Stony Brook University

Non-Termination Analysis and Cost-Based Optimization for Logic Programs

A Dissertation Presented

by

Senlin Liang

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2013

Stony Brook University

The Graduate School

Senlin Liang

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation.

Michael Kifer – Dissertation Advisor
Professor, Department of Computer Science

Yanhong Annie Liu – Chairperson of Defense
Professor, Department of Computer Science

David S. Warren
Professor, Department of Computer Science

Gopal Gupta – External Committee Member
Professor, Department of Computer Science
University of Texas at Dallas

This dissertation is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Dissertation

Non-Termination Analysis and Cost-Based Optimization for Logic Programs

by

Senlin Liang

Doctor of Philosophy

in

Computer Science

Stony Brook University

2013

Rule systems have seen an upsurge of interest in the past decade, as many in the academia and industry started to discover more applications for rules such as the Semantic Web, policy management, and program analysis. In response, high-level logic languages such as $\mathcal{F}$LORA-2[1] and SILK[2] have been invented as tools for developing complex knowledge bases by knowledge engineers who are not programmers. The knowledge bases that are created by this type of users are typically complex and they stress the engine capabilities. Therefore, there are new challenges in order for logic engines to be able to process these knowledge bases, and this thesis focuses on two of them: non-termination analysis and cost-based optimization.

Non-termination analysis examines program execution history when the program is suspected to not terminate and informs the programmer about the exact reasons for this behavior. We study the problem of non-termination in tabled logic engines and propose a suite of algorithms, called non-*Termin*ation Anal*yzer* or `Terminyzer`, which analyze forest logging traces of program execution and output sequences of tabled subgoal calls and their host rule ids that cause non-termination. Moreover, `Terminyzer` identifies the precise set of subgoals, together with their host rules, that recursively call one another and lead to non-termination. `Terminyzer` also attempts

[1] `http://flora.sourceforge.net`

[2] `http://silk.semwebcentral.org`

to automatically rectify non-terminating programs by heuristically fixing some causes of misbehavior.

Besides the non-termination problem, cost-based optimizations similar to those in Database systems are needed in order for rule systems to be practical in processing complex queries against large knowledge bases. To this end, this thesis proposes a *Cost*-based opt*imizer*, called `Costimizer`, which first efficiently estimates predicate statistics and then applies them to greedily optimize rules and queries. `Costimizer` performs size estimation through an abstract evaluation of rules and it distinguishes itself from previous size estimation approaches by efficiently preserving argument dependencies.

`Terminyzer` and a prototype of `Costimizer` have been implemented for $\mathcal{F}$LORA-*2* and SILK. Their effectiveness and practicability have been validated through extensive experimental studies.

To my family and teachers

Contents

List of Figures

List of Tables

Acknowledgements

I am really grateful to my thesis advisor, Professor Michael Kifer, for his continuous support and guidance throughout my entire graduate study at Stony Brook University. His enthusiasm, dedication, and attitude towards research and teaching have greatly impressed me and set a perfect example for my professional career. I have always had the freedom to explore my research interests and access to his knowledge in all areas. Without his commitment, I could not have finished this thesis. I would also like to thank my other committee members: Professor Yanhong Annie Liu and Professor David S. Warren for all the encouraging discussions and invaluable lessons on programming languages and WAM, and Professor Gopal Gupta for agreeing to be my external committee member and his patience with my progress on this thesis. My sincere thanks also go to my master's thesis advisor, Professor Inna Pivkina, who offered me the opportunity to continue my graduate study and introduced me to the field of declarative languages. I am also grateful to Professor Enrico Pontelli for his support and encouragement.

It will always be my fortune to have taken courses from world-class professors at Stony Brook University, including compiler design from Professor Radu Grosu, database systems from Professor Himanshu Gupta, programming languages from Professor Eugene Stark, and information retrieval and extraction from Professor Amanda Stent. I appreciate Professor I.V. Ramakrishnan and Professor C.R. Ramakrishnan for the encouraging conversations and useful tips about graduate study. I also thank Dr. Terry Swift for enabling part of this thesis by implementing subgoal abstraction and forest logging in XSB and Dr. Benjamin Grosof for a number of suggestions that helped to improve this thesis. I would also like to extend my thanks to our department's staff members, especially Brian Tria, Cynthia Scalzo, and Betty Knittweis.

It was a great pleasure to have worked with and learned from several very hard-working and smart fellow graduate students during these past several years: Paul Fodor, Hui Wan, Anu Singh, Song Feng, Polina Kuznetsova, Reza Basseda, Li Niu, and many more. I am sure to remember all the interesting moments that we had together, especially our best new year wishes that "I donot want to see you again next year"—wishing each other soon graduate.

I am also grateful to many personal friends who have encouraged me and helped me in various ways, including Xiaomeng Ban, Yao Chen, Xiaoling Cui, Qiumin Dong, Shaofeng Duan, Lei E, Siyi Guo, Wei Hu, Zhitao Li, Shun Liang, Jie Liu, Yilu Mao, Jidong Wang, Wei Xu, Shang Yang, Guangxi Zeng, Xiang Zeng, Fen Zhang, Han Zhang, Wenbin Zhang, Xiao Zhang, Gaoming Zhao, Dengpan Zhou, Lisha Zhou, and many others.

My sincerest thanks go to my family for their love, support, and encouragement, especially my parents, my grandparents, my brother and sister-in-law, and my uncles and aunts. They are always there for me and I can not describe how much I owe each of them.

Chapter 1

Introduction

The development of high-level logic languages such as $\mathcal{F}$LORA-*2* and SILK aims at making logic-based knowledge representation accessible to knowledge engineers who are not programmers. The knowledge bases that are created by this type of users are typically complex and stressing the engine capabilities, because knowledge engineers usually have very limited knowledge of logic engine's actual evaluation strategy. Therefore, there are new challenges in order for logic engines to be able to process these knowledge bases, and we focus on two of them: non-termination analysis and cost-based optimization.

1.1 Non-Termination Analysis

The problem of run-away computations in logic programs is much more serious than in procedural programming because of the declarative nature of the logic languages and the large gap between the declarative semantics and the actual evaluation strategy. This problem is even more vexing in high-level logic languages, which position themselves as tools for developing complex knowledge bases by knowledge engineers. These engineers cannot be expected to debug the procedural aspects of the rule bases that they create and thus they require special support. Non-termination has been flagged as a key issue standing on the way of creating complex biological knowledge bases in the SILK project, where the use of function symbols is more common due to the higher-order features of HiLog [CKW93] and F-logic [KLW95], and due to the

proliferation of Skolemized head-existentials that are passed down to the engine by the knowledge acquisition system.

There are three main scenarios where programs may not terminate. First, the use of recursion plagues Prolog under the standard SLD-resolution evaluation strategy. This can be illustrated by the following simple Example 1.1. The prevalent way to address this problem is to use *tabling*, which is also known under the more technical term of *SLG-resolution*. Tabling was pioneered by the XSB system [SW12] and is now supported by a number of other systems, such as Yap [CDR12], B-Prolog [Zho12], and Ciao [HBC$^+$12]. In Example 1.1, tabling the predicate, p, will cause the evaluation to terminate.[1]

Example 1.1 *In the following program,*

```
p(X) :- p(X).
?- p(a).
```

there are a recursive loop of calls. Therefore, Prolog's SLD-resolution evaluation strategy will not terminate the query. □

The second scenario where programs might not terminate, even under the SLG-resolution, occurs when patterns of increasingly deep nested calls keep being generated during the evaluation, as in the following Example 1.2. There, query evaluation will successively call $p(a)$, $p(f(a))$, $p(f(f(a)))$, and so on. Since neither call subsumes the other, tabling will not help terminate the evaluation process. However, a surprisingly simple technique known as *subgoal abstraction* [RSar], also pioneered by XSB, takes care of this problem. The idea is to modify the calls by "abstracting" deeply nested subterms and replacing them with new variables. For instance, in Example 1.2, we could abstract calls once the depth limit of 3 has been reached.[2] As a result, $p(f(f(f(a))))$ and all the subsequent calls would be abstracted to $p(f(f(X)))$, $X = f(a)$.

[1] The derivatives in XSB are as follows.
`:- table p/1` for variant tabling, or
`:- table p/1 as subsumptive` for subsumptive tabling.

[2] The derivatives in XSB are as follows.
`:- set_prolog_flag(max_table_subgoal_depth,3).`
`:- set_prolog_flag(max_table_subgoal_action,abstract).`

Example 1.2 *In the following program,*

```
:- table p/1.
p(X) :- p(f(X)).
?- p(a).
```

an infinite number of tabled calls will be made due to recursion and the use of functor symbol f. □

For instance, in XSB (which to our knowledge is the only system that supports both tabling and subgoal abstraction), the above query in Example 1.2 will terminate. Generally, tabling enhanced with subgoal abstraction is able to completely evaluate all queries that have a finite number of answers. The only remaining major source of non-termination, with both tabling and subgoal abstraction being used, is when the number of answers to a query or its subqueries is infinite. A program that exhibits this behavior is given in Example 1.3.

Example 1.3 *In this program,*

```
:- table p/1.
p(a).
p(f(X)) :- p(X).
?- p(X).
```

the query has an infinite number of answers: $p(a)$, $p(f(a))$, $p(f(f(a)))$, and so on. □

Clearly, such queries cannot be evaluated completely, but if the program is what the user intended, the user could ask the system to stop after getting the required number of answers or avoid generating increasingly deep terms using *radial restraints* [GS13]. However, in our experience, the user usually does not intend to construct infinite predicates. Finding out how the infinite number of answers came about and fixing the problem is difficult even for an experienced programmer and even for programs that have just a few dozens of rules. For knowledge bases that have thousands of rules, like the ones we have been dealing with in the SILK project, diagnosing this problem is an onerous and frustrating job. In the absence of subgoal abstraction, this difficulty also exists for the aforesaid problem of detecting sequences

of subgoals of increasing depth. For a knowledge engineer who is not a programmer, debugging non-termination is out of the question.

We note that neither the problem of program termination nor that of whether the number of answers is finite is decidable [SD94, Sip96], so no algorithm can prove termination or non-termination in general. Sufficient conditions for termination of logic programs have been proposed in the literature [BAK91, SD94, VDSS01, LSS04, BCG+07, NDS07, NGSKDS08, SkGS+10], but most deal with Prolog or Prolog-like evaluation strategies. Although many of these results are very deep, their practical impact is limited because they provide only sufficient conditions for termination. The precise classes of programs for which these algorithms work are typically punishingly inexpressive and usually not investigated at all (see Section 2.6 for a discussion). Moreover, neither tabling nor subgoal abstraction are taken into account by most of these works, so they have limited use for advanced logic engines like XSB and its derivatives, $\mathcal{F}$LORA-*2* and SILK. This thesis takes a different approach and develops a suite of algorithms, called non-*Termin*ation anal*yzer* or `Terminyzer` [Lia12, LK13c, LK13b], to help users to precisely locate and explain non-termination causes.

We first study the problem for tabled logic engines with subgoal abstraction. By analyzing forest logging traces, `Terminyzer` detects both tabled subgoal sequences that cause non-termination, and, more importantly, the exact rules where these calls occur and the rule sequences that are fired in a cyclic manner, thus leading to non-termination. This makes `Terminyzer` amenable to serving as a back-end for user-friendly graphical interfaces, which can greatly simplify the debugging process. Such an interface has been constructed by the SILK team and is currently underway for the open-source $\mathcal{F}$LORA-*2* system. Moreover, we make a step towards automatic remediation of non-terminating programs by proposing an algorithm that heuristically fixes some of the faulty programs. Finally, in order for `Terminyzer` to be applicable in more tabled logic engines, we relax its system requirements and study non-termination in tabled logic engines *without* subgoal abstraction.

1.2 Cost-Based Optimization

As mentioned earlier, the knowledge bases created by knowledge engineers are normally complex and large, and more importantly these knowledge bases tend to stress the engine capabilities. Therefore, whether meaningful queries can be processed reasonably fast becomes an even more important task. To tackle this problem, this thesis investigates adapting cost-based optimization similar to those in Database system query optimizers to logic engines.

In selecting query execution plans, Database query optimizers depend on accurate and fast algorithms for estimating predicate (relation) sizes, which in turn depend on joint data distributions of values in different arguments. Traditionally, query optimizers estimate these sizes using statistical summaries for base predicates and propagating them to relational expressions assuming *argument independence* [Chr84]. Assuming that distributions of values in different arguments of a predicate are independent, joint data distribution can be derived relatively easily.

To make size estimation practical, data distributions must be summarized accurately and efficiently. *Histogram* is one such summarization technique that is in wide use in all major Database systems. Briefly, histograms group values of similar frequencies into buckets and estimate the frequencies of values in each bucket in a uniform and efficient way. Different types of histograms have been proposed in the literature, which differ in their complexity, cost, and accuracy. As usual, accuracy comes at the expenses of time and space, and the problem of balancing these conflicting requirements was discussed in [IP95]. Comprehensive surveys and classification of various histograms can be found in [Ioa03, PHIS96].

Provided that data statistics of involved predicates for a given query are available, optimizers search for the most efficient execution plans within predetermined time and space budgets. There has been quite a lot of research on optimization algorithms which are either *optimal* or *greedy* search. The two most popular and well studied optimal search algorithms are *dynamic programming* [OL90, MN06, MN08] and *top-down partition search* [VM96, DT07], and they have similar performances in both time and space [DT07]. However, because of the exponential cost nature of optimal search, most Database systems (e.g., DB2 [GLSW93] and Sybase SQL Anywhere [BP00]) implement greedy algorithms and restrict search space to *left-deep trees* [SAC$^+$79].

There are two major challenges in adapting similar optimizations to rule systems. First, because of the argument independence assumption, which is rarely true for real application datasets, size estimates based on histograms can be off by orders of magnitude [SLMK01], and it has been shown that estimation errors grow exponentially with the number of involved join predicates [IC91, LK10, LK12, Lia12]. In logic programming, this problem is exacerbated by the presence of recursive predicates. Second, optimization algorithms have to consider not only how basic operations are performed in logic engines, but also how facts are indexed since indexing is one of the most important performance factors of all major rule systems as previously observed [LFWK09b].

Aiming at addressing these two problems, this thesis proposes a *Cost*-based opt*imizer*, called `Costimizer`, which consists of two components: a *cost estimator* and an *optimizing unit*. The cost estimator implements *statistics for derived predicates* (SDP) [LK10, LK12], which is a suite of algorithms to perform size estimation for different rule types (e.g., selection, union, intersection, projection, join, product, negation, and recursion) where *dependency matrices* are used to store estimated predicate statistics. Dependency matrices can be viewed as extensions of histograms and they are designed to preserve argument dependencies, thus taking care of the problem caused by argument independent assumption. The optimizing unit, using estimated predicate statistics, greedily optimizes rules and queries by reordering relevant predicates and adding necessary indexing commands that are believed to benefit targeted logic engines. It is worth mentioning that our cost estimator is independent of the optimizing unit, and thus its cost estimates can be used to benefit *any* cost-based optimizer.

Chapter 2

Non-Termination Analysis

2.1 Preliminaries

2.1.1 Tabling in XSB

The limitations of Prolog's standard SLD-resolution based evaluation strategy are well-known: it is incomplete and can go into an infinite loop even for simple Datalog rule sets. To address this problem, tabling (also known as SLG-resolution) was developed over two decades ago and [SW12] provides the most recent insight into this mechanism. In tabled evaluation, calls to tabled predicates are cached in a *table* $\mathcal{T}$ for subsequent calls. $\mathcal{T}$ can be viewed as a set of subgoal-answers pairs of the form $(sub, ansrs)$ where $ansrs$ are proven instances of subgoal sub.

When a tabled subgoal, sub, is issued, tabling examines whether there is a pair $(sub', ansrs') \in \mathcal{T}$ such that sub is *similar* (to be explained shortly) to sub'. If so, then answer clause resolution is performed instead of program clause resolution, i.e., $ansrs'$ are used to satisfy sub. In this case, sub is referred to as the *consumer* of sub' while sub' is the *producer* of sub. If no tabled answers can be used above, a new table entry of the form $(sub, ansrs)$ is added to $\mathcal{T}$, where initially $ansrs = \emptyset$. Then sub is resolved against program clauses, as usual in Prolog. In such a case, all newly derived answers for sub are added to $ansrs$, sub becomes a producer of these answers, and all subsequent subgoals that are similar to sub become consumers of sub's answers.

There are two main ways to define subgoal-similarity mentioned above. Depending on which notion is chosen, the tabling strategy is called *variant* or *subsumptive*. In

variant tabling, *sub* is similar to *sub′* if *sub* is a *variant* of *sub′*, i.e., they are identical up to variable renaming. In subsumptive tabling, *sub* is similar to *sub′* if *sub* is *subsumed* by *sub′*, i.e., there is a variable substitution σ such that $\sigma(sub') = sub$. Note that in this case the notion of similarity is asymmetric. Since only unique answers are added to the table and returned to consumers, tabled evaluation terminates *if* there is only a finite number of tabled subgoals and each tabled subgoal has finitely many answers. For instance, this is the case in Datalog, i.e., when function symbols are not present. It has been proven that tabled evaluation terminates for any program with the *bounded term depth property*, i.e., when all terms that are ever generated in the course of SLG-resolution, including all subgoals and answers, have an upper bound on their depth [SW12].

The workings of SLG-resolution can be captured by an *SLG-forest*, which has an *SLG-tree* for every new (dissimilar) subgoal to tabled predicates. The SLG-tree for *sub* has root of the form *sub* :- *sub*, and each non-root node is of the form $\theta(sub)$:- $\theta(left\_subs)$, where θ is the substitution obtained from resolving *sub* against the knowledge base and $\theta(left\_subs)$ are the remaining subgoals needed to prove *sub*. If $\theta(left\_subs)$ is an empty clause, $\theta(sub)$ is an answer to *sub*. Children of a root node are obtained through resolution of a tabled subgoal against program clauses. Children of non-root nodes are obtained through answer clause resolution if the leftmost selected literal is tabled or through program clause resolution if the leftmost selected literal is not tabled. Each edge in the tree corresponds to a derivation step of program or answer clause resolution.

Example 2.4 *The SLG-forest for the evaluation of this following XSB program is shown in Figure 1, where each node is labeled with an ordinal denoting the creation order (a timestamp) of the node during evaluation.*

```
:- table path/2.
edge(1,2).    edge(1,3).    edge(2,1).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
?- path(1,Y).
```

Children of node 11 are obtained through answer clause resolution since its leftmost selected literal $path(1, Y)$ *is tabled, while children of nodes 2 and 5 are obtained through*

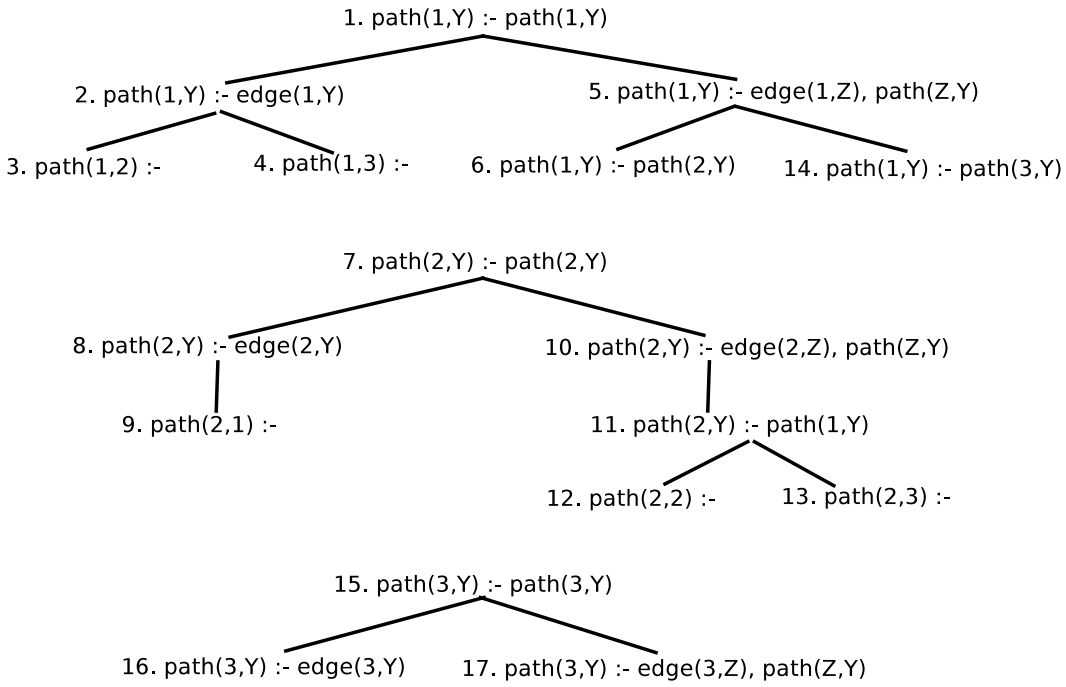


Figure 1: SLG-Forest of Example 2.4

program clause resolution because their leftmost selected literals are not tabled. This is a simplified version of an example in [SWS+13]. □

2.1.2 Forest Logging in XSB

Compared to Prolog systems, logic engines that support tabling are much more involved. They suspend and resume computation paths, delay negated subgoals that are involved in loops through negation, simplify these subgoals once their truth values become known, and manage the table accordingly. For debugging and performance optimization purposes, programmers may need to inspect table operations during evaluation. To this end, XSB has recently provided a new facility, called *forest logging* (`logforest`), which makes certain table events available to the programmer.[1] Currently, `logforest` records these following events.

[1]Although currently XSB is the only system supporting forest logging, all tabled logic engines have the requisite information internally and could expose it to the user to take advantage of the advanced debugging facilities, such as `Terminyzer`, that are enabled by this feature.

- *Call to a tabled predicate.* If a subgoal *parent* calls another subgoal *child*, i.e., the evaluation of *parent* fires a rule that issues *child*, a Prolog fact of the form $tc(child, parent, status, timestamp)$ is logged. Here *timestamp* is the timestamp of the event representing its sequence order. *Status* is the current status of *child*, and it can take the following values:
  - *new* if *child* is a newly issued subgoal,
  - *cmp* if the evaluation of *child* has been completed, and
  - *incmp* if *child* is not a new subgoal, but is yet to be completely evaluated.

  If the subgoal is negative, a similar fact $nc(child, parent, status, timestamp)$ is logged. If *child* is the first tabled subgoal in an evaluation, *parent* is the special subgoal *root*.

- *Derivation of a new answer.* When a new answer, *ansr*, is derived for *sub* and added to the table, the fact $na(ansr, sub, timestamp)$ is added to the log. When a new conditional answer *ansr* `:-` *delayed_literals* is derived for *sub* and added to the table, a log entry of the form $nda(ansr, sub, delayed\_literals, timestamp)$ is recorded. Here *ansr* is the answer substitution and *delayed_literals* are the delayed literals whose truth value is yet to be determined (this usually occurs due to recursive loops through negation).

- *Return of an answer to a consumer.* If an answer, *ansr*, is returned to a consumer subgoal, *child*, which is called by *parent*, a fact of the form $ar(ansr, child, parent, timestamp)$ is added to the log. If the answer is conditional, $dar(ansr, child, parent, timestamp)$ is recorded.

- *Subgoal completion.* When all mutually recursive subgoals in a set, S, are completed, `logforest` records $cmp(sub, sccnum, timestamp)$ for each $sub \in S$, where *sccnum* is an ordinal that identifies S. If *sub* is *completed early* (i.e., its truth is established without the need to fully evaluate all the dependent subgoals), a similar fact $cmp(sub, ec, timestamp)$ is recorded where *ec* stands for *early completion*.

- *Other events.* `Logforest` also records delays of negative literals, table abolishes, and errors. These events are not needed for our purposes and are omitted.

Example 2.5 *For the evaluation of the program in Example 2.4, the forest logging trace is given in the first column of Table 1. The second column in the table is the*

Log	*Node*	*Explanation*
`tc(path( 1,_v0),root,new,0)`	*1*	*initial call*
	2	*program clause resolution*
`na([ 2],path( 1,_v0),1)`	*3*	*program clause resolution*
		new answer
`na([ 3],path( 1,_v0),2)`	*4*	*program clause resolution*
		new answer
	5	*program clause resolution*
	6	*program clause resolution*
`tc(path( 2,_v0),path( 1,_v0),new,3)`	*7*	*new call made by node 6*
	8	*program clause resolution*
`na([ 1],path( 2,_v0),4)`	*9*	*program clause resolution*
		new answer
	10	*program clause resolution*
`tc(path( 1,_v0),path( 2,_v0),incmp,5)`	*11*	*repeated unfinished call*
`ar([ 2],path( 1,_v0),path( 2,_v0),6)`	*12*	*answer clause resolution*
		answer to consumer
`na([ 2],path( 2,_v0),7)`	*12*	*new answer*
`ar([ 3],path( 1,_v0),path( 2,_v0),8)`	*13*	*answer clause resolution*
		answer to consumer
`na([ 3],path( 2,_v0),9)`	*13*	*new answer*
	14	*program clause resolution*
`tc(path( 3,_v0),path( 1,_v0),new,10)`	*15*	*new call made by node 14*
	16	*program clause resolution*
	17	*program clause resolution*
`cmp(path( 3,_v0),3,11)`	*15*	*evaluation completed*
`ar([ 1],path( 2,_v0),path( 1,_v0),12)`	*9*	*return to consumer*
`na([ 1],path( 1,_v0),13)`	*6*	*new answer*
`ar([ 2],path( 2,_v0),path( 1,_v0),14)`	*12*	*return to consumer*
`ar([ 3],path( 2,_v0),path( 1,_v0),15)`	*13*	*return to consumer*
`ar([ 1],path( 1,_v0),path( 2,_v0),16)`	*1*	*return to consumer*
`cmp(path( 1,_v0),1,17)`	*1*	*evaluation completed*
`cmp(path( 2,_v0),1,18)`	*7*	*evaluation completed*

Table 1: Forest Log for the Evaluation of Example 2.4

label of the node in the SLG-trees of Figure 1 where a corresponding event happens. The third column provides an explanation. An answer for a subgoal is represented as a substitution for the list of variables in the subgoal. For instance, in the second log entry $na([2], path(1, \_v0), 1)$*, which records the derivation of a new answer, the*

answer is represented as [2] and its subgoal's list of variables is [_v0], meaning that the substitution _v0 = 2 is an answer. □

2.2 Adding Ids to Rules

One key enabling idea in `Terminyzer` is a transformation that adds unique ids to rules in such a way that this information is preserved in the forest logging trace. For our purposes, we want each subgoal call in the trace to "remember" the rule from which this call was issued. Although this information is not available in the original program, one can instrument any logic program so that each subgoal call to tabled predicates would be stamped with the id of its *host rule*, i.e., rule from whose body the call was issued.

The transformation processes the original program rule by rule and assigns a new id to each newly encountered rule. In each such rule, all non-tabled predicates stay unchanged and each tabled predicate, p/n, is augmented with one more argument so that p/n is replaced with $p/(n+1)$. The additional (last) argument is a new variable for a tabled head predicate, and it is the rule's id for every tabled body predicate. Details are given in Algorithm 1.

1 **while** *unprocessed rules remain* **do**
2 Get the next program rule R: $head(x_1, ..., x_n)$:- $body$.;
3 Generate a new rule id, $id(R)$;
4 **if** $head/n$ *is tabled* **then**
5 Change the head literal to $head(x_1, ..., x_n, Newvar)$;
6 **else**
7 Leave the head literal unchanged;
8 **end**
9 Replace each *tabled* subgoal, $p(y_1, ..., y_m)$, in $body$ with $p(y_1, ..., y_m, id(R))$;
10 **end**

Algorithm 1: Program Transformation: Adding Rule Ids

Queries are modified as follows: if the query predicate is not tabled, the query is not changed. If that predicate is tabled, an additional (last) argument is added, which contains a new variable. All tabling declarations are also modified accordingly such that `:- table p/n` is replaced with `:- table p/(n+1)`.

It is easy to see that the new program is equivalent to the original one in the sense that non-tabled queries to both programs have the same answers and the answers to the tabled predicates are the same if the last component in each answer tuple is chopped off.[2] However, now each subgoal call recorded in the log will be labeled with its host rule id.

Example 2.6 *Consider the program of Example 2.4, where path/2 is tabled while edge/2 is not. Assuming that the assigned rule ids for* `path(X,Y) :- edge(X,Y).` *and* `path(X,Y) :- edge(X,Z), path(Z,Y).` *are* r1 *and* r2 *respectively, the transformation will modify the program as follows*

```
:- table path/3.
edge(1,2).    edge(1,3).    edge(2,1).
path(X,Y,_) :- edge(X,Y).
path(X,Y,_) :- edge(X,Z), path(Z,Y,r2).
?- path(1,Y,_).
```

whose correctness can be easily verified. □

The transformation of Algorithm 1 has been implemented for $\mathcal{F}$LORA-*2* and SILK, although the form of the last argument there is made more complex to provide additional support for truth maintenance.

2.3 `Terminyzer` for Tabled Logic Engines with Subgoal Abstraction

Since `logforest`, presented in Section 2.1.2, records only table operations of an evaluation and is implemented in C, it works much faster and produces much smaller logs compared with traditional tracing facilities. This makes it possible to understand non-terminating tabled evaluation by analyzing forest logging traces and thus help to debug large programs. This section presents two such techniques, *call sequence analysis* and *answer flow analysis*, for tabled logic engines with subgoal abstraction.

[2]We assume that the programs have no aggregate functions such as *count*, which are sensitive to duplicate answers.

Both techniques are based on stopping the execution after a time limit set by the user or after the evaluation starts producing queries or answers that exceed certain size limits, and then analyzing the logs. Our examples assume that the system stops after generating queries or answers of depth greater than 10.

We should stress that due to the undecidability results mentioned in Section 1.1, no algorithm can detect non-termination in all cases unless infinite logs are available. Pragmatically, this means that, in working with `Terminyzer`, one must assume that the available logs are *long enough*.

2.3.1 Call Sequence Analysis

Recall that, with subgoal abstraction, the only way for tabled query evaluation to not terminate is when the query or its subgoals have infinitely many answers. Call sequence analysis, in this case, finds the *exact* sequence of subgoal calls to tabled predicates and, for each subgoal, its host rule's id, that lead to non-termination. Moreover, it identifies the potential sets of recursive predicates and rules that are responsible for generating increasingly deeper nested terms.

Definition 2.1 *A tabled subgoal call is said to be* **finished** *if it has been completely evaluated and all its answers have been recorded in the table. Otherwise, it is an* **unfinished** *subgoal.* □

As discussed in Section 2.1.2, when a tabled subgoal *sub* is finished, `logforest` records a log entry of the form $cmp(sub, sccnum, timestamp)$. Therefore, unfinished subgoals can be found via the following rule:

```
unfinished(Child,Parent,TS) :-
    (tc(Child,Parent,new,TS) ; nc(Child,Parent,new,TS)),
    not_exists(cmp(Child,SCCNum,TS1)).
```

Here `not_exists` is the XSB well-founded negation operator, which, in this case, existentially quantifies `SCCNum` and `TS1`. The fact $unfinished(child, parent, timestamp)$ says that unfinished subgoal *child* is called by *parent* and the event timestamp is *timestamp*. Since *parent* is waiting for the answers from *child*, *parent* is a child of another unfinished subgoal. The initial subgoal, *root*, has no parent.

Theorem 2.1 (Soundness of the call sequence analysis) *Consider a query to a program all of whose predicates are tabled, and assume that the system supports subgoal abstraction. If there are unfinished calls in the complete infinite forest logging trace, then*

i. the sequence of unfinished calls, sorted by their timestamps, is the exact *sequence of unfinished calls that caused non-termination, and*

ii. the ids of the rules that issued each of these unfinished calls appear in the last arguments of these calls. □

Proof: (*i*) Clearly, non-termination can be caused only by unfinished calls. As described in Section 2.1.2, either $tc(child, parent, stage, timestamp)$ or $nc(child, parent, stage, timestamp)$ must be logged whenever a tabled subgoal *child* is called by *parent*, and only when *child* is completely evaluated, $cmp(child, sccnum, timestamp)$ is recorded. The timestamps of these log entries preserve the sequential order of the corresponding events. Therefore, the sequence of unfinished calls defined above, sorted by their timestamps, record exactly those unfinished calls that cause one specific non-termination.

(*ii*) The program transformation described in Algorithm 1 generates a new rule id for each rule and embeds it in each of the rule's tabled body subgoals as their last argument, and these rule ids appear in each call to these subgoals just as their other arguments. Therefore, the log entry for each unfinished subgoal includes the id of the rule calling that subgoal. □

Clearly, however, one cannot obtain the complete infinite trace for a non-terminating evaluation. In practice, one would let the program execute long enough until it starts producing answers exceeding some size limits and then analyze the available portion of the log.

Knowing the exact sequence of unfinished calls, the user still needs to spend quite an amount of time understanding precisely which subgoals are forming recursive cycles. We now turn to developing a more precise machinery for this task.

Definition 2.2 *The* **unfinished-call child-parent graph** *(CPG) for a forest logging trace is a directed graph* $G_{uc} = (\mathcal{N}, \mathcal{E})$ *whose nodes are its unfinished subgoals, i.e.,* $\mathcal{N} = \{child \mid unfinished(child, parent, timestamp)\} \cup \{root\}$. *A directed*

edge (sub_1, sub_2) *is in* $\mathcal{E}$ *if and only if* sub_1 *is an unfinished parent-subgoal of* sub_2*, i.e.,* $unfinished(sub_2, sub_1, timestamp)$ *is true.* □

In Definition 2.2, subgoals that are variants of each other (i.e., identical up to the variable renaming) are treated as the same subgoal. Given an unfinished-call CPG $G_{uc} = (\mathcal{N}, \mathcal{E})$, each $sub \in \mathcal{N}$ is labeled with the timestamp of the first call to sub; it is written as $sub.timestamp$. The timestamp of the initial subgoal *root*, $root.timestamp$, is -1. The edge that corresponds to the fact $unfinished(sub_2, sub_1, timestamp)$ is *labeled* with the timestamp of this fact and is denoted $(sub_1, sub_2).timestamp$. The timestamps of nodes and edges preserve the temporal order of their creation in the forest logging trace.

An *unfinished-call path* is a path with *no repeated edges* in G_{uc}; it is called an *unfinished-call loop* if it is a cycle. An unfinished-call path of the form $[sub, sub]$ means that there is an edge $(sub, sub) \in \mathcal{E}$ and it is also an unfinished-call loop. Loops that represent the same cycles in CPG are considered to be the same and we keep only one representative for each set of such loops. For instance, $[a, b, c, a]$ and $[b, c, a, b]$ are the same loop while $[a, b, c, a]$ and $[a, c, b, a]$ are not. *Unfinished-call loops contain recursive subgoals that are potential causes of non-termination.*

Example 2.7 *Consider the query* `?- r(X)` *and the rules, below, where* `@!ruleid` *indicates the id of the corresponding rule:*

```
:- table p/1, q/1, s/1, r/1.
@!r1  p(a).
@!r2  p(f(X)) :- q(X).
@!r3  q(b).
@!r4  q(g(X)) :- p(X).
@!r5  r(X) :- r(X).
@!r6  r(X) :- p(X), s(X).
@!r7  s(f(b)).
```

The evaluation produces logs containing these unfinished calls:

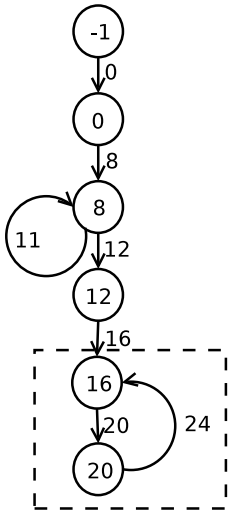
```
unfinished(r(_h9900,_h9908), root, 0)
unfinished(r(_h9870,r5), r(_h9870,_h9889), 8)
```

```
unfinished(r(_h9840,r5), r(_h9840,r5), 11)
unfinished(p(_h9810,r6), r(_h9810,r5), 12)
unfinished(q(_h9780,r2), p(_h9780,r6), 16)
unfinished(p(_h9750,r4), q(_h9750,r2), 20)
unfinished(q(_h9720,r2), p(_h9720,r4), 24)
```

This is the exact sequence of calls causing the non-termination. There are 6 unfinished subgoals as shown in Figure 2(A), where each subgoal's timestamp and the host rule's id are also given. Its unfinished-call CPG has 7 edges, shown in Figure 2(B), where

timestamp	subgoal	ruleid
-1	root	
0	r(_h9900,_h9908)	_h9908
8	r(_h9870,r5)	r5
12	p(_h9810,r6)	r6
16	q(_h9780,r2)	r2
20	p(_h9750,r4)	r4

(A)



(B)

Figure 2: Unfinished-Call CPG of Example 2.7

timestamps are used to represent nodes instead of actual subgoals and each edge is labeled with its timestamp. There are two unfinished-call loops in the CPG: $[8, 8]$ *and* $[16, 20, 16]$. □

Algorithm 11, below, constructs the unfinished-call CPG $G_{uc} = (\mathcal{N}, \mathcal{E})$ from the set of unfinished calls of a forest logging trace. Construction starts by adding the root call to the CPG. Then, for each log record of the form $unfinished(child, parent, timestamp)$, the node *child* and the edge $(parent, child)$ are added, if *child* has not been added before. All unfinished calls are processed in the order of their timestamps, i.e., their addition to the log, which is also the order in which these unfinished calls are made during evaluation.

Thus, when the record $\mathit{unfinished}(\mathit{child}, \mathit{parent}, \mathit{timestamp})$ is encountered, we know that *parent* must have been added to the graph as an child-subgoal of its parent, i.e., $\mathit{unfinished}(\mathit{parent}, p', \mathit{timestamp}')$ must be true for some p' and $\mathit{timestamp}' < \mathit{timestamp}$. We have two cases:

1. $\mathit{child} \in \mathcal{N}$. The evaluation calls a previously issued subgoal.

2. $\mathit{child} \notin \mathcal{N}$. A new subgoal is called and a new node is added to the graph.

In the first case, an unfinished-call loop exists, so the current evaluation path of *parent* is suspended and alternative derivations is explored. This implies an important property of unfinished-call CPGs: an unfinished-call loop is created out of an *acyclic* path always by adding a final edge of the form $(\mathit{sub}_1, \mathit{sub}_2)$, where $\mathit{sub}_1.\mathit{timestamp} \geq \mathit{sub}_2.\mathit{timestamp}$. We call such an edge a *critical loop edge* — see the edges labeled with *11* and *24* in Figure 2 of Example 2.7.

```
1  Let UC be the set of unfinished calls, N = {root}, root.timestamp = −1, and
   E = ∅;
2  while UC ≠ ∅ do
3      Choose unfinished(child, parent, timestamp) from UC, where timestamp is
       the smallest among UC, and remove it;
4      if child ∉ N then
5          N = N ∪ {child};
6          child.timestamp = timestamp;
7      end
8      E = E ∪ {(parent, child)};
9      (parent, child).timestamp = timestamp;
10 end
11 return G_uc = (N, E)
```

Algorithm 2: Unfinished-Call CPG Construction

If critical loop edges are taken out, any unfinished-call CPG becomes a connected directed acyclic graph (i.e., a tree) in which every edge goes from a node with a smaller timestamp to a node with a larger timestamp. An unfinished-call path P, which connects the *root* node to a subgoal node *sub*, can be obtained by posing the query `?- uc_path`(sub, P) to the following rules. After computing all unfinished-call paths, without critical loop edges, from *root* to other nodes, all distinct unfinished-call loops can be computed by checking whether there is a critical loop edge from the

last vertex of a path to any other node in the same path. Consider an unfinished-call path $P = [root, sub_1, \ldots, sub_n]$. If there is a critical loop edge (sub_n, sub_i), $1 \leq i \leq n$, then the second part of P from sub_i to sub_n, $[sub_i, \ldots, sub_n, sub_i]$, is an unfinished-call loop.

```
:- table reversed_uc_path/2.
uc_path(C,P) :- reversed_uc_path(C,RevP), reverse(RevP,P).
reversed_uc_path(C,[C,root]) :- unfinished(C,root,_Timestamp).
reversed_uc_path(C,[C|P]) :-
    unfinished(C,Parent,Timestamp),
    Parent.timestamp < Timestamp,
    reversed_uc_path(Parent,P).
```

Theorem 2.2 (Completeness of the call sequence analysis) *Consider a query and a program all of whose predicates are tabled and assume that the system supports subgoal abstraction. If the evaluation does not terminate, then*

i. there is at least one unfinished-call loop in the unfinished-call CPG constructed for the complete infinite forest logging trace, and the loop's subgoals are responsible for the generation of infinite number of answers, and

ii. the last arguments of these subgoals specify the rule ids from whose bodies these subgoals were called. □

Proof: (i) There has to be at least one loop. Suppose there is no unfinished-call loop in the corresponding unfinished-call CPG $G_{uc} = (\mathcal{N}, \mathcal{E})$. Subgoal abstraction ensures that only a finite number of calls to tabled predicates can exist, so G_{uc} is a finite graph. Since there is no unfinished-call loop, there must be terminal nodes that have no outgoing edges. Let $\mathcal{N}^t \subseteq \mathcal{N}$ denote this set of nodes. It means that their SLG-children are not in $\mathcal{N}$, i.e., they are not unfinished subgoals. Therefore the SLG-children of $\mathcal{N}^t$ are either completely evaluated tabled subgoals or base facts. But then, after long enough time, all subgoals in $\mathcal{N}^t$ should have been completely evaluated and completed. This contradicts the assumption that $\mathcal{N}^t \subseteq \mathcal{N}$, i.e., the subgoals in $\mathcal{N}^t$ are unfinished.

At least one of the loops must be responsible for the generation of an infinite number answers; otherwise all answers would be derived and the evaluation would terminate.

(*ii*) This is proved by the same argument as in Theorem 2.1 (ii). □

Since complete infinite traces for non-terminating computations cannot be had, in practice one would let the program execute long enough until it starts producing answers exceeding some size limits, and then analyze the available portion of the log. Clearly, this opens up the possibility for false negatives, i.e., for blaming sequences of calls that in actuality do terminate after a long time. However, even in this case, such sequences are possible computational bottlenecks and identifying them is useful in its own right.

Identification of the exact rules that cause infinite computations in Theorems 2.1 and 2.2 (and later in Theorems 2.3 and 2.4) is a major advance in debugging non-termination as rule ids can be gainfully exploited by graphical tools, such as the one built for SILK.

2.3.2 Answer Flow Analysis

Call sequence analysis finds the exact sequences of subgoal calls and the corresponding host rules that are involved in a non-terminating computation. These subgoals are marked as incomplete in the trace because they are waiting for answers for themselves or their children. However, many of these subgoals do not actually produce an infinite number of answers and they are not true reasons for non-termination. A much more useful outcome of the call sequence analysis are the sets of recursive predicates that form the unfinished-call loops and cause generation of infinitely many answers. Unfortunately, the number of such loops in an unfinished-call CPG can be exponential and, moreover, not all of these loops may be the reason for non-termination. For instance, Figure 2 has two unfinished-call loops, but only $[16, 20, 16]$ is at fault. This problem of precisely identifying the faulty loops is dealt with using *answer flow analysis*, described below.

We say that an unfinished-call loop is a *culprit* if it is a cause for non-termination. Answer flow analysis looks for the log entries that specify the answers being returned to parents (the *ar*-facts and *dar*-facts) at the end of the `logforest` trace and

produces child-parent relationships among unfinished subgoals. These child-parent relationships help to identify precisely which unfinished-call loops are culprits, so we could track how answers percolate through the unfinished subgoals.

When there are infinitely many answers, each new answer, *ansr*, to an unfinished subgoal, *sub*, is returned to the parents of *sub* and these parents use *ansr* to derive their own answers. The newly derived answers for the parents of *sub* are returned to the parents of the parents, and this gives rise to an endless process in which subgoals continue to receive, derive, and return answers. An *answer-flow child-parent sequence* is the sequence of child-parent pairs found in all the log entries for answers returned to parents; it captures the child-parent relationships in the above endless process. The pairs of an answer-flow child-parent sequence are sorted by their creation order (*timestamp*). A child might continue returning multiple answers to a certain parent before the parent starts deriving its own answers. In this case, only one child-parent pair is recorded for all such answer returns, since all these pairs are identical.

Definition 2.3 *An answer-flow child-parent sequence, cps, contains a* **child-parent pattern***, cpp, if cpp is a* finite *subsequence of cps such that* $cps = prefix \bullet cpp^{\alpha}$*, where* $\bullet$ *is the sequence concatenation operator,* $\alpha > 1$ *is a positive integer or* ω *(the first infinite ordinal), and* cpp^{α} *represents the concatenation of* α *cpp's. We call* cpp^{α} *the cpp-***suffix** *of cps.* □

Example 2.8 *For instance,* $[(c_2, p_2), (c_3, p_3)]$ *is a child-parent pattern of length two in* $[(c_1, p_1), (c_2, p_2), (c_3, p_3), (c_2, p_2), (c_3, p_3)]$*, and its* $[(c_2, p_2), (c_3, p_3)]$*-suffix is* $[(c_2, p_2), (c_3, p_3), (c_2, p_2), (c_3, p_3)]$. □

Definition 2.4 *The* **optimal child-parent pattern** *in a child-parent sequence cps is the* shortest *child-parent pattern, cpp, such that the cpp-suffix is the longest in cps (longest by containment among all suffixes of child-parent patterns in cps).* □

For an infinite trace, its child-parent sequence and the *cpp*-suffix of any of its child-parent patterns are infinite, but all child-parent patterns have finite lengths. Since there can be only a finite number of unfinished subgoals due to subgoal abstraction, the answer-flow child-parent sequence of a non-terminating trace must have an optimal child-parent pattern (Theorem 2.3 below).

Given a child-parent sequence, let *pattern* be the subsequence containing the last n elements in the sequence. The predicate `pattern`($cps, length, pattern, times$) specifies the number of times *times* a child-parent pattern *pattern* of length *length* repeats at the end of *cps*. Patterns of different lengths can be computed by posing the query `?- pattern`($cps, length, Pattern, Times$) to the following rules, where the *length* parameter successively assumes the values *1*, *2*, and so on. In this way, we will either find an optimal child-parent pattern or determine that there is no pattern.

```
pattern(CPS,Len,Pat,Times) :-
    length(Pat,Len),
    %% This binds Pat to the suffix of CPS of length Len
    append(CPSPrefix,Pat,CPS),
    aux_pattern(CPSPrefix,Pat,Times).
aux_pattern(CPS,Pat,Times) :-
    append(CPSPrefix,Pat,CPS), !,
    pattern(CPSPrefix,Pat,TimesPrefix),
    Times is TimesPrefix+1.
aux_pattern(CPS,Pattern,1).
```

Example 2.9 *The child-parent sequence of the forest logging trace for Example 2.7 is as follows:*

```
cps = [(q(_h599,r2),p(_h599,r4)), (p(_h599,r4),q(_h599,r2)),
       (q(_h619,r2),p(_h619,r4)), (p(_h639,r4),q(_h639,r2)),
       (q(_h659,r2),p(_h659,r4)), (p(_h679,r4),q(_h679,r2)),
       (q(_h699,r2),p(_h699,r4)), (p(_h719,r4),q(_h719,r2)),
       (q(_h739,r2),p(_h739,r4)), (p(_h759,r4),q(_h759,r2)),
       (q(_h779,r2),p(_h779,r4))].
```

There are two child-parent patterns in the above cps. The first one is $cpp_1 = [(\mathtt{p(\_h759,r4)}, \mathtt{q(\_h759,r2)}), (\mathtt{q(\_h779,r2)}, \mathtt{p(\_h779,r4)})]$ *of length two and it repeats five times. The second one is* $cpp_2 = cpp_1^2$ *of length four, twice repeated. The optimal child-parent pattern is* cpp_1*, as it covers* $2 \times 5 = 10$ *entries in cps compared to* cpp_2*, which covers* $4 \times 2 = 8$ *entries.* □

As in the call sequence analysis, child-parent relationships in the optimal child-parent pattern for a forest logging trace can be modeled as a graph.

Definition 2.5 *Let cpp_{opt} be the optimal child-parent pattern for a forest logging trace. The trace's* **answer-flow child-parent graph** *(CPG) is a directed graph $G_{af} = (\mathcal{N}, \mathcal{E})$, where its nodes are the set of children and parent-subgoals in cpp_{opt}, i.e., $\mathcal{N} = \{sub \mid (sub, ...) \in cpp_{opt}$ or $(..., sub) \in cpp_{opt}\}$, and its edges are the child-parent pairs in cpp_{opt}, i.e., $\mathcal{E} = \{(child, parent) \mid (child, parent) \in cpp_{opt}\}$.* □

A path in G_{af} is called an *answer-flow path*; such a path is called an *answer-flow loop* if it is a cycle. Two answer-flow loops that consist of the same nodes and edges are considered to be the same and we will keep only one representative loop in such a case. Answer-flow paths and loops represent information flow among unfinished subgoals in the infinite process of answer derivation. All answer-flow paths from node *child* to node *parent* can be computed using the predicate `af_path`(*child*, *parent*, *path*); all answer-flow loops starting from *child* can be computed using the predicate `af_loop`(*child*, *loop*), defined below.[3]

```
:- table af_path/3.
af_path(Child,Parent,[Child]) :- optimal_cpp(Child,Parent).
af_path(Child,Parent,[Child|P]) :-
    optimal_cpp(Child,Sub),
    af_path(Sub,Parent,P),
    \+ member(Child,P).
af_loop(Sub,Loop) :- af_path(Sub,Sub,Loop).
```

Example 2.10 *Consider cpp_1, the optimal child-parent pattern of Example 2.9. Its answer-flow graph is the subgraph shown inside the rectangle in Figure 2(B). The only answer-flow loop is [16, 20, 16], which tells us that subgoal p called from rule r4 and subgoal q called from rule r2 return answers to each other in an infinite answer derivation loop.* □

Theorem 2.3 (Completeness of the answer flow analysis) *Consider a query to a program all of whose predicates are tabled and assume that the inference engine supports subgoal abstraction. If the query evaluation does not terminate, then:*

[3] We use *optimal_cpp*(*child*, *parent*) to denote the fact that (*child*, *parent*) is in cpp_{opt}.

i. there is an optimal child-parent pattern in its complete infinite trace,

ii. $G_{af} = (\mathcal{N}, \mathcal{E})$ *contains at least one answer-flow loop,*

iii. every $sub \in \mathcal{N}$ *appears in at least one answer-flow loop, and*

iv. each edge $(sub_1, sub_2) \in \mathcal{E}$*, where* sub_1 *is of the form* $predicate(..., ruleid)$*, tells us that* sub_2 *calls* sub_1 *from the body of a rule with the id* $ruleid$*.* □

Proof: (i) There can be only a finite number of unfinished subgoals due to subgoal abstraction, and thus there must be at least one child-parent pattern. Otherwise the evaluation would have terminated. Therefore, an optimal child-parent pattern must exist in the forest logging trace.

(ii) Suppose there is no answer-flow loop in G_{af}. There must be a set $\mathcal{N}^t \subseteq \mathcal{N}$ of terminal nodes and, since these nodes are terminal, the graph has no edges going out of $\mathcal{N}^t$. The SLG-children of these terminal nodes are therefore not in $\mathcal{N}$ and answers for these SLG-children are *not* being repeatedly derived. Recall that, due to subgoal abstraction, G_{af} can have only a finite number of nodes and, if we let the engine run long enough, all possible edges in G_{af} will be generated and further computation will not change that graph. Therefore, the nodes for which answers are not derived repeatedly cannot stay unfinished (in the sense of unfinished SLG subgoals) infinitely long. So, after a while, all SLG-children of $\mathcal{N}^t$ must either become completely evaluated tabled subgoals or they must have been base facts all along. This implies that, given enough time, all subgoals in $\mathcal{N}^t$ would be completed, contrary to the assumption that $\mathcal{N}^t \subseteq \mathcal{N}$. Therefore, there must be an answer-flow loop.

(iii) If $sub \in \mathcal{N}$ and sub is not contained in any answer-flow loop, then sub's evaluation would have been completed and it cannot be part of any child-parent pattern, a contradiction.

(iv) Consider an edge $(sub_1, sub_2) \in \mathcal{E}$, where sub_1 is of the form $predicate(..., ruleid)$. We know sub_2 calls sub_1 and sub_1 keeps returning answers to sub_2, by the definition of the edges in G_{af}. It follows from the argument made in (ii) of Theorem 2.1 that this call of sub_1 must have been made from the rule with the id $ruleid$. □

Theorem 2.4 (Soundness of the answer flow analysis) *Consider a query to a program all of whose predicates are tabled. If the complete infinite trace of that query has an optimal child-parent pattern then the query evaluation does not terminate.* □

Theorem 2.4 follows directly from the definitions, since the optimal child-parent pattern captures the information flow among unfinished subgoals in a non-terminating computation. These theorems tell us that the set of subgoals contained in the optimal child-parent pattern of a non-terminating trace, i.e., the nodes of the pattern's answer-flow CPG, are *exactly* the subgoals for which infinitely many answers continue being derived. We call these subgoals the *culprit* unfinished subgoals.

In call sequence analysis, an unfinished-call CPG is constructed and the suspected unfinished-call loops are flagged. Similarly, in answer-flow analysis, one builds answer-flow CPG and computes culprit loops, which shed light on how answers flow among culprit subgoals. The following Theorem 2.5 connects these two approaches.

Theorem 2.5 (Relationship between unfinished-call and answer-flow CPGs) *Let $G_{uc} = (\mathcal{N}_{uc}, \mathcal{E}_{uc})$ be the unfinished-call CPG and let $G_{af} = (\mathcal{N}_{af}, \mathcal{E}_{af})$ be the answer-flow CPG for a non-terminating forest logging trace. Then $\mathcal{N}_{af} \subset \mathcal{N}_{uc}$, and for every edge $(child, parent) \in \mathcal{E}_{af}$ there is an edge $(parent, child) \in \mathcal{E}_{uc}$. Furthermore, every answer-flow loop is a culprit unfinished-call loop.* □

Proof: If $sub \in \mathcal{N}_{af}$ then it must be an unfinished subgoal, since answers to *sub* continue to be derived. That is, the evaluation of *sub* has not been completed and $\mathcal{N}_{af} \subseteq \mathcal{N}_{uc}$. In fact, we even have that $\mathcal{N}_{af} \subset \mathcal{N}_{uc}$, since $root \in \mathcal{N}_{uc} \setminus \mathcal{N}_{af}$. For any edge $(child, parent) \in \mathcal{E}_{af}$, we know that *child* returns answers to *parent*, i.e., it is issued in a SLG tree for *parent*. Therefore $(parent, child) \in \mathcal{E}_{uc}$. This implies that any answer-flow loop is also an unfinished-call loop. □

Example 2.11 *Let $G_{uc} = (\mathcal{N}_{uc}, \mathcal{E}_{uc})$ be the unfinished-call CPG of Example 2.7 and $G_{af} = (\mathcal{N}_{af}, \mathcal{E}_{af})$ be the answer-flow CPG of Example 2.10. It is easy to verify that Theorem 2.5 holds.* □

Theorem 2.6 (No false-positives for finite traces) *If the evaluation of a query, Q, terminates, then both the unfinished-call CPG and the answer-flow CPG for Q's forest logging trace are empty.* □

Proof: We know that the set of nodes of the unfinished-call CPG for a trace is its set of unfinished subgoals. In case of a terminating evaluation, all subgoals are completed and thus there are no unfinished subgoals, i.e., its unfinished-call CPG must be empty, as it has no nodes. It follows from Theorem 2.5 that the corresponding answer-flow CPG is likewise empty. □

Theorem 2.6 assures that neither the unfinished call nor the answer flow analysis yield false-positive results for finite traces. Of course, for infinite traces, false-positives are possible, as one can inspect only a finite prefix in such cases.

2.3.3 Auto-Repair of Rules

Call sequence analysis tells us the exact sequence of unfinished calls and their respective host rule ids in the original program that initiate a non-termination, and answer flow analysis further identifies a subset of these unfinished subgoals as culprit. However, sometimes query evaluation does not terminate not because the query has infinitely many answers but because one of its subgoals does. In such cases, the query *may* terminate if a different evaluation order for its subgoals is used. This section describes one such heuristic technique for fixing certain non-termination queries by delaying the evaluation of unfinished subgoals.

Suppose that $G_{uc} = (\mathcal{N}_{uc}, \mathcal{E}_{uc})$ is the unfinished-call CPG of a non-terminating evaluation. For each $(parent, child) \in \mathcal{E}_{uc}$, we know that the call to *child* from *parent* has not been completed. Moreover, we know:

- the host rule for this call, and
- the common set of the unbound arguments of *parent* and *child*, which are also the arguments whose bindings are to be derived.

To reduce the possibility that *parent* gets an infinite number of bindings from *child* and thus diminish the possibility of non-termination caused by that call to *child*, we can delay the evaluation of *child* in the host rule until the aforesaid unbound arguments get bound. If later in the evaluation it is established that the arguments cannot be bound, the delay of *child* ceases and the subgoal is executed. Similar evaluation delays can be applied to all unfinished calls in $\mathcal{E}_{uc}$, which constitutes our auto-repair technique.

Example 2.12 *Consider the evaluation and unfinished-call CPG of Example 2.7, where edge (12, 16) which represents* $(p(\_h9780, r6), q(\_h9780, r2))$*. Their common set of arguments consists of their only argument, i.e., their first argument, and the rule id contained in* $q(\_h9780, r2)$ *is r2. Therefore, our technique will delay the evaluation of* $q(\_h9780)$ *in the rule r2 until its first argument becomes bound.* □

$\mathcal{F}$LORA-*2* and SILK support delay quantifiers of the form `wish`(*cond*) and `must`(*cond*), where *cond* is an `and/or` combination of `ground`(*variables*) and `nonvar`(*variables*). This is similar to the `when/2` predicate found in many prologs with the difference being that the delayed subgoal is eventually tried even if the binding conditions are not met. A delayed literal is of the form *delay-quantifier*`^`*goal*. When such a literal is to be executed, the attached *delay-quantifier* is checked. If the quantifier's condition is satisfied, *goal* is executed immediately. Otherwise, the literal is delayed until such time that the condition is satisfied. If the condition is eventually satisfied, *goal* is called. If the engine determines that satisfying the quantifier's condition is impossible, *goal* is called anyway (in case of the `wish` quantifier) or an error is issued (in case of the `must` quantifier).

Example 2.13 *Consider the program of Example 2.7. Our auto-repair heuristic will delay the unfinished subgoals and modify the program as follows:*

```
@!r1  p(a).
@!r2  p(f(X)) :- wish(ground(X))^q(X).
@!r3  q(b).
@!r4  q(g(X)) :- wish(ground(X))^p(X).
@!r5  r(X) :- wish(ground(X))^r(X).
@!r6  r(X) :- wish(ground(X))^p(X), s(X).
@!r7  s(f(b)).
?- wish(ground(X))^r(X).
```

The modified program successfully terminates with an answer `X = f(b).` □

It should be clear, however, that the above is only a heuristic and no automatic fool-proof auto-repair technique is possible, in general. Since `Terminyzer` serves as a debugging tool, the user needs to manually suggest the appropriate delay quantifiers

to unfinished subgoals upon the detection of non-termination. A graphical interface can help to ease the process.

2.4 `Terminyzer` for Tabled Logic Engines without Subgoal Abstraction

We now turn to non-termination analysis that does *not* rely on subgoal abstraction. This relaxation makes `Terminyzer` applicable in more tabled logic engines since none of them (except XSB) currently supports subgoal abstraction. As discussed in Chapter 1, non-termination may then also be caused by generation of infinitely many subgoals. In this case, `Terminyzer` analyzes the sequence of unfinished subgoals and reports the predicates and their respective host rule ids that form increasingly deep nested subgoals. As before, we assume that users stop the execution after a time limit or when subgoals or answers become too large.

For an unfinished subgoal, its *simplified* version is constructed out of the subgoal's predicate and the rule id as $predicate(ruleid)$. For instance, $p(f2(f1(a)), b, r3)$ is simplified to $p(r3)$. The *simplified unfinished subgoal sequence* is the sequence of simplified unfinished subgoals sorted by the order of their first appearance in the trace. When non-termination is caused by an infinite number of subgoals, these subgoals must have increasingly deeply nested terms. Since a finite program has only a finite number of predicates and functors, there must be repetitions in the aforesaid sequence of simplified unfinished subgoals.

Definition 2.6 *A simplified unfinished subgoal sequence, uss, contains a* **subgoal pattern***, subp, if subp is a* finite *subsequence of uss such that* $uss = prefix \bullet subp^{\alpha}$*, where* $\alpha > 1$ *is a positive integer or* ω *(the first infinite ordinal). The suffix* $subp^{\alpha}$ *is called the subp-***suffix** *of uss. The* **optimal subgoal pattern** *in a simplified unfinished subgoal sequence uss is the* shortest *subgoal pattern, subp, such that its subp-suffix is the longest in uss (longest by containment among all suffixes of subgoal patterns in uss).* □

Similar to the computation of optimal child-parent patterns in answer flow analysis of Section 2.3.2, the *optimal subgoal pattern* of a simplified unfinished subgoal

sequence can be computed. For an infinite trace, its simplified unfinished subgoal sequence *uss* and the *subp*-suffixes of any of its subgoal patterns are infinite, but all subgoal patterns have finite lengths. Theorem 2.7, below, tells us that non-termination implies the existence of an optimal subgoal pattern. This pattern will show which subgoals in which rules recursively call one another and create increasingly deeper and deeper terms.

Example 2.14 *The evaluation of the query* `?- r(a)` *given the program*

```
@!r1  p(a).
@!r2  p(X) :- q(f1(X)).
@!r3  q(X) :- p(f2(X)).
@!r4  r(X) :- r(X).
@!r5  r(X) :- p(X), s(X).
@!r6  s(a).
```

produces a forest log containing infinitely many unfinished calls, the first of which are:

```
unfinished(r(a,_h46), root, 0).
unfinished(r(a,r4), r(a,_h27), 8).
unfinished(r(a,r4), r(a,r4), 11).
unfinished(p(a,r5), r(a,r4), 12).
unfinished(q(f1(a),r2), p(a,r5), 16).
unfinished(p(f2(f1(a)),r3), q(f1(a),r2), 19).
unfinished(q(f1(f2(f1(a))),r2), p(f2(f1(a)),r3), 22).
unfinished(p(f2(f1(f2(f1(a)))),r3), q(f1(f2(f1(a))),r2), 25).
unfinished(q(f1(f2(f1(f2(f1(a))))),r2), p(f2(f1(f2(f1(a)))),r3), 28).
unfinished(p(f2(f1(f2(f1(f2(f1(a)))))),r3),
           q(f1(f2(f1(f2(f1(a))))),r2), 31).
unfinished(q(f1(f2(f1(f2(f1(f2(f1(a))))))),r2),
           p(f2(f1(f2(f1(f2(f1(a)))))),r3), 34).
```

Its simplified unfinished subgoal sequence is `[root, r(_h46), r(r4), r(r4), p(r5), q(r2), p(r3), q(r2), p(r3), q(r2), p(r3), q(r2)]`. *It has an optimal subgoal pattern as* `[p(r3), q(r2)]`, *which means that the predicates q in rule r2 and p in rule r3 are the ones causing the generation of increasingly deep subgoals.* □

Theorem 2.7 (Soundness and completeness) *Consider a query to a tabled program and assume that the engine does* not *perform subgoal abstraction. The forest logging trace has an optimal subgoal pattern if and only if the computation is non-terminating due to infinitely many subgoals.* □

Proof: (*Soundness*) If an optimal subgoal pattern exists then the evaluation does not terminate. Indeed, if the evaluation terminates, there would be no unfinished subgoals and thus no optimal subgoal pattern. Suppose the evaluation produces only a finite number of subgoals. Since there are only two causes for non-termination in a tabled logic engine *without* subgoal abstraction—infinite number of answers or infinite number of subgoals—non-termination must be due to an infinite number of answers. As described in Section 2.3.2, this means that a *finite* subset of these subgoals, contained in the trace's optimal child-parent pattern, keeps receiving, deriving, and returning answers. Since there is an optimal subgoal pattern, this requires certain predicates from certain rules to recursively and *repeatedly* call each other, supplying deeper and deeper terms as arguments. These calls would then be causing new subgoals of bigger and bigger sizes to appear in the child-parent pattern, contrary to the assumption that the evaluation produces only a finite set of subgoals.

(*Completeness*) As discussed above, when non-termination happens because of an infinite number of subgoals, these subgoals must have increasingly deep function terms as arguments, and these subgoals' predicates must be recursive. Otherwise there would be only a finite number of terms in a finite program. Therefore, there must be repetitions in the simplified unfinished subgoal sequence of the trace, which implies that there must be an optimal subgoal pattern. □

Once the optimal subgoal pattern is computed, the user can easily find the subgoals and the rules that are likely causes of non-termination. Note that without subgoal abstraction, the auto-repair technique presented in Section 2.3.3 does not apply here since no subgoal reordering can cause the query to terminate.

2.5 Experiments

`Terminyzer` has been implemented for the $\mathcal{F}$LORA-*2* and SILK systems, and we report our experiments below. All tests were performed on a dual core 2.4GHz Lenovo X200

with 3 gigabytes of main memory running Ubuntu 11.04 with Linux kernel 2.6.38. The sources of the test programs as well the reports produced by `Terminyzer` are available online.[4]

2.5.1 Test Programs

Here we include four test cases: T_1, T_2, T_3, and T_4, and none of them terminates. The first three tests are performed with subgoal abstraction enabled, while T_4 was tested without subgoal abstraction. T_1 is the query and the rule set of Example 2.7. T_2 and T_3 are very large programs which were derived from $\mathcal{F}$LORA-*2* programs used in the SILK project. T_2 has 844 rules and facts, and its corresponding XSB program (after $\mathcal{F}$LORA-*2*-to-XSB translation) is estimated to have 2,000 rules and facts. T_3 consists of 4,774 rules and 919 facts, and its XSB program has over 1,000 facts and over 5,500 rules.[5] T_4 is the program of Example 2.14.

For T_1 and T_2, we set XSB to abort after the answer depth reached 30. For T_3, we let the evaluation continue until all available memory was consumed. The reason is that T_3 is a really complex program, and in order to get a usable prefix of its infinite trace, we have to let it run "long enough." The execution of T_2 produces a log trace of 3 megabytes with around 26,000 log entries, and the trace for T_3 is in excess of 2 gigabytes with more than 14 million log entries.

2.5.2 Test Results and Analysis

`Terminyzer` produced expected results in all the test cases. For T_1, `Terminyzer` constructed the unfinished-call graph shown in Figure 2 and identified its culprit loop. The auto-repair technique presented in Section 2.3.3 successfully fixed the non-termination problem as demonstrated in Example 2.13.

For T_2, `Terminyzer` determined that the predicate `entailed(X)` of the following rule was generating and infinite number of answers:

```
entailed(conjunction(Antecedent1,Antecedent2)) :-
    entailed(Antecedent1), entailed(Antecedent2).
```

[4] `http://rulebench.projects.semwebcentral.org/terminyzer+`

[5] We also tested other, fairly large real programs from the SILK project with similarly positive results.

The heuristic auto-repair method of Section 2.3.3 fails to fix this non-terminating query since it is the query itself, not its subqueries, that has infinitely many answers.

For T_3, the unfinished-call CPG has *14* nodes and *34* edges, and its answer-flow CPG has *9* nodes and *28* edges. Our auto-repair method successfully removes the cause of non-termination and the remedied program terminates with one answer. We should mention that an experienced knowledge engineer spent hours debugging T_3 — all in vein.

For T_4, `Terminyzer` successfully identified the optimal subgoal pattern, as described in Example 2.14.

2.5.3 Computation Times

For T_1, T_2, and T_4, `Terminyzer` took a tiny fraction of one second for each program. For the much more complex T_3, it took *170* seconds. Compared to the fruitless hours spent by our knowledge engineer, `Terminyzer` appears to be a much more inviting alternative.

One optimization would be to split forest logging traces into multiple files for different analyzers, since different analysis approaches largely make use of different entries in the trace: call sequence analysis uses only the *tc*, *nc*, and *cmp*-facts; answer-flow analysis needs *ar* and *dar*-facts; while `Terminyzer` for logic engines without subgoal abstraction uses *tc*, *nc* and *cmp*-facts. Entries that are irrelevant for a particular analysis can be deleted thereby significantly reducing the size of the data set that we need to deal with. This optimization is implemented in `Terminyzer` as a pre-processor.

2.6 Related Work

There have been many works on termination analysis for logic programs [BAK91, Sah93, SD94, DDSL+98, She97, OCM00, SYY01, VDSS01, LSS04, BCG+07, NDS07, NGSKDS08, SkGS+10, SDSV10] while *non*-termination analysis received much less attention [NM99, Pay07, PM06, SYY01, VDS09, SDSV10, VDS11]. Most of these studies are either *norm*-based or *transformation*-based. In norm-based approaches [BAK91, Sah93, SD94, DDSL+98, She97, SYY01, VDSS01, LSS04, BCG+07, SDSV10, VDS11], termination analysis is performed by proving certain well-founded

sufficient conditions for termination, which involve norms, i.e., abstractions of the size of a term (e.g., the number of symbols, depth, etc.). Transformation-based algorithms [NM99, OCM00, Pay07, PM06, NGSKDS08, SkGS+10] rewrite logic programs so that the termination property of the rewritten program could be used to prove termination of the original program.

There are three main points that differentiate `Terminyzer`. First, a log-based approach to *debugging* expounded by `Terminyzer` is fundamentally different from the works on *proving* termination. We do not aim to prove termination because if a query terminates then there is nothing for `Terminyzer` to do. Second, the problems discussed in most previous work of the subject—except [DDSL+98, VDSS01]—are non-issues in our framework, since they stem from the severe incompleteness of the Prolog inference mechanism and, therefore, do not apply to the inference engines under consideration. Third, `Terminyzer` aims at helping the programmer to debug programs *without syntactic restrictions*. All other approaches perform static or dynamic analysis in order to *prove* termination or non-termination for *restricted* classes of logic programs, such as function-free programs, positive programs, etc. These restrictions, if at all stated, are typically very strong; stated or not, they always exist because both of the above problems are undecidable. This also applies to [DDSL+98, VDSS01], which are the only works that study the termination problem for tabling engines.

Among all these previous studies, only the loop checker approach in [SYY01] resembles our analysis of non-termination in the *absence of subgoal abstraction*. This work aims at detecting repetitions of subgoals and clauses, which are akin to `Terminyzer`'s optimal subgoal patterns. However, there are two major differences between `Terminyzer`'s analysis in Section 2.4 and the loop checkers in [SYY01]. First, Shen et al. work with Prolog without tabling. For instance the following query:

```
p(X) :- p(f(X)).
?- p(X).
```

terminates without answers in our framework (with subsumptive tabling or subgoal abstraction) and thus is a non-issue at all, while their loop checker will report non-termination because it detects an infinite SLD-derivation. Second, they perform static analysis of the original program clauses and try to detect possible loops, while `Terminyzer` analyzes logs for actual execution. For instance, this query

```
p(X) :- p(f(X)).
p(f(a)).
?- p(a).
```

will be reported as terminating in their framework since there is a successful SLD-derivation. However, this is a drawback because this analysis considers only *some* derivations, while Prolog may explore more. For instance, in the above example, if the user asks for *another answer* by typing a ";" then Prolog will go into an infinite loop. So, in that sense, this analysis is overly optimistic and not completely adequate. In contrast, `Terminyzer` would consider the actual executions. For tabled engines *without* subgoal abstraction that, like Prolog, return one answer at a time (e.g., the batched engines of XSB and YAP), `Terminyzer` will report the first successful derivation of `p(a)` as terminating and the subsequent ones as non-terminating. Furthermore, *with* subgoal abstraction, the computation terminates and `Terminyzer` will report this properly.

Chapter 3

Cost-Based Optimization

3.1 Preliminaries

Consider a n-ary predicate $p(x_1, \ldots, x_n)$. If p is a base predicate, then its associated set of facts are denoted by $factset(p)$. The *value sequence*, v_i $(1 \leq i \leq n)$, is the *sorted* sequence of x_i-values that are present in $factset(p)$, and v_i^j is the j-th value of v_i. The *frequency*, f_i^j, of v_i^j is the number of facts in $factset(p)$ with $x_i = v_i^j$. We will use v^i to denote the i-th element of a sequence v and $\|...\|$ to denote the length of a sequence or the cardinality of a set. Since we are dealing with discrete values in finite relations, all argument values can be assumed to be integers. Without loss of generality, we adopt this assumption in the sequel, for simplicity.

Definition 3.7 *Given a fact-set for an n-ary predicate $p(x_1, \ldots, x_n)$, the* **data distribution***, d_i, for x_i is the sequence of value-frequency pairs $[(v_i^1, f_i^1), \ldots, (v_i^m, f_i^m)]$ where $m = \|v_i\|$.* □

Data distribution is the basis for size estimation in all cost-based query optimizers, but this information is normally too large to store and use efficiently. One critical step of all size estimation algorithms is to partition data distributions into *distribution segments* and summarize these segments in such a way that they can be approximated efficiently both in time and space. Histogram is one such summarization method that groups values of similar frequencies into buckets and estimates the frequencies of values in each bucket in a uniform and efficient way [Ioa03, PHIS96].

The two most important aspects in constructing histograms are *partition rules* and *value frequency approximation*. Partition rules describe how the value-frequency pairs in data distributions are grouped and summarized into buckets, and value frequency approximation is an algorithm that estimates the values and their frequencies in each individual bucket.

There are many different partition rules and value frequency approximation algorithms which provide various computational complexities and estimation accuracy and one is free to choose those that work best in their applications. In this thesis, we use the *maxdiff* partition rule (formally defined later), which groups a data distribution into β partitions using its $\beta - 1$ largest frequency differences as differentiators, for its simplicity and efficiency. In term of value frequency approximation, all values are assumed to be *uniformly distributed* and thus their frequencies are *averaged*. However, our size estimation algorithms can be easily adjusted to work for *any* other partition rule.

Definition 3.8 *Given a data distribution* $d = [(v^1, f^1), \ldots, (v^n, f^n)]$ *and the number of partitions* β $(1 \leq \beta \leq n)$, **maxdiff partition rule** *groups* d *into* β *partitions as* $d^{[1]} \mid \ldots \mid d^{[\beta]}$ *such that*

- d *is the concatenation of* $d^{[1]}, \ldots, d^{[\beta]}$, *i.e.,* $d = d^{[1]} \bullet \ldots \bullet d^{[\beta]}$, *and*
- $\{|f^{\perp}_{d^{[i]}} - f^{\top}_{d^{[i-1]}}| \mid 2 \leq i \leq \beta\}$ *contains the largest* $\beta - 1$ *frequency differences of* $\{|f^j - f^{j-1}| \mid 2 \leq j \leq n\}$, *where* $(v^{\perp}_{d^{[i]}}, f^{\perp}_{d^{[i]}})$ *and* $(v^{\top}_{d^{[i]}}, f^{\top}_{d^{[i]}})$ *are the first and last element of* $d^{[i]}$, *respectively.* □

Each partition $d^{[i]}$ has four parameters: *floor, ceiling, size, count*; they are defined as $d^{[i]}.floor = v^{\perp}_{d^{[i]}}$, $d^{[i]}.ceiling = v^{\top}_{d^{[i]}}$, $d^{[i]}.size = \|d^{[i]}\|$, $d^{[i]}.count = \sum_{(v,f) \in d^{[i]}} f$. The floor and ceiling of $d^{[i]}$ are the minimal and maximal values that it contains. Its size is the number of distinct values in $d^{[i]}$ and count is the sum of frequencies for those values. These four parameters constitute the summary of a partition and they are stored by histogram buckets.

Definition 3.9 *Given a fact-set for an n-ary predicate* $p(x_1, \ldots, x_n)$, *let* d_i *be the data distributions of* x_i *for* $1 \leq i \leq n$ *and* $d_i^{[j_i]}$ $(1 \leq j_i \leq \beta_i)$ *be the maxdiff partitions of* d_i. *The* **maxdiff histogram** H_{x_i}, *or* $H_{p.x_i}$ *to make the predicate name explicit, for* x_i *consists of* β_i *buckets,* $H^{j_i}_{x_i}$, *defined as follows.*

- *Histogram bucket* $H_{x_i}^{j_i}$. *It has four parameters* $H_{x_i}^{j_i}.floor$, $H_{x_i}^{j_i}.ceiling$, $H_{x_i}^{j_i}.size$, *and* $H_{x_i}^{j_i}.count$ *whose values are the same as those of* $d_i^{[j_i]}$. *These four parameters constitute the bucket as* $(H_{x_i}^{j_i}.floor, H_{x_i}^{j_i}.ceiling, H_{x_i}^{j_i}.size, H_{x_i}^{j_i}.count)$.

- *Value frequency approximation of* $H_{x_i}^{j_i}$. *The set of values contained in* $H_{x_i}^{j_i}$ *is defined as* $vals(H_{x_i}^{j_i}) = \{v \mid H_{x_i}^{j_i}.floor \leq v \leq H_{x_i}^{j_i}.ceiling\}$. *The frequency of* $vals(H_{x_i}^{j_i})$ *is approximated by their average, denoted* $avgf(H_{x_i}^{j_i}) = \frac{H_{x_i}^{j_i}.count}{H_{x_i}^{j_i}.size}$. □

In Definition 3.9, $H_{x_i}^{j_i}$ summarizes $d_i^{[j_i]}$ and H_{x_i} summarizes d_i. Note that $vals(H_{x_i}^{j_i})$ is the set of all possible values between its floor and ceiling, which is usually different from the set of values that *actually* appear in its fact set. The set of actual argument values corresponding to $d_i^{[j_i]}$, $\{v \mid (v, f) \in d_i^{[j_i]}\}$, are not stored since it is prohibitively expensive to do so [IP95]. Size estimation for the relational operations of select, project, and join based on histograms were described in [BC02]. The following Example 3.15 illustrates the idea for the case of select.

Example 3.15 *Consider the following fact-set for predicate* $p(x_1, x_2)$*:*

```
p(2,2).   p(3,7).   p(3,8).   p(4,4).   p(5,5).   p(5,7).
p(5,8).   p(6,6).   p(7,5).   p(7,6).   p(8,1).   p(8,3).
```

Its data distributions are $d_1 = [(2,1),(3,2),(4,1),(5,3),(6,1),(7,2),(8,2)]$ *and* $d_2 = [(1,1),(2,1),(3,1),(4,1),(5,2),(6,2),(7,2),(8,2)]$. *One maxdiff partition, assuming* $\beta_1 = \beta_2 = 3$, *groups them as* $d_1 = [(2,1),(3,2),(4,1)] \bullet [(5,3)] \bullet [(6,1), (7,2),(8,2)]$ *and* $d_2 = [(1,1)] \bullet [(2,1),(3,1),(4,1)] \bullet [(5,2),(6,2),(7,2),(8,2)]$. *Its histogram buckets for* x_1 *and* x_2 *are* $H_{x_1}^1 = (2,4,3,4)$, $H_{x_1}^2 = (5,5,1,3)$, $H_{x_1}^3 = (6,8,3,5)$, $H_{x_2}^1 = (1,1,1,1)$, $H_{x_2}^2 = (2,4,3,3)$, *and* $H_{x_2}^3 = (5,8,4,8)$.

For the selection defined by $q(x_1, x_2)$:- $p(5, x_2)$., *only values in* $vals(H_{x_1}^2) = \{5\}$ *can produce results. Since* $avgf(H_{x_1}^2) = \frac{H_{x_1}^2.count}{H_{x_1}^2.size} = \frac{3}{1} = 3$ *and the selection covers* $\{5\}$, *one can estimate the result size as the product of average frequency and the number of values covered by the selection, i.e.,* $size(q) = avgf(H_{x_1}^2) \times \|\{5\}\| = 3$. □

The histograms for the arguments of q in Example 3.15 can be computed as follows. Since the selection covers the second bucket of the histogram for $p.x_1$, $H_{p.x_1}^2$, we know that the histogram for $q.x_1$ consists of one single bucket which is the same as $H_{p.x_1}^2$, i.e.,

$H^1_{q.x_1} = (5, 5, 1, 3)$. By assuming argument independence [Chr84], we can compute the histogram buckets for $q.x_2$ as $H^1_{q.x_2} = (1, 1, 1, 0.25)$, $H^2_{q.x_2} = (2, 4, 3, 0.75)$, and $H^3_{q.x_2} = (5, 8, 4, 3, 2)$, where $H^i_{q.x_2}.[floor, ceiling, size] = H^i_{p.x_2}.[floor, ceiling, size]$[1] and $H^i_{q.x_2}.count = \frac{size(q)}{size(p)} \times H^i_{p.x_2}.count$.

Although histogram produces good estimate for predicate q, it loses the argument dependency information while computing its histograms, as illustrated by the following Example 3.16.

Example 3.16 *Consider the selection:* $r(x_1, x_2)$:- $q(x_1, x_2), x_2 \leq 4.$, *where* q *is the same predicate in Example 3.15. Since only values in* $vals(H^1_{q.x_2})$ *and* $vals(H^2_{q.x_2})$ *satisfy the selection, we can estimate the size of* r *as* $size(r) = H^1_{q.x_2}.count + H^2_{q.x_2}.count = 1$. *However, there is* no *fact of the form* $p(5, x_2)$ *such that* $x_2 \leq 4$. *This estimation error is caused by the information loss when we compute histograms for* $q(x_1, x_2)$ *by assuming argument independence.* □

3.2 Dependency Matrices

This section presents our data structure *dependency matrices*, which can be viewed as an extension of histograms, to store predicate statistics. Dependency matrices successfully preserve argument dependency and thus take care of the problems (as demonstrated in Example 3.16) caused by argument independence assumption. We also describe intervals, interval sequences, and several dependency matrix operations, which form an important part of our size estimation algorithms.

3.2.1 Definition of Dependency Matrices

Example 3.17 *Consider the fact-set of predicate* $p(x_1, x_2)$ *in Example 3.15. Those facts can be represented as a 2-dimension* fact-matrix F *as shown in Figure 3(A) where* $F(x_1, x_2) = 1$ *if and only if* $p(x_1, x_2)$ *is true, i.e.,* $p(x_1, x_2) \in factset(p)$. *If we segment the rows of* F *according to the partitioning of the distribution* d_1 *of Example 3.15, i.e.,* $|2, 3, 4|5|6, 7, 8|$, *and the columns according to the partitioning of*

[1] Given two entities e_1 and e_2 and a sequence of parameters $par_1, ..., par_n$, we will use $e_1.[par_1, ..., par_n] = e_2.[par_1, ..., par_n]$ to represent $e_1.par_1 = e_2.par_1, ..., e_1.par_n = e_2.par_n$, in short.

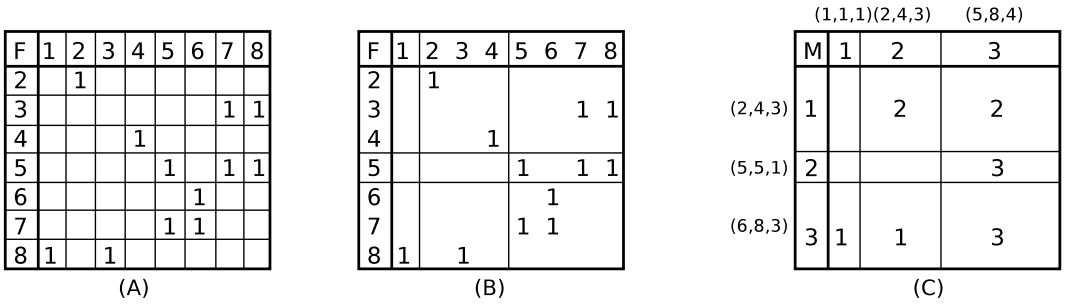


Figure 3: Dependency Matrix of Example 3.17

d_2 of Example 3.15, i.e., $|1|2,3,4|5,6,7,8|$, we obtain a partition of the matrix into rectangular regions as shown in Figure 3(B). The dependency matrix *shown in Figure 3(C) is a* summary *of Figure 3(B): each region of Figure 3(B) is reduced to a single square in Figure 3(C) and the number in the square represents the number of 1's in the corresponding region of Figure 3(B).*[2] *In addition, three of the four parameters of each distribution segment are stored in Figure 3(C) in the form of $(floor, ceiling, size)$. For instance, the floor, ceiling, and size of the third distribution segment of d_1 are $6, 8$, and 3 respectively. Therefore, the bottom coordinate on the vertical axis is annotated with $(6, 8, 3)$.* □

Definition 3.10 *Let $p(x_1, \ldots, x_n)$ be an n-ary predicate and let d_i $(1 \leq i \leq n)$ be the distribution for x_i. Suppose each d_i is partitioned into β_i segments $d_i^{[j_i]}$, $1 \leq j_i \leq \beta_i$. The* **dependency matrix** *for p, denoted $\boldsymbol{M}\langle p \rangle$, is a matrix whose $(j_1, \ldots, j_n)$-th element, $\boldsymbol{M}\langle p \rangle(j_1, \ldots, j_n)$, is*

$$\|\{p(x_1, \ldots, x_n) \in factset(p) \mid \wedge_{1 \leq i \leq n} x_i \in vals(d_i^{[j_i]})\}\|$$

In addition, the j_i-th coordinate on its i-th axis, denoted $\boldsymbol{M}\langle p \rangle_i^{j_i}$, is associated with three parameters: floor, ceiling, and size whose values are the same as the corresponding values associated with the distribution segment $d_i^{[j_i]}$. □

We often use $\mathbf{M}\langle p \rangle_i$ to denote the i-th axis of a matrix $\mathbf{M}\langle p \rangle$ and $\mathbf{M}\langle p \rangle_i^{j_i}$ to denote the j_i-th coordinate on $\mathbf{M}\langle p \rangle_i$, as in Definition 3.10. For instance, let $\mathbf{M}\langle p \rangle$ be the 2-dimension matrix in Figure 3(C), then $\mathbf{M}\langle p \rangle_1$ denotes the first axis of $\mathbf{M}\langle p \rangle$, i.e., the

[2] *0's are represented as blank spaces since dependency matrices are stored as* sparse *matrices.*

vertical axis, and $\mathbf{M}\langle p\rangle_1^2$ denotes the second coordinate on $\mathbf{M}\langle p\rangle_1$. Please note that $\mathbf{M}\langle p\rangle_i^{j_i}$ does not have a *count* parameter, since it is already stored as dependency matrix values. A 1-dimension dependency matrix $\mathbf{M}\langle p\rangle$ for $p(x)$ is actually a histogram where the *count* value of each histogram bucket is stored in the dependency matrix, i.e. $H_x^j.count = \mathbf{M}\langle p\rangle(j)$.

Consider a dependency matrix $\mathbf{M}\langle p\rangle$ for an n-ary predicate $p(x_1, ..., x_n)$. $\mathbf{M}\langle p\rangle(j_1, \ldots, j_n)$ *summarizes* this set of argument values: $vals(d_1^{[j_1]}) \times ... \times vals(d_n^{[j_n]})$ whose cardinality is denoted by $\mathbf{M}\langle p*\rangle(j_1, \ldots, j_n) = \Pi_{1\leq i\leq n}\mathbf{M}\langle p\rangle_i^{j_i}.size$. The dependency matrix value stored in $\mathbf{M}\langle p\rangle(j_1, \ldots, j_n)$ is the number of facts of the form $p(x_1, \ldots, x_n)$ such that $(x_1, \ldots, x_n)$ is summarized by this matrix element. Therefore, we always have $\mathbf{M}\langle p\rangle(j_1, \ldots, j_n) \leq \mathbf{M}\langle p*\rangle(j_1, \ldots, j_n)$.

Example 3.18 *Let $\boldsymbol{M}\langle p\rangle$ be the dependency matrix in Figure 3(C) of Example 3.17. $\boldsymbol{M}\langle p\rangle(1, 2)$ summarizes $\{2, 3, 4\} \times \{2, 3, 4\} = \{(2, 2), (2, 3), (2, 4), (3, 2), (3, 3), (3, 4), (4, 2), (4, 3), (4, 4)\}$, from which only $p(2, 2)$ and $p(4, 4)$ are in the fact-set. Thus, $\boldsymbol{M}\langle p\rangle(1, 2) = 2 \leq \boldsymbol{M}\langle p*\rangle(1, 2) = 9$.* □

Given a $\beta_1 \times \ldots \times \beta_n$ dependency matrix $\mathbf{M}\langle p\rangle$, the size estimate of p, $size(p)$ or $size(\mathbf{M}\langle p\rangle)$, can be computed as the sum of all dependency matrix elements, i.e., $size(p) = \sum_{i_1,\ldots,i_n} \mathbf{M}\langle p\rangle(i_1, \ldots, i_n)$.[3] Therefore, one could always estimate the size of a predicate by computing its dependency matrix.

3.2.2 Intervals and Interval Sequences

An *interval* l is formed by a pair of integers as $(low, high)$, $low \leq high$. The set of integers contained in such an interval is defined as $vals(l) = \{i \mid low \leq i \leq high\}$. We often use $l.low$ and $l.high$ to denote the interval's lower and upper bounds. Given two intervals l_1 and l_2, we say that they *overlap* if $vals(l_1) \cap vals(l_2) \neq \emptyset$, and they are *disjoint* otherwise. We also say that l_1 is *contained* in l_2 if $vals(l_1) \subseteq vals(l_2)$, $l_1 < l_2$ if $l_1.high < l_2.low$, and $l_1 = l_2$ if $vals(l_1) = vals(l_2)$.

An *interval sequence* is a sequence, $s = [s^1, \ldots, s^n]$, of intervals, and the set of values associated with s is $vals(s) = \cup_{1\leq i\leq n} vals(s^i)$. It is *sorted* if $s^i < s^j$ for $1 \leq i < j \leq n$. Two interval sequences $s_1 = [s_1^1, \ldots, s_1^n]$ and $s_2 = [s_2^1, \ldots, s_2^n]$ are

[3] Note that if p is a base predicate then $size(p)$ is its actual size.

equivalent, written as $s_1 = s_2$, if $s_1^i = s_2^i$ for all $1 \leq i \leq n$. Given an n-dimension $\beta_1 \times \ldots \times \beta_n$ dependency matrix $\mathbf{M}\langle p \rangle$ for $p(x_1, ..., x_n)$, we will use $l(\mathbf{M}\langle p \rangle_i^{j_i})$ to denote the interval $(\mathbf{M}\langle p \rangle_i^{j_i}.floor, \mathbf{M}\langle p \rangle_i^{j_i}.ceiling)$ and $s(\mathbf{M}\langle p \rangle_i)$ to denote the sorted interval sequence $[l(\mathbf{M}\langle p \rangle_i^1), \ldots, l(\mathbf{M}\langle p \rangle_i^{\beta_i})]$.

Let $s_1 = [s_1^1, \ldots, s_1^m]$ and $s_2 = [s_2^1, \ldots, s_2^n]$ be sorted interval sequences. Their *intersection* $s_1 \otimes s_2$ and *union* $s_1 \oplus s_2$ are also sorted interval sequences which are computed by Algorithms 3. Intuitively, $s_1 \otimes s_2$ contains the common part of s_1 and s_2 and $s_1 \oplus s_2$ covers the union of s_1 and s_2. The complexity of Algorithm 3 is $O(m + n)$ since each iteration of the `while`-loop takes constant time and there can be at most $m + n$ such iterations since each iteration removes at least one interval from either s_1 or s_2.

1 Initialize $s_1 \otimes s_2 = [\,]$ and $s_1 \oplus s_2 = [\,]$;
2 **while** $s_1 \neq [\,]$ *and* $s_2 \neq [\,]$ **do**
3 Remove the first intervals from s_1 and s_2 as l_1 and l_2, respectively;
4 **if** $l_1 < l_2$ **then**
5 $s_1 \oplus s_2 = (s_1 \oplus s_2) \bullet [l_1]$;
6 $s_2 = [l_2] \bullet s_2$;
7 **else if** $l_1 > l_2$ **then**
8 $s_1 \oplus s_2 = (s_1 \oplus s_2) \bullet [l_2]$;
9 $s_1 = [l_1] \bullet s_1$;
10 **else**
11 $s_1 \otimes s_2 = (s_1 \otimes s_2) \bullet [(\max\{l_1.low, l_2.low\}, \min\{l_1.high, l_2.high\})]$;
12 **if** $l_1.low < l_2.low$ **then**
13 $s_1 \oplus s_2 = (s_1 \oplus s_2) \bullet [(l_1.low, l_2.low - 1)]$;
14 **else if** $l_1.low > l_2.low$ **then**
15 $s_1 \oplus s_2 = (s_1 \oplus s_2) \bullet [(l_2.low, l_1.low - 1)]$;
16 **end**
17 $s_1 \oplus s_2 = (s_1 \oplus s_2) \bullet [(\max\{l_1.low, l_2.low\}, \min\{l_1.high, l_2.high\})]$;
18 **if** $l_1.high < l_2.high$ **then**
19 $s_2 = [(l_1.high+1, l_2.high)] \bullet s_2$;
20 **else if** $l_1.high > l_2.high$ **then**
21 $s_1 = [(l_2.high+1, l_1.high)] \bullet s_1$;
22 **end**
23 **end**
24 **end**
25 $s_1 \oplus s_2 = (s_1 \oplus s_2) \bullet s_1 \bullet s_2$;

Algorithm 3: Intersection and Union of Interval Sequences

Example 3.19 *Let $\boldsymbol{M}\langle p\rangle$ be the dependency matrix of Example 3.17. We know that $s(\boldsymbol{M}\langle p\rangle_1) = [(2,4),(5,5),(6,8)]$ and $s(\boldsymbol{M}\langle p\rangle_2) = [(1,1),(2,4),(5,8)]$. Their intersection and union are as follows: $s(\boldsymbol{M}\langle p\rangle_1) \otimes s(\boldsymbol{M}\langle p\rangle_2) = [(2,4),(5,5),(6,8)]$ and $s(\boldsymbol{M}\langle p\rangle_1) \oplus s(\boldsymbol{M}\langle p\rangle_2) = [(1,1),(2,4),(5,5),(6,8)]$.* □

Theorem 3.8 *Let $s_1 \otimes s_2$ and $s_1 \oplus s_2$ be the intersection and union of two sorted interval sequences s_1 and s_2, respectively. Then,*

i. for each $l \in s_1 \otimes s_2$, there is exactly one $l_1 \in s_1$ (respectively $l_2 \in s_2$) such that l is contained in l_1 (respectively l_2), and

ii. for each $l \in s_1 \oplus s_2$, either l is contained in exactly one interval of s_1 (respectively s_2) or l is disjoint from all intervals of s_1 (respectively s_2). □

Proof: During each `while`-loop, the first internals of s_1 and s_2 are removed as l_1 and l_2 at line 3. Then, the common part of l_1 and l_2 is extracted and added to $l \in s_1 \otimes s_2$ and the parts that are covered by either l_1 or l_2 are added to $l \in s_1 \oplus s_2$. Moreover, both s_1 and s_2 are sorted. Therefore, the theorem holds. □

Theorem 3.8 tells us one important property about interval sequence operations that will be used when defining dependency matrix operations below in Section 3.2.3.

3.2.3 Dependency Matrix Operations

In Section 3.3 we will estimate the sizes of derived predicates using certain operations on dependency matrices. We define these operations in this section.

Definition 3.11 *Consider a $\beta_1 \times \ldots, \times \beta_n$ dependency matrix $\boldsymbol{M}\langle p\rangle$, axis index a ($1 \le a \le n$), and coordinate index c ($1 \le c < \beta_a$) of its a-th axis. The* **merge** *of $\boldsymbol{M}\langle p\rangle_a^c$ and $\boldsymbol{M}\langle p\rangle_a^{c+1}$, denoted merge($\boldsymbol{M}\langle p\rangle, a, c$), produces a $\beta'_1 \times \ldots \times \beta'_n$ dependency matrix $\boldsymbol{M}\langle p\rangle'$ that can be obtained from $\boldsymbol{M}\langle p\rangle$ as follows.*

- *Axis sizes. $\beta'_k = \beta_k$ for $1 \le k \le n$ and $k \ne a$, and $\beta'_a = \beta_a - 1$.*
- *Coordinate parameters. For $1 \le k \le n, k \ne a$ and $1 \le i_k \le \beta'_k$, $\boldsymbol{M}\langle p\rangle'^{i_k}_k.[floor, ceiling, size] = \boldsymbol{M}\langle p\rangle^{i_k}_k.[floor, ceiling, size]$. $\boldsymbol{M}\langle p\rangle'^{i_a}_a.[floor, ceiling, size] =$*

$$\begin{cases} \boldsymbol{M}\langle p\rangle^{i_a}_a.[floor, ceiling, size] & \text{for } 1 \le i_a < c, \\ \boldsymbol{M}\langle p\rangle^{i_a+1}_a.[floor, ceiling, size] & \text{for } c < i_a \le \beta'_a. \end{cases}$$

$\boldsymbol{M}\langle p\rangle'^{c}_{a}.floor = \boldsymbol{M}\langle p\rangle^{c}_{a}.floor$, $\boldsymbol{M}\langle p\rangle'^{c}_{a}.ceiling = \boldsymbol{M}\langle p\rangle^{c+1}_{a}.ceiling$, $\boldsymbol{M}\langle p\rangle'^{c}_{a}.size = \boldsymbol{M}\langle p\rangle^{c}_{a}.size + \boldsymbol{M}\langle p\rangle^{c+1}_{a}.size$.

- *Matrix values. For* $1 \le i_1 \le \beta'_1, ..., 1 \le i_n \le \beta'_n$, $\boldsymbol{M}\langle p\rangle'(i_1, ..., i_a, ..., i_n) =$

$$\begin{cases} \boldsymbol{M}\langle p\rangle(i_1, ..., i_a, ..., i_n) & \text{if } 1 \le i_a < c, \\ \boldsymbol{M}\langle p\rangle(i_1, ..., c, ..., i_n) + \boldsymbol{M}\langle p\rangle(i_1, ..., c{+}1, ..., i_n) & \text{if } i_a = c, \\ \boldsymbol{M}\langle p\rangle(i_1, ..., i_a{+}1, ..., i_n) & \text{if } c < i_a \le \beta'_a. \end{cases}$$

In other words, $\boldsymbol{M}\langle p\rangle'$ *is like* $\boldsymbol{M}\langle p\rangle$ *except that* $\boldsymbol{M}\langle p\rangle'^{c}_{a}$ *is obtained by element-wise additions of* $\boldsymbol{M}\langle p\rangle^{c}_{a}$ *and* $\boldsymbol{M}\langle p\rangle^{c+1}_{a}$ *and then deleting* $\boldsymbol{M}\langle p\rangle^{c+1}_{a}$. □

Merge of coordinates is used to reduce storage requirements of dependency matrices. The *best merge* of $\mathbf{M}\langle p\rangle$ is the merge $merge(\mathbf{M}\langle p\rangle, a, c)$ such that $|avgf(\mathbf{M}\langle p\rangle^{c}_{a}) - avgf(\mathbf{M}\langle p\rangle^{c+1}_{a})|$ is minimum among all $1 \le a \le n$ and $1 \le c \le \beta_a$. Best merges try to minimize the information loss caused by reducing dependency matrix sizes.

Example 3.20 *Consider* $\boldsymbol{M}\langle p\rangle$ *of Example 3.17, which is copied in Figure 4(A). Its best merge is* $merge(\boldsymbol{M}\langle p\rangle, 2, 1)$ *since* $|avgf(\boldsymbol{M}\langle p\rangle^{1}_{2}) - avgf(\boldsymbol{M}\langle p\rangle^{2}_{2})| = 0$ *is minimal among all average frequency differences. The resulting dependency matrix is given in Figure 4(B).* □

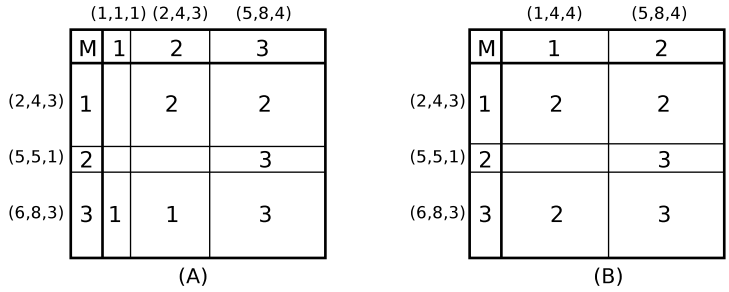


Figure 4: The Best Merge of Dependency Matrix of Example 3.20

Definition 3.12 *Consider an n-dimension* $\beta_1 \times ... \times \beta_n$ *dependency matrix* $\boldsymbol{M}\langle p\rangle$, *an axis index a and a sorted interval sequence s. Assume that for each* $l \in s$, *either*

l is contained in some interval of $s(\boldsymbol{M}\langle p\rangle_a)$ *or l is disjoint from all intervals of* $s(\boldsymbol{M}\langle p\rangle_a)$.[4] *The* **alignment** *of the a-th axis of* $\boldsymbol{M}\langle p\rangle$ *with s produces a* $\beta'_1 \times \ldots \times \beta'_n$ *dependency matrix* $\boldsymbol{M}\langle p\rangle' = align(\boldsymbol{M}\langle p\rangle, a, s)$*, defined below.*

- *Axis sizes.* $\beta'_k = \beta_k$ *for* $1 \leq k \leq n$ *and* $k \neq a$*, and* $\beta'_a = \|s\|$.
- *Coordinate parameters. For* $1 \leq k \leq n, k \neq a$ *and* $1 \leq i_k \leq \beta'_k$, $\boldsymbol{M}\langle p\rangle'^{i_k}_k.[floor, ceiling, size] = \boldsymbol{M}\langle p\rangle^{i_k}_k.[floor, ceiling, size]$. *For* $i = 1, ..., \|s\|$,
  - $\boldsymbol{M}\langle p\rangle'^{i}_a.floor = s^i.low$ *and* $\boldsymbol{M}\langle p\rangle'^{i}_a.ceiling = s^i.high$;
  - *If* s^i *is contained in some* $l(\boldsymbol{M}\langle p\rangle^c_a)$, *then* $\boldsymbol{M}\langle p\rangle'^{i}_a.size = \boldsymbol{M}\langle p\rangle^c_a.size \times \frac{vals(s^i)}{vals(\boldsymbol{M}\langle p\rangle^c_a)}$; *otherwise,* $\boldsymbol{M}\langle p\rangle'^{i}_a.size = 0$.
- *Matrix values. For* $1 \leq i_1 \leq \beta'_1, ..., 1 \leq i_n \leq \beta'_n$, *if* s^{i_a} *is contained in some* $l(\boldsymbol{M}\langle p\rangle^c_a)$, *then* $\boldsymbol{M}\langle p\rangle'(i_1, ..., i_n) = \boldsymbol{M}\langle p\rangle(i_1, ..., i_{a-1}, c, i_{a+1}, ..., i_n) \times \frac{vals(s^{i_a})}{vals(\boldsymbol{M}\langle p\rangle^c_a)}$; *otherwise,* $\boldsymbol{M}\langle p\rangle'(i_1, ..., i_n) = 0$; □

The alignment, $\mathbf{M}\langle p\rangle' = align(\mathbf{M}\langle p\rangle, a, s)$, creates $\mathbf{M}\langle p\rangle'$ from $\mathbf{M}\langle p\rangle$ by *splitting* $\mathbf{M}\langle p\rangle_a$ such that $s(\mathbf{M}\langle p\rangle'_a) = s$. For each s^i, if it is contained in some $\mathbf{M}\langle p\rangle^c_a$, then that portion of $\mathbf{M}\langle p\rangle^c_a$ that overlaps with s^i is extracted out as $\mathbf{M}\langle p\rangle'^{i}_a$; otherwise, $\mathbf{M}\langle p\rangle'^{i}_a$ contains only *0*'s.

Example 3.21 *Let* $\boldsymbol{M}\langle p\rangle$ *be the 2-dimension dependency matrix of Example 3.17, repeated in Figure 5(A), and* $s = s(\boldsymbol{M}\langle p\rangle_1) \oplus s(\boldsymbol{M}\langle p\rangle_2) = [(1,1),(2,4),(5,5),(6,8)]$. $\boldsymbol{M}\langle p\rangle' = align(\boldsymbol{M}\langle p\rangle, 1, s)$ *is shown in Figure 5(B), where* $\boldsymbol{M}\langle p\rangle'^{1}_1$ *contains only 0's since* s^1 *is disjoint with any interval of* $s(\boldsymbol{M}\langle p\rangle_1)$. $\boldsymbol{M}\langle p\rangle'' = align(\boldsymbol{M}\langle p\rangle', 2, s)$ *is shown in Figure 5(C), where* $\boldsymbol{M}\langle p\rangle''^{3}_2$ *and* $\boldsymbol{M}\langle p\rangle''^{4}_2$ *are obtained by exacting corresponding portions from* $\boldsymbol{M}\langle p\rangle'^{3}_2$. *Their parameters are in the form of* $(floor, ceiling, size)$. □

Consider a $\beta^p_1 \times ... \times \beta^p_m$ dependency matrix for predicate $p(x_1, ..., x_m)$, a $\beta^q_1 \times ... \times \beta^q_n$ dependency matrix for predicate $q(x_1, ..., x_n)$, and two axis indexes $1 \leq a_p \leq m$ and $1 \leq a_q \leq n$. We say that the a_p-th axis of $\mathbf{M}\langle p\rangle$ and the a_q-th axis of $\mathbf{M}\langle q\rangle$ are *aligned* if $s(\mathbf{M}\langle p\rangle_{a_p}) = s(\mathbf{M}\langle q\rangle_{a_q})$. For any pair of dependency matrices

[4] *For instance, it is the case if* $s = s(\boldsymbol{M}\langle p\rangle_a)$ *operator* s' *where operator can be* $\otimes$ *or* $\oplus$ *and* s' *is another sorted interval sequence.*

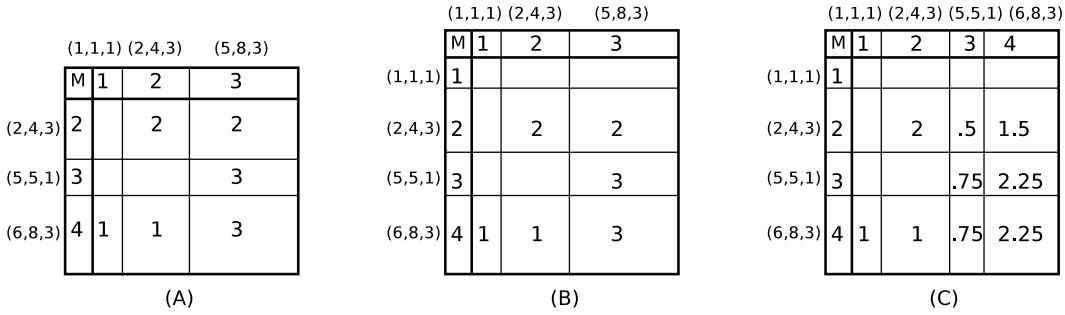


Figure 5: Dependency Matrix Alignment of Example 3.21

$\mathbf{M}\langle p\rangle$ and $\mathbf{M}\langle q\rangle$ and a pair of axis indexes a_p and a_q, we can always make $\mathbf{M}\langle p\rangle_{a_p}$ and $\mathbf{M}\langle q\rangle_{a_q}$ aligned by performing operations $\mathit{align}(\mathbf{M}\langle p\rangle, a_p, s)$ and $\mathit{align}(\mathbf{M}\langle q\rangle, a_q, s)$ where $s = s(\mathbf{M}\langle p\rangle_{a_p}) \oplus s(\mathbf{M}\langle q\rangle_{a_q})$.

Let $\mathbf{M}\langle p\rangle$ and $\mathbf{M}\langle q\rangle$ both be $\beta_1 \times \ldots \times \beta_n$ matrices. It follows directly from the definitions that $\mathbf{M}\langle p\rangle(i_1, ..., i_n)$ and $\mathbf{M}\langle q\rangle(i_1, ..., i_n)$ summarize the same values for all $i_1, ..., i_n$ if and only if $\mathbf{M}\langle p\rangle_a$ and $\mathbf{M}\langle q\rangle_a$ are aligned for $a = 1, ..., n$. In such case, we say that $\mathbf{M}\langle p\rangle$ and $\mathbf{M}\langle q\rangle$ are *perfectly aligned*. Since, as noted above, any two axises of two matrices can always be aligned, it follows that any pair matrices can be refined so that they are perfectly aligned.

3.3 Statistics for Derived Predicates

The dependency matrices for base predicates are obtained directly following the constructive Definitions 3.8 and 3.10 as illustrated by Examples 3.17. Dependency matrices of derived predicates are computed by abstractly evaluating their defining rules, where rule bodies are replaced with algebraic expressions over the dependency matrices for the body predicates. Recursive rules are evaluated iteratively until *approximate fixed points* (to be defined later) are reached. This section first describes the abstraction interpretation for each individual rule type and then presents algorithms to compute dependency matrices for all query related predicates. Complexity analysis for different operations of SDP is also included.

3.3.1 Dependency Matrix for Selection

Let $\mathbf{M}\langle p\rangle$ be the n-dimension dependency matrix for $p(x_1, ..., x_n)$. We consider the following two types of selections: *selection with constant equalities* (`s1`) and *selection with range restrictions* (`s2`)

```
r(X1,...,Xn) :- p(X1,...,Xn), Xa == val.          (s1)
r(X1,...,Xn) :- p(X1,...,Xn), Xa >= l, Xa =< h.   (s2)
```

where $1 \leq a \leq n$. Observing that `s1` is a special case of `s2` when $l = h = val$, we can focus on computing $\mathbf{M}\langle r\rangle$ for `s2`. Let $s = s(\mathbf{M}\langle p\rangle_a) \otimes [(l, h)]$, and we know that the selection is to restrict x_a-values to $vals(s)$. Therefore, the dependency matrix $\mathbf{M}\langle r\rangle$ can be computed by $\mathbf{M}\langle r\rangle = align(\mathbf{M}\langle p\rangle, a, s)$ as given in Definition 3.12.

Example 3.22 *Consider the same fact-set for predicate $p(x_1, x_2)$ as in Example 3.15 and these two selections:*

```
q(X1,X2) :- p(X1,X2), X1 == 5.
r(X1,X2) :- q(X1,X2), X2 =< 4.
```

$\boldsymbol{M}\langle p\rangle$ is repeated in Figure 6(A) for easy reference. Now, we compute $\boldsymbol{M}\langle q\rangle$ and $\boldsymbol{M}\langle r\rangle$. Let $s_1 = s(\boldsymbol{M}\langle p\rangle_1) \otimes [(5, 5)] = [(2, 4), (5, 5), (6, 8)] \otimes [(5, 5)] = [(5, 5)]$, then the dependency matrix for q can be computed by $\boldsymbol{M}\langle q\rangle = align(\boldsymbol{M}\langle p\rangle, 1, s_1)$, as given in Figure 6(B). Similarly, let $s_2 = s(\boldsymbol{M}\langle q\rangle_2) \otimes [(-\infty, 4)] = [(1, 1), (2, 4), (5, 8)] \otimes [(-\infty, 4)] = [(1, 1), (2, 4)]$, then $\boldsymbol{M}\langle r\rangle = align(\boldsymbol{M}\langle q\rangle, 2, s_2)$, shown in Figure 6(C).

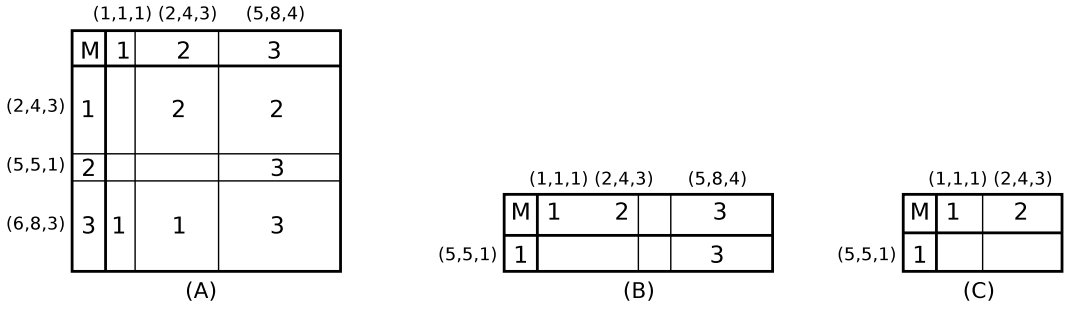


Figure 6: Dependency Matrix for Selection

The size estimate of q and r are $size(q) = 3$ and $size(r) = 0$, respectively. Please note that here we have $size(r) = 0$, which is different from its size estimate given in

Example 3.16 (it was then estimated as 1). It is easy to see that dependency matrix produces more accurate estimates by preserving argument dependencies. □

3.3.2 Dependency Matrix for Union

Consider the predicate p defined as the union of r and s by the following rules:

```
p(X1,...,Xn) :- r(X1,...,Xn).
p(X1,...,Xn) :- s(X1,...,Xn).
```

Before performing the union, we assume that $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ are both $\beta_1 \times ... \times \beta_n$ dependency matrices and they are perfectly aligned. This assumption guarantees that any pair of matrix elements, $\mathbf{M}\langle r\rangle(i_1, ..., i_n)$ and $\mathbf{M}\langle s\rangle(i_1, ..., i_n)$, summarize the same set of possible $(x_1, ..., x_n)$-values so that union can be performed on them.

The dependency matrix for the union is also a n-dimension $\beta_1 \times ... \times \beta_n$ dependency matrix that is computed by $union(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ given in Algorithm 4. There, lines 1 – 4 compute the floor, ceiling, and size parameters in a natural way, where line 3 is based on the *containment assumption* [SAC+79]. This assumption states that each individual value in a smaller set matches some value from the larger set. It is a common assumption in the literature on size estimation. Then, line 6 computes the values of $\mathbf{M}\langle p\rangle$ by integrating $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ in an element-wise manner.

```
1 for all 1 ≤ i ≤ n and 1 ≤ j_i ≤ β_i do
2     M⟨p⟩_i^{j_i}.[floor, ceiling] = M⟨r⟩_i^{j_i}.[floor, ceiling];
3     M⟨p⟩_i^{j_i}.size = max{M⟨r⟩_i^{j_i}.size, M⟨s⟩_i^{j_i}.size};
4 end
5 for 1 ≤ i_1 ≤ β_1, ..., 1 ≤ i_n ≤ β_n do
6     M⟨p⟩(i_1, ..., i_n) = integrate({M⟨r⟩(i_1, ..., i_n), M⟨s⟩(i_1, ..., i_n)}, M⟨p*⟩(i_1, ..., i_n))
      /* to be explained later */
7 end
8 return M⟨p⟩;
```

Algorithm 4: $union(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$

SDP investigates three alternative definitions of $integrate(\{\mathbf{M}\langle r\rangle(i_1, ..., i_n), \mathbf{M}\langle s\rangle(i_1, ..., i_n)\}, \mathbf{M}\langle p*\rangle(i_1, ..., i_n))$ (line 6 of Algorithm 4): *minimal, maximal*, and *normalized integrations*, which yield the minimal, maximal and expected number of

results respectively. Minimal integration is based on the containment assumption and defines it as $\max\{\mathbf{M}\langle r\rangle(i_1, ..., i_n), \mathbf{M}\langle s\rangle(i_1, ..., i_n)\}$. Maximal integration assumes that the fact-sets of r and s do not intersect and takes the sum of corresponding dependency matrix values: $\min\{\mathbf{M}\langle r\rangle(i_1, ..., i_n) + \mathbf{M}\langle s\rangle(i_1, ..., i_n), \mathbf{M}\langle p*\rangle(i_1, ..., i_n)\}$ (Recall that $\mathbf{M}\langle p*\rangle(i_1, ..., i_n)$ is the upper limit for $\mathbf{M}\langle p\rangle(i_1, ..., i_n)$). The most involved definition of *integrate* is normalized integration. Consider a fact $s(x_1, ..., x_n)$ such that $(x_1, ..., x_n)$ is summarized by $\mathbf{M}\langle s\rangle(i_1, ..., i_n)$, there is a probability of $\frac{\mathbf{M}\langle r\rangle(i_1,...,i_n)}{\mathbf{M}\langle p*\rangle(i_1,...,i_n)}$ that $r(i_1, ..., i_n)$ is also a fact, meaning that there is a duplicate in the union results. Therefore, after removing expected number of duplicates, the number of union results can be estimated as $\mathbf{M}\langle r\rangle(i_1, ..., i_n) + (1 - \frac{\mathbf{M}\langle r\rangle(i_1,...,i_n)}{\mathbf{M}\langle p*\rangle(i_1,...,i_n)}) \times \mathbf{M}\langle s\rangle(i_1, ..., i_n)$.

These three definitions of *integrate* differ in their treatments of duplicates. Minimal integration removes most duplicates, maximal integration does not remove duplicates, while normalized integration normalizes results by removing expected number of duplicates. They can be extended to handle a set of values, V, as $integrate(V, \mathbf{M}\langle p*\rangle(i_1, ..., i_n))$ defined below.

- minimal: $\max\{v \in V\}$;
- maximal: $\min\{\sum_{v\in V} v, \mathbf{M}\langle p*\rangle(i_1, ..., i_n)\}$;
- normalized: $norm\_sum(V, \mathbf{M}\langle p*\rangle(i_1, ..., i_n))$ as defined by Algorithm 5.

1 Let V be a set of numbers and b be an upper bound;
2 Initialize $norm\_sum(V, b) = 0$;
3 **while** $V \neq \emptyset$ **do**
4 Remove v from V;
5 $norm\_sum(V, b) = norm\_sum(V, b) + v - \frac{norm\_sum(V,b)\times v}{b}$;
6 **end**
7 **return** $norm\_sum(V, b)$

Algorithm 5: Normalized Integration

3.3.3 Dependency Matrix for Intersection

Consider the following intersection:

```
p(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).
```

Similar to the case of union, we assume that $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ are both $\beta_1 \times ... \times \beta_n$ dependency matrices and they are perfectly aligned.

Algorithm 6 computes the n-dimension $\beta_1 \times ... \times \beta_n$ dependency matrix for the above intersection as $intersect(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$. The outline of the algorithm is the same as in the case of union operation, but the details are simpler. Lines 3 and 6 are based on the containment assumption [SAC+79].

1 **for** *all* $1 \leq i \leq n$ *and* $1 \leq j_i \leq \beta_i$ **do**
2 $\quad \mathbf{M}\langle p\rangle_i^{j_i}.[floor, ceiling] = \mathbf{M}\langle r\rangle_i^{j_i}.[floor, ceiling]$;
3 $\quad \mathbf{M}\langle p\rangle_i^{j_i}.size = \min\{\mathbf{M}\langle r\rangle_i^{j_i}.size, \mathbf{M}\langle r\rangle_i^{j_i}.size\}$;
4 **end**
5 **for** $1 \leq i_1 \leq \beta_1, ..., \ 1 \leq i_n \leq \beta_n$ **do**
6 $\quad \mathbf{M}\langle p\rangle(i_1, ..., i_n) = \min\{\mathbf{M}\langle r\rangle(i_1, ..., i_n), \mathbf{M}\langle s\rangle(i_1, ..., i_n)\}$;
7 **end**
8 **return** $\boldsymbol{M}\langle p\rangle$;

Algorithm 6: $intersect(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$

3.3.4 Dependency Matrix for Projection

Consider a projection on predicate $r(x_1, ..., x_n)$ that projects out a subset of arguments $out\_args = \{x_{out_1}, ..., x_{out_k}\}$ and leaves $in\_args = \{x_{in_1}, ..., x_{in_m}\}$ where $out\_args \cup in\_args = \{x_1, ..., x_n\}$. Such a projection can be defined using a rule of the form

$\mathtt{p}(\mathtt{X}_{\mathtt{in}_1}, ..., \mathtt{X}_{\mathtt{in}_m})$:- $\mathtt{r}(\mathtt{X}_1, ..., \mathtt{X}_n)$.

where $1 \leq in_i \leq n$ and $1 \leq i \leq m$. Let $\mathbf{M}\langle r\rangle$ be a n-dimension $\beta_1 \times ... \times \beta_n$ dependency matrix. The m-dimension $\beta_{in_1} \times ... \times \beta_{in_m}$ dependency matrix, $\mathbf{M}\langle p\rangle = project(\mathbf{M}\langle r\rangle, \{x_{in_1}, ..., x_{in_m}\})$, is computed via Algorithm 7, below.

3.3.5 Dependency Matrix for Join

The most complicated part of any size estimation algorithm in query optimizers is finding accurate estimates for the result of a join. Consider the following join:

```
p(Z1,...,Zk,Xk+1,...,Xm,Yk+1,...,Yn) :- r(Z1,...,Zk,Xk+1,...,Xm),
                                        s(Z1,...,Zk,Yk+1,...,Yn).
```

1 **for** $1 \leq i \leq m$ *and* $1 \leq j_i \leq \beta_{in_i}$ **do**
2 $\quad \mathbf{M}\langle p\rangle_i^{j_i}.[floor, ceiling, size] = \mathbf{M}\langle r\rangle_{in_i}^{j_i}.[floor, ceiling, size]$;
3 **end**
4 **for** $1 \leq i_1 \leq \beta_{in_1}, ..., 1 \leq i_m \leq \beta_{in_m}$ **do**
5 $\quad V = \{\mathbf{M}\langle r\rangle(j_1, ..., j_n) \mid \wedge_{1 \leq k \leq m} i_k = j_{in_k}\}$;
6 $\quad \mathbf{M}\langle p\rangle(i_1, ..., i_m) = integrate(V, \mathbf{M}\langle p*\rangle(i_1, ..., i_m))$;
7 **end**
8 **return** $\boldsymbol{M}\langle p\rangle$;

Algorithm 7: $project(\mathbf{M}\langle r\rangle, \{x_{in_1}, ..., x_{in_m}\})$

where r and s join on k arguments, $z_1, ..., z_k$, which are listed first, for simplicity. Assume $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ are m-dimension $\beta_1^r \times ... \times \beta_m^r$ and n-dimension $\beta_1^s \times ... \times \beta_n^s$ dependency matrices respectively, and the first k axes of $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ are aligned. The $(m+n-k)$-dimension dependency matrix for p can be computed as $join(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ by Algorithm 8. The abstract evaluation of the join is inspired by the *sort-merge-join* algorithm (e.g., [KBL06]).

The overall outline of the algorithm is the same as before. Lines 1 – 6 set the number of coordinates for each axis and lines 7 – 16 compute parameter values of each coordinate. However, the computation details of dependency matrix values are somewhat involved. They are inspired by [BC02] and described below.

Lines 17 – 18 compute $\mathbf{M}\langle r\rangle'$ and $\mathbf{M}\langle s\rangle'$ as projections of $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ which keep only join arguments. $\mathbf{M}\langle r\rangle'(i_1, ..., i_k)$ is the estimated number of $(z_1, ..., z_k)$-values such that $z_j \in vals(l(\mathbf{M}\langle r\rangle_j^{i_j}))$ for $1 \leq j \leq k$, and $\mathbf{M}\langle s\rangle'(i_1, ..., i_k)$ is the estimated number of $(z_1, ..., z_k)$-values such that $z_j \in vals(l(\mathbf{M}\langle s\rangle_j^{i_j}))$ for $1 \leq j \leq k$. Now, consider one fixed $(z_1, ..., z_k)$-value. We know that there are, on average, $\frac{\mathbf{M}\langle r\rangle(i_1,...,i_m)}{\mathbf{M}\langle r\rangle'(i_1,...,i_k)}$ facts of the form $r(z_1, ..., z_k, x_{k+1}, ..., x_m)$ that are summarized by $\mathbf{M}\langle r\rangle(i_1, ..., i_m)$, and $\frac{\mathbf{M}\langle s\rangle(i_1,...,i_k,i_{m+1},...,i_{m+n-k})}{\mathbf{M}\langle s\rangle'(i_1,...,i_k)}$ facts of the form $s(z_1, ..., z_k, y_{k+1}, ..., y_n)$ that are summarized by $\mathbf{M}\langle s\rangle(i_1, ..., i_k, i_{m+1}, ..., i_{m+n-k})$. Thus, the number of resulting $p(z_1, ..., z_k, x_{k+1}, ..., x_m, y_{k+1}, ..., y_n)$ from the join on this fixed $(z_1, ..., z_k)$-value can be estimated as $\frac{\mathbf{M}\langle r\rangle(i_1,...,i_m)}{\mathbf{M}\langle r\rangle'(i_1,...,i_k)} \times \frac{\mathbf{M}\langle s\rangle(i_1,...,i_k,i_{m+1},...,i_{m+n-k})}{\mathbf{M}\langle s\rangle'(i_1,...,i_k)}$. The containment assumption [SAC+79] says that the number of such fixed $(z_1, ..., z_k)$-values is $\min\{\mathbf{M}\langle r\rangle'(i_1, ..., i_k), \mathbf{M}\langle s\rangle'(i_1, ..., i_k)\}$. Therefore, we have the formula at line 20.

Example 3.23 *Assume that dependency matrices* $\boldsymbol{M}\langle r\rangle$ *and* $\boldsymbol{M}\langle s\rangle$ *are the same dependency matrix as* $\boldsymbol{M}\langle p\rangle''$ *in Example 3.21. Consider the following join rule*

for $1 \leq i \leq m$ **do**
    $\beta_i^p = \beta_i^r$;
end
for $m < i \leq m+n-k$ **do**
    $\beta_i^p = \beta_{i-m+k}^s$;
end
for $1 \leq i \leq k$ *and* $1 \leq j_i \leq \beta_i^p$ **do**
    $\mathbf{M}\langle p\rangle_i^{j_i}.[floor, ceiling] = \mathbf{M}\langle r\rangle_i^{j_i}.[floor, ceiling]$;
    $\mathbf{M}\langle p\rangle_i^{j_i}.size = \min\{\mathbf{M}\langle r\rangle_i^{j_i}.size, \mathbf{M}\langle s\rangle_i^{j_i}.size\}$;
end
for $k < i \leq m$ *and* $1 \leq j_i \leq \beta_i^p$ **do**
    $\mathbf{M}\langle p\rangle_i^{j_i}.[floor, ceiling, size] = \mathbf{M}\langle r\rangle_i^{j_i}.[floor, ceiling, size]$;
end
for $m < i \leq m+n-k$ *and* $1 \leq j_i \leq \beta_i^p$ **do**
    $\mathbf{M}\langle p\rangle_i^{j_i}.[floor, ceiling, size] = \mathbf{M}\langle s\rangle_{i-m+k}^{j_i}.[floor, ceiling, size]$;
end
$\mathbf{M}\langle r\rangle' = project(\mathbf{M}\langle r\rangle, \{z_1, ..., z_k\})$;
$\mathbf{M}\langle s\rangle' = project(\mathbf{M}\langle s\rangle, \{z_1, ..., z_k\})$;
for $1 \leq i_1 \leq \beta_1^p, ..., 1 \leq i_{m+n-k} \leq \beta_{m+n-k}^p$ **do**
    $\mathbf{M}\langle p\rangle(i_1, ..., i_{m+n-k}) = \min\{\mathbf{M}\langle r\rangle'(i_1, ..., i_k), \mathbf{M}\langle s\rangle'(i_1, ..., i_k)\} \times \frac{\mathbf{M}\langle r\rangle(i_1,...,i_m)}{\mathbf{M}\langle r\rangle'(i_1,...,i_k)} \times \frac{\mathbf{M}\langle s\rangle(i_1,...,i_k,i_{m+1},...,i_{m+n-k})}{\mathbf{M}\langle s\rangle'(i_1,...,i_k)}$
end
return $\boldsymbol{M}\langle p\rangle$;

Algorithm 8: $join(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$

```
p(Z,X,Y) :- r(Z,X), s(Z,Y).
```

where r and s join on one argument z. Since $\boldsymbol{M}\langle r\rangle$ and $\boldsymbol{M}\langle s\rangle$ are the same, their first axes, which correspond to the join argument z, are already aligned. Their projections that keep the z-argument are computed as $\boldsymbol{M}\langle r\rangle' = project(\boldsymbol{M}\langle r\rangle, \{z\})$ and $\boldsymbol{M}\langle s\rangle' = project(\boldsymbol{M}\langle s\rangle, \{z\})$, and both $\boldsymbol{M}\langle r\rangle'$ and $\boldsymbol{M}\langle s\rangle'$ are the same 1-dimension dependency matrix, i.e., histogram. If "minimal integration" is used as the definition of integrate function when computing these two projections, we will have $\boldsymbol{M}\langle r\rangle'^1_1 = \boldsymbol{M}\langle s\rangle'^1_1 = (1, 1, 1, 0)$, $\boldsymbol{M}\langle r\rangle'^2_1 = \boldsymbol{M}\langle s\rangle'^2_1 = (2, 4, 3, 2)$, $\boldsymbol{M}\langle r\rangle'^3_1 = \boldsymbol{M}\langle s\rangle'^3_1 = (5, 5, 1, 1)$, and $\boldsymbol{M}\langle r\rangle'^4_1 = \boldsymbol{M}\langle s\rangle'^4_1 = (6, 8, 3, 2.25)$.

Finally, dependency matrix values are computed. For instance, $\boldsymbol{M}\langle p\rangle(2, 2, 4) = \min\{\boldsymbol{M}\langle r\rangle'(2), \boldsymbol{M}\langle s\rangle'(2)\} \times \frac{\boldsymbol{M}\langle r\rangle(2,2)}{\boldsymbol{M}\langle r\rangle'(2)} \times \frac{\boldsymbol{M}\langle s\rangle(2,4)}{\boldsymbol{M}\langle s\rangle'(2)} = \min\{2, 2\} \times \frac{2}{2} \times \frac{1.5}{2} = 1.5$. □

3.3.6 Dependency Matrix for Cross Product

Consider the following cross product

```
p(X1,...,Xm,Y1,...,Yn) :- r(X1,...,Xm), s(Yk,...,Yn).
```

where $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ are m-dimension $\beta_1^r \times ... \times \beta_m^r$ and n-dimension $\beta_1^s \times ... \times \beta_n^s$ dependency matrices respectively. The $(m+n)$-dimension $\beta_1^p \times ... \times \beta_{m+n}^p$ dependency matrix for p can be estimated via the operation $product(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ defined in Algorithm 9, which follows the intuitions behind the cross product operation.

for $1 \leq i \leq m$ **do**
    $\beta_i^p = \beta_i^r$;
end
for $1 \leq i \leq n$ **do**
    $\beta_{i+m}^p = \beta_i^s$;
end
for $1 \leq i \leq m$ *and* $1 \leq j_i \leq \beta_i^p$ **do**
    $\mathbf{M}\langle p\rangle_i^{j_i}.[floor, ceiling, size] = \mathbf{M}\langle r\rangle_i^{j_i}.[floor, ceiling, size]$;
end
for $1 < i \leq n$ *and* $1 \leq j_i \leq \beta_i^s$ **do**
    $\mathbf{M}\langle p\rangle_{i+m}^{j_{i+m}}.[floor, ceiling, size] = \mathbf{M}\langle s\rangle_i^{j_i}.[floor, ceiling, size]$;
end
for $1 \leq i_1 \leq \beta_1^p, ..., 1 \leq i_{m+n} \leq \beta_{m+n}^p$ **do**
    $\mathbf{M}\langle p\rangle(i_1, ..., i_{m+n}) = \mathbf{M}\langle r\rangle(i_1, ..., i_m) \times \mathbf{M}\langle s\rangle(i_{m+1}, ..., i_{m+n})$;
end
return $\boldsymbol{M}\langle p\rangle$;

Algorithm 9: $product(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$

3.3.7 Dependency Matrix for Negation

Let $\mathbf{M}\langle r\rangle$ be a n-dimensional $\beta_1 \times ... \times \beta_n$ dependency matrix for r and $\mathbf{M}\langle s\rangle$ be a m-dimensional $\beta_1 \times ... \times \beta_m$ dependency matrix for s. We consider the following negation

```
p(X1,...,Xn) :- r(X1,...,Xn), not s(X1,...,Xm).
```

where $m \leq n$ and the first m arguments are chosen to be common to r and s, for simplicity. We also assume that the first m axes of $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ are aligned.

Algorithm 10 computes the operation $minus(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$, where lines 5 – 7 compute dependency matrix values using function *negate_ratio* whose details are given below.

1 **for** $1 \leq i \leq n$ *and* $1 \leq j_i \leq \beta_i$ **do**
2 $\quad \mathbf{M}\langle p\rangle_i^{j_i}.[floor, ceiling, size] = \mathbf{M}\langle r\rangle_i^{j_i}.[floor, ceiling, size]$;
3 **end**
4 $\mathbf{M}\langle r\rangle' = project(\mathbf{M}\langle r\rangle, \{x_1, ..., x_m\})$;
5 **for** $1 \leq i_1 \leq \beta_1, ..., 1 \leq i_n \leq \beta_n$ **do**
6 $\quad \mathbf{M}\langle p\rangle(i_1, ..., i_n) = negate\_ratio(\mathbf{M}\langle r\rangle', \mathbf{M}\langle s\rangle, \{i_1, ..., i_m\}) \times \mathbf{M}\langle r\rangle(i_1, ..., i_n)$
7 **end**
8 **return** $\boldsymbol{M}\langle p\rangle$;

Algorithm 10: $minus(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$

Given m-dimension dependency matrices $\mathbf{M}\langle r\rangle'$ and $\mathbf{M}\langle s\rangle$ and coordinate indexes $i_1, ..., i_m$, SDP explores three definitions of $negate\_ratio(\mathbf{M}\langle r\rangle', \mathbf{M}\langle s\rangle, \{i_1, ..., i_m\})$: *minimal*, *maximal* and *normalized negations* which compute the minimal, maximal and expected percentages of $(x_1, ..., x_k)$-values that are summarized by $r'(i_1, ..., i_m)$ but not by $s(i_1, ..., i_m)$ respectively. Minimal negation assumes that there is a fact of the form $r'(x_1, ..., x_m)$ for every fact $s(x_1, ..., x_m)$, thus $negate\_ratio(\mathbf{M}\langle r\rangle', \mathbf{M}\langle s\rangle, \{i_1, ..., i_m\}) = \frac{\max\{\mathbf{M}\langle r\rangle'(i_1,...,i_m) - \mathbf{M}\langle s\rangle(i_1,...,i_m), 0\}}{\mathbf{M}\langle r\rangle'(i_1,...,i_m)}$. Opposite to minimal negation, the maximal approach assumes that the intersection of the set of $(x_1, ..., x_m)$-values that are summarized by $\mathbf{M}\langle r\rangle'(i_1, ..., i_m)$ and those values that are summarized by $\mathbf{M}\langle s\rangle(i_1, ..., i_m)$ are minimal, therefore producing maximal possible results. It is given by the formula $1 - \frac{\max\{\mathbf{M}\langle r\rangle'(i_1,...,i_m) + \mathbf{M}\langle s\rangle(i_1,...,i_m) - \mathbf{M}\langle r\rangle'^*(i_1,...,i_n), 0\}}{\mathbf{M}\langle r\rangle'^*(i_1,...,i_m)}$. Similar to the intuition of normalized integration, normalized negation computes the expected percentage as $1 - \frac{\mathbf{M}\langle s\rangle(i_1,...,i_m)}{\mathbf{M}\langle r\rangle'^*(i_1,...,i_m)}$.

3.3.8 Dependency Matrix for Recursive Predicates

The matrix for recursive predicates are computed iteratively until their size estimates reach *approximate* fixed points. For a set of mutually recursive predicates, the iteration stops when their size estimates simultaneously reach approximate fixed points.

Definition 3.13 *Consider a recursive predicate p. According to the definitions in Sections 3.3.1 – 3.3.7, the dependency matrix for p can be defined by the following recurrent equation:* $\boldsymbol{M}\langle p\rangle = expr(\boldsymbol{M}\langle p\rangle, other)$*, where expr is an expression*

in the algebra of estimation operators defined in earlier sections. This equation is recursive in $\boldsymbol{M}\langle p\rangle$ but expr may take other arguments as well. We say that $\boldsymbol{I}^{n+1}$ is an **Δ-approximation** *of $\boldsymbol{M}\langle p\rangle$ if $\frac{|size(\boldsymbol{I}^{n+1})-size(\boldsymbol{I}^{n})|}{size(\boldsymbol{I}^{n})} \leq \Delta$, where $\boldsymbol{I}^0 = \emptyset$ and $\boldsymbol{I}^{i+1} = expr(\boldsymbol{I}^i, other)$. SDP uses Δ-approximations to estimate the size of p.* □

Example 3.24 *Consider the following recursive program, which defines two mutually recursive predicates tcp and tcq:*

```
tcp(X1,X2,X3) :- p(X1,X2,X3).
tcq(X1,X2,X3) :- q(X1,X2,X3).
tcp(X1,X2,X4) :- tcq(X1,X2,X3), p(X2,X3,X4).   %% recp
tcq(X1,X2,X4) :- tcp(X1,X2,X3), q(X2,X3,X4).   %% recq
```

Initially, the dependency matrices $\boldsymbol{M}\langle p\rangle$ and $\boldsymbol{M}\langle q\rangle$ are computed and propagated to $\boldsymbol{M}\langle tcp\rangle$ and $\boldsymbol{M}\langle tcq\rangle$ using the first two rules. Then, the following iterative steps are performed.

1. *Compute $\boldsymbol{M}\langle tcp\rangle$ using the rule* `recp` *as in the case of a join followed by a projection, and then union with current dependency matrix $\boldsymbol{M}\langle tcp\rangle$, i.e., $\boldsymbol{M}\langle tcp\rangle = union(\boldsymbol{M}\langle tcp\rangle, project(join(\boldsymbol{M}\langle tcq\rangle, \boldsymbol{M}\langle p\rangle), \{x_1, x_2, x_4\}))$. In this case, the expression used in Definition 3.13 is $expr(\boldsymbol{M}\langle tcp\rangle, \boldsymbol{M}\langle tcq\rangle, \boldsymbol{M}\langle p\rangle) = union(\boldsymbol{M}\langle tcp\rangle, project(join(\boldsymbol{M}\langle tcq\rangle, \boldsymbol{M}\langle p\rangle), \{x_1, x_2, x_4\}))$.*

2. *Similarly, $\boldsymbol{M}\langle tcq\rangle = union(\boldsymbol{M}\langle tcq\rangle, project(join(\boldsymbol{M}\langle tcp\rangle, \boldsymbol{M}\langle q\rangle), \{x_1, x_2, x_4\}))$ and the expression used in Definition 3.13 is $expr(\boldsymbol{M}\langle tcq\rangle, \boldsymbol{M}\langle tcp\rangle, \boldsymbol{M}\langle q\rangle) = union(\boldsymbol{M}\langle tcq\rangle, project(join(\boldsymbol{M}\langle tcp\rangle, \boldsymbol{M}\langle q\rangle), \{x_1, x_2, x_4\}))$.*

3. *If the iteration reaches Δ-approximation for both $\boldsymbol{M}\langle tcp\rangle$ and $\boldsymbol{M}\langle tcq\rangle$ (as defined in Definition 3.13), the computation stops. Otherwise, we keep iterating.* □

In the above Example 3.24, one can also first compute $\mathbf{M}\langle tcq\rangle$ and then $\mathbf{M}\langle tcp\rangle$, i.e., switching the first two steps, during each iteration. That is to say, there exist many abstraction evaluation orders at each iteration step if multiple predicates are mutually recursive. We choose the order in which these predicates are first defined by rules in our current implementation. Since we are computing Δ-approximations, we

assume that this evaluation order is trivial with respect to final estimates. However, more experimental studies are needed to validate this assumption.

The parameter Δ can be selected in various ways. Larger values make computation reach Δ-approximation sooner, while smaller Δ's cause longer computations, but produce better estimates. Note that the evaluation is *not* guaranteed to reach an approximate fixed point for a chosen Δ, since $size(\mathbf{I}^n)$ in Definition 3.13 may oscillate. In this case, we can stop the iteration over $\mathbf{I}^n$ once oscillation of $size(\mathbf{I}^n)$ is detected and, as a practical measure, we can do with a coarser approximation.

3.3.9 Dependency Matrices for All Predicates

This section presents our algorithm to compute size estimates for all predicates in a bottom-up fashion using the algebra over matrices defined earlier.

Definition 3.14 *Given a knowledge base K, its* **predicate dependency graph** *is a directed graph $G_{pdg}(K) = (\mathcal{N}, \mathcal{E})$ where the set of nodes, $\mathcal{N}$, consists of all predicates contained in K and $(p_1, p_2) \in \mathcal{E}$ if and only if there is a rule in K such that p_1 is the rule's head predicate and p_2 is one of its body predicates.* □

A graph is *strongly connected* if there is a path between any pair of nodes. The *strongly connected components* (SCC) of a directed graph are its *maximal* strongly connected subgraphs. An SCC in a predicate dependency graph contains a maximal subset of *recursive* predicates that mutually depend on one another. Thus, the dependency matrices for all predicates in the same SCC should be iteratively computed until their size estimates all become Δ-approximate for some chosen Δ.

Definition 3.15 *The* **condensation** *of a predicate dependency graph $G_{pdg}(K) = (\mathcal{N}, \mathcal{E})$, written as $G^c_{pdg}(K) = (\mathcal{N}^c, \mathcal{E}^c)$, is a directed* acyclic *graph (a forest of trees) where $\mathcal{N}^c$ consists of all the SCC's of $G_{pdg}(K)$. There is an edge $(scc_1, scc_2) \in \mathcal{E}^c$, where $scc_1 = (\mathcal{N}_1, \mathcal{E}_1)$ and $scc_2 = (\mathcal{N}_2, \mathcal{E}_2)$, if and only if there are $p_1 \in \mathcal{N}_1$ and $p_2 \in \mathcal{N}_2$ such that $(p_1, p_2) \in \mathcal{E}$.* □

Example 3.25 *Let K be the set of rules of Example 3.24, then its predicate dependency graph $G_{pdg}(K) = (\mathcal{N}, \mathcal{E})$ is given in Figure 7(A). There are three SCC's: $scc_1 = (\{tcp, tcq\}, \{(tcp, tcq), (tcq, tcp)\})$, $scc_2 = (\{p\}, \emptyset)$, and $scc_3 = (\{q\}, \emptyset)$; they are given in Figure 7(B). The condensation of $G_{pdg}(K)$ is shown in Figure 7(C).* □

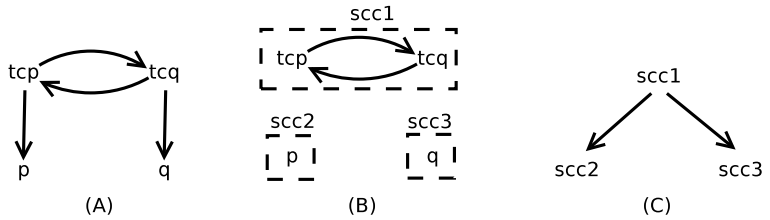


Figure 7: Predicate Dependency Graph of Example 3.25

Given a knowledge base K, let $G^c_{pdg}(K)$ be the condensation of its predicate dependency graph. The dependency matrices for the set of predicates in each tree of $G^c_{pdg}(K)$ can be computed by a bottom-up traversal of the tree, as shown in Algorithm 11. The algorithm resembles the usual naive bottom-up procedure for evaluating Horn rules except that here we employ abstract computation over the size estimation algebra. For instance, let $G^c_{pdg}(K)$ be the graph in Figure 7(C) and T be its only tree. Initially, $sccs = \{scc_2, scc_3\}$ at line 3 of the algorithm. During the first iteration of the `while`-loop, the dependency matrices for predicates contained in scc_2 and scc_3, i.e., $\mathbf{M}\langle p\rangle$ and $\mathbf{M}\langle q\rangle$, are computed, and then, on line 8, we follow the edges of T upwards and set $sccs = \{scc_1\}$. After the second iteration of the `while`-loop, all dependency matrices are computed.

```
1  Let G^c_pdg(K) = (N^c, E^c) be the predicate dependency condensation graph of K;
2  for each tree T ∈ G^c_pdg(K) do
3      Let leaf_sccs be set of leaf SCC's of T;
4      while leaf_sccs ≠ ∅ do
5          for each scc ∈ leaf_sccs do
6              Compute dependency matrices for predicates in scc;
7          end
8          leaf_sccs = {scc_1 ∈ N^c | (scc_1, scc_2) ∈ E^c, scc_2 ∈ leaf_sccs};
9      end
10 end
```

Algorithm 11: Compute Dependency Matrices for All Predicates in K

Algorithm 11 computes dependency matrices for all predicates in the knowledge base in a bottom-up manner, without considering the query. If a query is given then we only need to evaluate one tree in the condensation graph, i.e., the tree that contains the query predicate. Moreover, we need to perform the computation only

for the subtree rooted in the clique that contains the query predicate. We call this subtree the *query spanning tree* of the given query.

Note that, given a set of rules for a predicate p, we can apply estimation operators for computing $\mathbf{M}\langle p\rangle$ in different orders, and this may yield different estimates. Currently, we choose the order in which these rules appear in the program. We believe that this evaluation order is trivial with respect to final estimates.

3.3.10 Complexity Analysis

This section gives the time and space complexity of SDP operations defined in early sections. We assume that predicates mentioned in this section are all n-ary, dependency matrices are all n-dimension $\beta \times ... \times \beta$ and they are stored as *sparse* matrices, the fact-set sizes of base predicates are bounded by *factsize*, and argument domains are bounded by *domainsize* (note that *domainsize* is usually much smaller than *factsize*). The complexity of SDP operations that are defined in earlier sections is summarized in Table 2. It is worth mentioning that typical applications in the Semantic Web community mostly use triple stores and thus their arities are bounded by 4, which makes the overhead of computing predicate sizes affordable. Below, we elaborate the complexity for each operation.

Operation	Complexity
construct $\mathbf{M}\langle p\rangle$	$O(n \times factsize)$
$align(\mathbf{M}\langle r\rangle, a, s)$	$O(\beta^n)$
$project(\mathbf{M}\langle r\rangle, in\_args)$	$O(\beta^n)$
$union(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$	$O(n \times \beta^n)$
$intersect(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$	$O(n \times \beta^n)$
$minus(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$	$O(n \times \beta^n)$
$join(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ on k arguments	$O(\beta^{m+n-k})$
$product(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$	$O(\beta^{m+n})$

Table 2: Complexity of SDP Operations

To construct $\mathbf{M}\langle p\rangle$ from the fact-set $factset(p)$ of a base predicate p is $O(n \times factsize)$. There are three steps in computing $\mathbf{M}\langle p\rangle$ as given in Definition 3.10. First, all maxdiff partitions $d_i^{[j_i]}$ are computed. Consider a fixed i ($1 \leq i \leq n$), d_i can be computed within a scan of $factset(p)$ and thus it is $O(factsize)$. Moreover, d_i can

be grouped into β distribution segments $d_i^{[j_i]}$ in $O(domainsize \times \lg(domainsize))$ since it involves sorting the number of *domainsize* frequency differences to find the $\beta-1$ frequency differentiators. Thus, the complexity of computing all maxdiff partitions is $O(n \times (domainsize \times \lg(domainsize) + factsize))$. Second, dependency matrix values are computed. For each fact, it takes $O(n)$ to determine which $\mathbf{M}\langle p\rangle(i_1, ..., i_n)$ summarizes the fact since we can build indexes from argument values to partitions for all arguments. Therefore, the complexity of computing all dependency matrix values is $O(n \times factsize)$. Finally the attribute values can be computed in $O(n \times domainsize)$ from maxdiff partitions. We then know that the complexity to construct $\mathbf{M}\langle p\rangle$ is $(n \times (domainsize \times \lg(domainsize) + factsize)) + O(n \times factsize) + O(n \times domainsize)$, which is $O(n \times factsize)$, considering *domainsize* is *much smaller* than *factsize*.

The complexity of $align(\mathbf{M}\langle r\rangle, a, s)$ is $O(\beta^n)$. In Definition 3.12, dependency matrix coordinate sizes and parameters are computed in $O(n)$ and $O(n \times \beta)$ respectively, and dependency matrix values are computed in $O(\beta^n)$ since there are $O(\beta^n)$ such values and each value can be computed in constant time. Therefore, the complexity of $align(\mathbf{M}\langle p\rangle, a, s)$ is $O(\beta^n)$.

The complexity of $project(\mathbf{M}\langle r\rangle, in\_args)$ is $O(\beta^n)$. In Algorithm 7, dependency matrix parameters are computed $O(m \times \beta)$. There are $O(\beta^m)$ dependency matrix values, and the complexity to compute each $\mathbf{M}\langle p\rangle(i_1, ..., i_m)$ is $O(\beta^{n-m})$ since the size of $\{\mathbf{M}\langle r\rangle(j_1, ..., j_n) \mid \wedge_{1 \le k \le m} i_k = j_{in_k}\}$ is $O(\beta^{n-m})$ and the *integrate* function is linear. Therefore, the complexity of $project(\mathbf{M}\langle r\rangle, in\_args)$ is $O(\beta^m) \times O(\beta^{n-m}) = O(\beta^n)$.

$Union(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ of Algorithm 4 and $intersect(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ of Algorithm 6 are $O(n \times \beta^n)$. Algorithm 4 assumes that $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ are perfectly aligned, which is $O(n \times \beta^n)$ since *align* is $O(\beta^n)$ and there are n such operations. The parameters are computed in $O(n \times \beta)$ from lines 1 – 4. The dependency matrix values are computed in $O(\beta^n)$ in line 6. Therefore, the complexity of $union(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ is $O(n \times \beta^n)$. Similar argument holds for $intersect(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$.

The complexity of $minus(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ of Algorithm 10 is $O(n \times \beta^n)$. As discussed above in this section, it is $O(n \times \beta^n)$ to perfectly align $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$. dependency matrix parameters are computed in $O(n \times \beta)$ at lines 1 – 3, $\mathbf{M}\langle r\rangle'$ is $O(\beta^n)$ at line 4, and dependency matrix values are computed in $O(\beta^n)$ at lines 5 – 7.

Now, consider the join operation, $join(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$, defined in Algorithm 8. To make $\mathbf{M}\langle r\rangle$ and $\mathbf{M}\langle s\rangle$ aligned on their first k axes is $O(k \times \beta^n)$. Dependency matrix coordinate sizes are computed in $O(n)$ at lines 1 – 6, and parameters are computed in $O(n \times \beta)$ from lines 7 – 16. The projects $\mathbf{M}\langle r\rangle'$ and $\mathbf{M}\langle s\rangle'$ are computed in $O(\beta^n)$ at lines 17 and 18, and finally dependency matrix values are computed in $O(\beta^{m+n-k})$ in line 20 considering there are $O(\beta^{m+n-k})$ such values and each is computed in constant time. Therefore, the complexity of $join(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ on k arguments is $O(\beta^{m+n-k})$.

It is obvious that the complexity of $product(\mathbf{M}\langle r\rangle, \mathbf{M}\langle s\rangle)$ is $O(\beta^{m+n})$ considering there are $O(\beta^{m+n})$ dependency matrix values to compute at lines 13 – 15 of Algorithm 9.

Theorem 3.9 *Given a program with m predicates. Assume that their maximal arity is n, then the number of space units needed to store their dependency matrices is at most $m \times (\beta^n + 3 \times n \times \beta)$.* □

Theorem 3.9 obviously holds considering each dependency matrix needs $(\beta^n + 3 \times n \times \beta)$ amount of space units, where *unit* is the amount of bytes to store one dependency matrix value or coordinate parameter. Given a limited space budget *budget*, the coordinate size β can be any integer satisfying $m \times (\beta^n + 3 \times n \times \beta) \leq budget$. However, larger β values produce better estimates and are more expensive in computation as discussed above.

3.4 Optimization Algorithms

Given a query `?- query` to a knowledge base whose predicate statistical information is available, the query optimizer *greedily* searches for an optimized join order for `query` and each rule's body predicates and adds appropriate indexing commands for each predicate. In our implementation, the search space of join ordering is restricted to left-deep trees which enforce that at least one predicate must be a base predicate in every join step. Figure 8 shows a left-deep search tree for an n-way join of predicates $\{p_i \mid 1 \leq i \leq n\}$ and the tree represents the join ordering of $O = [p_{o_1}, ..., p_{o_n}]$. Therefore, the main task of the optimizing unit is to find such a join ordering based on predetermined performance measures.

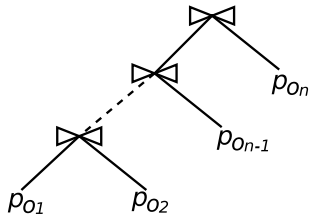


Figure 8: A Left-Deep Search Tree

Algorithm 12 defines the greedy search algorithm of `Costimizer`'s optimizing unit. It starts with an empty join ordering and performs the join of current result r with one more predicate during each step until all predicates are processed. Line 1 initializes the join ordering as $O = [\,]$, $\mathbf{M}\langle r\rangle$ as an empty matrix, the set of arguments of r, denoted $r.args$, as $\emptyset$, and the set of predicates to be ordered as $P = \{p_i \mid 1 \leq i \leq n\}$. During each `while`-loop from line 2 to 17, one predicate p is chosen at line 11 (to be explained later) to join with r. Lines 12 – 13 build the index of p on arguments $index\_args_p$, on which p joins r. Line 14 adds $p.args$ to $r.args$, line 15 updates $\mathbf{M}\langle r\rangle$, and line 16 appends p to O.

1 Initialize $O = [\,]$, r as the join results of predicates in O, $\mathbf{M}\langle r\rangle$ be an empty matrix, $r.args = \emptyset$, and $P = \{p_i \mid 1 \leq i \leq n\}$;
2 **while** $P \neq \emptyset$ **do**
3 **for** *each* $p_i \in P$ **do**
4 **if** $\boldsymbol{M}\langle r\rangle$ *is empty* **then**
5 $\mathbf{M}\langle r \bowtie p_i\rangle = \mathbf{M}\langle p_i\rangle$;
6 **else**
7 $\mathbf{M}\langle r \bowtie p_i\rangle = join(\mathbf{M}\langle r\rangle, \mathbf{M}\langle p_i\rangle)$;
8 **end**
9 $join\_args_{p_i} = (p_i.args \cup r.args) \cap (\cup_{p_j \in P \setminus \{p_i\}} p_j.args)$;
10 **end**
11 Choose p from P where $\frac{size(\mathbf{M}\langle r \bowtie p\rangle)}{\|join\_args_p\|}$ is minimal, and remove it;
12 $index\_args_p = r.args \cap p.args$;
13 Build indexes on $index\_args_p$ for p;
14 $r.args = r.args \cup p.args$;
15 $\mathbf{M}\langle r\rangle = \mathbf{M}\langle r \bowtie p\rangle$;
16 $O = O \bullet [p]$;
17 **end**
18 **return** O

Algorithm 12: Greedy Search Optimization

Now we explain how p is chosen at line 11. Suppose p_i is chosen to join with r at line 3 and the result is $r \bowtie p_i$. Then, line 8 computes $\mathbf{M}\langle r \bowtie p_i \rangle$ and line 9 defines $join\_args_{p_i}$ as the common set of arguments of $r \bowtie p_i$ and the predicates yet to be joined. The idea in greedily choosing p on line 11 is based on two observations. First, the size of the current join result $r \bowtie p$, $size(\mathbf{M}\langle r \bowtie p \rangle)$, should be minimized since $r \bowtie p$ will join with the rest of the predicates in P. Second, the number of arguments in $join\_args_p$ should be maximized because these arguments will be used to restrict future joins.

Now we come back to another important issue that is critical for the performance and scalability of logic engines, *indexing*, as observed in [LFWK09b]. Indexing commands which are believed to benefit query evaluation are automatically generated during the greedy search. Line 12 of Algorithm 12 computes the set of arguments, $index\_args_p$, that should be indexed on predicate p, since the join of p with previous results r is performed on this exact set of arguments.

3.5 Experiments

We tested the performance of the SDP approach to estimate sizes and applied `Costimizer` to query optimization, and this section shares the results and provides analysis. All tests were performed on a machine with an Intel Xeon processor E5-1650 (6-core and 3.2GHz) and 64 gigabytes of main memory. The machine was running Ubuntu 12.10 (Quantal Quetzal) with Linux kernel 3.5.0-17-generic. Our implementation uses XSB Prolog version 3.3.7 (Pignoletto).[5]

3.5.1 Size Estimation

We performed experiments of SDP on size estimation using several rule sets some of which also appeared in [LK10, LK12]. This section analyzes the results for these programs: *transitive closure* (TC), *same generation* (SG), the *mutual recursive transitive closure* (MRTC), and one negation rule set.

[5] `http://xsb.sourceforge.net`

3.5.1.1 Test Parameters

There are three parameters involved in size estimation tests and they are the dependency matrix coordinate size (β), alternative definitions of *integrate* function (Section 3.3.2), and alternatives of *negate_ratio* function (Section 3.3.7). In all tests, we used $\beta \times \ldots \times \beta$ dependency matrices for $\beta = 10, 20, 30, 40, 50$ and all alternative definitions of *integrate* and *negate_ratio*. For recursive predicates, we computed their 5%-approximations.

3.5.1.2 Test Programs

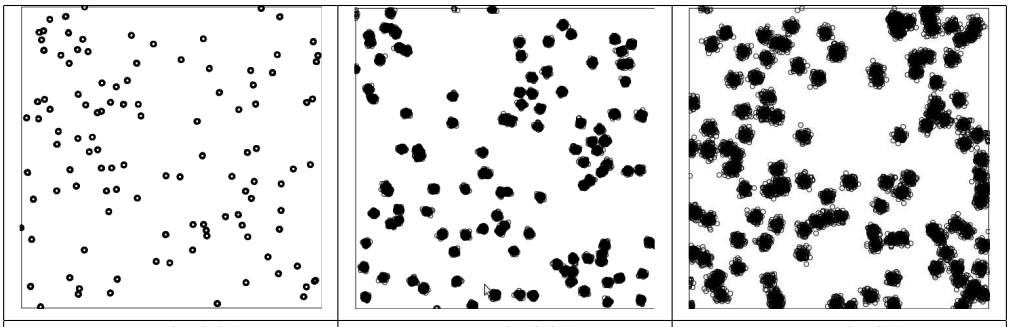
TC. The well-known transitive closure rule set consists of the following two rules

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

where there is a path from x to y, $path(x, y)$, if there is an edge from x to y, $edge(x, y)$, or there is an edge from x to z and a path from z to y. Datasets for *edge* are randomly generated from the Thomas process using $rThomas(\kappa, \sigma, \mu)$ [BT05]. The Thomas process first generates a uniform Poisson point process of parent points with intensity κ and then each parent point is replaced by a random cluster of points. The number of points in each cluster follows Poisson distribution with intensity μ and their positions are isotropic Gaussian displacements (σ) from the cluster's parent point location.

Different datasets can be generated with different parameter values. In our experiments, we used $\kappa = 100$, $\sigma = 0.001, 0.005, 0.01$, and $\mu = 10, 20, 100$. In each case, the number of facts generated is approximately $\kappa \times \mu$ and the argument domain is $[1, 1000]$. Figure 9 shows the datasets generated with $\mu = 100$ but different σ values, where the (x, y)-pair of a fact $edge(x, y)$ is plotted as a point in two dimension planes. It is obviously observable that data dependencies decrease with increasing σ values. Experiments with TC over datasets generated with alternative distributions can be found in [LK10].

SG. The following same generation rule set says that x and y are of the same generation, $sg(x, y)$, if they are siblings, $sibling(x, y)$, or their parents, px and py, are of the same generation.



$\sigma = 0.001$	$\sigma = 0.005$	$\sigma = 0.01$

Figure 9: TC Datasets Generated Using $rThomas(100, \sigma, 100)$

```
sg(X,Y) :- sibling(X,Y).
sg(X,Y) :- parent(X,PX), parent(Y,PY), sg(PX,PY).
```

The number of sibling facts in each dataset is roughly twice that of parent facts. Each test dataset is generated by two separate Thomas processes: $rThomas(\kappa, \sigma, \mu_{sib})$ for *sibling* and $rThomas(\kappa, \sigma, \mu_{par})$ for *parent*, where $\mu_{sib} = 2 \times \mu_{par}$. We tested $\kappa = 100$, $\sigma = 0.001, 0.005, 0.01$, and $\mu_{par} = 5, 10, 50$.

MRTC. MRTC consists of the four rules of Example 3.24 and each dataset has about one million facts of p and one million facts of q from the domain of $[1, 1000]$. The set of facts for p (similarly for q) are randomly generated from a two-step homogeneous Poisson process using system R [BT05]. First, a homogeneous Poisson process of $10,000$ parent facts are generated, and then each parent point is replaced by a cluster of 100 child facts. Each cluster of child points are generated by a separate homogeneous Poisson process from domain $[1, \sigma]$, where σ controls dependencies in the generated datasets. Datasets generated with smaller σ values have higher dependency since they are more clustered together, while those generated with larger σ values have lower dependency. We used $\sigma = 6, 8, 10, 12, 14, 16$ in our tests.

Negation test. We test SDP on negation using this following rule

```
long_path(X,Y) :- path(X,Y), not edge(X,Y).
```

where *path* and *edge* are the same as those in the TC test. The fact $long\_path(x, y)$ says that there is a path of length at least 2 from x to y, i.e., there is a path from x to y but there does not exist an edge from x to y. We use the same datasets as in the case of TC test.

3.5.1.3 Test Results and Analysis

This section presents and analyzes our size estimation results. Since the results of SG and MRTC are similar to those of TC, we will only elaborate the case of TC and omit SG and MRTC in this thesis. More complete experiment details can be found in [LK13a].

TC. Figure 10 shows the size estimates of *path* by SDP against real sizes[6] for datasets generated with $rThomas(100, \sigma, \mu)$. The number of coordinates of each axis used here is $\beta = 30$. For each different μ, one figure presents the sizes estimated with different integration methods (see Section 3.3.2) for datasets with decreasing dependencies (increasing σ).

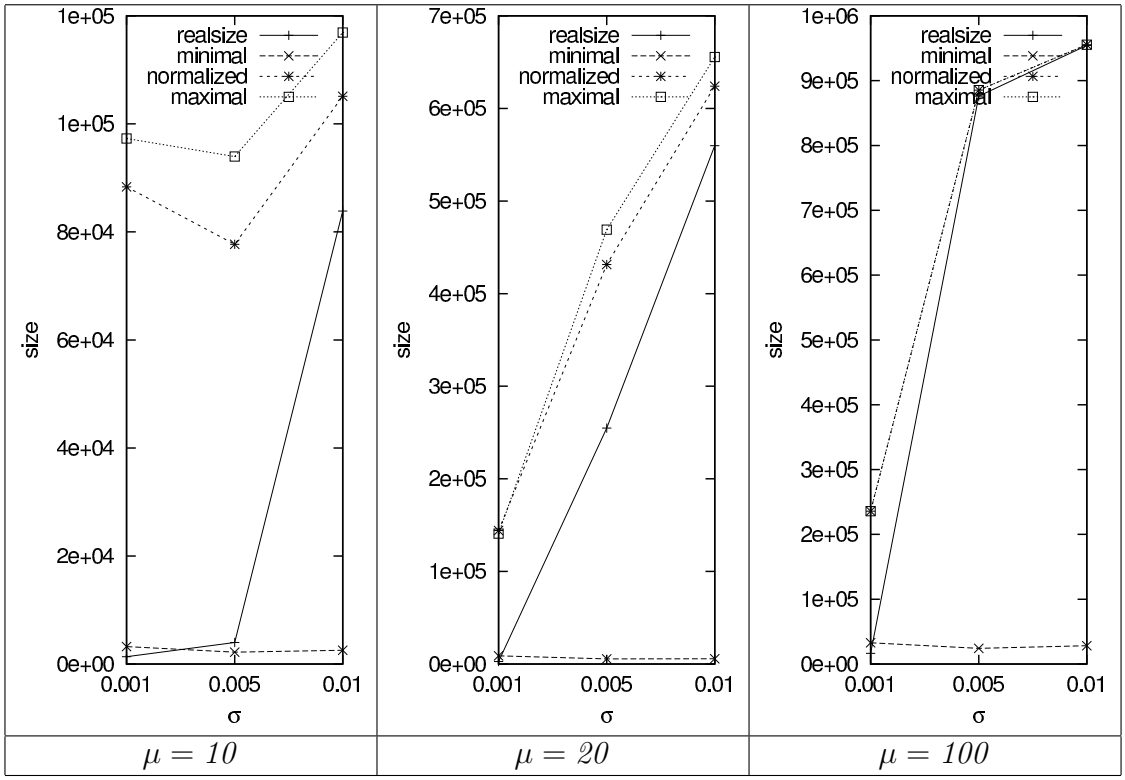


Figure 10: TC Results with Different Integration Methods

There are three important observations. First, in most cases (except that minimal integration performs better for datasets generated by $rThomas(100, \sigma, 10)$ with $\sigma = 0.001, 0.005$), normalized integration produces better estimates than minimal

[6]Real sizes are obtained by evaluating the actual queries.

and maximal integrations, as expected. Second, when normalized integration is used, the estimates preserve the relative order of real sizes except for the dataset of $rThomas(100, 0.005, 10)$. Since most query optimization algorithms use relative sizes to decide on the proper join orderings, this means that our size estimates are sufficiently accurate to be used for that purpose. Third, consider the datasets generated with the same μ parameter. They have roughly the same number of $edge(x, y)$ facts, but the real sizes of their transitive closure ($path(x, y)$) vary dramatically due to the different dependencies between $edge.x$ and $edge.y$ among these datasets. It further validates that cost estimators should take data dependency into account in order to produce useful statistics for recursive predicates.

Figure 11 presents the size estimates of *path* by SDP against real sizes for same datasets from a different perspective. Here, normalized integration is used and for each different μ, one figure presents the sizes estimated with different dependency matrix sizes (β) for datasets with decreasing dependencies. We can see that larger β values, i.e., larger number of coordinates in each axis, produce better estimates. The obvious reason is that larger dependency matrices are preferred to store more accurate dependencies and thus reduce information loss.

Negation test. Negation test results are shown in Figure 12, where the size estimates of *long_path* are compared against its real sizes. Here we only report results for normalized negation because it performs better than minimal and maximal negation. As in the cases of TC tests, for each different μ, one figure presents the size estimates using different β values for all three datasets of decreasing dependencies. We can see that in most cases larger β values, i.e., larger dependency matrices, produce better estimates, as expected. Moreover, our estimates preserve the relative orders of real sizes for reasonably large dependency matrices (e.g., $\beta = 30$), which is very important as discussed above.

SDP Overhead. As all other optimization techniques, our size estimator incurs some overhead. Fortunately, with $\beta = 30$ and normalized integration and negation in our tests, SDP produces good estimates which keep the relative order of real sizes. Under these settings, the cputimes taken by SDP are on average less than 3% of the cputimes to compute real sizes. This overhead is very acceptable compared with the potential performance gain that its size estimates can bring, as demonstrated in

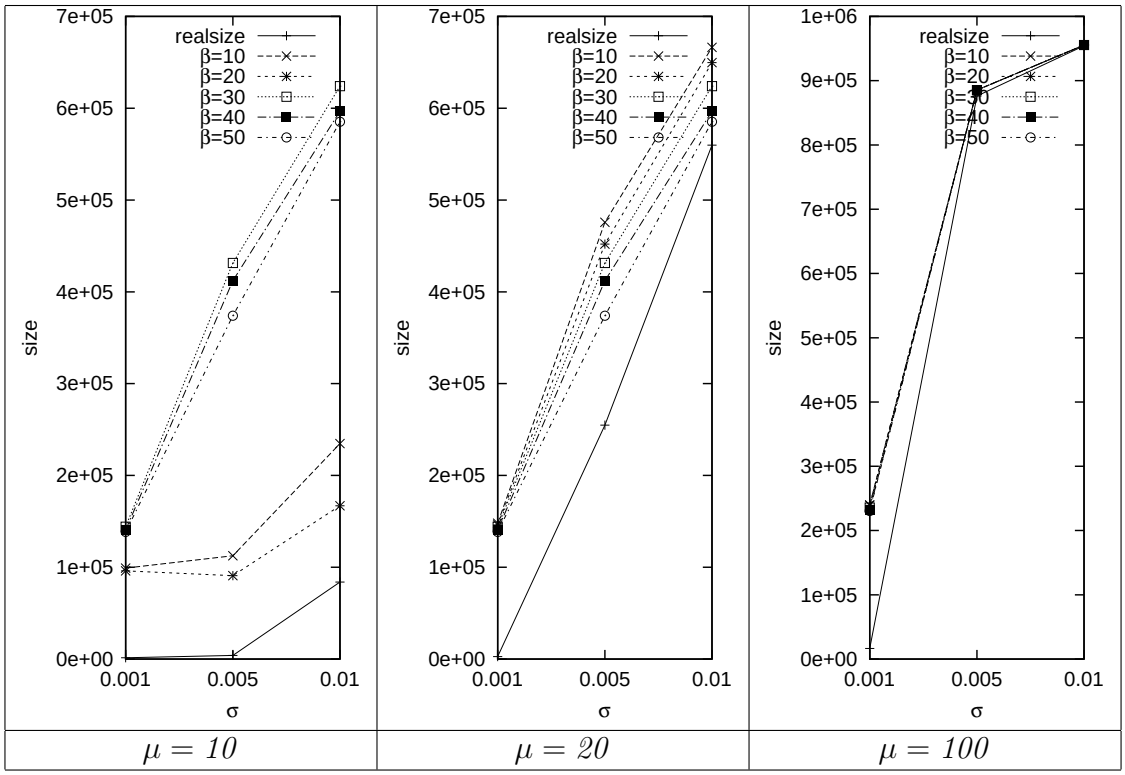


Figure 11: TC Results with Different Dependency Matrix Sizes

Section 3.5.2. Furthermore, we believe this overhead can be substantially reduced considering our implementation itself can be further optimized. We are currently working on this optimization.

3.5.2 Query Optimization

In order to evaluate the performance of `Costimizer` on query optimization, we experimented with several most popular benchmarks including the WordNet tests[7] and the Lehigh University Benchmark (LUBM) [GPH05] which also appeared in OpenRuleBench [LFWK09b], and the Berlin SPARQL Benchmark (BSBM) [BS09].

3.5.2.1 Test Parameters

When computing predicate statistics, we used normalized integration and the number of coordinates of each axis is $\beta = 30$. Given a knowledge base, queries and rule bodies

[7] `http://wordnet.princeton.edu`

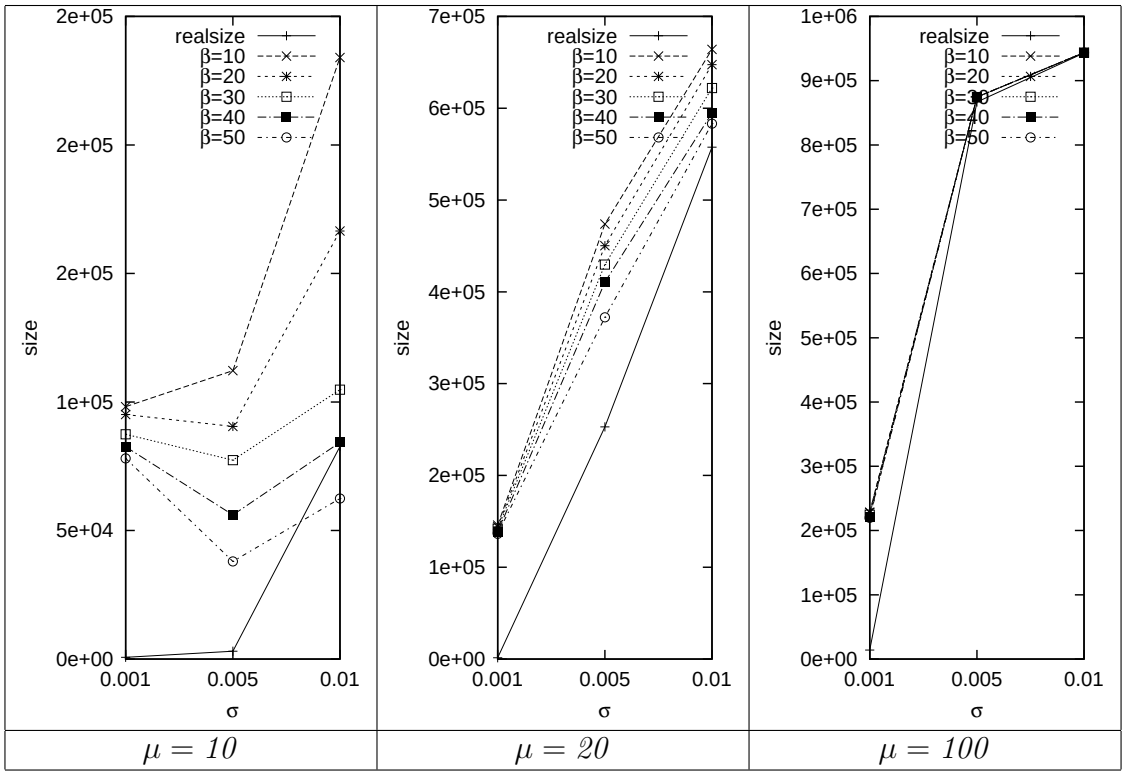


Figure 12: Negation Test Results with Different Dependency Matrix Sizes

are optimized using Algorithm 12. If a query takes more than *30* minutes to evaluate, we kill its evaluation and record a `timeout`.

3.5.2.2 Test Programs

WordNet. The WordNet tests include common queries from natural language processing based on WordNet, a semantic lexicon for the English language. These queries seek to find all hypernyms – words more general than a given word, hyponyms – words more specific than a given word, meronyms – words related by the part-of-a-whole semantic relation, holonyms – words related by the composed-of relation, troponyms, same-synset – groups of semantically equivalent words, glosses, antonyms, and adjective-clusters. The base facts were extracted from WordNet Version 3.0 and converted to the XSB syntax. The database consists of about $115,000$ synsets with over $150,000$ words in total. In most tests, the number of solutions is over $400,000$ (some queries return more than $2,000,000$ answers). Below, we provide the rules

defining *hypernyms*, which is obtained by a join of three predicates following a transitive closure computation. It demonstrates the way in which most of the queries in WordNet tests are defined — see [LFWK09a] for a complete description of the rules and datasets.

```
hypernyms(W1,W2) :- s(S1,_,W1,_,_,_),
                    hypernymSynsets(S1,S2),
                    s(S2,_,W2,_,_,_).
hypernymSynsets(S1,S2) :- hypernym(S1,S2).
hypernymSynsets(S1,S2) :- hypernym(S1,S3), hypernymSynsets(S3,S2).
```

LUBM. The Lehigh University Benchmark is a university database where the number of universities, departments, and students can vary. The dataset for each university contains around $100,000$ facts, and we generated datasets for $1, 5, 10, 50$, and 100 universities. We tested all the 14 queries in the benchmark and here we elaborate on four representative cases. The third LUBM query, *lubm*3, retrieves the publications of a given professor; it is similar to queries $1, 5, 10, 11$, and 13. The fourth query seeks the set of professors, together with their names, emails, and phones, who work for some given department; it is similar to queries 8 and 12. The sixth query finds all students, and it is similar to query 14, which finds all undergraduate students. The ninth query finds these triples of the form <*student*, *faculty*, *course*>, where *faculty* is a faculty advisor of *student*, and *student* takes the *course* from *faculty*; it represents queries 2 and 7.

```
lubm3(Publication) :-
    publication(Publication),
    publicationAuthor(Publication,professor).
lubm4(Professor,Name,Email,Phone) :-
    professor(Professor),
    worksFor(Professor,department),
    name(Professor,Name),
    emailAddress(Professor,Email),
    telephone(Professor,Phone).
lubm6(Student) :- student(Student).
```

```
lubm9(Student,Faculty,Course) :-
    student(Student),
    faculty(Faculty),
    course(Course),
    advisor(Student,Faculty),
    teacherOf(Faculty,Course),
    takesCourse(Student,Course).
```

The *14* queries in LUBM tests explore many different query patterns and they return dramatically different numbers of answers. Therefore, they can be used to test the robustness of `Costimizer`. For the *100*-university dataset, the number of answers for the above four queries are given in Table 3.

query	3	4	6	9
answers	18	34	1,048,533	27,247

Table 3: Number of Answers of LUBM for *100*-University Dataset

BSBM. The Berlin SPARQL Benchmark is built around an e-commerce use case. In this benchmark, a set of products is offered by different vendors and consumers review the products. Its knowledge base consists of *10* base relations such as *vendor*, *product*, *person*, and *review*. The provided data generator supports creation of arbitrarily large datasets using the number of products as the scale factor. We generated datasets with different number of products: $666, 2, 785, 25, 000$, and $70, 812$; these datasets roughly have $50, 000, 200, 000, 2, 000, 000$, and $6, 000, 000$ base facts, respectively. Its queries illustrate search and navigation patterns when a user looks for a product. In our test, we used *10* out of the *12* queries included in BSBM's explore use case and computed all answers, and we detail two of them blow as examples.

```
bsbm1(Number,Label,propertynum1,feature1,feature2,type) :-
    product(Number,Label,_,_,PN1,_,_,_,_,_,_,_,_,_,_,_,_,_,_),
    producttypeproduct(Number,type),
    PN1 > propertynum1,
    productfeatureproduct(Number,feature1),
    productfeatureproduct(Number,feature2).
```

```
bsbm5(Nubmer,Label,product) :-
    product(Number,Label,_,_,PN1,PN2,_,_,_,_,_,_,_,_,_,_,_,_,_),
    product(product,_,_,_,PN1O,PN2O,_,_,_,_,_,_,_,_,_,_,_,_,_),
    productfeatureproduct(Number,Feature),
    productfeatureproduct(product,Feature),
    Number =\= product,
    PN1 < PN1O+120,
    PN1 > PN1O-120,
    PN2 < PN2O+170,
    PN2 > PN2O-170.
```

The first query *bsbm*1 finds a product *number* and *label* given product *type*, a range for *perpertynum1*, and features *feature1* and *feature2*. Query *bsbm*5 finds products that are similar to a given product. In each query, all given argument values are randomly picked from their respective set of possible values. For instance, in *bsbm*5, we randomly selected a product number from the set of all products and provided it to the query as `product`.

All queries except the fifth query return very few answers. For the datasets generated with *70*, *812* products, the number of answers for these queries are given in Table 4.

query	*bsbm*1	*bsbm*2	*bsbm*3	*bsbm*4	*bsbm*5	*bsbm*8	*bsbm*9	*bsbm*10	*bsbm*11	*bsbm*12
answers	0	37	0	0	501	3	22	5	1	1

Table 4: Number of Answers of BSBM for *70*, *812*-Product Dataset

3.5.2.3 Test Results and Analysis

Definition 3.16 *Given a program K and its* `Costimizer`*-optimized version* K_{opt}*, and let cputime and* $cputime_{opt}$ *be the cputimes of evaluating K and* K_{opt}*, respectively. The* **speedup** *of* `Costimizer` *is defined as* $\frac{cuptime}{cputime_{opt}}$. □

WordNet. The original program without optimization took *930.95* seconds to compute all hypernyms, hyponyms, meronyms, holonyms, troponyms, same-synset, glosses, antonyms, and adjective-clusters. If indexing on all arguments of all predicates were added and no cost-based optimization was performed, they took *3.692*

seconds, while our optimized version finished in *3.448* seconds. Here, our optimization did not gain much in performance compared with the version with all arguments indexed, which already has a speedup of *252.15* times. The reason is that all these queries have the pattern where a transitive closure computation is followed by a join of two or three predicates and most of the rules are already in their optimized form. Therefore, the key performance factor here is *indexing*. For instance of the predicate *hypernyms* defined above, the definition of *hyponymSynsets* and the join order of the body predicates in *hypernyms* rule (s, *hyponymSynsets*, and s) are already optimal. The slight performance improvement of `Costimizer` is achieved by optimizing a couple of rules and reducing the overhead of indexing all arguments in all predicates.

LUBM. The results for LUBM tests are given in Table 5, where only the results of above four representative queries are included. Query *lubm*3 sees speedups of *3–8.8*

universities	optimized?	*lubm*3	*lubm*4	*lubm*6	*lubm*9
1	no	0.004	0.004	0.000	1055.218
	yes	0.000	0.000	0.004	0.404
5	no	0.048	0.008	0.048	timeout
	yes	0.016	0.000	0.036	22.093
10	no	0.116	0.012	0.096	timeout
	yes	0.024	0.004	0.104	122.060
50	no	0.976	0.072	0.612	timeout
	yes	0.132	0.004	0.700	timeout
100	no	2.352	0.140	1.268	timeout
	yes	0.268	0.004	1.568	timeout

Table 5: CPU Times of LUBM

times with increasingly large datasets. The fourth query achieves speedups of *3.0–35* times for different datasets. The sixth query, *lubm*6, experiences a slight slow-down. The reason is that *lubm*6 is a simple query, which does not benefit from cost-based optimization, while `Costimizer` brings some small overhead. Query *lubm*9, the most complex query of these four representatives, has a performance improvement of thousands of times. There are three observations worth mentioning. First, `Costimizer` has very good scalability behavior in terms of performance gains. With increasingly larger datasets, the seepdups become even larger. Second, `Costimizer` achieves better performance gains for more complex queries (e.g., *lubm*9) and thus has the potential for optimizing real complex queries. Finally, as with all optimizers, there is some overhead incurred by `Costimizer` and we may experience performance loss for some queries (e.g., the simple query *lubm*6). Actually, one of the important requirements

for a successful query optimizer is that it should provide performance gain in most cases without significant slow-down when it fails to optimize. In our case, the performance loss is within reasonable ranges (*speedups* = *0.8* in the worst case), and it is even less significant considering the slow-downs are for simple queries.

BSBM. The results for the BSBM tests are given in Table 6. Since these queries return very few answers, our optimization did not achieve much performance gain. In these cases, very few tuples are generated as intermediate join results even without optimization, and this limits the efficiency of any cost-based optimization. For the fifth query, which returns more than just a few answers, `Costimizer` achieved a speedup of more than *30* times for all test datasets.

products	optimized?	bsbm1	bsbm2	bsbm3	bsbm4	bsbm5	bsbm8	bsbm9	bsbm10	bsbm11	bsbm12
666	no	0.008	0.000	0.004	0.004	0.016	0.000	0.004	0.004	0.008	0.004
	yes	0.008	0.000	0.004	0.008	0.000	0.004	0.004	0.004	0.004	0.004
2785	no	0.016	0.004	0.080	0.024	0.052	0.012	0.016	0.024	0.020	0.016
	yes	0.020	0.000	0.080	0.024	0.000	0.012	0.016	0.020	0.020	0.016
25000	no	0.788	0.012	0.192	0.212	0.668	0.128	0.140	0.196	0.184	0.168
	yes	0.792	0.012	0.184	0.208	0.020	0.108	0.116	0.172	0.188	0.172
70812	no	2.880	0.032	0.616	0.620	1.908	0.336	0.388	0.528	0.448	0.480
	yes	2.881	0.032	0.564	0.588	0.044	0.284	0.320	0.472	0.448	0.480

Table 6: CPU Times of BSBM

3.6 Related Work

3.6.1 Size Estimation

Traditional estimation techniques. Traditional size estimates use base table statistics and propagate them through derived predicates by assuming independence among arguments. However such estimates could be off by orders of magnitude [SLMK01], which was confirmed by our implementation of the histogram algorithms [LK10] given in [BC02]. In contrast, SDP maintains argument dependencies for both base and derived predicates, providing more accurate estimates.

Graph-based approaches. The present paper was inspired by the size estimation techniques based on *dependency graphs* [SAdM08] and on the *graph-based selectivity estimate approach* [SP06]. In [SAdM08], the statistics for predicates (both base and derived) were kept in dependency graphs, which were essentially $1\times\ldots\times1$ dependency

matrices. Recursive predicates were unfolded up to three steps without considering data distribution. Here dependency matrices keep more fine-grained dependency information and SDP computes the statistics for recursive predicates via incremental evaluation that approximates fixed points and takes base dataset distributions into account. It is unclear how to do this using the framework of [SAdM08] and whether such a generalization is possible at all.

Graph based selectivity estimates were proposed in [SP06], where relational datasets were modeled as graphs and join queries as graph traversals. Fixing a specific set of binary joins, the task in [SP06] was to summarize base data distributions and the joins within the given storage allowance. There are two obvious points that differentiate our work. First, [SP06] required an a priori fixing of all the joins of interest, while SDP does not and thus is more flexible. Second, SDP handles recursion, while [SP06] dealt only with relational queries.

Multi-dimensional histograms. Multi-dimensional histograms have been proposed to keep argument dependency information in literature [MD88, PI97, FM99, DGR01]. [PI97] introduced several different definitions of multi-dimensional histograms and compared their performance with traditional histograms. Multi-dimensional histograms are able to accurately capture argument dependency since they approximate joint data distributions directly by heuristically grouping similar values. They have been implemented to estimate the selectivity of spacial data in Geographic Information Systems [APR99] and dynamic multi-dimensional histograms for continuous data streams were studied in [TGIK02]. However, multi-dimensional histograms are quite expensive to compute and for an n-ary predicate, there are an exponential number of multi-dimensional histograms to compute [PI97]. Moreover, it is not clear how to efficiently compute multi-dimensional histograms for joins and recursions.

Size estimation for recursive predicates. Computing the expected sizes for recursive predicates was studied in [LN89, SN91]. An adaptive sampling algorithm was proposed to estimate the sizes of transitive closures in [LN89], where base relations were modeled as digraphs. They provided estimates within certain confidence intervals of the real sizes. [SN91] studied the expected sizes of transitive closure,

same generation, and canonically factorable recursion of uniformly distributed base datasets. There they proved many asymptomatic expressions about the expected sizes. Our method, SDP, also performs size estimates for recursive predicates, but it is different in that SDP summarizes base relations using dependency matrices and does *not* assume any specific data distribution, although our experimental datasets were generated from the Thomas process and homogeneous Poisson distributions. Second, [SN91] focused on deriving theoretical asymptotic expressions for the expected sizes, while SDP computes size estimates by maintaining data statistics for both base and derived predicates. Therefore, our size estimation algorithms are more practical since most real world datasets do not follow any predetermined distributions.

3.6.2 Optimization Algorithms

There have been extensive research on optimization algorithms, most of which fall into two categories: optimal and greedy algorithms. Two most popular and well studied optimal search algorithms are dynamic programming [OL90, MN06, MN08] and top-down partition search [VM96, DT07], which have similar performances in time and space [DT07]. However, most database systems such as DB2 [GLSW93] and Sybase SQL Anywhere [BP00] implement *greedy algorithms* and restrict search space to left-deep trees [SAC$^+$79].

Dynamic programming computes solutions for a given problem by integrating solutions for its sub-problems of the same form in a bottom-up fashion. When applied to join order optimization, dynamic programming computes the best join order for a n-way join of predicates $P = \{p_1, \ \ldots, p_n\}$ in two steps. First, it computes the optimized join orders for all non-empty subsets of P. Then, it compares the joins of all pairs of P_1 and P_2 such that $P_1 \cup P_2 = P$ and $P_1 \cap P_2 = \emptyset$, and picks the pair with lowest overall join cost. Several dynamic programming algorithms for different join patterns were extensively studied and compared in [OL90, MN06, MN08].

Unlike dynamic programming which computes optimal solutions by composing those for sub-problems, top-down partition search obtains optimality in the opposite direction [DT07]. Consider a join of predicates $P = \{p_1, \ \ldots, p_n\}$. It first partitions P into P_1 and P_2 such that $P_1 \cup P_2 = P$ in all possible ways. Then, unlike dynamic

programming which requires that optimized solutions for P_1 and P_2 be computed, top-down partition search, combined with memoization, recursively computes optimized solutions for P_1 and P_2 and combines them. While not sacrificing optimality, top-down partition search can benefit from several heuristics such as exploring multiple interesting orders and utilizing partial information, which is not feasible in dynamic programming. Compared with dynamic programming, top-down partition search can achieve better performance in computing optimal join orders [DT07]. However,they have the same computational complexity.

Despite their optimality and continuous performance improvements in both time and space, dynamic programming and top-down partition search are rarely implemented in popular systems because of their exponential cost nature, which is prohibitive for large joins. Instead, most systems such as DB2 [GLSW93] and Sybase SQL Anywhere [BP00] implement heuristic algorithms that sacrifice acceptable optimality by restricting search spaces but achieve significant efficiency gain. These greedy algorithms have great practical advantages over optimal search in terms of time and space, and meanwhile produce reasonably competitive results. One of the most common such heuristics is restricting the search space to left-deep trees.

With a known search algorithm, optimizers in traditional database systems then unfold queries and compute execution plans of the unfolded queries based on cost estimates. The challenge in utilizing similar strategies in logic engine optimizers is that such unfolding is not always practical due to recursive predicates. Therefore, `Costimizer` takes a different track by reordering query predicates and each rule's body predicates individually.

Chapter 4

Conclusion

This thesis attempted to address two major problems standing on the way of enabling logic engines to process meaningful queries against large and complex knowledge bases: non-termination analysis and cost-based optimization.

To help user detect, locate, and examine non-termination, we have developed a non-termination analyzer, called `Terminyzer`, for tabled logic engines with subgoal abstraction such as XSB. It includes a suite of algorithms which analyze program execution history and report non-termination causes, i.e., sequences of tabled subgoal calls and their host rule ids. It also reports the rule sequences that get fired cyclically, thus causing non-termination. We then relax the system requirements and study non-termination in tabled logic engines *without* subgoal abstraction. Furthermore, `Terminyzer` attempts to automatically rectify non-terminating programs by heuristically fixing some causes of misbehavior. `Terminyzer` back-ends have been developed for the SILK and $\mathcal{F}$LORA-*2* systems. A graphical interface has been developed by the SILK team and is currently underway for $\mathcal{F}$LORA-*2*.

This thesis also presented a cost-based optimizer, `Costimizer`, which aims at adapting successful optimizations of database systems to rule engines. Such optimizations are highly desirable in order for rule systems to be practical in processing complex queries against large knowledge bases, especially for those engine unfriendly knowledge bases created by knowledge engineers who usually have very limited knowledge of logic engine's evaluation strategies. `Costimizer` first efficiently estimates predicate statistics by preserving argument dependencies and then applies them to

greedily optimize rules and queries. The practicality and usefulness of `Costimizer` have been validated by extensive experimental studies of several major benchmarks. `Costimizer` achieved exciting performance gains in most cases, and, more importantly, it did not significantly slow down queries where it failed to optimize.

Chapter 5

Future Work

Our next step for non-termination analysis will be focused on offering better back-end supports to user-friendly graphical interfaces and performing more experimental studies using large and real-world knowledge bases. We will pursue several directions for cost-based optimization. First, we would like to fully implement `Costimizer` such that it allows user to specify whether optimization should be performed or not, and automatically does it in cases where it is asked. Second, more cost estimation algorithms need to be explored. For instance, partial cost information can be obtained from an execution's forest log. Third, we would like to investigate more optimization algorithms such as the near optimal join algorithms in [Wil02, NPRR12]. Fourth, we want to evaluate `Costimizer` on real large and complex knowledge bases. Finally, the relationships between cost-based optimization and other optimization techniques should be investigated. For example, how mode analysis [Mel85, CLP86, MJMB89, UM94, CU96] can be integrated into `Costimizer`.

Bibliography

[APR99] Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. Selectivity estimation in spatial databases. In *ACM SIGMOD Conference on Management of Data*, pages 13–24, New York, NY, USA, 1999. ACM.

[BAK91] Roland N. Bol, Krzysztof R. Apt, and Jan Willem Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–79, 1991.

[BC02] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *ACM SIGMOD Conference on Management of Data*, pages 263–274, New York, NY, USA, 2002. ACM.

[BCG+07] Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems*, 29, April 2007.

[BP00] I.T. Bowman and G.N. Paulley. Join enumeration in a memory-constrained environment. In *International Conference on Data Engineering*, pages 645–654, Washington, DC, USA, 2000. IEEE Computer Society.

[BS09] Christian Bizer and Andreas Schultz. The Berlin sparql benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24, 2009.

[BT05] A. Baddeley and R. Turner. Spatstat: an R package for analyzing spatial point patterns. *Journal of Statistical Software*, 12(6):1–42, 2005.

[CDR12] Vítor Santos Costa, Luís Damas, and Ricardo Rocha. The YAP prolog system. *Theory and Practice of Logic Programming*, 12:5–34, 2012.

[Chr84] S. Christodoulakis. Implications of certain assumptions in database performance evauation. *ACM Transactions on Database Systems*, 9(2):163–186, 1984.

[CKW93] Weidong Chen, Michael Kifer, and David Scott Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[CLP86] T Y Chen, J-L Lassez, and G S Port. Maximal unifiable subsets and minimal non-unifiable subsets. *New Generation Computing*, 4(2):133–152, April 1986.

[CU96] Kenta Cho and Kazunori Ueda. Diagnosing non-well-moded concurrent logic programs. In *Joint International Conference and Symposium on Logic Programming*, pages 215–229, Bonn, Germany, 1996.

[DDSL+98] Stefaan Decorte, Danny De Schreye, Michael Leuschel, Bern Martens, and Konstantinos F. Sagonas. Termination analysis for tabled logic programming. In *Logic-based program synthesis and transformation*, pages 111–127, London, UK, UK, 1998. Springer-Verlag.

[DGR01] Amol Deshpande, Minos Garofalakis, and Rajeev Rastogi. Independence is good: dependency-based histogram synopses for high-dimensional data. *ACM SIGMOD Conference on Management of Data*, 30(2):199–210, 2001.

[DT07] David DeHaan and Frank Wm. Tompa. Optimal top-down join enumeration. In *ACM SIGMOD Conference on Management of Data*, pages 785–796, New York, NY, USA, 2007. ACM.

[FM99] Pedro Furtado and Henrique Madeira. Summary grids: Building accurate multidimensional histograms. In *Database Systems for Advanced Applications*, pages 187–194, 1999.

[GLSW93] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. Query optimization in the IBM DB2 family. *Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society)*, 16(4):4–18, 1993.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[GS13] Benjamin Grosof and Terrance Swift. Radial restraint: A semantically clean approach to bounded rationality for logic programs. In *National Conference on Artificial Intelligence*. AAAI Press, 2013.

[HBC+12] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. *ACM SIGMOD Conference on Management of Data*, 20(2):268–277, 1991.

[Ioa03] Yannis Ioannidis. The history of histograms (abridged). In *Int'l Conference on Very Large Data Bases*, pages 19–30. VLDB Endowment, 2003.

[IP95] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *ACM SIGMOD Conference on Management of Data*, pages 233–244, New York, NY, USA, 1995. ACM.

[KBL06] Michael Kifer, Arthur Bernstein, and Philip M. Lewis. *Database Systems: An Application Oriented Approach, Complete Version*. Addison-Wesley, Boston, MA, 2006.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.

[LFWK09a] S. Liang, P. Fodor, H. Wan, and M. Kifer. Openrulebench: Detailed report. *Manuscript.* *http://semwebcentral.org/docman/view.php/158/69/report.pdf*, 2009.

[LFWK09b] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. Openrulebench: an analysis of the performance of rule engines. In *Int'l Conference on World Wide Web*, pages 601–610, New York, NY, USA, 2009. ACM.

[Lia12] Senlin Liang. Non-termination analysis and cost-based query optimization of logic programs. In *Web Reasoning and Rule Systems*, pages 284–290, Berlin, Heidelberg, 2012. Springer-Verlag.

[LK10] Senlin Liang and Michael Kifer. Deriving predicate statistics in datalog. In *International Conference on Principles and Practice of Declarative Programming*, pages 45–56, Hagenberg, Austria, July 2010. ACM.

[LK12] Senlin Liang and Michael Kifer. Deriving predicate statistics for logic rules. In *Web Reasoning and Rule Systems*, pages 139–155, Berlin, Heidelberg, 2012. Springer-Verlag.

[LK13a] Senlin Liang and Michael Kifer. Cost based query optimization of logic programming. In *Technical Report*, 2013.

[LK13b] Senlin Liang and Michael Kifer. A practical analysis of non-termination in large logic programs. In *Int'l Conference on Logic Programming*, 2013.

[LK13c] Senlin Liang and Michael Kifer. Terminyzer: An automatic non-termination analyzer for large logic programs. In *Practical Aspects of Declarative Languages*, Berlin, Heidelberg, New York, 2013. Springer-Verlag.

[LN89] R. J. Lipton and J. F. Naughton. Estimating the size of generalized transitive closures. In *Int'l Conference on Very Large Data Bases*, pages 165–171, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[LSS04] Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. Proving termination for logic programs by the query-mapping pairs approach. In *Program Developments in Computational Logic*, pages 453–498. Springer LNCS, 2004.

[MD88] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *ACM SIGMOD Conference on Management of Data*, pages 28–36, Chicago, Illinois, 1988. ACM.

[Mel85] Chris Mellish. Some global optimizations for a prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.

[MJMB89] André Mariën, Gerda Janssens, Anne Mulkers, and Maurice Bruynooghe. The impact of abstract interpretation: An experiment in code generation. In *Int'l Conference on Logic Programming*, pages 33–47, Edinburgh, U.K, 1989.

[MN06] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Int'l Conference on Very Large Data Bases*, pages 930–941. VLDB Endowment, 2006.

[MN08] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *ACM SIGMOD Conference on Management of Data*, pages 539–552, New York, NY, USA, 2008. ACM.

[NDS07] Manh Thang Nguyen and Danny De Schreye. Polytool: proving termination automatically based on polynomial interpretations. In *Logic-based program synthesis and transformation*, pages 210–218, Berlin, Heidelberg, 2007. Springer-Verlag.

[NGSKDS08] Manh Thang Nguyen, Jürgen Giesl, Peter Schneider-Kamp, and Danny De Schreye. Termination analysis of logic programs based on dependency graphs. In *Logic-Based Program Synthesis and Transformation*, pages 8–22. Springer-Verlag, Berlin, Heidelberg, 2008.

[NM99] Ulrich Neumerkel and Frédéric Mesnard. Localizing and explaining reasons for non-terminating logic programs with failure-slices. In *International Conference on Principles and Practice of Declarative Programming*, pages 328–342, London, UK, 1999. Springer-Verlag.

[NPRR12] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *ACM Symposium on Principles of Database Systems*, pages 37–48, New York, NY, USA, 2012. ACM.

[OCM00] Enno Ohlebusch, Claus Claves, and Claude March. TALP: A tool for the termination analysis of logic programs. In *Rewriting Techniques and Applications*, pages 270–273. Springer-Verlag, LNCS, 2000.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Int'l Conference on Very Large Data Bases*, pages 314–325, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[Pay07] Étienne Payet. Detecting non-termination of term rewriting systems using an unfolding operator. In *Logic-based program synthesis and transformation*, pages 194–209, Berlin, Heidelberg, 2007. Springer-Verlag.

[PHIS96] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM SIGMOD Conference on Management of Data*, pages 294–305, New York, NY, USA, 1996. ACM.

[PI97] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Int'l Conference on Very Large Data Bases*, pages 486–495, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[PM06] Etienne Payet and Fred Mesnard. Nontermination inference of logic programs. *ACM Transactions on Programming Languages and Systems*, 28:256–289, March 2006.

[RSar] Fabrizio Riguzzi and Terrance Swift. Terminating evaluation of logic programs with finite three-valued models. *ACM Transactions on Computational Logic*, to appear.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM SIGMOD Conference on Management of Data*, pages 23–34, New York, NY, USA, 1979. ACM.

[SAdM08] Damien Sereni, Pavel Avgustinov, and Oege de Moor. Adding magic to an optimising datalog compiler. In *ACM SIGMOD Conference on Management of Data*, pages 553–566, New York, NY, USA, 2008. ACM.

[Sah93] Dan Sahlin. Mixtus: An automatic partial evaluator for full prolog. *New Generation Computing*, 12(1):7–51, March 1993.

[SD94] Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[SDSV10] Yi-dong Shen, Danny De Schreye, and Dean Voets. Termination prediction for general logic programs. *Theory and Practice of Logic Programming*, 9(6):751–780, January 2010.

[She97] Yi-Dong Shen. An extended variant of atoms loop check for positive logic programs. *New Generation Computing*, 15(2):187–203, May 1997.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[SkGS+10] Peter Schneider-kamp, JÜrgen Giesl, Thomas StrÖder, Alexander Serebrenik, and RenÉ Thiemann. Automated termination analysis for logic programs with cut*. *Theory and Practice of Logic Programming*, 10(4-6):365–381, July 2010.

[SLMK01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's learning optimizer. In *Int'l Conference on Very Large*

Data Bases, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[SN91] S. Seshadri and Jeffrey F. Naughton. On the expected size of recursive datalog queries. In *ACM Symposium on Principles of Database Systems*, pages 268–279, New York, NY, USA, 1991. ACM.

[SP06] Joshua Spiegel and Neoklis Polyzotis. Graph-based synopses for relational selectivity estimation. In *ACM SIGMOD Conference on Management of Data*, pages 205–216, New York, NY, USA, 2006. ACM.

[SW12] Terrance Swift and David Scott Warren. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12:157–187, January 2012.

[SWS+13] Terrance Swift, David S. Warren, Konstantinos Sagonas, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, Luis de Castro, Rui F. Marques, Diptikalyan Saha, Steve Dawson, and Michael Kifer. The XSB system, version 3.3.x. volume 1: Programmer's manual. 2013.

[SYY01] Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Loop checks for logic programs with functions. *Theoretical Computer Science*, 266(1-2):441–461, 2001.

[TGIK02] Nitin Thaper, Sudipto Guha, Piotr Indyk, and Nick Koudas. Dynamic multidimensional histograms. In *ACM SIGMOD Conference on Management of Data*, pages 428–439, New York, NY, USA, 2002. ACM.

[UM94] Kazunori Ueda and Masao Morita. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.

[VDS09] Dean Voets and Danny De Schreye. A new approach to non-termination analysis of logic programs. In *Int'l Conference on Logic Programming*, pages 220–234, Berlin, Heidelberg, 2009. Springer-Verlag.

[VDS11] Dean Voets and Danny De Schreye. Non-termination analysis of logic programs using types. In *Logic-based program synthesis and transformation*, pages 133–148, Berlin, Heidelberg, 2011. Springer-Verlag.

[VDSS01] Sofie Verbaeten, Danny De Schreye, and Konstantinos Sagonas. Termination proofs for logic programs with tabling. *ACM Transactions on Computational Logic*, 2(1):57–92, January 2001.

[VM96] Bennet Vance and David Maier. Rapid bushy join-order optimization with cartesian products. In *ACM SIGMOD Conference on Management of Data*, pages 35–46, New York, NY, USA, 1996. ACM.

[Wil02] Dan E. Willard. An algorithm for handling many relational calculus queries efficiently. *Journal of Computer and System Sciences*, 65(2):295–331, 2002.

[Zho12] Neng-Fa Zhou. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218, 2012.