INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Estimating the Cost of GraphLog Queries

by

Carlos Escalante Osuna
Licentiateship (B.Sc.), Universidad Iberoamericana, Mexico, 1988
M.Sc., University of Victoria, 1992

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. R.N. Horspool, Supervisor (Department of Computer Science)

Dr. W.W. Wadge, Departmental Member (Department of Computer Science)

Dr. M. van Emden, Departmental Member (Department of Computer Science)

Dr. W.J.R. Hoefer, Outside Member (Department of Electrical and Computer Engineering)

Dr. A.G. Ryman, External Examiner (IBM Canada Laboratory)

Supervisor: Dr. R. Nigel Horspool.

ABSTRACT

This dissertation develops a cost model for a particular implementation of the database query language GraphLog. The order in which the subgoals of a GraphLog query are executed has a major effect on the total processing time. Our model may be used to compare the expected execution costs for different orderings of the same general query, thus, allowing us to select an efficient execution plan. We describe two cost models: one that is tailored to a specific architecture and another that is more general. Both models assume a top-down evaluation strategy. In particular, we address the issue of how to handle recursive predicates. We also provide some experimental results that confirm the validity of our work.

Examiners:

Dr. R.N. Horspool, Supervisor (Department of Computer Science)

Dr. W.W. Wadge, Departmental Member (Department of Computer Science)

Dr. M. van Emden, Departmental Member (Department of Computer Science)

Dr. W.J.R. Hoefer, Outside Member (Department of Electrical and Computer Engineering)

Dr. A.G. Ryman, External Examiner (IBM Canada Laboratory)

Table of Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank Dr. Horspool for his patience and encouragement; IBM Toronto Laboratory for suggesting the topic, providing a Ph.D. fellowship and hosting a work term; Dr. Wadge and Dr. Ryman, who offered a number of insights; and, finally, last but certainly not least, my parents and brother, for their long-standing devotion and support.

To:

Jan Doumen (DeJean)

Horacio Franco

Sjoerd Mullender

Bruno Cornec

Gwenael Faucher

Bogislav Rauschert

Federico Marincola

Dave Lampson

György Varga

Ken-ichi Murata

Shel Ritter

Gustav Leonhardt

Sigiswald Kuijken

Grupo Cinco Siglos

Chapter 1. Introduction. Query Optimization in GraphLog

In this dissertation, we propose a cost model for GraphLog, a query language that is based on a graph representation of both databases and queries. Specifically, GraphLog is the query language used by *4Thought*, a software engineering tool aimed at helping engineers understand and solve a class of software engineering problems that involve large sets of objects and complex relationships amongst them [Consens92] [Ryman92] [Ryman93]. GraphLog queries ask for patterns that must be present or absent in the database graph. Our framework is able to estimate the relative cost of execution of different orderings of semantically equivalent GraphLog queries, thus allowing us to reject those query orderings whose execution may be more inefficient. Our model assumes a top-down evaluation strategy [Ceri90].

Given the fact that one of the distinguishing characteristics of GraphLog is the capability to express queries with recursion or closures, and since no previous cost model has addressed the cost estimation of recursion and closures for a GraphLog-like language, our original solution to this problem is of particular interest. Our methodology has been evaluated on several real-life databases with encouraging results.

In this chapter, we analyze some general issues relevant to query optimization in general, and query reordering in particular. We also introduce the language that our work will be applied to. Finally, we give an overview of what we have accomplished.

1.1 Query Optimization

Query optimization [Jarke84] is directly concerned with the efficient execution of database queries. Its main goal is to minimize the resources needed to evaluate a query that retrieves information from a given database. A query optimizer normally generates and analyzes different alternatives to determine an efficient plan of execution. Optimizing a query can reduce processing time by a factor whose value depends on the sizes of the

database definitions†. This decision is often based on cost models that capture the contributions due to different factors such as the sizes of the relations under consideration or the expected number of tuples retrieved by an intermediate operation.

If, for instance, a user poses the query "find all Japanese collectors who own a Stradivarius violin", the query optimizer would usually need some information about the statistical profile of the database (how many Japanese collectors are stored in the database, how many individuals are expected to own a Stradivarius violin, and more). Given these premises, the optimizer may establish a suitable plan to solve the problem efficiently. A plan of execution has to take into account several different factors, including the order of operations, the searching algorithms that are used and the database structure itself.

Some of the most common strategies adopted in query optimization include:

1. Selection of the most efficient overall evaluation method (i.e., the computational model that derives all the solutions to the query). The algorithm that is used to search for the answers clearly has an influence on the efficiency of execution of the query. No evaluation method is intrinsically superior to the others. In fact, the performance of different evaluation methods depends on the nature of the problem. Typical evaluation methods include bottom-up evaluation, top-down-evaluation, and combinations of both. Here, the optimization (i.e., the decision as to which evaluation method is the most suitable for the given query) is performed during the evaluation process itself.

2. Determination of the best syntactic rearrangement of the query subgoals. Given that the order of execution of the subgoals can substantially influence the time that is required to retrieve the answers to the query, it is usually advantageous to find the goal ordering that is the least expensive to execute. Unfortunately, since the number of combinations increases geometrically with the number of subgoals in the query, an exhaustive search through all possible combinations may become

---

†For instance, we will show a simple example in which a reduction factor of 2,000 is achieved.

prohibitive. A practical cost model is needed to compare the performance of different orderings and select a suitable (efficient) ordering.

3. Transformation of the original user query into an equivalent one which can be executed more efficiently. In some cases, standard simplifications may be applied to the new query, whereas they may not have been applicable to the initial query. However, this process of query rewriting does not guarantee that a more efficient query will be found. In some cases, a loss in efficiency may occur.

If the evaluation is performed by a specific "machine", we will be more interested in the last two approaches to query optimization (a fixed evaluation strategy is the usual case for many query languages).

Our work will address the issue of selecting the best syntactic rearrangement of the query subgoals for a specific query language, namely GraphLog [Consens89]. We will refer to this problem as *query reordering*.

1.2 Datalog

There has been extensive work directed towards tackling the traditional *database programming paradigm*. However, with a recent trend towards integrating the database and logic programming paradigms, new requirements and challenges demand a different approach to the special problems raised by the *logic programming paradigm*. This dissertation is specifically focused on GraphLog, a language that incorporates the two above-mentioned programming paradigms. Since GraphLog is closely related to Datalog, a relatively well-known logic query language, we proceed to give a brief overview of this language.

Datalog [Ullman88] is a language that applies the principles of logic programming to the field of databases. Datalog was specifically designed for interacting with large databases. The language is based on first-order Horn clauses without structures as arguments, i.e., only constants and variables are allowed. Constant arguments are also referred to as *ground* atoms. Most underlying Datalog concepts are similar to those in Logic Programming [Ceri90]. In fact, the design of Datalog has been noticeably influenced

by one of the most popular logic programming languages, Prolog [Clocksin81]. We proceed to give a brief description of the language. A more detailed coverage of the language can be found in the literature [Ullman88] [Gardarin89] [Ceri90].

A Datalog program consists of a finite set of logic clauses often referred to as *facts* and *rules*. Facts are assertions that define true statements about some objects and their relationships. Typical facts are "*Felix is a man*" or "*The square of 5 is 25*". The Datalog notation for these facts is:

```
male(felix).
square(5, 25).
```

The atomic symbol that names the relationship is said to be the *predicate* definition. In the example, *male* and *square* are predicate symbols. The objects that are affected by the relationships are named the *arguments* or data objects. In our example, these are the constant values *felix*, *5* and *25*. As a notational convention, both predicate symbols and constant arguments are written with an initial lower-case letter. The collection of facts is usually referred to as the *database*.

Rules are collections of statements that establish some general properties of the objects and their relationships. Broadly speaking, rules permit the derivation of facts from other facts. A Datalog rule is expressed in the form of Horn clauses [Horn51], that is, clauses having the general form:

$$P \text{ if } Q_1 \text{ and } Q_2 \text{ and } \ldots \text{ and } Q_n$$

or, in Datalog notation,

```
p :- q1, q2, ...., qn.
```

p being the *head* of the rule and the conjunctive part being the *body* of the rule. Each q_i is named a *subgoal* of the rule.

Rules usually make use of *variables* to represent general objects rather than specific ones. Variables are represented by identifiers that must commence with a capital letter.

For example, the predicate

```
son(X,Y) :- male(X), parent(Y,X).
```

can be interpreted as "*X is a son of Y if X is male and Y is a parent of X*". The predicates *male* and *parent* should be defined elsewhere, either as facts or as rules.

The user may request information from the database by entering *queries*. These are Horn clauses which lack a head and can be evaluated or verified against the facts and rules in the program. For example, the query

```
:- patient(Name, Disease), tropical(Disease).
```

may be used to retrieve the names of those patients that have suffered a tropical disease according to their clinical history. The answer to this query is given by the set of *all* tuples that satisfy the query.†

1.3 GraphLog

A related language is GraphLog [Consens89]. GraphLog is a graphical database query language based on Datalog, and enriched by some additional features (specifically, the formulation of path regular expressions). One of its original aims was to facilitate programming via a graphical representation of the programmer's designs and intentions. The main idea is that a relational database can be represented as a graph, and graphs are a very natural representation for data in many application domains (for instance, transportation networks, project scheduling, parts hierarchies, family trees, concept hierarchies and Hypertext) [Consens89] [Consens90] [Fukar91] [Consens92] [Ryman92] [Ryman93].

Each *node* in the graph is labelled by a tuple of values: they correspond to the attribute values in the database. Each *edge* in the graph is labelled by a name of a relation and an optional tuple of values. The set of values in both the edge label and the nodes connected by the edge, together with the name of the relation in the edge, correspond to

---

†For practical reasons, some systems have the option of retrieving just a subset of the whole answer (by reporting the first instances of the solution that are derived).

one tuple in the database. Figure 1.1 shows three equivalent graph representations of the fact:

square(5, 25).

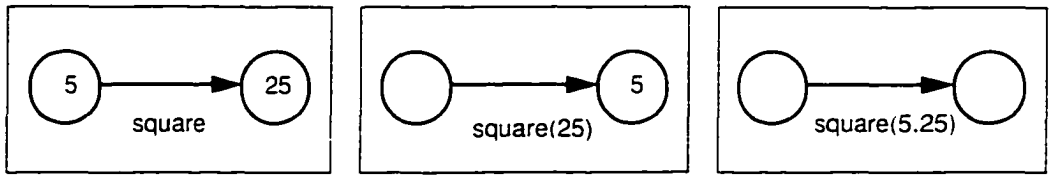


Figure 1.1 Three representations of a given database tuple

General relations (rules) and queries may also be represented by graphs. Every edge in the graph represents a relation amongst data objects as represented in the nodes connected by the edge (and optionally in the edge). These data objects are the predicate arguments and they can be either variables or constants. The rule itself is represented by a special edge (called the *distinguished* edge) that also connects a pair of nodes. For instance, Figure 1.2 shows a graph representation of the rule:

son(X,Y) :- male(X), parent(Y,X).

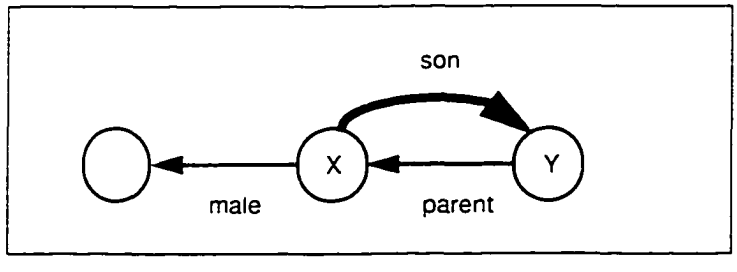


Figure 1.2 A graph representation of a rule

Another example of a GraphLog relation is given in Figure 1.2. In this case, the following rule is defined:

updown(X,YU,XU,Y) :- up(X,XU), down(YU,Y).

This example shows that the graph does not have to be a connected graph. Note also that the arguments are ordered as follows[†]: (a) first those appearing in the "starting" node; (b) those shown in the "ending" node; and (c) those specified in the edge.

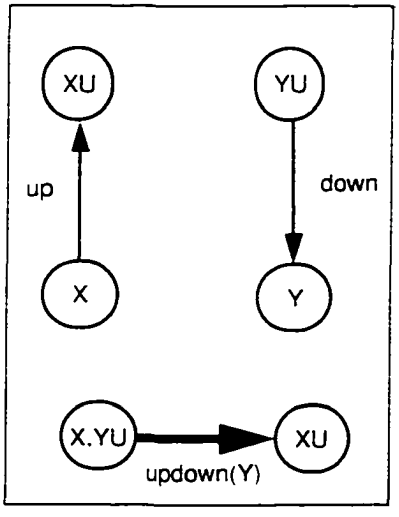


Figure 1.3 A graph representation of a GraphLog relation

GraphLog is a language that represents database facts, rules and queries as graphs as described above. A formal definition of this query language can be found in [Consens89]. It is shown that a GraphLog program has an equivalent Datalog program associated with it. Of particular relevance is the fact that GraphLog allows programmers to express *recursive* relations, thus providing a greater expressive power than that of traditional relational algebra.

1.4 The Importance of Query Reordering

The efficiency with which a logic programming language[‡] executes a query is critically dependent on the order in which goals are expressed in a conjunction [Warren81]. *Query reordering* is an important query optimization technique for finding more efficient evaluation orders for the predicates. The main goal of this technique is to reduce the number of alternatives to be explored.

[†]In fact, arguments may be specified in prefix, postfix or infix notation. *4Thought* favours the infix convention.

[‡]It is assumed that the specific resolution technique used is SLD-resolution [Ceri90].

To determine more efficient ways of evaluating a given set of subgoals, it is convenient to have some information about the actual (extensional) database. Knowledge of some parametric values of the database can help determine an approximate execution cost that is to be associated with every subgoal. Query reordering usually requires at least three different processes: (a) gathering a database profile or some general knowledge on the characteristics of the database tuples, (b) estimating costs for different orderings (in the ideal case, for *all* possible valid orderings)†, and (c) determining the best order. In this dissertation, we concentrate on the second issue, i.e., trying to predict the (relative) cost of evaluating a query (*any* query) for a given database.

1.4.1 Effect of Query Reordering

To illustrate the effect that query reordering may have on the performance of a query, we use the following example that describes a Prolog database‡.

Example. Consider a database that consists of three predicates:

- book(Title, Publisher_Name, Author_Name). A collection of book titles along with their publishers and authors.
- publisher(Publisher_Name, City). A list of different cities where book publishers have an authorized distributor.
- author(Author_Name, Nationality). A group of facts that relate authors to their respective nationalities.

Suppose that we wish to retrieve a list of tuples <Title, Publisher_Name, City, Author_Name> of those publications whose author has Dutch nationality.

---

†Although the database profile may be used to estimate the cost of some simple subgoals (for instance, facts), the cost of more complex (derived) subgoals requires some additional computational work.

‡These results also apply to GraphLog, especially since GraphLog queries are usually translated into Prolog under current implementations of the language.

Since this query involves all three predicates, there are 3! different ways to express it:

```
:- book(T, P, A), publisher(P, C), author(A, dutch).
:- book(T, P, A), author(A, dutch), publisher(P, C).
:- publisher(P, C), book(T, P, A), author(A, dutch).
:- publisher(P, C), author(A, dutch), book(T, P, A).
:- author(A, dutch), publisher(P, C), book(T, P, A).
:- author(A, dutch), book(T, P, A), publisher(P, C).
```

The answer will be the same, regardless of the chosen order. However, depending on the characteristics of the underlying database, the timings of the queries will not be the same. For example, we applied all six orderings to a particular database with 3,000 book titles, 20 different publishers, 450 authors, 30 nationalities and 380 cities worldwide, and observed the costs shown in Table 1.1. The figures were obtained using SICStus Prolog version 1.2 and Stony Brook Prolog (SB-Prolog) version 3.0 measured on a Sun SPARCstation SLC. All execution times are estimated, according to the implementation manuals, in "artificial" units. The database under consideration comprised 3,000 facts for the book predicate, 2,766 facts for the publisher predicate and 450 facts for the author predicate.

ordering	cost using SICStus Prolog	cost using SB-prolog
publisher-author-book	3434745	3152460
author-publisher-book	3438660	3125060
publisher-book-author	260040	443900
book-publisher-author	41345	242080
author-book-publisher	2690	2810
book-author-publisher	1635	3215

Table 1.1 Cost of the evaluation of a given query using different orderings

It is clear from this example that the order of the subgoals substantially affects the performance of the Prolog query. It is also evident that the particular Prolog implementation may affect the choice of the best ordering as well.

1.5 Our Dissertation

A cost model of a particular implementation of the language GraphLog (in which Prolog is the target program) is proposed in this dissertation. In particular, we address the issue of ranking different (syntactically-equivalent) arrangements of a given query in order to select the (potentially) most efficient ordering. One major feature of our methodology is the ability to estimate the cost of recursive queries and transitive closures.

1.5.1 The Problem Solved

Essentially, we have derived a methodology that allows us to choose a potentially less expensive ordering amongst a group of valid subgoal orderings. In other words, our proposed framework is able to rank different orderings according to their expected execution cost. Rather than assigning absolute values (i.e., exact execution times) to the different orderings under consideration, we are only interested in predicting their expected relative cost. Execution time is used as the determining factor in the analysis.

We may state the general problem as follows:

Given a GraphLog query q of the form:

$:- s_1, s_2, \ldots, s_m.$

we are to estimate the relative cost of any given ordering of the subgoals.

Our methodology only ranks different orderings. It does not select potentially good candidates from the whole spectrum of valid orderings. It is the responsibility of a preprocessor to select a subset of potentially cheap orderings to start with (especially if the number of permutations of orderings would make an exhaustive analysis prohibitive). In fact, since we are interested in finding a permutation of the subgoals that yields a more efficient plan of execution, there are at most $m!$ possible orderings (some of them may be invalid as they may not comply with the safety rules of the query language) so that it is not always feasible to test them all individually. A practical approach is to select a subset of the orderings, namely those that are potentially less expensive to execute. Then, we can estimate the cost of execution of each ordering in the subset to determine a good

ordering. There are several methods to select subsets of potentially efficient orderings, amongst them, Sheridan's algorithm [Sheridan91] and simulated-annealing-based algorithms [Ioannidis90].

1.5.2 Overview of Our Cost Model

In general, we have assumed that some information about the underlying database† is available. Sheridan's algorithm [Sheridan91] is the framework of choice when no information regarding the databases can be obtained.

For any given ordering, a mode analysis [Debray88] is performed to determine the degree of instantiation of the subgoal arguments. For the case of the previously-mentioned Prolog implementation of GraphLog, our model takes into account the specific evaluation strategy of this language under a particular implementation (namely, the WAM [Aït91]).

We have chosen to consider what we call the *average* behaviour for queries. Given *all* possible valid queries that the user may pose for a particular calling pattern (cf. Debray's framework), we estimate an *average* value of all their expected execution timings and use this value as the *expected* cost of the given query.‡ The framework in its present state does not produce any additional information such as measures of the dispersion of the values with respect to the average value, or corresponding upper and lower bounds.†† Furthermore, rather than a detailed and expensive exact solution, our model considers the process of solving a query as a set of general actions only.

We have determined that a convenient way to obtain a suitable ranking for the orderings under study is to consider the existence of what we have called *cost contributors*, that we proceed to explain in the following subsection.

---

†For instance, we assume that the number of tuples for each database fact and the number of distinct values for each argument position are available.

‡Thus, we are assuming that all queries have an equal probability of being posed, which is a major assumption.

††In fact, we decided not to use intervals to characterize the results based on the fact that for a transitive closure, the resulting intervals were normally too wide to be of practical use.

Additionally, we have developed a methodology to estimate the average number of solutions associated with the query, this being an implementation-independent quantity. In fact, Debray and Lin's related work [Debray93], that derives a cost model of logic programs, is mainly concerned with this sole issue. Our model is more general as it handles recursive and closure predicates.

One major consideration that was regarded as essential since the inception of this dissertation was to produce a *simple* as possible framework, while producing yet acceptable results. We strongly believe that our model is simple, both conceptually and from the point of view of a practical implementation. We have tested our methodology on several real-life (large) databases. Some detailed case studies are given in Chapter 6.

Cost Contributors

Rather than analyzing the nature of the exact machine code that is generated (for instance, in the form of machine cycles that are required to execute the instructions), a simpler analysis is often desirable, although at the expense of a potential loss in precision. The general idea is to determine some generic activities or groups of operations that are directly related to the cost of execution of the query and then estimate the individual costs associated with such components. Therefore, we wish to single out some "cost contributors" that influence the efficiency of the code execution. Some typical cost contributors are (1) the number of tuples in the database that are visited to find the global solution, (2) the number of matching (unification) attempts that take place during the resolution process, and (3) the number of solutions or answers to the query that are gathered and displayed (we also have to consider any associated backtracking that may occur when new solutions are attempted). Some contributors may have a greater impact on the query performance than others. For instance, it has been reported that a Prolog program may spend 55-70% of its time unifying and 15-35% of its time backtracking [Woo85].†

†This behaviour is specially relevant to our work, since the current implementation of the GraphLog interpreter generates Prolog code as the target language. For this reason, the number of visited tuples is a relevant cost contributor (if not the most relevant).

Unfortunately, many of these quantities are both model- and machine dependent. For example, if the model uses *clause indexing* to narrow down the number of clauses to be explored, fewer tuple visits and unifications will be performed. Similarly, if specialized code optimizations are incorporated, this may have an impact on various cost contributors (for instance, tail recursion optimization [Kruse87] may reduce the cost associated with backtracking). The only cost contributor that is independent of the execution model seems to be the total number of solutions to the query, but, in the case of GraphLog, this number is also independent of whatever ordering of the subgoals is selected!

In our model, one initial task consists of defining which *cost contributors* are more relevant. By eliminating some cost contributors, the process of cost estimation will be simplified at the expense of some loss in precision. As we will argue later, many real-life examples can be characterized by only a handful of cost contributors (in some cases, only one may suffice).

Database Profiling

Once a selected set of cost contributors is determined, a simple way to determine the expected value of these quantities must be found. This is usually done by using a database profile rather than the exact values in the database. Traditional statistical profiles are specified by means of four categories of quantitative descriptors [Mannino88]: (1) descriptors of central tendency; (2) descriptors of dispersion; (3) descriptors of size; and (4) descriptors of frequency distribution. Usually, the more precise the descriptors, the more accurate the predictions. There are many widely-used "standard" descriptors: mode, mean, median; variance, standard deviation; cardinality of the relations; normality, uniformity, to mention only a few. Many real-life databases can be characterized by these common descriptors with the advantage of a simpler, more general cost analysis, normally at the expense of some loss in accuracy. In fact, many frequency distributions have been extensively studied in the area of statistics [Mannino88].†

†Given an arbitrary database, it is not always easy to establish which "standard" set of descriptors approximates the data best. Sets of tests have been developed for some of the most popular approximation functions in the literature.

However, derived relations and complex queries do not deal with simple distribution functions, but rather with combinations (specifically, joins, semijoins, selections and projections) of distributions that require a more complex analysis. Most of the research work[†] has been devoted to just a few distribution functions (uniform, Pearson, normal and Zipf) and not all basic database operators have been studied with the same degree of depth or success. A substantial part of the work has concentrated on the estimation of the number of output tuples to the query[‡]. Given these deficiencies, it is not unusual that query optimizers automatically assume a distribution function that is simple and well understood (typically the uniform distribution). An additional problem occurs when the actual distribution function is not known (databases are constantly changing and it is not always possible to keep track of the changes in the shape of the distribution) or only known in a non-parametric form (usually histograms). Our model will normally assume a uniform distribution of attribute values in compliance with the standard trend.

Given a certain degree of instantiation of the arguments of a GraphLog subgoal, our claim is that it is feasible to estimate an expected value for the selected set of cost contributors. As it is always the case with abstract interpretation techniques [Cousot77], [Cousot92], the more information we have about the subgoal, the more accurate the estimates can be.

For the case of extensional database predicates, in our model, such an estimate is obtained by simple statistical considerations[††]. In the ideal case, if we know the exact values of the database tuples as well as the exact subgoal (query retrieval) under consideration, the expected value of a cost contributor can be calculated accurately. If our knowledge is more limited, we have to introduce some assumptions (as mentioned be-

---

†See [Mannino88] for a thorough (although slightly out-of-date) survey on the topic.

‡After all, in traditional database query planning, the sizes of intermediate relations are usually regarded as important (if not the most important) contributors to the total execution cost of a query.

††The estimation of a simple fact retrieval (i.e., direct extensional database searches) is mostly a statistical problem since the distribution followed by its arguments is assumed to be known in advance or can be somehow determined.

fore, we will normally assume a uniform distribution of independent attribute values), yet still achieving acceptable results.

For the case of intensional database predicates, the estimation of the expected value of a cost contributor requires a more elaborate process, which we proceed to sketch.

Cost of a General Query

Given a query whose cost we wish to estimate, we propose to decompose the query into simpler components. To simplify the problem, we assume that queries are independent of each other†. The simplest choice consists of defining a GraphLog *subgoal* as the primitive entity to be analyzed. A subgoal is then treated as a "black box": given some inputs (such as degree of instantiation of the arguments, number of times that the subgoal is expected to be invoked, average number of solutions that are expected to be returned by the subgoal, etc.), the expected values of the cost contributors may be estimated (as the outputs of the black box) and used by successive blocks as their respective inputs. The subgoal itself has to provide some information about internal characteristics such as distribution of attribute values or correlation amongst arguments (see Figure 1.4 as an example of this idea. Note that *average* values are obtained, since the actual values of the ground terms are not taken into consideration: a uniform distribution of attribute values is assumed instead).

The total cost of the query is then estimated as the sum of the individual costs of the subgoals. Again, standard abstract interpretation techniques are used to determine the degree of instantiation of the arguments and *propagate* the intermediate results through all successive query components. This instantiation information may also be used to reject unsafe orderings [cf. Section 2.1.3].

The estimation of a general predicate call can be obtained as the *sum* of the costs associated with each individual rule (Figure 1.5). This holds largely true as long as rules are independent of each other (i.e., they do not have common solutions). However, it is quite common that two or more rules provide common solutions. A mutual exclusion

†We will see that a more complex framework is required to deal with dependencies amongst components.

```
nation(canada).
nation(belgium).
nation(uk).
language(canada, french).
language(canada, english).
language(belgium, dutch).
language(belgium, french).
language(belgium, german).
language(uk, english).
```

query:

```
german_speaking_nation(N) :- nation(N), language(N, german).
```

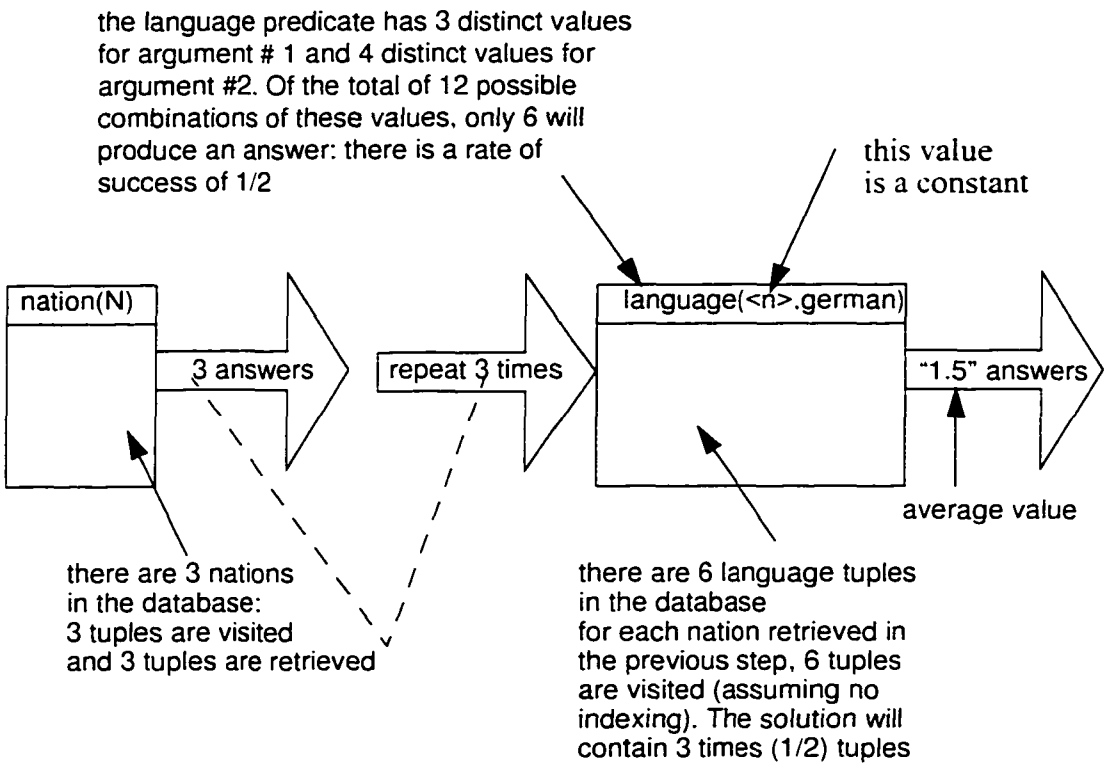


Figure 1.4 A query as a series of successive operations

analysis may help, but the general problem of duplication resulting from independent rules seems to be difficult to solve. Our cost model does not take this source of duplication of tuples into account.†

†We must distinguish between the cost of finding *all answers* (i.e., the sum of the costs of the individual rules) and the cost of finding all *distinct solutions* (whose estimation has to take into account the process of elimination of duplicates).

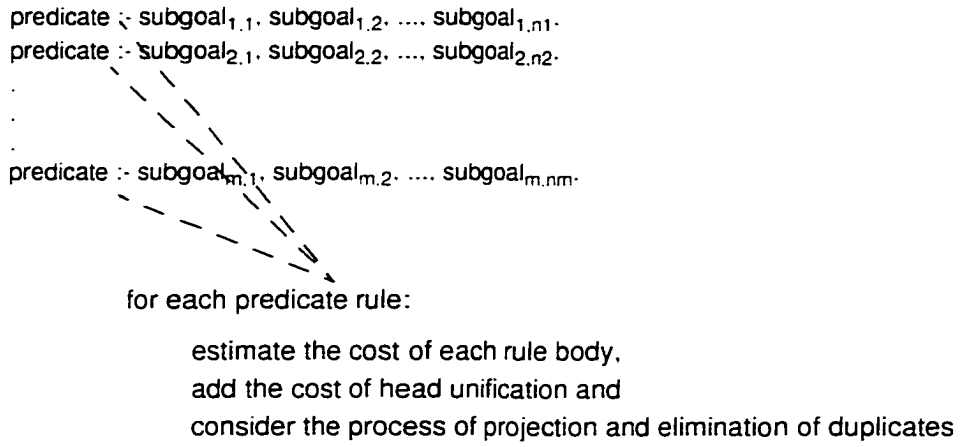


Figure 1.5 The cost of a general predicate is the sum of the cost of its individual rules

When we are dealing with general predicate calls, we have to consider some additional issues, such as (a) head unification, (b) clause indexing, (c) independence of subgoals and (d) the fact that the distribution of the tuples may be difficult to predict. Head unification and clause indexing are implementation-specific issues and they are taken into account in our model by assigning to each rule in the predicate a probability of success, (usually) given the degree of instantiation of the arguments involved. Each rule is then weighted based on this probability factor.

In some instances, the output of a subgoal is affected by the nature of other subgoals. Consider, for instance, a sequence of subgoals p(X, T), q(T, Y), and suppose that the set of values that the first subgoal derives for variable T are such that they do not form part of the domain for the first argument in predicate q. Unless we keep track of all intermediate values for variable T (which is normally contrary to abstract interpretation principles), we have no easy way to determine that predicate q will fail for all its inputs. By the same token, since we will not know the exact values of the variables involved, we have no direct method to estimate the shape of the distribution of attribute values for general predicates. In our cost model, we will ignore the issues of independence of subgoals and distribution for intermediate results.

Once the determination of the outputs of the subgoals has been solved (that is, the equivalent of the selection operation of relational algebra), we need to couple different black boxes (i.e., tackle the analogue of the join and projection operations of relational algebra). Several hurdles arise at this point, but the two most problematic are the duplication of solutions after a projection of arguments (noted before) and the correlation between the arguments of two or more different subgoals (interdependence amongst subgoals). Our model in its present form does not tackle these issues.

Our model also handles *recursive* queries which, in the specific case of GraphLog, are in the form of a predicate closure. Specifically, our methodology estimates the expected average number of solutions of a recursive predicate. The basic idea is that any linearly recursive query can be expressed as a transitive closure (possibly preceded and followed by some non-recursive predicates) [Jagadish87]. Therefore, we estimate the number of solutions of the recursive predicate by estimating the number of solutions of an equivalent query expressed in terms of transitive closure. Thus, we propose a method to estimate the average number of solutions of a transitive closure. An entire chapter will be devoted to explain how our framework deals with recursive queries.

Other issues not currently considered by our cost model include (a) aliasing or sharing of a common variable within the same subgoal, (b) consideration of invalid inputs, and (c) more complex forms of recursion.

As we will see in a subsequent chapter, more accurate results may be achieved when the methodology is tailored to the specific abstract machine and the particular characteristics of the system used to execute the queries. If we wish to obtain more accurate results, we would also require specific knowledge of the evaluation methods that are used (which is crucial when dealing with recursive queries) and the special optimization techniques that are implemented. Note that, under this scheme, a new analysis would be required for each different system. As can be seen, this process may become quite tedious. An alternative, more general solution would require making rough assumptions and concentrating on more "high-level" cost contributors. Thus, given a general evaluation strategy (*top-down* evaluation in our case [cf. Section 2.1.2]), we are able to estimate the cost

of a given GraphLog query without specific knowledge of the particular abstract machine that is being used by the GraphLog system under consideration. Our framework addresses both approaches, so that we propose a model tailored to a specific machine, the WAM [Aït91], as well as a model based on more "high-level" cost contributors and relatively independent of the underlying abstract machine (Figure 1.6).

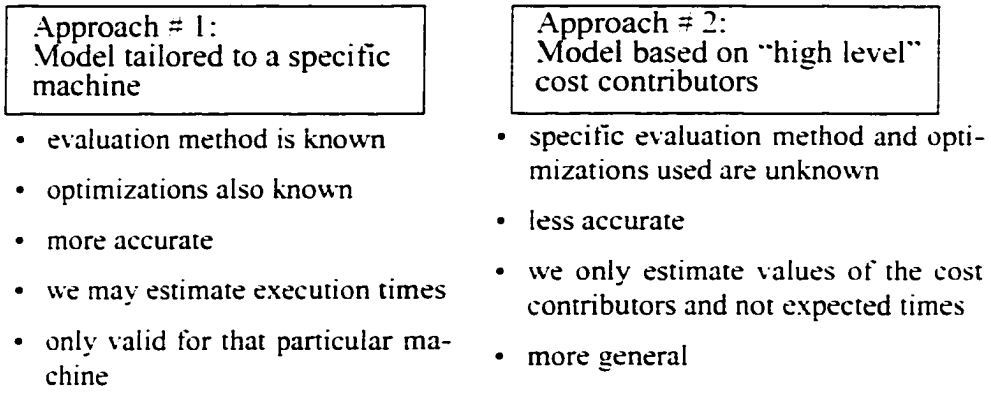


Figure 1.6 Two general alternatives for a cost model framework

Chapter 2. Cost Modeling

A cost model may be visualized as an abstraction that attempts to estimate the *efficiency* of the actual execution of some piece of code (in our case, a GraphLog query). Different parameters may be used to measure the degree of efficiency. The most commonly used metrics are the *time* or *memory* that are required to answer the entire query. It can be argued that, as memory continues to become cheaper, emphasis should be given to estimating time efficiency rather than memory efficiency.

Different orderings of the same group of subgoals in a GraphLog query will usually result in a different degree of *efficiency* of execution. Such a difference is due to many factors, ranging from some that are rather predictable (such as the size and nature of the machine code that is generated, or the series of systematic code optimization techniques that are performed) to those that are shaped by the current environment in which the program is executed (such as current system load, or the number of processes competing for common resources). The latter considerations are hard to take into account and are normally ignored.

In this chapter, we start with an overview of some issues related to query reordering in Datalog (which also apply to GraphLog). We also give a brief account of some related work in the area of query reordering.

2.1 Evaluation Methods for Datalog

Given a Datalog program, a computational model that derives all the facts satisfying the user's query is required. Normally, the chosen evaluation method computes solutions according to the so-called *least fixpoint* model [Ceri91].

Although pure Logic Programming does not include *built-in* predicates such as arithmetic or comparison operators, most implementations permit the use of such predicates. An additional useful construct not available in pure Datalog is the use of *negation*.

Negation is often handled by using the *closed world assumption*, a mechanism of negation as failure that states that the negation of a fact that cannot be logically derived from the Datalog program is considered to be valid.

Several evaluation methods have been proposed for solving Datalog queries, i.e., determining whether a user's query is valid given the collection of rules and facts that are formulated in the program. We can categorize these methods into two major groups according to the general evaluation strategy, namely bottom-up and top-down evaluations [Ceri91].

2.1.1 Bottom-up Evaluation

Bottom-up evaluation methods apply the principle of matching rules (usually called *intensional database predicates*) against the facts (also called *extensional database predicates*) to obtain valid values for the variables involved in the corresponding rules. Those rules whose head variables acquire ground values are then considered in a similar manner to extensional database predicates, and the process is repeated until all necessary facts have been derived. Most bottom-up evaluation methods have been borrowed or adapted from well-known algorithms originally developed to solve systems of equations in Numerical Analysis (for example, the Jacobi algorithm for finding least fixpoints). Most extensions of the basic algorithms are aimed at avoiding duplication in the evaluation of intermediate solutions. Bottom-up evaluation is the natural method for set-oriented languages like Datalog.

2.1.2 Top-down Evaluation

Top-down evaluation methods use the principle of *unification* between a given subgoal and the intensional or extensional database predicates. This process of unification provides a set of valid bindings that then are propagated to the other subgoals that constitute the query. A so-called *derivation tree* is generated. A fairly well-known method that is based on this resolution principle is the SLD-resolution procedure and its several extensions (which constitute the evaluation method of choice for the language Prolog). Top-down evaluation is well-suited for solving simple transitive closure problems when the

extensional database relation has no cycles, or when just one answer to the query is needed.

In one of the current implementations due to Fukar [Fukar91], the query language GraphLog is translated into Prolog. Thus, the GraphLog database can be viewed as a Prolog database, and the executable program as a Prolog program. As a result, under this particular implementation, GraphLog is evaluated using a top-down strategy. For this very reason, all cost models that we propose in this dissertation are tailored to a top-down evaluation strategy.

2.1.3 Safety Considerations

Safety is an important issue related to the evaluation strategy that is chosen. Generally speaking, a query is safe to evaluate if it has a finite number of answers and the computation that is performed to find them terminates, i.e., all the answers are obtained after a finite number of computations. For this reason, query safety plays a very important rôle when a plan of execution is selected. The issue of the safety of rules has been extensively studied in the literature and safety conditions have been derived for different logic programming languages, and Datalog is not an exception [Bancilhon86].

2.1.4 Query Reordering in Datalog

In pure logic programming, both rules and subgoals can be reordered at will without changing the meaning of the program. In practice, some orderings may yield more efficient executions of the program. However, we have already seen that some orderings may lead to non-terminating computations.

A distinction exists between inherently non-terminating queries and queries whose computation does not terminate for just some orderings. In this latter case, the reordering algorithm must reject such unsafe orderings.

The two principal causes of non-terminating computations for otherwise safe queries are:

- *Evaluable* predicates, i.e., predicates that require that some of their arguments have a ground value prior to the predicate invocation. This is a consequence of the fact that built-in predicates usually deal with *infinite* relations. In general, if the predicate arguments do not have ground values before the call, the evaluable predicate will produce an infinite number of answers. Typical examples of evaluable predicates are arithmetic expressions and comparison operators. For instance, consider the evaluable predicate *plus(X, Y, Z)* which represents the arithmetic expression $X + Y = Z$. This predicate is unsafe if two or more arguments are not integer constants. Thus, a query such as *:- plus(5, Y, Z)* would yield an infinite number of answers.

- *Negation*, which is normally handled under the so-called *Closed World Assumption*, considers anything that cannot be logically derived from the rules and facts to be false. The Datalog fixpoint evaluation procedure handles negation by computing the complement of the relation that is being negated. If the domain of such a relation happens to be infinite, the complement may be infinite too. For this reason, the negation of a predicate with at least one variable argument is a potential source for an infinite computation.

Safety rules for GraphLog have been formulated by Fukar [Fukar91]. It is shown that, when GraphLog is translated into Prolog, safety is achieved when the following order for the subgoals is observed: (1) positive (i.e., non-negated) database predicates first; (2) evaluable predicates next; and (3) negated predicates last. However, this specification is harshly restrictive, since evaluable predicates and negations of predicates are only unsafe under certain circumstances.

A less limiting condition restricts evaluable and negated predicates to positions where they are guaranteed to be safe. For the case of evaluable predicates, we have to define a *set* of lists of arguments that are required to be ground in order to be safe (i.e., yield a finite number of answers). Figure 2.1 shows two examples of such sets of lists.

In the case of negation of predicates, we must guarantee that all arguments become ground prior to the evaluation of the predicate.

% built-in predicate >	% built-in predicate –
% >(A,B) :- true if A is *greater than* B	% –(A,B,C) :- true if C = A *minus* B.
% A, B: integer values	% A, B, C: integer values
This evaluable predicate is safe when both arguments are ground; otherwise it is not safe.	This evaluable predicate is safe whenever two or more arguments are ground; not safe otherwise.
Set of lists of ground arguments that guarantees safety:	Set of lists of required ground arguments that guarantees safety:
{ [A,B] }	{ [A,B], [A,C], [B,C], [A,B,C] }

Figure 2.1 Sets of lists of arguments for two evaluable predicates that ensure safety

2.2 Some Recent Work on Query Reordering

Several cost models for logic programming languages have been proposed in the past. McCarthy [McCarthy82] proposed the use of graph-colouring algorithms to mimic the evaluation process of a conjunction of literals. Gooley and Wah [Gooley89] suggested a heuristic method for reordering Prolog clauses using Markov chains and probabilities for success and failure. McEnery and Nikolopoulos [McEnery90] described a reordering system that rearranges non-recursive Prolog clauses by applying both static and dynamic reorderings: the dynamic reordering uses statistical information from previous executions. Sheridan [Sheridan91] designed a "bound-is-easier" heuristic algorithm for reordering conjunctions of literals by selecting subgoals containing ground arguments to be placed before other subgoals. Wang, Yoo and Cheatham [Wang93] developed a heuristic reordering system for C-Prolog based on the probability of success or failure as estimated by a statistical profiler. Finally, Debray and Lin [Debray93] developed a method for cost analysis of Prolog programs based on knowledge about "size" relationships between arguments of predicates, this being specially aimed to handle recursion (although some common cases of recursion, such as transitive closure and chain recursion, are not solved at all).

2.2.1 Efficient Reordering of Prolog Programs by Using Markov Chains

Gooley and Wah's work [Gooley89] has proposed a model that approximates the evaluation strategy of Prolog programs by means of a Markov process. The cost is measured as the number of predicate calls or unifications that take place. The method needs to know in advance the probability of success and the cost of execution of each predicate.

Gooley and Wah's reordering method takes into account the fact that different levels of instantiation (*modes*) for the arguments in the subgoals lead to different values of probabilities and costs. A Markov chain is proposed for *each* valid calling mode. The values of costs and the probabilities of success are to be provided by the user (at least in the case of the base predicates). To avoid exploring all permutations of the subgoals, Gooley and Wah propose the use of a best-first search.

The method also considers that there are some orderings that must be rejected because of safety conditions. However, no practical solution is given for recursive predicates. The results for the simple Prolog programs that are presented have some acceptable ratios of improvement, although the method seems to be quite expensive to implement. Appendix A1.1 gives a more detailed view of this method.

2.2.2 A Meta-Interpreter for Prolog Query Optimization

McEnery and Nikolopoulos [McEnery90] describe a meta-interpreter for Prolog which reorders clauses and predicates. It has two components: (a) a static component in charge of rearranging the clauses "a priori", and (b) a dynamic component that reorders the clauses according to probabilistic profiles built from previously answered queries.

This method's static reordering phase consists of rearranging the clauses that define a predicate in such a way that the most successful clauses are tried first, and the subgoals within a clause are reordered in descending order of success likelihood.

Subgoal reordering is performed by using a generalization of a heuristic due to D.H.D. Warren [Warren81]. Warren proposed a formula for the cost c of a simple query q as given by $c_q = s/a$, where s is the size in tuples (i.e., the number of solutions) of the subgoal, and a is the product of the sizes of the domains of each instantiated argument.

The generalized formula proposed by McEnery and Nikolopoulos is given by:

$$c = \frac{s}{a \times p}$$

where s and a are defined as in Warren's formula, and p is the probability of success of the clause under analysis.

The method does not handle recursive queries and it explores all permutations of possible reorderings, which may be very expensive for large queries. For a more in-depth view of this method, the reader is referred to Appendix A1.2.

2.2.3 Efficient Reordering of C-Prolog

Wang, Yoo and Cheatham [Wang93] have implemented a reordering mechanism for Prolog programs which assumes that the cost of evaluating a subgoal is a constant that can be estimated by means of cumulative statistics. A profiler collects the number of subgoals that are invoked for a given predicate p, as well as the number of times that the call fails. The *average* value of these metrics over the total number of calls to predicate p is used as a measure of the cost of evaluating such a predicate.

The probabilities of success and failure collected during statistical profiling are then used to determine a suitable ordering. In fact, the system only accumulates the number of calls to a predicate and the number of times a failure occurs. The probability of failure of a conjunction of subgoals, $s_1, \ldots, s_n$, is then calculated as the product of the individual probabilities of failure of the subgoals.

$$\text{failure rate}_{\text{subgoal}} = \frac{\text{number of failures}}{\text{number of calls}}$$

$$\text{probability of failure}_{\text{conjunction}} = \prod_{i=1}^{n} \text{failure rate}_i$$

An evident advantage of this method is that handling recursion is not a major problem, since we are only interested in the number of calls and failures, without paying attention to whether the calls are recursive or not. An obvious disadvantage of the method

is that the degree of instantiation of the subgoals is totally ignored, and, therefore, there is no distinction between different calling *modes* of the same predicate, and these usually yield different execution costs. Another drawback is that safety conditions are not incorporated and it is the responsibility of the user to inform the system about which predicates are not suitable for reordering.

2.2.4 On Reordering Conjunctions of Literals; A Simple, Fast Algorithm

Sheridan [Sheridan91] has formulated a good heuristic algorithm for reordering conjunctions of subgoals in Prolog programs. This method differs from many others in that it does not require profile information of the underlying database. Although the method is simple, it yields surprisingly good results. The method exploits the notion of "ground is better", i.e., the fact that the more instantiated the arguments in a subgoal are, the less expensive its execution is. The goal of the method is to maximize the so-called *sideways information passing* [Ullman85] from left to right.

Sheridan's algorithm distinguishes three groups of subgoals: (a) positive built-in literals, (b) negative literals and (c) other positive literals. This classification of the subgoals has to do with safety considerations. For instance, a built-in predicate may require that some of its arguments have instantiated values before the predicate call (an *enabling list* of arguments). For example, consider the predicate *sum(A, B, C)* that evaluates the operation A = B + C. Typical enabling lists (i.e., lists of arguments that guarantee that the given predicate is immediately evaluable) for this arithmetic predicate are: [A, B, C], [A, C], [A, B], and [B, C]. In other words, the predicate is safe whenever two or three of the arguments are instantiated to an integer value. By the same token, a negative literal is safe if all its arguments are constant values. For example, given non-built-in predicates *q* and *p*, the following orderings are safe ones.

```
q(X), sum(X, 3, Z)
q(X), ~p(X, 3)
```

whereas these are not:

```
sum(X, 3, Z), q(X)

~p(X, 3), q(X).
```

Note that the algorithm exploits the property of Datalog-like programs where each argument is guaranteed to have a constant value after *any* call. Thus, given this specific property, any occurrence of a variable other than the first one is guaranteed to have a constant value.

The algorithm nondeterministically selects subgoals according to the following criteria (in descending order of priority):

1. non-negative non-built-in subgoals with at least one ground argument (either an explicit constant or a variable that is known to be instantiated to a constant value by virtue of having appeared in a previously selected subgoal);
2. non-negative built-in subgoals that are safe (i.e., at least one of its enabling lists is entirely composed of ground arguments);
3. negative subgoals that are safe (i.e., all its arguments are ground);
4. non-negative non-built-in subgoals with no ground arguments.

The algorithm can use an additional heuristic rule which gives preference to subgoals with a larger number of bound arguments within each criteria group.

An important feature of this algorithm is that no knowledge of the underlying database is required. The main advantage of this fact is that there is no requirement for a database profile to be obtained, and this may represent a substantial saving. An obvious restriction of Sheridan's algorithm is that no distinction is made between a predicate that retrieves a huge number of tuples and one that is associated with a very small set of tuples, and, as a result, the expensive predicate may be given priority over a possibly better choice.

2.2.5 Cost Analysis of Logic Programs

Debray and Lin [Debray93] have proposed a more general framework to analyze the cost of logic programs, including simple forms of recursion. In particular, the method estimates the number of solutions of a logic program based on the *sizes* of the predicate arguments. The method derives size relationships amongst predicate arguments. This size information is then used to compute the number of solutions generated by each predicate.

The methodology is applicable to all non-recursive predicates and to those recursive predicates with the property of having an argument whose size is reduced at every recursive step, until a base-case value is obtained. Unfortunately, this leaves out some interesting cases of recursion (such as transitive closure or chain recursion). This method is described in more detail in Appendix A1.3.

Chapter 3. A Machine-Dependent Cost Model

We now proceed to study our cost model for a specific abstract machine. Since the current version of the GraphLog interpreter generates Prolog code [Fukar91], our analysis will be focused on this particular target language. Furthermore, we have chosen a particular execution model for Prolog, namely the WAM abstract machine [Aït91], because it is widely used for the Prolog language. (Prolog is the most widely used logic programming language.)

When dealing with databases, it is usual to separate logic predicates into two categories: *extensional* database predicates, which comprise a finite set of positive ground facts, and *intensional* database predicates, which include all other predicates. We will devote the initial part of this chapter to deriving a framework for extensional database predicates, and tackle the case of intensional database predicates thereafter.

3.1 Cost model, Initial Assumptions

We start from two assumptions. First, we suppose that some parametric values of the database are known in advance (such as the number of distinct values for every argument position for all database facts, a model for the distribution that is followed by these attribute values, etc.). Furthermore, we assume that the model of Prolog's execution closely follows the design of the Warren Abstract Machine (WAM) model [Aït91].

We normally consider three different costs that can be estimated for a given subgoal: (a) the cost of retrieving *all* solutions to the subgoal; (b) the cost of finding the *first answer* to the subgoal; and (c) the cost of obtaining the *next* valid answer for a given state. We will concentrate on the all-solutions case, since this is the usual scenario for standard database queries.

3.2 Fact Retrieval, All Solutions

The simplest possible Prolog subgoal is one that only retrieves facts from the extensional database. In this section we find the cost associated with finding all solutions to the subgoal.

Consider a subgoal p of arity n of the form:

$$p(P_1, P_2, \ldots, P_n)$$

where $P_1, P_2, \ldots, P_n$ are the arguments to the subgoal. The evaluation of this subgoal may require the execution of a specific set of WAM instructions, such as: predicate calls, allocation and deallocation of stack frames, unification operations, attempt to examine the different unifiable clauses, variable unwinding (in case of unification failure and backtracking), etc. One straightforward way of estimating the cost of evaluating the subgoal is to deduce the exact sequence of machine instructions that is executed. If we know the costs of the individual WAM instructions, a total cost for the fact retrieval operation may be calculated.†

For instance, the WAM defines several term manipulation instructions to handle unification. Their behaviour depends on the mode set by a *get_structure* instruction. If *read* mode is set, the unification algorithm is applied to both the instruction operand and the current heap cell (the WAM stores new terms onto a memory area called the *heap*). If, instead, *write* mode is specified, a new cell is allocated on the heap. A typical translation of a fact is shown in Figure 3.1 (the WAM instructions are shown to the left).

Note that the number and nature of the arguments will determine the set of instructions that corresponds to the WAM translation. And the existence of two modes (read and write) has to be considered as well.

However, a simpler approach can be proposed instead. We can neglect or disregard those instructions that either are executed regardless of the position of the subgoal in a

---

†See [Gorlick87] for an attempt to use this approach. The proposed model only considers very simple clauses without disjunctions (therefore leaving out clause indexing), and does not address the issue of the degree of instantiation (or "modes") of the predicate arguments either.

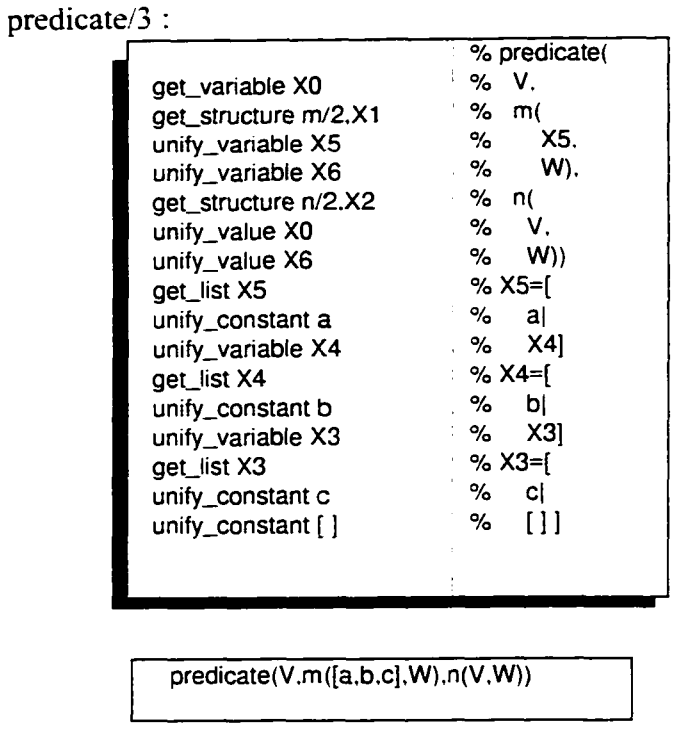


Figure 3.1 Partial translation of a fact

conjunctive clause (as in the case of the predicate call) or that do not incur a significant cost (such as, for example, WAM's *switch* instructions which support argument indexing). This latter group of instructions can be safely neglected when we are dealing with fairly large databases, when other operations (variable unwinding, tuple visiting) dominate the execution performance.

We have found experimentally that three groups of WAM instructions are usually responsible for the major part of the time spent evaluating a subgoal. These are: (a) instructions that are used to manipulate choice points (*try_me_else*, *retry_me_else* and *trust_me*); (b) instructions that perform the unification algorithm for terms; and (c) instructions that restore a previous state when a new solution is required (since a process of backtracking is launched). Our general cost function is based on these observations.

For ease of analysis, we will usually assume a uniform distribution of independent attribute values, a commonly used assumption in the database field [Mannino88].

3.2.1 Choice Point Manipulation

The first group of WAM instructions that are heavily used during fact retrieval is concerned with physical access to the tuples. In our model, we propose to write the cost due to choice point traversal as:

$$cost\_traversal = n_{chp} \times T_{chp}$$

where

- n_{chp} is the total of number of choice points that are "visited", and
- T_{chp} is the expected cost of executing the instructions associated with a single choice point.

T_{chp} is assumed to be a constant that depends on the Prolog system in use, and its value may be determined experimentally. The number of choice points, i.e., the number of alternatives that must be explored during an all-solutions retrieval can be estimated from the database profile. Given the instantiations of the arguments and the scheme of clause indexing that is used, we may estimate the number of tuples whose unification will be attempted. Appendix 1 gives a formula that holds when a uniform distribution of independent attribute values is being used.

3.2.2 Unification Operations

This second group of WAM instructions is concerned with unification applied to terms (*get_constant* instructions in our case, since we are considering simple facts). To simplify our analysis, let us consider the two simplest cases of term unification: *ground* (constant) and *not ground* (variable) unifications[†]. A variable unification is always guaranteed to succeed, whereas a ground unification can fail. In our model, we can estimate the cost associated with the unification operation as follows:

$$cost\_unification = n_{scu} \times T_{scu} + n_{ucu} \times T_{ucu} + n_{vu} \times T_{vu}$$

[†]In fact, constants and variables are the only two terms that are allowed in GraphLog.

where

- n_{scu} is the number of successful constant unifications that take place;
- n_{ucu} is the number of unsuccessful constant unifications that take place;
- n_{vu} is the number of (successful) variable unifications that take place;
- T_{scu} is the expected cost of performing one successful constant unification;
- T_{ucu} is the expected cost of performing one unsuccessful constant unification; and
- T_{vu} is the expected cost of performing one (successful) variable unification.

The three numbers can be derived from the database profile (the instantiation of the arguments and the distribution of attribute values may be used for this purpose); the three cost factors may be determined experimentally.

Consider again a subgoal p of arity n of the form:

$$p(P_1, P_2, \ldots, P_n)$$

To estimate the value of n_{scu}, n_{ucu} and n_{vu}, two quantities have to be determined for every argument position: (a) the number of unification attempts and (b) the number of successful unifications. Clearly the number of unification attempts that take place for position k has exactly the same value as the number of successful unifications that occurred for position k-1 (k>1), assuming that arguments are unified from left to right.

The number of *successful* unifications at a given argument position is a fraction of the total number of unification attempts that are made. We propose the following formula for $n_{sunif}(k)$, the number of successful unifications at position k:

$$n_{sunif}(k) = \frac{n_{unif\_att}(k)}{K(k)}$$

where

- $K(k)$ is a reduction factor for argument position k, which also represents the savings due to clause indexing (if implemented); and
- $n_{unif\_att}(k)$ is the number of unification attempts at argument position k.

Additionally,

$$n_{unif\_att}(k) = n_{sunif}(k-1), \quad k > 1$$

$$n_{unif\_att}(1) = n_{chp}$$

Appendix 1 shows some formulae that apply to the special case of a uniform distribution of independent attribute values.

3.2.3 Backtracking

The contribution to the cost of retrieving all solutions to a fact depends for the most part on the total number of solutions that can be retrieved, and is represented by operations that restore prior states during backtracking that are not part of the "choice point manipulation" previously considered†. We propose the following formula:

$$cost\_backtracking = n\_sol \times T_{back}$$

where

- T_{back} is the expected time associated with the process of restoring a previous state when a new solution is searched, and can be determined experimentally; and
- *n_sol* is the expected number of solutions to the query.

Consider a subgoal p of arity n of the form:

$$p(P_1, P_2, \ldots, P_n)$$

For a uniform distribution of attribute values, the total number of solutions is given by:

$$n\_sol = n_{sunif}(n)$$

$n_{sunif}(n)$ being the number of successful unifications that occur for the last argument P_n (Section 3.2.2).

---

†Another action that is directly related to the number of solutions has to do with the actual display of the results.

In general, the total number of solutions may be derived from the database profile. Much work has been published on this subject [Mannino88].

3.2.4 General Formula

A global formula simply takes all the above considerations into account. Given that we have decided to restrict our scope to the three previously mentioned cost contributors, our final formula is as follows:

$$total\_cost = cost\_traversal + cost\_unification + cost\_backtracking$$

$$= n_{chp} \times T_{chp} + n_{scu} \times T_{scu} + n_{ucu} \times T_{ucu} + n_{vu} \times T_{vu} + n\_sol \times T_{back}$$

3.3 Experimental Values for the Elementary Constants

Here we explain how to obtain empirical values for the constants T_{chp}, T_{vu}, T_{scu}, T_{ucu} and T_{back}, for the particular case of a uniform distribution of attribute values. It must be emphasized that these values are heavily dependent on the actual implementation that is used.

A good strategy to determine the values of the above-mentioned constants consists of building several perfectly uniform databases, and then measuring the execution time for different types of queries involving both ground and variable arguments. A ternary predicate seems to be a convenient choice because it contains most important variants without having to deal with huge databases (Figure 3.2). A binary predicate may work as well, but less accurate results can be expected.

```
pred1(ba,ba,ba).
pred1(ba,ba,aa).
pred1(ba,aa,ba).
pred1(ba,aa,aa).
pred1(aa,ba,ba).
pred1(aa,ba,aa).
pred1(aa,aa,ba).
pred1(aa,aa,aa).
```

(a)

```
:- pred1(ba,aa,ba).
:- pred1(ba,aa,Z).
:- pred1(ba,Y,ba).
:- pred1(ba,Y,Z).
:- pred1(X,aa,ba).
:- pred1(X,aa,Z).
:- pred1(X,Y,ba).
:- pred1(X,Y,Z).
```

(b)

Figure 3.2 (a) An extract from one of the databases that were used and (b) typical subgoals which retrieve these facts

The basic idea consists of building a kind of database for which we can theoretically predict the number of WAM instructions that get executed for our different queries (a symmetric database is a suitable choice given its predictability with regards to the number of WAM operations that are expected to be executed). Thus, we can derive theoretical formulae based upon some parametric variables for all contributors that we consider relevant. Then, we experimentally obtain the costs of executing the queries, and relate these costs to the parametric variables. In the case of a perfectly uniform database, all contributors may be expressed as functions of the sizes S_i of the argument domains and their respective products. Therefore, we may propose a general formula of the form:

$$\text{cost} = c_0 + \sum c_i \times \xi_i(S_1, S_2, \ldots, S_N)$$

$$\text{where each } \xi_i(S_1, S_2, \ldots, S_N) = \prod_{k \in P_i} S_k$$

$$\text{and each } P_i \subseteq \{1, 2, \ldots, N\}$$

where N is the number of arguments in the subgoal. For the ternary case, we have:

$$\text{cost} = c_0 + c_1 \times S_1 + c_2 \times S_2 + c_3 \times S_3 + c_4 \times S_1 \times S_2 + c_5 \times S_1 \times S_3 + c_6 \times S_2 \times S_3 + c_7 \times S_1 \times S_2 \times S_3$$

where S_n represents the domain size of argument n; and the c_i are constants that are related to the weight or influence of the corresponding term in the total cost – a zero value would mean no contribution whatsoever due to that particular term. In fact, when several independent experiments are launched, only a few constants show both measurably "large" and consistent values in repeated experiments, and these are obvious candidates to be considered significant.

All experimental results mentioned in this section were obtained on both SICStus Prolog, version 2.1, and SB-Prolog, version 3.0, executing on a SUN IPC SPARCstation. The experimental values were measured using the profiling routines provided by SICStus Prolog and SB-Prolog; all execution times are estimated, according to the implementation manuals, in "artificial" units.

Approximately 1,000 different databases were built, with sizes ranging from 10 to about 25,000 different tuples. For every database, all possible combinations of ground and variable arguments in the query were tried (see Figure 3.2). Using the least squares method for curve fitting, the value of constants c_i (i.e., the dependency of the execution times upon the parametric values of the database) were obtained. Initial experiments showed that all these dependencies were approximately linear.

Table 3.1 and Table 3.2 summarize some actual results for a complete experiment. S_k stands for the number of distinct values for argument position k. Those cells in the table containing values that are clearly distinct from zero (and may indicate that the term under consideration may contribute to the total cost) have been marked in bold font. A decision was made as to consider as few constants as possible, for instance, disregarding some values for variables S_1, S_2 and S_3 (as well as the independent term), which will normally hold smaller values than their products. Some variables may have values clearly distinct from zero after one experiment, but no consistent values from experiment to experiment; we decided to ignore these constants as well†. The eight different cases of ground and not ground combinations are abbreviated using the letters *g* (for *ground*) and *f* (for *free* variable, i.e., not ground).

At the same time, the corresponding WAM instructions and the number of times that they had been executed were calculated. We assumed the first-argument indexing characteristic of SICStus Prolog. A rough estimate of the number of times that the WAM instructions were expected to be executed is shown in Table 3.3.

The fact that for the all-ground-argument case (i.e., *ggg*) there was no clear dependence on the value of variable S_3 (constant c_3 is negligible), and for the first-not-ground-the-rest-ground case (i.e., *fgg*) no appreciable dependency on variable (S_1S_3) was observed (incidentally, the expressions in which these terms appear are highlighted by a light shading on Table 3.3), suggests that the contribution of constant unifications (i.e., T_{scu} and T_{ucu}) may be neglected. Thus, a simplified table (Table 3.4) is obtained.

†We observed that some apparently significant *negative* quantities showed no consistent values from experiment to experiment, and most of the time their values were close to zero. For instance, the value -0.14 in the first row of column c_3 in Table 3.1.

Variable	$S_1S_2S_3$	S_2S_3	S_1S_3	S_1S_2	S_3	S_2	S_1	1
Case	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
ggg	-0.00	**0.019**	**0.068**	0.000	-0.14	-0.00	0.000	-0.02
ggf	-0.00	**0.020**	0.000	0.000	**0.055**	-0.00	-0.00	-0.06
gfg	-0.00	**0.027**	0.000	0.000	-0.00	**0.049**	-0.00	-0.03
gff	-0.00	**0.082**	0.000	-0.09	-0.00	-0.70	-0.00	-0.05
fgg	**0.026**	0.002	0.003	0.000	-0.03	-0.02	**0.019**	**0.088**
fgf	**0.026**	0.003	**0.060**	0.002	-0.05	-0.03	-0.04	**0.180**
ffg	**0.030**	0.002	0.002	**0.054**	-0.03	-0.03	-0.04	**0.100**
fff	**0.083**	**0.010**	0.008	0.006	-0.07	-0.05	-0.06	**0.180**

Table 3.1 Typical Experimental Results for a Ternary Predicate for SICStus Prolog

Variable	$S_1S_2S_3$	S_2S_3	S_1S_3	S_1S_2	S_3	S_2	S_1	1
Case	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
ggg	-0.00	**0.022**	0.004	0.002	-0.02	-0.01	-0.01	0.211
ggf	-0.00	**0.023**	0.004	0.002	0.011	-0.01	-0.01	**0.138**
gfg	-0.00	**0.033**	0.011	0.008	-0.07	-0.00	-0.06	**0.515**
gff	-0.00	**0.069**	0.002	-0.00	-0.03	0.007	0.009	0.080
fgg	**0.024**	-0.00	-0.00	-0.00	-0.01	0.016	**0.057**	-0.10
fgf	**0.024**	-0.00	**0.039**	-0.00	0.021	0.015	0.022	-0.20
ffg	**0.030**	-0.00	-0.00	**0.030**	0.001	**0.032**	0.002	-0.00
fff	**0.072**	-0.00	-0.00	-0.00	0.006	-0.00	-0.00	**0.081**

Table 3.2 Typical Experimental Results for a Ternary Predicate for SB-Prolog

Thus, if we decide to consider only the remaining three constants, T_{chp} (directly related to "retry_me_else" operations), T_{vu} (associated with successful "get_variable" instructions) and T_{back} (connected to the number of solutions of the retrieval), then we proceed to establish which products of our S variables are expected to contribute to the cost of the retrieval. For instance, the product $S_2 \times S_3$ is significant for the "ggg" case, and products $S_1 \times S_2 \times S_3$ and $S_1 \times S_3$ are significant for the "fgf" case. We may build a table

Case	call predi-cate	switch on term	switch on constant	try me else	retry me else	trust me	successful get constant	success-ful get variable	unsuccess-ful get constant	total number of solutions
ggg	1	1	1	1	S_2S_3-2	1	$S_2S_3+S_3+1$	0	S_2S_3-1	1
ggf	1	1	1	1	S_2S_3-2	1	$S_2S_3+S_3$	S_3	S_2S_3-S_3	S_3
gfg	1	1	1	1	S_2S_3-2	1	$S_2S_3+S_2$	S_2S_3	S_2S_3-S_2	S_2
gff	1	1	1	1	S_2S_3-2	1	S_2S_3	$2S_2S_3$	0	S_2S_3
fgg	1	1	0	1	$S_1S_2S_3$-2	1	$S_1S_3+S_1$	$S_1S_2S_3$	$S_1S_2S_3$-S_1	S_1
fgf	1	1	0	1	$S_1S_2S_3$-2	1	S_1S_3	$S_1S_2S_3+S_1S_3$	$S_1S_2S_3$-S_1S_3	S_1S_3
ffg	1	1	0	1	$S_1S_2S_3$-2	1	S_1S_2	$2S_1S_2S_3$	$S_1S_2S_3$-S_1S_2	S_1S_2
fff	1	1	0	1	$S_1S_2S_3$-2	1	0	$3S_1S_2S_3$	0	$S_1S_2S_3$

Table 3.3 Number of times that the WAM Instructions are executed

Case	retry me else	success-ful get variable	total number of solutions
ggg	S_2S_3-2	0	1
ggf	S_2S_3-2	S_3	S_3
gfg	S_2S_3-2	S_2S_3	S_2
gff	S_2S_3-2	$2S_2S_3$	S_2S_3
fgg	$S_1S_2S_3$-2	$S_1S_2S_3$	S_1
fgf	$S_1S_2S_3$-2	$S_1S_2S_3+S_1S_3$	S_1S_3
ffg	$S_1S_2S_3$-2	$2S_1S_2S_3$	S_1S_2
fff	$S_1S_2S_3$-2	$3S_1S_2S_3$	$S_1S_2S_3$

Table 3.4 Number of times that the WAM Instructions are executed (simplified version)

showing such dependencies (Table 3.5) and then proceed to connect these theoretical values with the experimental values.

To derive the final values for our constants T_{chp}, T_{vu} and T_{back}, we must solve a system of simultaneous equations. For instance, for the SICStus Prolog single experiment of Table 3.1, we would consider the following system of approximate equations:

$$T_{chp} + T_{vu} \approx 0.026 \qquad T_{chp} \approx 0.020$$

$$T_{chp} + T_{vu} \approx 0.026 \qquad T_{chp} + T_{vu} \approx 0.027$$

$$T_{chp} + 2T_{vu} \approx 0.030 \qquad T_{chp} + 2T_{vu} + T_{back} \approx 0.082$$

$$T_{chp} + 3T_{vu} + T_{back} \approx 0.083 \qquad T_{vu} + T_{back} \approx 0.060$$

$$T_{chp} \approx 0.019 \qquad T_{vu} + T_{back} \approx 0.054$$

Case	$S_1S_2S_3$	S_2S_3	S_1S_3	S_3	S_1S_2	S_2	S_1
ggg		T_{chp}					
ggf		T_{chp}		$T_{vu} + T_{back}$			
gfg		$T_{chp} + T_{vu}$					
gff		$T_{chp} + 2T_{vu} + T_{back}$					
fgg	$T_{chp} + T_{vu}$						
fgf	$T_{chp} + T_{vu}$		$T_{vu} + T_{back}$				
ffg	$T_{chp} + 2T_{vu}$						
fff	$T_{chp} + 3T_{vu} + T_{back}$						

Table 3.5 Approximate Theoretical Values for a Ternary Predicate

Note that some equations are redundant. Sometimes the same terms are equated to slightly dissimilar values, serving to remind us that our results are only approximate. There is no unique method to solve such an overdetermined† system of equations. A simple method described in [Fröberg85] solves the system by using a maximum norm. We have computed the following approximate values for our particular environment when using our particular version of SICStus Prolog:

$$T_{chp} = 0.020.$$
$$T_{vu} = 0.007.$$
$$T_{back} = 0.048.$$

†An overdetermined (or inconsistent) system has more equations than unknowns.

Table 3.6 summarizes the deviation between the experimental values (i.e., the actual execution times in artificial units as obtained during the experiments) and the proposed theoretical values when using these values for SICStus Prolog (i.e., applying these values to the formula described in Section 3.2.4)†. The greatest discrepancies occur for the all-ground-argument case, due in part to the fact that constant unifications play a major rôle here, and this is ignored by our approximation (i.e., values for T_{ucu} and T_{scu} were not derived).

Case	average deviation between theoretical and experimental values (1,000 different databases)
ggg	11.14 %
ggf	5.40 %
gfg	6.27 %
gff	7.62 %
fgg	6.46 %
fgf	5.50 %
ffg	4.26 %
fff	3.21 %

Table 3.6 Average cost error introduced by our approximation

3.4 Conjunction of Simple Queries, All Solutions

We now proceed to study the case of a conjunction of facts. Again, we are interested in the all-solutions case. Consider a conjunction of simple queries of the form

$$p_1/a_1, p_2/a_2, \ldots, p_n/a_n$$

where the notation *p/a* means that predicate *p* has an arity *a*.

---

†Since the databases in the experiments were forced to have a uniform distribution of independent attribute values, for each database, we can easily estimate the values of n_{chp}, n_{scu}, n_{ucu}, n_{vu} and *n_sol* required by the formula.

If, at every point in the evaluation of this query, we know the instantiation of the arguments of every subgoal, we can determine the cost of evaluating each subgoal by using the formula described in Section 3.2. The following formula could be applied to estimate the global cost of finding all solutions (i.e., the total cost of evaluating a conjunction of subgoals):

$$cost\_conj\_facts =$$

$$c\_all(1) + n\_sol(1) \times (c\_all(2) + n\_sol(2) \times (\ldots + n\_sol(n\text{-}1) \times c\_all(n)\ )\ )$$

where

- *c_all(n)* is the cost associated with finding all solutions to subgoal p_n;
- *n_sol(n)* is the estimated (average-case) number of solutions to subgoal p_n;

Note that each successive subgoal will be called as many times as there are distinct solutions that the previous subgoal is able to retrieve.

Since we are dealing with subgoals that retrieve tuples from a database, we may determine in advance the actual instantiation of every argument in the conjunction. Thus, every variable that appears for the first time in a subgoal will be uninstantiated, whereas any variable that has appeared before in another subgoal will be instantiated at that point.

To find the least overall cost, all possible orders of subgoals must be considered. Table 3.7 shows a comparison between (a) the experimental costs for the book database example and (b) those costs predicted when only using the primitive constants in our formula and assuming a uniform distribution. Since the book database, like most real databases, does not follow a uniform distribution, significant differences can be observed. However, in this particular case, the uniform distribution model can still be used to predict a general trend, i.e., we may still obtain the most efficient evaluation order, but there is no guarantee that this will be the case in a general situation. The model was also tested against (c) another (artificially generated) book database which was designed to follow a strictly uniform distribution, and, not surprisingly, our theoretical values predicted the costs more accurately. All values in Table 3.7 are reported in SICStus Prolog's artificial units.

order	(a) real database, experimental costs using SICStus Prolog	(b) uniform distribution, theoretical value	(c) uniform distribution, experimental costs using SICStus Prolog	difference between (b) and (c)
book-publisher-author	41345	103305	114655	9.90 %
book-author-publisher	1635	2935	3270	10.24 %
publisher-book-author	260040	696433	713245	2.36 %
publisher-author-book	3434745	9333301	9485305	1.60 %
author-publisher-book	3438660	9244272	9398510	1.64 %
author-book-publisher	2690	4168	4430	5.91 %

Table 3.7 The book titles database

Normally, we will pay more attention to the relative cost amongst different orderings rather than to the "exact" cost values. Table 3.8 shows that we were able to predict the correct order of the costs of the different orderings.

(a) Ranking of SICStus Prolog (actual measurements)	(b) Ranking of theoretical predictions	(c) Ranking of SICStus Prolog when using a uniform database
1. book-author-publisher (1635)	1. book-author-publisher (2935)	1. book-author-publisher (3270)
2. author-book-publisher (2690)	2. author-book-publisher (4168)	2. author-book-publisher (4430)
3. book-publisher-author (41345)	3. book-publisher-author (103305)	3. book-publisher-author (114655)
4. publisher-book-author (260040)	4. publisher-book-author (696433)	4. publisher-book-author (713245)
5=[a]. publisher-author-book (3434745)	5=. publisher-author-book (9333301)	5=. publisher-author-book (9485305)
5=. author-publisher-book (3438660)	5=. author-publisher-book (9244272)	5=. author-publisher-book (9398510)

Table 3.8 Orderings ranked by their costs

a. Since these last two orderings are within 0.1% of each other (given actual measurements), a similar rank is shown

3.5 Intensional Database Predicates

As mentioned before, it is common practice to separate logic predicates into two categories: *extensional* database predicates and *intensional* database predicates. One advantage of this division is that extensional predicates typically have large numbers of clauses (they can be seen as the database itself), whereas intensional predicates normally have a small number of clauses. Additionally, one can infer some properties for extensional predicates, such as a distribution for the attribute values or correlation factors amongst them, that characterize the database under consideration. Normally, these properties are constant in time (cf. previous sections). In other words, one can predict, within certain parameters, how a query will behave when applied to that database.

On the other hand, intensional predicates are less predictable. They require a more complex analysis framework, whose predictions are normally less accurate. A standard approach for analyzing the execution behaviour of a program is to use abstract interpretation techniques [Cousot77, Cousot92], which transfer the problem to a different, easier to handle domain at the expense of some loss of precision. In the specific case of logic programs, for example, instead of keeping track of the exact values that every variable holds during program execution, one may want to consider a simpler, more general property. One such property is the *mode* of the variable, that is, its degree of instantiation [Mellish85] [Debray89]. We will not know the exact value, but at least we can ascertain that the variable under consideration is an uninstantiated variable, or a ground constant, or a term with a combination of both (i.e., a partially grounded structure). We can infer these attributes by performing a static *mode analysis*.

3.6 Mode analysis

In general, Prolog programs are undirected, that is, there is no distinction between input and output parameters for a given predicate. This notion of bi-directionality presents a major challenge to the production of efficient code, since the depth-first search strategy with chronological backtracking that Prolog uses to implement non-determinism is itself a very inefficient strategy [Mellish85]. However, Prolog predicates are typically written with one sole direction in mind and, as a result, some parameters are meant to be exclu-

sively input or output. Knowledge of such directionality can be expressed using the notion of *modes*, a concept which was introduced by D.H.D. Warren (and refined by Mellish) to classify the ways in which a Prolog predicate is used during the execution of a program. If the programmer provides such clues to help the compiler identify directionality, the generated code can be dramatically improved. A possible alternative is to infer the mode information by performing a global analysis of the program [Debray88].

The standard approach for determining the mode information of a logic program statically uses *abstract interpretation* [Cousot77], [Cousot92]. This is a general technique where the standard semantics of a program are projected onto a different (and simpler) domain. Several solutions to the problem of finding the modes of a Prolog program have been proposed. A quite extensive survey is given in the introduction of [Debray89]. In this section, the mode inference algorithm of Debray [Debray89] is described, since this framework is the basis of the determinacy analysis for our work.

3.6.1 Modes

The mode of a predicate in a Prolog program specifies which arguments are input arguments and which are output arguments, taking into account all possible calls that can occur during the execution of a program. Depending on the nature of the problem, a set of modes must be defined to characterize the modes of the arguments in a Prolog predicate. Debray proposed the family of modes $\Delta = \{$ **c**, **d**, **e**, **f**, **nv** $\}$, where **c** denotes the set of fully-instantiated (ground) terms, **d** (don't know) the universal set of all terms, **e** the empty set, **f** (free variable) the set of un-instantiated variables, and **nv** the set of non-variable terms (that is, structured terms which are not fully instantiated). The set Δ forms a complete lattice under the inclusion operator (Figure 3.3):

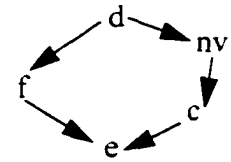


Figure 3.3 Debray's lattice for mode analysis

Given a set of terms T, its *instantiation* is defined to be the element of Δ that best characterizes it. Thus, the least upper bound [Birkhoff40] for all terms in T is chosen.

Prolog's unification operation can be understood in the mode's domain (called the *abstract* domain) as an operation that, given the instantiations of the arguments in a call, refines them according to the nature of the head arguments. Debray [Debray89] defined the lattice in a way that given two term instantiations T_1 and T_2, the unification of them is chosen to be the least upper bound of their instantiations under the following partial ordering:

$$\mathbf{f} \subseteq \mathbf{d} \subseteq \mathbf{nv} \subseteq \mathbf{c} \subseteq \mathbf{e}$$

The unification of terms is modelled by applying the *join* operation to two elements of the lattice, a and b, which returns the least upper bound of a and b. The join operator for the ordering under consideration is written as ∇.

Some examples are shown in Figure 3.4. Note that, since some information is not taken into account in the abstract domain, the results are usually less accurate than in the concrete (and more complex) world.†

3.6.2 General Mode Analysis Method

Debray's method uses the *procedural view* for Prolog, which recognizes the existence of mechanisms such as procedure call, success, failure, backtracking, etc. Debray's static inference of Prolog modes is based on keeping track of individual variable instantiations throughout the execution of a program. Such information is propagated in the usual way, from caller to callee at any predicate invocation and from callee to caller at the time of the predicate's completion. Thus, at any point during program execution, an *instantiation state* is defined, which contains instantiation information for every variable in the program.

The notion of an instantiation state can be extended to any arbitrary non-variable term. A constant term will have ground instantiation (**c**) and an empty dependency set.

---

†In particular, unification in Debray's model can never fail.

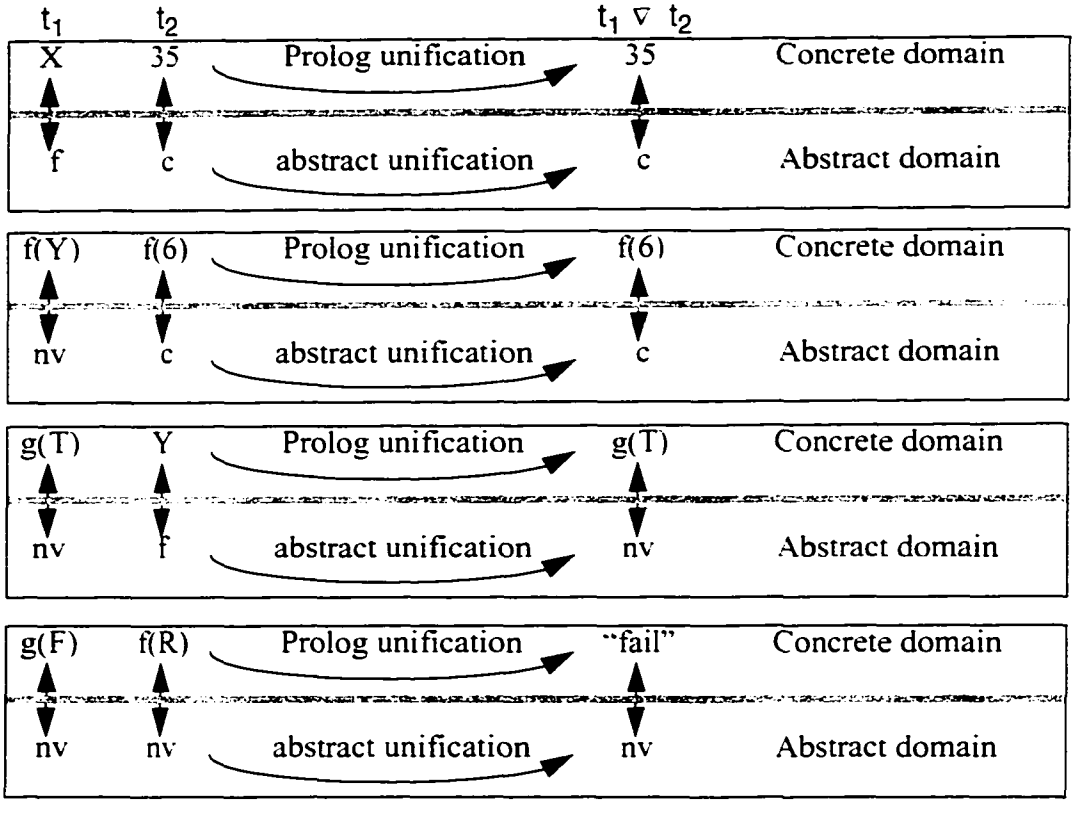


Figure 3.4 Abstract interpretation applied to Prolog unification given two terms *t1* and *t2*

A structured term will have ground instantiation (**c**) if all its arguments are ground and non-variable instantiation (**nv**) otherwise.

In order to facilitate the propagation of mode information, Debray's method defines the existence of *instantiation patterns* for every procedure call. An instantiation pattern will contain, for every procedure argument, some information related to its instantiation.

3.6.3 Abstract Domains

In essence, we can characterize every predicate call by a previous state (in which the instantiations of the arguments are grouped in a so-called *calling pattern*), and by a resulting state, which differs from the original state in the same degree as the arguments do (the new set of modes for the arguments is referred to as the *success pattern*). In the framework proposed by Debray [Debray89], we have knowledge about the degree of instantiation of every term at any execution point. Although not explicitly formulated by

Debray, we may characterize his abstract domain as pairs of <calling patterns/success patterns> for each predicate call in the program. More formally,

$$\text{Debray's Abstract domain} = \{ (Pred_r, cpat_{m_r}, spat_{m_r,n_r}) \},$$

$$1 \le m_r \le \text{ncpat}(Pred_r),$$

$$1 \le n_r \le \text{nspat}(Pred_r, cpat_{m_r}),$$

$$1 \le r \le \text{number of distinct predicates}$$

where

- $Pred_r$ represents the r-th predicate in the database;
- $cpat_n$ is a feasible calling pattern for predicate *Pred*; a calling pattern is an ordered k-tuple (where k is the number of arguments in predicate *Pred*) in which the k-th element represents the current mode for the k-th argument;
- $spat_{n,m}$ represents the m-th valid success pattern for the given calling pattern $cpat_n$; a success pattern is identical in form to a calling pattern and it differs from the calling pattern in that the modes for the arguments are updated as a result of the predicate call (by using Debray's unification rules in the abstract domain);
- *ncpat(Pred)* is the number of distinct calling patterns that *Pred* can be invoked with (as determined by a static analysis); and
- *nspat(Pred, cpat)* is the number of different success patterns that results from a call to *Pred* given a calling pattern *cpat*.

We start with Debray's approach and propose enriching the domain with probabilities of occurrence for the various success patterns, as well as quantities related to the cost of that particular execution path. We tailor the analysis to the *all-solutions* case. Our analysis will be restricted to non-recursive programs. We may easily extend it to allow recursion controlled by an argument that reduces its size at every recursive step as De-

bray does [Debray93], but this type of recursion does not occur in pure Datalog programs.

To illustrate how cost contributors are incorporated into our abstract domain, consider a variant of the books database introduced in Section 2.1.4. Consider the following order of subgoals for an arbitrary query:

```
:- book(T,P,S,A), publisher(P,C), author(A,canadian).
```

and assume that the database attributes follow a uniform distribution of values. The database profile is shown in Table 3.9.

Predicate name	number of tuples	distinct values in argument 1	distinct values in argument 2	distinct values in argument 3	distinct values in argument 4
book/4	3,000	3,000	20	10	450
publisher/2	7,600	20	380	—	—
author/2	450	450	30	—	—

Table 3.9 The books database profile

For our particular query, the *book* predicate will be invoked with all arguments unbound (i.e, a calling pattern [f, f, f, f])†. The *publisher* predicate will be called with a calling pattern [g, f], since the first argument will have a constant value after the call to predicate *book* has been completed. Finally, the *author* predicate will have both arguments bound to a constant value (i.e., a calling pattern [g, g]).

In this particular case, we have the following instances that characterize the execution of the query:

(a) *book* predicate:

In Debray's domain, the following instance is generated:

```
<book, [f, f, f, f], [g, g, g, g]>.
```

†As before, we use "g" to denote a ground terms and "f" to indicate a free variable.

where the third element of the tuple is the success pattern that results from a successful call. In our domain, we wish to include some cost contributors, namely the number of tuples that are visited, n_t, the number of variable unifications that take place, n_v, and the expected number of solutions, n_s. Thus, we would generate the following instance of the book predicate:

<book, [f, f, f, f], [g, g, g, g], 3000, 12000, 3000>,

where the three numerical values represent the cost metrics n_t, n_v, and n_s, respectively.

(b) *book* predicate:

Debray's instance for this predicate would have the form:

<book, [g, f], [g, g]>.

Our enriched domain would be:

<book, [g, f], [g, g], 380, 380, 380>.

(c) *author* predicate:

Using Debray's domain, we would obtain:

<author, [g, g], [g, g]>.

while our domain would provide additional information:

<author, [g, g], [g, g], 1, 0, 0.0333>.†

More formally, our enriched domain is defined as follows:

$$\text{Cost Abstract Domain} = \{\langle Pred_r, Clause_q, cpat_{m_r}, metrics_{m_{q_r}}\rangle\}.$$

$1 \le m_r \le \text{ncpat}(Pred_r)$.

$1 \le n_r \le \text{nspat}\left(Pred_r, cpat_{m_r}\right)$.

$1 \le r \le$ number of distinct predicates

$1 \le q_r \le$ number of distinct clauses in predicate $Pred_r$

†Note that, in this case, the rate of success is given by 450/450/30.

where

- $Pred_r$, $cpat_n$, *ncpat(Pred)* and *nspat(Pred, cpat)* are the same as before;
- $Clause_q$ identifies the q-th clause of the predicate under consideration; for extensional predicates, all clauses may be collapsed into a single tuple template, making this element irrelevant;
- $cpat_n$ is the n-th feasible calling pattern for predicate *Pred*;
- *metrics* contains a list $<v_1, v_2, \ldots v_n>$ of values for n different cost contributors that we have decided beforehand are relevant to produce an estimate of the total cost associated with calling pattern $cpat_n$; for instance, if we have decided that our cost function will be based on the number of successful unifications (say, *nsucc_unif*) we may have a list of the form *<nsucc_unif, num_sol>*, where *num_sol* is the number of solutions that result from calling predicate *Pred*. In other words, we will record all those quantities that are required by our cost formula. Thus, for the formula derived for the WAM in Section 3.2, our list of metrics would most probably be of the form $< n_{chp}, n_{vu}, n_{back}, num\_sol>$. Note that the number of solutions associated with a predicate call is normally required in order to calculate costs for conjunction of queries (see Section 3.4);

In other words, given a predicate clause, we are mainly interested in obtaining some metrics related to the cost estimation for any viable calling pattern. We have omitted in the domain a place for the success patterns that are obtained. The reason for this is that such information is implicitly used by successive predicates in their respective calling patterns.†

3.7 Cost Function

In this section, we will proceed to explain the domain element *metrics*. Its purpose is to keep track of all relevant parametric values that are used to estimate the cost of the predicate clause. Normally, it will include values such as the average number of solutions that

†Note that, in the case of GraphLog, only *one* success pattern is obtained after any predicate call.

are expected, the number of tuples that are visited, the number of variable or constant unifications that take place, the number of times that the state should be restored, and so on.

In the general case, the values of our parametric values will have different values for different calling patterns. For instance, a call of the form p(X) with calling pattern <f> (i.e., a free variable) will retrieve all facts in the database, whereas a call of the form p(c) with calling pattern <g> (i.e., a ground term) will retrieve only one fact at most. Therefore, we have to keep track of feasible *abstract paths*, i.e. information regarding the calling patterns that the subgoals may be invoked with during the evaluation of that particular clause. Given a conjunction of subgoals:

$$s_1, s_2, \ldots, s_n.$$

we define an *abstract path* as a list of tuples:

$$\textit{abstract path} = [\langle s_1, cpat_1\rangle, \ldots, \langle s_n, cpat_n\rangle]$$

where

- $cpat_n$ represents a feasible calling pattern for subgoal s_n.

Note that an abstract path is defined for any GraphLog query or rule and its unique value is determined statically via a simple mode analysis. Although no success pattern appears in the definition of an abstract path, it should be clear that successive calling patterns are built from the success patterns that are obtained from previous subgoals.

3.7.1 Cost Function from the Perspective of Head Unifications

For intensional predicates, we will estimate the evaluation costs at the clause level, that is, we will obtain the contribution to the cost due to each one of the clauses of a given predicate. In this section, we propose a methodology that can be used to estimate the average cost of evaluating a predicate given a particular calling pattern. We will concentrate on the probability associated with the process of *head unification*. In the following section we will consider how to estimate the cost of evaluating a complete query or

clause body. For this reason, we start from the assumption that the average cost of every body is available before evaluation time.

We estimate the total cost of a single clause as follows (Eq. 3.1):

$$cost(Pred_r.Clause_q|_{cpat}) = cost(hunif(Pred_r.Clause_q)|_{cpat}) + P(hunif(Pred_r.Clause_q)|_{cpat}) \times cost(body(Pred_r.Clause_q)|_{h(cpat)})$$

where

- $cost(Pred_r.Clause_q|_{cpat})$ is the cost that results from the evaluation of the q-th clause in the r-th predicate, given an initial calling pattern *cpat*;
- $cost(hunif(Pred_r.Clause_q)|_{cpat})$ is the cost due to the process of head unification for the q-th clause in the r-th predicate, given a specific calling pattern *cpat*;
- $P(hunif(Pred_r.Clause_q)|_{cpat})$ is the probability that the process of head unification for the q-th clause in the r-th predicate is successful for the given calling pattern *cpat*;
- *h(cpat)* is the modified pattern that results after a successful head unification given a calling pattern *cpat*, which in turn is the initial calling pattern for the body of the clause;
- $cost(body(Pred_r.Clause_q)|_{h(cpat)})$ is the cost that is associated with the evaluation of the body of the q-th clause in the r-th predicate given a calling pattern *h(cpat)*; this value will be analyzed in a following section.

Besides the cost of the body, there are two unknown quantities at this point: the cost due to the process of head unification and the probability that the head of the clause successfully unifies with the arguments to the call. The estimation of the number of primitive operations that take place during a successful head unification is quite straightforward: roughly speaking, one tuple is visited, only one restoration process would be necessary in case of backtracking, and the number of variable and constant unifications can easily be determined from the calling pattern and the internal structure of the head.

Additionally, given a calling pattern, we wish to determine the probability that the head of a clause is successfully unified. Since we do not know the exact values that can appear at every argument position, nor the frequency with which these values appear, we are forced to make assumptions. A rough but simple assumption would consider that all ground values follow a uniform distribution. Our universe of ground values may be defined such that it comprises exactly those values that appear in the heads of the clauses. Alternatively, we may obtain the distribution and universe of attribute values from another source. If we want to attach probability values to each clause we are forced to define or select a universe of values that, at least, includes all constant arguments that occur in the heads.

The probability that a given calling pattern successfully unifies with a clause head is estimated as follows:

$$P\left(hunif(Pred_r, Clause_q)\big|_{cpat} \right) = \prod_{k=1}^{numarg} P\left(\text{succ\_unif}(a_k)\big|_{cpat[k]} \right) \qquad \text{... (Eq 3.2)}$$

where

- $P(hunif(Pred_r, Clause_q)|_{cpat})$ is the probability that the process of head unification is successful for the q-th clause in the r-th predicate, given a calling pattern *cpat*;
- $P(succ\_unif(a_k)|_{cpat[k]})$ is the probability that an argument with instantiation $cpat[k]$ can be successfully unified with the k-th argument in the actual head;
- $cpat[k]$ is the current instantiation of the k-th argument in the calling pattern *cpat*; and
- *numarg* is the number of arguments in the head.

In general, the value of $P(succ\_unif(a_k)|_{cpat[k]})$ can be estimated as follows:

$$P\left(succ\_unif(a_k)\big|_{cpat[k]} \right) = \begin{cases} 1, \text{ if } a_k = \mathbf{f} \text{ and } cpat[k] = \mathbf{f} \\ \kappa_k, \text{ otherwise} \end{cases} \qquad \text{... (Eq 3.3)}$$

It is important to realize that real-life predicates do not necessarily have independent probabilities for their argument unifications. In other words, our proposal assumes an ideal case: that there is no correlation amongst arguments.

The value of κ_k should be determined from whatever abstraction we use to characterize the distribution function of attribute values, and in our framework represents the *average* probability that a ground or partially ground term *cpat*[*k*] can be successfully unified with the actual argument a_k. If no distribution function is known or if our abstraction does not keep track of the actual values for the constants, we may simply assume a uniform distribution of values. With this crude assumption, the probability that a ground argument can be unified with another ground argument is given by $1/C(a_k)$, where $C(a_k)$ is the cardinality of the universe of values for argument position *k*. The probability that a ground term can be unified with a non-compatible argument (for example, a structure with a different arity) is zero.

For example, consider the following predicate, *a/3*:

```
a(3,t,g).   |  a(M,w,N).
a(4,v,g).   |  a(M,x,_).
a(5,w,i).   |  a(M,P,N).
a(6,v,i).   |
```

We may decide to estimate the universes of values as {3, 4, 5, 6} for the first argument, {t, v, w, x} for the second argument, and {g, i} for the third argument. Suppose that our abstraction for these attribute values consists of the number of distinct values for each attribute.

If we assume independence amongst attributes, the total probability that a head can be unified with any calling pattern is given by the product of the individual probabilities associated with each argument (*one* in the case of variables, a fraction over the number

of distinct values for constant arguments). Thus, in our example, the probability that the first clause succeeds would be given by:

$$P(a(3,t,g)) = \begin{cases} prob\{3 \text{ unifies with argument } 1\} \times \\ prob\{t \text{ unifies with argument } 2\} \times \\ prob\{g \text{ unifies with argument } 3\} \end{cases} \quad \ldots (\text{Eq } 3.4)$$

$$= \frac{1}{4} \times \frac{1}{4} \times \frac{1}{2} = \frac{1}{32}$$

Note that this probability is the same for any call in which all three arguments are constants (i.e., all initial four rules in predicate *a/3*).

Similarly, the probability that the fifth and sixth clauses succeed is estimated as:

$$P(a(M,w,N)) = \begin{cases} prob\{M \text{ unifies with argument } 1\} \times \\ prob\{w \text{ unifies with argument } 2\} \times \\ prob\{N \text{ unifies with argument } 3\} \end{cases} \quad \ldots (\text{Eq } 3.5)$$

$$= 1 \times \frac{1}{4} \times 1 = \frac{1}{4}$$

Finally, the probability that the clause a(M,P,N) succeeds would be one. Note that, although our universe sets are arbitrary and underestimates of the true argument domains may occur, there is a notion of which clauses are more likely to succeed.

3.7.2 Cost Function from the Perspective of Body Evaluations

Once we are able to assign a probabilistic value to each head clause given a calling pattern, we then estimate the cost of evaluating the corresponding bodies. First, we derive a formula that permits estimation of the cost of evaluating a single subgoal s_n. Since predicate $Pred_r$, the predicate invoked by the subgoal, contains q different clauses, the following formula can be used to determine an average cost associated with the whole predicate:

$$\text{cost}\left(s_n \big|_{cpat_m} \right) = \sum_{k=1}^{q} \text{cost}\left(Pred_r, Clause_q \big|_{cpat_m} \right) \quad \ldots (\text{Eq } 3.6)$$

where

- $cost(Pred_r.Clause_q|_{cpat})$ is the cost that results from the evaluation of the q-th clause in the r-th predicate given an initial calling pattern $cpat$, as analyzed in the previous section.

Now, given a conjunction of subgoals:

$$s_1, s_2, \ldots, s_n.$$

the cost of the compound sequence of subgoals will be decomposed into the individual costs due to each abstract path, as follows:

$$\text{cost (sequence of subgoals)} = \sum_{k=1}^{npaths} \text{cost (} abstract\ path_k \text{)} \qquad \ldots \text{(Eq 3.7)}$$

where

- *npaths* is the number of distinct abstract paths that a given clause of a predicate can yield when a calling pattern is initially used†: and
- $cost(path_k)$ is the cost that results from the evaluation of a complete k-th abstract path. This function can be expressed as follows (Eq. 3.8):

$$\text{cost (} abstract\ path \text{)} = \text{cost}\left(s_1|_{cpat_1} \right) + nsol_1 \times \left(\text{cost}\left(s_2|_{cpat_2} \right) + nsol_2 \times \left(\ldots \times \text{cost}\left(s_n|_{cpat_n} \right) \right) \right)$$

where

$$abstract\ path = [\langle s_1, cpat_1 \rangle, \ldots, \langle s_n, cpat_n \rangle]$$

- $nsol_k$ is the average number of solutions that subgoal s_k produces when invoked with a calling pattern $cpat_k$, as recorded in the domain element *metrics*.

Example. Consider an extended version of the books database introduced in Section 1.4.1.

---

†In pure Datalog, only one abstract path is actually derived.

- book(Title, Publisher_Name, Subject, Author_Name). A collection of book titles along with their publishers, subjects of the publications and authors.
- publisher(Publisher_Name, City). A list of different cities where book publishers have an authorized distributor.
- author(Author_Name, Nationality). A group of facts that relate authors to their respective nationalities.
- skilled(Author_Name, Subject). A list of the two more prominent authors on every possible subject.
- forte(Publisher_Name, Subject). A list of the two top publishing companies for every given subject.

Suppose that we wish to retrieve an exhaustive list of tuples of the general form <Title, Publisher_Name, City, Author_Name> for those "worthwhile" publications whose author has a certain nationality.

Database profile:

(a) Extensional DB predicates. We assume that the extensional database predicates follow a strict uniform distribution of attribute values. The corresponding database profile is given in Table 3.10.

Predicate name	number of tuples	distinct values in argument 1	distinct values in argument 2	distinct values in argument 3	distinct values in argument 4
book/4	3,000	3,000	20	10	450
publisher/2	7,600	20	380	—	—
author/2	450	450	30	—	—
skilled/2	20	20	10	—	—
forte/2	20	20	10	—	—

Table 3.10 The extended books database

(b) Intensional DB predicate. Suppose that our (only) intensional database predicate is defined as follows†:

```
%worthwhile/3: worthwhile(Publisher,Author,Subject). Tells us if a book
%is worth buying
worthwhile(publisher_1,_,_).
worthwhile(publisher_5,_,_).
worthwhile(publisher_10,_,_).
worthwhile(_,author_2,_).
worthwhile(_,author_7,_).
worthwhile(_,author_13,_).
worthwhile(Publisher,_,Subject):-forte(Publisher,Subject).
worthwhile(_,Author,Subject):-skilled(Author,Subject).
```

(c) Query. We consider the following query:

```
:- book(T,P,S,A), worthwhile(P,A,S), publisher(P,C), author(A,nationality_8).
```

Table 3.11 and Table 3.12 show the results of applying our framework to this example.

<clause, cpat>	n_{chp}	n_{vu}	n_sol	**cost=** ($n_{chp}\times T_{chp}+n_{vu}\times T_{vu}+n\_sol\times T_{back}$)
<book/4, [f,f,f,f]>	3,000	4×3,000	3,000	276.000
<publisher/2, [g,f]>	7,600/20	7,600/20	380	28.120
<author/2, [g,g]>	450/450	0	0.033	0.020
<skilled/2, [g,g]>	20/450	0	0.0044	0.001
<forte/2, [g,g]>	20/20	0	0.10	0.025
<worthwhile/3, [g,g,g]>	6.194	0	0.261	0.232

Table 3.11 Predictions for all predicates

†Strictly speaking, these predicate definitions are not safe. All anonymous variables should be explicitly constrained by direct references to the *book*, *publisher* and *author* predicates. For instance, the first definition should be defined as:

```
worthwhile(publisher_1,A,S):- book(_,_,S,A).
```

However, we omit these additional predicates to keep this example simple.

<clause, cpat>	p(hunif)	n_{chp}	n_{vu}	n_sol	**cost=** ($n_{chp}\times T_{chp}+n_{vu}\times T_{vu}$ $+n\_sol\times T_{back}$)
<worhwhile/3 #1, [g,g,g]>	1/20	1/20	0	0.05	0.0224
<worthwhile/3 #2, [g,g,g]>	1/20	1/20	0	0.05	0.0224
<worthwhile/3 #3, [g,g,g]>	1/20	1/20	0	0.05	0.0224
<worthwhile/3 #4, [g,g,g]>	1/450	1	0	0.0022	0.0201
<worthwhile/3 #5, [g,g,g]>	1/450	1	0	0.0022	0.0201
<worthwhile/3 #6, [g,g,g]>	1/450	1	0	0.0022	0.0201
<worthwhile/3 #7, [g,g,g]>	1	1+20/20	0	0.10	0.0448
<worthwhile/3 #8, [g,g,g]>	1	1+20/450	0	0.0044	0.0211

Table 3.12 Predictions for the intensional database predicate

- Experimental result:

  average cost for all possible queries†: 30,834.6

- Theoretical result:

$$\text{cost} = \text{cost}(\text{book}|_{[f,f,f,f]}) + \text{n\_sol}(\text{book}|_{[f,f,f,f]}) \times$$

$$(\text{cost}(\text{worthwhile}|_{[g,g,g]}) + \text{n\_sol}(\text{worthwhile}|_{[g,g,g]}) \times$$

$$(\text{cost}(\text{publisher}|_{[g,f]}) + \text{n\_sol}(\text{publisher}|_{[g,f]}) \times \text{cost}(\text{author}|_{[g,g]}))$$

$$\text{cost} = 276.0 + 3000.0 \times (0.232 + 0.261 \times (28.120 + 380.0 \times 0.022)) = 29,535.6$$

theoretical cost: 29,535.6 (an error of 3.6%, approximately).

†The experiment was repeated for all different author nationalities (the only ground argument in the query), and the average value is reported here.

- Empirical constants used:

$T_{chp} = 0.020.$

$T_{vu} = 0.006.$

$T_{back} = 0.048.$

Appendix 4 shows another example of our methodology applied to a different Prolog system.

3.8 Overview of the Model

In this chapter we first derived a crude method to estimate the cost of evaluating GraphLog queries when applied to extensional predicates. A top-down model of computation is assumed. The method requires an empirical estimation of various constants associated with the evaluation time of primitive operations. A profile of the underlying database is also required. By using this database profile, some formulae to estimate the expected number of primitive operations that will occur during query evaluation must be derived. We have considered the case when database values are distributed uniformly and independently. Although real databases seldom conform to a uniform distribution model, we may still be able to predict which evaluation orders give the best execution times.†

Additionally, we have managed to explain why some heuristic techniques for query reordering that are based upon a "bound-is-easier" heuristic procedure work [Sheridan91]. These non-deterministic algorithms usually select subgoals containing ground arguments to be placed before other subgoals. We have observed that the occurrence of a ground argument reduces the number of primitive operations that take place with respect to the case where that argument is not bound. For example, fewer tuples have to be visited, fewer variable unifications take place (the constant unifications that take place instead are, by far, less expensive operations), and, since fewer solutions are expected to occur, fewer state restorations will occur. Thus, a fact retrieval with ground arguments will be less expensive to evaluate than its non-ground counterpart.

---

†The closer the distribution resembles a uniform distribution, the better the results will be.

We then have proposed a general framework to estimate the performance of GraphLog (Prolog) queries based on abstract interpretation techniques and mode analysis. Again, a top-down execution is assumed. The method is applicable to the all-solutions case. The basic idea is to associate probabilities with the process of head unification, while considering the expected costs of the subgoals in the body of the clauses. Cost metrics and the average number of solutions are propagated throughout the bodies of the clauses in the usual manner.

Typically, the expressions for the number of solutions and primitive operations for a given tuple <*subgoal, cpat*> will normally be expressed in terms of the number of solutions of other queries represented by the tuples <$subgoal_k$, $cpat_k$>. This implies that an order of evaluation of the analysis equations should be found. If we restrict the queries to be *non-circular*, in the sense that they do not contain recursive calls (direct or indirect), it is always possible to find an order of evaluation which is guaranteed to terminate.

Chapter 4. A qualitative model

So far, we have studied how to obtain a cost model for a specific abstract machine. Unfortunately, if another abstract machine is used, we may not be able to apply our specific model. Furthermore, even if the same abstract machine is being used, but substantial additions or optimizations have been incorporated, the model may produce poor results. We now proceed to analyze how to derive a more general model in which underlying implementations are less relevant.

4.1 Fundamental Database Operations Revisited

We have already mentioned that any Datalog/GraphLog query can be expressed in terms of fundamental database operations (i.e., selections, joins and projections), and therefore the methodology for estimating the value of the cost contributors may be focused on these fundamental operations. Although, strictly speaking, this operational model usually assumes a bottom-up computation, we may also borrow some of the concepts and apply them to a top-down model.

As an example, consider a database of articles sold at a given store. Simplified relations would include base relations for (1) products, say, article(Article_Name, Price, Department, Distributor_Name), (2) internal bookkeeping, say, taxation(Department, Applicable_Tax), (3) personnel grouped by department, say, personnel(Department, Name), and (4) distributor information, say, distributor(Distributor_Name, Distributor_Data). A typical query to retrieve information to calculate the cost of an item (given a specific distributor) after taxes would have the form:

```
:- article(peaches, Price, Dept, stamina_inc), taxation(Dept, Tax).
```

If we know how many distributors are registered for the article peaches,† the calculation of the cost contributors is quite straightforward. In this case, the cost of executing the second subgoal is independent of the specific value of the department (Dept) that is retrieved by the first subgoal (assuming that every department has at most one tax rate in place). If the exact number of distributors that sell peaches to the store is not known, but an average of distributors per article or similar information is available, this average value can be used to estimate an "expected" number whose accuracy will depend on how "average" the article is.

Now, suppose that we pose a query to retrieve the names of the clerks that belong to the store department that sells peaches:

```
:- article(peaches, P, Dept, D), personnel(Dept, Clerk).
```

In this case, the value of the cost contributors associated with the second subgoal will be influenced by the actual department (Dept) that is retrieved by the first subgoal, assuming that different departments have different number of employees. If we do not know the department that will be the output of the first subgoal, accurate knowledge of the distribution function for the personnel relation will not be of much help (since that department can be *any* of the valid departments in the store).

This is a very common situation, and a compromise is needed (unless we want to execute the code to determine the exact value of the department!). One possible solution would be to "weight" all different departments by using a measure related to their probability of appearing in the query (some departments are more likely to be invoked) and take their weighted arithmetic mean as the value for the "average" department. If all departments have the same probability of being selected, a simple average may be used. The utilization of central tendency values seems to be more appropriate than the use of extreme (skewed) values, at least in the long run. It is obvious that the absence of an exact value for the department attribute inevitably produces a loss in the accuracy of the estimate.

---

†We also know that any article has only one price and forms part of one department exclusively.

Selection

The *selection* operation is the easiest of the three basic relational algebra operations to deal with. Several researchers [Selinger79, Christodoulakis83, Fedorowicz84] have proposed diverse formulae for different distribution functions. A straightforward application of these formulae applied to the information indicated in the profile is all we need to determine the expected cardinality (i.e., the average number of output tuples) of the result, and the estimated values of other cost contributors, such as the expected number of visited tuples or the expected number of term unifications, can also be derived from the formulae.

For instance, consider a database base relation r(Integer, String) whose first argument is known to follow an integer normal distribution with mean μ and standard deviation σ, and that we also know that the number of tuples is, say, N. If a query of the form r(X, Y) is used, where X is bound to a constant integer value while variable Y is a free variable, we may estimate the number of tuples that are expected to be retrieved by using our knowledge of how a normal distribution behaves, and by selecting appropriate ranges for our analysis (since we must "discretize" our representation to accommodate integer values exclusively). However, we must be aware that the database profile is often just a simple approximation of the real problem, and a real-life database will normally differ from the "ideal" case. Figure 4.1 shows a typical example of an attribute that follows a discrete version of a normal distribution. Note that its general shape is the one we expect for a normal distribution, but individual values have some deviations from the ideal representation.

An interesting problem occurs when the same variable is attached to two or more argument positions within a predicate. For instance, a subgoal such as a(X,X) establishes an additional restriction: that both arguments have the same value. Even for simpler distribution functions, this seemingly harmless restriction poses a difficult challenge that would require some additional information (for instance, the correlation amongst attributes) to be solved properly.

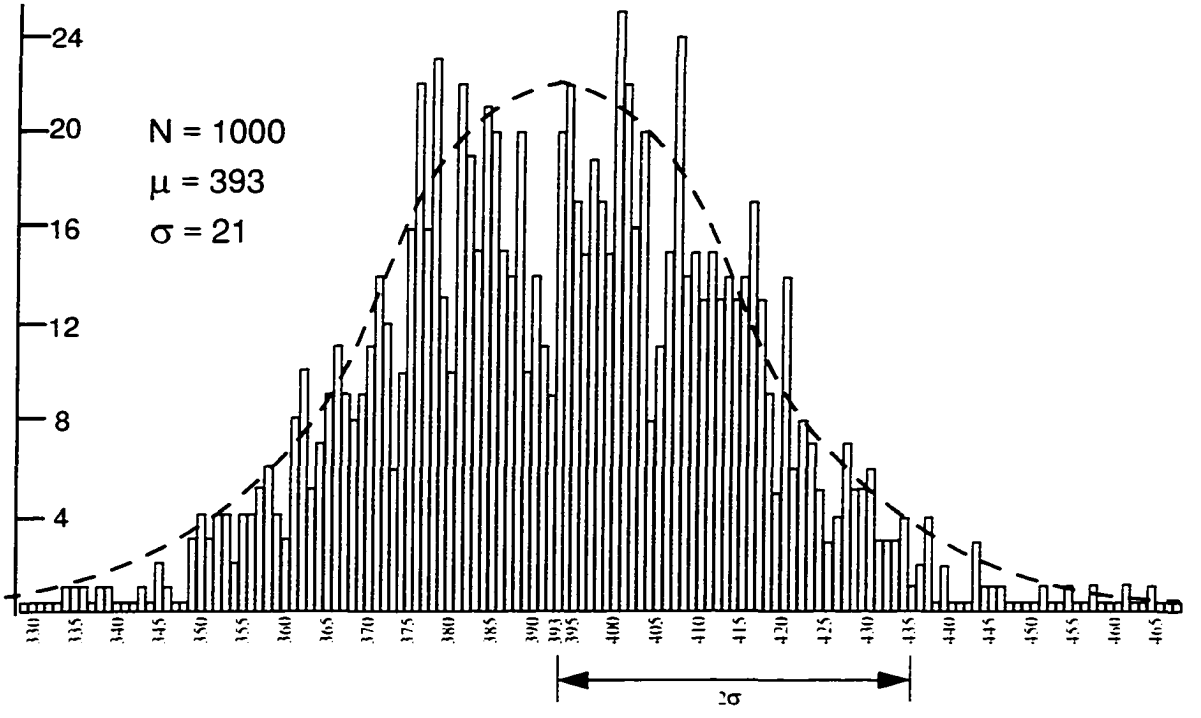


Figure 4.1 Frequency diagram of an attribute that may be approximated by a discrete normal distribution

As mentioned before, if we do not know the exact constant value involved in the selection, we may not be able to use the database profile, since this is often given as a function of the input value. For instance, in the example of Figure 4.1, if the constant value of the argument is known to have a value, say, $X = 381$, we may expect a cardinality of approximately 20 tuples (from the normal distribution). But if the value of X is unknown (perhaps because our abstract interpretation analysis did not keep track of constant values), all we can do is either propose a *range* of values (i.e., from 0 to 22, for the ideal curve) or calculate an *average* value (and we must establish a finite range of "X" values to do so). For example, if we decide to estimate the cardinality of the selection as a simple average, for the attribute depicted in Figure 4.1 we may choose a range of "X" values from $\mu - 3 \times \sigma$ to $\mu + 3 \times \sigma$, in which case we would have an approximate average value of $\nabla \approx 8.0$. If we choose a range of values that varies from $\mu - 2 \times \sigma$ to $\mu + 2 \times \sigma$, the average value will be approximately $\nabla \approx 11.5$. If we consider a range from $\mu - \sigma$ to $\mu + \sigma$, we will have $\nabla \approx 16.2$.

Join

Generally speaking, the *join* operation can be viewed as a Cartesian product of the two relations involved. It is used to combine tuples from two or more relations [Mishra92]. In the case of a Datalog/GraphLog query, a join of the form

```
... s1(A1,....AN), s2 (B1,.....BM), ...
```

can be analyzed (assuming independence of subgoals and a top-down evaluation strategy) as two separate selections (for s1 and s2, respectively) and then, realizing that the second subgoal will be invoked as many times as solutions the first subgoal provides, a particular cost contributor may be calculated as (Eq. 4.1):

$$\text{cost contributor}(s1 \text{ join } s2) = \text{cost contributor}(s1) + \text{solutions}(s1) \times \text{cost contributor}(s2)$$

Naturally, a simple ("mode") analysis must give information as to which arguments will hold constant values in s1 and s2. For instance, in the sequence

```
p(A,B), q(B,C), r(C,D), s(A,D).
```

predicate p will be invoked with two variable arguments, predicates q and r will be called with a first argument constant and a second argument variable and predicate s will have both arguments ground. Note that the simplicity of this analysis is due to the fact that Datalog-like languages guarantee that all arguments are bound to some constant value after any predicate call. Note that this analysis does not keep track of the actual constant values: only the fact that the argument is constant is established. Using similar notation to the one used in the previous chapter, our formula is as follows (Eq. 4.2):

$$\begin{aligned}\text{cost}(p(A,B),q(B,C),r(C,D),s(A,D)) = {} & \text{cost}(p(A,B)|_{[f,f]}) + \\ & nsol(p(A,B)|_{[f,f]}) \times (\text{cost}(q(B,C)|_{[g,f]}) + \\ & nsol(q(B,C)|_{[g,f]}) \times (\text{cost}(r(C,D)|_{[g,f]}) + \\ & nsol(r(C,D)|_{[g,f]}) \times \text{cost}(s(A,D)|_{[g,g]})))\end{aligned}$$

where the calling patterns are abbreviated as "*c*" for constant values and "*f*" for free (or unbound) variables.

Note that, the formula for a reordering of the subgoals should normally have a slightly different aspect, since the "calling" patterns will usually vary. Consider the following reordering of subgoals:

r(C,D), s(A,D), p(A,B), q(B,C).

In this case, our formula becomes (Eq. 4.3):

$$\begin{aligned} cost(r(C,D),s(A,D),p(A,B),q(B,C)) = {} & cost(r(C,D)|_{[f,f]}) + \\ & nsol(r(C,D)|_{[f,f]}) \times (cost(s(A,D)|_{[f,c]}) + \\ & nsol(s(A,D)|_{[f,c]}) \times (cost(p(A,B)|_{[c,f]}) + \\ & nsol(p(A,B)|_{[c,f]}) \times cost(q(B,C)|_{[c,c]}))) \end{aligned}$$

Projection

The most problematic of the basic relational algebra operations is the *projection* operation. The main challenge has to do with handling *duplication* of output tuples after the projection [Kwast94]. Again, statistical considerations regarding the distribution of attribute values may be used to tackle the problem [Gelenbe82, Astrahan85].

To illustrate this idea, consider again the example in Figure 4.1. Note that there are 1000 different tuples of the form r(Integer, String). However, if a projection is performed over the first argument (thus, eliminating the second argument), it becomes clear that we will obtain from 0 to 25 duplicates for each "X" value. Standard Datalog does not discriminate amongst duplicates, and only one value is reported. For this reason, if the first argument is known to be constant, we can establish that at most one valid answer will be derived. If the "X" value lies within a particular region of the distribution curve, we may assign a probability of that value producing such a valid answer. For instance, in our normal distribution for the example in Figure 4.1, if the "X" value is contained within the region from $\mu - 2 \times \sigma$ to $\mu + 2 \times \sigma$, we may estimate that the probability that the projection of that first argument will have cardinality 1 is approximately 95.44% (normal distribution). Note that, if we perform the projection over a first argument that is a free variable, the cardinality of the result will be given by the number of distinct attribute values for the first argument in the relation.

A more complicated scenario takes place when the projection involves more than one relation. A common example occurs when two subgoals in the same query share a common variable, such as in:

... s1(A_1,.....A_i,C,A_{i+1},.....A_N), s2 (B_1,.....B_i,C,B_{i+1},.....B_M), ...

In this case, the projection will be a new relation, say *s1s2*, having the union of all arguments from *s1* and *s2* (i.e., the join of both relations), but with only one instantiation of the common arguments (the projection proper):

... s1s2(A_1,....A_N,B_1,....,B_M,C), ...

As an example, consider the two base relations in Figure 4.2.:

```
% predicate s1

s1(ba,ba,ba).
s1(ba,ba,aa).
s1(aa,ba,aa).
s1(aa,aa,ba).
s1(aa,aa,aa).
```

(a)

```
%predicate s2

s2(ba,ba,ba).
s2(ba,aa,ba).
s2(ba,aa,aa).
s2(aa,ba,ba).
s2(aa,aa,aa).
```

(b)

Figure 4.2 Two ternary predicates s1 and s2

Suppose that we have to estimate the cardinality of query q(C,D), where:

q(C,D) :- s1(A,B,C), s2(A,D,E).

The join would yield the intermediate relation s_a shown in Figure 4.3.

Then, a selection is performed such that the first argument of predicate *s1* is equal to the first argument of predicate *s2*. The resulting relation s_b is shown in Figure 4.4. Finally, we must project arguments 3 and 5 to obtain the final result, as shown in Figure 4.5. Note that from the 12 tuples that are obtained for relation q, only 4 will be in the final answer (the remainder are discarded as duplicates). Fortunately, in this case we already know the upper bound for the cardinality of relation q, which is the product of the sizes of the domains of arguments 3 and 5, i.e. $2 \times 2 = 4$. Thus, if the cardinality of the selection

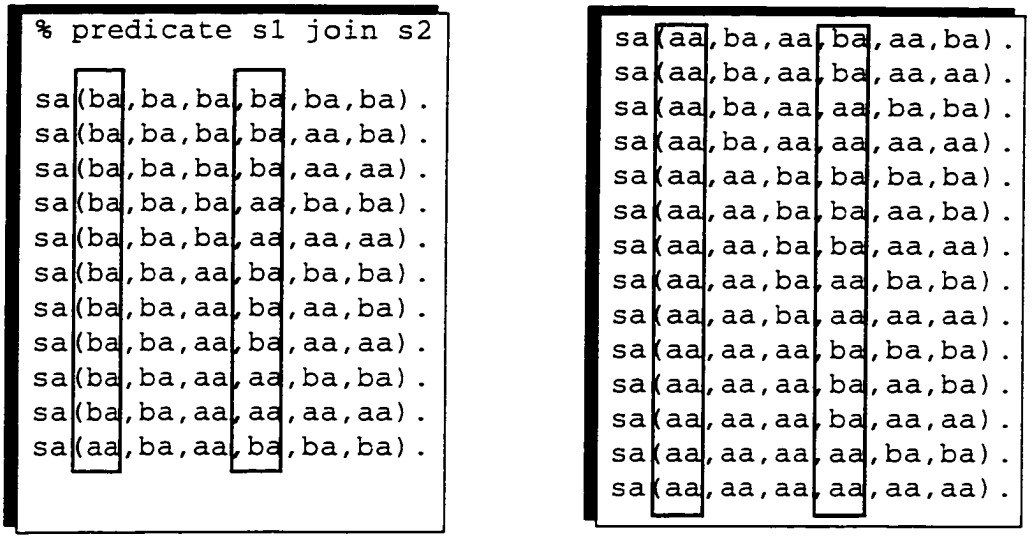


Figure 4.3 Join of predicates s1 and s2

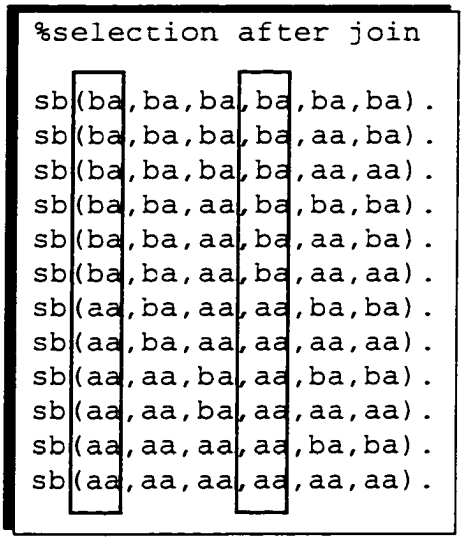


Figure 4.4 Selection after the join of predicates s1 and s2

after join has a value that exceeds this upper bound, we must automatically reduce the estimate to have a value that does not exceed the upper bound. The lower bound is almost always a cardinality of zero.

The whole picture

As has been mentioned before, it is not unusual to obtain formulae for combined relational algebra operations that occur relatively frequently (for instance, a selection after projection). The main advantage of this idea is that some sources of inaccuracy are elim-

```
%projection q

q(ba,ba).
q(ba,aa).
q(ba,aa).
q(aa,ba).
q(aa,aa).
q(aa,aa).
q(aa,ba).
q(aa,aa).
q(ba,ba).
q(ba,aa).
q(aa,ba).
q(aa,aa).
```

Figure 4.5 Final projection of arguments 3 and 5

inated (mainly, the fact that we simply do not know the shape of the distribution for intermediate results), not to mention that the estimation process requires less effort.

In traditional database query modelling, given a sequence of subgoals, we usually decompose it into primitive relational algebra operations, apply formulae derived for the specific characteristics of each participating relation to each of such components, and successively continue applying the formulae to the intermediate relations that result until the entire sequence is analyzed. Note that this approach is only valid when dealing with *non-recursive* queries.

In our model, we use an analogous approach. The estimation of the cost of an extensional database predicate call may simply apply formulae already derived in standard database research. We specifically define a simple formula to estimate the cost of a conjunction of subgoals (Eq. 4.1) that is applicable to the top-down model of execution.

As a final note, practically all methods assume that the *constant* values indicated in the query are valid ones, i.e., values defined in the domain of the respective attribute. This validation can be done for base relations without any major complication (for instance, a simple check for queries to predicate personnel(Department, Name) in which the first argument is a constant may determine whether this is a valid Department), but is not an easy task for intermediate (virtual) relations (unless we keep track of the entire inter-

mediate results instead of a simpler abstraction). For intermediate relations, the validity of constant values is usually automatically assumed for ease of analysis.

4.2 Recapitulation. Cost Estimation and Query Reordering

In this section we repeat some of the ideas that have been mentioned before in order to produce a clear picture of all the important issues that must be addressed.

Given a query of the form

$:- S_1, S_2, ..., S_m.$

whose cost we wish to estimate, we propose to decompose it into simpler components that are assumed to be independent from each other. The simplest choice consists of defining a *subgoal* as the primitive entity to be analyzed. A subgoal is then treated as a "black box": given some inputs (degree of instantiation of the arguments, number of times that the subgoal is expected to be invoked, etc.), the expected values of the cost contributors may be estimated (as the outputs of the black box) and used by successive blocks as their respective inputs (See Figure 1.4). The subgoal itself has to provide some information about internal characteristics such as distribution of attribute values or correlation amongst arguments. The total cost of the query is obtained as the sum of the individual costs of the subgoals. Standard abstract interpretation techniques may be used to determine the degree of instantiation of the arguments and propagate the intermediate results through all successive query components.

When a subquery is known to have at least one constant argument (whose exact value may be unknown at the analysis time), we are forced to choose a way to account for different possible scenarios that result from the selection (since different constant values will produce different values for the cost contributors). A simple compromise is to consider "average queries" that represent either the most typical query that is expected to occur or an amalgamation of all distinct possibilities in which a (weighted) average is calculated.

There are two general groups of subgoals that are treated separately: simple fact retrievals (i.e., extensional database predicates) and general predicate calls (i.e., intension-

al database predicates). The estimation of the cost of a simple fact retrieval can be reduced to a statistical problem since we know (or may determine) the distribution followed by the arguments. General predicate calls are more complex. Specifically, we have to deal with the following issues, amongst others: (a) head unification, (b) clause indexing, (c) independence of subgoals and (d) the fact that the distribution of intermediate results may be difficult to predict. Head unification and clause indexing may be taken into account by assigning to each rule in the predicate a probability of success given the degree of instantiation of the arguments involved. Each rule is then weighted based on this factor.

The problem that two or more rules may provide common solutions is not a trivial one. Given two rules r_1 and r_2, that provide set of answers A_1 and A_2, respectively, we wish to find a new set A_{12} that is the (set-)union of sets A_1 and A_2. Unfortunately, our analysis cannot provide enough information to solve this problem, since we do not know the nature of answers A_1 and A_2. A mutual exclusion analysis may help, in the sense that if we determine that A_1 and A_2 have no answers in common then we know that the cardinality of A_{12} is the sum of the cardinalities of A_1 and A_2. But the general problem of duplication resulting from independent rules is complex to solve.

We also have to handle recursive queries which, in the case of Datalog-like query languages, occur in the form of a predicate closure. In our scheme, a recursive query is also treated as a black box, although the estimation of cost contributors (outputs) has to be solved quite differently. The values of many of the cost contributors are totally method-dependent. Apparently, we may obtain good estimates of the number of tuples that result from the closure (which is a crucial value required by successive black boxes).

4.3 Our Proposed Framework

In this section we delineate how to determine the expected values of the cost contributors for a given subgoal. As a first step, the set of relevant cost contributors that we are going to work with must be selected. Unfortunately, unless we have a history of performance of the form of query under analysis, it is not straightforward to decide which cost contributors are important and which ones may be disregarded.[†]

Once the relevant cost contributors have been selected, we have to calculate their *average* values for each subgoal (represented as a black-box). We will estimate the expected average value for each cost contributor given some information, such as the actual calling pattern or a database profile. The estimation of such average values will normally require the application of formulae derived for the different basic operations of relational algebra previously-mentioned (selection, join, projection). We may use formulae described in the literature (if we happen to be working with a specific database distribution that has been previously studied), or simply consider a simple distribution (a uniform distribution of independent attribute values is the usual choice).

As has been mentioned before, the expected average *number of solutions* to a certain subgoal has to be estimated, since it will be used whenever a join operation occurs.

Once the expected *average* values of all relevant contributors have been estimated for each separate subgoal (by using formulae for the *selection* operation given a certain calling pattern), the *join* operation is considered. We observe that when the all-solutions case is considered (as is the case in a standard GraphLog query), any subgoal will be attempted as many times as solutions the subgoal to the left has provided† (See Figure 4.6). Thus, the values of the cost contributors are scaled by a factor given by the number of solutions of the previous subgoal.

In other words, the value of a cost contributor of a subgoal is estimated as (Eq. 4.4):

$$\text{value}(\text{cost\_contr}_m) = \text{num\_sol}_{m-1} \times \text{average\_value}(\text{cost\_contr}_m)$$

$$\text{num\_sol}_0 = 1$$

The calculation of the number of solutions to the whole query uses the value of the number of answers to the last subgoal (scaled by the values of the number of answers to all previous subgoals) as an upper bound.‡

$$\text{num\_sol}_{\text{query}} = \text{num\_sol}_1 \times \text{num\_sol}_2 \times \ldots \times \text{num\_sol}_m \qquad \ldots(\text{Eq. 4.5})$$

---

†Accumulative profiling is often the best aid to this end

‡For the case of the left-most subgoal, a factor of 1 must be considered

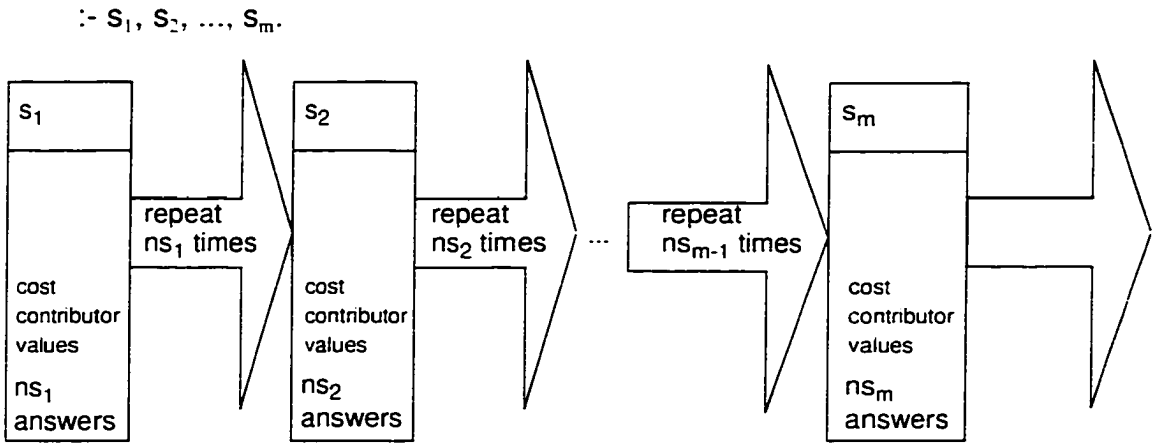


Figure 4.6 Cost contributors are estimated for each subgoal

This value does not consider duplicates, and therefore, we may obtain an overestimate if we use this value directly. To avoid this, our framework would require a way to take into consideration the removal of duplicate tuples from the final solution.

The calculation of the average value of a cost contributor for the whole query is accomplished by adding all individual values of the cost contributors for the different subgoals in the query under consideration (Eq. 4.6):

$$\text{value}(\text{cost\_contr}_{query}) = \text{value}(\text{cost\_contr}_1) + \text{value}(\text{cost\_contr}_2) + \ldots + \text{value}(\text{cost\_contr}_m)$$

Finally, once we know the values of all cost contributors for the whole query, we are in a position to determine the total cost of the query. If we know the expected average cost of each contributor per se (as a primitive operation), the problem is reduced to what we have already discussed in Chapter 3. However, if the empirical values of these primitive operations are unknown, we are forced to attach some weights to each of them, or give priority to some of them. The simplest strategy is to select one single cost contributor and base our rankings on this sole parameter. Otherwise, we face the problem of assigning specific weights to the cost contributors.

‡Unfortunately, errors in the estimation of the number of answers to the whole query may increase exponentially with the number of components [Ioannidis95].

Chapter 5. Handling Recursive Queries

So far, we have characterized the cost of a (non-recursive) predicate by means of simple cost measures, such as the number of visited tuples, the number of successful unification operations or the number of solutions that are obtained. The only complication we have encountered has to do with having to consider different variants of clause indexing, depending on the actual implementation of the query evaluator.

Extending those results to recursive predicates poses a real challenge. Undecidability of general recursion is well-known, and so is the potential occurrence of infinite computations. It does not come as a surprise that most researchers have concentrated on very specific cases of recursion. For instance, Debray and Lin [Debray93] have developed a method for cost analysis of Prolog programs based on knowledge about "size" relationships between arguments of predicates, which is only applicable to recursive definitions in which an argument decreases in size at each new recursive invocation.

5.1 Execution Cost of a Recursive Query

Besides recursion with decreasing size functions over new recursive steps, there are other cases of recursion that may be handled by our cost model. The most important of these is linear recursion over a database domain. In fact, one of the greatest advantages of query languages derived from Datalog is that every (database) query produces a finite number of answers, and infinite loops are therefore avoided by choosing an appropriate evaluation method.

One immediate consequence of the selection of a specific evaluation method is that the actual cost of evaluating a recursive predicate will depend on the chosen method. There are many different evaluation methods that deal with recursive queries [Ceri90]. In general, cost measures such as the number of visited tuples or the number of unification attempts are algorithm-dependent, and there are additional factors that add to the

evaluation costs (for instance, bookkeeping of structures or validation of certain conditions).

However, there is one cost measure that is totally independent of the evaluation method: the number of solutions to the query. Furthermore, the number of solutions is a quantity that is propagated to other subgoals in the query, since it affects the number of times that the successive subgoals will be invoked.

For these reasons, it is relevant to devise a method to estimate the number of solutions that is associated with a recursive query.

5.2 Formulation of a Recursive Query in Terms of Transitive Closure

Jagadish and Agrawal [Jagadish87] have shown that every linearly recursive query can be expressed as a transitive closure possibly preceded and followed by the usual operators of standard relational algebra (joins, projections, selections, etc.). A recursive rule is *linear* if there is exactly one occurrence of the recursive literal in the body. Bancilhon and Ramakrishnan have conjectured that most recursive queries are linear [Bancilhon86]. The significance of this result is that it is potentially feasible to predict the number of solutions of every linearly recursive query if we derive a general method that is able to determine the number of solutions of the transitive closure case.

Thus, we suggest the following methodology to estimate the number of solutions of a recursive predicate:

1. Transform the linearly recursive predicate into its equivalent form that involves transitive closure;
2. Estimate the cost of the transformed predicate in terms of its constituents (i.e., normal non-recursive predicates and the transitive closure itself).

Thus, it becomes clear that we need to devise a method to estimate the cardinality of a transitive closure.

5.3 Predicting the Average Number of Solutions of a Transitive Closure

One of the most common uses of recursion in GraphLog is simple transitive closure, exemplified by the following two rules:

```
tc(X,Y) :- b(X,Y).
tc(X,Y) :- b(X,Z), tc(Z,Y).
```

where *tc* defines the result of the transitive closure and *b* is the relation (or base predicate) over which the closure is performed. Note that only two (sets of) arguments are involved in the closure relation.

Our goal is to find the cardinality of *tc* (i.e., the number of tuples n_{tc} that are obtained as a result of applying the transitive closure operator) given some information about predicate *b*. It is evident that the nature of predicate *b* has a substantial impact on the cardinality of its transitive closure: if we represent a predicate by its equivalent graph in which a fact is represented by a directed edge between the two values (nodes) of the closure arguments, the transitive closure of a tree-like structure will produce fewer tuples than, for instance, that of a heavily connected structure with the same number of facts. By the same token, a predicate with a higher number of facts will normally produce more tuples after the application of the transitive closure operator than a similarly-structured predicate with fewer facts.

The simplest possible study of transitive closure is one that only considers the cardinality of *b* (that is, the number of tuples n_b that are associated with predicate *b*), disregarding any internal relationships between the arguments.

Suppose that we are interested in determining the number of tuples n_{tc} that result from applying transitive closure to predicate *b*. If we assume that the number of unique tuples n_b associated with predicate *b* is known, and so is the number of distinct attribute values for the relation, n_{at}, some upper- and lower bounds may be established. By properties of transitive closure, we know that $n_b \le n_{tc} \le (n_b)^2$ ...(Eq. 5.1). Furthermore, for $n_b \ge n_{at}^2 - n_{at} + 2$ ...(Eq. 5.2), the limit value $(n_{at})^2$ is obtained. Unfortunately, these frontier values are not that helpful for large values of n_{at}.

For even the simplest possible case, that of a uniform distribution of independent attributes, deriving exact formulae proves to be a hard task. For example, we derived the following formula for the average expected value in the trivial case when $n_b = 2$:

$$\overline{n_{tc}} = \frac{2\left(n_{at}^{4} + n_{at}^{3} - 3n_{at}^{2} + n_{at}\right)}{n_{at}^{4} - n_{at}^{2}} \qquad \text{...(Eq. 5.3)}$$

The complexity of the exact formulae increases as the value of n_b does, and each formula has to be obtained separately, which produces an impractical situation.

5.4 Estimating the Average Cardinality of Transitive Closure

As mentioned before, the cardinality of a transitive closure may vary from a value in which no tuples are added as a result of the closure to a maximum value given by the square of the original number of tuples (in a graph representation, this would correspond to a "complete" graph for the involved "input" nodes). Since this range of values may produce a vast interval, a compromise is to work with central tendency measurements, such as the arithmetic mean.

For this purpose, we have generated randomly distributed tuples for our base predicate *b* and obtained results for several values of n_{at} (the number of distinct attribute values for the relation) and n_b (the number of unique tuples for predicate *b*)†. After several experiments, it appears that the *average* number of tuples of the transitive closure can be characterized by means of three different regions: (a) a simple (linear) behaviour for small values of n_b; (b) a non-linear region for intermediate values of n_b; and (c) an exponential region for higher values of n_b (Figure 5.1 and Figure 5.2). We proceed to characterize these three regions.

5.4.1 Region of Small Values for the Number of Tuples in the Base Predicate

For small values of n_b, a surprisingly simple linear formula was empirically derived. If we express n_b in terms of n_{at} in the form $n_b = n_{at}/A$, the average number of tuples of the corresponding transitive closure can be expressed (approximately) as:

†In our experiments, n_b tuples were randomly selected from the $(n_{at})^2$ possible tuples that can be formed with n_{at} distinct attribute values at each argument position.

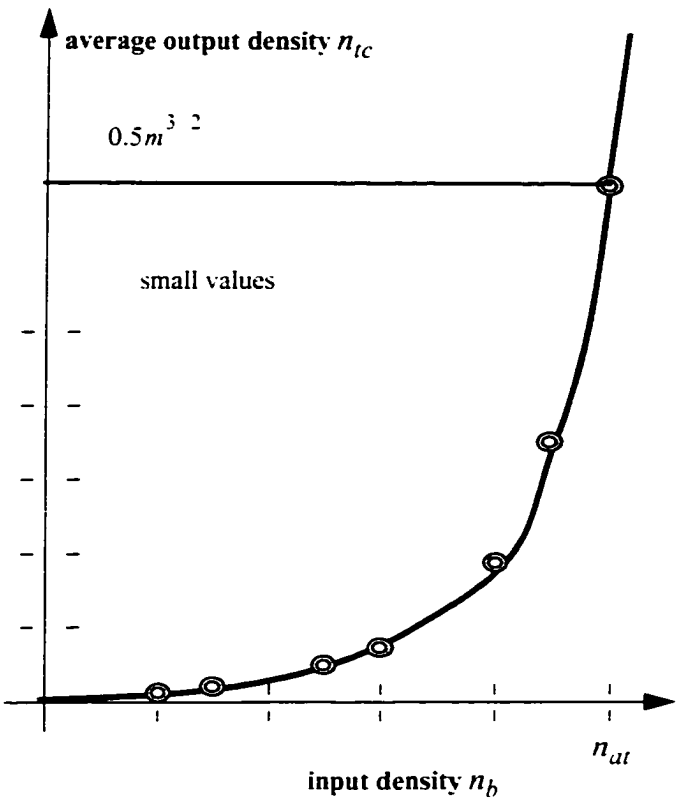


Figure 5.1 Region for small values

$\overline{n_{tc}} \approx n_{at}\ (A-1)$ ...(Eq. 5.4). For instance, if $A = 5$, $\overline{n_{tc}} \approx n_{at}\ 4$, or if $A = 4$, $\overline{n_{tc}} \approx n_{at}\ 3$, or if $A = 2$, $\overline{n_{tc}} \approx n_{at}$ (Table 5.1), and so on. The formula seems to work well for $A \geq 1.25$, although accuracy starts to degrade sharply in the neighbourhood of this value. Furthermore, the predicted value is more precise for higher values of n_{at}.

n_{at}	n_b	A	average n_{tc}	standard deviation	Formula value
300	60	5	75.74	5.22	75
600	120	5	149.92	9.05	150
900	180	5	226.20	10.52	225
1200	240	5	300.06	10.63	300
300	150	2	297.04	52.69	300
600	300	2	595.42	63.92	600
900	450	2	905.08	90.51	900
1200	600	2	1200.22	97.58	1200

Table 5.1 The linear region

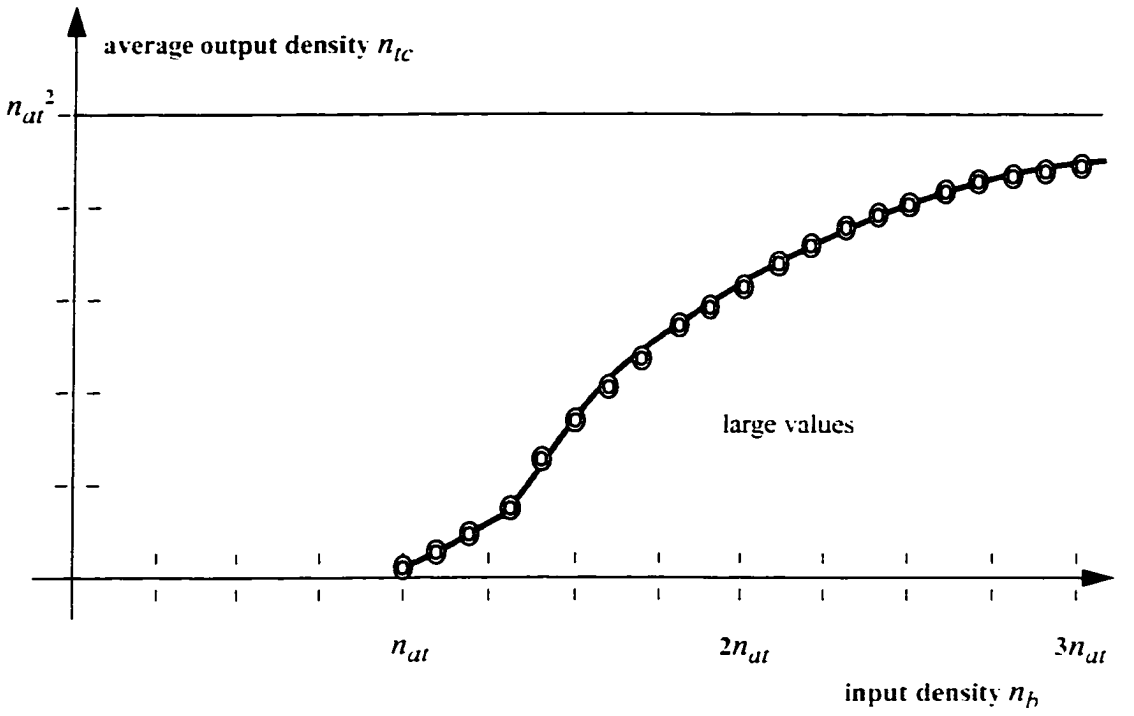


Figure 5.2 Region for large values

5.4.2 Region of Intermediate Values for the Number of Tuples in the Base Predicate

Our linear formula begins to fail when the constant A starts getting closer to one (Table 5.2). In fact, the standard deviation of the recorded values also becomes bigger. We have been unable to derive a simple general formula for this range. Fortunately, this "intermediate" region is represented by a relatively narrow interval of values that range from approximately $n_b \geq 0.8n_{at}$ to $n_b \leq 1.2n_{at}$. As a practical solution, we have obtained some approximate formulae for different values of n_b in this region. For instance, for $n_b = 0.9n_{at}$, we have applied the approximation $\overline{n_{tc}} \approx 4 \times n_{at}\ (3 \times (A-1))$, and for $n_b = n_{at}$, the formula $\overline{n_{tc}} \approx n_{at}^{1.5} \cdot 2$ gives a good approximation of the cardinality of the transitive closure. Note that, since this region is relatively small, the derivation of approximate formulae for different values of n_b represents a feasible strategy.†

†Naturally, our proposed formulae represent just simple approximations. We decided not to spend too much time in deriving more exact formulae, since these approximations satisfy our needs rather adequately.

n_{at}	n_b	A	average n_{tc}	standard deviation	Formula value
200	140	1.43	416.98	124.27	466.66
300	210	1.43	660.24	160.95	700
400	280	1.43	890.60	220.74	933.33
500	350	1.43	1099.84	243.22	1166.66
600	420	1.43	1345.46	204.80	1400
700	490	1.43	1567.90	262.90	1633.33
800	560	1.43	1861.22	384.86	1866.66
900	630	1.43	2051.42	336.96	2100
1000	700	1.43	2212.88	364.63	2333.33
1100	770	1.43	2482.68	449.78	2566.66
1200	840	1.43	2753.32	415.27	2800
200	160	1.25	622.18	185.01	800
300	240	1.25	1111.98	378.71	1200
400	320	1.25	1290.66	324.14	1600
500	400	1.25	1683.82	332.95	2000
600	480	1.25	2376.98	886.45	2400
700	560	1.25	2543.64	621.25	2800
800	640	1.25	2857.22	656.91	3200
900	720	1.25	3227.16	1038.85	3600
1000	800	1.25	3842.14	975.85	4000
1100	880	1.25	4149.82	741.25	4400
1200	960	1.25	4427.28	881.50	4800

Table 5.2 The intermediate region

5.4.3 Region of Large Values for the Number of Tuples in the Base Predicate

An important observation is that for values $n_b \geq 1.2n_{at}$, the *percentage* of the corresponding maximum value $(n_b)^2$ that is obtained after the closure can be considered almost a constant (as seen in Table 5.3). The values of some percentages are depicted in Table 5.4.

Furthermore, we observe that some of these percentages may be represented by very simple fractions. For instance, for $n_b = 1.3n_{at}$, the fraction is 1/6; for $n_b = 1.4n_{at}$, the associated fraction is 1/4; for $n_b = 1.5n_{at}$, the fraction is 1/3; for $n_b = 1.6n_{at}$, the fraction is 1/2.5; for $n_b = 1.75n_{at}$ the fraction is 1/2.

n_{at}	n_b	average n_{tc}	standard deviation	sample size	$P = n_{tc}/(n_{at}^2)$
200	320	16529.50	2596.02	50	41.32%
300	480	38156.16	4604.65	50	42.39%
400	640	65832.52	7812.30	50	41.14%
500	800	99464.94	8430.00	50	39.78%
600	960	149964.12	13676.95	50	41.65%
700	1120	205556.32	15957.11	50	41.95%
800	1280	257682.22	24452.03	50	40.26%
900	1440	330584.98	21814.16	50	40.81%
1000	1600	410680.24	30130.36	50	41.06%
1100	1760	495855.40	28209.98	50	40.98%
1200	1920	589839.44	34806.74	50	40.96%

Table 5.3 Percentages of the maximum value for $n_b = 1.6$ m

$n_b=1.2n_{at}$	$n_b=1.3n_{at}$	$n_b=1.4n_{at}$	$n_b=1.5n_{at}$	$n_b=1.6n_{at}$	$n_b=1.7n_{at}$	$n_b=1.8n_{at}$	$n_b=1.9n_{at}$
0.09	0.17	0.25	0.33	0.41	0.47	0.53	0.59

Table 5.4 Percentages of the maximum value for some factors

This particular behaviour may be approximated by an analytical formula. In fact, the values that are obtained strongly suggest that we may use an exponential formula to model this region. Thus, we have used the following simple general formula to characterize this subregion†:

$$\overline{n_{tc}} \approx n_{at}^2 \left(1 - \exp\left(-\frac{n_b}{n_{at}} + \beta \right)^2 \right) \qquad \text{...(Eq. 5.5)}$$

For instance, the values that are obtained when using this formula when $\beta = 0.9$ are shown in Table 5.5.

	$n_b = 1.1n_{at}$	$n_b = 1.2n_{at}$	$n_b = 1.3n_{at}$	$n_b = 1.4n_{at}$	$n_b = 1.5n_{at}$	$n_b = 1.6n_{at}$	$n_b = 1.7n_{at}$	$n_b = 1.8n_{at}$	$n_b = 1.9n_{at}$
derived (formula) values	0.0392	0.0861	0.1479	0.2212	0.3023	0.3874	0.4727	0.5551	0.6321
experimental values	0.06	0.09	0.17	0.25	0.33	0.41	0.47	0.53	0.59

Table 5.5 Comparison between the formula and the experimental results

Also, for the range $n_b \geq 2n_{at}$, we have used the following simple formula:

$$\overline{n_{tc}} \approx n_{at}^2 \left(1 - \exp\left(-\frac{n_b}{\alpha} \right) \right) \qquad \text{...(Eq. 5.6)}$$

†Again, this formula just represents a reasonable approximation of the values under consideration, and "better" formulae may be proposed as well if more accuracy is needed.

where the value $\alpha \approx 3n_{at} - \frac{n_b}{2}$ seems to give satisfactory results (Table 5.6).

5.5 Recursion Revisited

Once a formula that predicts the cardinality of transitive closure has been obtained, we may use it to predict the cardinality of a recursive predicate.

As an example, let us consider a generalized version of the same generation example proposed by Bancilhon and Ramakrishnan [Bancilhon86]:

```
p(X,Y) :- flat(X,Y).
p(X,Y) :- up(X,XU), p (XU,YU), down(YU,Y).
```

where *flat*, *up* and *down* are extensional database predicates, and *p* is the recursive (derived) predicate.

This predicate can be expressed in terms of transitive closure as follows:

```
updown(X,YU,XU,Y) :- up(X,XU), down(YU,Y).
p2(X,Y) :- flat(X,Y).
p2(X,Y) :- flat(XU,YU), updowntc(X,YU,XU,Y).
```

where *updowntc* indicates the transitive closure of predicate *updown*:

$$\text{updowntc}(X,YU,XU,Y) :- \text{updown}(X,YU,XU,Y)^{+}.$$

We will analyze the GraphLog program for the generalized version of the same generation problem as shown in Figure 5.3.

The visual representation of this program is shown in Figure 5.4.

We wish to estimate the number of tuples associated with recursive predicate *p2*.

```
p2(X,Y) :- flat(X,Y).
p2(X,Y) :- flat(XU,YU), updowntc(X,YU,XU,Y).
```

Since duplication of tuples must be considered (the second rule may produce tuples that are already part of the base predicate, *flat*), the cardinality n_{p2} of *p2* will be:

$$n_{\text{flat}} \le n_{p2} \le n_{\text{flat}} + n_{\text{updowntc}}$$

where n_{flat} is the number of tuples of predicate *flat* (which is known from the database profile) and n_{updown} is the number of tuples of the transitive closure of predicate *updown*.

```
%extensional DB predicates
db_schema(up, 2).
db_schema(down, 2).
db_schema(flat, 2).

%p(X,Y) :- flat(X,Y).
node( g4, n9, [v('X')] ).
node( g4, n10, [v('Y')] ).
edge( g4, n9, n10, flat ).
dist_edge( g4, n9, n10, p).

%p(X,Y) :- up(X,XU),p(XU,YU),down(YU,Y).
node( g3, n5, [v('X')] ).
node( g3, n6, [v('XU')] ).
node( g3, n7, [v('YU')] ).
node( g3, n8, [v('Y')] ).
edge( g3, n5, n6, up).
edge( g3, n6, n7, p).
edge( g3, n7, n8, down).
dist_edge( g3, n5, n8, p).

%uptc(X,Y) :- up(X,Y)+.
node( g6, n14, [v('X')] ).
node( g6, n15, [v('Y')] ).
edge( g6, n14, n15, up:+: ).
dist_edge( g6, n14, n15, uptc).

%downtc(X,Y) :- down(X,Y)+.
node( g8, n19, [v('X')] ).
node( g8, n20, [v('Y')] ).
edge( g8, n19, n20, down:+: ).
dist_edge( g8, n19, n20, downtc).
```

```
%updown(X,YU,XU,Y) :- up(X,XU),down(YU,Y).
node( g9, n21, [v('X')] ).
node( g9, n22, [v('Y')] ).
node( g9, n23, [v('XU')] ).
node( g9, n24, [v('YU')] ).
node( g9, n25, [v('X'),v('YU')] ).
node( g9, n26, [v('XU'),v('Y')] ).
edge( g9, n21, n23, up ).
edge( g9, n24, n22, down ).
dist_edge( g9, n25, n26, updown ).

%updowntc(X,YU,XU,Y) :- updown(X,YU,XU,Y)+.
node( g10, n27, [v('X'),v('YU')] ).
node( g10, n28, [v('XU'),v('Y')] ).
edge( g10, n27, n28, updown:+: ).
dist_edge( g10, n27, n28, updowntc ).

%p2(X,Y) :- flat(XU,YU), updowntc(X,YU,XU,Y).
node( g12, n29, [v('X'),v('YU')] ).
node( g12, n30, [v('XU'),v('Y')] ).
node( g12, n35, [v('X')] ).
node( g12, n36, [v('XU')] ).
node( g12, n37, [v('YU')] ).
node( g12, n38, [v('Y')] ).
edge( g12, n36, n37, flat).
edge( g12, n29, n30, updowntc).
dist_edge( g12, n35, n38, p2).

%p2(X,Y) :- flat(X,Y).
node( g13, n39, [v('X')] ).
node( g13, n40, [v('Y')] ).
edge( g13, n39, n40, flat ).
dist_edge( g13, n39, n40, p2).
```

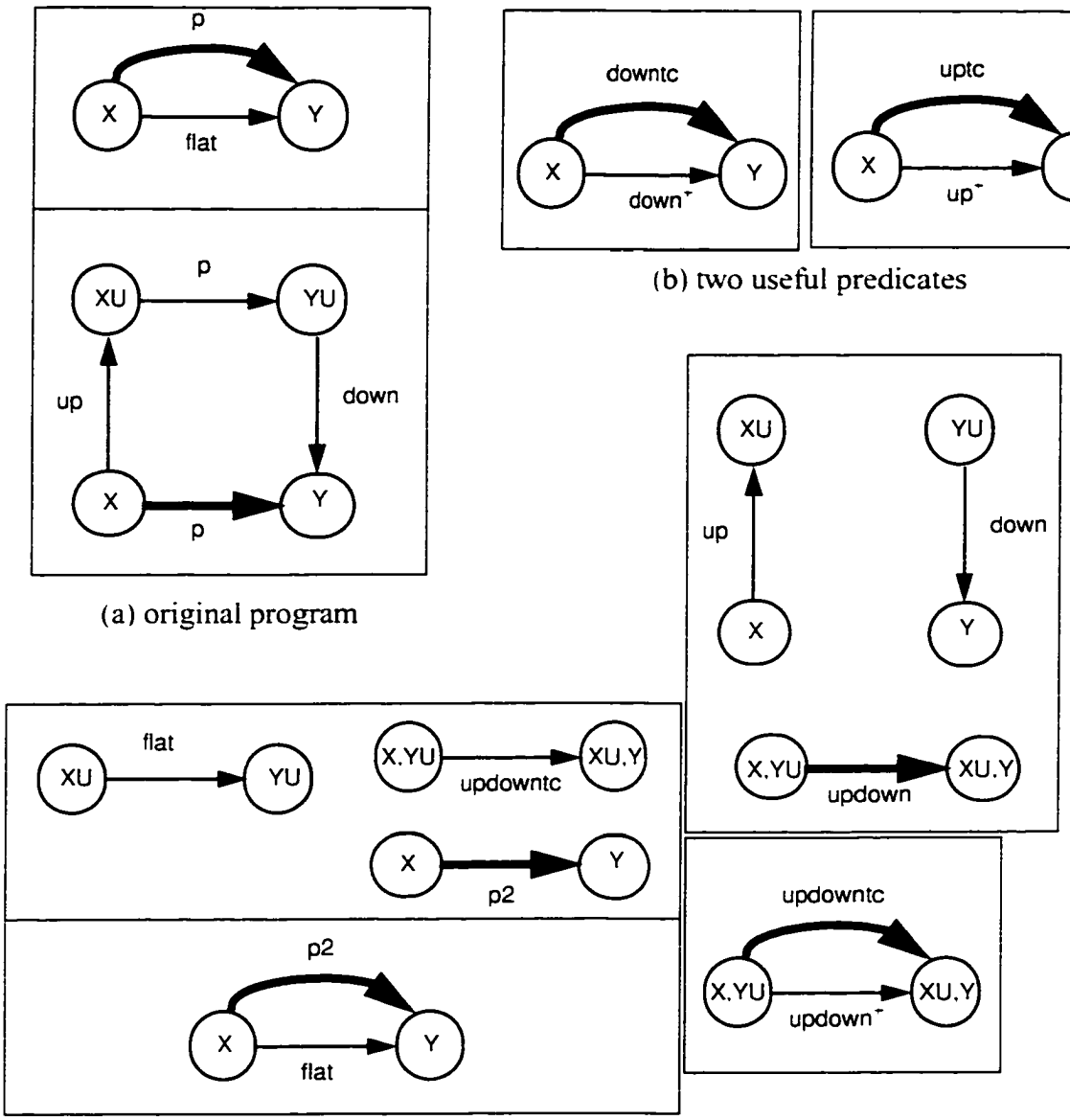
Figure 5.3 GraphLog program

updown(X,YU,XU,Y) :- up(X,XU),down(YU,Y).

The cardinality of predicate *updown* may be inferred by using the normal method for estimating the cost of non-recursive predicates. In this specific case, since the sub-goals do not share any variable, we have that

$$n_{\text{updown}} = n_{up} \times n_{\text{down}}$$

If we know the number N of distinct attribute values common to relations *up* and *down*, we might be tempted to use our formulae for cardinality of transitive closure, given $n_{at} = N^2$ and $n_b = n_{up} \times n_{\text{down}}$. This would be perfectly valid if there were no restrictions whatsoever regarding how the tuples are distributed, i.e., if we have a random distribution. Unfortunately, it seems not to be the case in real-life databases. We



(c) program in terms of transitive closure

Figure 5.4 GraphLog program for the recursive program

consistently observed that our formulae produced some overestimates for higher values of n_{at}. Empirical results have also shown that our predictions are adequate when $n_{updown} < N^2$, (i.e., in the region for "small values" of n_{updown}). Furthermore, we have observed that our estimates may be improved for the other two regions: for the region of "higher values" of n_{updown}, the cardinality of the transitive closure consistently seems to be directly related to the product of the cardinalities of the individual transitive closures of *up* and *down*; for the region of intermediate values, the cardinality of the transitive

m	n_b	average n_{tc}	standard dev.	$P = n_{tc}$ (m^2)	formula value
200	400	25606.54	1954.09	64.016 %	25284.82
300	600	57362.62	3095.88	63.736 %	56890.85
400	800	101008.30	5246.09	63.130 %	101139.28
500	1000	156544.92	7293.12	62.618 %	158030.14
600	1200	228413.40	9803.62	63.448 %	227563.40
700	1400	308662.10	14327.83	62.992 %	309739.07
800	1600	405487.96	16580.33	63.357 %	404557.15
900	1800	513573.90	16901.15	63.404 %	512017.65
1000	2000	629838.26	21458.76	62.984 %	632120.55
1100	2200	765588.20	26922.60	63.272 %	764865.87
1200	2400	909321.14	28480.84	63.147 %	910253.60
200	500	31946.38	1125.21	79.866%	30413.95
300	750	71680.06	2233.35	79.645%	68431.40
400	1000	127952.76	3654.25	79.970%	121655.83
500	1250	198747.26	5035.38	79.499%	190087.24
600	1500	286477.56	6290.05	79.577%	273725.62
700	1750	389129.06	9412.63	79.414%	372570.99
800	2000	510678.64	10154.18	79.794%	486623.33
900	2250	646398.50	12478.27	79.802%	615882.66
1000	2500	800567.00	12044.72	80.057%	760348.96
1100	2750	967210.24	13964.92	79.935%	920022.24
1200	3000	1147540.48	22606.77	79.690%	1094902.50
200	600	35422.04	946.28	88.555%	34586.589
300	900	79985.72	1958.93	88.873%	77819.82
400	1200	141916.32	2946.10	88.698%	138346.35
500	1500	220735.90	3680.58	88.294%	216166.17
600	1800	318177.28	5551.39	88.383%	311279.29
700	2100	433924.28	6937.07	88.556%	423685.71
800	2400	568159.20	7330.76	88.775%	553385.41
900	2700	716804.46	8854.25	88.494%	700378.42
1000	3000	884537.58	10584.63	88.454%	864664.71
1100	3300	1070330.12	14063.02	88.457%	1046244.30
1200	3600	1276891.64	13865.54	88.673%	1245117.19

Table 5.6 The exponential region

closure seems to be related to the products $n_{down} \times n_{uptc}$ and $n_{up} \times n_{downtc}$. Table 5.7 shows some typical results for different values.

m	n_{up}	n_{down}	n_{flat}	$n_{updowntc}$	n_b	m_{tc}	B= n_b m_{tc}	n_{uptc}	n_{downtc}	$n_{updowntc}$ using formula
150	220	50	7	26305.3	11000	22500	0.489	7500	75	21531.0
30	10	10	1	115.1	100	900	0.111	15	15	112.5
30	15	10	1	182.9	150	900	0.167	30	15	180.0
70	10	14	1	144.1	140	4900	0.029	11.7	17.5	144.1
10	20	20	3	4255.3	400	100	4.000	63.2	63.2	3994.24
30	60	45	5	130339.0	2700	900	3.000	568.9	300.0	170670.0
20	80	40	2	66819.5	3200	400	8.000	392.8	252.8	99290.3
10	20	40	3	5871.2	800	100	8.000	63.2	98.2	6204.22

Table 5.7 Estimating the cardinality of a transitive closure

Let us study a very simple example and try to explain why the distribution of the relation that the transitive closure is applied to is not uniform.

Example. Consider a database with $N = 10$ distinct attribute values and the following randomly generated extensional database predicates:

```
up(1,2).        down(1,1).      flat(7,8).
up(1,5).        down(1,8).      flat(10,6).
up(3,2).        down(3,2).
up(3,10).       down(3,10).
up(5,6).        down(7,3).
up(6,1).        down(8,6).
up(7,8).        down(9,5).
up(7,10).       down(10,10).
up(9,9).        flat(2,7).
up(10,5).
```

A graphical representation of the two predicates is shown in Figure 5.5.

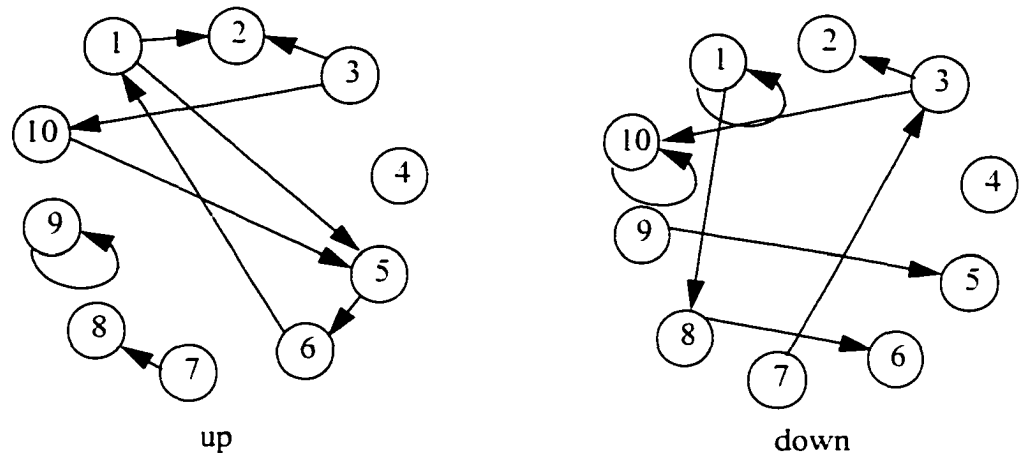


Figure 5.5 Graphical representation of base predicates up and down

In this example, $n_{up} = 10$ and $n_{down} = 8$. The transitive closure of predicate *up* follows the following behaviour:

predicate "up"	1	2	3	4	5	6	7	8	9	10
paths of length 1	<1,2>	<1,5>	<3,2>	<3,10>	<5,6>	<6,1>	<7,8>	<7,10>	<9,9>	<10,5>
paths of length 2	<1,6>	<3,5>	<5,1>	<6,2>	<6,5>	<7,5>	<10,6>			
paths of length 3	<1,1>	<3,6>	<5,2>	<5,5>	<6,6>	<7,6>	<10,1>			
paths of length 4	<1,2>	<1,5>	<3,1>	<5,6>	<6,1>	<7,1>	<10,2>	<10,5>		
paths of length 5	<1,6>	<3,2>	<3,5>	<5,1>	<6,2>	<6,5>	<7,2>	<7,5>	<10,6>	
paths of length 6	<1,1>	<3,6>	<5,5>	<5,2>	<6,6>	<7,6>	<10,1>			
paths of length 7	<1,2>	<1,5>	<3,1>	<5,6>	<6,1>	<7,1>	<10,2>	<10,5>		
paths of length 8	<1,6>	<3,2>	<3,5>	<5,1>	<6,2>	<6,5>	<7,2>	<7,5>	<10,6>	

Note that the tuples in the closure follow a recurrent pattern (all paths of length 6 are also paths of length 3; all paths of length 7 are paths of length 4 as well; and so on).

Similarly, predicate *down* has the following closure:

predicate "down"	1	2	3	4	5	6	7	8
paths of length 1	<1,1>	<1,8>	<3,2>	<3,10>	<7,3>	<8,6>	<9,5>	<10,10>
paths of length 2	<1,1>	<1,6>	<1,8>	<3,10>	<7,2>	<7,10>	<10,10>	
paths of length 3	<1,1>	<1,6>	<1,8>	<3,10>	<7,10>	<10,10>		
paths of length 4	<1,1>	<1,6>	<1,8>	<3,10>	<7,10>	<10,10>		

Predicate *updown* is the Cartesian product of both relations:

	(1,1)	(1,8)	(3,2)	(3,10)	(7,3)	(8,6)	(9,5)	(10,10)
(1,2)	(1,1,2,1)	(1,1,2,8)	(1,3,2,2)	(1,3,2,10)	(1,7,2,3)	(1,8,2,6)	(1,9,2,5)	(1,10,2,10)
(1,5)	(1,1,5,1)	(1,1,5,8)	(1,3,5,2)	(1,3,5,10)	(1,7,5,3)	(1,8,5,6)	(1,9,5,5)	(1,10,5,10)
(3,2)	(3,1,2,1)	(3,1,2,8)	(3,3,2,2)	(3,3,2,10)	(3,7,2,3)	(3,8,2,6)	(3,9,2,5)	(3,10,2,10)
(3,10)	(3,1,10,1)	(3,1,10,8)	(3,3,10,2)	(3,3,10,10)	(3,7,10,3)	(3,8,10,6)	(3,9,10,5)	(3,10,10,10)
(5,6)	(5,1,6,1)	(5,1,6,8)	(5,3,6,2)	(5,3,6,10)	(5,7,6,3)	(5,8,6,6)	(5,9,6,5)	(5,10,6,10)
(6,1)	(6,1,1,1)	(6,1,1,8)	(6,3,1,2)	(6,3,1,10)	(6,7,1,3)	(6,8,1,6)	(6,9,1,5)	(6,10,1,10)
(7,8)	(7,1,8,1)	(7,1,8,8)	(7,3,8,2)	(7,3,8,10)	(7,7,8,3)	(7,8,8,6)	(7,9,8,5)	(7,10,8,10)
(7,10)	(7,1,10,1)	(7,1,10,8)	(7,3,10,2)	(7,3,10,10)	(7,7,10,3)	(7,8,10,6)	(7,9,10,5)	(7,10,10,10)
(9,9)	(9,1,9,1)	(9,1,9,8)	(9,3,9,2)	(9,3,9,10)	(9,7,9,3)	(9,8,9,6)	(9,9,9,5)	(9,10,9,10)
(10,5)	(10,1,5,1)	(10,1,5,8)	(10,3,5,2)	(10,3,5,10)	(10,7,5,3)	(10,8,5,6)	(10,9,5,5)	(10,10,5,10)

From this table, it should be evident that the distribution of attribute values of pairs ([X, YU], [XU, Y]) is not random at all. Not only are many [XU, Y] pairs shared by some [X, YU] pairs, but also the XU value is totally determined by the X value. For instance, if X has a value of 1, the value of XU is either 2 or 5, and therefore, although there are 1000 possible candidates ([1, YU], [XU, Y]), only 200 of them comply with the restriction ([1, YU], [2, Y]) or the restriction ([1, YU], [5, Y]).

In other words, although we have derived some formulae to estimate the cardinality of the transitive closure of a uniform distribution of attribute values, they may not be accurate for "real" predicates, as different distribution functions may be encountered.

In summary, to estimate the cardinality of the transitive closure of a cartesian product we propose to consider three different regions: (a) small values of n_c, the cardinality of the cartesian product, with respect to n_{at}^2; (b) intermediate values of n_c; and (c) higher values of n_c. Experimental results have indicated that we may use our formulae when $n_c < n_{at}^2$, in the region that we have called "small values of n_c" with some accuracy. Once more, it is the intermediate region which poses the major challenge: a good estimate of the cardinality of the transitive closure can be obtained by using the products $n_{\text{down}} \times n_{\text{uptc}}$ and $n_{\text{up}} \times n_{\text{downtc}}$, either the midpoint or some value in between. Finally, for the region of

"higher values of n_c", the cardinality of the transitive closure seems to be directly related to the (product of the) cardinalities of the individual transitive closures for *up* and *down*.

As a final note, it should be mentioned that our study of transitive closure was restricted to the estimation of its cardinality. Once we are able to determine the expected number of solutions that result from the transitive closure of a predicate (that is equivalent to the original recursive predicate), we may propagate this value to the other "black boxes" in our model†. However, nothing has been said about the actual *cost* of executing the recursion or closure. In fact, there is not much to be said, other than it will be totally dependent on the actual implementation. The evaluation method that is used by the system to handle recursive queries will determine the cost of solving the recursion. Bancilhon and Ramakrishnan [Bancilhon86] have derived some formulae for several commonly used evaluation methods for some elementary forms of recursion. In fact, the design and evaluation of the performance of transitive closure algorithms has been an active area of research [Agrawal90].

We must add that some systems may decide to compute the transitive closure of a predicate only once and store the results for future use (instead of computing the closure any time that a user query requests it) [Cheiney94].‡ Additionally, it is not uncommon to use a pre-processor that transforms the original form of closure to another that is equivalent and more efficient to execute [Lu93].

5.6 Algorithm to Estimate the Cost of a GraphLog Query

We now formulate our general algorithm to estimate the cost of a given GraphLog query.

Input: a query q of the form

$q :- s_1, s_2, \ldots, s_m.$

and a calling pattern $cpat_q$.

---

†Recall that the only *input* quantity that a "black box" requires is the number of tuples retrieved by the previous black box.

‡Explicit storage of the transitive closure results minimizes access time and only requires a single (usually expensive) computation of the transitive closure.

Output: a cost estimate, $cost_q$, and the expected number of solutions to the query, num_sol_q.

General Algorithm:

```
algorithm estimate_cost(q, cpat_q, cost_q, num_sol_q);
/* assume that there are m subgoals in the query */
begin
    perform_mode_analysis(q, cpat_q, calling_pattern(s_1), ...,
  calling_pattern(s_m));
    for i:=1 to m do
    begin
       estimate_number_of_solutions(s_i, calling_pattern(s_i), num_sol_i);
       estimate_relevant_cost_metric(s_i, cost_i);
    end;
    num_sol_0:= 1; total_cost:=0; solutions:=1;
    for j:=1 to m do
    begin
       total_cost:= num_sol_j-1 × cost_j + total_cost;
       solutions:= solutions × num_sol_j;
    end;
    cost_q:= total_cost; num_sol_q:= solutions;
end;


procedure perform_mode_analysis(q, cpat_q, call_pat(s_1),...,call_pat(s_m));
begin
    This analysis determines the calling pattern of each subgoal s_i in the
      query†
end;


procedure estimate_number_of_solutions(s, calling_pattern(s), num_sol);
begin
    if s is an extensional DB predicate then use the database profile;‡
    if s is an intensional non-recursive DB predicate then
        estimate_numsol_nonrecursive(s, calling_pattern(s), num_sol);
    if s is an intensional recursive DB predicate then
       estimate_numsol_recursive(s, calling_pattern(s), num_sol);
end;
```

---

†See Section 3.6

‡See Sections 3.2 and 4.1

```
procedure estimate_relevant_cost_metric(s, calling_pattern(s), cost);
begin
    if s is an extensional DB predicate then use the database profile;
    if s is an intensional non-recursive DB predicate then
        estimate_cost_metric_nonrecursive(s, calling_pattern(s), cost);
    if s is an intensional recursive DB predicate then
        estimate_cost_metric_recursive(s, calling_pattern(s), cost);
end


procedure estimate_numsol_nonrecursive(s, cpat(s), num_sol);
/* assume that predicate s has n different clauses */
begin
    for k:=1 to n do
        estimate_cost(body(clause_k), cost_body_k, num_sol_k, h†(cpat(s)));
    estimate_number_of_solutions(num_sol, num_sol_1, ..., num_sol_n);‡
end;


procedure estimate_numsol_recursive(s, calling_pattern(s), num_sol);
begin
    transform the recursive predicate to an equivalent transitive closure;††
    estimate the number of solutions by applying properties of transitive
      closure;‡‡

end;


procedure estimate_cost_metric_nonrecursive(s, cost);

begin

/* assume that predicate s has n different clauses */
begin
    for k:=1 to n do
    begin
        estimate_cost(body(clause_k), cost_body_k, num_sol_k, h(cpat(s)));
        cost_k:= cost_hunif†††(clause_k, cpat(s))+
```

---

†h(cpat) is the calling pattern that is obtained after a successful head unification and is determined by a simple mode analysis

‡See Sections 3.5 and 4.3

††See Section 5.2

‡‡See Section 5.3

```
                P†(hunif(clause_k, cpat(s))) × cost_body_k;
    end;
    total_cost:=0;
    for k:=1 to n do
        total_cost:= total_cost + cost_k;
    cost:= total_cost;
end;


procedure estimate_cost_metric_recursive(s, cost);
begin
    estimate the cost based on knowledge about the recursive algorithm
      that is used;
end;
```

Most procedures can be performed mechanically once the abstraction of the database profile has been chosen. In general, we have assumed a uniform distribution of attribute values, but that does not have to be the case.

There are two procedures that pose some difficulties regarding their automation. One of them, the automatic transformation of a recursive query into an equivalent form of transitive closure, has just recently been addressed by researchers in the field [Consens89]‡. The other, the estimation of the cost contributors when a recursive predicate is solved, would imply a thorough analysis of the recursive algorithm in place. We will partially address this issue in the following chapter.

---

†††cost_hunif is the cost due to the process of head unification [Section 3.7.1].

†P(hunif(clause$_k$, cpat(s)) is the probability that the head unification is successful [Section 3.7.1].

‡See Section 5.2

Chapter 6. Some Case Studies

In this chapter, we will apply our framework to some typical databases. In particular, we are interested in showing how some real-life issues such as high correlation amongst attributes or duplication of tuples may affect the accuracy of the results. We also compare our results with one of the best algorithms for query reordering, namely Sheridan's algorithm [Sheridan91].

6.1 The congressional voting records database

This publicly available database contains the votes for each of the U.S. House of Representatives Congressmen on several key votes in the 1984 session (the so-called 1984 United States Congressional Voting Records Database). A very simple profile of this database is shown in Figure 6.1.† An extract of the corresponding GraphLog database is shown in Figure 6.2.

```
party(1,rep).      party(2,dem).                    party(435,rep).
project1(1,n).     project1(2,n).                   project1(435,n).
project2(1,y).     project2(2,y).                   project2(435,y).
project3(1,n).     project3(2,n).                   project3(435,n).
project4(1,y).     project4(2,y).                   project4(435,y).
project5(1,y).     project5(2,y).                   project5(435,y).
project6(1,y).     project6(2,y).                   project6(435,y).
project7(1,n).     project7(2,n).                   project7(435,n).
project8(1,n).     project8(2,n).                   project8(435,n).
project9(1,n).     project9(2,n).        • • •      project9(435,n).
project10(1,y).    project10(2,n).                  project10(435,y).
project11(1,a).    project11(2,n).                  project11(435,n).
project12(1,y).    project12(2,y).                  project12(435,y).
project13(1,y).    project13(2,y).                  project13(435,y).
project14(1,y).    project14(2,y).                  project14(435,y).
project15(1,n).    project15(2,n).                  project15(435,a).
project16(1,y).    project16(2,a).                  project16(435,n).
```

Figure 6.2 The GraphLog database

†In fact, there are three attribute values for each vote. The third attribute value (besides "yes" and "no" votes) can be regarded as an abstention.

Number of Instances: 435 (267 democrats, 168 republicans)

Number of Attributes: 16 + class name = 17 (all Boolean valued: y = yes; n= no; a = abstention)

Attribute Information:
1. Class Name: 2 attribute values (democrat, republican)
2. handicapped-infants: 2 attribute values (y,n)
3. water-project-cost-sharing: 2 attribute values (y,n)
4. adoption-of-the-budget-resolution: 2 attribute values (y,n)
5. physician-fee-freeze: 2 attribute values (y,n)
6. el-salvador-aid: 2 attribute values (y,n)
7. religious-groups-in-schools: 2 attribute values (y,n)
8. anti-satellite-test-ban: 2 attribute values (y,n)
9. aid-to-nicaraguan-contras: 2 attribute values (y,n)
10. mx-missile: 2 attribute values (y,n)
11. immigration: 2 attribute values (y,n)
12. synfuels-corporation-cutback: 2 attribute values (y,n)
13. education-spending: 2 attribute values (y,n)
14. superfund-right-to-sue: 2 attribute values (y,n)
15. crime: 2 attribute values (y,n)
16. duty-free-exports: 2 attribute values (y,n)
17. export-administration-act-south-africa: 2 attribute values (y,n)

Figure 6.1 The 1984 United States Congressional Voting Records Database

Example 1

Suppose that we wish to compare two orderings for a query that retrieves those individuals who voted "yes" on issue # 4 and "no" on issues # 3 and # 5. These two orderings are shown in Figure 6.3.

```
order1(Id,Party) :- project4(Id,y), party(Id,Party), project3(Id,n), project5(Id,n).
order2(Id,Party) :- party(Id,Party), project4(Id,y), project3(Id,n), project5(Id,n).
```

Figure 6.3 Two orderings that we wish to compare

We will assume that the GraphLog translator generates code for a system that uses first-argument indexing. Additionally, consider that we are mostly interested in making our decision based on the number of visited tuples only†.

†As it happens, for this particular example, the number of visited tuples (i.e., the number of subgoal unification attempts that take place) is indeed the most relevant contributor to the cost of executing this query

Let us consider ordering # 1 first:

```
order1(Id,Party) :- project4(Id,y), party(Id,Party), project3(Id,n), project5(Id,n).
```

A very simple analysis will determine the calling patterns for the different subgoals in this query as:

```
order1([f, f]) :- project4([f, g]), party([g, f]), project3([g, g]), project5([g, g]).
```

(*g* stands for "ground argument"; *f* represents a "free variable)". We proceed to estimate the cost (as the expected number of visited tuples) of each subgoal. Successively, we obtain:

- `project4([f, g])`

This predicate call will visit all 435 instances of this particular project. Only a fraction will actually succeed. In the absence of any additional information, we are forced to assume a particular distribution for the three attributes in the second argument, namely *y* ("yes"), *n* ("no") and *a* ("abstention"). For instance, we may decide to use a uniform distribution of independent values (so that we are expecting a fairly high number of abstentions!). Under this crude consideration, we would retrieve 435/3 (i.e., 145) tuples.

- `party([g, f])`

Now, we will visit as many *party* tuples as solutions we got from the previous subgoal. Our estimation would indicate that 145 tuples had to be visited (recall that first-argument indexing is assumed). The second argument poses no restriction whatsoever, so that all 145 tuples are expected to succeed.

- `project3([g, g]), project5([g, g])`

The final two subgoals are very similar from the point of view of our analysis. Since first-argument indexing occurs, a first argument ground will result in that, for each of the 145 tuples obtained from the previous phase, only one `project3` tuple has to be visited, with a 1/3 rate of success - roughly 48 tuples (recall our uniform assumption for the attribute). Similarly, for each successfully retrieved `project3` tuple, one `project5` tuple will be visited with a 1/3 rate of success as per our assumptions (approximately 16 tuples).

Let us turn our attention to ordering # 2:

```
order2(Id,Party) :- party(Id,Party), project4(Id,y), project3(Id,n), project5(Id,n).
```

Again, a simple analysis will determine the calling patterns for the different subgoals in this query as:

```
order2([f, f]) :- party([f, f]), project4([g, g]), project3([g, g]), project5([g, g]).
```

and we proceed to estimate the expected number of visited tuples for each subgoal on the right hand side.

- party([f, f])

This predicate call will visit all 435 instances of predicate party. Since both arguments are variable, all 435 instances will be retrieved as a result of the call.

- project4([g, g])

Now, we will visit as many project4 tuples as solutions we got from the previous subgoal (note that first-argument indexing is assumed). Our estimation would indicate that, from those 435 tuples, only a fraction will actually comply with the restriction posed by the second argument. If a uniform distribution of independent attribute values is assumed, roughly 435/3 (i.e., 145) tuples will be successfully retrieved.

- project3([g, g]), project5([g, g])

The analysis of the final two subgoals is identical to that for the alternative ordering. Since first-argument indexing occurs, a first argument ground will result in that, for each of the 145 tuples obtained from the previous phase, only one project3 tuple has to be visited, with a 1/3 rate of success - roughly 48 tuples, and, similarly, for each successfully retrieved project3 tuple, one project5 tuple will be visited with a 1/3 rate of success as per our assumptions (approximately 16 tuples).

Figure 6.4 show the abstract values for the different subgoals, represented by "black boxes". Figure 6.5 shows how these "black boxes" are interconnected to obtain the global values for the whole query.

The results of both analyses are sketched in Table 6.1 and Table 6.2.

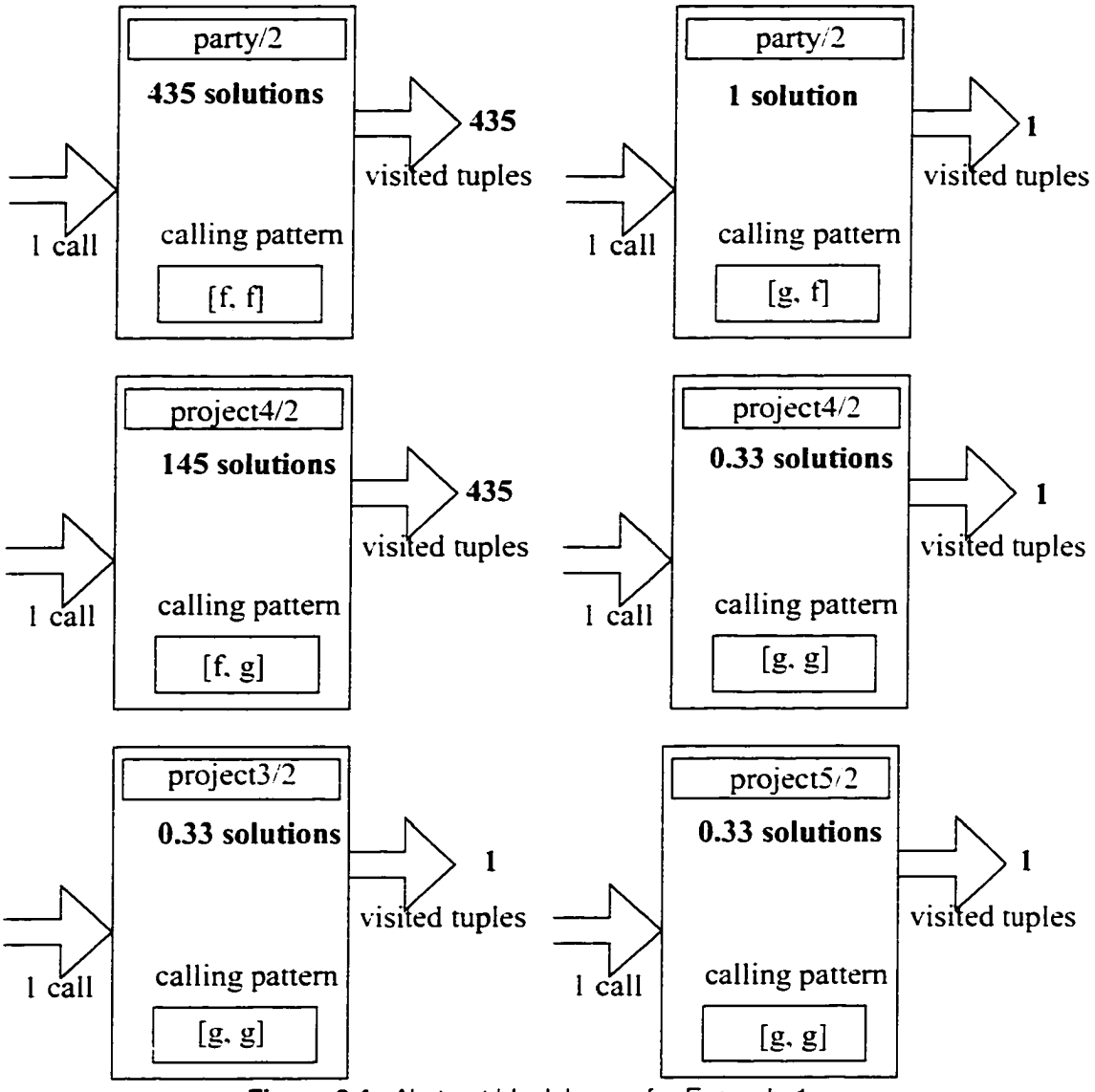
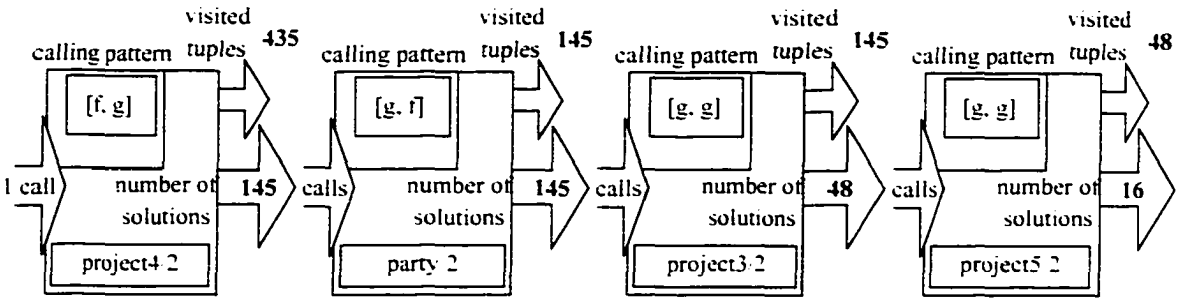


Figure 6.4 Abstract black boxes for Example 1

<clause, cpat>	number of visited tuples	expected number of solutions
<project4/2,[f,g],	435	145
<party/2,[g,f],	145	145
<project3/2,[g,g],	145	48
<project5/2,[g,g]>	48	16
TOTAL	773	16

Table 6.1 Number of visited tuples for ordering # 1

order1([f, f]) :- project4([f, g]), party([g, f]), project3([g, g]), project5([g, g]).



order2([f, f]) :- party([f, f]), project4([g, g]), project3([g, g]), project5([g, g]).

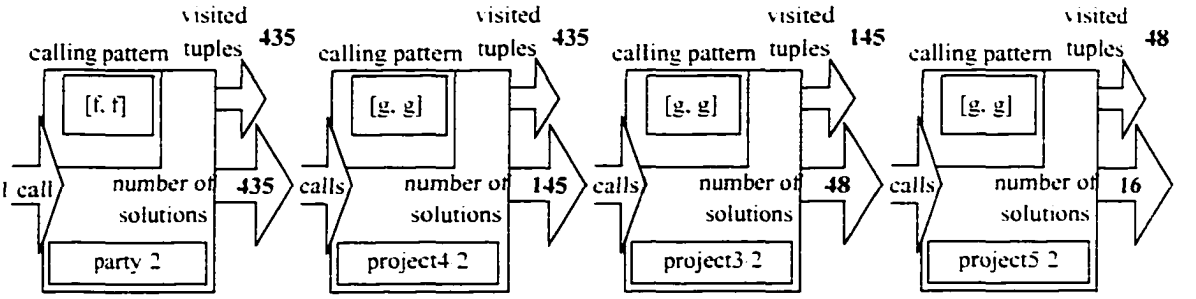


Figure 6.5 Interconnection of the black boxes for Example 1

<clause, cpat>	number of visited tuples	expected number of solutions
<party/2,[f,f],	435	435
<project4/2,[g,g],	435	145
<project3/2,[g,g],	145	48
<project5/2,[g,g]>	48	16
TOTAL	1063	16

Table 6.2 Number of visited tuples for ordering # 2

We may conclude that we expect ordering # 1 to be a better option with respect to ordering # 2 (which has a 38% additional cost). Experimental results for this query confirm our prediction. These results, using SICStus Prolog version 2.1 on a Sun SPARCstation SLC, are shown in Table 6.1 and Table 6.2. From the figures, ordering # 2 is 43%

more expensive than ordering #1. Cost measurements are given in Prolog's artificial units.

```
order1(Id,Party) :- project4(Id,y), party(Id,Party), project3(Id,n), project5(Id,n).
order2(Id,Party) :- party(Id,Party), project4(Id,y), project3(Id,n), project5(Id,n).
```

ordering	average cost (1000 experiments)	number of solutions
order1	58.2	4
order2	83.4	4

Figure 6.6 Experimental results for both orderings

Although our assumption of a uniform distribution is clearly inaccurate, we still can predict which ordering will be less expensive to execute (from the point of view of visited tuples). Our estimate of the number of solutions is obviously poor, but only a more detailed profile would yield better results.

As an additional note, we must mention that this particular database has a very high correlation factor amongst attributes. In other words, "republicans" are expected to vote as a block on some (if not most) issues, and so are "democrats". If we pose a query of the form:

```
:- party(Id,dem), project4(Id,y), project3(Id,n), project5(Id,n).
```

requesting those democrats that voted some way, we should not be surprised to find that our estimates regarding the number of successful tuples that are retrieved are even less accurate.

Example 2

Suppose that we wish to compare all orderings for a query that retrieves those individuals that voted "yes" on issue 16 and "no" on issue 6. These orderings are shown in Figure 6.7.

```
order1(Id,Party) :- party(Id,Party), project16(Id,y), project6(Id,n).
order2(Id,Party) :- party(Id,Party), project6(Id,n), project16(Id,y).
order3(Id,Party) :- project16(Id,y), party(Id,Party), project6(Id,n).
order4(Id,Party) :- project16(Id,y), project6(Id,n), party(Id,Party).
order5(Id,Party) :- project6(Id,n), party(Id,Party), project16(Id,y).
order6(Id,Party) :- project6(Id,n), project16(Id,y), party(Id,Party).
```

Figure 6.7 Six orderings that we wish to compare

A very simple static analysis determines the corresponding calling patterns which are shown in Figure 6.8.

```
order1[f,f] :- party[f,f], project16[g,g], project6[g,g].
order2[f,f] :- party[f,f], project6[g,g], project16[g,g].
order3[f,f] :- project16[f,g], party[g,f], project6[g,g].
order4[f,f] :- project16[f,g], project6[g,g], party[g,f].
order5[f,f] :- project6[f,g], party[g,f], project16[g,g].
order6[f,f] :- project6[f,g], project16[g,g], party[g,f].
```

Figure 6.8 Six orderings that we wish to compare

We wish to estimate the number of visited tuples associated with each predicate/calling pattern pair that appears in the different orderings. Following a similar reasoning to that in Example 1, we are able to deduce the values shown in Figure 6.9 in which we may use "black boxes" to identify the abstract values that we estimate. Note that first-argument indexing is assumed. These "black boxes" are then interconnected as shown in Figure 6.10 to obtain the global values of an entire query.

We are now in a position to estimate the total number of tuples expected for each ordering. We summarize the (analytical) results in Table 6.3. Experimental results for SICStus Prolog are shown in Table 6.4, where they are compared to our estimated values.

We were able to detect the two least efficient orderings. However, our predictions regarding the other four orderings are not very accurate. This is due to the fact that we are assuming a distribution that behaves in a certain way, whereas the real database follows a different pattern. Predicate project16 clearly has a different behaviour from that of predicate project6 (in fact, the latter predicate gets executed more efficiently than the former), and our framework cannot make this distinction in the absence of a more de-

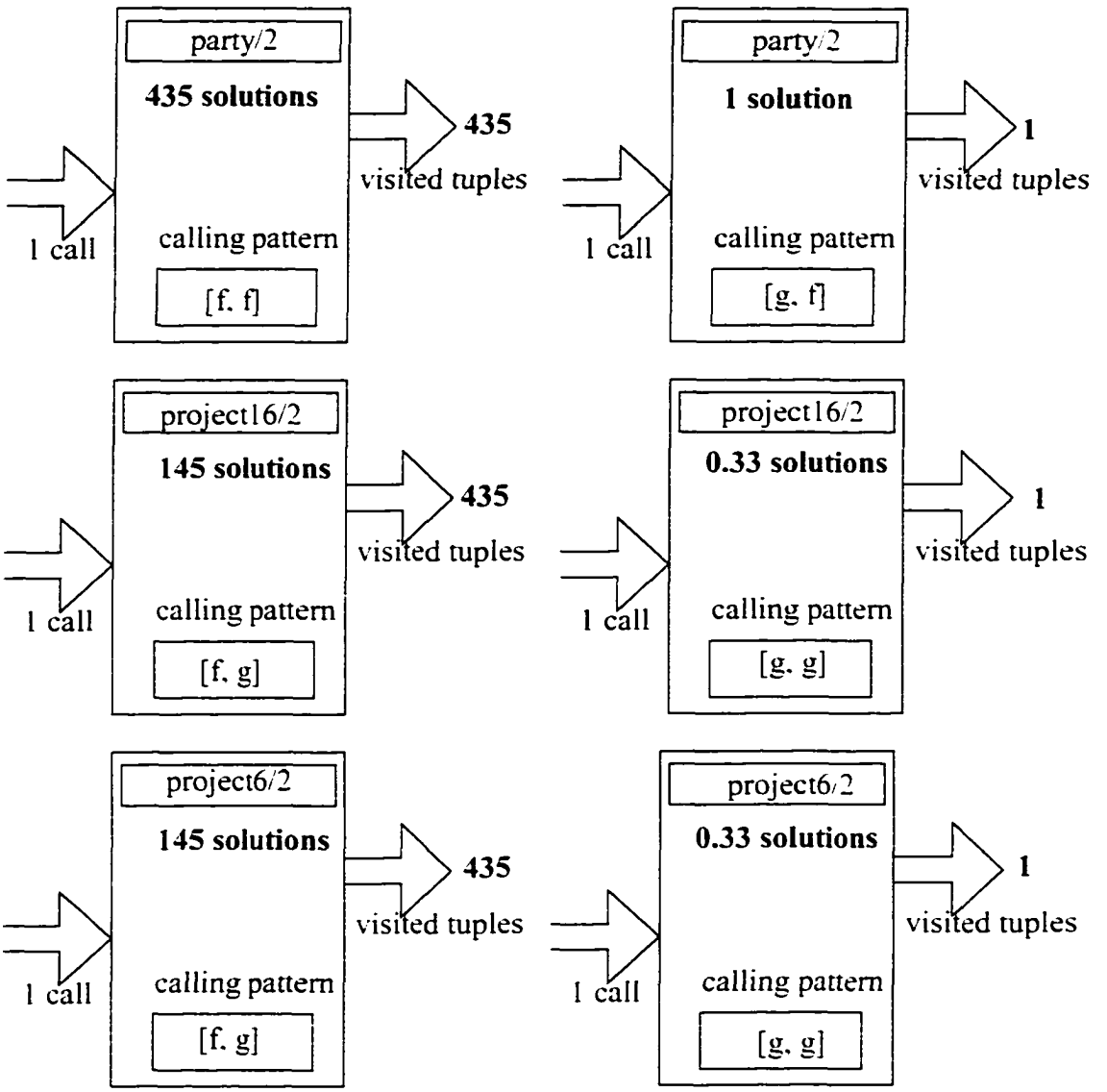


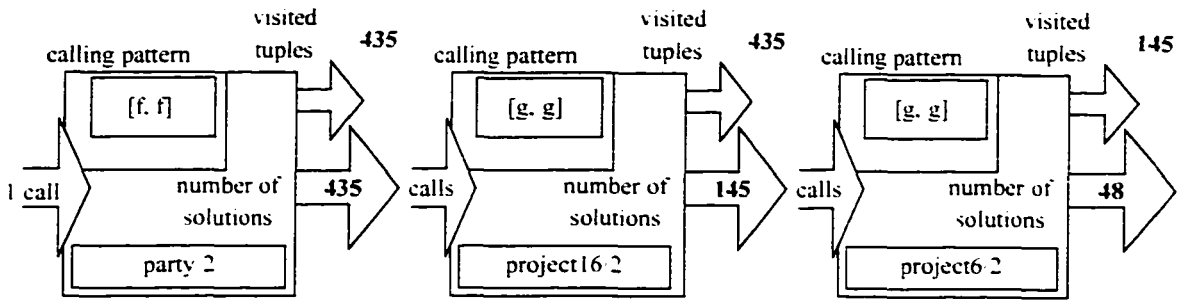
Figure 6.9 Abstract black boxes for Example 2

tailed database profile. But, our framework is able to detect that if a "project" predicate is selected first, it is more efficient to place the other "project" as the next predicate, leaving the party predicate to the last position. Similarly, our framework determines that it is not convenient to place the party predicate as the first subgoal in the clause.

6.2 The Performers Database

We proceed to study another real database. Our second database contains detailed information of 888 classical-music compact discs. The information available for each com-

order1[f,f] :- party[f,f], project16[g,g], project6[g,g].



order3[f,f] :- project16[f,g], party[g,f], project6[g,g].

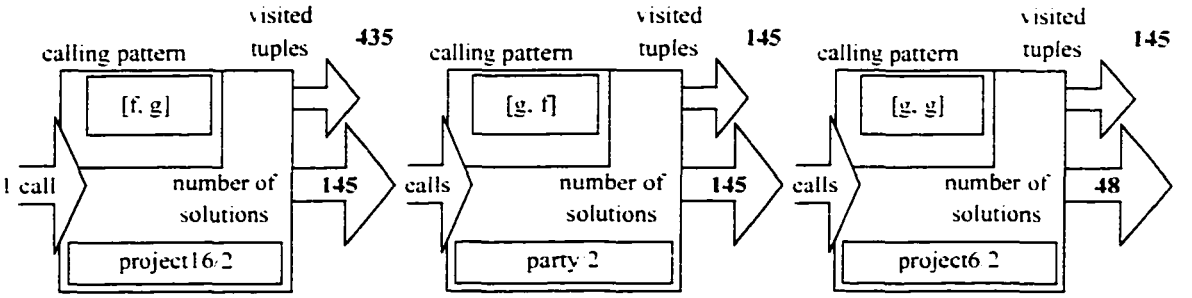


Figure 6.10 Interconnection of two black boxes in Example 2

ordering	estimated number of visited tuples
order1[f,f] :- party[f,f], project16[g,g], project6[g,g].	$435 + 435 \times (1 + 0.33 \times (1)) = 1013.55$
order2[f,f] :- party[f,f], project6[g,g], project16[g,g].	$435 + 435 \times (1 + 0.33 \times (1)) = 1013.55$
order3[f,f] :- project16[f,g], party[g,f], project6[g,g].	$435 + 145 \times (1 + 1 \times (1)) = 725.00$
order4[f,f] :- project16[f,g], project6[g,g], party[g,f].	$435 + 145 \times (1 + 0.33 \times (1)) = 627.85$
order5[f,f] :- project6[f,g], party[g,f], project16[g,g].	$435 + 145 \times (1 + 1 \times (1)) = 725.00$
order6[f,f] :- project6[f,g], project16[g,g], party[g,f].	$435 + 145 \times (1 + 0.33 \times (1)) = 627.85$

Table 6.3 Expected number of visited tuples

pact disc includes a list of individual tracks, a list of individual and collective performers, as well as some technical data regarding the production of the recording. For the purposes of our case study, we will consider the portion of the database that is related to the musicians and their instruments.

ordering	experimental value	experimental ranking	theoretical ranking
order1[f,f] :- party[f,f], project16[g,g], project6[g,g].	83.0	[6]	[5=]
order2[f,f] :- party[f,f], project6[g,g], project16[g,g].	72.2	[5]	[5=]
order3[f,f] :- project16[f,g], party[g,f], project6[g,g].	64.1	[4]	[3=]
order4[f,f] :- project16[f,g], project6[g,g], party[g,f].	50.6	[3]	[1=]
order5[f,f] :- project6[f,g], party[g,f], project16[g,g].	43.0	[2]	[3=]
order6[f,f] :- project6[f,g], project16[g,g], party[g,f].	39.4	[1]	[1=]

Table 6.4 Comparison between the predicted and experimental values

6.2.1 Primitive Entities

The main entities to be considered are: (1) compact disc numbers, (2) artists or musicians, (3) instruments used by the musicians and (4) compact disc labels. To produce a more interesting example, we will introduce an additional entity, namely: (5) the overseas distributors for the compact discs.

Compact Disc Numbers

Every compact disc can be identified by a manufacturer's number, which is an alphanumeric code. Each compact disc number is then assigned an internal code to be used by other relations:

```
recording(CD_Code, Manufacturer_Number)
```

Artists

Each individual performer or musical ensemble in the database is identified by an internal code.

```
artist(Performer_Code, Performer_Name).
```

Instruments

Similarly, every instrument description has a unique code assigned to it. Different descriptions for the same instruments are treated as different entities.

```
instrument(Instrument_Code, Instrument_Name).
```

Companies

We also find entries for the different companies that produce the compact discs stored in the database. Again, a specific code is provided for each label.

```
company(Label_Code, Label_Name).
```

6.2.2 The Extensional Database

Once we have introduced the main entities in the database, we proceed to explain the set of facts that conform the performers database. We will consider the following relations, available as extensional DB predicates:

Performers

For each compact disc represented in the database, a list of performers is available. For a given compact disc code, there is one entry for each performer listed for that production. If the same artist utilizes more than one instrument, there is one separate entry for each instrument used by that performer.

```
performer(CD_Code, Performer_Code, Instrument_Code).
```

Labels

This relation gives the internal code for the company that has produced the compact disc.

```
label(CD_Code, Label_Code).
```

Distributors

Finally, each label may have one or more "overseas distributors", that is, independent companies that import and distribute the compact discs in different parts of the world.

```
distributor(Label_Code, Distributor).
```

Typical sample tuples of the different relations in the performers database are sketched in Figure 6.11.

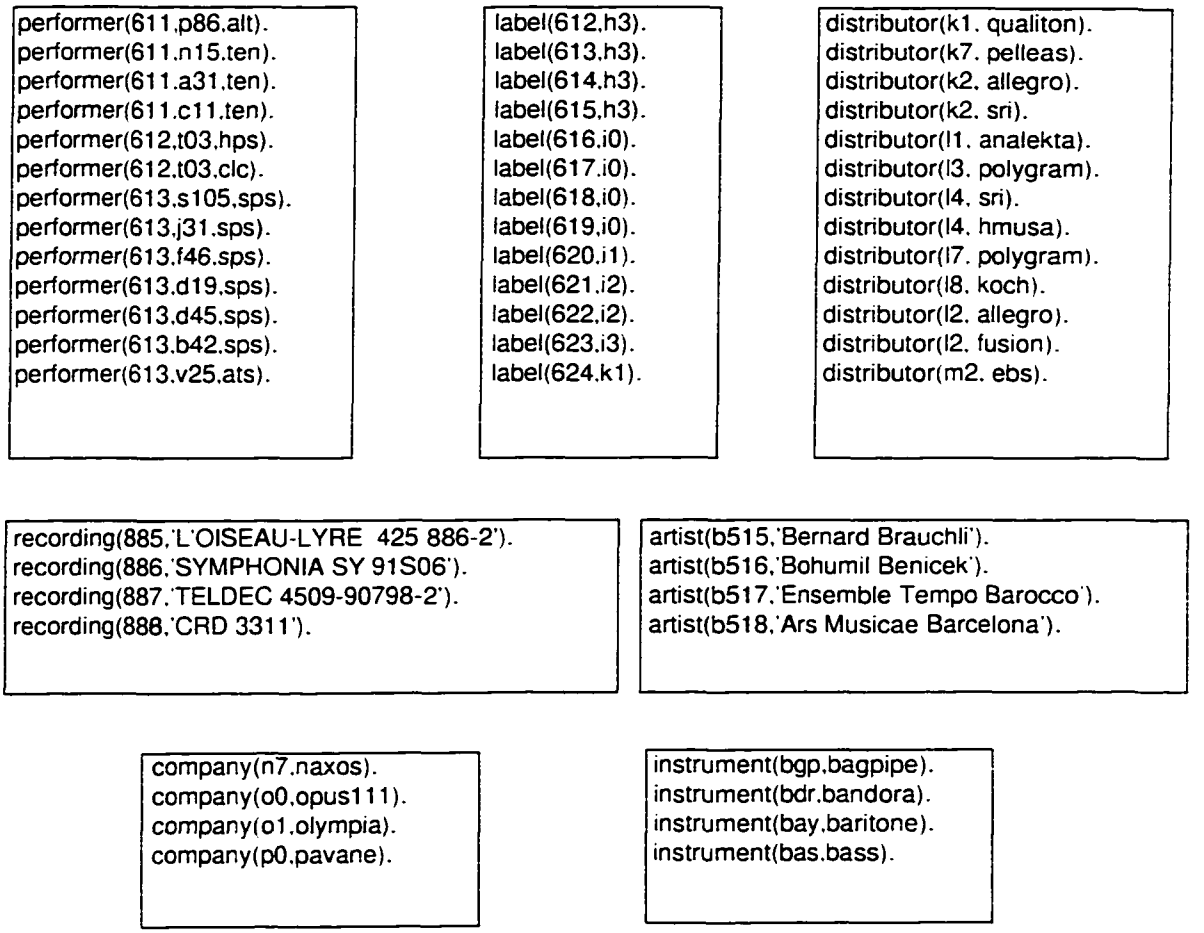


Figure 6.11 Sample tuples from the performers database

6.2.3 A Non-recursive Query

We start by analyzing a simple non-recursive query. For instance, we may be interested in knowing the codes of the performers of some particular instrument that are available from a given overseas distributor. The following Datalog predicate defines such a relationship:

```
instrumentists_available_from_a_distributor(A,D,I) :- performer(R,A,I), label(R,L), distributor(L,D)
```

Suppose that we are interested in the following family of queries:

```
:- instrumentists_available_from_a_distributor(A,a_particular_distributor, a_particular_instrument).
```

In other words, we will study a query with a calling pattern [f, g, g]: the first argument is a variable, whereas the second and third arguments are constants. There are six different orderings in which the three subgoals may be arranged (calling patterns are shown in square brackets):

performer [f,f,g], label [g,f], distributor [g,g]
performer [f,f,g], distributor [f,g], label [g,g]
label [f,f], performer [g,f,g], distributor [g,g]
label [f,f], distributor [g,g], performer [g,f,g]
distributor [f,g], performer [f,f,g], label [g,g]
distributor [f,g], label [f,g], performer [g,f,g]

Our goal consists of selecting the most efficient ordering of all six. All we know about the user's query is the calling pattern: instrumentists_available_from_a_distributor [f, g, g].

Typical queries are shown as follows:

```
:- instrumentists_available_from_a_distributor(A, sri, ten).
```

(tenors on a label distributed by Scandinavian Record Imports):

```
:- instrumentists_available_from_a_distributor(A, qualiton, sop).
```

(sopranos on a label distributed by Qualiton Imports):

```
:- instrumentists_available_from_a_distributor(A, allegro, obo).
```

(oboists on a label distributed by Allegro Imports).

The database profile of the performers database is shown in Table 6.5.

Predicate name	number of tuples	distinct values in argument 1	distinct values in argument 2	distinct values in argument 3
performer/3	12,851	888	3,727	710
label/2	888	888	92	—
distributor/2	177	111	23	—

Table 6.5 The performers database predicates

In the absence of further information, we will treat the database as a uniform and independent distribution of attribute values (although we know that this is probably not the case). We will try to assign cost values to all six different possible orderings for the subgoals in the predicate definition. We will consider a couple of cost contributors in our analysis: number of visited tuples and number of variable unifications that are performed.

Since we use SICStus Prolog to execute the code generated by the GraphLog translator, we have to assume that first-argument indexing is used. Our abstract "black boxes" for all (statically detected) combinations of calling patterns for the three predicates are shown in Figure 6.12.

The expected average number of tuples is either the total number of tuples for that predicate if the first argument is a variable, or this value divided by the number of distinct values in the first argument position otherwise. The total number of variable unifications may be calculated with the aid of the formula that is shown in Appendix 2, or if a simpler approximation is considered, by multiplying the number of expected visited tuples by the number of variable arguments in the predicate call. Finally, the expected average number of solutions is calculated by dividing the total number of tuples by the number of distinct values at each ground argument position.

Once we have determined the expected values for our cost contributors when a single call is considered, we proceed to "interconnect" all three predicates, for each ordering under consideration. This is illustrated in Figure 6.13 for only one of the specific orderings.

Table 6.6 shows the values that are estimated for both cost contributors given all six different orderings. Table 6.7 summarizes the corresponding experimental results for different sets of ground terms. From these tables, we observe that we accurately predict the best ordering, as well as the two worst orderings. Interestingly enough, the ordering that we expect to be the second most efficient one (i.e., ordering #6), is only so for a few of the experiments. In fact, for some distributors that carry many labels (sri, qualiton, allegro), orderings #1 and #3 seem to be more efficient. We must realize that our predic-

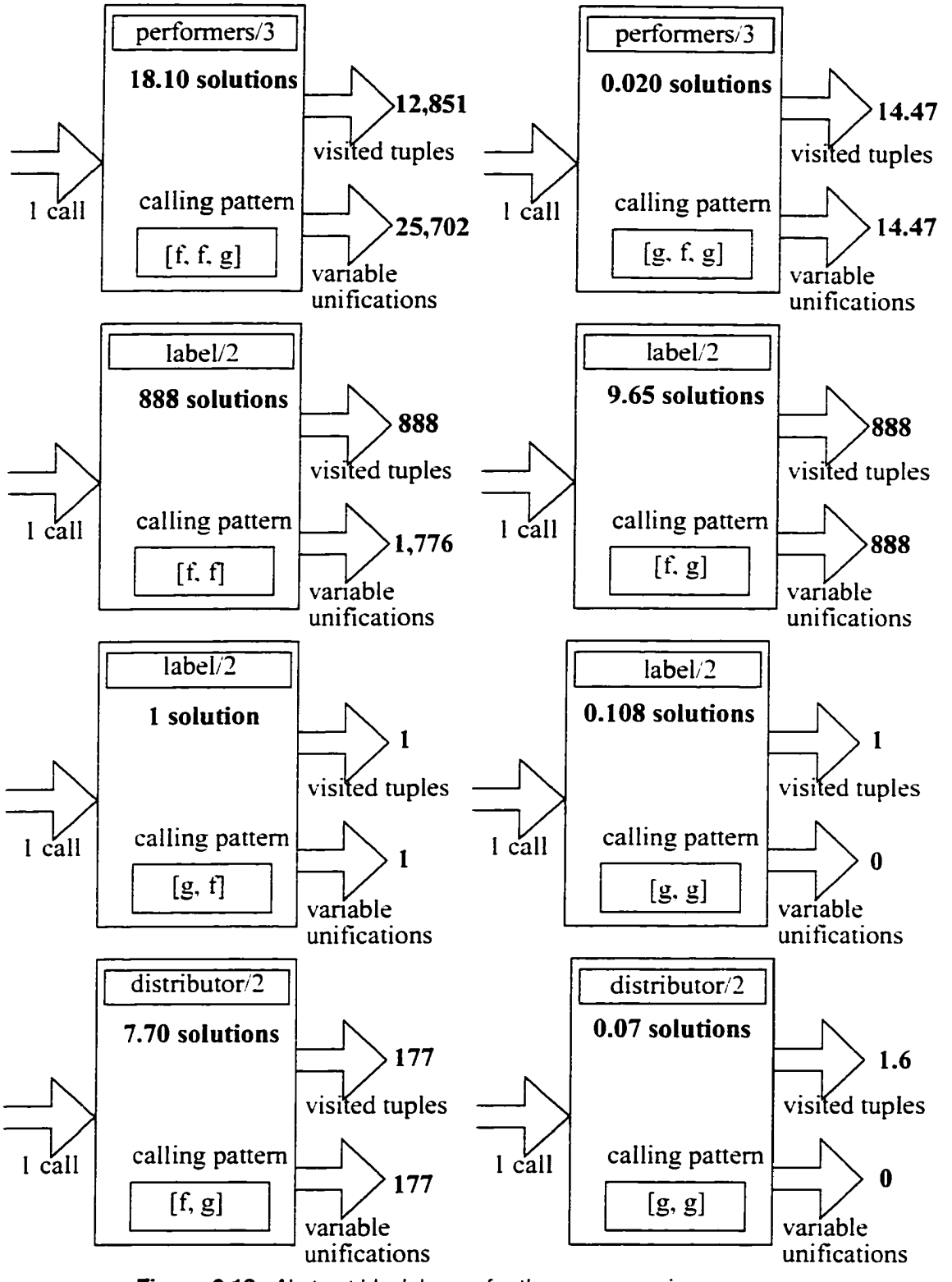


Figure 6.12 Abstract black boxes for the non-recursive query

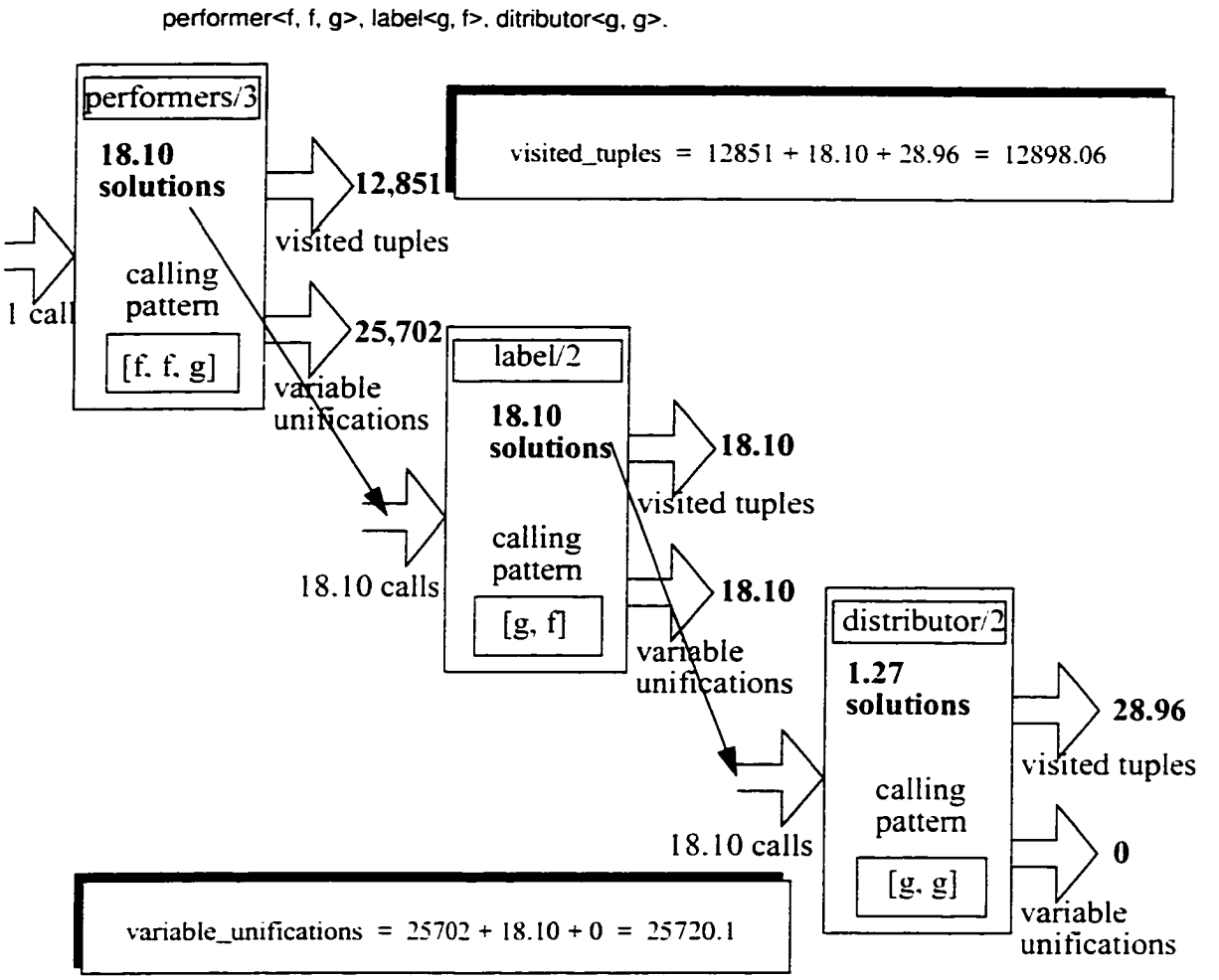


Figure 6.13 Expected values for the cost contributors for a specific ordering

tions are based on uniform distributions of independent attribute values, and the database does not follow this type of distribution. However, we are still able to select efficient orderings and reject those that perform poorly.

6.2.4 An Example Involving a Closure

We will study an example that includes a closure predicate, which requires special treatment in our framework. For practical reasons, we had to use a smaller database, because the current implementation of the transitive closure algorithm is inefficient. The modified database profile for the performers predicate is shown in Table 6.8.

#	ordering	expected number of visited tuples n_{vt}	ranking for n_{vt}	expected number of variable unifications n_{vu}	ranking for n_{vu}
1	performer, label, distributor	12898	[3]	25720	[4]
2	performer, distributor, label	16194	[5]	28906	[5]
3	label, performer, distributor	15764	[4]	16623	[3]
4	label, distributor, performer	3208	[1]	2676	[1]
5	distributor, performer, label	99269	[6]	197913	[6]
6	distributor, label, performer	8095	[2]	7926	[2]

Table 6.6 Predicted values of two cost contributors for the non-recursive query

#	ordering	A, sri, ten	A, qualiton, sop	A, allegro, obo	A, fusion, ten	A, ebs, sop	A, hmusa, sop
1	performer, label, distributor	355 [2]	369 [2]	296 [2]	314 [3]	319 [3]	387 [3]
2	performer, distributor, label	2072 [5]	2406 [5]	940 [5]	1683 [5]	1993 [5]	1990 [5]
3	label, performer, distributor	392 [3]	405 [3]	348 [3]	391 [4]	394 [4]	463 [4]
4	label, distributor, performer	235 [1]	216 [1]	137 [1]	114 [2]	123 [1]	267 [1]
5	distributor, performer, label	6197 [6]	8874 [6]	6932 [6]	2045 [6]	5118 [6]	4166 [6]
6	distributor, label, performer	475 [4]	599 [4]	438 [4]	111 [1]	270 [2]	362 [2]

Table 6.7 Experimental results for the non-recursive query (rankings in square brackets)

Predicate name	number of tuples	distinct values in argument 1	distinct values in argument 2	distinct values in argument 3
performers/3	204	9	132	45

Table 6.8 The modified performers database profile

We now proceed to define some additional relations to be applied to the performers database. We are interested in the definition of a predicate that uses some form of recursion or closure.

Let us define a predicate colleague that is true when two musicians participate in the same recording production:

```
colleague(A,B) :- performer(X,A,_), performer(X,B,_).†
```

†Strictly speaking, we should also impose the restriction that A and B are different musicians. We will omit this additional constraint here and in future definitions to simplify the analysis.

We also define that musician *A* is an "indirect" colleague of a musician *B* if both have recorded at least one compact disc with a mutual "colleague" as defined before:

```
colleague_of_a_colleague(A,B) :- colleague(A,B)+.
```

and therefore a transitive closure is used.

Further suppose that we wish to define a more specific type of "indirect" colleague, in which musician *A* has participated in a recording project performing the *same* instrument as musician *B*, and both having recorded with a mutual "colleague" of the same instrument. This could be done by defining a predicate that includes the additional restriction of the musicians performing on same instrument:

```
colleague_same_instrument(A,B) :- performer(X,A,I), performer(X,B,I).
```

and then taking the closure over this predicate. However, we will use a different set of predicates for illustrative purposes (i.e., showing how closure predicates are handled by our framework).

Thus, we define a "same-instrument" indirect colleague *A* as a musician who has participated in a recording project performing the same instrument as another musician *B*, and both have recorded with a mutual "colleague of a colleague" (as defined above) as:

```
same_instrument_colleague_of_a_colleague(A,B) :-
        performer(_,A,I), performer(_,B,I), colleague_of_a_colleague(A,B).
```

where the last subgoal is a transitive closure.

We observe that there are six different orderings for the right-hand side of the predicate.

```
performer(_,A,I), performer(_,B,I), colleague_of_a_colleague(A,B)
performer(_,A,I), colleague_of_a_colleague(A,B), performer(_,B,I)
performer(_,B,I), performer(_,A,I), colleague_of_a_colleague(A,B)
performer(_,B,I), colleague_of_a_colleague(A,B), performer(_,A,I)
colleague_of_a_colleague(A,B), performer(_,A,I), performer(_,B,I)
colleague_of_a_colleague(A,B), performer(_,B,I), performer(_,A,I)
```

If, say, the calling pattern for the predicate same_instrument_colleague_of_a_colleague is known to be [g, f], the calling patterns of the predicate subgoals for all six orderings are as follows†:

performer [f,g,f], performer [f,f,g], colleague_of_a_colleague [g,g]
performer [f,g,f], colleague_of_a_colleague [g,f], performer [f,g,g]
performer [f,f,f], performer [f,g,g], colleague_of_a_colleague [g,g]
performer [f,f,f], colleague_of_a_colleague [g,g], performer [f,g,g]
colleague_of_a_colleague [g,f], performer [f,g,f], performer [f,g,g]
colleague_of_a_colleague [g,f], performer [f,g,f], performer [f,g,g]

As mentioned before, predicates that involve closures or recursion have to be treated as special black boxes: unless we know the exact algorithm that is being used to solve the closure or recursion, nothing can be said about the values of the cost contributors for the predicate, except for the expected number of solutions (a value that is propagated to other black boxes when we interconnect them to find the global values of the cost contributors).

Since the base predicate for this closure is given by the following two subgoals:

performer(X,A,I), performer(X,B,I)

we may estimate the average number of tuples as $204 \times (204 \quad 9) \approx 4624$, since there are 204 performer facts and 9 label facts (i.e., recordings) in the database.

The average number of solutions of the closure predicate is estimated by noting that the number of distinct attribute values for the relation is $n_{at} = 132 \times 132 = 17424$ (132 is the number of performers), and the number of unique tuples for the base predicate is estimated as $n_b = 4624$. We then determine the region that corresponds to these values: their ratio is calculated as $A = n_{at} \; n_b = 3.77$. This corresponds to the "region of small values for the number of tuples in the base predicate" as explained in Section 5.4.1. Applying the formula mentioned in that section, we estimate the number of solutions of the closure as $\overline{n_{tc}} \approx 17424 / (3.77 - 1) \approx 6290$. This is the expected number of tuples for the whole closure. If one argument is ground, only a fraction of the tuples will form part of the solution. We already know that the tuples produced by a transitive closure do not fol-

†Note that the two last orderings are equivalent in the abstract domain.

low a uniform distribution (See Section 5.5), but we may produce a rough estimate by making this assumption, thus dividing the total number of tuples by the number of distinct values for that particular ground argument (i.e., the number of performers, in our example), or $\overline{n_{tc}}[f, g] = \overline{n_{tc}}[g, f] \approx 6290 / 132 \approx 47.7$. Similarly, if both arguments are ground, our estimate would be: $\overline{n_{tc}}[g, g] \approx 47.7 \quad 132 \approx 0.36$. Thus, our black boxes for this example are shown in Figure 6.14.

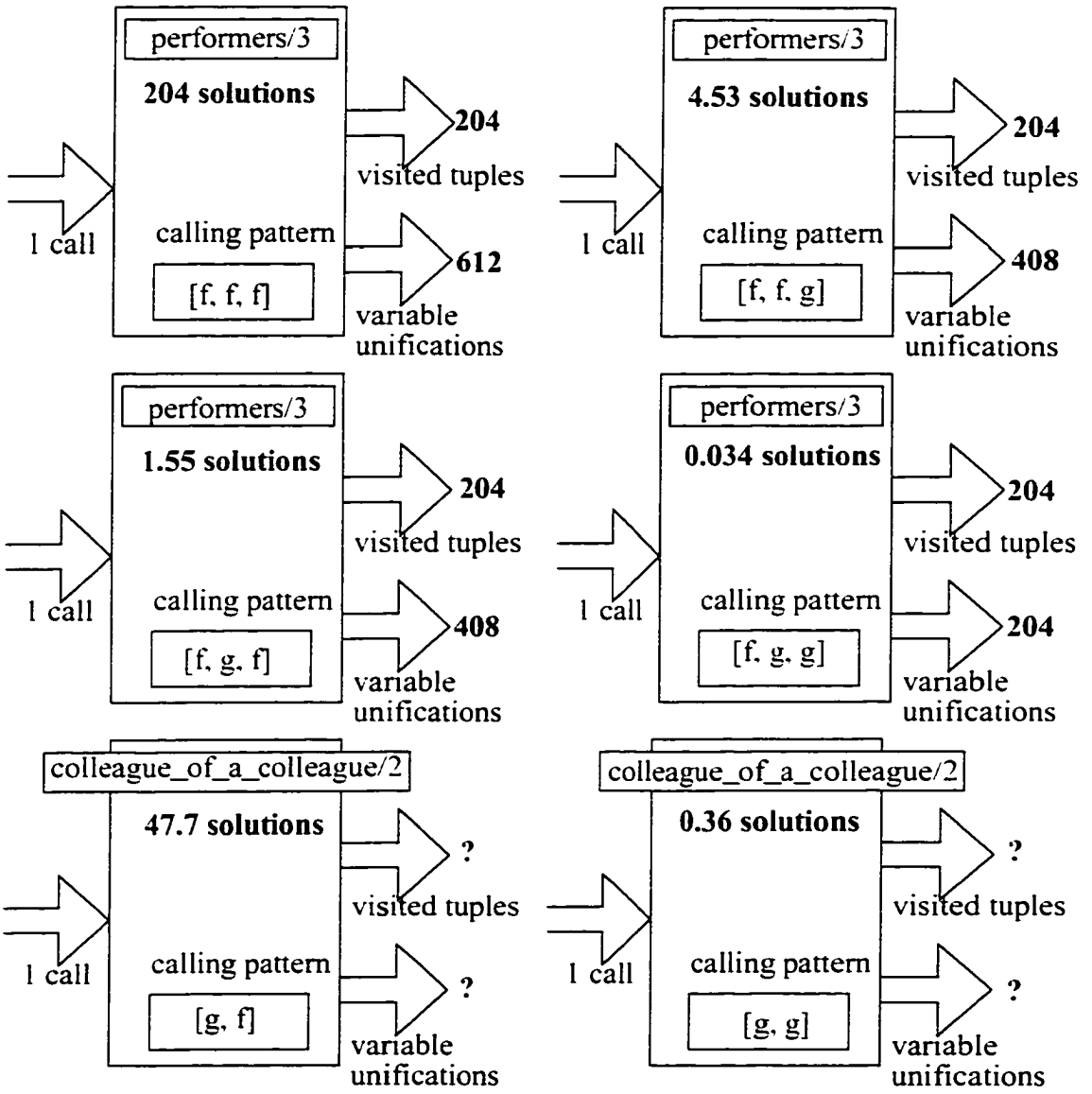


Figure 6.14 Abstract black boxes for the recursive query

Suppose that we wish to use the number of visited tuples as the relevant cost contributor. The abstract representation of our six orderings would be as shown in Figure 6.15. We have used the names N_{gf} and N_{gg} to denote the unknown number of visited tuples associated with the closure predicate with calling patterns [g, f] and [g, g], respectively.

Thus, the expected number of visited tuples for the different orderings are obtained as:

performer [f,g,f], performer [f,f,g], colleague_of_a_colleague [g,g]: visited tuples = 520.2 + 7.02 N_{gg}

performer [f,g,f], colleague_of_a_colleague [g,f], performer [f,g,g]: visited tuples = 15286.7 + 1.55 N_{gf}

performer [f,f,f], performer [f,g,g], colleague_of_a_colleague [g,g]: visited tuples = 41820 + 6.94 N_{gg}

performer [f,f,f], colleague_of_a_colleague [g,g], performer [f,g,g]: visited tuples = 15186.8 + 204 N_{gg}

colleague_of_a_colleague [g,f], performer [f,g,f], performer [f,g,g]: visited tuples = 24813.5 + N_{gf}

colleague_of_a_colleague [g,f], performer [f,g,f], performer [f,g,g]: visited tuples = 24813.5 + N_{gf}

At this point, we need estimates of the magnitudes of N_{gf} and N_{gg}. Unless we have a detailed performance analysis of the algorithm that is used to execute the transitive closure, we are forced to propose some suitable value. For instance, we know that simple algorithms to solve the transitive closure problem [Warren75] are computed in time at most proportional to the cube of the size of n, the number of distinct values in the relation, or $\Theta(n^3)$ [Baase88].

We also know that typical transitive closure algorithms compute practically the same instructions for calling patterns [g, g] and [g, f] [Fukar91]. This is because the algorithm first obtains all pairs that are reachable from the first argument, and only then the nature of the second argument is taken into account. In fact, many transitive closure algorithms have similar behaviour even for the calling pattern [f, g], since the computation is started from the second argument rather than from the unbound first argument.

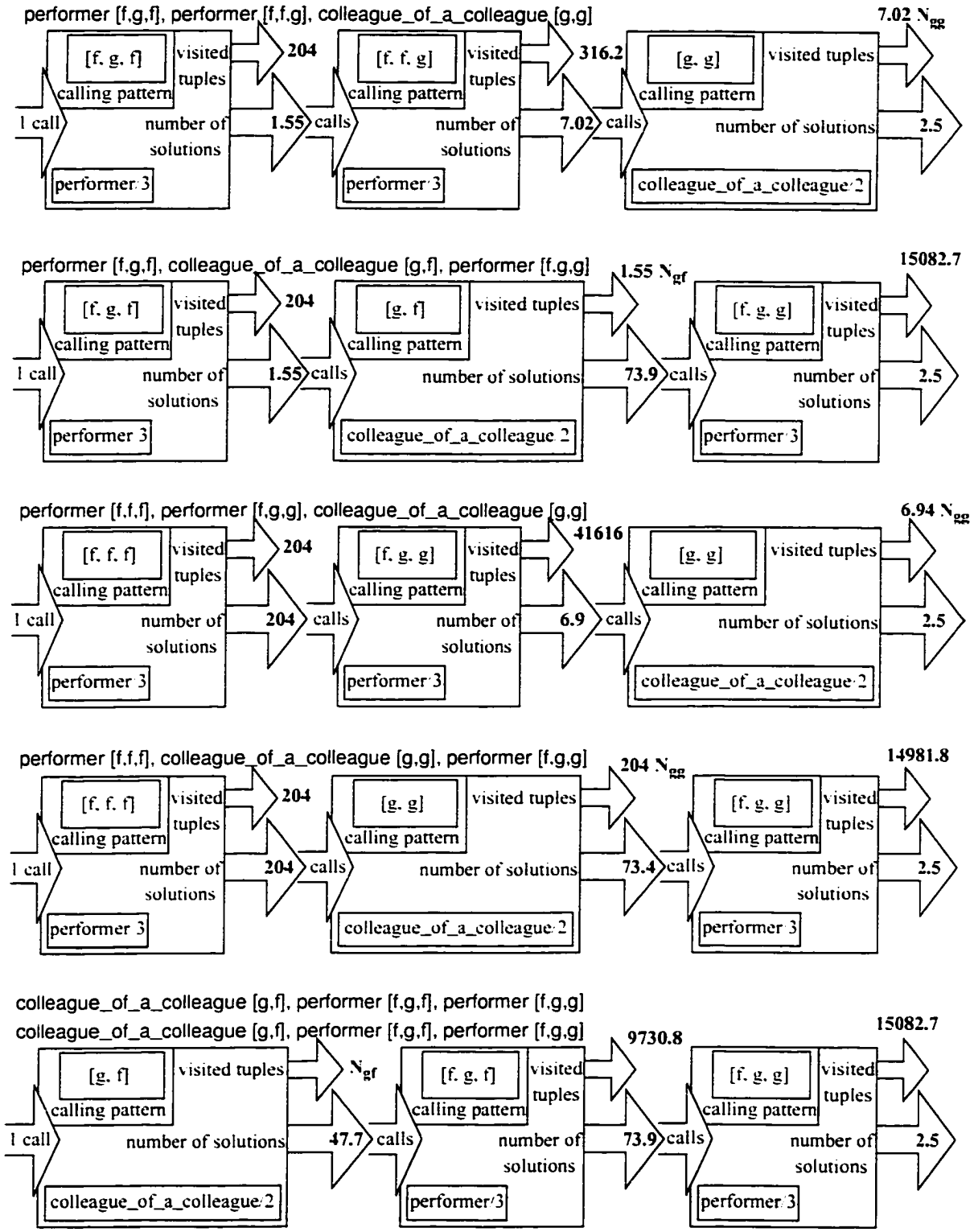


Figure 6.15 Abstract representation of the different orderings

One educated guess as to the values of N_{gf} and N_{gg} would be to use the cube of the number of tuples in the base relation ($4624^3 = 98 \times 10^9$, in our example).† Given this value, our estimates become (rankings shown in square brackets):

performer [f,g,f], performer [f,f,g], colleague_of_a_colleague [g,g]:	visited tuples = 694×10^9 [4]
performer [f,g,f], colleague_of_a_colleague [g,f], performer [f,g,g]:	visited tuples = 153×10^9 [3]
performer [f,f,f], performer [f,g,g], colleague_of_a_colleague [g,g]:	visited tuples = 686×10^9 [5]
performer [f,f,f], colleague_of_a_colleague [g,g], performer [f,g,g]:	visited tuples = 20×10^{12} [6]
colleague_of_a_colleague [g,f], performer [f,g,f], performer [f,g,g]:	visited tuples = 99×10^9 [1]
colleague_of_a_colleague [g,f], performer [f,g,f], performer [f,g,g]:	visited tuples = 99×10^9 [1]

Typical experimental results for this family of queries when using SICStus Prolog as the target language are shown in Table 6.9 (again, rankings are shown in square brackets). Note that our first choice for the most efficient query ordering is close to the one that is experimentally best (in fact, there is a virtual "tie" amongst the three orderings with best experimental performance). We are also able to discover those orderings that are more inefficient and therefore should be discarded. Not surprisingly, the most significant term is the one that relates to the closure predicate.

ordering	experimental results	
performer [f,g,f], performer [f,f,g], colleague_of_a_colleague [g,g]:	127918.0	[4]
performer [f,g,f], colleague_of_a_colleague [g,f], performer [f,g,g]:	43040.0	[1]
performer [f,f,f], performer [f,g,g], colleague_of_a_colleague [g,g]:	128141.0	[4=]
performer [f,f,f], colleague_of_a_colleague [g,g], performer [f,g,g]:	7496210.0	[6]
colleague_of_a_colleague [g,f], performer [f,g,f], performer [f,g,g]:	43330.0	[1=]
colleague_of_a_colleague [g,f], performer [f,g,f], performer [f,g,g]:	43472.0	[1=]

Table 6.9 Experimental results for the recursive predicate

Incidentally, the ratio between the most and less efficient orderings in the experiments is $7496/43 \approx 174.3$; the corresponding ratio in our predictions is $20000/99 \approx 202.0$. Our "educated guess" turned out to be reasonably accurate.

†As mentioned before, both calling patterns require visits to similar number of tuples during the computation of the closure: the only difference is that the calling pattern [g, g] will result in fewer tuples to be kept in the final answer.

We also launched a series of experiments to determine the performance of the transitive closure algorithm used by the GraphLog translator. The results when SICStus Prolog is used are shown in Table 6.10 (data values are expressed in "artificial units").

calling pattern	execution time
colleague_of_a_colleague[f, f]	5769549.0
colleague_of_a_colleague[g, f]	41943.5
colleague_of_a_colleague[f, g]	62922.5
colleague_of_a_colleague[g, g]	42143.5

Table 6.10 Efficiency of the transitive closure for different calling patterns

Not surprisingly, the efficiency of a totally unbound transitive closure is very poor as compared to the case when one or both arguments are ground. There is a factor of almost 138 between the most and least efficient calling patterns. Also, as we had predicted, there is no visible difference between the efficiencies of calling patterns [g, f] and [g, g]. The fact that the efficiency of calling pattern [f, g] is approximately 1.5 times the efficiency of the other two calling patterns that involve a ground term suggests that this particular transitive closure algorithm is not symmetric.

6.3 The Packages Example

Our final case study will be based on the "Packages Example" described in Appendix 4. Our database profile is shown in Table 6.11.

Predicate name	number of tuples	distinct values in argument 1	distinct values in argument 2	distinct values in argument 3
part/3	1640	136	16	1640
uses/2	4075	1203	1288	—

Table 6.11 The extensional database predicates

We introduce a predicate that computes packages that are in a cycle:

```
cycle(X) :- pkg_uses(X,X)+.
```

It defines a unary predicate cycle(X) to be true when there is a path of one or more arcs labeled pkg_uses from X to itself, that is, when X is related to itself by the transitive closure of pkg_uses [Consens92].

Query to be analyzed

Suppose that we wish to determine the best ordering of the following arbitrary query†:

```
arbitrary_part(Y) :- part_of(Y,X), part_of(Y,U), cycle(X), cycle(U).
```

and consider the case when the Y argument is ground prior to the call. We will proceed to estimate the cost of all different orderings. They are shown, with their respective calling patterns, in Table 6.12. Note that only 10 of the orderings are unique.

It becomes clear that we must determine the abstract properties of predicates part_of and cycle. In Appendix 4, we have already obtained the information related to two intensional database predicates (namely part_of and pkg_uses). The properties of predicate cycle are related to those of predicate pkg_uses, so that we have to deduce the properties of this intensional predicate. We know the specific ordering that is used to implement the pkg_uses predicate.‡ Once we select this ordering we can draw the "black boxes" for the relevant calling patterns for the subgoals and then derive the "black box" for the pkg_uses predicate. As before, we interconnect the "black boxes" that correspond to the selected ordering and then obtain the expected values of our cost contributors once the expected values for the number of solutions are propagated. This is depicted in Figure 6.16 for one of the many possible cost contributors: the number of visited tuples.

Regarding the number of solutions of predicate pkg_uses, we must recall that a system predicate was ignored in the analysis of Appendix 4. The actual number of solutions once the system predicate is considered is approximately ten times smaller.

Now we are able to come up with a "black box" for the cyclic predicate, i.e., the transitive closure of the pkg_uses predicate. Disregarding the additional call and head uni-

†Although this query has no special intent, it is still an interesting example, given the fact that it contains two transitive closures.

‡Although we already know what the best ordering for this predicate would be, sometimes we are not able to modify (and recompile) the already existing code.

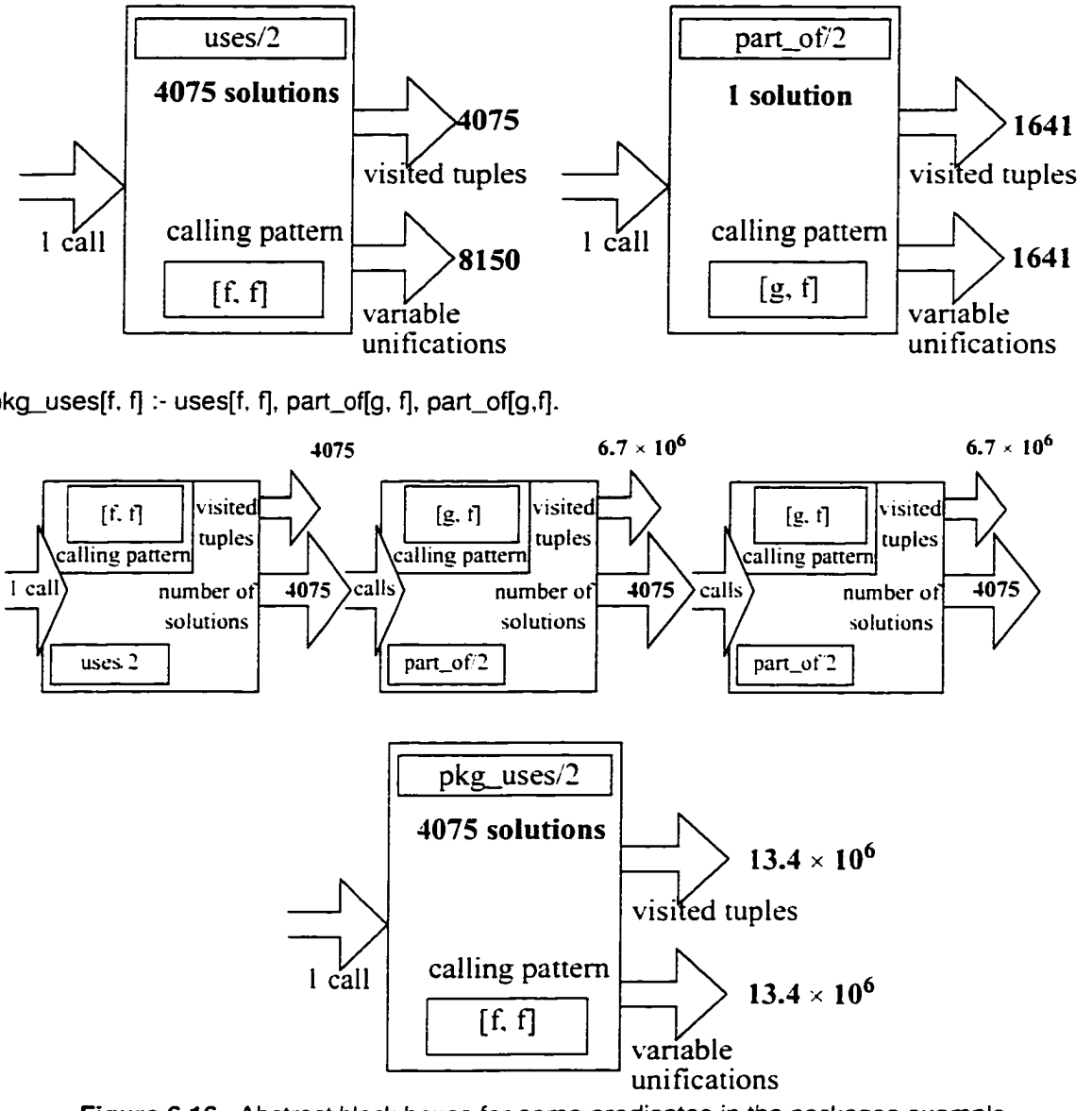


Figure 6.16 Abstract black boxes for some predicates in the packages example

fication due to the indirect call to the transitive closure via the cycle predicate, we apply our formulae for transitive closure to the pkg_uses predicate. In this example, the number of distinct attribute values for the relation is $n_{at} = 1640 \times 1640 = 2689600$ (since 1640 is the number of parts) and the number of unique tuples for the base predicate is estimated as $n_b = 4075$. With these values, we proceed to determine the region that corresponds to their ratio $A = n_{at}/n_b \approx 660.0$. This value lies in the "region of small values for the number of tuples in the base predicate" as explained in Section 5.4.1. Applying the formula derived in that section, we estimate the average number of solutions that results af-

ordering	calling patterns	ordering #
part_of(Y,X), part_of(Y,U), cycle(X), cycle(U).	part_of[f,f], part_of[g,f], cycle[g], cycle[g].	1
part_of(Y,X), part_of(Y,U), cycle(U), cycle(X).	part_of[f,f], part_of[g,f], cycle[g], cycle[g].	1
part_of(Y,X), cycle(X), part_of(Y,U), cycle(U).	part_of[f,f], cycle[g], part_of[g,f], cycle[g].	2
part_of(Y,X), cycle(X), cycle(U), part_of(Y,U).	part_of[f,f], cycle[g], cycle[f], part_of[g,g].	3
part_of(Y,X), cycle(U), part_of(Y,U), cycle(X).	part_of[f,f], cycle[f], part_of[g,g], cycle[g].	4
part_of(Y,X), cycle(U), cycle(X), part_of(Y,U).	part_of[f,f], cycle[f], cycle[g], part_of[g,g].	5
part_of(Y,U), part_of(Y,X), cycle(X), cycle(U).	part_of[f,f], part_of[g,f], cycle[g], cycle[g].	1
part_of(Y,U), part_of(Y,X), cycle(U), cycle(X).	part_of[f,f], part_of[g,f], cycle[g], cycle[g].	1
part_of(Y,U), cycle(X), part_of(Y,X), cycle(U).	part_of[f,f], cycle[f], part_of[g,g], cycle[g].	4
part_of(Y,U), cycle(X), cycle(U), part_of(Y,X).	part_of[f,f], cycle[f], cycle[g], part_of[g,g].	5
part_of(Y,U), cycle(U), part_of(Y,X), cycle(X).	part_of[f,f], cycle[g], part_of[g,f], cycle[g].	2
part_of(Y,U), cycle(U), cycle(X), part_of(Y,X).	part_of[f,f], cycle[g], cycle[f], part_of[g,g].	3
cycle(X), part_of(Y,X), part_of(Y,U), cycle(U).	cycle[f], part_of[f,g], part_of[g,f], cycle[g].	6
cycle(X), part_of(Y,X), cycle(U), part_of(Y,U).	cycle[f], part_of[f,g], cycle[f], part_of[g,g].	7
cycle(X), part_of(Y,U), part_of(Y,X), cycle(U).	cycle[f], part_of[f,f], part_of[g,g], cycle[g].	8
cycle(X), part_of(Y,U), cycle(U), part_of(Y,X).	cycle[f], part_of[f,f], cycle[g], part_of[g,g].	9
cycle(X), cycle(U), part_of(Y,X), part_of(Y,U).	cycle[f], cycle[f], part_of[f,g], part_of[g,g].	10
cycle(X), cycle(U), part_of(Y,U), part_of(Y,X).	cycle[f], cycle[f], part_of[f,g], part_of[g,g].	10
cycle(U), part_of(Y,X), part_of(Y,U), cycle(X).	cycle[f], part_of[f,f], part_of[g,g], cycle[g].	8
cycle(U), part_of(Y,X), cycle(X), part_of(Y,U).	cycle[f], part_of[f,f], cycle[g], part_of[g,g].	9
cycle(U), part_of(Y,U), part_of(Y,X), cycle(X).	cycle[f], part_of[f,g], part_of[g,f], cycle[g].	6
cycle(U), part_of(Y,U), cycle(X), part_of(Y,X).	cycle[f], part_of[f,g], cycle[f], part_of[g,g].	7
cycle(U), cycle(X), part_of(Y,X), part_of(Y,U).	cycle[f], cycle[f], part_of[f,g], part_of[g,g].	10
cycle(U), cycle(X), part_of(Y,U), part_of(Y,X).	cycle[f], cycle[f], part_of[f,g], part_of[g,g].	10

Table 6.12 Different orderings for the query under consideration

ter the computation of the closure as $\overline{n_{tc}} \approx 2689600 / (660.0 - 1) \approx 4081.2$ (expected number of tuples for the entire closure). If one argument is ground, only a fraction of the tuples will be in the solution. We already know that the transitive closure does not follow a uniform distribution (See Section 5.5), but we may produce a rough estimate by making this assumption, thus dividing the total number of tuples by the number of distinct values for that particular ground argument (i.e., the number of parts, in our example), or

$\overline{n_{tc}}[f, g] = \overline{n_{tc}}[g, f] \approx 4081.2 / 1640 \approx 2.49$. Similarly, if both arguments are ground, our estimate would be: $\overline{n_{tc}}[g, g] \approx 2.49 / 1640 \approx 0.0015$. With these values, we are able to propose the "black boxes" for predicate cycle as shown in Figure 6.17.† We also need those "black boxes" that correspond to predicate part_of (Figure 6.18).

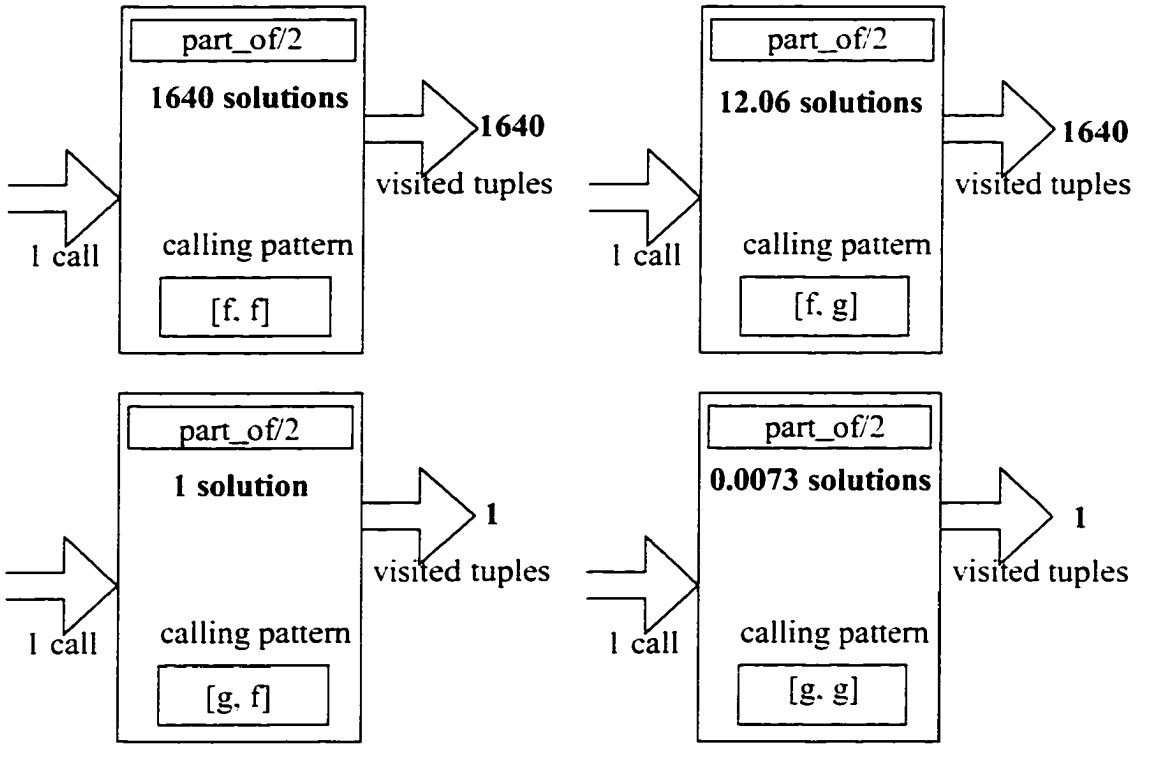


Figure 6.18 Abstract black boxes for predicate part_of

As in the previous case study, we propose that the number of visited tuples in the computation of the closure is in the order of $N \approx (n_b)^3 = 4075^3 \approx 67.7 \times 10^9$ when the first argument is ground. In the previous case study we have already seen that the value of N_{ff} is expected to be some 138 times the values of N_{gg} or N_{gf}, whereas the values of N_{fg} is just about 1.5 times that of N_{gf}. If we use the values $N_{gg} \approx (n_b)^3 = 4075^3 \approx 67.7 \times 10^9$ and $N_{ff} = 138 \times 4075^3 \approx 9.34 \times 10^{12}$ as approximations for the number of visited tuples in the closures, we are finally able to estimate the cost of the ten different orderings. Initially, we will study the case where all arguments are initially uninstantiated, i.e., the case when

†Strictly speaking, we should only have to derive the "black boxes" that correspond to calling patterns [f, f] and [g, g]

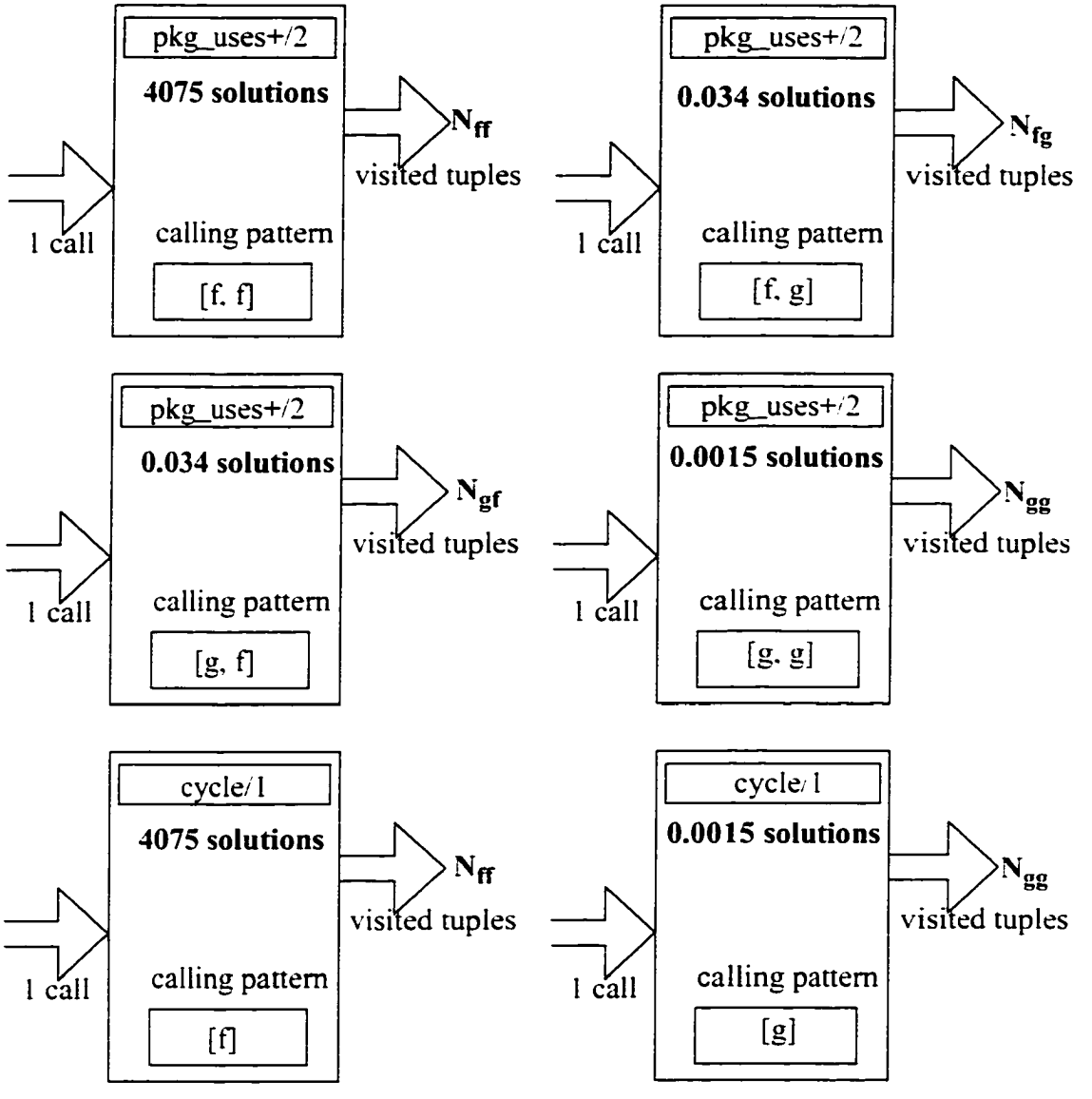


Figure 6.17 Abstract black boxes for predicate cycle

we wish to retrieve all solutions to the general query. As before, our cost contributor will be the expected number of visited tuples. Then, we will consider the case when the user specifies a ground term, i.e., we wish to retrieve only a fraction of the tuples in the general solution.

Example 1

For the case when the query has the form

```
:- arbitrary_part(Y)
```

our estimates are as follows (n_{vt} stands for "number of visited tuples" and n_{sol} implies "number of solutions"):

$$n_{vt}(\text{part\_of}|_{[f,f]}) = n_{vt}[\,\text{p}|_{f,f}] = 1640$$
$$n_{vt}(\text{part\_of}|_{[g,f]}) = n_{vt}[\,\text{p}|_{g,f}] = 1$$
$$n_{vt}(\text{part\_of}|_{[g,g]}) = n_{vt}[\,\text{p}|_{g,g}] = 1$$
$$n_{sol}(\text{part\_of}|_{[f,f]}) = n_{sol}[\,\text{p}|_{f,f}] = 1640$$
$$n_{sol}(\text{part\_of}|_{[f,g]}) = n_{sol}[\,\text{p}|_{f,g}] = 12.06$$
$$n_{sol}(\text{part\_of}|_{[g,f]}) = n_{sol}[\,\text{p}|_{g,f}] = 1$$
$$n_{sol}(\text{part\_of}|_{[g,g]}) = n_{sol}[\,\text{p}|_{g,g}] = 0.0073$$
$$n_{vt}(\text{part\_of}|_{[f,g]}) = n_{vt}[\,\text{p}|_{f,g}] = 1640$$

$$n_{vt}(\text{cycle}|_{[f]}) = n_{vt}[\,\text{c}|_{f}] = 9.34 \times 10^{12}$$
$$n_{vt}(\text{cycle}|_{[g]}) = n_{vt}[\,\text{c}|_{g}] = 67.7 \times 10^{9}$$
$$n_{sol}(\text{cycle}|_{[f]}) = n_{sol}[\,\text{c}|_{f}] = 4075$$
$$n_{sol}(\text{cycle}|_{[g]}) = n_{sol}[\,\text{c}|_{g}] = 0.0015$$

We estimate the number of visited tuples for the different orderings as follows:

$$n_{vt}\#1 = n_{vt}[\,p|_{f,f}] + n_{sol}[\,p|_{f,f}] \times (n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#2 = n_{vt}[\,p|_{f,f}] + n_{sol}[\,p|_{f,f}] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times (n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#3 = n_{vt}[\,p|_{f,f}] + n_{sol}[\,p|_{f,f}] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times n_{vt}[\,p|_{g,g}]))$$

$$n_{vt}\#4 = n_{vt}[\,p|_{f,f}] + n_{sol}[\,p|_{f,f}] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{g,g}] + n_{sol}[\,p|_{g,g}] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#5 = n_{vt}[\,p|_{f,f}] + n_{sol}[\,p|_{f,f}] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times n_{vt}[\,p|_{g,g}]))$$

$$n_{vt}\#6 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{f,g}] + n_{sol}[\,p|_{f,g}] \times (n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#7 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{f,g}] + n_{sol}[\,p|_{f,g}] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times n_{vt}[\,p|_{g,g}]))$$

$$n_{vt}\#8 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{f,f}] + n_{sol}[\,p|_{f,f}] \times (n_{vt}[\,p|_{g,g}] + n_{sol}[\,p|_{g,g}] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#9 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{f,f}] + n_{sol}[\,p|_{f,f}] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times n_{vt}[\,p|_{g,g}]))$$

$$n_{vt}\#10 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{f,g}] + n_{sol}[\,p|_{f,g}] \times n_{vt}[\,p|_{g,g}]))$$

or, after value substitution:

$$n_{vt}\#1 = 1640 + 1640 \times (1 + 1 \times (67.7 \times 10^9 + 0.0015 \times 67.7 \times 10^9)) \approx 111 \times 10^{12}$$

$$n_{vt}\#2 = 1640 + 1640 \times (67.7 \times 10^9 + 0.0015 \times (1 + 1 \times 67.7 \times 10^9)) \approx 111 \times 10^{12}$$

$$n_{vt}\#3 = 1640 + 1640 \times \left(67.7 \times 10^9 + 0.0015 \times \left(9.34 \times 10^{12} + 4075 \times 1\right)\right) \approx 134 \times 10^{12}$$

$$n_{vt}\#4 = 1640 + 1640 \times \left(9.34 \times 10^{12} + 4075 \times (1 + 0.0073 \times 67.7 \times 10^9)\right) \approx 19 \times 10^{15}$$

$$n_{vt}\#5 = 1640 + 1640 \times \left(9.34 \times 10^{12} + 4075 \times (67.7 \times 10^9 + 0.0015 \times 1)\right) \approx 467 \times 10^{15}$$

$$n_{vt}\#6 = 9.34 \times 10^{12} + 4075 \times (1640 + 12.06 \times (1 + 1 \times 67.7 \times 10^9)) \approx 3 \times 10^{15}$$

$$n_{vt}\#7 = 9.34 \times 10^{12} + 4075 \times \left(1640 + 12.06 \times \left(9.34 \times 10^{12} + 4075 \times 1\right)\right) \approx 459 \times 10^{15}$$

$$n_{vt}\#8 = 9.34 \times 10^{12} + 4075 \times (1640 + 1640 \times (1 + 0.0073 \times 67.7 \times 10^9)) \approx 3 \times 10^{15}$$

$$n_{vt}\#9 = 9.34 \times 10^{12} + 4075 \times (1640 + 1640 \times (67.7 \times 10^9 + 0.0015 \times 1)) \approx 452 \times 10^{15}$$

$$n_{vt}\#10 = 9.34 \times 10^{12} + 4075 \times \left(9.34 \times 10^{12} + 4075 \times (1640 + 12.06 \times 1)\right) \approx 38 \times 10^{15}$$

and we conclude that it is likely that the first three orderings are the most efficient ones from the viewpoint of the number of visited tuples.

Example 2

For the case when the query has the form

```
:- arbitrary_part(a_constant_part)
```

our estimates are modified as follows (again, n_{vt} stands for "number of visited tuples" and n_{sol} implies "number of solutions"):

$$n_{vt}\#1 = n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times (n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,g}] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#2 = n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times (n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#3 = n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times n_{vt}[\,p|_{g,g}]))$$

$$n_{vt}\#4 = n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{g,g}] + n_{sol}[\,p|_{g,g}] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#5 = n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times n_{vt}[\,p|_{g,g}]))$$

$$n_{vt}\#6 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{g,g}] + n_{sol}[\,p|_{g,g}] \times (n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#7 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{g,g}] + n_{sol}[\,p|_{g,g}] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times n_{vt}[\,p|_{g,g}]))$$

$$n_{vt}\#8 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times (n_{vt}[\,p|_{g,g}] + n_{sol}[\,p|_{g,g}] \times n_{vt}[\,c|_g]))$$

$$n_{vt}\#9 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{g,f}] + n_{sol}[\,p|_{g,f}] \times (n_{vt}[\,c|_g] + n_{sol}[\,c|_g] \times n_{vt}[\,p|_{g,g}]))$$

$$n_{vt}\#10 = n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,c|_f] + n_{sol}[\,c|_f] \times (n_{vt}[\,p|_{g,g}] + n_{sol}[\,p|_{g,g}] \times n_{vt}[\,p|_{g,g}]))$$

or, after value substitution:

$$n_{vt}\#1 = 1 + 1 \times (1 + 1 \times (67.7 \times 10^9 + 0.0015 \times 67.7 \times 10^9)) \approx 67.8 \times 10^9$$

$$n_{vt}\#2 = 1 + 1 \times (67.7 \times 10^9 + 0.0015 \times (1 + 1 \times 67.7 \times 10^9)) \approx 67.8 \times 10^9$$

$$n_{vt}\#3 = 1 + 1 \times \{ 67.7 \times 10^9 + 0.0015 \times \{ 9.34 \times 10^{12} + 4075 \times 1 \} \} \approx 81.7 \times 10^9$$

$$n_{vt}\#4 = 1 + 1 \times \{ 9.34 \times 10^{12} + 4075 \times (1 + 0.0073 \times 67.7 \times 10^9) \} \approx 11 \times 10^{12}$$

$$n_{vt}\#5 = 1 + 1 \times \{ 9.34 \times 10^{12} + 4075 \times (67.7 \times 10^9 + 0.0015 \times 1) \} \approx 285 \times 10^{12}$$

$$n_{vt}\#6 = 9.34 \times 10^{12} + 4075 \times (1 + 0.0073 \times (1 + 1 \times 67.7 \times 10^9)) \approx 11 \times 10^{12}$$

$$n_{vt}\#7 = 9.34 \times 10^{12} + 4075 \times \{ 1 + 0.0073 \times \{ 9.34 \times 10^{12} + 4075 \times 1 \} \} \approx 287 \times 10^{12}$$

$$n_{vt}\#8 = 9.34 \times 10^{12} + 4075 \times (1 + 1 \times (1 + 0.0073 \times 67.7 \times 10^9)) \approx 11 \times 10^{12}$$

$$n_{vt}\#9 = 9.34 \times 10^{12} + 4075 \times (1 + 1 \times (67.7 \times 10^9 + 0.0015 \times 1)) \approx 285 \times 10^{12}$$

$$n_{vt}\#10 = 9.34 \times 10^{12} + 4075 \times \{ 9.34 \times 10^{12} + 4075 \times (1 + 0.0073 \times 1) \} \approx 38 \times 10^{15}$$

and, again, we conclude that the first three orderings are the most likely to be more efficient from the perspective of the number of visited tuples. There is a factor of almost 162 between the three least expensive orderings and the next most efficient ordering.

We obtained some experimental values (only for the most efficient orderings) which are summarized in Table 6.13.†

calling patterns	ordering #	theoretical ranking	experimental result	experimental ranking
part_of[f,f], part_of[g,f], cycle[g], cycle[g].	1	1=	1340538	1=
part_of[f,f], cycle[g], part_of[g,f], cycle[g].	2	1=	1343804	1=
part_of[f,f], cycle[g], cycle[f], part_of[g,g].	3	3	1351705	1=

Table 6.13 Experimental results for the three most efficient orderings

6.4 Comparison to Sheridan's algorithm

In this section we will compare our results to those obtained by the application of Sheridan's algorithm [See Section 2.2.4], one of the most successful reordering algorithms in the literature.

6.4.1 Why is Sheridan's algorithm so successful?

The main idea exploited by Sheridan's algorithm is that the more instantiated the arguments in a predicate call are, the less expensive that call will be. For instance, in the results shown in Table 6.14 (borrowed from Appendix 4), we notice that having an additional ground argument always guarantees a lower cost of execution.

<clause, cpat>	cost
<part/3, [f,f,f]>	20.99
<part/3, [f,f,g]>	13.12
<uses/2, [f,f]>	44.01
<uses/2, [f,g]>	24.46
<uses/2, [g,f]>	0.22
<uses/2, [g,g]>	0.01
<part_of/2, [g,f]>	1
<part_of/2, [f,f]>	1640

Table 6.14 Cost metrics for all predicates

†A single experiment for the most efficient ordering (orderings # 1, 2, 3) required several hours of computation. The computation of a single experiment for the next group of orderings (orderings # 4, 6, 8) would require close to one month of uninterrupted execution!

6.4.2 Our framework versus Sheridan's

When comparing our results with those obtained by using Sheridan's algorithm, our framework gives consistently better results than Sheridan's for at least two groups of situations: (a) when the *position* of the ground arguments within a predicate call is crucial, and (b) when the performance of two syntactically similar predicate calls varies considerably because of noticeably different sizes of their underlying database definitions.

A simple glance at Table 6.14 will reveal that, for some predicates, the exact locations of the ground arguments will have an impact on the performance of the predicate call. For example, predicate call *uses* with a first argument constant and a second argument variable (i.e., calling pattern [g, f]) will perform far better than the call with a second argument constant and a first argument variable (i.e., calling pattern [f, g]). Sheridan's algorithm has no way to determine such a difference, and which of the two will be given preference is a matter of chance.

By a similar token, due to the fact that Sheridan's algorithm does not take into account any information regarding the underlying database, there is no obvious way to distinguish between a potentially very expensive predicate and a less expensive one based only on the nature (i.e., the *mode*) of the arguments. Consider a very simple case, that is shown in Figure 6.19.

Here we have two database predicates with a remarkably different number of facts. Sheridan's algorithm would consider both orderings shown in the figure as equally expensive: the first predicate is executed with both arguments variable; the second predicate is executed with one ground argument and one variable argument. However, since it is dramatically more expensive to retrieve all several thousand tuples of predicate p as opposed to only one tuple for predicate q, it is better to place the call to predicate q before the call to predicate p. Again, since our framework uses a profile of the database in use, we are able to make this kind of prediction, whereas Sheridan's algorithm is not.

A third situation in which our method has a potential advantage over Sheridan's algorithm can be observed when the query contains recursive predicates or path regular expressions. Sheridan's algorithm does not treat these predicates as special cases, and then,

predicate	number of tuples (experiment #1)	number of tuples (experiment # 2)
p	10.000	90.000
q	1	1

```
order1(A,B,C):- p(A,B),q(B,C).
order2(A,B,C):- q(B,C),p(A,B).
```

ordering	average cost (experiment #1)	average cost (experiment # 2)
order1	1020	8970
order2	330	2850

Figure 6.19 Impact of the underlying database on the performance of the call

a potentially expensive recursive call may be chosen as one of the subgoals to be executed first. Our methodology permits us to estimate the sizes of those special predicates and their repercussions on succeeding predicates.

Chapter 7. Conclusions and Future Work

In this chapter, we summarize our results and address the limitations of our framework. We also propose some additional work that is required.

7.1 Contributions of this Dissertation

We have proposed a new methodology to estimate the cost of a general GraphLog query. It is based on the assumption that a profile of the underlying database is known. We have been able to predict with good accuracy the costs of conjunctions of queries when the program is translated into a WAM-based version of Prolog. This is done by obtaining empirical values for the diverse primitive operations in which the query is decomposed, in combination with predictions regarding the number of times these primitive operations will be invoked. We have also explained how to predict the costs when different abstract machines are used by defining relevant cost contributors whose values are propagated in conjunction with the expected number of solutions for each subgoal in the query. Our predictions are normally able to detect the most efficient reorderings as well as the most expensive ones. The accuracy of the results is noticeably influenced by the nature of the underlying database, and our predictions are best for databases whose distribution resembles a uniform distribution of attribute values.

Our predictions are usually superior to those obtained when using Sheridan's algorithm, because we make use of more information and are also able to consider some special forms of subgoals.

A major contribution of our work is the ability to handle recursive queries and closures. We have shown that the key factor is the estimation of the output density after applying the transitive closure. We have provided some guidelines and formulae to estimate such output density which have not appeared elsewhere in the literature. Our results may be of special interest given that SQL, the *de facto* standard for relational databases,

has finally included recursive constructs [Dar93], and recursive query languages will soon become the norm [Ahad93].

Our results should be directly applicable to pure Datalog and any other function-free database language.

7.2 Limitations of Our Framework

The main deficiency of our methodology is that we have assumed "ideal" databases. We have not addressed some important issues such as duplication of tuples after a projection or correlation amongst attributes. Many of our claims may be applicable to uniform distributions of attribute values only.

Another issue that has not been addressed by our framework is the impact that ground arguments have on the cost of the unification algorithm. It stands to reason that the unification of long strings of characters consumes more time and resources than unification of an integer or an atom with a short name. In fact, some systems transform the real attributes to shorter and easier to handle equivalent codes [Graefe91], as is the case in the performers database in Chapter 6.

7.3 Future work

Our current framework does not consider the special case of aliasing of variables (especially when the same variable is used several times within the same predicate). A domain analysis of the arguments that are involved may establish some upper bounds for the number of tuples that are retrieved.

We also propose to address the inclusion of correlation factors and the analysis of other path regular expressions besides transitive closure. Our framework would also be more complete if built-in predicates were also to be included in the analysis.

It is likely that additional information regarding the cardinality of the base relations may be used to refine our results. Further study of this idea could be fruitful.

If we wish to use our framework in a practical situation, we require a pre-process that determines a set of suitable query orderings to be analyzed. Some methods have

been proposed to this effect (randomized algorithms [Ioannidis90] or even Sheridan's algorithm may be adapted to that purpose). Naturally, we need to incorporate a phase that determines the calling patterns of the different subgoals, but this has been already solved elsewhere [Debray88].

So far, we have not mentioned what to do with built-in predicates, a common extension to pure GraphLog/Datalog. In fact, we consider that the inclusion of built-in predicates fits quite naturally in our framework. Estimating the number of solutions of a built-in predicate becomes trivial when the domain of the attributes is known in advance, and so do the values of the different cost contributors we might be interested in.

Several extensions have been proposed for GraphLog [Consens89]. These include the definition of aggregates and the option of using functional arguments. *Aggregates* are constructs that are used to summarize data (typical aggregates are the average, maximum, minimum, sum or count of an attribute). We believe that our framework can be easily extended to handle these constructs once we determine what additional operations take place. In general, aggregates have to perform an action over all the tuples that satisfy a condition. Our framework already estimates the cost of retrieval of the tuples, and we only need to add the cost due to the aggregate action (which will usually require to visit each and every tuple in the solution). Handling functional arguments is a more complex matter. We require to derive a richer abstract domain to distinguish partially instantiated arguments, and modify the rules of the corresponding abstract unification.

It is not unusual to use a cache in order to reduce the cost of processing a query by preventing multiple evaluations of the same predicate call [Sellis87]. This is specially useful for inherently expensive queries (such as transitive closures). A practical cost model would have to take into consideration this and other implementation issues.

References

[Ahad93] Ahad, R., and Yao, B. RQL: A Recursive Query Language. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 3, June 1993, pp. 451-461.

[Agrawal90] Agrawal, R., Dar, S., and Jagadish, H.V. Direct transitive closure algorithms: Design and performance evaluation. *ACM Transactions on Database Systems*, Vol. 15, No. 3, September 1990, pp. 427-458.

[Aït91] Aït-Kaci, H. *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, Mass., 1991.

[Baase88] Baase, S. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 2nd ed., 1988.

[Bancilhon86] Bancilhon, F., Ramakrishnan, R. An amateur's introduction to recursive query processing strategies. *Proceedings of the 1986 ACM-SIGMOD Conference*, 1986, pp. 16-52.

[Birkhoff40] Birkhoff, G. *Lattice theory*. American Mathematical Society Colloquium Publications, Vol. 25, New York, 1940.

[Ceri90] Ceri, S., Gottlob, G., and Tanca, L. *Logic Programming and Databases*. Springer-Verlag Berlin, 1990.

[Ceri91] Ceri, S., Gottlob, G., and Tanca, L. Datalog: A Self-Contained Tutorial (Part I). Published in *Programmirovanie*, No. 4, July-August, 1991, pp. 20-38.

[Cheiney94] Cheiney, J.-P., and Huang, Y.-N. Efficient maintenance of explicit transitive closure with set-oriented update propagation and parallel processing. *Data and Knowledge Engineering*, Vol. 13, No. 3, October 1994, pp. 197-226.

[Clocksin81] Clocksin, W.F., and Mellish, C.S. Programming in Prolog. Springer-Verlag, New York, 1981.

[Consens89] Consens, M.P. *Graphlog: "Real Life" Recursive Queries Using Graphs*, MS thesis, Department of Computer Science, University of Toronto, January 1989.

[Consens90] Consens, M., Mendelzon, A. GraphLog: a Visual Formalism for Real Life Recursion. *Proceedings of the 9th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1990, pp. 404-416.

[Consens92] Consens, M., Mendelzon, A., and Ryman, A. Visualizing and Querying Software Structures. *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[Cousot77] Cousot, P., and Cousot, R. Abstract Interpretation: a Unified Framework for Static Analysis of Programs by Construction of Approximation of Fixpoints. *Proceedings of the 4th ACM Conference on Principles of Programming Languages*, The Association for Computing Machinery, New York, N.Y., 1977.pp. 238-252.

[Cousot92] Cousot, P., and Cousot, R. *Abstract Interpretation and Application to Logic Programs*. Laboratoire d'Informatique de l'École Normale Supérieure, Research Report LIENS-92-12, June 1992.

[Dar93] Dar, S., and Agrawal, R. Extending SQL with Generalized Transitive Closure Functionality. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 5, October 93, pp. 799-812.

[Debray89] Debray, S.K. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 11 No. 3, July 1989, pp. 418-450.

[Debray93] Debray, S.K., and Lin, N. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 15 No. 5, Nov 1993, pp. 826-875.

[Debray88] Debray, S.K. and Warren, D.S. Automatic Mode Inference for Logic Programs. *The Journal of Logic Programming*, Vol. 5 no. 3, Sept. 1988, pp. 207-229.

[Fröberg85] Fröberg, C.-E. *Numerical Mathematics* - Theory and Computer Applications. The Benjamin/Cummings Publishing Co., 1985.

[Fukar91] Fukar, M. *Translating Graphlog into Prolog*. Technical Report TR 74.080. Centre for Advanced Studies, IBM Canada Laboratory, 1991.

[Gardarin89] Gardarin, G. and Valduriez, P. *Relational Databases and Knowledge Bases*. Addison-Wesley, 1989.

[Gooley89] Gooley, M.M., and Wah, B.W. Efficient Reordering of PROLOG Programs. *IEEE Transactions on Knowledge and Data Engineering*, Volume 1, Number 4, 1989, pp. 470-482.

[Gorlick87] Gorlick, M.M., and Kesselman, C.F. Timing Prolog Programs without Clocks. *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, CA, pp. 426-432.

[Graefe91] Graefe, G., and Shapiro, L.D. Data compression and database performance. *1991 Proceedings of the ACM/IEEE-Computer Science Symposium on Applied Computing*.

[Horn51] Horn, A. On Sentences which are True of Direct Unions of Algebras. *Journal of Symbolic Logic*, Vol. 16, pp. 14-21.

[Ioannidis90] Ioannidis, Y.E., and Kang, Y.C. Ramdomized Algorithms for Optimizing Large Join Queries. *Proceedings of the 1990 ACM-SIGMOD Conference on the Management of Data*, Atlantic City, NJ, USA, May 1990, pp. 312-321.

[Ioannidis95] Ioannidis, Y.E., and Poosala, V. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, pp. 233-244.

[Jagadish86] Jagadish, H.V., and Agrawal, R. A Study of Transitive Closure as a Recursion Mechanism. *Proceedings of the ACM Special Interest Group on Management of Data 1987 Annual Conference* (SIGMOD Record Vol. 16 No. 3, December 1987), San Francisco, CA, USA, May 27-29 1987, pp. 331-344.

[Jarke84] Jarke, M., and Koch, J. Query Optimization in Database Systems. *ACM Computing Surveys*, Vol. 16, No. 2, June 1984, pp. 111-152.

[Kruse87] Kruse,R.L. *Data Structures & Program Design*. Prentice-Hall Software Series, New Jersey, USA, 1987.

[Kwast94] Kwast, K.L., and van Denneheuvel, S.J. Duplicates in SQL. *Data and Knowledge Engineering*, Vol. 13, No. 1, August 1994, pp. 31-66.

[Lu93] Lu, W., and Lee, D.L. Characterization and Processing of Simple Prefixed-Chain Recursion. *Information sciences*, Vol. 68, No. 3, March 1993.

[Mannino88] Mannino, M.V. et al. Statistical Profile Estimation in Database Systems. *ACM Computing Surveys*, Vol. 20, No. 3, September 1988, pp. 191-221.

[McCarthy82] McCarthy, J. *Coloring Maps and the Kowalski Doctrine*. Report STAN-CS-82-903, Department of Computer Science, Stanford University, 1982.

[McEnery90] McEnery, A., and Nikolopoulos, C. A Meta-Interpreter for Prolog Query Optimization. *Proceedings of the IASTED International Symposium on Expert Systems Theory and Applications*, 1990. pp. 75-77.

[Mellish85] Mellish, C.S. Some Global Optimizations for a Prolog Compiler. *The Journal of Logic Programming*, vol. 2 no. 1, April 1985, pp. 43-66.

[Mishra92] Mishra, P., and Eich, M.H. Join Processing in Relational Databases. *ACM Computing Surveys*, Vol. 25, No. 2, June 1993, pp. 63-113.

[Ryman92] Ryman, A. Foundations of 4Thought. *Proceedings of CASCON '92*, 1992, pp. 133-155.

[Ryman93] Ryman, A. Illuminating Software Specifications. *Proceedings of CASCON '93*, Vol. I, 1993, pp. 412-428.

[Ryman93a] Ryman, A. Constructing Software Design Theories and Models, in *Studies in Software Design*, LNCS 1078, Springer, 1993, pp. 103-114.

[Sellis87] Sellis, T.K. Efficiently supporting procedures in relational database systems. *Proceedings of the 1987 ACM SIGMOD Conference*, San Francisco, CA, pp. 278-291.

[Sheridan91] Sheridan, P. B. On Reordering Conjunctions of Literals; a Simple, Fast Algorithm. *Proceedings of the 1991 Symposium on Applied Computing*, Kansas City, Mo, USA, April 1991. IEEE Computer Society, pp. 73-79.

[Ullman85] Ullman, J.D. Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems*, Vol. 10, No. 3, 1985, pp. 289-321.

[Ullman88] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, Vol. I and II, Computer Science Press, 1988-89.

[Wang93] Wang, J., Yoo, J., and Cheatham, T. Efficient Reordering of C-PROLOG. *Proceedings of the 21st ACM Computer Science Conference*, NY, USA, 1993, pp. 151-155.

[Warren75] Warren, E.S. A Modification of Warshall's algorithm for the transitive closure of binary relations. *Communications of the ACM*, Vol. 18, No. 4, April 1975, pp. 218-220.

[Warren81] Warren, D.H.D. Efficient Processing of Interactive Relational Database Queries Expressed in Logic. *Proceedings of the 7th International Conference on Very Large Databases*, pp. 272-281, IEEE, 1981.

[Woo85] Woo, N.S. A Hardware Unification Unit: Design and Analysis. *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, MA, 1985, pp. 198-205.

Appendix 1 A Detailed View of Other Approaches to Query Reordering

A1.1 Efficient Reordering of Prolog Programs by Using Markov Chains

Gooley and Wah's work [Gooley89] has proposed a model that approximates the evaluation strategy of Prolog programs by means of a Markov process. The cost is measured as the number of predicate calls or unifications that take place. The method needs to know in advance the probability of success and the cost of execution of each predicate. With this initial information, the cost of a particular ordering for the subgoals within a single clause is calculated as follows:

Consider a predicate clause p with subgoals $s_1, \ldots, s_n$.

$$p :- s_1, \ldots, s_n.$$

If q_i is the probability that subgoal s_i fails, and if c_i is the cost associated with executing subgoal s_i, the cost of a failure is given by the formula:

$$c_f = \{q_1 \times c_1\} + \{(1-q_1) \times q_2 \times (c_1 + c_2)\} + \ldots +$$
$$\{(1-q_1) \times (1-q_2) \times \ldots \times (1-q_{n-1}) \times q_n \times (c_1 + \ldots + c_n)\}$$

or in closed form:

$$c_f = \sum_{m=1}^{n} \left\{ \left(\prod_{i=1}^{m-1} (1-q_i) \right) \times q_m \times \left(\sum_{r=1}^{m} c_r \right) \right\}$$

The goal of the method is to make failing clauses fail earlier and thus reduce backtracking. In other words, goals that are "more likely to fail" (and inexpensive to evaluate) are placed near the head of the clause. This is usually accomplished by ordering the subgoals in decreasing order of their ratios q_i/c_i.

A very similar approach is used to estimate a suitable ordering for the clauses in a given predicate:

Consider a predicate p defined by clauses k_1, ... k_m,

k_1: p :- s_{11}, ... s_{1n}.
k_2: p :- s_{21}, ... s_{2n}.

.

.

.

k_m: p :- s_{m1}, ... s_{mn}.

If p_i is the probability that clause k_i fails, and if d_i is the cost associated with executing clause k_i, the cost of a single success is given by the formula:

$$d_f = \{p_1 \times d_1\} + \{(1-p_1) \times p_2 \times (d_1 + d_2)\} + \ldots +$$
$$\{(1-p_1) \times (1-p_2) \times \ldots \times (1-p_{n-1}) \times p_n \times (d_1 + \ldots + d_n)\}$$

or in closed form:

$$d_f = \sum_{m=1}^{n} \left\{ \left\{ \prod_{i=1}^{m-1} (1-p_i) \right\} \times p_m \times \left\{ \sum_{r=1}^{m} d_r \right\} \right\}$$

The goal in this case is to get an initial answer as quickly and inexpensively as possible. In other words, goals that are "more likely to succeed" (and inexpensive to evaluate) are placed near the beginning of the predicate. This is intuitively accomplished by ordering the clauses in a decreasing order of their ratios p_i/d_i.

Note that these formulae assume that costs and success/failure probabilities of the subgoals in a clause are independent of each other, which is usually not the case in Prolog or GraphLog. Thus, the model is just a coarse approximation, although the behaviour of the clauses may still be predicted with some accuracy.

The success/failure probability and cost of executing the body of a clause (that does not involves recursion) are both calculated once the subgoals s_1, ... s_n are modelled by either the Markov chain in Figure A1.1 if we are interested in the first solution only, or the Markov chain in Figure A1.2 if we want the cost of finding all solutions.

Note that each subgoal s_i corresponds to a distinct state. From each subgoal state there are two transition arcs which are labelled with the probabilities of success (p_i) and

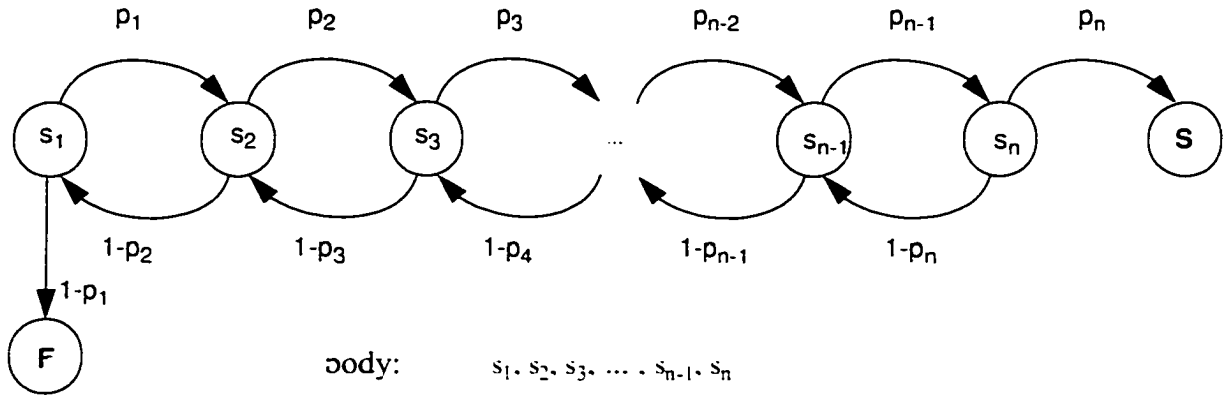


Figure A1.1 Markov chain for the single solution case

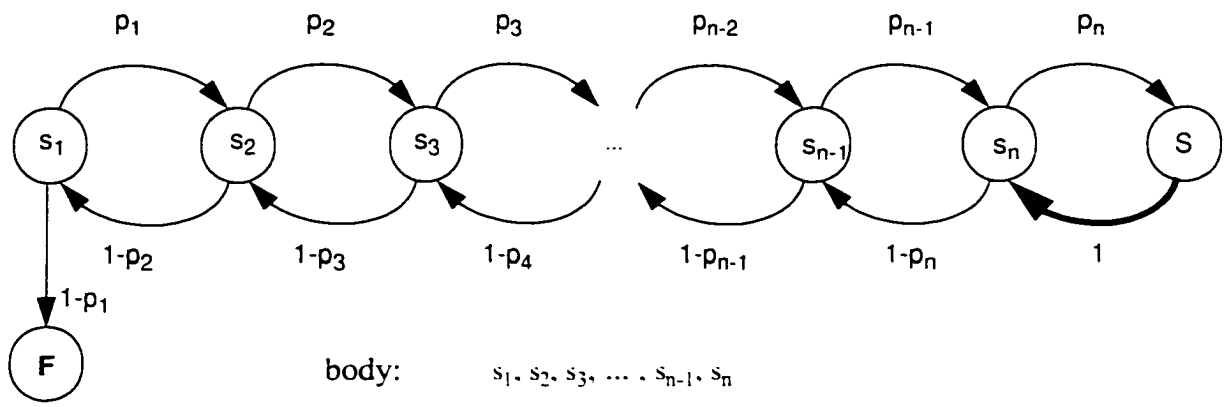


Figure A1.2 Markov chain for the all-solutions case

failure ($1\text{-}p_i$). The success transition arc connects the subgoal state with the next subgoal state, while the failure transition connects it with the previous one. For the special case of the first subgoal, the failure transition arc goes to an absorbing state labelled *F* (clause failure). Similarly, the success transition arc for the last subgoal reaches another absorbing state labelled S (clause success).

For GraphLog queries, we are often interested in deriving all solutions rather than just the first one. Notice that when the all-solutions case is under consideration, the *S* state is no longer an absorbing state and it has a failure transition arc of probability one assigned to it (which mimics backtracking).

Gooley and Wah [Gooley89] explain that these Markov chains can be represented mathematically by means of $r \times r$ matrices, where r is the number of states in the chain. The transition matrix for the single-solution case has the following structure [Revuz84]:

$$\begin{bmatrix} 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1-p_1 & 0 & p_1 & \dots & 0 & 0 \\ 0 & 0 & 1-p_2 & 0 & p_2 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & 0 & 0 & 0 & 1-p_n & 0 & p_n \\ p_n & \dots & 0 & 0 & 0 & 1-p_n & 0 \end{bmatrix}$$

A similar matrix is obtained for the all-solutions case:

$$\begin{bmatrix} 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 1-p_1 & 0 & p_1 & \dots & 0 & 0 & 0 \\ 0 & 1-p_2 & 0 & p_2 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & 0 & 0 & 1-p_n & 0 & p_n & \dots \\ \dots & 0 & 0 & 0 & 1-p_n & 0 & p_n \\ 0 & \dots & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Several cost metrics (such as the number of visits to the success state S and the probabilities and costs for the clause body) can be obtained after some mathematical manipulation of these matrices. For instance, the expected cost of a solution in the all-solutions case is given by:

$$c_{multiple} = \sum_{i=1}^{n} c_i \left(\prod_{i=1}^{n} \frac{p_{i-1}}{1-p_i} \right)$$

A1.2 A Meta-Interpreter for Prolog Query Optimization

McEnery and Nikolopoulos [McEnery90] describe a meta-interpreter for Prolog which reorders clauses and predicates. It has two components: (a) a static component in charge of rearranging the clauses "a priori", and (b) a dynamic component that reorders the clauses according to probabilistic profiles built from previously answered queries.

This method's static reordering phase consists of rearranging the clauses that define a predicate in such a way that the most successful clauses are tried first, and the subgoals within a clause are reordered in descending order of success likelihood.

Subgoal reordering is performed by using a generalization of a heuristic due to D.H.D. Warren [Warren81]. Warren proposed a formula for the cost c of a simple query q as given by $c_q = s/a$, where s is the size in tuples (i.e., the number of solutions) of the subgoal, and a is the product of the sizes of the domains of each instantiated argument. For example, given the following Prolog database:

```
nation(canada).
nation(belgium).
nation(uk).
language(canada, french).
language(canada, english).
language(belgium, dutch).
language(belgium, french).
language(belgium, german).
language(uk, english).
language(quebec, french).
language(texas, english).
```

and the following predicate definition:

```
french_speaking_nation(N) :- nation(N), language(N, french).
```

the cost of the query

```
:- french_speaking_nation(R).
```

would be obtained as follows:

The cost associated with the execution of predicate *nation* with an unbound argument is given by:

$$c_{\text{nation}} = \frac{3}{1} = 3$$

(there are three nations, and thus $s = 3$; there are no instantiated arguments, therefore $a = 1$). Similarly, the cost of subgoal *language* with both arguments bound is estimated as:

$$c_{\text{language}} = \frac{8}{5 \times 4} = 0.4$$

(there are eight tuples for the predicate, and thus $s = 8$; the value of a is derived from the fact that there are five regions and four different languages). Thus, the cost of the whole query would be given by:

$$c_{\text{french speaking nation}} = c_{\text{nation}} + c_{\text{language}} = 3.4$$

if the textual ordering is to be applied.

If the alternative order

```
french_speaking_nation(N) :- language(N, french), nation(N).
```

is to be used instead, our estimates will change accordingly. We calculate the cost of subgoal *language* when the first argument is unbound and the second argument is ground:

$$c'_{\text{language}} = \frac{8}{4} = 2$$

and the cost of subgoal *nation* with a bound argument:

$$c'_{\text{nation}} = \frac{3}{3} = 1$$

and we obtain the cost of the alternative order as:

$$c'_{\text{french speaking nation}} = c'_{\text{language}} + c'_{\text{nation}} = 3$$

In other words, this second ordering is estimated to be more efficient than the original one. ❐

The generalized formula proposed by McEnery and Nikolopoulos is given by:

$$c = \frac{s}{a \times p}$$

where s and a are defined as in Warren's formula, and p is the probability of success of the clause under analysis. The authors propose a dynamic evaluation of the value of p, according to the accumulated history of the predicate. Note that the higher the value of p, the lower its cost. This success rate is physically stored in the database as an ordered tuple. Every time a clause succeeds, its success rate is increased by one. The probabilistic profile is given by the ratio of the success rate and the overall sample space.

A1.3 Cost Analysis of Logic Programs

Debray and Lin [Debray93] have proposed a framework to analyze the cost of logic programs, including programs with simple recursion. The method estimates the number of solutions of a logic program based on the *sizes* of the diverse predicate arguments. Various measures are included under the generic name "size", such as integer-value, list-length, term-depth, or term-size. Thus, some *type* information must be inferred and propagated for each argument via a static program analysis.

The method derives size relationships amongst predicate arguments. This size information is propagated to compute the number of solutions generated by each predicate.

The size properties of predicate arguments are described by means of two functions: (a) *size(arg)*, which provides the actual size of argument *arg*, and (b) *diff(arg1, arg2)*, that calculates the size difference between two arguments, *arg1* and *arg2*. Each of these functions has a different definition depending on the measure under consideration. For example, the definition of *size(arg)* for the particular measure "list-length" is as follows†:

$$\text{size}(t) = \begin{cases} 0 & \text{if } t \text{ is the empty list} \\ 1 + \text{size}(t_1) & \text{if } t \text{ is of the form } [\_|t_1] \text{ for some term } t_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

†We use the standard Prolog notation for lists. [H|T] refers to a list whose initial element is H (the *head* of the list) and the rest of the list is another list T (the *tail* of the list). An underscore "_" represents an anonymous variable, i.e. a variable whose exact binding is irrelevant for our purposes.

Note that when the value of a particular size (or difference between sizes) cannot be determined from the context, the functions return a special value of "undefined".

A distinction between "output" and "input" arguments is also made. The size of an input argument is always calculated from the sizes of previous occurrences of the variables appearing in that argument (the so-called *predecessors* of the input position). Consider the following predicate clause in which the measure under consideration is "list-length":

```
nrev([H|L], R) :- nrev(L, R1), append(R1, [H], R).
```

in which the call has the following input/output argument positions:

```
nrev(<input>, <output>),
```

and, from this calling pattern, we may derive the calling patterns of the subgoals on the right hand side as follows:

```
nrev(<input>, <output>) :- nrev( <input>, <output>), append(<input>, <input>, <output>).
```

For pedagogical reasons, we number the argument positions (and literals) as follows:

```
nrev_a(<input_1>, <output_1>) :- nrev_b( <input_2>, <output_2>), append(<input_3>, <input_4>,
<output_3>).
```

The size of $<input_4>$ (i.e., [H]) is simply calculated as the size of a unitary list as given by function *size(arg)*, i.e., size($<input_4>$) = 1. The size of $<input_2>$ (i.e., L) is obtained by applying function *diff(arg1, arg2)* to $<input_1>$ ([H|L], predecessor of $<input_2>$) and $<input_2>$ itself, which gives size($<input_2>$) = size($<input_1>$) - 1. Finally, the size of $<input_3>$ (i.e., R1) is expressed in terms of that argument predecessor, $<output_2>$ (whose value must be calculated elsewhere) as size($<input_3>$) = size($<output_2>$).

Size relationships for output argument positions are derived as functions expressed in terms of the sizes of the different input arguments, symbolically expressed as Sz(S,Arg,size($input_1$), ..., size($input_n$)), where $input_1$ ... $input_n$ are the input arguments of subgoal *S* and *Arg* is the argument position whose size is being calculated.

In our example, the size of <output$_2$> is expressed in terms of the input argument as size(<output$_2$>) = Sz(nrev$_b$,2,size(<input$_2$>)), and the size of <output$_3$> is derived from the sizes of its two input arguments as size(<output$_3$>) = Sz(append,3,size(<input$_3$>), size(<input$_4$>)). Similarly, the size of <output$_1$> is obtained as a function of <output$_3$>, the argument that originates the output value of the whole clause, and in this case, size(<output$_1$>) = size(<output$_3$>).

The set of size relations that is obtained for a given predicate clause is then expressed in terms of head input arguments only, a process that is called *normalization*. For instance, if we already know that Sz(nrev,2,size(X)) = size(X) for a given input X, then size(<input$_3$>) should be transformed successively into:

size(<input$_3$>) = size(<output$_2$>) = Sz(nrev$_b$,2,size(<input$_2$>)) = size(<input$_2$>) = size(<input$_1$>) - 1

which is expressed in terms of the head input argument exclusively. Once normalization has been performed, a system of difference equations is obtained. To get closed form expressions, these difference equations need to be solved. Unfortunately, solving difference equations automatically is a difficult problem, although automatic solutions for a wide variety of them have been proposed in the literature.

The number of solutions generated by a predicate is estimated from the size relationships by counting the number of possible values that every variable in the clause may be bound to. The method exploits two properties of unification that hold in many logic programming languages. One of these properties is that if a variable appears n times in the subgoals of a clause, the total number of distinct bindings that such a variable may have is at most the minimum value of the set $\{b_1, ..., b_n\}$, the number of possible bindings for the variable at each argument position as they would be computed independently. For instance, consider the predicate clause:

```
m(X,Y) :- n(X), o(Y), p(X,Y), q(Y).
```

in which predicates o, p, and q are predicates that always return a bound term for argument positions 1, 2, and 1, respectively. It follows that variable Y will be bound to only

those ground values that are common to all three predicates o, p, and q. If the number of distinct values at those argument positions are b_o, b_p, and b_q, respectively, the number of distinct bindings for variable Y is at most $min\{b_o, b_p, b_q\}$.

Another useful property of unification is that for subgoals that contain more than one variable, an upper bound of the number of bindings for such a subgoal is given by the product of the number of bindings of each one of its variables. For example, given a subgoal

s(Y,X,Y)

if b_Y is the number of bindings that variable Y can take and b_X is the number of bindings for variable X, the maximum number of distinct tuples that can be obtained for the subgoal is given by $b_X \times b_Y$. Suppose that variable Y can be bound to values a and b, whereas variable X can be unified to values b, c and d. Then, the 2×3 distinct tuples that can be obtained are: s(a, b, a), s(a, c, a), s(a, d, a), s(b, b, b), s(b, c, b) and s(b, d, b). These tuples are called the *instances* of the subgoal. A function called *instance(T)* is defined to compute the number of instances of a term T.

Two additional quantities (functions) are defined for each predicate p in the program: Rel_p, that represents the size of the relation defined by p (i.e., the number of tuples that the predicate generates when all its arguments are uninstantiated), and Sol_p, the solution size for p (i.e., the number of tuples that are obtained for a particular input, as specified by the size values of the (instantiated) input arguments). Function *instance(S)* is used to calculate both quantities, the main difference being that only output variables are considered for the derivation of Sol_p, whereas all variables (input and output) are used for the calculation of Rel_p.

To obtain values for Sol_p and Rel_p, we need to determine values for the number of bindings that are possible for the variables contained in the predicate clause under consideration, denoted by β_v. Two cases are considered: the number of variable bindings for input arguments (denoted by β_{vi}), and the number of variable bindings for output arguments (denoted by β_{vo}). The value β_{vi} for input arguments is derived by using the above-mentioned properties of unification (i.e., function *instance(S)*). On the other hand, β_{vo}

for output arguments is bounded by the product of the number of solutions that are expected from the predicate given a particular input size (as obtained from the size relationships for predicate arguments) and β_{vi} itself. Again, if recursive predicates are present in the program, a set of difference equations will be obtained.

The upper bounds for β_{vi} and β_{vo} can be substantially improved if special cases are also considered. Such special cases include: distinct variables that are bound by the same literal, output variables that are instantiated according to the bindings of the input variables, etc. Similarly, the detection of mutually exclusive clauses can produce more precise results.

To demonstrate how the method is applied, consider the following recursive program which permutes a list of elements:†

```
perm([ ], [ ]).
perm(X, [L|L1]) :- select(L, X, Y), perm(Y, L1).
select(H, [H|T], T).
select(X, [H|T1], [H|T2]) :- select(X, T1, T2).
```

Suppose that we are using "list-length" as the relevant measure, as well as the following input/output mapping:

perm(<$input_a$>, <$output_b$>).
select(<$output_c$>, <$input_d$>, <$output_e$>)

or, in an alternative, expanded form:

$perm_1$(<$input_1$>, <$output_1$>).
$perm_2$(<$input_2$>, <$output_2$>) :- $select_a$(<$output_3$>, <$input_3$>, <$output_4$>), $perm_a$(<$input_4$>, <$output_5$>).
$select_1$(<$output_6$>, <$input_5$>, <$output_7$>).
$select_2$(<$output_8$>, <$input_6$>, <$output_9$>) :- $select_b$(<$output_{10}$>, <$input_7$>, <$output_{11}$>).

†The symbol [] denotes an empty list, i.e., a list with no elements at all.

We will concentrate on the simpler predicate *select*. We try to derive the size of both output arguments, $output_c$ and $output_e$.† The size relations for the first output argument ($output_c$) of predicate *select* are computed as:

$$size(<output_6>) = undefined. \qquad (rule\ select_1)$$
$$size(<output_8>) = size(<output_{10}>) = Sz(select.1,size(<input_7>)). \qquad (rule\ select_2)$$

This system of equations yields:

$$Sz(select.1,size(<input_d>)) = size(output_c) = undefined.$$

The size relations for the other output argument ($<output_e>$) of *select* are as follows:

$$size(<output_7>) = size(<input_5>) - 1. \qquad (rule\ select_1)$$
$$size(<output_9>) = size(<output_{11}>) + 1. \qquad (rule\ select_2)$$

where

$$size(<output_{11}>) = Sz(select.3,size(<input_7>)),$$
$$size(<input_7>) = size(<input_6>) - 1.$$

or, in normalized form:

$$size(<output_{11}>) = Sz(select.3,size(<input_7>)) = Sz(select.3,size(<input_6>)-1)$$

Combining this set of conditions, we finally obtain:

$$Sz(select.3,size(<input_d>)) = size(<output_7>) = size(<input_d>) - 1. \qquad (rule\ select_1)$$
$$Sz(select.3,size(<input_d>)) = size(<output_9>) = Sz(select.3,size(<input_d>)-1) + 1. \qquad (rule\ select_2)$$

This results in a system of difference equations of the form:

$$f(x) = x - 1,$$
$$f(x) = f(x - 1) + 1.$$

In this case, the solution is straightforward (which is not always the case):

$$f(x) = x - 1,$$

or, after variable substitution:

$$Sz(select.3,size(<input_d>)) = size(<output_e>) = size(<input_d>) - 1.$$

---

†For each output argument position, we obtain one equation for each rule which is expressed in terms of its respective inputs: $input_5$ for rule $select_1$, and $input_6$ for rule $select_2$.

The next step is to estimate the number of solutions that predicate *select* is expected to generate. Consider first subgoal $select_b$ in clause $select_2$. The number of bindings for the input variable *T1* (<input$_7$>) is equal to 1 (since every input variable has an initial, unique value). We know that the number of bindings of both output variables *X* and *T2* is bounded by:

$$\beta_{vo} = \beta_{vi} \times Sol(select,size(<input_7>)) = 1 \times Sol(select,size(<input_6>) - 1).$$

In this case, the total number of solutions for $select_2$ equals the number of bindings for the output arguments and is expected to be:

$$Sol(select,size(<input_6>)) = Sol(select,size(<input_6>) - 1).$$

Consider now the other predicate clause, $select_1$. Again, the number of bindings for input variables *H* and *T1* (<input$_5$>) is equal to 1. Since they are also the only output variables, we get:

$$Sol(select,size(<input_5>)) = 1.$$

In other words, the following equations are obtained:

$$f(x) = f(x-1),$$
$$f(x) = 1,$$

which can be combined into:

$$f(x) = f(x-1) + 1,$$

since both clauses are mutually exclusive.

Using boundary conditions (namely, that f(0) = 0 must hold), the final answer is obtained by solving the difference equation†:

$$Sol(select,size(<input_d>)) = size(<input_d>).$$

and we conclude that predicate *select* will generate at most *n* solutions for an input of size *n*.

---

†This particular difference equation has a trivial solution. However, one major challenge faced by Debray and Lin's framework is that many real-life difference equations cannot be solved automatically.

Appendix 2 Primitive Constants in a Uniform Distribution

For the special case of a uniform and independent distribution of attribute values n_{choice_points} is given by:

$$n_{choice\_points} = \frac{n_{tuples}}{K_1 \times K_2 \times \ldots \times K_n}$$

where

- n_{tuples} is the number of tuples in the database for predicate $p(P_1, P_2, \ldots, P_n)$;
- K_k is a *reduction* factor for argument P_k.

$$K_k = \begin{cases} 1, \text{ if no indexing is applied to argument position } k \\ \text{number of distinct values for argument position } k, \text{ if argument indexing is involved and the argument is ground} \end{cases}$$

(note that the value of K_k is the same, regardless of whether hash table collisions occur or not).

Note that this formula assumes a uniform distribution.

$$K_u(k) = \begin{cases} 1, \text{ if the argument is a variable or if the argument position is subjected to indexing} \\ \text{number of distinct values for argument position } k, \text{ if argument indexing is not involved} \end{cases}$$

Thus, if we define

$$U_c(k) = \begin{cases} 1, \text{ if } P_k \text{ is ground,} \\ 0, \text{ otherwise} \end{cases}$$

and

$$U_v(k) = \begin{cases} 1, \text{ if } P_k \text{ is a variable,} \\ 0, \text{ otherwise} \end{cases}$$

we can derive the following final equations:

$$n_{\text{succ\_const\_unif}} = \sum_{k=1}^{n} n_{\text{succ\_unif}}(k) \times U_c(k)$$

$$n_{\text{unsucc\_const\_unif}} = \sum_{k=1}^{n} (n_{\text{unif\_att}}(k) - n_{\text{succ\_unif}}(k)) \times U_c(k)$$

$$n_{\text{var\_unif}} = \sum_{k=1}^{n} n_{\text{succ\_unif}}(k) \times U_v(k)$$

For the trivial case of a uniform distribution, the following formula can be used to calculate F_f, the expected number of tuples that, in average, have to be visited in order to find the *first* solution (n represents the arity of the subgoal).

$$F_f = \frac{\left\{\sum_{k=1}^{n} f(k)\right\} + 2}{2}$$

$$f(k) = \begin{cases} 0, \text{ if } P_k \text{ is a ground term } \textit{and } P_{k-1} \text{ is also a ground term} \\ \left\{\prod_{m=k}^{n} \Theta(m)\right\} - 1 \quad , \text{ if } P_k \text{ is a ground term } \textit{and } k = 1 \\ \left\{\prod_{m=k}^{n} \Theta(m)\right\} - 1 \quad , \text{ if } P_k \text{ is a ground term } \textit{and } P_{k-1} \text{ is not ground} \\ -\left\{\left\{\prod_{m=k}^{n} \Theta(m)\right\} - 1\right\} \quad , \text{ if } P_k \text{ is not ground } \textit{and } P_{k-1} \text{ is a ground term} \\ 0, \text{ if } P_k \text{ is not ground } \textit{and } k = 1 \\ 0, \text{ if } P_k \text{ is not ground } \textit{and } P_{k-1} \text{ is not ground} \end{cases}$$

$$\Theta(m) = \begin{cases} 1, \text{ if } m \text{ is an indexed position and } P_m \text{ is ground} \\ \text{number of distinct values for argument position } m, \text{ otherwise} \end{cases}$$

Table A2.1 shows the corresponding values of F_f for the ternary case, assuming that no position is indexed. Once again, S_k stands for the number of distinct values for argument position k.

Case	Value of F_f
ggg	$(S_1S_2S_3+1)/2$
ggn	$(S_1S_2S_3-S_3+2)/2$
gng	$(S_1S_2S_3-S_2S_3+S_3+1)/2$
gnn	$(S_1S_2S_3-S_2S_3+2)/2$
ngg	$(S_2S_3+1)/2$
ngn	$(S_2S_3-S_3+2)/2$
nng	$(S_3+1)/2$
nnn	1

Table A2.1 Values of the Traversal Factor for the Ternary Predicate Example

Appendix 3 Method of Measurement

The general method to measure the CPU time required to execute a given Prolog query consists of repeating the execution of the query a certain number of times, and taking the average of these measurements. The general scheme is shown in Figure A3.3, where *main* represents the query under consideration. Note that we must discard the contribution to the execution time due to the loop itself.

```
runtest(N) :-
    for(1, N, _),
    runtestforallsolutions,
    fail.
runtest(_).

runcontrol(N) :-
    for(1, N, _),
    runcontrolforallsolutions,
    fail.
runcontrol(_).

runtestforallsolutions :- main.

runcontrolforallsolutions :- control.

control.

for(I, I, I) :- !.
for(I, _, I).
for(I, J, K) :- inc(I, I1), for(I1, J, K).

inc(N,NS) :- NS is N+1.

st(N) :- statistics(runtime, [T0|_]),runtest(N),
    statistics(runtime,[T2|_]),runcontrol(N),
    statistics(runtime, [T1|_]),
    T is ((T2-T0)-(T1-T2))/N,write(T),nl.
```

Figure A3.3 General method to measure CPU execution times

Appendix 4 A Performance Model for QUINTUS Prolog

In this section, the performance model is applied to QUINTUS Prolog for the packages example in [Consens92]†. Essentially, the database contains the following extensional DB predicates:

- *part/3*: an enumeration of 1,640 parts in the system;
- *uses/2*: a set of 4,075 facts which establish which part uses another part.

An intensional DB predicate that can be used to determine the relation "*A is part of package B*" is as follows:

```
part_of(A,B) :- part(B,_,A).
```

The following Prolog query may be used to determine all the packages X, such that X contains a part A that uses a part B which is in turn contained in a package Y different from X:

```
pkg_uses(X,Y) :- uses(A,B), part_of(A,X), part_of(B,Y), \+(X=Y).
```

This conjunction of four subgoals can be re-arranged into several different orders.without affecting the accuracy of the result. Some orderings are forbidden due to the fact that the subgoal `\+(X=Y)` involves negation thus requiring that both of its arguments have a bound value before the evaluation of the predicate is made. The following table (Table A4.1) shows all valid orderings for the query.

---

†Consens, M., Mendelzon, A., and Ryman, A. Visualizing and Querying Software Structures. *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

ordering	subgoal # 1	subgoal # 2	subgoal # 3	subgoal # 4
1	uses(A,B).	part_of(A,X).	part_of(B,Y).	\+(X=Y).
2	uses(A,B).	part_of(B,Y).	part_of(A,X).	\+(X=Y).
3	part_of(A,X).	uses(A,B).	part_of(B,Y).	\+(X=Y).
4	part_of(A,X).	part_of(B,Y).	uses(A,B).	\+(X=Y).
5	part_of(A,X).	part_of(B,Y).	\+(X=Y).	uses(A,B).
6	part_of(B,Y).	uses(A,B).	part_of(A,X).	\+(X=Y).
7	part_of(B,Y).	part_of(A,X).	uses(A,B).	\+(X=Y).
8	part_of(B,Y).	part_of(A,X).	\+(X=Y).	uses(A,B).

Table A4.1 Valid orderings for the query pkg_uses/2

A4.1 Database profile

The packages example can be viewed as a set of extensional and intensional predicates along with the safe queries that are to be applied to the database.

(a) Extensional predicates.

We will assume that they follow a strict uniform distribution of attribute values.

Predicate name	number of tuples	distinct values in argument 1	distinct values in argument 2	distinct values in argument 3
part/3	1640	136	16	1640
uses/2	4075	1203	1288	—

Table A4.2 The extensional database predicates

(b) Intensional predicate.

There is one intensional predicates which is defined upon one of the extensional predicates, a feature that simplifies the analysis.

part_of/2.

(c) Queries.

We consider all eight valid orderings for the *package_uses* example:

:- uses(A,B), part_of(A,X), part_of(B,Y), ~~\+(X=Y)~~. ordering #1
:- uses(A,B), part_of(B,Y), part_of(A,X), ~~\+(X=Y)~~. ordering #2
:- part_of(A,X), uses(A,B), part_of(B,Y), ~~\+(X=Y)~~. ordering #3
:- part_of(A,X), part_of(B,Y), uses(A,B), ~~\+(X=Y)~~. ordering #4
:- part_of(A,X), part_of(B,Y), ~~\+(X=Y)~~, uses(A,B). ordering #5
:- part_of(B,Y), uses(A,B), part_of(A,X), ~~\+(X=Y)~~. ordering #6
:- part_of(B,Y), part_of(A,X), uses(A,B), ~~\+(X=Y)~~. ordering #7
:- part_of(B,Y), part_of(A,X), ~~\+(X=Y)~~, uses(A,B). ordering #8

Note that the built-in predicate has been left out, since the performance model is not applicable to system predicates.

A4.2 Abstract Domains

Table A4.3 describes Debray's domain for this particular query. Table A4.4 indicates the cost domain that applies to the extensional predicates, while Table A4.5 gives the cost domain for the intensional predicate and the different queries.

$\{<Pred_r$	$cpat_{mr}$	$spat_{mr,nr}>\}$
<part/3,	[f,f,f],	[g,g,g]>
<part/3,	[f,f,g],	[g,g,g]>
<uses/2,	[g,g],	[g,g]>
<uses/2,	[g,f],	[g,g]>
<uses/2,	[f,g],	[g,g]>
<uses/2,	[f,f],	[g,g]>

Table A4.3 Debray's domain for all predicates

{<$Pred_r$	$Clause_q$	$cpat_{mr}$	$metrics_{mqr}$	$path\ list_{mqr}$>}
<part/3,	(all clauses)	[f,f,f],	{$cost_1$,n_sol_1}.	[ ]>
<part/3,	(all clauses)	[f,f,g].	{$cost_2$,n_sol_2}.	[ ]>
<uses/2,	(all clauses)	[g,g].	{$cost_3$,n_sol_3}.	[ ]>
<uses/2,	(all clauses)	[g,f].	{$cost_4$,n_sol_4}.	[ ]>
<uses/2,	(all clauses)	[f,g].	{$cost_5$,n_sol_5}.	[ ]>
<uses/2,	(all clauses)	[f,f].	{$cost_6$,n_sol_6}.	[ ]>

Table A4.4 Cost domain for the extensional predicates

Table A4.5 summarizes the specific values of the cost domain for both the intensional predicate and the different queries. (In fact, once the built-in predicate is removed from the queries, the eight valid orderings yield only four distinct queries: (a) ordering #1 = ordering #2; (b) ordering #3; (c) ordering #4 = ordering #5 = ordering #7 = ordering 8; and (d) ordering #6.)

A4.3 Cost metrics

Some cost metrics are summarized in this subsection. The values of the basic constants are specific to QUINTUS Prolog under AIX.

- Head unification probabilities:

  $prob_1$=1 (it always unifies)

- Empirical constants used:

  T_{chp} = 0.040
  T_{vu} = 0.020
  T_{back} = 0.028

- Clause cost metrics: Table A4.6 shows a summary of the cost metrics for all predicates, whereas Table A4.7 provides them for the intensional predicate.

{<$Pred_r$	$Clause_q$	$cpat_{mr}$	$metrics_{mqr}$	$path\ list_{mqr}$>}
part_of/2,	clause#1,	[g,f],	{$prob_1$,$cost_7$,n_sol_7},	[<part/3,[f,f,g],1>]
part_of/2,	clause#1,	[f,f],	{$prob_1$,$cost_8$,n_sol_8},	[<part/3,[f,f,f],1>]
ordering # 1,	-	[f,f,f,f],	{1,cost_ord#1,n_sol_query},	[<uses/2,[f,f],1>,<part_of/2,[g,f],$ninv_{1,1}$>,<part_of/2,[g,f],$ninv_{1,2}$>]
ordering # 2,	-	[f,f,f,f],	{1,cost_ord#2,n_sol_query},	[<uses/2,[f,f],1>,<part_of/2,[g,f],$ninv_{2,1}$>,<part_of/2,[g,f],$ninv_{2,2}$>]
ordering # 3,	-	[f,f,f,f],	{1,cost_ord#3,n_sol_query},	[<part_of/2,[f,f],1>,<uses/2,[g,f],$ninv_{3,1}$>,<part_of/2,[g,f],$ninv_{3,2}$>]
ordering # 4,	-	[f,f,f,f],	{1,cost_ord#4,n_sol_query},	[<part_of/2,[f,f],1>,<part_of/2,[f,f],$ninv_{4,1}$>,<uses/2,[g,g],$ninv_{4,2}$>]
ordering # 5,	-	[f,f,f,f],	{1,cost_ord#5,n_sol_query},	[<part_of/2,[f,f],1>,<part_of/2,[f,f],$ninv_{5,1}$>,<uses/2,[g,g],$ninv_{5,2}$>]
ordering # 6,	-	[f,f,f,f],	{1,cost_ord#6,n_sol_query},	[<part_of/2,[f,f],1>,<uses/2,[f,g],$ninv_{6,1}$>,<part_of/2,[g,f],$ninv_{6,2}$>]
ordering # 7,	-	[f,f,f,f],	{1,cost_ord#7,n_sol_query},	[<part_of/2,[f,f],1>,<part_of/2,[f,f],$ninv_{7,1}$>,<uses/2,[g,g],$ninv_{7,2}$>]
ordering # 8,	-	[f,f,f,f],	{1,cost_ord#8,n_sol_query},	[<part_of/2,[f,f],1>,<part_of/2,[f,f],$ninv_{8,1}$>,<uses/2,[g,g],$ninv_{8,2}$>]

Table A4.5 Cost domain for the intensional predicate and the main query

<clause, cpat>	n_{chp}	n_{vu}	n_sol	**cost=** $(n_1 \times T_{chp} + n_2 \times T_{vu} + n_3 \times T_{back})$
<part/3,[f,f,f],	1640	3×1640	1640	20.99
<part/3,[f,f,g],	1640	2×1640	1	13.12
<uses/2,[f,f],	4075	2×4075	4075	44.01
<uses/2,[f,g],	4075	4075	2.48	24.46
<uses/2,[g,f],	4075/1640	4075/1640	4075/1640	0.22
<uses/2,[g,g],	4075/1640	0	0.0015	0.01

Table A4.6 Cost metrics for all predicates

<clause, cpat>	p(hunif)	n_{chp}	n_{vu}	n_sol	**cost=** $(n_1 \times T_{chp} + n_2 \times T_{vu} + n_3 \times T_{back})$
<part_of/2, [g,f]>	1	1+1640	1+2×1640	1	13.13
<part_of/2, [f,f]>	1	1+1640	2+3×1640	1640	21.00

Table A4.7 Cost metrics for the intensional predicate

A4.4 Query cost formulae

$$\text{cost}(ordering1) = \begin{array}{l}\text{cost}(\text{uses}/2|_{[f,f]}) + nsol_6 \times (\text{cost}(\text{part\_of}/2|_{[g,f]}) + \\ \qquad nsol_7 \times (\text{cost}(\text{part\_of}/2|_{[g,f]}))\end{array}$$

$$= 44.01 + 4075 \times (13.13 + 1 \times (13.13)) = 107053.51$$

$$\text{cost}(ordering3) = \begin{array}{l}\text{cost}(\text{part\_of}2|_{[t,t]}) + nsol_8 \times (\text{cost}(\text{uses}2|_{[g,t]}) + \\ nsol_4 \times (\text{cost}(\text{part\_of}2|_{[g,t]}))\end{array}$$

$$= 21.00 + 1640 \times (0.22 + 2.48 \times (13.13)) = 53784.14$$

$$\text{cost}(ordering4) = \begin{array}{l}\text{cost}(\text{part\_of}2|_{[t,t]}) + nsol_8 \times (\text{cost}(\text{part\_of}2|_{[t,t]}) + \\ nsol_8 \times (\text{cost}(\text{uses}2|_{[g,g]}))\end{array}$$

$$= 21.00 + 1640 \times (21.00 + 1640 \times (0.01)) = 61357.00$$

$$\text{cost}(ordering6) = \begin{array}{l}\text{cost}(\text{part\_of}2|_{[t,t]}) + nsol_8 \times (\text{cost}(\text{uses}2|_{[\ ]([t,g])} + \\ nsol_5 \times (\text{cost}(\text{part\_of}2|_{[g,t]}))\end{array}$$

$$= 21.00 + 1640 \times (24.46 + 2.48 \times (13.13)) = 93537.74$$

A4.5 Comparison between the Model Prediction and the Experimental Results

Table A4.8 presents a summary of the values predicted by the performance model as compared with the values that are obtained experimentally. It should be noticed that the performance model was able to predict the correct order of performance of the queries.

Ordering #	Theoretical value	Experimental value	% Error
1	107054	93021	15.08%
3	53784	46712	15.13%
4	61357	69045	11.13%
6	93537	90204	3.69%

Table A4.8 Theoretical and Experimental Values for the Packages Example