuples) are needed to represent the final itemsets.

Finally, we describe how to extract the final itemsets from the cube prefix table. The condition is the same as that in the rules that derive `itemset2`, where a tuple is kept iff the path starting from

Table 5.16: A compact cube for the tuples in Table 5.14.

ID	**1**	C1	**2**	C2	**3**	C3	**4**	C4
1	I1	6	I2	4	I5	1		
4					I4	1		
8					I3	2	I5	1
1(+36)					I5	2		
4(+36)					I4	1		
5			I3	2				
1(+18)			I5	1				
4(+18)			I4	1				
5(+18)			I3	4				
8(+18)					I5	1		
1(+54)			I5	2				
4(+54)			I4	1				
2	I2	3	I4	1				
3			I3	2				
2(+18)	I4	1						
3(+18)	I3	2						
1(+9)	I2	7	I5	1				
2(+9)			I4	2				
3(+9)			I3	4				
8(+9)					I5	1		
1(+45)			I5	2				
2(+45)			I4	2				
5(+9)	I3	2						
1(+27)	I5	1						
2(+27)	I4	2						
3(+27)	I3	6						
8(+27)			I5	1				
1(+63)	I5	2						
2(+63)	I4	2						

the root and ending at the current node does not contain any node with a no value. This condition also maps to a different implementation under the current representation. For each tuple in the cube prefix table, let l_1 be the list of all the items on the corresponding path, l_2 be the list of all the items that were removed to derive the tuple, if the union of l_1 and l_2 equals {I1, . . ., Ik} where Ik is the last item in l_1, then the tuple is kept. It is important to note that this condition can also be used to speedup the computation where the derivations for certain tuples are not performed. In the i-th iteration of the semi-naive evaluation, for each tuple that is already in the data cube, if the first

item that is missing on the corresponding path is Ij where $j < i$, then we can skip this tuple. The tuples that are pruned through this optimization are marked in red in Table 5.16.

5.6 Classification

In this type of tasks, we seek to extract models from existing data, and use these models to predict class labels for future data. The existing data and future data is often referred to as the training data and testing data, respectively. Thus, each tuple in the training data has a column that indicates the class label, e.g., the `PlayTennis` column in Example 5.1, and we use its verticalized representation `vtrain2`(`ID`, `Col`, `Val`, `PlayTennis`) as an example to show how to build classifiers.

5.6.1 Decision Trees

Data cubes represent all subsets and from these we can generate all the edges in the lattice relating such subsets. For example, in the lattice shown in Figure 5.1, a cube contains all the links in solid lines, but the links in dotted lines are not generated. We add an additional column `Lev` to the

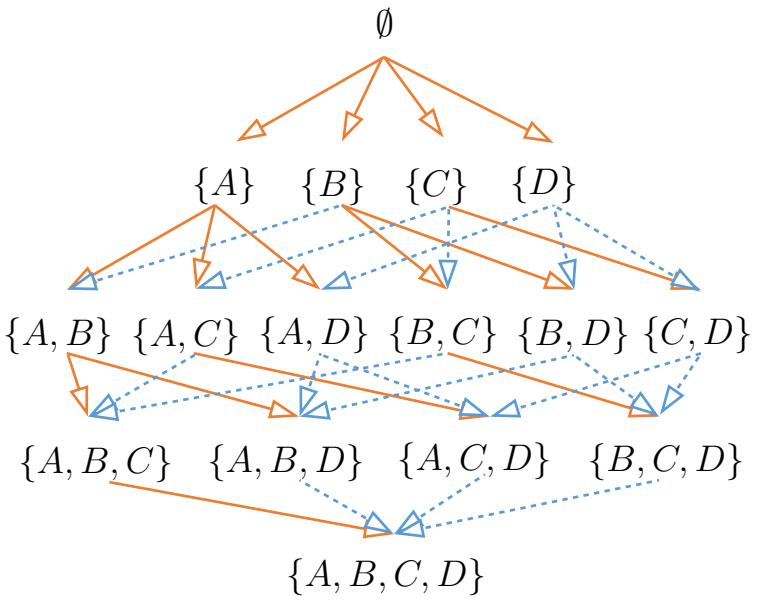

Figure 5.1: Links in a lattice.

cube prefix table, which denotes the number of nodes on the path from the root to the node, and is generated alongside the construction of the cube prefix table. Then two nodes are on the same level in the lattice if the values are the same for column `Lev`, and these additional links can be generated

by the simple rule in Example 5.9, that all the children of a node's grandparent with the same Col and Val are also the node's parents.

Example 5.9. *From a cube prefix table to a lcube prefix table.*

$$\begin{aligned}\texttt{lcpt}(\texttt{Lev1}, \texttt{ID}, \texttt{Col1}, \texttt{Val1}, \texttt{Cnt}, \texttt{PID1}) \texttt{ <- } & \texttt{cpt}(\texttt{Lev}, \texttt{PID}, \texttt{Col}, \texttt{Val}, \_, \texttt{GP}),\\ & \texttt{Lev1} = \texttt{Lev} + 1, \texttt{cpt}(\texttt{Lev1}, \texttt{ID}, \texttt{Col1}, \texttt{Val1}, \texttt{Cnt}, \texttt{PID}),\\ & \texttt{cpt}(\texttt{Lev}, \texttt{PID1}, \texttt{Col1}, \texttt{Val1}, \_, \texttt{GP}).\end{aligned}$$

We refer to a prefix table obtained from the rule of Example 5.9 as a *lcube prefix table*. Such a prefix table provides a suitable representation for decision tree classifiers. Assume we want to build a decision tree classifier for the example in Table 5.1 to predict the values for the PlayTennis column.

First, we define a cube prefix table as follows:

$$\begin{aligned}\texttt{ptcpt}(\texttt{cube}\langle\texttt{ID}\rangle, \texttt{Col}, \texttt{Val}, \texttt{sum}\langle\texttt{P}\rangle, \texttt{sum}\langle\texttt{NP}\rangle, \texttt{prev}\langle\texttt{ID}\rangle) \texttt{ <-}\\ & \texttt{vtrain2}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{PlayTennis}),\\ & \texttt{if}(\texttt{PlayTennis} = \texttt{yes then P} = 1 \texttt{ else P} = 0),\\ & \texttt{NP} = 1 - \texttt{P}.\end{aligned}$$

This cube prefix table computes two aggregates for each node, counting the number of yes and no decisions among all the training data that satisfy the conditions defined by the path leading to the node.

Second, we obtain a lcube prefix table ptlcpt from ptcpt using a rule similar to the one in Example 5.9. For each node in ptlcpt that stores the count of yes and no values, we can derive Gini index, information gain, or other functions that evaluate the purity of the node. For example, the Gini index of a node is computed by the rule below, where Cy and Cn are the number of yes and no decisions, respectively:

$$\texttt{gini}(\texttt{Cy}, \texttt{Cn}, \texttt{G}) \texttt{ <- C} = \texttt{Cy} + \texttt{Cn}, \texttt{G} = 1 - (\texttt{Cy} * \texttt{Cy} + \texttt{Cn} * \texttt{Cn})/(\texttt{C} * \texttt{C}).$$

Third, we compute a decision tree where each intermediate node shows a `maybe` decision, as shown in Example 5.10. Clearly once the Gini index falls below the prescribed threshold, no more branches are generated. However our compiler will be able to realize this only once it understands that the children of nodes with an index below the threshold will have Gini indexes that are below the threshold as well.

Example 5.10. *A decision tree. The Gini index of leaves must be under a threshold:*

$$\begin{aligned}
&\texttt{branch}(1, 0, 0, 1, \texttt{maybe}).\\
&\texttt{branch}(\texttt{PID}, \texttt{Col}, \texttt{Val}, \texttt{ID}, \texttt{Dec}) \texttt{<-}\ \texttt{branch}(_, _, _, \texttt{PID}, \texttt{maybe}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \texttt{bestsplit}(\texttt{PID}, \texttt{Col}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \texttt{ptlcpt}(_, \texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{Cy}, \texttt{Cn}, \texttt{PID}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \texttt{decision}(\texttt{Cy}, \texttt{Cn}, \texttt{Dec}).\\
&\texttt{decision}(\texttt{Cy}, \texttt{Cn}, \texttt{maybe}) \texttt{<-}\ \texttt{gini}(\texttt{Cy}, \texttt{Cn}, \texttt{G}), \texttt{threshold}(\texttt{T}), \texttt{G} \geq \texttt{T}.\\
&\texttt{decision}(\texttt{Cy}, \texttt{Cn}, \texttt{yes}) \texttt{<-}\ \texttt{gini}(\texttt{Cy}, \texttt{Cn}, \texttt{G}), \texttt{threshold}(\texttt{T}), \texttt{G} < \texttt{T}, \texttt{Cy} > \texttt{Cn}.\\
&\texttt{decision}(\texttt{Cy}, \texttt{Cn}, \texttt{no}) \texttt{<-}\ \texttt{gini}(\texttt{Cy}, \texttt{Cn}, \texttt{G}), \texttt{threshold}(\texttt{T}), \texttt{G} < \texttt{T}, \texttt{Cy} \leq \texttt{Cn}.
\end{aligned}$$

Now we have to determine the best column to split for a given node in the lcube prefix table:

$$\begin{aligned}
&\texttt{bestsplit}(\texttt{PID}, \texttt{Col}) \texttt{<-}\ \texttt{improv}(\texttt{PID}, \texttt{Col}, \texttt{C}, \texttt{WG}), \texttt{G} = \texttt{WG}/\texttt{C},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \texttt{not}(\texttt{improv}(\texttt{PID}, \texttt{Col1}, \texttt{C1}, \texttt{WG1}), \texttt{G1} = \texttt{WG1}/\texttt{C1}, \texttt{G1} < \texttt{G}).\\
&\texttt{improv}(\texttt{PID}, \texttt{Col}, \texttt{sum}\langle\texttt{C}\rangle, \texttt{sum}\langle\texttt{WG}\rangle) \texttt{<-}\ \texttt{ptlcpt}(_, \texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{Cy}, \texttt{Cn}, \texttt{PID}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \texttt{gini}(\texttt{Cy}, \texttt{Cn}, \texttt{G}), \texttt{C} = \texttt{Cy} + \texttt{Cn}, \texttt{WG} = \texttt{G} * \texttt{C}.
\end{aligned}$$

We have two ways to achieve that. The first is to specify in our query that we are only interested in branches that have been generated by parents with an index above the threshold, and the second is to add to the rules defining the lcube prefix table a condition that states that the Gini index of a child node must be less than that of its parents.

5.6.2 Bayes Classifiers

Next we describe how to build a different type of classifier called *naive Bayes classifier*. Let $C_1, \ldots, C_m$ be the classes we aim to predict. In the PlayTennis example, m equals 2, C_1 and C_2 represent the values of the `Play` column (we use `Play` to refer to the `PlayTennis` column for short) being "yes" and "no", respectively. Let $p(C_i)$ be the *prior* probability of class C_i, i.e., the probability of having a tuple with class C_i in the training data. For each tuple, let x_j be the value of the j-th column, then the *likelihood* $p(x_j|C_i)$ is the probability that a tuple with class C_i also having value x_j. Given a tuple from the testing data, the classifier predicts a class for the tuple by finding the class C_i that maximizes the *posterior*, which is equivalent to maximizing $\log(p(C_i)) + \sum_{j=1}^{k} \log(p(x_j|C_i))$. In the PlayTennis example, we use a multinomial distribution for each likelihood, then $p(x_j|C_i) = \mathsf{count}\langle x_j \rangle / \mathsf{count}\langle C_i \rangle$, where $\mathsf{count}\langle x_j \rangle$ is the number of tuples with the value of the j-th column being a specific x_j, $\mathsf{count}\langle C_i \rangle$ is the number of tuples with class C_i, and both numbers are computed using tuples in the training data. The example below shows how to implement such a classifier in KDDlog.

Example 5.11. *A naive Bayes classifier. The training data is scanned to count the number of "yes"/"no" decisions for the* `Play` *column, and the number of such decisions for each* $(\mathtt{Col}, \mathtt{Val})$ *combination. (The condition* $\mathtt{Col} = 1$ *in the first rule ensures that the aggregate* $\mathsf{count}\langle \mathtt{T} \rangle$ *counts each unique value of* `T` *only once.) Relation* `domaincnt` *stores the number of unique values for each column, where* `countd` *is equivalent to* COUNT DISTINCT *in SQL. Then we compute the likelihood using Laplace smoothing.*

$$
\begin{aligned}
&\mathtt{prior}(\mathtt{Play}, \mathsf{count}\langle \mathtt{T} \rangle) \text{ <- } \mathtt{vtrain2}(\mathtt{T}, \mathtt{Col}, \_, \mathtt{Play}), \mathtt{Col} = 1.\\
&\mathtt{occur}(\mathtt{Col}, \mathtt{Val}, \mathtt{Play}, \mathsf{count}\langle \mathtt{T} \rangle) \text{ <- } \mathtt{vtrain2}(\mathtt{T}, \mathtt{Col}, \mathtt{Val}, \mathtt{Play}).\\
&\mathtt{domaincnt}(\mathtt{Col}, \mathsf{countd}\langle \mathtt{Val} \rangle) \text{ <- } \mathtt{occur}(\mathtt{Col}, \mathtt{Val}, \_, \_).\\
&\mathtt{likelihood}(\mathtt{Col}, \mathtt{Val}, \mathtt{Play}, \mathtt{Prob}) \text{ <- } \mathtt{occur}(\mathtt{Col}, \mathtt{Val}, \mathtt{Play}, \mathtt{C1}),\\
&\qquad\qquad \mathtt{prior}(\mathtt{Play}, \mathtt{C2}), \mathtt{parameter}(\mathtt{K}), \mathtt{domaincnt}(\mathtt{Col}, \mathtt{C3}),\\
&\qquad\qquad \mathtt{Prob} = \mathsf{log}(\mathtt{C1} + \mathtt{K}) - \mathsf{log}(\mathtt{C2} + \mathtt{K} * \mathtt{C3}).
\end{aligned}
$$

Now assume that table `test` *stores the test data. For each test point, we compute the posterior for "yes"/"no" decisions, and select the decision with higher posterior as the prediction for this point.*

$$\begin{aligned}
&\texttt{test_likelihood}(\texttt{T}, \texttt{Play}, \mathsf{sum}\langle\texttt{Prob1}\rangle) \texttt{ <- } \texttt{test}(\texttt{T}, \texttt{Val@Col}), \texttt{prior}(\texttt{Play}, \texttt{C1}),\\
&\qquad\qquad \texttt{if}(\texttt{likelihood}(\texttt{Col}, \texttt{Val}, \texttt{Play}, \texttt{Prob})\\
&\qquad\qquad\quad \texttt{then Prob1} = \texttt{Prob}\\
&\qquad\qquad\quad \texttt{else parameter}(\texttt{K}), \texttt{domaincnt}(\texttt{Col}, \texttt{C2}),\\
&\qquad\qquad\qquad \texttt{Prob1} = \mathsf{log}(\texttt{K}) - \mathsf{log}(\texttt{C1} + \texttt{K} * \texttt{C2})).\\
&\texttt{predict}(\texttt{T}, \texttt{Play}) \texttt{ <- } \texttt{test_likelihood}(\texttt{T}, \texttt{Play}, \texttt{Prob}), \texttt{prior}(\texttt{Play}, \texttt{C}),\\
&\qquad\qquad \texttt{Prob1} = \texttt{Prob} + \mathsf{log}(\texttt{C}), \mathsf{is_max}((\texttt{Play}), (\texttt{Prob1})).
\end{aligned}$$

5.7 Regression Analysis

In this section, we use a new running example to explain how regression analysis works. Assume that relation $\texttt{house}(\texttt{ID}, \texttt{Price}, \ldots)$ stores the information about house listings, where the values of the first two columns are the ID and listing price of a house, respectively, and the remaining columns represented by "..." contain other basic properties of this house, such as area, built year, longitude, and latitude. Moreover, we assume that all values are numeric, and the values of house ID form an integer sequence $1, \ldots, n$, where n is the number of houses. Given a relation like `house`, regression analysis can help us learn the relationship between house price and other basic properties, i.e., find a function that takes the values of other basic properties as the input and the output is a predicted house price that is very close to the actual house price provided in the `house` relation. Such a function is usually referred to as a *regression model*.

To perform regression analysis, we need to decide the representation of the function we want to learn. For example, we choose to approximate the relationship we want to learn with a linear function, then the regression analysis under this assumption is called *linear regression*, and the resulting regression model is a k-dimensional vector, where k is the number of columns in `house` (excluding the ID column). The rules below produce a verticalized view for `house` that

represents each house as a k-dimensional vector—on top of the results produced by our standard verticalization construct, an additional column 0 with a fixed value 1 is added for every house.

$$\texttt{vhouse}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{Price}) \texttt{ <- } \texttt{house}(\texttt{ID}, \texttt{Val@Col}, \texttt{Val@1}), \texttt{Price} = \texttt{Val@1}.$$
$$\texttt{vhouse}(\texttt{ID}, 0, 1, \texttt{Price}) \texttt{ <- } \texttt{vhouse}(\texttt{ID}, \_, \_, \texttt{Price}).$$

We also need to define a *cost function* that measures how close the predicted house prices are w.r.t. the actual house prices. Let us assume that the "closeness" is defined as the square of the difference between a predicted house price and the actual house price, and the cost function is the average of the "closeness" values for all houses in the table. Such a cost function is referred to as the *least-squares cost function*. In the following, we describe three algorithms on finding a model that minimizes the value of the cost function. These algorithms can also be used to find regression models for functions in other forms.

Example 5.12. *Batch Gradient Descent. We start with an initial model that is a zero vector (or a random vector), and iteratively compute a new model using the model obtained in the previous iteration. For each column* Col*, the value of this column in the new model is obtained by adding* $2.0/\texttt{N} * \texttt{LR} * \texttt{G}$ *to its current value, where* N *is the total number houses,* LR *is a parameter that represents the learning rate, and* G *is the* Col*-th element of the gradient vector. This method is called batch gradient descent since the gradient vector is computed by aggregating the gradient vectors for every house.*

$$\begin{aligned}
&r1.\ \texttt{model}(0, \texttt{Col}, 0) \texttt{ <- } \texttt{vhouse}(_, \texttt{Col}, _, _).\\
&r2.\ \texttt{model}(\texttt{J1}, \texttt{Col}, \texttt{Val1}) \texttt{ <- } \texttt{model}(\texttt{J}, \texttt{Col}, \texttt{Val}), \texttt{gradient}(\texttt{J}, \texttt{Col}, \texttt{G}),\\
&\qquad\qquad\qquad\qquad \texttt{parameter}(\texttt{LR}), \texttt{housecnt}(\texttt{N}),\\
&\qquad\qquad\qquad\qquad \texttt{Val1} = \texttt{Val} + 2.0/\texttt{N} * \texttt{LR} * \texttt{G}, \texttt{J1} = \texttt{J} + 1.\\
&r3.\ \texttt{gradient}(\texttt{J}, \texttt{Col}, \texttt{sum}\langle\texttt{G}\rangle) \texttt{ <- } \texttt{vhouse}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{Price}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{cp}(\texttt{J}, \texttt{ID}, \texttt{Predict}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{G} = (\texttt{Price} - \texttt{Predict}) * \texttt{Val}.
\end{aligned}$$

$$r4.\ \texttt{cp}(\texttt{J}, \texttt{ID}, \texttt{sum}\langle\texttt{Prod}\rangle) \texttt{<-}\ \texttt{vhouse}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \_), \texttt{model}(\texttt{J}, \texttt{Col}, \texttt{Val1}),$$
$$\texttt{Prod} = \texttt{Val} * \texttt{Val1}.$$
$$r5.\ \texttt{housecnt}(\texttt{count}\langle\_\rangle) \texttt{<-}\ \texttt{vhouse}(\_, \texttt{Col}, \_, \_), \texttt{Col} = 0.$$

Example 5.13. *Stochastic Gradient Descent. This method can be obtained by replacing $r3$ and $r4$ in Example 5.12 with the rule below.*

$$\begin{aligned}\texttt{model}(\texttt{J1}, \texttt{Col}, \texttt{Val1}) \texttt{<-}\ & \texttt{housecnt}(\texttt{N}), \texttt{model}(\texttt{J}, \texttt{Col}, \texttt{Val}),\\ & \texttt{vhouse}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{Price}), \texttt{choice}((\texttt{J}), (\texttt{ID})),\\ & \texttt{choice}((\texttt{ID}), (\texttt{J}\%\texttt{N})), \texttt{cp}(\texttt{J}, \texttt{ID}, \texttt{Predict}),\\ & \texttt{G} = (\texttt{Price} - \texttt{Predict}) * \texttt{Val}, \texttt{parameter}(\texttt{LR}),\\ & \texttt{Val1} = \texttt{Val} + 2.0 * \texttt{LR} * \texttt{G}, \texttt{J1} = \texttt{J} + 1.\end{aligned}$$

Instead of computing the gradient vector by averaging the gradient vector for every house, we use the gradient vector at a certain house as an approximation, and compute a new model based on this approximated gradient vector. In each pass over the data set, we repeat this computation for every house. Thus, the iteration counter J *is increased by* N *after each pass, where* N *is the number of houses. Moreover, the data set is shuffled once before the actual computation.*[5] *These behaviors are achieved through the two choice goals, where the former ensures that only one* ID *is selected for each* J*, and the latter ensures that each* ID *is always associated with a* J *that has the same value of* J%N*. Thus, each* ID *is selected exactly once in each pass, and the* IDs *are accessed in the same order in every pass, while the ordering can be any one among all possible* N! *orderings.*

Example 5.14. *Coordinate Descent. Instead of updating the whole model in every iteration, this algorithm changes only one element in the model in each iteration, where all the remaining elements are fixed, and the selected element is updated with a new value that minimizes the cost function. The algorithm repeatedly iterates over all the elements in the model following a preselected ordering. This control flow is implemented by rules $r3$, $r4$, and $r5$, where $r4$ selects an*

[5]The data set is shuffled before each pass in the standard stochastic gradient descent. However, it is very expensive to do shuffling in each pass for large data sets. Here, we adopt a solution proposed in [FKR12] which only shuffles the data set before the first pass, and it was shown that this strategy is a good trade-off.

ordering, and $r5$ performs the actual computation following the ordering selected by $r4$.

Similar to the previous two examples, a tuple of the form $(\texttt{J}, \texttt{Col}, \texttt{Val})$ *in relation* `model` *indicates* `Val` *is the value of the* `Col`*-th element in the model at iteration* `J`*. However, in this example, there is only one tuple in* `model` *for each value of* `J`*, while the values for the remaining elements are not changed in the current iteration, and can be found from the tuples of the form* $(\texttt{J1}, \_, \_)$ *in* `model` *for* $\texttt{J} - \texttt{K} < \texttt{J1} < \texttt{J}$*, where* `K` *is the number of columns in* `vhouse`*.*

$r1$. $\texttt{colcnt}(\texttt{count}\langle\texttt{Col}\rangle) \texttt{<-} \texttt{vhouse}(\texttt{ID}, \texttt{Col}, \_, \_), \texttt{ID} = 1.$

$r2$. $\texttt{colsum}(\texttt{Col}, \texttt{sum}\langle\texttt{Val1}\rangle) \texttt{<-} \texttt{vhouse}(\_, \texttt{Col}, \texttt{Val}, \_), \texttt{Val1} = \texttt{Val} * \texttt{Val}.$

$r3$. $\texttt{model}(0, 0, 0).$

$r4$. $\texttt{model}(\texttt{J1}, \texttt{Col}, 0) \texttt{<-} \texttt{model}(\texttt{J}, \_, \_), \texttt{colcnt}(\texttt{K}), \texttt{J} < \texttt{K}, \texttt{colsum}(\texttt{Col}, \_),$
$\texttt{choice}((\texttt{J}), (\texttt{Col})), \texttt{choice}((\texttt{Col}), (\texttt{J})), \texttt{J1} = \texttt{J} + 1.$

$r5$. $\texttt{model}(\texttt{J1}, \texttt{Col}, \texttt{MV1}) \texttt{<-} \texttt{model}(\texttt{J}, \_, \_), \texttt{colcnt}(\texttt{K}), \texttt{J} \geq \texttt{K}, \texttt{J1} = \texttt{J} + 1,$
$\texttt{J2} = \texttt{J1} - \texttt{K}, \texttt{model}(\texttt{J2}, \texttt{Col}, \texttt{MV}),$
$\texttt{gradient}(\texttt{J}, \texttt{Col}, \texttt{G}), \texttt{colsum}(\texttt{Col}, \texttt{S}),$
$\texttt{MV1} = \texttt{MV} + \texttt{G}/\texttt{S}.$

$r6$. $\texttt{error}(\texttt{K}, \texttt{ID}, \texttt{Price}) \texttt{<-} \texttt{vhouse}(\texttt{ID}, \texttt{Col}, \_, \texttt{Price}), \texttt{Col} = 0, \texttt{colcnt}(\texttt{K}).$

$r7$. $\texttt{error}(\texttt{J1}, \texttt{ID}, \texttt{EV1}) \texttt{<-} \texttt{error}(\texttt{J}, \texttt{ID}, \texttt{EV}), \texttt{J1} = \texttt{J} + 1, \texttt{model}(\texttt{J1}, \texttt{Col}, \texttt{MV1}),$
$\texttt{colcnt}(\texttt{K}), \texttt{J2} = \texttt{J1} - \texttt{K}, \texttt{model}(\texttt{J2}, \texttt{Col}, \texttt{MV2}),$
$\texttt{vhouse}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \_),$
$\texttt{EV1} = \texttt{EV} - (\texttt{MV1} - \texttt{MV2}) * \texttt{Val}.$

$r8$. $\texttt{gradient}(\texttt{J}, \texttt{Col}, \texttt{sum}\langle\texttt{P}\rangle) \texttt{<-} \texttt{vhouse}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \_), \texttt{error}(\texttt{J}, \texttt{ID}, \texttt{EV}),$
$\texttt{P} = \texttt{Val} * \texttt{EV}.$

5.8 Cluster Analysis

The goal of cluster analysis is to partition a set of objects into subsets, where each subset is a cluster such that the objects in this cluster are similar to each other while dissimilar to objects in other clusters. We continue our discussion with the `house` relation introduced in the previous

section, and we refer to each house as a point. In the following, we describe four types of cluster analysis algorithms with an example for each of them.

5.8.1 Partitioning Methods

Given a parameter k, a partitioning method organizes a given set of objects into k disjoint partitions, where each partition represents a cluster.

Example 5.15. *k-Means. Let* K *be the number of clusters the user wants to find. During the initialization, we randomly select* K *points from the data set.*

$$\begin{aligned}
&\texttt{init}(0,0,0).\\
&\texttt{init}(\texttt{C1},\texttt{Col},\texttt{Val}) \texttt{<-}\ \texttt{mc}(\texttt{C}),\texttt{threshold}(\texttt{K},_),\texttt{C} < \texttt{K},\texttt{house}(\texttt{ID},\texttt{Val@Col}),\\
&\qquad\qquad \mathsf{choice}((\texttt{C}),(\texttt{ID})),\mathsf{choice}((\texttt{ID}),(\texttt{C})),\texttt{C1} = \texttt{C}+1.\\
&\texttt{mc}(\mathsf{mmax}\langle\texttt{C}\rangle) \texttt{<-}\ \texttt{init}(\texttt{C},_,_).
\end{aligned}$$

We use these K *selected points as the initial cluster centers. Then in each iteration, we assign every point* ID *to a cluster* C *such that the Euclidean distance between point* ID *and the center of cluster* C *is the closest among all the clusters, and the new cluster center of each cluster is the average of all the points that are assigned to this cluster.*

$$\begin{aligned}
&\texttt{center}(1,\texttt{C},\texttt{Col},\texttt{Val}) \texttt{<-}\ \texttt{init}(\texttt{C},\texttt{Col},\texttt{Val}),\texttt{C} > 0.\\
&\texttt{center}(\texttt{J1},\texttt{C},\texttt{Col},\mathsf{avg}\langle\texttt{Val}\rangle) \texttt{<-}\ \texttt{house}(\texttt{ID},\texttt{Val@Col}),\texttt{dp}(\texttt{J},\texttt{ID},\texttt{C},\texttt{Dist}),\\
&\qquad\qquad \mathsf{is_min}((\texttt{C}),(\texttt{Dist})),\texttt{J1} = \texttt{J}+1.\\
&\texttt{dp}(\texttt{J},\texttt{ID},\texttt{C},\mathsf{sum}\langle\texttt{Diff}\rangle) \texttt{<-}\ \texttt{house}(\texttt{ID},\texttt{Val@Col}),\texttt{center}(\texttt{J},\texttt{C},\texttt{Col},\texttt{Val1}),\\
&\qquad\qquad \texttt{Diff} = (\texttt{Val}-\texttt{Val1})*(\texttt{Val}-\texttt{Val1}).
\end{aligned}$$

We say that a cluster changes in iteration J1 *if the Euclidean distance between the cluster center in iteration* J1 *and the cluster center in iteration* $\texttt{J1}-1$ *is larger than a prescribed threshold* T. *We add the constraint* $\mathsf{not}(\texttt{change}(\texttt{J1}))$ *to select the final clusters from* $\texttt{center}(\texttt{J1},\texttt{C},\texttt{Col},\texttt{Val})$ *such that no cluster changes in iteration* J1*, and the final constraint* $\mathsf{is\_min}((),(\texttt{J1}))$ *ensures that*

only the results from the first iteration that the previous constraint is satisfied are selected.

$$\begin{aligned}
&\texttt{dc}(\texttt{J1}, \texttt{C}, \mathsf{sum}\langle\texttt{Diff}\rangle) \texttt{<-} \texttt{center}(\texttt{J}, \texttt{C}, \texttt{Col}, \texttt{Val}), \texttt{J1} = \texttt{J} + 1,\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{center}(\texttt{J1}, \texttt{C}, \texttt{Col}, \texttt{Val1}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Diff} = (\texttt{Val} - \texttt{Val1}) * (\texttt{Val} - \texttt{Val1}).\\
&\texttt{change}(\texttt{J1}) \texttt{<-} \texttt{dc}(\texttt{J1}, \texttt{C}, \texttt{Dist}), \texttt{threshold}(_, \texttt{T}), \texttt{Dist} > \texttt{T}.\\
&\texttt{cluster}(\texttt{C}, \texttt{Col}, \texttt{Val}) \texttt{<-} \texttt{center}(\texttt{J1}, \texttt{C}, \texttt{Col}, \texttt{Val}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{not}(\texttt{change}(\texttt{J1})), \mathsf{is_min}((), (\texttt{J1})).
\end{aligned}$$

In the above example, the constraints $\mathsf{not}(\texttt{change}(\texttt{J1}))$ and $\mathsf{is\_min}((), (\texttt{J1}))$ in the last rule can be pushed into the recursive computation of `center` so that only tuples up to iteration `J1` are produced where `J1` is the value that satisfies both constraints. Moreover, in each iteration, we only need to keep the tuples produced in the current iteration and the previous iteration, and all the tuples produced in the iterations before the previous iteration can be removed.

5.8.2 Hierarchical Methods

A hierarchical method organizes the objects into a hierarchy of clusters. It can be done by iteratively merging small clusters into a large cluster, or splitting a large cluster into small ones. A method that adopts the former approach is called an agglomerative hierarchical clustering method, while a method that adopts the latter approach is called a divisive hierarchical clustering method. The example below presents an agglomerative hierarchical clustering method.

Example 5.16. *AGglomerative NESting (AGNES). Initially, every point is in a cluster of its own, with the point ID as the cluster ID. Then, at each step, we select two clusters* `C1` *and* `C2` *such that neither is selected before and the distance between these two clusters (defined in* $r9$*) is the smallest. We merge them to create a new cluster* $\texttt{J} + 1$*, where* `J` *is the current maximal cluster ID. This merge operation is recorded by two tuples* $(\texttt{J} + 1, \texttt{C1})$ *and* $(\texttt{J} + 1, \texttt{C2})$ *in* `cluster`*, indicating that cluster* $\texttt{J} + 1$ *is the parent of both cluster* `C1` *and cluster* `C2`*.*

$r1$. $\texttt{cluster}(\texttt{ID}, \texttt{ID}) \texttt{<-} \texttt{house}(\texttt{ID}, \texttt{Val@1})$.

$r2$. $\texttt{mc}(\texttt{mmax}\langle\texttt{C}\rangle) \texttt{<-} \texttt{cluster}(\texttt{C}, \_)$.

$r3$. $\texttt{merge}(\texttt{J1}, \texttt{C1}, \texttt{C2}) \texttt{<-} \texttt{mc}(\texttt{J}), \texttt{cluster}(\texttt{C1}, \_), \texttt{cluster}(\texttt{C2}, \_),$
$\texttt{C1} < \texttt{C2}, \texttt{choice}((\texttt{C1}), (\texttt{J})), \texttt{choice}((\texttt{C2}), (\texttt{J})),$
$\texttt{choice}((\texttt{J}), (\texttt{C1}, \texttt{C2})), \texttt{dc}(\texttt{C1}, \texttt{C2}, \texttt{Dist}),$
$\texttt{is\_min}((\texttt{C1}, \texttt{C2}), (\texttt{Dist})), \texttt{J1} = \texttt{J} + 1$.

$r4$. $\texttt{cluster}(\texttt{J1}, \texttt{C1}) \texttt{<-} \texttt{merge}(\texttt{J1}, \texttt{C1}, \_)$.

$r5$. $\texttt{cluster}(\texttt{J1}, \texttt{C2}) \texttt{<-} \texttt{merge}(\texttt{J1}, \_, \texttt{C2})$.

These clusters form a hierarchical structure, where the points in a cluster can be found by collecting all the points that belong to its child clusters. Then the distance Dist *between two clusters* C1 *and* C2 *is defined as the minimal Euclidean distance between every pair of points* (ID1, ID2)*, where* ID1 *and* ID2 *belong to* C1 *and* C2*, respectively.*

$r6$. $\texttt{cp}(\texttt{ID}, \texttt{ID}) \texttt{<-} \texttt{cluster}(\texttt{ID}, \texttt{ID})$.

$r7$. $\texttt{cp}(\texttt{C}, \texttt{ID}) \texttt{<-} \texttt{cluster}(\texttt{C}, \texttt{C1}), \texttt{cp}(\texttt{C1}, \texttt{ID})$.

$r8$. $\texttt{dp}(\texttt{ID1}, \texttt{ID2}, \texttt{sum}\langle\texttt{Diff}\rangle) \texttt{<-} \texttt{house}(\texttt{ID1}, \texttt{V1@Col}), \texttt{house}(\texttt{ID2}, \texttt{V2@Col}),$
$\texttt{Diff} = (\texttt{V1} - \texttt{V2}) * (\texttt{V1} - \texttt{V2})$.

$r9$. $\texttt{dc}(\texttt{C1}, \texttt{C2}, \texttt{min}\langle\texttt{Dist}\rangle) \texttt{<-} \texttt{cp}(\texttt{C1}, \texttt{ID1}), \texttt{cp}(\texttt{C2}, \texttt{ID2}), \texttt{dp}(\texttt{ID1}, \texttt{ID2}, \texttt{Dist})$.

In the above example, the $\texttt{is\_min}((\texttt{C1}, \texttt{C2}), (\texttt{Dist}))$ goal in $r3$ is a constraint—the set of rules without this constraint defines all possible hierarchical structures of clusters, while the current formulation that contains this constraint selects the one that is the output of the AGNES algorithm.

5.8.3 Density-Based Methods

A density-based method models clusters as dense regions separated by sparse regions.

Example 5.17. *Density-Based Spatial Clustering of Applications with Noise (DBSCAN). There are two user-specified parameters* Eps *and* T*. The former is used to define the* Eps*-neighborhood of a point, which is the space within a radius* Eps *centered at the current point. The latter is then*

used to define a core object, which is a point with an Eps*-neighborhood that contains at least* T *points.*

$$r1.\ \texttt{dp}(\texttt{ID1}, \texttt{ID2}, \texttt{sum}\langle\texttt{Diff}\rangle) \texttt{<-} \texttt{house}(\texttt{ID1}, \texttt{V1@Col}), \texttt{house}(\texttt{ID2}, \texttt{V2@Col}),$$
$$\texttt{Diff} = (\texttt{V1} - \texttt{V2}) * (\texttt{V1} - \texttt{V2}).$$
$$r2.\ \texttt{neighbor}(\texttt{ID1}, \texttt{ID2}) \texttt{<-} \texttt{dp}(\texttt{ID1}, \texttt{ID2}, \texttt{Dist}), \texttt{threshold}(\texttt{Eps}, \_),$$
$$\texttt{Dist} \leq \texttt{Eps}.$$
$$r3.\ \texttt{neighborcnt}(\texttt{ID1}, \texttt{count}\langle\texttt{ID2}\rangle) \texttt{<-} \texttt{neighbor}(\texttt{ID1}, \texttt{ID2}).$$
$$r4.\ \texttt{core}(\texttt{ID}) \texttt{<-} \texttt{neighborcnt}(\texttt{ID}, \texttt{Cnt}), \texttt{threshold}(\_, \texttt{T}), \texttt{Cnt} \geq \texttt{T}.$$

We say that two core objects ID1 *and* ID2 *are density-connected if there is a chain of core objects* $c_1, c_2, \ldots, c_k$*, such that* $c_1 =$ ID1*,* $c_k =$ ID2*, and* c_{i+1} *is in the* Eps*-neighborhood of* c_i *for* $i = 1, \ldots, k-1$*. It is easy to see that the relation density-connected* dc *is an equivalence relation, and for each equivalence class, we pick the point with the minimal ID to represent all the points in this class. Then, all the points in the same equivalence class together with all the points in their* Eps*-neighborhoods form one cluster, and we use the ID of the point that is selected to represent all the points in the equivalence class as the cluster ID.*

$$r5.\ \texttt{dc}(\texttt{ID}, \texttt{ID}) \texttt{<-} \texttt{core}(\texttt{ID}).$$
$$r6.\ \texttt{dc}(\texttt{ID1}, \texttt{ID3}) \texttt{<-} \texttt{dc}(\texttt{ID1}, \texttt{ID2}), \texttt{neighbor}(\texttt{ID2}, \texttt{ID3}), \texttt{core}(\texttt{ID3}).$$
$$r7.\ \texttt{repr}(\texttt{min}\langle\texttt{ID1}\rangle, \texttt{ID2}) \texttt{<-} \texttt{dc}(\texttt{ID1}, \texttt{ID2}).$$
$$r8.\ \texttt{cluster}(\texttt{ID1}, \texttt{ID2}) \texttt{<-} \texttt{repr}(\texttt{ID1}, \texttt{ID2}).$$
$$r9.\ \texttt{cluster}(\texttt{ID1}, \texttt{ID3}) \texttt{<-} \texttt{repr}(\texttt{ID1}, \texttt{ID2}), \texttt{neighbor}(\texttt{ID2}, \texttt{ID3}).$$

It is important to note that $r7$ represents a constraint over dc, and this constraint can be pushed into the recursive computation of dc. Moreover, the aggregate min that is used to select one point for each equivalence class can be replaced by other constructs, such as max and choice, to delivery logically equivalent clustering results—for each equivalence class, the set of points in the corresponding cluster remains the same, while the cluster ID might be different. In fact, we obtain the following procedures when choice is used.

1. Initially all the core objects are unselected.

2. We select an unselected core object `ID`, create a new cluster with `ID` as its cluster ID, and add the core object to the cluster.

3. After a core object is added, we also select all the core objects that belong to its `Eps`-neighborhood and add them to the cluster. This operation is repeated until no more core objects can be added to the cluster.

4. We complete the current cluster by augmenting it with all the points in the `Eps`-neighborhoods of the core objects in the cluster.

5. We repeat from Step (2) until all the core objects are selected.

The operations in Step (3) and Step (4) can be performed together, where we add all the points in the `Eps`-neighborhood of a core object to the cluster, and we repeat this operation for every newly added point as long as it is a core object. This is indeed the DBSCAN algorithm.

5.8.4 Grid-Based Methods

A grid-based clustering algorithm uses a multi-resolution grid data structure. To simplify the discussion, we use an example in two dimensional space, although it can be easily generalized to higher dimensional spaces. Assume that the schema of the `house` relation contains four columns, where a tuple of the form $(\texttt{ID}, \texttt{Price}, \texttt{X}, \texttt{Y})$ represents that the listing price for house `ID` is `Price`, and the latitude and longitude of the house are `X` and `Y`, respectively. The example below explains how does a grid-based clustering algorithm work on this data set.

Example 5.18. *STatistical INformation Grid (STING). We first build the hierarchical structure from the root to the leaves, where the each non-leaf node has four children, and each child corresponds to one quadrant of the parent node. The average number of points in a leaf is no more than a prescribed threshold.*

$$\texttt{delta}(1, 0, -1, 0, -1).$$
$$\texttt{delta}(2, 1, 0, 0, -1).$$

$$\texttt{delta}(3, 0, -1, 1, 0).$$

$$\texttt{delta}(4, 1, 0, 1, 0).$$

$$\texttt{range}(\mathsf{count}\langle \texttt{ID}\rangle, \mathsf{min}\langle \texttt{X}\rangle, \mathsf{max}\langle \texttt{X}\rangle, \mathsf{min}\langle \texttt{Y}\rangle, \mathsf{max}\langle \texttt{Y}\rangle) \texttt{ <- } \texttt{house}(\texttt{ID}, \_, \texttt{X}, \texttt{Y}).$$

$$\texttt{grid}(0, 1, \texttt{Cnt}, \texttt{XL}, \texttt{XH}, \texttt{YL}, \texttt{YH}) \texttt{ <- } \texttt{range}(\texttt{Cnt}, \texttt{XL}, \texttt{XH}, \texttt{YL}, \texttt{YH}).$$

$$\begin{aligned}\texttt{grid}(\texttt{ID1}, \texttt{MT1}, \texttt{Cnt1}, \texttt{XL1}, \texttt{XH1}, \texttt{YL1}, \texttt{YM1}) \texttt{ <- } & \texttt{grid}(\texttt{ID}, \texttt{MT}, \texttt{Cnt}, \texttt{XL}, \texttt{XH}, \texttt{YL}, \texttt{YH}),\\ & \texttt{threshold}(\texttt{T}), \texttt{Cnt} > \texttt{T}, \texttt{delta}(\texttt{DID}, \texttt{DXL}, \texttt{DXH}, \texttt{DYL}, \texttt{DYH}),\\ & \texttt{ID1} = \texttt{ID} + \texttt{MT} * \texttt{DID}, \texttt{MT1} = \texttt{MT} * 4, \texttt{Cnt1} = \texttt{Cnt}/4,\\ & \texttt{DX} = (\texttt{XH} - \texttt{XL})/2, \texttt{XL1} = \texttt{XL} + \texttt{DX} * \texttt{DXL}, \texttt{XH1} = \texttt{XH} + \texttt{DX} * \texttt{DXH},\\ & \texttt{DY} = (\texttt{YH} + \texttt{YL})/2, \texttt{YL1} = \texttt{YL} + \texttt{DY} * \texttt{DYL}, \texttt{YH1} = \texttt{YH} + \texttt{DY} * \texttt{DYH}.\end{aligned}$$

Then, we scan the data set once to obtain parameters for each leaf, where the parameters are computed based on all the points that fall into the range of a leaf. After that, we propagate these parameters from the leaves to the root, and the parameters of a non-leaf node is computed by aggregating the corresponding parameters of all its children. Finally, we store the final results in `sting`.

$$\begin{aligned}\texttt{param}(\texttt{ID}, \texttt{MT}, \mathsf{mcount}\langle \_\rangle, \mathsf{msum}\langle \texttt{P}\rangle, \mathsf{msum}\langle \texttt{P2}\rangle, \mathsf{mmin}\langle \texttt{P}\rangle, \mathsf{mmax}\langle \texttt{P}\rangle) \texttt{ <-}\\ \texttt{grid}(\texttt{ID}, \texttt{MT}, \texttt{Cnt}, \texttt{XL}, \texttt{XH}, \texttt{YL}, \texttt{YH}), \texttt{threshold}(\texttt{T}),\\ \texttt{Cnt} \leq \texttt{T}, \texttt{house}(\_, \texttt{P}, \texttt{X}, \texttt{Y}), \texttt{XL} \leq \texttt{X}, \texttt{X} \leq \texttt{XH},\\ \texttt{YL} \leq \texttt{Y}, \texttt{Y} \leq \texttt{YH}, \texttt{P2} = \texttt{P} * \texttt{P}.\end{aligned}$$

$$\begin{aligned}\texttt{param}(\texttt{ID1}, \texttt{MT1}, \mathsf{msum}\langle \texttt{Cnt}\rangle, \mathsf{msum}\langle \texttt{SP}\rangle, \mathsf{msum}\langle \texttt{SP2}\rangle, \mathsf{mmin}\langle \texttt{MinP}\rangle,\\ \mathsf{mmax}\langle \texttt{MaxP}\rangle) \texttt{ <- } \texttt{param}(\texttt{ID}, \texttt{MT}, \texttt{Cnt}, \texttt{SP}, \texttt{SP2}, \texttt{MinP}, \texttt{MaxP}),\\ \texttt{MT1} = \texttt{MT}/4, \texttt{ID1} = \texttt{ID}\%\texttt{MT1}.\end{aligned}$$

$$\begin{aligned}\texttt{param2}(\texttt{ID}, \texttt{MT}, \mathsf{max}\langle \texttt{Cnt}\rangle, \mathsf{max}\langle \texttt{SP}\rangle, \mathsf{max}\langle \texttt{SP2}\rangle, \mathsf{min}\langle \texttt{MinP}\rangle, \mathsf{max}\langle \texttt{MaxP}\rangle) \texttt{ <-}\\ \texttt{param}(\texttt{ID}, \texttt{MT}, \texttt{Cnt}, \texttt{SP}, \texttt{SP2}, \texttt{MinP}, \texttt{MaxP}).\end{aligned}$$

$$\begin{aligned}\texttt{sting}(\texttt{PID}, \texttt{ID}, \texttt{Cnt}, \texttt{MP}, \texttt{StdP}, \texttt{MinP}, \texttt{MaxP}) \texttt{ <-}\\ \texttt{param2}(\texttt{ID}, \texttt{MT}, \texttt{Cnt}, \texttt{SP}, \texttt{SP2}, \texttt{MinP}, \texttt{MaxP}), \texttt{Cnt} > 0,\\ \texttt{if}(\texttt{MT} = 1 \texttt{ then PID} = \texttt{ID else PID} = \texttt{ID}\%(\texttt{MT}/4)),\\ \texttt{MP} = \texttt{SP}/\texttt{Cnt}, \texttt{StdP} = \mathsf{sqrt}(\texttt{SP2}/\texttt{Cnt} - \texttt{MP} * \texttt{MP}).\end{aligned}$$

The above example shows the key steps in the STING algorithm, and some details are simplified. In the Datalog program that implements the complete algorithm, we need to (i) make proper treatment to the points on the boundary of some rectangle areas so that each point is counted only once on the leaf level, and (ii) collect an additional parameter for each node that indicates the type of distribution that prices of houses fall into this node follow.

5.9 User-Defined Aggregate Functions

Most existing systems support data mining functionalities through built-in functions. This approach has proven to be natural for many users. In KDDlog, we will also support this approach, where we ship our data mining algorithm implementations with *user-defined aggregate* (UDA) functions. Basically, an aggregate can be viewed as a mapping between tuples of a predicate that are streamed in, and a set of tuples returned at the end or during the computation and defines a new derived predicate. In order to support tables with different number of columns, these UDA functions are defined over tables that use verticalized representations.

We can create a UDA function for the predicate `likelihood` in Example 5.11 as follows:

```
aggregate nbctrain(ID, Col1, Val1, Play1): (Col2, Val2, Play2, Prob)
[mytable, likelihood AS:
  occur(Col, Val, Play, count⟨ID⟩) <- mytable(ID, Col, Val, Play).
  prior(Play, sum⟨C⟩) <- occur(Col, Val, Play, C), Col = 1.
  total(sum⟨C⟩) <- prior(Play, C).
  likelihood(Col, Val, Play, P) <- occur(Col, Val, Play, C1),
                                   prior(Play, C2), P = log(C1) − log(C2).
  likelihood(0, all, Play, P) <- prior(Play, C1), total(C2),
                                 P = log(C1) − log(C2).
]
```

This example assumes that the Datalog definition of the aggregate is also used as executable code in the aggregate library. But our framework does not limit the language to Datalog. For instance,

the code in the brackets can be written in Java or other programming languages.

UDA functions are typically called in the body of rules, as per the following example:

$$\texttt{save}(\texttt{Col2}, \texttt{Val2}, \texttt{Play2}, \texttt{Prob}) \texttt{<-} \texttt{train}(\texttt{ID}, \texttt{Val@Col}, \texttt{Val@5}), \texttt{Play} = \texttt{Val@5},$$
$$\texttt{nbctrain}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{Play}){:}(\texttt{Col2}, \texttt{Val2}, \texttt{Play2}, \texttt{Prob}).$$

In many situations, besides the names of the input and output predicates, we will also specify those of auxiliary tables where important parameters are kept. For instance, the aggregate for predicting in a naive Bayes classifier, we will also include the name of the table save whereby its aggregate definition becomes:

$$\begin{aligned}
&\textsf{aggregate}\ \texttt{nbctest}(\texttt{ID1}, \texttt{Col1}, \texttt{Val1}){:}\ (\texttt{ID2}, \texttt{Col2}, \texttt{Val2}, \texttt{Play})\\
&[\texttt{testtable}, \texttt{predict}, \texttt{likelihood}\ \texttt{AS}{:}\\
&\quad \texttt{test_likelihood}(\texttt{ID}, \texttt{Play}, \textsf{sum}\langle\texttt{Prob}\rangle) \texttt{<-} \texttt{testtable}(\texttt{ID}, \texttt{Col}, \texttt{Val}),\\
&\qquad\qquad \texttt{likelihood}(\texttt{Col}, \texttt{Val}, \texttt{Play}, \texttt{Prob}).\\
&\quad \texttt{result}(\texttt{ID}, \texttt{Play}) \texttt{<-} \texttt{test_likelihood}(\texttt{ID}, \texttt{Play}, \texttt{Prob1}),\\
&\qquad\qquad \texttt{likelihood}(\texttt{0}, \texttt{all}, \texttt{Play}, \texttt{Prob2}),\\
&\qquad\qquad \texttt{Prob} = \texttt{Prob1} + \texttt{Prob2}, \textsf{is_max}((\texttt{Play}), (\texttt{Prob})).\\
&\quad \texttt{predict}(\texttt{ID}, \texttt{Col}, \texttt{Val}, \texttt{Play}) \texttt{<-} \texttt{testtable}(\texttt{ID}, \texttt{Col}, \texttt{Val}), \texttt{result}(\texttt{ID}, \texttt{Play}).\\
&]
\end{aligned}$$

Then, $\texttt{nbctest}(\texttt{ID1}, \texttt{Col1}, \texttt{Val1}){:}(\texttt{ID2}, \texttt{Col2}, \texttt{Val2}, \texttt{Play})$ can be used in goals of rules to classify tuples as shown in the following example, where we pass in the name of the auxiliary table in the square bracket:

$$\begin{aligned}
&\texttt{decision}(\texttt{ID1}, \texttt{Col1}, \texttt{Val1}, \texttt{Play}) \texttt{<-} \texttt{test}(\texttt{ID}, \texttt{Val@Col}),\\
&\qquad\qquad \texttt{nbctest}(\texttt{ID}, \texttt{Col}, \texttt{Val}){:}(\texttt{ID1}, \texttt{Col1}, \texttt{Val1}, \texttt{Play})[\texttt{save}].
\end{aligned}$$

Horizontal Calls for Vertical Aggregates. Our UDA functions are defined over tables that use vertical representations. They are called in the body of the rules in a very natural fashion. In

KDDlog, the basic built-in aggregates are also called in the head of the rules, in a style that is inspired by SQL. Thus KDDlog will also support the invocation of UDA functions in the head of the rules. This will enable the importation and invocation of those UDA functions in Spark SQL, which is the SQL module on Apache Spark, thus combining the benefits of horizontal and vertical representations. For example, for our `nbctest` aggregate, KDDlog will support a call such as the following:

$$\texttt{decision}(\texttt{Out}, \texttt{Temp}, \texttt{Hum}, \texttt{Wind}, \texttt{nbctest}\langle\texttt{save}\rangle) \texttt{ <- } \texttt{test}(\texttt{Out}, \texttt{Temp}, \texttt{Hum}, \texttt{Wind}).$$

which will actually be rewritten and interpreted as follows:

$$\begin{aligned}&\texttt{testid}(\texttt{tupleid}\langle\rangle, \texttt{Out}, \texttt{Temp}, \texttt{Hum}, \texttt{Wind}) \texttt{ <- } \texttt{test}(\texttt{Out}, \texttt{Temp}, \texttt{Hum}, \texttt{Wind}).\\ &\texttt{decision}(\texttt{Val1@Col1}, \texttt{Play}) \texttt{ <- } \texttt{testid}(\texttt{ID}, \texttt{Val@Col}),\\ &\qquad\qquad\qquad \texttt{nbctest}(\texttt{ID}, \texttt{Col}, \texttt{Val}):(\texttt{ID1}, \texttt{Col1}, \texttt{Val1}, \texttt{Play})[\texttt{save}].\end{aligned}$$

Here `tupleid` is an aggregate that assigns a unique ID, e.g., a sequence number, to tuples.

We refer to the use of a UDA function in the head of a rule as a *horizontal call*. In order to use horizontal calls, the number of columns in the head must be the same as those in the output of UDA functions. This is true in our case since besides the $(\texttt{ID1}, \texttt{Dec1})$ pair, we also include `Col1` and `Val1`. For rollups and data cubes, we fill the missing values with nulls to satisfy this requirement. Note that the aggregate `nbctest` takes only one argument, but any number of arguments (including zero) can be entered at any time.

Finally, horizontal calls can be easily integrated into SQL, where the corresponding SQL query for the above example is shown as follows:

```
SELECT Out, Temp, Hum, Wind, kmeans(save) AS Play
FROM test
GROUP BY Out, Temp, Hum, Wind
```

5.10 Conclusion

We have presented the KDDlog language and its support for major KDD applications. While these results are still preliminary, they show that logic constructs can provide enough expressive power to express efficiently KDD algorithms. Moreover, traditional database applications and graph applications are naturally supported by KDDlog since these applications can be efficiently expressed using basic Datalog constructs and monotonic aggregates that are supported by KDDlog. Thus, the results presented here also demonstrate that it is possible to provide a declarative language that supports various types of applications in a unified framework, while its efficient implementation and parallelization remain as a future work.

CHAPTER 6

Conclusion and Future Work

In this thesis, we have presented a declarative language solution for the unified support of advanced analytics and various big data applications, and discussed the scalable and portable implementation of the language over various computing environments.

We presented an experimental study that compared several recursive query evaluation algorithms for the in-memory query evaluation on multicore machines. Our experimental results showed that the simple SSC algorithms significantly outperformed other compared algorithms in terms of speed and memory utilization.

We presented the DeAL system with focus on the compilation techniques that enable efficient in-memory parallel evaluation of Datalog programs on shared-memory multicore machines. The proposed techniques (i) recognize when a given program is lock-free, (ii) transform a locking program into a lock-free program, and (iii) find an efficient parallel plan that correctly evaluates the program while minimizing the use of locks and other overhead required for parallel evaluation. We demonstrated the effectiveness of the proposed techniques with extensive experiments.

We have designed the KDDlog language by extending DeAL with new constructs and representations for descriptive analytics. We start with the problem of supporting algorithms that can operate on tables having arbitrary number of columns, and this leads to the verticalization constructs that support verticalized representations for tables. Then we provide compact representations called *prefix tables* for rollups and data cubes. Then we show how the constructs and repre-

sentations for descriptive analytics can be used to express a wide spectrum of predictive analytics tasks, including association rule mining, classification, regression analysis, and cluster analysis. Thus, KDDlog enables a unified support for both descriptive analytics and predictive analytics. Finally, the language supports the exportation of KDD algorithms implemented in KDDlog and importation of algorithms implemented in other languages through user-defined aggregate functions.

The results presented in this thesis are encouraging, but much future work is still required to fully realize the ambitious goal of unifying various types of applications in a single declarative language. An immediate future work is to develop a system that supports the parallel evaluation of KDDlog programs on multicore machines and clusters. Another interesting topic for future work is to support other types of applications, including extending our languages to support machine learning applications, and implementing the language proposal in [Zan12] to support streaming applications. Lastly, we plan to investigate techniques that accelerate the query evaluation by taking advantage of new hardwares, including solid state drives, graphics processing units, and field-programmable gate arrays.

References

[ABC11] Foto N Afrati, Vinayak Borkar, Michael Carey, Neoklis Polyzotis, and Jeffrey D Ullman. "Map-Reduce Extensions and Recursive Queries." In *EDBT*, pp. 1–8. ACM, 2011.

[ACG15] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. "Design and Implementation of the LogicBlox System." In *SIGMOD*, pp. 1371–1382. ACM, 2015.

[AJ87] Rakesh Agrawal and HV Jagadish. "Direct Algorithms for Computing the Transitive Closure of Database Relations." In *VLDB*, pp. 1–4. Morgan Kaufmann Publishers Inc., 1987.

[AJ88] Rakesh Agrawal and HV Jagadish. "Multiprocessor Transitive Closure Algorithms." In *DPDS*, pp. 56–66. IEEE, 1988.

[AJ90] Rakesh Agrawal and HV Jagadish. "Hybrid Transitive Closure Algorithms." In *VLDB*, pp. 326–334. Morgan Kaufmann Publishers Inc., 1990.

[AKN12] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. "Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems." *PVLDB*, **5**(10):1064–1075, 2012.

[ANB11] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. "Relational Transducers for Declarative Networking." In *PODS*, pp. 283–292, 2011.

[AOT03] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. "The Deductive Database System LDL++." *TPLP*, **3**(1):61–94, 2003.

[AU12] Foto N Afrati and Jeffrey D Ullman. "Transitive Closure and Recursive Datalog Implemented on Clusters." In *EDBT*, pp. 132–143. ACM, 2012.

[AXL15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. "Spark SQL: Relational Data Processing in Spark." In *SIGMOD*, pp. 1383–1394, 2015.

[BBC12a] Vinayak R Borkar, Yingyi Bu, Michael J Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. "Declarative Systems for Large-Scale Machine Learning." *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, **35**(2):24–32, 2012.

[BBC12b] Yingyi Bu, Vinayak R. Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. "Scaling Datalog for Machine Learning on Big Data." *arXiv preprint arXiv:1203.0160*, 2012.

[BCG11] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. "Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing." In *ICDE*, pp. 1151–1162. IEEE, 2011.

[BS09] Martin Bravenboer and Yannis Smaragdakis. "Strictly Declarative Specification of Sophisticated Points-to Analyses." In *OOPSLA*, pp. 243–262. ACM, 2009.

[BSH91] David A. Bell, J Shao, and M. Elizabeth C. Hull. "A Pipelined Strategy for Processing Recursive Queries in Parallel." *Data & Knowledge Engineering*, **6**(5):367–391, 1991.

[BTA13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, et al. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware." In *ICDE*, pp. 362–373. IEEE, 2013.

[BZN05] Peter A Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In *CIDR*, pp. 225–237, 2005.

[CCH93] Filippo Cacace, Stefano Ceri, and Maurice Houtsma. "A Survey of Parallel Execution Strategies for Transitive Closure and Logic Programs." *Distributed and Parallel Databases*, **1**(4):337–382, 1993.

[CGK90] Danette Chimenti, Ruben Gamboa, Ravi Krishnamurthy, Shamim Naqvi, Shalom Tsur, and Carlo Zaniolo. "The LDL System Prototype." *TKDE*, **2**(1):76–90, March 1990.

[CJL08] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets." *PVLDB*, **1**(2):1265–1276, 2008.

[cpua] "SPEC® CINT2006 Result. Cisco Systems: Cisco UCS C460 M4 (Intel Xeon E7-4890 v2, 2.80GHz)." `https://www.spec.org/cpu2006/results/res2014q1/cpu2006-20140224-28687.pdf`.

[cpub] "SPEC® CINT2006 Result. Dell Inc.: PowerEdge R720 (Intel Xeon E5-2690, 2.90 GHz)." `https://www.spec.org/cpu2006/results/res2012q1/cpu2006-20120228-19541.pdf`.

[cpuc] "SPEC® CINT2006 Result. Supermicro: Supermicro A+ Server 2042G-6RF (AMD Opteron 6376, 2.30GHz)." `https://www.spec.org/cpu2006/results/res2012q4/cpu2006-20121005-24693.pdf`.

[CSN09] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. "Power-Law Distributions in Empirical Data." *SIAM review*, **51**(4):661–703, 2009.

[CW89] Simona Cohen and Ouri Wolfson. "Why a Single Parallelization Strategy is Not Enough in Knowledge Bases." In *PODS*, pp. 200–216. ACM, 1989.

[dea] "Deductive Application Language System." `http://wis.cs.ucla.edu/deals/`.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *CACM*, **51**(1):107–113, 2008.

[dlva] "DLV (parallel version)." `http://www.mat.unical.it/ricca/downloads/parallelground10.zip`.

[dlvb] "DLV (single-processor version)." `http://www.dlvsystem.com/files/dlv.x86-64-linux-elf-static.bin`.

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

[DR94] Shaul Dar and Raghu Ramakrishnan. "A Performance Study of Transitive Closure Algorithms." *SIGMOD Rec.*, **23**(2):454–465, 1994.

[DS13] Jonathan Dees and Peter Sanders. "Efficient Many-Core Query Execution in Main Memory Column-Stores." In *ICDE*, pp. 350–361. IEEE, 2013.

[FKR12] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. "Towards a Unified Architecture for In-RDBMS Analytics." In *SIGMOD*, pp. 325–336. ACM, 2012.

[fli] "Apache Flink." `https://flink.apache.org`.

[FSG98] Min Fang, Narayanan Shivakumar, Hector Garcia-molina, Rajeev Motwani, and Jeffrey D Ullman. "Computing Iceberg Queries Efficiently." In *VLDB*, pp. 299–310. Morgan Kaufmann Publishers Inc., 1998.

[GKK14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. "Clingo=ASP+Control: Preliminary report." *arXiv preprint arXiv:1405.3694*, 2014.

[gra] "GraphStream." `http://graphstream-project.org`.

[GST92] Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. "Parallel Bottom-Up Processing of Datalog Queries." *The Journal of Logic Programming*, **14**(1):101–126, 1992.

[GST95] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. "Mapping Datalog Program Execution to Networks of Processors." *TKDE*, **7**(3):351–361, 1995.

[GXD14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. "GraphX: Graph Processing in a Distributed Dataflow Framework." In *OSDI*, pp. 599–613. USENIX Association, 2014.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. "Mining Frequent Patterns Without Candidate Generation." In *SIGMOD*, pp. 1–12. ACM, 2000.

[Hul89] Guy Hulin. "Parallel Processing of Recursive Queries in Distributed Architectures." In *VLDB*, pp. 87–96. Morgan Kaufmann Publishers Inc., 1989.

[Ioa86] Yannis E Ioannidis. "On the Computation of the Transitive Closure of Relational Operators." In *VLDB*, pp. 403–411. Morgan Kaufmann Publishers Inc., 1986.

[IR88] Yannis E Ioannidis and Raghu Ramakrishnan. "Efficient Transitive Closure Algorithms." In *VLDB*, pp. 382–394. Morgan Kaufmann Publishers Inc., 1988.

[IT15] Matteo Interlandi and Letizia Tanca. "On the CALM Principle for BSP Computation." In *AMW*, 2015.

[Jak91] Håkan Jakobsson. "Mixed-Approach Algorithms for Transitive Closure." In *PODS*, pp. 199–205. ACM, 1991.

[KIC92] Robert Kabler, Yannis E Ioannidis, and Michael J Carey. "Performance Evaluation of Algorithms for Transitive Closure." *Information Systems*, **17**(5):415–441, 1992.

[KKL09] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs." *PVLDB*, **2**(2):1378–1389, 2009.

[Lat08] Chris Lattner. "LLVM and Clang: Next Generation Compiler Technology." In *The BSD Conference*, pp. 1–2, 2008.

[LBG12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud." *PVLDB*, **5**(8):716–727, 2012.

[LK14] Jure Leskovec and Andrej Krevl. "SNAP Datasets: Stanford Large Network Dataset Collection." `http://snap.stanford.edu/data`, June 2014.

[LKF05] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations." In *SIGKDD*, pp. 177–187. ACM, 2005.

[LLD09] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters." *Internet Mathematics*, **6**(1):29–123, 2009.

[LPF06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. "The DLV System for Knowledge Representation and Reasoning." *TOCL*, **7**(3):499–562, 2006.

[MAB10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-Scale Graph Processing." In *SIGMOD*, pp. 135–146. ACM, 2010.

[MBK02] Stefan Manegold, Peter Boncz, and Martin Kersten. "Optimizing Main-Memory Join on Modern Hardware." *TKDE*, **14**(4):709–730, 2002.

[MBY16] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. "MLlib: Machine learning in Apache Spark." *JMLR*, **17**(34):1–7, 2016.

[Mit97] Tom M Mitchell. *Machine Learning*. McGraw-Hill Boston, MA, 1997.

[MMI13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System." In *SOSP*, pp. 439–455, 2013.

[MSZ13a] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. "A Declarative Extension of Horn Clauses, and its Significance for Datalog and its Applications." *TPLP*, **13**(4-5):609–623, 2013.

[MSZ13b] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. "Extending the Power of Datalog Recursion." *The VLDB Journal*, **22**(4):471–493, 2013.

[NAB15] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. "Join Processing for Graph Patterns: An Old Dog with New Tricks." In *GRADES*. ACM, 2015.

[ORS08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig Latin: A Not-So-Foreign Language for Data Processing." In *SIGMOD*, pp. 1099–1110. ACM, 2008.

[PRS13] Simona Perri, Francesco Ricca, and Marco Sirianni. "Parallel Instantiation of ASP Programs: Techniques and Experiments." *TPLP*, **13**(02):253–278, 2013.

[Prz88] Teodor C. Przymusinski. "Perfect Model Semantics." In *ICLP/SLP*, pp. 1081–1096, 1988.

[roa] "Eastern USA Road Network." `http://www.dis.uniroma1.it/challenge9/data/USA-road-d/USA-road-d.E.gr.gz`.

[RS86] Louiqa Raschid and Stanley Y. W. Su. "A Parallel Processing Strategy for Evaluating Recursive Queries." In *VLDB*, pp. 412–419. Morgan Kaufmann Publishers Inc., 1986.

[RS92] Kenneth A Ross and Yehoshua Sagiv. "Monotonic Aggregation in Deductive Databases." In *PODS*, pp. 114–126. ACM, 1992.

[RSS92] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. "CORAL—Control, Relations and Logic." In *VLDB*, pp. 238–250. Morgan Kaufmann Publishers Inc., 1992.

[SGL13] Jiwon Seo, Stephen Guo, and Monica S Lam. "SociaLite: Datalog Extensions for Efficient Social Network Analysis." In *ICDE*, pp. 278–289. IEEE, 2013.

[Shk16] Alexander Shkapsky. *A Declarative Language for Advanced Analytics and its Scalable Implementation*. PhD thesis, UCLA, 2016.

[SK93] Bart Selman and Henry Kautz. "Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems." In *IJCAI*, pp. 290–295. Morgan Kaufmann, 1993.

[SKC93] Bart Selman, Henry Kautz, and Bram Cohen. "Local Search Strategies for Satisfiability Testing." *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, **26**:521–532, 1993.

[SKH12] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. "Optimizing Large-Scale Semi-Naive Datalog Evaluation in Hadoop." In *Datalog in Academia and Industry*, pp. 165–176. Springer, 2012.

[SKN94] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. "Cache Conscious Algorithms for Relational Query Processing." In *VLDB*, pp. 510–521. Morgan Kaufmann Publishers Inc., 1994.

[SL91] Jürgen Seib and Georg Lausen. "Parallelizing Datalog Programs by Generalized Pivoting." In *PODS*, pp. 241–251. ACM, 1991.

[SLM92] Bart Selman, Hector J Levesque, and David G Mitchell. "A New Method for Solving Hard Satisfiability Problems." In *AAAI*, pp. 440–446. AAAI Press / The MIT Press, 1992.

[Soc] "SociaLite." `https://github.com/socialite-lang/socialite`.

[Spe93] William M Spears. "Simulated Annealing for Hard Satisfiability Problems." *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, **26**(533–558), 1993.

[SPS13] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. "Distributed Socialite: A Datalog-Based Language for Large-Scale Graph Analysis." *PVLDB*, **6**(14):1906–1917, 2013.

[sql] "SQL Server 2014." `http://www.microsoft.com/en-us/server-cloud/products/sql-server/`.

[STA98] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. "Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications." In *SIGMOD*, pp. 343–354. ACM, 1998.

[SYI16] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. "Big Data Analytics with Datalog Queries on Spark." In *SIGMOD*, pp. 1135–1149. ACM, 2016.

[SYZ15] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. "Optimizing Recursive Queries with Monotonic Aggregates in DeALS." In *ICDE*, pp. 867–878. IEEE, 2015.

[SZZ13] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. "Graph Queries in a Next-Generation Datalog System." *PVLDB*, **6**(12):1258–1261, 2013.

[tpca] "TPC-H." `http://www.tpc.org/tpch/`.

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

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

[TSJ09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. "Hive: A Warehousing Solution over a Map-Reduce Framework." *PVLDB*, **2**(2):1626–1629, 2009.

[Ull85] Jeffrey D Ullman. "Implementation of Logical Query Languages for Databases." *TODS*, **10**(3):289–321, 1985.

[Van93] Allen Van Gelder. "Foundations of Aggregation in Deductive Databases." In *DOOD*, pp. 13–34. Springer, 1993.

[VB86] Patrick Valduriez and Haran Boral. "Evaluation of Recursive Queries Using Join Indices." In *Expert Database Conf.*, pp. 271–293, 1986.

[vec] "Vectorwise." `http://www.actian.com/`.

[Vel14] Todd L. Veldhuizen. "Triejoin: A Simple, Worst-Case Optimal Join Algorithm." In *ICDT*, pp. 96–106, 2014.

[VK88] Patrick Valduriez and Setrag Khoshfian. "Parallel Evaluation of the Transitive Closure of a Database Relation." *International Journal of Parallel Programming*, **17**(1):19–42, 1988.

[War62] Stephen Warshall. "A Theorem on Boolean Matrices." *JACM*, **9**(1):11–12, 1962.

[War75] Henry S Warren Jr. "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations." *CACM*, **18**(4):218–220, 1975.

[WBH15] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. "Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines." *PVLDB*, **8**(12):1542–1553, 2015.

[wik] "Wikipedia." `http://www.wikipedia.org`.

[Wol88] Ouri Wolfson. "Sharing the Load of Logic-Program Evaluation." In *DPDS*, pp. 46–55. IEEE, 1988.

[WS88] Ouri Wolfson and Avi Silberschatz. "Distributed Processing of Logic Programs." In *SIGMOD*, pp. 329–336. ACM, 1988.

[WZB92] Ouri Wolfson, Weining Zhang, Harish Butani, Akira Kawaguchi, and Kui Mok. "Parallel Processing of Graph Reachability in Databases." *International Journal of Parallel Programming*, **21**(4):269–302, 1992.

[YDC14] Mohan Yang, Bolin Ding, Surajit Chaudhuri, and Kaushik Chakrabarti. "Finding Patterns in a Knowledge Base Using Keywords to Compose Table Answers." *PVLDB*, **7**(14):1809–1820, 2014.

[YIF08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language." In *OSDI*, pp. 1–14. USENIX Association, 2008.

[YSZ15] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. "Parallel Bottom-Up Evaluation of Logic Programs: DeALS on Shared-Memory Multicore Machines." In *Technical Communications of ICLP*, 2015.

[YSZ16] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. "Scaling up the Performance of more Powerful Datalog Systems on Multicore Machines." *The VLDB Journal*, 2016.

[YZ14a] Mohan Yang and Carlo Zaniolo. "Main Memory Evaluation of Recursive Queries on Multicore Machines." In *IEEE BigData*, pp. 251–260. IEEE, 2014.

[YZ14b] Mohan Yang and Carlo Zaniolo. "Main Memory Evaluation of Recursive Queries on Multicore Machines." Technical Report 140014, UCLA CSD, 2014.

[YZ16] Mohan Yang and Carlo Zaniolo. "KDDlog: Knowledge Discovery in Datalog." Technical Report 160007, UCLA CSD, 2016.

[Zan12] Carlo Zaniolo. "Logical Foundations of Continuous Query Languages for Data Streams." In *Datalog in Academia and Industry*, pp. 177–189. Springer, 2012.

[ZCD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." In *NSDI*, pp. 15–28, San Jose, CA, 2012. USENIX.

[ZCF97] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, and Roberto Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers Inc., 1997.

[ZWC95] Weining Zhang, Ke Wang, and Siu-Cheung Chau. "Data Partition and Parallel Evaluation of Datalog Programs." *TKDE*, **7**(1):163–176, 1995.

[ZYD16a] Carlo Zaniolo, Mohan Yang, Ariyam Das, and Matteo Interlandi. "Improving the Power, Performance and Usability of Datalog by Pushing Constraints into Recursion." Technical Report 160003, UCLA CSD, 2016.

[ZYD16b] Carlo Zaniolo, Mohan Yang, Ariyam Das, and Matteo Interlandi. "The Magic of Pushing Extrema into Recursion: Simple, Powerful Datalog Programs." In *AMW*, 2016.