A Unifying Approach For Queries and Updates in Deductive Databases

Wang-chan Wong

Technical Report #91-56

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

UNIVERSITY OF CALIFORNIA

Irvine

A Unifying Approach For Queries and Updates in Deductive Databases

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

by

Wang-chan Wong

1991

The dissertation of Wang-chan Wong is approved,

and is acceptable in quality and form for

publication on microfilm:

[illegible]

[illegible]

[illegible]

Committee Chair

University of California, Irvine

1991

Dedication

To my wife Wendy
whose love, support, patience, and constant prayers
helped me through these seemingly endless years of study.

Contents

List of Figures

Acknowledgements

I would like to give my special thanks to my advisor, Professor Lubomir Bic, for his guidance and support through the years, and for the financial assistance from NSF grant CCR-8709817.

My gratitude also goes to my thesis committee, Professor Tatsuya Suda and Professor Dennis Kibler, for their support and reading of my dissertation.

I would also like to thank my friend Professor Man-tak Shing of Naval Postgraduate School at Monterey, for his moral and spiritual support through his prayers, sharing and encouragement. Finally, I have to thank my ex-officemates, Dr. Jonathan Gilbert, Elke Rundensteiner and Monica Yin, for their friendship, encouragement, discussions, insights, and gossips.

Curriculum Vitae
Wang-chan Wong

1976	B.B.A. (Honors) in Business Administration The Chinese University of Hong Kong
1977	Teaching and Research Assistant, Marketing Department The Chinese University of Hong Kong
1979	M.S. in Business Administration University of California, Irvine
1983	M.S. in Computer Science University of California, Irvine
1983-1986	Teaching Assistant, ICS Department University of California, Irvine
1987-1988	Research Assistant, ICS Department University of California, Irvine
1989-1990	Lecturer, Computer Science Department Calfornia State University, Dominguez Hills
1990-present	Assistant Professor, Computer Science Department Calfornia State University, Dominguez Hills
1991	Ph.D. in Computer Science University of California, Irvine Dissertation: "A Unifying Approach for Queries and Updates in Deductive Databases"

Publications

Wong, Wang-chan and Lubomir Bic, "A Tagging Scheme to Prevent Infinite Recursion in First-Order Databases" in *Proceedings of the Second International Conference on Computers and Applications, June, 1987*, IEEE.

Wong, Wang-chan, Tatsuya Suda, and Lubomir Bic, "Performance Analysis of A Message-Oriented Knowledge-Base" in *Proceedings of the Int'l Seminar on Performance of Distributed and Parallel Systems, IFIP WG 7.3*, (Edited by Yutaka Takahashi), North-Holland, Kyoto, December, 1988.

Wong, Wang-chan, Tatsuya Suda, and Lubomir Bic, "Performance Analysis of A Message-Oriented Knowledge-Base" in Transactions on Computers, IEEE, Vol 39, No 7, July 1990.

Technical Reports

Wong, Wang-chan and Lubomir Bic, "Efficient Recursion Termination For Function-Free Horn Logic", TR 86-26, ICS Dept, University of California, Irvine, 1986.

Wong, Wang-chan, Tatsuya Suda, and Lubomir Bic, "Performance Analysis of A Message-Oriented Knowledge-Base", TR 87-11, ICS Dept, University of California, Irvine, 1987.

Working Papers

Wong, Wang-chan, "On Terminating Infinite Recursion for Function-Free Horn Logic", Parallel Computation Notes, No.4 Parallel Computing Group, ICS Dept, University of California, Irvine, August 1987.

Wong, Wang-chan, "On Compilation of Recursive Rules", Parallel Computation Notes, No.5, Parallel Computing Group, ICS Dept, University of California, Irvine, Sept, 1987.

Fields of Study

Major Field: Computer Science

Studies in databases, parallel and distributive computing

Professor Lubomir Bic

Studies in system performance evaluation

Professor Tatsuya Suda

Abstract of the Dissertation

A Unifying Approach For Queries and Updates in Deductive Databases

by

Wang-chan Wong

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1991

Lubomir Bic, Chair

This dissertation presents a *unifying approach* to process (recursive) queries and updates in a deductive database. To improve query performance, a combined top-down and bottom-up evaluation method is used to compile rules into iterative programs that contain relational algebra operators. This method is based on the *lemma resolution* that retains previous results to guarantee termination.

Due to *locality* in database processing, it is desirable to materialize frequently used queries against *views* of the database. Unfortunately, if updates are allowed, maintaining materialized view tables becomes a major problem. We propose to materialize views *incrementally*, as queries are being answered. Hence views in our approach are only *partially* materialized. For such views, we design algorithms to perform updates *only* when the underlying view tables are actually affected.

We compare our approach to two conventional methods for dealing with views: *total materialization* and *query-modification*. The first method materializes the entire view when it is defined while the second recomputes the view on the fly without maintaining any physical view tables. We demonstrate that our approach is a compromise between these two methods and performs better than either one in many situations.

It is also desirable to be able to update views just like updating base tables. However, view updates are inherently ambiguous and the semantics of update propagation on recursively defined views were not well understood in the past. Using *dynamic logic programming* and lemma resolution, we are able to define the semantics of recursive view updates. These are expressed in the form of *update translators* specified by the database administrator when the view is defined. To guarantee completeness, we identify a subset of *safe* update translators. We prove that this subset of translators always terminate and are complete.

CHAPTER 1
Introduction

This dissertation deals with the issues of integrating *logic programming*, as represented by the language Prolog (PROgramming in LOGic), and relational databases to generate an intelligent database system. In particular, we address the issues on improving the expressiveness of a query language, efficiency of query processing, updates on base tables, and updates on view tables. In recent years, there have been many studies devoted on combining logic programming with a relational database system. These research efforts were accelerated by the fact that Prolog was selected as *the* language for the intelligent inference engine and relational database was chosen for storing large collections of data by the Japanese *Fifth Generation Project* [ITOH86].

The research of combining Logic Programming with the relational model is by no means *coincidental.* The relational model as expressed by relational algebra is not expressive enough for some applications. For example, a view that is a transitive closure of a base table cannot be expressed with a single stand-alone relational algebra expression without using a loop. This deficiency is a major handicap of the relational model. Transitive closure can be expressed simply as a single recursion. Since logic programming is good at expressing recursive definitions, it becomes a good candidate for a relational query language. Therefore, the first issue to improve the expressiveness of a query language can be achieved by using logic programming as a query language.

The studies on combining logic programming with relational systems are mainly focused on *recursive query processing*, ignoring the necessity to solve the update problem at the same time. From our experience, the performance of a database system is closely related to both query and update processes. Using logic programming to handle recursive query processing is fine but it only addresses half of the problem. We believe that there should be a unifying approach that allows us to address both queries and updates within the same framework while at the same time maximizing the overall performance. This is the primary motivation of this research. In this dissertation, we demonstrate that such an approach exists. The approach is based on a modified SLD resolution by retaining materialized queries as lemmas [VIE87]. With lemma resolution, we show how a recursive query can be compiled. We further show that the derived results of a query should be retained permanently (as materialized views) in order to speed up future query processing time. However, the materialized views would cause trouble if updates were allowed. Within the same framework of lemma resolution, we show that the views can be materialized and updated *incrementally.* Hence it is a compromise between a completely materialized scheme that materializes the total views when views are defined, and the on-the-fly approach that does not materialize a view all at once but derives the view whenever it is needed. Combining lemma resolution, query compilation and dynamic logic programming, we further propose the semantics of view updates.

The lemma resolution provides us with a framework that allows us to address the problems in both query and update processing. First, based on lemma resolution, we design our incremental query and base update processing algorithms. Second, by expanding the lemma resolution, we define the semantics of recursive view updates.

In the remainder of this chapter, we shall first describe the fundamentals and the background on the subject matters.

1. Deductive Databases – What are they?

In this section, we informally define what a deductive database is and also define the terminology used in this dissertation.

A database is a collection of data organized to provide information necessary for the operation of the enterprise that it is supposed to serve. According to the ANSI/SPARC architecture of a database [JAR77], there are three levels of a database system: the external model that describes the views of different users to the database; the conceptual model (logical model or data model) that captures the semantics of the application domain; and the internal model that specifies the storage methods, the access path, the communication network (if the data are distributed). The basic principle in a modern database is to make the users *data independent.* That is, when using a database system, the user should not be concerned with the data storage or the access paths taken when a piece of information is needed.

The relational database was designed with this objective in mind [CODD70]. As a matter of fact, the relational model was born as a simplification of complex hierarchical and network models, in which the users have to know the pointers, the access path, and the storage methods in order to use the data. In the relational model, facts are organized in a finite collection of relations (i.e. tables) with the relational operators such as insertion, deletion, selection, projection, product, join, union, intersection and difference [CODD72]. Every relation (table) is represented by a finite set of tuples. A database user accesses the information stored in these tables by means of queries expressed in a query language (e.g. SQL, Structural Query Language). The database management system plays the role of an interface between the user and the physical database, and is responsible for accepting the user's queries and updates as well as returning answers to the user.

1) $p_1(a,b)$.
2) $p_1(b,c)$.
3) $q(X,Y) : -p_1(X,A), q(A,Y)$.
4) $q(X,Y) : -p_1(X,Y)$.
5) $d(X,Y) : -p_2(X,A), l(A,Y)$.
6) $l(X,Y) : -p_3(X,A), f(A,Y)$.
7) $f(X,Y) : -d(A,Y), p_3(X,A)$.
8) $d(X,Y) : -p_4(X,Y)$.
9) *(query)* :- $q(a, Ans)$.

Figure 1

An Example of Horn Database

Conceptually, a deductive database can be viewed as a relational database with an *inference engine* that is capable of complex reasoning from the content of the relational database. Logic programming is viewed as a better query language than conventional relational languages such as SQL because of its expressiveness. The following is a brief description of the syntax of a logic program. We assume readers are familiar with logic programming, specially, the concepts of unification and resolution as described in [KOW79B], [LLO84], [CGT90] and [CM81].

A logic program is a collection of *clauses* of the form:

$$p_n : -p_0, p_1 \ldots p_{n-1}$$

Each p_i is called a *literal* and has the form $p(t_1, ..., t_m)$, where p is a *predicate* symbol and $t_1, ..., t_m$ are *terms*. The semantics of this clausal form can be interpreted procedurally [KOW79A, KOW79B]: if p_0 and p_1 and .... p_{n-1} are true then the literal p_n is true. The literal p_n on the left hand side, which is a *positive* literal, is called the *head* of the clause; the remaining literals form its *body* and are *negative*.[1] A Horn clause contains *at most* one positive literal. A database that is

[1] The words *negative* and *positive* refer to the situation when the implication of the clausal form is rewritten in its disjunction form. For example, the above clause

comprised of nothing but Horn clauses is called a *Horn database*. A Horn database is *function-free* if the terms are restricted to either constants or variables (i.e. no functions or structures.) In this dissertation, we are interested in function-free Horn programs. A clause with an empty body is called an *assertion* and is used to represent explicit facts. Clauses with a non-empty body are called *rules*. A query is a *goal clause* that has an empty consequent (i.e. a clause with only a body but no head.) For instance, clause (9) in Figure 1 is a query. In the logic programming sense, the goal is to prove the satisfiability of *q(a, Ans)*. If the goal is satisfiable, the bindings of variable *Ans* from the unification are returned.

The relationship between logic programming and the relational database is very close. A literal in a Horn database is a *relation* of the terms of that predicate. Relations defined by assertions (EDB) are called *base* relations while relations defined as the head of a rule will be referred to as *virtual* relations or, in a relational model, *views*. For example, p_1 in Figure 1 is a base relation and q, d, l and f are virtual relations, or derived relations, or views.

The query in Figure 1 can be implemented with the following relational algebra expression:

$$\pi_2(\sigma_{1='a'}Q)$$

where Q is the view table (defined by q), π_2 denotes the projection of the second attribute of Q, and $\sigma_{1='a'}$ is a selection of tuples of Q where the first attribute is equal to the constant a. The entire expression simply means to select the tuples of Q where the first attribute is equal to 'a'. The resulting table of the selection is then projected to return only the second attribute.

Non-recursive views expressed by logic programming can be translated to relational algebra expression directly. Consider the following rule as an example.

$$t(X,Y) : -p(X,A), q(A,Y).$$

can be expressed as: $\neg p_0 \vee \neg p_1 \vee \ldots \vee \neg p_{n-1} \vee p_n$, where $p_0, p_1, \ldots p_{n-1}$ are negative literals and p_n is a positive literal.

If the literals p and q are non-recursive, the rule is identical to the following relational algebra expression that defines the view t:

$$\pi_{X,Y} P \bowtie_{2,1} Q$$

where π is a projection and $\bowtie$ denotes a join. The expression then says to join the tables P (defined by literal p) and Q (defined by literal q) on the second attribute of P with the first attribute of Q; the join results are then projected out to the pair of X and Y attributes. A view of this type is called *Select-Project-Join* or simply an *SPJ* view.

In a deductive database, a relation is defined either as a base or a virtual relation, but not both. This divides the database into two parts: the extensional database (EDB), comprising all base relations, and the intensional database (IDB), comprising all rules.

2. Mismatch Between Logic Programming and Relational Databases

Even though logic programming and relational databases are closely related, there exists two major discrepancies between them. These make it very difficult to combine them in an efficient way. In this section, we describe these differences.

2.1. Functionality Mismatch

The first mismatch concerns the functionalities of logic programming and the relational database. These two formalisms were designed for very different purposes and, as a result, are quite different in their behavior.

We list the differences between them as follows:

(1) Relational algebra, which defines operations in a relational database, is procedurally oriented, while logic programming tends to be specification oriented. The relational algebra basically describes the procedures to compute the results. For example, the expression $\{\pi_{X,Y}(\sigma_{X='a'}P \bowtie Q)\}$ prescribes

the following steps. First, select tuples in table P if the attribute value of X is equal to a. Then the resulting table is naturally joined with table Q. Finally, the joined results are projected to the X and Y columns. On the other hand, a logic programming clause is just descriptive. For example, the clause $p(X,Y) : -q(X,A), r(Y,A)$. means the following: to conclude the fact that there is a relation p between X and Y, one has to show that $q(X,A)$ and $r(T,A)$ are true. The variable names X, Y, A will eventually be bound to some constants if the query is successful. However, the clause does *not* include any instructions on how to compute these bindings. In fact, the procedure is already built-in in the resolution process, for example, the interpreter of the depth-first SLD resolution. Thus, to integrate logic program to relational database, it is necessary to integrate the resolution method with the relational database.

(2) In a relational database two major operations are required, query processing and updates. Unfortunately, update in logic programming (e.g. Prolog) are not quite well understood. For example, insertion and deletion in Prolog are implemented with non-logical predicates such as *asserta*, *assertz*, and *retract*. These predicates are *ad hoc* in nature and their semantics are not well defined. (The details of these problems are discussed in Chapter 5.)

(3) A relational database typically deals with large quantities of data while in logic programming, data occurs in much smaller quantities. On the other hand, rules in logic programming are much more frequent than views in a relational database. A useful deductive database must be able to handle fairly large amounts of rules and data at the same time. This implies that both the inference engine and the database manager have to cooperate in a way that neither component becomes the bottleneck of the other.

(4) In a relational database, recursive views cannot be expressed in terms of relational calculus or algebra. However, recursion is the second nature of logic programs. Therefore, any integration of logic programming and relational databases has to solve the problem of how to represent recursion, including complex recursion.

2.2. Performance Mismatch

The second mismatch concerns performance. Logic programming derives results in a tuple-at-a-time manner while a relational database works in a set-at-a-time manner. Logic programming further suffers from poor performance, mainly due to its use of depth-first search and backtracking. Since there are large quantities of data in any real-life application, combining logic programming with a relational database magnifies the performance problem if the method of computation does not change.

There are basically two approaches to coupling of logic programming with a relational database: loose-coupling, and tight-coupling. In a loosely-coupled deductive database system, the inference engine is no more than a front end query processor. In this case, the inference engine will process the rules and the database will handle the usual database operations on the base tables, separately. The inference engine can still work a tuple-at-a-time with backtracking. Whenever it wants a tuple, it sends a request to the database manager for retrieval. The results of the request (in a set) are returned to the inference engine. The resolution resumes by taking the required tuple. Hence the interaction between the inference engine and the database is minimal. This is the easiest way to integrate logic programming and relational database but, obviously, not an efficient one.

With a *tightly-coupled* integration, either the logical inference engine is expanded to include database operations *or*, the relational database is extended to include the inference engine. For the first case, several studies have been done in this direction. The most notable one is the LDL project of Microelectronics and Computer Technology Corporation (MCC) [TSUR86, BEER87]. The LDL project had two specific goals: (1) design a new language LDL (Logic Data Language) which is a declarative language for data-intensive applications. The language is an extension of the function-free Horn logic and is able to work with set-at-a-time data retrieval, negation and some base updates. (2) develop a system supporting LDL, which integrates the language with all the necessary database operations such as transaction management, secondary memory access, integrity constraints and recovery.

The other type of tightly-coupled integration is to put the inference engine inside the relational database. This approach is sometimes referred to as *rule compilation.* The idea originated with Henchen and Naqvi [HN84] who suggested that a recursive rule can be *compiled* into an iterative program containing relational algebra expression. The compiled program is then placed inside the relational system like a procedure. Upon receiving a recursive query from the user, the appropriate *procedure* is invoked. Hence, the compilation of a query has only a one-time cost. Once the queries are compiled and placed inside the database system, the external inference engine is no longer needed.

3. Background and Related Work

As noted above, past studies on relational databases and deductive databases have concentrated on query processing or updates, but *not both.* It is our intention to unify these two issues in a single framework to generate an intelligent database system. Nevertheless, there are many separate and independent studies on which our studies are based. In this section, we describe some of the important results and the major principles which influenced our studies.

3.1. Recursive Query Compilation

Gallaire, Minker, and Nicolas [GM78, GMN84] laid the groundwork for of deductive databases. Following the pioneering work by Chang [CHANG81], McKay and Shapiro [MS81], and Henschen and Naqvi [HN84], numerous methods have been proposed to evaluate recursive queries more efficiently by compiling the rules into iterative programs. In general, there are two types of evaluation methods, *top-down* and *bottom-up*. Top-down evaluation or *backward chaining* tries to verify the premises which are needed in order for the conclusion to hold. In a top-down evaluation, the initial goal is unified with the left-hand side of some rule (i.e. the head), and generates a *resolvent* that corresponds to the right-hand side literals (i.e. the body of the matched rule.) A resolvent contains more goals to be resolved. The process is continued until the proof tree is completed.

Top-down evaluation can further be distinguished into *breadth-first* or *depth-first*. In a depth-first top-down evaluation, the generation of resolvents is governed by the ordering of the right-hand side subgoals (the so-called selection function.) Most commonly, the left-most literal becomes the next goal to be resolved. Hence, this process produces a search tree that grow in a depth-first manner. The Prolog interpreter, for example, uses a top-down depth-first evaluation. In a breadth-first top-down evaluation, resolvents are generated for all matched rules and thus produce the search tree one level at a time. In generating the next resolvents, top-down evaluation method takes advantage of the bound argument (i.e. the constants) of a given goal. Hence, the search in a top-down method is more *focused* than in a naive bottom-up method.

Bottom-up evaluation or *forward chaining* starts from the existing facts to infer new facts, thus proceeding toward the conclusion. Bottom-up evaluations consider rules as *productions* that are applied to the facts to produce all possible consequences. A subset of these consequences that satisfy the goal become the answer. A naive bottom-up approach (e.g. as in McKay and Shapiro [MS81]) does not take advantage of the bound arguments (i.e. the constants) of the goal and will generate far more consequences that may not be relevant to satisfying the goal. There are several variants of the bottom-up evaluation method, for example, the Magic Sets [BMSU86], or the Counting and Reverse Counting – a variant of Magic Sets [BR86]. These variants rewrite the original rules according to some algorithms and thus perform better than the naive bottom-up evaluation method.

Details of the top-down and bottom-up evaluation methods plus several other variants can be found in the landmark study of Bancilhon and Ramakrishnan [BR86]. In this study, a specific application domain, i.e. a simple single recursive query, is chosen to evaluate each of these methods. Their results suggest that top-down evaluation, such as the one by Henchen and Naqvi, performs best among all.

3.2. Termination Problem of Top-Down Evaluation

Unfortunately, the top-down evaluation method suffers a major drawback – the termination of the search process. For instance, the compilation method of Hanchen and Naqvi only works for linear recursion. There have been many studies on how to terminate the search process of the top-down evaluation. For example, in [BW84], Brough and Walker identified two distinct approaches to solving the termination problem: a *goal termination* strategy and a *rule termination* strategy. With *goal termination*, a branch of a derivation tree is terminated if a goal is repeated in its own branch of the derivation tree. With *rule termination*, a branch of the derivation tree is terminated if a rule (clause) with the same instances is repeated in the existing derivation tree. Unfortunately, as shown by Brough and Walker in that same study, any *preorder* search strategy (i.e. top-down left-to-right) where termination is based on examining the current partial derivation tree, is incomplete, i.e. it must miss some answers. That includes both rule termination and goal termination.

We have also proposed a solution to the termination problem by transforming all recursive rules into *safe* loops which are left-recursion free [WONG86, WONG87]. The goal termination strategy is then implemented using a simple and efficient coloring scheme by distributing the goal history containing all executed goals over the facts (the assertions), rather than maintaining it as a centralized data structure. Unfortunately, our scheme is also incomplete, i.e. it also suffers the termination problem for arbitrary rules.

Several other methods were also proposed to terminate the search process by using the previously computed results in the resolution process, for example, the Extension Table Method [DW85, DW86] and the OLDT Method [TSP86]. These methods are based on the *memoing* idea as in the parsing algorithm by Earley [EAR70]. Hence, they are sometimes referred to as *Earley deduction.* Unfortunately, they are also incomplete for arbitrary recursive rules.

Finally, a method which we refer to as *lemma resolution* in this dissertation was proposed by Vieille [VIE87]. The lemma resolution is similar to the above memoing approaches. In fact, the lemma resolution combines both top-down and bottom-up evaluation approach in the resolution process and is proven to be complete (i.e. it terminates). (As a side note, we had also developed an idea very similar to the lemma resolution but were a step behind the work of Vieille in publication. The term Lemma Resolution was, in fact, coined in our paper [WONG87B].)

With lemma resolution, we show in Chapter 2 that any recursive query can be compiled into an iterative program containing relational operations that can be executed in a relational database very efficiently.

3.3. View Materialization and Base Updates

Since we need to retain previously computed results in order to guarantee termination of the search process, the next logical step is to retain them permanently in the database. Retaining query results is called *view materialization* in the database literature. More studies have been done on this issue in database research than in logic programming. Researchers in databases have been aware of view materialization for quite some time, even though most relational systems still use the *query modification method* proposed by Stonebraker [STONE75]. The query modification method does not retain the results. All queries on views are computed on-the-fly when they are evaluated.

Several studies have proposed to keep the materialized view in order to speed future repeated queries. However, maintaining the materialized view tables becomes a big problem if updating is allowed. As mentioned earlier, updates are the one of the weakest areas in logic programming. At the same time, updates are essential in databases and thus must also be supported in deductive databases. The main problem is that updating the base tables may affect the consistency of the view tables.

Updates can further be divided into two types: *base updates* and *view updates.* Base updates refer to updates on the base tables. For example, the insertions and deletions in table P_1 of the database of Figure 1 are base updates. On the other hand, users should be able to insert or delete tuples in the view tables themselves if they are kept permanently. This type of update is called *view update*. For example, literal f in Figure 1 is a virtual literal. The materialized table of f is the view table F. Updates performed directly on the table F are called view updates.

Several approaches have been proposed to address the base update issue, for example, *immediate view maintenance* and *deferred view maintenance* as described in [HAN87]. If every update were to trigger recomputation of the entire view, then the cost of maintaining view tables would definitely outweigh the speedup of some potential future queries. Therefore, any view maintenance scheme must have an efficient *screening* test that determines if the update would affect the view. The screening test is important because it directly affects the overall performance. Several screening algorithms have been proposed, for example, in [BC79], [BLT86], and [HAN87]. However, most of these studies apply to only a specific type of view: the so-called select-project-join view which is not recursive.

View maintenance is also similar to the *Truth Maintenance System (TMS)* in AI research [DOY79]. However, truth maintenance addresses the updates on both assertions (facts) and rules. Hence the system may become non-monotonic [MD80]. In our approach, we only consider updates on facts. Rules are assumed to be fixed and are not updatable in our systems.

In Chapter 3, we address these issues by linking query and base update into a single framework.

3.4. Recursive View Updates

If we allow users to update base tables, it would be desirable to also allow them to update the materialized view tables. However, updates on views must be propagated to all supportive tables so that the updated tuples are supported. For example, consider deleting a tuple from a view. If the tuple is simply deleted from the view table, it could be derived in the future, since the underlying supportive tables have not been updated. Unfortunately, propagation of update effects can be ambiguous. In general, there are two distinct approaches to handle view updates: the *theoretic approach* and the *heuristic* approach. In the theoretic approach [FUV83, RN88], view updates are translated into new "theories" and are stored alongside the original rules. Undeniably, these newly added theories are complete and will entail the updates. Unfortunately, this approach has been shown to be intractable since the new theories are mostly non-Horn and contain large numbers of negations. Hence, such an inference engine is not suitable to be integrated with a relational system containing large quantities of data.

The heuristic approach, on the other hand, depends on the database administrator to designate the semantics of updates because view updates are inherently ambiguous. The ambiguity exists at two levels: at the rule level, and within the rule. At the rule level, if a view is defined by multiple rules, it is not clear which of the rules the update should propagate to. For example, consider the view q defined by rules (3) and (4) in Figure 1. If the user updates the view table Q, the effect could be propagated to rules (3) or (4) or both.

Within a rule, the update propagation is also ambiguous due to multiple literals. For example, assume that we decide to propagate the update effects of q to rule (3). It is not clear which of the two literals (or both) should be updated. The heuristic approach, as in [KEL85] and [MW88], argues that the database administrator should be able to designate the update actions based on his/her experience and the users' requirements.

Unfortunately, the semantics of logic programming are not sufficient to describe the dynamic nature such these update activities. However, based on dynamic logic programming proposed by Harel [HA79], Manchanda and Warren defined the semantics of update translators that allow the database administrator to designate the update effect propagation [MW88]. These translators serve as update procedures to propagate the update effects into the designated tables. Unfortunately, their method cannot deal with recursive view updates, which, in their own words, are "not well understood" and hence, are not allowed. If the view is defined recursively, and if the effect is propagated to the recursive literal, the meaning of such an operation is not well understood. For example, if the update effect of q is to be propagated to the literal $q(A,Y)$ in rule (3) of Figure 1, what could happen? In Chapter 5, based on the lemma resolution, we propose an extension to their update translators that define the semantics for recursive view update. The extended update translators, however, are incomplete in general since they may not terminate for arbitrary update requests. We further identify a subclass of safe update translators that terminate and hence are complete.

To summarize, we unify query processing, base updates, and view updates within the same framework of lemma resolution to make the relational system more intelligent.

4. Major Contributions

The deductive database framework proposed in this dissertation overcomes the two types of impedance mentioned in section 2 while integrating logic programming with relational databases. The integration is in terms of rule compilation based on lemma resolution. We chose the rule compilation approach for two reasons. First, it is not necessary to re-define the concepts of relational databases and logic programming since we do not need to augment either of these two formalisms. Second, the implementation is more straightforward, since compiled programs can be stored as data objects and can be called within the relational database.

We propose a unifying approach that integrates both query processing and updates within the framework of lemma resolution. To be specific, we have accomplished the following:

(1) Based on lemma resolution, we designed an algorithm for query compilation that allows us to incrementally materialize views.

(2) We designed a query process that works with the compiled queries and the partially materialized view tables with the ability to improve overall performance.

(3) We designed an algorithm to allow incremental updates. Our algorithm does not require views to be computes in their entirety. Instead, smaller partially materialized views are maintained;

(4) We further demonstrated using empirical studies that our incremental view update method is superior to both the totally materialized method and the query modification (on-the-fly) method in many cases;

(5) Based on lemma resolution and dynamic logic programming, we further defined the semantics of recursive view update translators to allow updates of recursive views;

(6) Lastly, we proved that our view update translators are correct and complete and will terminate.

5. Roadmap

The organization of the dissertation is as follows: in Chapter 2, we describe the lemma resolution. We further propose a compilation method that transforms any complex recursive query into an iterative program using relational algebra expressions.

In Chapter 3, we propose a query processing strategy that incrementally materializes the view tables. We further propose a method to efficiently implement this query processing strategy. Moreover, we propose to retain the lemmas even after the query is successfully answered in order to speed up repeated future queries. However, since the views are materialized, the view validation becomes an issue. We propose a method that allows updates on the base tables to be screened to decide if the update effects should be propagated to the view table or not.

In Chapter 4, we compare the behavior of the method proposed in Chapter 3 against the other two methods: total materialization and on-the-fly methods. We show that our method is a compromise between these two and, in many cases, outperforms both of them. We also discuss the factors that are important in determining which method to use.

In Chapter 5, based on lemma resolution and dynamic logic programming, we propose a new method to define the semantics of recursive view updates. We further propose the semantics of the view translators. We identify a subclass of view translators that is capable of updating recursive views, is complete, and will always terminate.

In Chapter 6, we list the topics for future studies and give a general discussion regarding this dissertation.

CHAPTER 2
Lemma Resolution and Recursive Query Compilation

In the past five years, different methods have been proposed to integrate deductive rules expressed in function-free Horn clauses with existing relational database systems. From a performance point of view, the most desirable approach is to *compile* the given Horn clauses into a sequential program containing database operations expressed in a relational query language such as SQL. The main advantage of the compilation approach is that, once the program starts execution, it does not have to make any explicit references to the original Horn clauses.

There exist two possible approaches to compilation, resulting in programs evaluated in either a *top-down* [HN84] or *bottom-up* [ULL85, ULL89, BMSU86, SZ86, MS81] manner. It has been shown that top-down evaluation of single recursion has better performance since the bindings of the original query are utilized to restrict the search space [WHA88, BR86]. However, the top-down evaluation method suffers from a serious drawback, namely *termination*. Consequently, all proposed top-down compilation methods are restricted to linear recursion only [HN84, WHA88].

The original bottom-up approach does not have the termination problem, however, its performance lags behind the top-down method, despite numerous efforts to optimize the program evaluation by the so-called "sideways information passing" [ULL85, BMSU86, SZ87]. Details of the arguments can be found in [WHA88, BR86, HN84]. Although Ullman recently argued that the bottom-up evaluation can be implemented more efficiently than most top-down evaluation methods for complex recursions [ULL89], the lemma resolution described in this chapter is proven to be equally efficient [VIE88]. Furthermore, the method proposed in [ULL85A] needs to rewrite the original rules, which destroys the intuitive semantics of the rules. Lastly, the view update semantics defined in Chapter 5 are based on top-down evaluation. It is still unclear how the semantics of recursive view update can be intuitively defined for the bottom-up evaluation method. Thus, in this dissertation, we concentrate on the top-down evaluation method.

Several closely related solutions to the termination problem for top-down evaluation have been proposed [VIE87, TS86, DW86]. We shall refer to this method as *lemma resolution*, since the basic idea is to retain previously resolved subgoals referred to as "lemmas", i.e., temporary results that contribute to solving the original query (goal). Lemma resolution essentially combines top-down and bottom-up evaluation, as will be explained in Section 1.

In this chapter we present a method for processing recursive queries based on lemma resolution. It has the following properties:

1. It is a compiled method and hence it preserves the efficiency of a top-down method;
2. It is guaranteed to terminate and generate all possible answers in the presence of any form of function-free recursion, including non-linear recursion.

The organization of this chapter is as follows: in Section 1, we briefly describe the lemma resolution. In Section 2, we present the compilation method that is based on the lemma resolution. In Section 3, we consider different methods for performance improvement.

1. Lemma Resolution

The process of resolution can conveniently be viewed as a tree, where each node represents a goal (the body of a clause) and each edge indicates dereferencing between two clauses. Each literal inside a node is called a subgoal. We first define

(1) *ancestor(X, Y) :- ancestor(X,Z), ancestor(Z, Y).*

(2) *ancestor(X, Y) :- parent(X, Y).*

(3) *parent(a,b).*

(4) *parent(b,c).*

(5) *?-ancestor(X, Y).*

Figure 2

An Example of a Double Recursive Program

the concepts of a subgoal instance and an expandable node. Given two identical recursive subgoals G_1 and G_2, G_2 is an *instance* of G_1 if $G_2 = \theta G_1$, where θ is a proper substitution. A node that begins with a subgoal instance will be called an *expandable node*. (The justification for this terminology will be given shortly.)

With these definitions, we now give an intuitive description of the lemma resolution process. It starts with the given query and generates a sequence of trees as follows. The first tree is generated by trying all possible paths of the search tree (using SLD resolution[2]) until one of the following situations occurs: (1) the empty clause is derived, i.e., a solution is found, (2) the path fails, or (3) a recursive subgoal instance is encountered. As a result, a leaf of this first tree will be either the empty clause, a failure, or an expandable node (with a recursive subgoal instance as its left-most subgoal.) Solutions to any recursive goals in this tree become lemmas.

To illustrate this first step, consider the program in Figure 2. Starting with the initial query *ancestor(X, Y)* as the root, we apply rules 1 and 2, which yield the two resolvents N_1 and N_2 as shown in Figure 3. (Note that we abbreviated the predicate names *ancestor* and *parent* to *an* and *par*, respectively.) Since the first subgoal of N_1 is an instance of *ancestor(X, Y)* in N_0, N_1 does not expand any

[2] In the following discussion, we assume that a breadth-first search strategy is used. The lemma resolution works for both breadth-first and depth-first search.

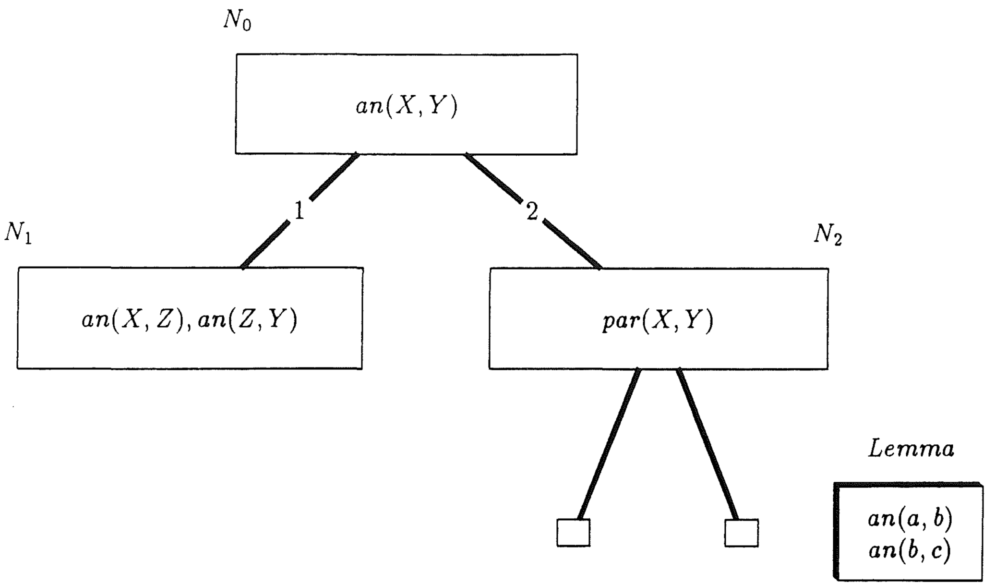


Figure 3

The First Tree of Lemma Resolution

further; it becomes a leaf of the first tree. N_2 is resolved with the two assertions (line 3 in Figure 2); both succeed, yielding the two lemmas *ancestor(a, b)* and *ancestor(b, c)*. This completes the first tree.

Subsequent trees are generated by expanding the immediately preceding tree as follows: the leftmost subgoal of each expandable node N is resolved with all unifiable lemmas that were generated in previous trees but have not yet been used to resolve the same node N. Similar to the first tree, each branch expands until one of the following occurs: (1) the empty clause is derived, (2) a failure occurs, (3) the process encounters a recursive subgoal instance that is not unifiable with any lemmas generated in previous trees. New lemmas generated as a result of the expansion are added to the current set of lemmas. The resolution process terminates when there is no new lemma generated that could further resolve any expandable node.

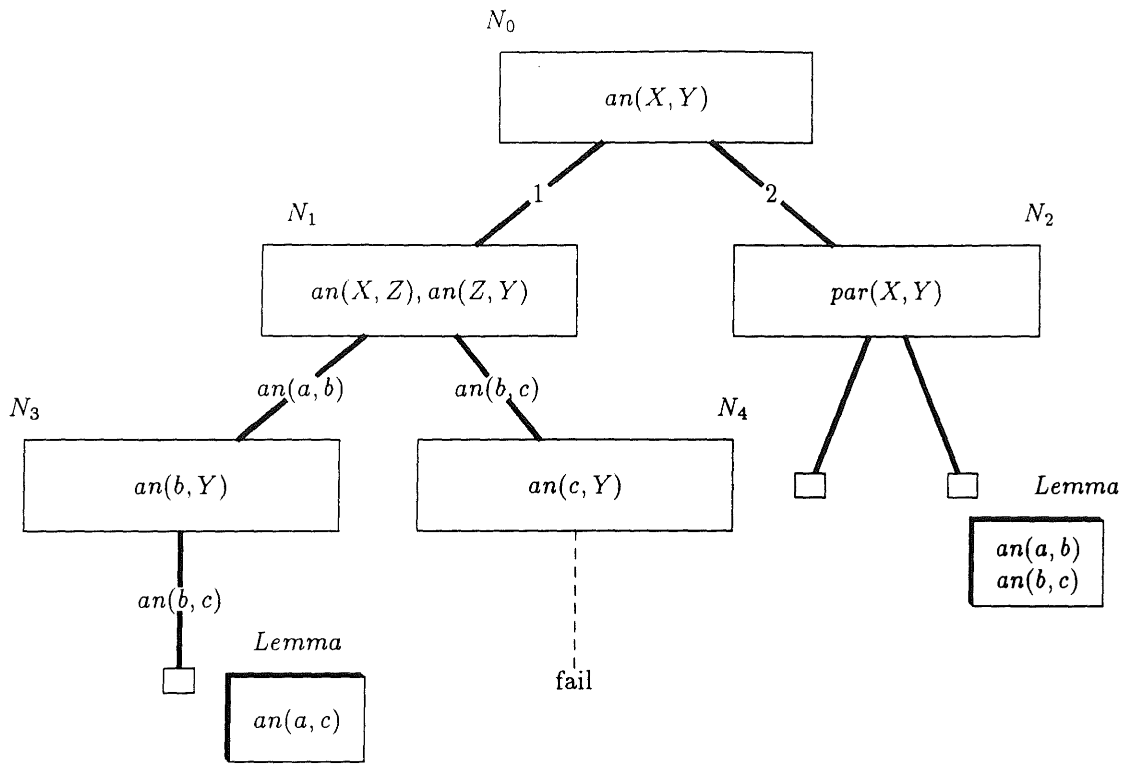


Figure 4

The Second Tree of Lemma Resolution

Returning to our example, Figure 4 shows the tree derived by expanding the tree in Figure 3. In particular, the expandable node N_1 is resolved with the two lemmas *ancestor(a, b)* ad *ancestor(b, c)* that were generated in the first tree. This yields the new resolvents N_3 and N_4, respectively. These two resolvents are both expandable. Since there is a unifiable lemma *ancestor(b, c)* for N_3, the process continues. It derives the empty clause, which yields the new lemma *ancestor(a, c)*. The second path from N_1, on the other hand, leads to failure.

Since a new lemma has been generated, the process continues as shown in Figure 5. The lemma *ancestor(a, c)* is used to resolve the expandable node N_1, resulting in the new resolvent N_5. This is also an expandable node but is not unifiable with any existing lemmas; hence the branch fails.

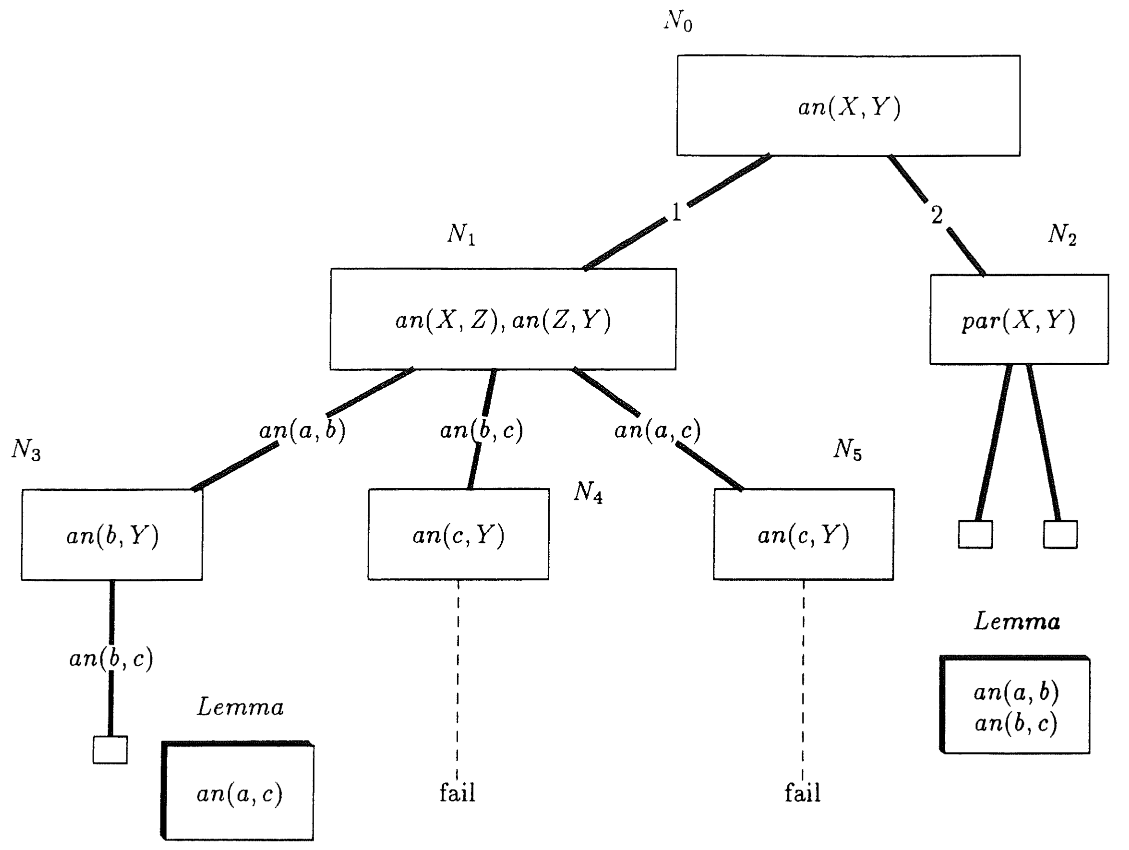


Figure 5

The Third Tree of Lemma Resolution

At this point, there is no new lemma that could be used to expand the third tree (Figure 5) and hence the process terminates. The proof of completeness of the lemma resolution can be found in [VIE87].

2. A Compilation Method Based on Lemma Resolution

In this section, we describe how the lemma resolution is applied to the compilation of recursive rules. The general concept is depicted in Figure 6. The first step is to adorn the input set of rules as in [ULL85A] by denoting which variables are free and which are bound. The next step is to transform the bodies of these adorned rules into their corresponding relational expressions; various heuristics may

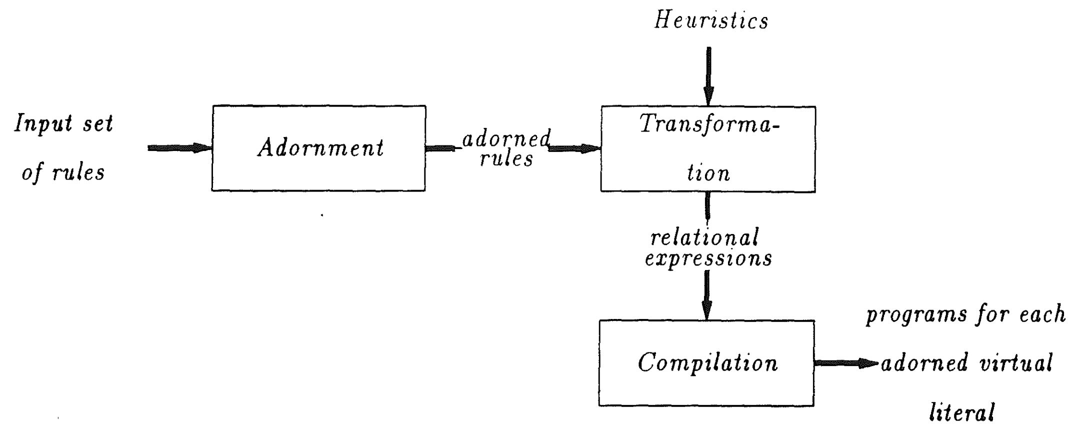


Figure 6

The Compilation Process

be applied here to improve performance during execution. The last step takes the relational expressions of these adorned rules and synthesizes sequential programs for each of the adorned goals. These sequential programs contain only relational database operators. Once they are generated, the execution can be done entirely in the database system without having to alternate between the database (EDB) and the rules (IDB) for dereferencing.

2.1. Basic Assumptions

First, we state the assumptions and give some definitions to facilitate the discussion.

(1) Rules are restricted to function-free Horn clauses.

(2) We use the terms "literal" and "predicate" in the traditional logic programming sense; predicate names are represented in lower case. For example, $p(X, Y)$ is a literal and the predicate name is p. However, in the database sense, the extension of a literal forms a table. We denote such tables by capitalizing the underlying predicate names. These tables contain columns identical to the variables of their predicate counterparts. For example, P

is the "relational" table of literal $p(X, Y)$ with two columns of attributes X and Y.

(3) *Base literals* refer to tables of the Extensional Database (EDB). These tables are stored in secondary memory. *Virtual literals* or *views* are defined in terms of base literals and/or other virtual literals. We further distinguish two types of virtual literals: *non-recursive* and *recursive.* Non-recursive virtual literals are defined using only base and/or other non-recursive virtual literals. Recursive virtual literals (or simply recursive literals) are defined using base, non-recursive virtual and recursive literals. We further assume that these three types of literals are disjoint and their types are known at compile time. We allow rules to contain non-linear recursion, in particular, mutual recursion.

(4) Without loss of generality, we assume that any literal in the body of a rule shares one or more variables with at least one other literal in that rule. Hence, side-way information passing can be accomplished by a join operation between tables that share variables. As a matter of fact, such a join is a "projected join" which means that a projection is applied to the resulting joined table. (In database terms, this is the so-called *select, project, and join*, or simply *SPJ* view.) If literals in the body of a rule do not share variables, the body may be separated into independent queries.

2.2. Lemmas for Non-Recursive Virtual Literals

Non-recursive virtual literals require that they be bound whenever they are referenced. To avoid recomputing the same results for repeated references, we can apply the same principles of retaining previously computed results in the form of lemmas as described in Section 1. Hence lemmas may be used not only to prevent infinite recursion but also to improve performance during execution by acting as a *caching* mechanism for virtual literals. Any future encounters of the same literal

(1) *ancestor(f/X,f/Y):- ancestor(f/X,f/Z), ancestor(f/Z,f/Y).*

ancestor(f/X,f/Y):- parent(f/X,f/Y).

(2) *ancestor(f/X,b/Y):- ancestor(f/X,f/Z), ancestor(f/Z,b/Y).*

ancestor(f/X,b/Y):- parent(f/X,b/Y).

(3) *ancestor(b/X,f/Y):- ancestor(b/X,f/Z), ancestor(f/Z,f/Y).*

ancestor(b/X,f/Y):- parent(b/X,f/Y).

(4) *ancestor(b/X,b/Y):- ancestor(b/X,f/Z), ancestor(f/Z,b/Y).*

ancestor(b/X,b/Y):- parent(b/X,b/Y).

The adornments f/X ancestord b/Y denote that variable X is a *free* variable while Y is a *bound* variable (i.e. a constant).

Figure 7

The Adornment for program in Figure 2

then do not require re-computation. Instead, a simple search (i.e. selection) of the lemma table yields all possible answers.

Note that retaining results of non-recursive virtual literals is *not* required to guarantee completeness but only to improve performance. Similarly, eliminating non-recursive virtual literals by program expansion and goal substitution represents a trade-off between memory requirements and speed. In this dissertation, we assume the most general case where results for both recursive and non-recursive virtual literals are retained. As a result, there are three different types of *tables* in our system: base tables, lemma tables for recursive literals (R-tables for short) and lemma tables for non-recursive virtual literals (NR-tables for short). Both R- and NR- tables are *incremental* in nature because they are only partially materialized tables which will be built up to their entirety when the processing of the literals involved is completed.

2.3. Predicate Adornments

The first step of our compilation method is to adorn predicates as in [ULL85] by denoting which variables are bound and which are constants. The adornment of variables in the head are then propagated to the body of the rules. Adornment of predicates are used later on to generate compiled programs by focusing on the bound variables. For each n-ary goal, there are 2^n distinct adornment patterns and hence that many distinct adorned rules. For example, consider the clauses of Figure 2. Since predicate *ancestor* is binary, four adornment patterns are generated as shown in Figure 7. The adornment simply denotes the types of variable names in the predicates. For example, *ancestor(f/X,b/Y)* means that variable X is free while Y is bound to an input constant. For each set of these adornments, a program will be synthesized to take advantage of the particular bindings of variables. For this example, four programs are generated for the adornments *ancestor(f/X,f/Y)*, *ancestor(f/X,b/Y)*, *ancestor(b/X,f/Y)*, and *ancestor(b/X,b/Y)*, respectively. (Note that invoking these programs is comparable to procedural calls in a general programming language.) For instance, consider the second adornment *ancestor(f/X,b/Y)*. To resolve any goal of this pattern (e.g. with X free and Y bound) is equivalent to invoking the synthesized program *ancestor(f/X,b/Y)* where parameter X is a variable and parameter Y is a constant. The program will evaluate the underlying database and return the value of X with respect to the constant Y.

Note that this step of the compilation is optional. We could disregard adornments and create only one relational algebra expression (program) for every clause. Creating a separate expression for every combination of adornments is to increase performance since it takes advantage of the current bindings at run time: the more variables are bound in the expression, the more database select operations may be applied, rather than having to perform the less efficient join operations.

2.4. Generating Relational Expressions For Rules

The second step of our compilation method is to generate the relational expressions for the body of each rule. Rules of the input set can be classified into two categories. The first type of rules are *recursive* since their bodies contain at least one recursive literal. The second type of rules are *non-recursive* since their bodies contain only base and non-recursive literals. (Note that a rule with a recursive head but non-recursive body is a non-recursive rule. These rules are sometimes called *exit* clauses [HN84] since they are the only rules through which a recursive goal is ever able to exit from a loop.)

The basic idea of our compilation method is to generate programs for each pattern of adornment of virtual literals (both recursive and non-recursive). According to the procedural semantics of Horn logic, resolving virtual literals is equivalent to resolving the bodies of their corresponding rules. We first transform these bodies of literals into relational algebra expressions. To resolve the heads of the rules is then equivalent to evaluating their corresponding relational algebra expressions, which is identical to the breadth-first search strategy in the SLD resolution. For example, consider the following two rules:

(1) p(X,Y):- e(X,Y).
(2) p(X,Y):- e(X,Z), f(Z,Y).

There are four possible combinations of adornments for p(X,Y). They are p(b/X, b/Y), p(b/X, f/Y), p(f/X, b/Y), and p(f/X, f/Y). The four relational algebra expressions corresponding to rule 1 are:

(1) $\sigma_{restriction}E$; $restriction = \{X = \_, Y = \_\}$.
(2) $\sigma_{restriction}E$; $restriction = \{X = \_\}$.
(3) $\sigma_{restriction}E$; $restriction = \{Y = \_\}$.
(4) $\sigma_{\emptyset}E$.

where underscore represents the current binding (constant) and $\sigma_{\emptyset}$ is a selection with an empty restriction, i.e., a retrieval of the entire relation.

Similarly, the four relational algebra expressions corresponding to rule 2 are:

(1) $\sigma_{restriction} E \bowtie F$; $restriction = \{X = \_, Y = \_\}$.
(2) $\sigma_{restriction} E \bowtie F$; $restriction = \{X = \_\}$.
(3) $\sigma_{restriction} E \bowtie F$; $restriction = \{Y = \_\}$.
(4) $\sigma_{\emptyset} E \bowtie F$.

Assuming that e and f are base literals, answering the query p(a,Y)?, for example, would result in evaluating the following relational algebra expression:

$$\{\sigma_{(X=a)} E\} \cup \{\sigma_{(X=a)} E \bowtie F\}$$

i.e., the union of the results obtained by evaluating the expressions corresponding to the adornment p(X/b,Y/f) of rule 1 and rule 2.

To generate relational algebra expressions from rule bodies, the following two general rules apply:

(1) Each bound variable corresponds to a possible *select* operation on the corresponding relation (e.g. $\sigma_{X=a} E$);

(2) Each pair of literals sharing one or more variables corresponds to a possible *join* operation between the corresponding two tables (e.g. $E \bowtie F$).

Given these two rules, it is always possible to transform any sequence of literals into a relational algebra expression comprising select and join (i.e. projected join) operations. However, which of the possible combinations of select and join operations should actually be used and in what order is a matter of optimization. In general, query optimization is an NP-complete problem and hence only heuristics can be applied. We shall return to this problem in section 3.1. For the purposes of this section, we assume the existence of a translator which takes the adorned literals of each clause body as input and returns the corresponding relational algebra expression (the query plan). This expression contains both base and virtual literals connected through select and join operations. Parenthesis may be used to indicate the desired order of execution.

2.5. Incremental Select and Join Operations

The main problem in evaluating relational algebra expressions is that we may encounter virtual tables (R- or NR-tables). In order to apply select or join operations, each such table must first be materialized. For NR tables, this can always be accomplished by evaluating the corresponding relational algebra expression. (Recall that for each adornment pattern of a virtual literal there is one such expression containing select and join operations on only non-recursive relations.) For recursive relations, however, this approach would fail, since, in general, the expression evaluation would not necessarily terminate.

The solution to this problem is to apply the principles of lemma resolution explained in Section 1 (to guarantee completeness) and in Section 3.1 (to improve performance). This states that, when a virtual literal G_2 is an instance of another literal G_1 that occurred previously, G_2 should not be resolved using rules. Instead, lemmas generated for G_1 thus far can be used to resolve G_2. The same principle can be applied in the context of executing the relational algebra expressions. Whenever we encounter an operand that is a virtual literal, its bindings depends on whether it is an instance of a goal encountered previously. If so, the current virtual literal is not bound by calling the corresponding sequential program. Instead we take advantage of the existence of any lemmas generated so far. For that purpose, we introduce two special kinds of select and join operations, called *incremental select* and *incremental join*, denoted by σ^I and $\bowtie^I$, respectively. These are used instead of the regular select and join whenever the associated operands are virtual relations.

Assume we need to perform a select on a virtual relation R (i.e. $\sigma^I_a R$). Recall that R is the lemma table for the predicate r. Let's assume that $\sigma^I_a R$ is equivalent to solving r(a,Y) and returning all possible bindings for the variable Y. All instances of r will be kept in R. To resolve $r(a, Y)$, we distinguish two cases. First, if $r(a, Y)$ is an instance of some previously occurred goal r_0, then this goal is to be resolved with the lemmas of r_0 (which are contained in R). To be able to determine that, we need to maintain an execution history list H_r, recording all occurrences of the goal r. Whenever r is to be resolved, we consult this list. If r is an instance of some previously resolved goal on this list, there already exists a partial materialization of the literal r – the lemma table R. In this case, the resulting table of $r(a,Y)$ is bound by performing a select operation $\sigma_a R$, where the restriction a is given by the bound variables of r. We will refer to this operation as *lemma select.* Otherwise, R is bound by evaluating the appropriate relational algebra expression for the current adornment pattern of R, i.e. by invoking the corresponding synthesized program. This will be referred to as a *call select.*

The following procedure summarizes the implementation of the incremental select $\sigma_a^I R$:

```
if r of $\sigma_a^I R$ is an instance of a goal on $H_r$
    then $\sigma_a R$; (* lemma select *)
    else call $[R_a]$; (* call select, where $[R_a]$ is a procedure call *)
```

The incremental join operator $A \bowtie^I B$ is used if at least one of the operands is a virtual relation. Without loss of generality, we assume that A has already been bound through a previous select, join or call operation, or by being a base relation. Similar to the incremental select, B is bound using either existing lemmas or by evaluating the appropriate expression (represented by a synthesized program), depending on whether B is an instance of a previous goal or a new occurrence. Unfortunately, to determine that is not as simple as with the incremental select because the bindings of B depend on those of A due to shared variables.

To illustrate the incremental join $A \bowtie^I B$, let's assume that the underlying predicates are a(e,Z) and b(Z,Y). A can be bound by the select operation $\sigma_e A$, yielding a set of bindings for Z, which are passed to B. This divides B conceptually into two parts — the first, say B_1, contains all tuples that are instances of some previous occurrences of b (and are contained in table B), while the second part, B_2, contains tuples that represent new goals. We propose that an incremental join be implemented by a projected group-by operation (i.e. a projection applied after the group-by operation) and an incremental select. The general form for the operation $(A \bowtie^I B)$ is as follows (assuming that A has already been bound).

(1) Group A By the join column(s) $Z'i$;
$Z'i$ are projected out from the resulting table;

(2) for each unique $Z'i$ do $\sigma^I_{Z_i} B$.

The materialized relation A is *grouped by* the join column of A (i.e. the variable Z). The join columns (Z) of the resulting tables are projected out. For each distinct constant of Z, we implement an incremental select on B. The join of this kind builds up the resulting table in an incremental manner, hence the term "incremental" join.

To summarize, results of previously solved recursive literals are retained as lemmas. When select and join are applied to a recursive or non-recursive virtual literal, we need to determine if the recursive literal is an instance of a previous goal or not, and act accordingly (either use lemmas or invoke another procedure). This is the major step that guarantees the termination of recursion (for the case of recursive literals) and that improves the performance by avoiding repetitive computations (for the case of non-recursive literals).

An Example

Consider the first adornment pattern in Figure 7. The rule with body *parent(f/X, f/Y)* can be transformed into the relational operation $\sigma_\emptyset Parent$ to retrieve the entire table of $PARENT$. The other clause body for this adornment is *ancestor(f/X, f/Z), ancestor(f/Z, f/Y)*, which comprises two recursive literals both referring to the same R-table $ANCESTOR$. The conjunction of these two literals can be transformed to the relational algebra expression: $(\sigma^I_\emptyset ANCESTOR \bowtie^I ANCESTOR)$, which means that an incremental select on R-table $ANCESTOR$

is performed first, followed by an incremental join. Since the select does not have any selection constants, the entire current *ANCESTOR* table is selected. This is then incrementally joined with itself.

2.6. Compilation of Relational Expressions

The third step of the compilation takes the set of relational algebra expressions generated in the previous step as input and compiles a sequential program for it. The general structure of the program is shown in Figure 8. The following data structures are required: a lemma table Lg for each virtual predicate g, an *E_list* whose functions will be elaborated in detail later, and the history list H_g of execution of every recursive goal g. Note that for every virtual goal g (either non-recursive or recursive), there is a lemma table L_g. However, a history list H_g is needed only for recursive goal.

In general, the input set of relational algebra expressions I can be partitioned into two sets, N and Q, of clauses: the set N does not contain any virtual tables (e.g. R or NR-tables) in the clause bodies while the set Q does. The set N is sometimes called the set of *exit* clauses [HN84]. They are required in any recursion to generate the first-level results. Hence, they generate the initial set of lemmas for the recursion. This is reflected by the loop starting on line 1 of Figure 8.

As illustrated in section 1, new lemmas can be generated by evaluating the recursion through an expandable node. Therefore, an expandable node is both a producer and a consumer of lemmas. In other words, expandable nodes need to be re-evaluated against lemmas generated from the previous stages. The list structure *E_list* is used to hold those paths that contain expandable nodes. Recall that an expandable node begins with a recursive literal. Since the expressions in the set of Q represent all paths that contain some R-tables, they are placed on the E_list as shown in line 2. The program terminates when none of the elements on the E_list generates any new lemmas. The main processing is done by the procedure EVAL

```
Notations: $I$= input set of relational expressions of Adorned $g$;
           $N$= relational expressions that do not contain recursive tables;
           $Q$= relational expressions that contain recursive tables;
           $I = N \cup Q$;
Data Structures:
           $L_g$: Lemma table for for each virtual predicate $g$;
           E_list: list to hold expandable nodes;
           $H_g$: history list of recursive goal of predicate $g$;

1.    for each $n \in N$ do
          $L_g$:= $L_g \cup$ EVAL(n);
2.    for each $q \in Q$ do
          put $q$ in E_list;
3.    repeat
          for each $q$ in E_list do
              $L_g$:= $L_g \cup$ EVAL(q);
      until no change in $L_g$
      return ($L_g$);

Procedure EVAL(E)
4.  while $E$ is not completely parsed do
5.      begin
6.        token:= parser($E$) (*token in the form of $\sigma X$ or $A \bowtie B$*);
7.        if no incremental operator in token
8.           then token_table:= dbperform(token)
9.           else  (*token is either $\sigma^I R$ or $A \bowtie^I R$ *)
10.                case token of:
11.                     $\sigma^I$:        if $R$ is non-recursive virtual table
12.                                      then token_table:= $\sigma L_R$;
13.                                           if token_table = $\emptyset$ then call [$R$];
14.                                      else (* $R$ is recursive*)
15.                                           convert $\sigma^I R$ into predicate form $g$;
16.                                           if $g$ is an instance of some element in $H_g$
17.                                              then token_table := $\sigma L_R$
18.                                              else  put $g$ in $H_g$;
19.                                                    call [$R$];
20.                     $A \bowtie^I R$:   A_table:= EVAL(A);
21.                                  Group A_table By join column(s) into $Z$'s;
22.                                  for each $Z_i$ in $Z$ do
23.                                      tmp:= EVAL($\sigma^I_{Z_i} R$);
24.                                      token_table:= token_table $\cup$ assemble(A_table, tmp);
25.               end case;
26.       end if;
27.       reduce(token_table, E);
28.     end while;
29. return(token_table);
```

Figure 8

General Framework for the Compiler

which invokes four other procedures, *parser*, *dbperform*, *assemble*, and *reduce*. We will explain these procedures along with the explanation of the algorithm of EVAL. Given a relational expression, the procedure EVAL will evaluate it in an inside-out manner, according to the order denoted by parentheses. For this purpose the *parser* procedure always returns the next appropriate relational operation, together with the corresponding operands (line 6). For example, if the relational expression is $\sigma(X \bowtie \sigma Y)$, the first token returned by the *parser* would be σY, because it is the inner-most operation. Note that any token extracted by the *parser* has the form of either a selection or a join.

If the token does not contain any incremental operator, the procedure *dbperform* is invoked to evaluate the operation directly. This procedure *dbperform* simply performs the corresponding select or join operation on the database and returns the resulting table as the *token_table* (line 8). If the token contains an incremental select operator, $\sigma^I R$, the procedure tests if R is a recursive virtual table. If that is not the case, a select on the lemma table L_R is carried out. If the selection results in an empty table, it implies that R is not an instance of a previous goal (we have not encountered R with such a binding before). In that case, a *call select* is carried out with the constant bindings of the incremental selection. These choices are reflected in lines 11 to 13.

Note that the incremental select can be implemented in this way only for the case of non-recursive virtual tables because these tables are *not* expandable as discussed in section 3.2. Once a non-recursive virtual literal is encountered, all derivable tuples are obtained at the same time and they will not be the producers of further results.

If R is recursive, it is necessary to implement the instance test explicitly as shown on lines 15 and 16. First , we convert the relational algebra expression $\sigma^I R$ back into predicate form g. We then test if g is an instance of some element in

H_g.[3] If this is the case, then a lemma resolution is performed simply by carrying out a selection on L_R.

If the token is an incremental join between two tables A and R, the incremental join is implemented by the group-by and incremental select as described in section 3.5, which is shown in lines 21 to 24. Since the group-by operation and the incremental select are done separately, a procedure called *assemble* is invoked to put the results of the A-table and the *tmp* table which is result of the incremental select, into a table that represents the result of the original incremental join of A and R. Finally, a procedure *reduce* is invoked to reduce the resulting table (i.e. token_table) with the original relational expression E. For example, given the expression $\sigma(X \bowtie \sigma Y)$, the procedure *reduce* takes the results of σY and renames it to *result_1*. The expression E thus becomes $\sigma(X \bowtie result\_1)$. By parsing E again, the token $X \bowtie result\_1$ is extracted and processed. Parsing of an expression is done when the original expression is reduced to one single table (possibly empty).

3. Performance Considerations

The ultimate purpose of this approach is to integrate an inference mechanism into a relational database system such that large quantities of facts can be deduced efficiently. The compilation method based on the lemma resolution described in previous sections is aimed in this direction. As noted in section 2.4, while forming relational algebra expressions from rules, optimization can only be done through heuristics. There are two basic rules in forming a relational algebra expression from the conjunctions of literals: bound variables of a literal may be potentially translated into a select operation of the underlying relational table while shared variables between literals indicate candidate join operations between tables. As such, classical query optimization techniques [ULL89] can be applied here. In this

[3] In practice, the conversion step is not really necessary. If the history list is maintained as a table, the instance test then can be implemented as a select on that table.

section, we address mainly heuristics for query plan optimization related to lemma resolution.

3.1. Heuristics for Query Optimization

There are two major inefficiencies in our current compilation method. The first type occurs within a single recursive rule. We call it *inter-rule* redundancy. This inefficiency derives from the fact that non-recursive literals that are part of a recursive rule are re-evaluated each time the rule is invoked. This is because the rule is kept in the *E_list* and is evaluated against any newly derived answers. Consider, for example, the following conjunction of literals: $p(a, B), d(B, Z), q(Z, Y)$ where p and d are base literals while q is recursive. This corresponds to evaluating the relational expression $\sigma_a P \bowtie D \bowtie^I Q$. From section 2, we know that, by the time the literals p and d are resolved, the node containing $q(Z, Y)$ becomes an expandable node and will be evaluated against the lemma table Q for further expansion. Lemmas are then fed back to the expandable nodes until no further lemmas are generated. Note that each time the expression $\sigma_a P \bowtie D \bowtie^I Q$ is evaluated, the operation $\sigma_a P \bowtie D$ yields the same answers. This, obviously, is unnecessary. Instead, any *leading* conjunction of base or non-recursive literals can be retained as "wavefront" or "frontier" which is basically the collection of immediate results of these non-recursive literals.

The second inefficiency occurs between different rules with the same head; it is called *inter-rule* redundancy. Consider the following two rules:

(1) $q(X,Y) :- p(X,A), d(A,B), e(B,C), q(C,Y).$
(2) $q(X,Y) :- p(X,A), d(A,B), f(B,C), q(C,Y).$

Our compilation method will evaluate these two rules independently. Since the leading predicates *p(X,A),d(A,B)* appear in both rules, if p and d are base relations, it means that the tables P and D are retrieved twice from the secondary storage. In order to minimize disk access, these two tables can be retrieved only once and kept in primary memory. If the predicate $q(X,Y)$ is defined over a set of rules which share some predicates, it is possible to "pre-compile" these rules by finding common structures sharing these predicates. The general principle is simple: for each adornment pattern, the bodies of the goal form a graph whose vertices are the distinct adorned variables and edges are the predicate names. Two graphs are then connected by identifying the intersection between them. By joining all the graphs together, a supergraph is formed. With this resulting graph, heuristics are applied to decide the execution order of the graph. The shared subgraphs are the common substructures of the original rules. Therefore, their bindings are retained in order to minimize disk access. The details of this pre-compilation should be investigated in details in the future.

3.2. Sliding Window of Lemma Tables

Another possible inefficiency lies in the manner in which the expandable nodes are evaluated against their lemma tables. For example, consider the recursive relational expression $\sigma^I Q$ where Q is a recursive lemma table. $\sigma^I Q$ will be evaluated repeatedly as long as there are new tuples generated in Q. Obviously, it is not necessary to evaluate the entire Q. Only the newly added tuples of Q are of interest. With this observation, we can improve the performance by using a pointer for each element in the E_list. This points to the first lemma in the lemma table that has not yet been used for the current E_list element. When the lemma is applied, the pointer is advanced accordingly. Hence, each pointer denotes the beginning of the subtable of Q that is still relevant to that particular E_list element.

4. Chapter Summary

In this chapter, we propose a compilation method for recursive queries based on lemma resolution which is both complete and terminates in all cases. Our method combines both top-down (the binding information of the query determines the compiled program) and bottom-up (the lemmas) methods. It is different from most bottom-up methods such as [ULL85, BMSU86, SZ86]. The features of our compilation method are listed as follows:

1. Our method is based on the *lemma resolution* which is proved to be correct and complete for function-free Horn logic.
2. When generating the relational algebra expressions, many conventional database optimization techniques can be applied.
3. Our incremental join can handle complex recursions.
4. Our way of implementing the incremental join resembles the hash join where the original table is decomposed into a set of small tables. Each of these small tables has the value from the join column. By doing the join with these small tables instead of larger tables, the performance can be improved.

CHAPTER 3
View Materialization and Base Updates

In Chapter 2, we have shown how lemma resolution solves the termination problems of recursion and how deductive rules can be compiled into iterative programs containing relational algebra. The ultimate goal of compiling deductive rules is to improve the performance of processing view queries. In a conventional relational database system, a *materialized* view is a table of results after retrieving a view from the database. Usually, views are not materialized until they are needed and are discarded afterward. It is observed that query processing can be improved by keeping *frequently* needed views materialized. This is particularly true if these views are recursive since they will be more costly to recompute.

Retaining results to improve further computations is not unique in database research. For example, it has been shown in programming languages research that retaining results can improve execution time of recursive programs. The notable studies are the *tabulation method* [BIR80], the *memo functions method* [MIC68] and other various approaches such as those shown in [CHAN73] and [COH83]. However, the problem becomes unique in database research since, if views are to be kept permanently as *view tables*, then updates of base relations or other views will affect the integrity of these view tables.

There are two distinct types of updates with different consequences: (1) updates of base tables, and (2) updates of view tables directly. Updates on base tables (this will be referred to as *base updates*) have two effects: first, the base tables get updated, which is identical to ordinary updates in relational tables. For instance, each update request is subject to the validation of integrity constraints that have been specified for that particular table. Second, updates on base tables

(1) *ancestor(X, Y) :- parent(X, Y).*

(2) *ancestor(X, Y) :- parent(X,A), ancestor(A, Y).*

Figure 9

Single Recursion Ancestor View

may affect certain view tables if these base tables are supportive relations of those view tables. On the other hand, if views are kept physically as tables, a user should be able to update them just like base tables. Updating a view may affect its supportive relations which can be base relations or other views. Such an update is termed *view update*. To illustrate what the base and view updates are, consider the *ancestor* view in Figure 9. Note that this example is similar to the example in Figure 2 except that the *ancestor* is defined with a single recursion instead of a double recursion.

A *base update* refers to the case when the base table PARENT is updated. From the second rule of the view definition, it is obvious that updates on PARENT[4] may affect the view table ANCESTOR. On the other hand, assuming that all views are materialized and kept as view tables, a user may directly update the ANCESTOR view table. Updates of the view table ANCESTOR are called *view updates* and their effects should be propagated into the body of its definition (the right-hand side of the rule). View updates will be covered in Chapter 5.

In this chapter, we focus on base updates. In particular, we present a unifying approach, that is based on the lemma resolution of Chapter 2, to tie both the query and the update process of recursive queries together. The organization of this chapter is as follows: section 1 will describe what view materialization is and will also discuss related studies. In section 2, we describe our unique incremental update

[4] We use the same notation as in Chapter 2 that predicate name is expressed in lower case; the same name in upper case denotes the relational table.

and query processing strategies. With our incremental update and query processing techniques, in section 3, we discuss how the base updates can be implemented.

1. View Materialization and Related Work

Generating materialized views is straightforward if they are not recursively defined. For example, in the *query modification method* [STON75], a query on a view is transformed into a sequence of subqueries on the base relations. Its limitation is, of course, that it only allows non-recursive views. (In general, recursion is not supported in a relational system because the relational algebra does not support recursion.) If no view tables are to be maintained, updates on base relations do not require any view update since the view will be re-computed on the fly when needed. However, if views are materialized and updates are allowed, integrity of the materialized views has to be maintained.

In the relational database research, there are basically two different approaches to base updates: *immediate view maintenance* and *deferred view maintenance* [HAN87]. In a naive immediate view maintenance approach, a view is materialized in its entirety and every update to the base relations triggers updates on the view tables immediately, mostly by re-evaluating the whole view table. If every base update triggers the re-evaluation of the view tables, the cost to maintain view tables can be tremendous. Several refinements have been proposed. For example, in [BC79], a method is proposed to analyze the base updates to determine if they affect any view tables. If they do, or if the analysis fails to detect whether the view may be affected, the view will be completely recomputed. In [BLT86], Blakeley et al propose a two-step mechanism for maintaining materialized views. The first step checks if the base updates are relevant to the view. The second step gives a *differential* view update algorithm for the relevant updates. However, these approaches work only for so-called SPJ (Select-Project-Join) views; they do not work for other combinations of relational operators, let alone recursive view definitions.

For the deferred view maintenance approach, updates on the base tables are *not* propagated to the view table immediately. Instead, the affected views are updated just before data is retrieved from them. Hanson [HAN87] describes a similar approach, where the sets of tuples inserted and deleted are saved for a certain period of time. At the end of each period a differential update algorithm is applied to the whole group. Again, this approach works only for the SPJ views.

Another interesting but more restricted method is the ***snapshot*** approach [LHM86]. A database snapshot is a read-only table whose contents are derived from other tables in the database. Hence, it is a special case of a view which does not allow users to perform view updates. The snapshot contents can be periodically *refreshed* to reflect the current state of the database. When the snapshots are restricted to a simple restriction and projection (no join), a differential refresh technique was proposed. Their techniques can reduce the message and update costs of the snapshot refresh operation.

An important issue to improve the performance of these two view materialization algorithms is the design of an efficient screening algorithm. A screening algorithm is used to test each inserted or deleted tuple from the base tables. If an update request fails the screening test, its effect may affect the integrity of its associated views, so the update has to be propagated to the view. Otherwise, it does not need to refresh the view. There have been several studies on designing efficient screening algorithm [BLT86, BC79, SSH87]. However, as in the case of designing view maintenance algorithms, these screening tests are restricted to SPJ views. Since SPJ views are non-recursive, these algorithms can *statically* analyze the effects of the updates and determine if the update has immediate effects or not. Unfortunately, these screening tests are not applicable if views are allowed to be recursive. In section 3, a simple screening test applicable to recursive views is described in detail.

2. Incremental Materialization and Query Process

One commonality of the above two view maintenance approaches (i.e. immediate, or deferred) is that, in both cases, a view table is generated and maintained in its entirety. This can be very costly, especially if the view is recursive. It has been observed [KDC87] that, user queries exhibit a certain kind of *locality* in most database applications. This locality refers to the repetition of user queries. Typically, users of a given database tend to ask repeated queries. The implications of this observation are twofold. First, it implies that there is a definite need to retain the results of frequently asked view queries. Second, it implies that not all data of a particular view are needed. Thus, it would be a waste to materialize the entire view table at once if only a small fraction of it is needed.

To improve the performance, we propose the following:

(1) the view table is not completely materialized; instead, materialization is done incrementally. That is, the bindings are substantiated progressively as user queries are evaluated. For example, consider the view *ancestor* again. It would be wasteful to materialize the complete ANCESTOR table, if, for instance, users are mostly interested in finding out the ancestor relationship of no more than three generations of a few families.

(2) It would be ideal if we could perform the deferred view maintenance by saving the sets of tuples inserted and deleted for a period of time, and then applying a differential update algorithm to the whole group. Unfortunately, the differential algorithms in [LHM86] are restricted to views of select, project and join (SPJ). It is still unknown how to differentiate updates in a recursive view. Therefore, the immediate view maintenance is adopted in

our approach, in which, if the base update affects the partially materialized view table, then the view is updated immediately. Otherwise, only base tables are updated.

In this chapter, we design, using the lemma resolution concepts, an incremental update process that will implement the above two principles. The idea is to maintain a set of smaller partially materialized view tables and to extend the query process to accommodate the incremental update process.

In order to understand how base updates work, it is necessary to first describe how query processing is done in our approach.

As mentioned above, our approach does not require the complete view tables to be derived. For example, let $q(X, Y, Z)$ be a view. Initially, we do not materialize the view table Q at all. At a certain time, a user may issue the query *q(a, Y,Z)*, which is processed as discussed in Chapter 2. If the query is satisfiable, the bindings of Y and Z are returned. Only then are the tuples of these bindings retained and saved as the view table Q. In addition, all lemmas generated while processing the query are also retained. Note that the tuples retained are not identical to the complete materialized view table Q; it is only partially materialized. For instance, only view tuples of q that contain the constant a as the first attribute are in Q.

By retaining the results, the time to process another query can be improved by first searching the view table using a simple relational operation (i.e., selection). Only if the query is ***not*** found in the view table, the query is evaluated in the usual manner: to call the relevant compiled programs according to its query adornment. As users query the database more often and with different constants and adornment patterns, the view table is built up *incrementally*. However, because our view tables are only partially materialized, this query strategy is not always complete. To illustrate the problem, let us assume that query *q(m, Y,n)* is asked and is satisfied. The view table Q now contains tuples that have the constants m and

n as their first and last attributes, respectively. Suppose that the next query is *q(m, Y,Z)*. According to the above strategy, we would simply issue a relational operation $\sigma_{1=m}Q$, which would select tuples of Q (the partially materialized view table containing tuples with constants m and n as the first and last attributes) where the first attribute equals constant m. This particular selection will return all tuples of Q. However, if the query processing stops at this point, we have only returned a partial answer to the query. The reason is that *q(m, Y,n)* is an instance of *q(m, Y,Z)* (instances are discussed in Chapter 2, section 1). Therefore, the set of tuples returned by processing query *q(m, Y,n)* will always be a subset of those returned by the query *q(m, Y,Z)*.

To overcome the above deficiency, we propose the following modification:

(1) conceptually, there will be a separate Q table for each query adornment patterns. For example, after the query of *q(m, Y,n)* is processed, we will create a view table of that particular adornment pattern $Q(b/X, f/Y, b/Z)$ (the view table of Q in which the first and third columns are bound and the second column is free.)

(2) the query will first be looked up in its relevant view table according to the query's adornment. If the lookup fails, then the compiled program of the specific adornment is called to process the query (as described in Chapter 2.)

2.1. Table Subsumption

There exists some redundancy among the many view tables created by step (1) above. Even though the adornment patterns are all unique, some of them are more general than others. By applying the same test to check if a predicate is an instance of another predicate, we can determine if a view table is a subset of the others. Hence, to reduce the number of partially materialized view tables we employ a *table subsumption* strategy. For instance, if table Q_i is more general than an existing table Q_j (e.g. $Q_i(b/X, f/Y, f/Z)$ vs $Q_j(b/X, f/Y, b/Z)$), then Q_i subsumes Q_j.

```
(1) if $Q_j$ is more general than $Q_i$ then
(2)         if the constants of $Q_j$ are in $Q_i$
(3)                 then $Q_j$ subsumes $Q_i$
(4)                 else $Q_j = Q_j \cup Q_i$
```

Figure 10

Table Subsumption Algorithm

Unfortunately, subsuming tables by simply checking instances of adornments is still incomplete. It is incomplete because the adornment pattern does not reveal the actual constants for the bound variables. For the case of $Q_i(b/X, f/Y, f/Z)$ and $Q_j(b/X, f/Y, b/Z)$, Q_j is an instance of Q_i. However, if Q_j contains constants for X that are not in Q_i, then Q_i cannot subsume Q_j. For example, consider the following scenario: a query *q(m,n,p)* is posed and satisfied. The success of answering the query then generates a partially materialized view table $Q(b/X, b/Y, b/Z)$. Next, another query *q(t, s, Z)* is asked and satisfied and hence a view table $Q(b/X, b/Y, f/Z)$ is created. Based on the adornment patterns, $Q(b/X, b/Y, f/Z)$ is more general than $Q(b/X, b/Y, b/Z)$. Unfortunately, $Q(b/X, b/Y, f/Z)$ cannot subsume $Q(b/X, b/Y, b/Z)$ because the tuples in $Q(b/X, b/Y, f/Z)$ are not a complete set for the adornment $Q(b/X, b/Y, f/Z)$, but only a fraction of it; the resulting table contains only the tuples begin with constants t and s. Therefore, in order to guarantee the completeness of table subsumption, it is necessary to check if the constants of the new view table have been seen before.

In general, assume that Q_i and Q_j are two view tables of the same predicate q, and Q_j is the most recently materialized table. The table subsumption procedure is shown in Figure 10. First, the adornment pattern of Q_j is compared to that of Q_i. If Q_j is more general, then it may be possible that Q_j is a superset of Q_i. To determine whether this is the case, it is necessary to check if all constants in the adornment of Q_j appear in the same positions of Q_i. For instance, the adornment of Q_j is $Q_j(b/a, b/c, f/W, f/Y)$ and the adornment of Q_i is $Q_i(b/a, b/c, f/W, b/d)$ where b/a, b/c denote the fact that the first and second terms in Q_i and Q_j are bound to constants a and c, respectively. In this case, Q_j is more general than Q_i and all constant bindings in Q_j also occurs in the corresponding positions in Q_i. If this is true, then Q_j is indeed a superset of Q_i and hence subsumes Q_i. Otherwise, it implies that some tuples in Q_j and Q_i are disjoint (there is no set subset relationship between them). In this case, the correct result is to union both tables (line 3). Note that step (2) of the procedure can be further simplified by checking the indexed tree of Q_i without doing any actual disk I/O since keys of these indexed trees have to be the composite of the constants in the adornment pattern.

Note further that, if a *for_all* type query (i.e. *q(f/X, f/Y, f/Z)* is posed by the user, the results may supersede and subsume all other tables. Partial materialization would then be reduced to the case of the complete materialized view approach mentioned in section 1, which we try to avoid. One solution to this problem is to distribute the results of the *for_all* query to each possible adornment patterns (if an adornment does not currently exist, then it would be created.) Future *for_all* queries would have to retrieve tuples from all disjoint adornment view tables with the union operator. To illustrate this situation, consider the *ancestor* view again. The view ANCESTOR has four distinct adornment patters: *ancestor(f/X, f/Y)*, *ancestor(f/X, b/Y)*, *ancestor(b/X, f/Y)* and *ancestor(b/X, b/Y)*. However, either *ancestor(f/X, b/Y)* or *ancestor(b/X, f/Y)* would have subsumed *ancestor(b/X, b/Y)*. It results in two disjoint subview tables (i.e. *ancestor(f/X, b/Y)* and *ancestor(b/X, f/Y)*. Suppose that the *for_all* query (i.e. *ancestor(f/X, f/Y)* is successfully processed. Instead of retaining the results for *ancestor(f,f)*, the results are distributed to *ancestor(f/X, b/Y)* and *ancestor(b/X, f/Y)* by: first, group

```
(1) if there exist an index file $I_q$ with identical or more general adornment of $q$
(2) then select $I_q$ with constants in $q$
(3)         if empty then expand $q$
(4)         else return bindings for $q$
(5) else expand $q$
(6) if $q$ succeeds (tuples returned in $Q$ table and a new index file $I_q$ is generated according to
    $q$'s adornment)
(7) then apply the table subsumption procedure to $I_q$ with the other index files
(8) else $q$ is not satisfiable
```

Figure 11

Query Process With Indexed Files

ANCESTOR(f/X, f/Y) by the first attribute and replace the ANCESTOR(b/X, f/Y); second, group ANCESTOR(f/X, f/Y) by the second attribute and replace the ANCESTOR(f/X, b/Y). Note that the *group-by* is a common operation in relational databases. After the distribution, the ANCESTOR(f/X, f/Y) can be discarded. Any future *for_all* query is simply a union of ANCESTOR(b/X, f/Y) and ANCESTOR(f/X, b/Y), provided these two tables are maintained properly.

2.2. Index Files and the Modified Query Processing Strategy

If we assume that all partially materialized view tables are maintained as physically separate tables, then many tuples could be replicated across different tables. For instance, the same tuple (m, n, p) could be retained in $Q(b/X, f/Y, b/Z)$ and $Q(b/X, b/Y, f/Z)$ for the queries *q(m, Y,p)* and *q(m,n,Z)*, respectively.

In practice, it is not necessary to keep all these tables separately. Instead, we can maintain one Q table (with no duplicates) but with multiple index files, each representing a unique adornment pattern. The query process works with the index files instead of the real data file. To process query q with index files, the algorithm is modified and shown in Figure 11. The index files are implemented as B+ trees which are balanced multiway trees with leaves located at the same level containing the keys and the addresses of location of the tuple on the secondary storage. Each node of the tree contains some number of keys. The keys are therefore the constants in the adornment pattern. The algorithm in Figure 11 is straightforward. For each query q, we first check if there already exists an index file for the adornment of q (i.e. I_q). If it does not, this implies that it is the first time the adornment of q is encountered. Therefore, we expand q (line 5) by calling the compiled program as discussed in Chapter 2. If the index file I_q already exists, a selection with constants from q is applied to I_q (line 2). If the selection is successful, the results of matched tuples are returned (line 4). If the selection returns an empty table, it implies that the constants of q have not been processed. The compiled program for q is called to expand q (line 4). If the compiled program of q terminates without bindings, it means that query q is unsatisfiable (line 8). If the query q is successful, the bindings will be returned. At the same time, it is necessary to see if the newly derived table can subsume or can be subsumed by some other tables. Even though there exists only one table Q (with many index files), the table subsumption procedure in Figure 10 is still applicable, with slight modifications as shown below:

```
     Assume that $I_q$ is the index file for the adornment pattern of query $q$, and
     $I_j$ is each of the remaining current index files. For each $I_j$,
(1)  if adornment of $I_q$ is more general than that of $I_j$ then
(2)          if the constants in the adornment of $I_j$ are in the adornment of $I_q$
(3)                  then $I_q$ subsumes $I_j$
(4)                  else merge $I_q$ and $I_j$
```

The procedure is similar to the table subsumption discussed above except that there exists only one table Q with many index files. The index file I_q is the newly generated index file for the satisfied query q. If the query adornment of q

is more general than another index file (e.g. I_j), it is necessary to check if the constants in the adornment of I_j are all in the adornment of I_q (line 3). If this is true, index file I_q subsumes I_j. Otherwise, it means that I_j and I_q do not have a set-subset relation. Therefore, these two index files are merged (line 4). Note that merging the index files is identical to the union operation applied to the partial view tables as in line 4 of Figure 10.

3. Base Updates

In this section, we discuss how base updates are accomplished with our incremental update and query processing techniques described in section 2.

Without loss of generality, we make the following assumptions regarding base updates in our studies:

(1) update requests refer to insertions and deletions only. Modification can be implemented by a sequence of selections, deletions and insertions.

(2) update requests are atomic; null values and free variables are not allowed. Hence, an update request, such as *a 10% raise in salary of all employees* will have to translate into *a 10% raise in salary of* x_i where x_i is the i^{th} domain value of employee names;

(3) integrity constraints (ICs) for base and view tables are not considered here. We assume that there is a separate process to check for update integrity.

Since not every base update request would update the views, screening the update requests for possible effects on views becomes a vital step toward improving performance. Our approach is similar to the immediate view maintenance approach mentioned in section 1. The main difference is that our view tables are not always completely materialized, which requires a different screening test. Consider a deletion of a tuple in the PARENT table in Figure 9. If the deletion affects any *partially materialized* view tables of ANCESTOR, those view tables will be updated; otherwise, the update effects will not propagate. This is possible in our approach because view tables are partially materialized and also queries are incrementally processed. For example, if the deleted base tuples do not affect any ANCESTOR tables, the update is terminated without propagating to the ANCESTOR. Since there are no views affected, it means that the views derivable from those base tuples have never be asked. Now that the base tuples are deleted, future queries will never derive those view tuples.

Similarly, if a tuple is inserted to PARENT and if it affects some of the ANCESTOR tables, the effects have to be propagated to these affected tables. If the update does *not* affect any ANCESTOR tables, the update effects do not need to propagate to ANCESTOR immediately. Note that this is very unique in our approach. When the screening test decides that the base update does not affect any *existing* view tables, it does not mean that these base updates cannot derive other view tables. It implies, in our approach, that either (1) these inserted base tuples would not derive new view tuples; or (2) they might be able to derive new view tuples but they have not yet been queried. Since our query processing strategy is incremental, those view tuples that are supposed to be generated by the newly inserted base tuples can *always* be derived when a query is posed.

3.1. Screening Test

As noted above, the incremental base update is affected significantly by the efficiency of the screening algorithm it employs. For example, consider the following view definition:

$$v(X,Y,Z) :- p_1(X,A), p_2(A,B), v(B,Y,Z).$$

The view table V with view variables X, Y, and Z is defined as a tail recursion using base tables P_1 and P_2. Suppose that the view has been incrementally materialized. That is, users have queried (directly or indirectly) this particular view successfully and the resulting bindings are retained. Updates on the P_1 or P_2 tables have to validate the view table. Unfortunately, if updates on the base tables were to trigger updates on V, the total cost of maintaining the view table would definitely outweigh the benefit of speeding up future queries. Thus a screening test is necessary in order to cut down the frequency of updating the view table. Most of the screening tests studied in the past fall into one category: check if the update qualifier affects the selection qualifier of the view definition [BLT86]. In this approach, a PSJ view can be defined as:

$$W = \pi_X(\sigma_{C(Y)}(R_1 \times R_2 \times ... \times R_p))$$

where *C(Y)* is a boolean expression and X and Y are sets of variables denoting some or all of the attributes of relations R_i's. For example, assume that there are two relational tables *R(A,B)* and *S(L,M)* and a view is defined as:

$$W(A,M) = \pi_{A,M}(\sigma_{(B=L)\wedge(A>10)}R \times S)$$

where *C(Y)* can be expressed as $C(A,B,L) \leftarrow (B = L) \wedge (A > 10)$. Then, if an update request is to insert $(5,12)$ to table R, the constants 5 and 12 substitute the variables A and B in $C(Y)$, which results in the boolean expression $C(5,12,L) \leftarrow (12 = L) \wedge (5 > 10)$. This expression is obviously unsatisfiable and hence it can be concluded that the update request would not affect the materialized view V.

This approach works only if the view is defined as a non-recursive one, since it is always possible to judge if *C(Y)* is satisfiable or not. Unfortunately, it is not applicable in our situation where a view may be defined recursively. For example, the above tail recursive view V can be rewritten as:

$$v(X,Y,Z) : -- p_1(X,L), p_2(M,N), v(S,Y,Z), L = M, N = S$$

which can further be expressed as:

```
(1):  if none of the update attributes are view variables
(2):          then update the view table;
(3):  if some of the update attributes are also view variables
              and the actual binding constants of the update attributes
              also exist in the current view table
(4):          then update the view table;
(5):          else the view is unaffected;
```

Figure 12

A Screening Test Algorithm

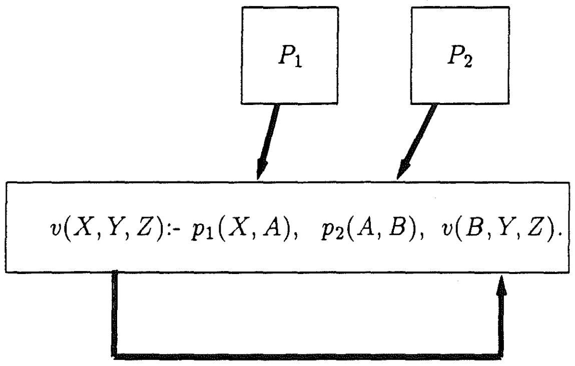


Figure 13

An Example of Rule-Goal Graph

$$V(X,Y,Z) = \pi_{X,Y,Z}(\sigma_{(L=M)\wedge(N=S)}P_1 \times P_2 \times V)$$

The recursion will have a different *C(Y)* at each recursive level (due to the different bindings of V.) The satisfiability of *C(Y)* is difficult to obtain without actually executing the view.

One viable screening test is to check the attribute of the update tuple against the view variables. The screening test algorithm is shown in Figure 12. Assume that the possible affected views are identified. This can be done easily by using the rule-goal graphs as suggested in [ULL85]. A rule-goal graph is a structure to show the inter relationship of base tables and view tables. For example, consider the rule-goal graph in Figure 13, in which, P_1 and P_2 are two base tables and V is a view table. The rule-goal graph simply indicates that the view table V is supported by these two base tables P_1 and P_2. We further define the term *update attributes* to be those attributes that are being updated within the view definition. For instance, if the table P_1 is being updated, the update attributes in respect to V are X and A. Similarly, if P_2 is being updated, the update attributes are A and B. We also define the term ***view variables*** to be the input variables of the view. For example, X, Y and Z are view variable in respect to the view V.

For each affected view candidate, the algorithm in Figure 12 determines if it needs to be updated or not. From the rule-goal graph, the candidate view table is identified. If none of the update attributes are view variables then it is not sure whether the base update will or will not affect the view. Therefore, we have no choice but to update the view to guarantee the data integrity (Lines 1 and 2). If some of the update attributes are also view variables and if the actual constant bindings of these shared attributes already exist in the current view table, it means that the view is affected and needs to be updated (Lines 3 and 4.) If some of the update attributes are also view variables but the actual constant bindings of these shared attributes do not exist in the current view, it implies that the view tuples have not been queried yet by the users. Therefore, updates on the base tables are sufficient and it is not necessary to update the view.

To illustrate-this screening test, consider again the example in Figure 13. If an update on P_2 is successful, it may or may not affect the current view table V. Unfortunately, it is impossible to tell whether the update would affect the view or not since the update attributes of P_2 are free variables (i.e. A and B) that are not part of the view variables. The bindings of A determine the bindings of X, which may imply that the view V is affected. However, the update effect cannot be decided statically. In order to guarantee the data integrity, we have to update the view table.

On the other hand, for the case of P_1, a successful update request may or may not affect the current view V. In this case, however, some of the update attributes are in the view variables. This implies that the actual binding of the first attribute of the update request of P_1 (i.e. X in p_1) will affect the current view if and only if the same binding exists in V. If this is the case, the current view table has to be updated. If the binding of the update request on P_1 does not exist in the current view table V, it implies that the view with these particular bindings has not been derived yet (using the incremental query processing method) and hence the update has no effects on the current view table.

Note that in order to determine if the update attribute constants exist in the current view, it is necessary to test for each adornment pattern of existing view tables. For instance, let us assume that the update request (insertion or deletion) of $p_1(l,m)$ is successful. It is detected that the constant l is bound to variable X which is one of the view variables. Therefore, this update request may potentially affect the view V. The screening test then further investigates if the current partial view tables are affected. In this case, there are at most three current partial view tables: $V(b/X, f/Y, f/Z)$, $V(f/X, b/Y, f/Z)$ and $V(f/X, f/Y, b/Z)$ (since $V(f/X, f/Y, f/Z)$ will subsume all others; $V(b/X, b/Y, b/Z)$ will be subsumed by all others, and $V(b/X, b/Y, f/Z)$ will be subsumed by either one of the above three tables.) That is, the worst case scenario is to generate three separate selection operations to select l in the first column of $V(b/X, f/Y, f/Z)$, $V(f/X, b/Y, f/Z)$ and $V(f/X, f/Y, b/Z)$. The view is *not* affected if all of these operations fail to return tuples (again, it can be implemented by just checking the relative index files instead of actually retrieving tuples.) In summary, each update request may update many indexed files, depending on the adornment of existing partial view tables.

3.2. Updating Affected Views

Now that, given an update request, it can be determined if a view is affected or not by using the screening test. To derive the view tuples that are consequences of a base update (for example, a set of view tuples that are to be deleted or inserted), it has to work from the view definition (i.e. the right-hand side of the rule). The process is similar to querying the view. For example, in a regular query process, the bindings of input variables are passed to the literals of the body of the rule and each literal becomes a subgoal. Each subgoal is solved and its bindings will also be passed onto those remaining literals (i.e. unification). For the case of base updates, the updated base tuple contains the bindings and they are passed onto the other literals in the rule. Each of these literals becomes a subgoal. If all these subgoals are resolved successfully, then the bindings of the input variables are returned, which becomes the affected tuples of the view. If the update is an insertion, these derived affected view tuples are then unioned to the existing view table. If the update is an deletion, these derived affected view tuples should be deleted from the existing view table by applying the difference operation.

For instance, assume that a tuple is successfully updated in P_2 of the example in Figure 13. Conceptually, two join operations between P_1 and P_2, and between P_2 and V are implemented to generate the set of view tuples related to this update (i.e. $P_1 \bowtie P_2 \bowtie^I\text{-}V$.) Since we assume only atomic update requests are allowed, only one update tuple in P_2 can take place at any time. Therefore, these two joins can be separated into two operations. The first join (i.e. $P_1 \bowtie P_2$) is to select the P_1 where its second column is equal to the constant binding of A, the first update attribute of P_2. The other join (i.e. $P_2 \bowtie^I V$) is equivalent to posing the query $v(b/B,\ f/Y,\ f/Z)$ since the constant binding of B in p_2 also bound the binding of B in *v(B,Y,Z)*. The query is evaluated by invoking the appropriate compiled program. If the query is satisfiable, the returned bindings of Y and Z are paired with X to become the tuples affected in view V. If the update is to delete, then the tuples with bindings of (X, Y, Z) int V are to be deleted. If the update request is an insertion, then the tuples of bindings of (X, Y, Z) are to be inserted in the view tables. All relevant index files are then updated accordingly.

Similarly, assume that an update request to insert $p_1(l, m)$ is successful. With the above screening test, the insertion is potentially affecting the current view table. If the constant l is further found to exist as the first attribute of $V(b/X, f/Y, f/Z)$, $V(f/X, b/Y, f/Z)$, or $V(f/X, f/Y, b/Z)$ it implies that it is necessary to update these affected tables. By substituting the inserted tuple of P_1, the view can be expressed as:

$$v(l, Y, Z) : -p_1(l, m), p_2(m, B), v(B, Y, Z)$$

which can further be translated into:

$$\pi_{X,Y,Z}\sigma_{A=m}P_2 \bowtie^I V$$

The above relational algebra expression with the incremental join can be executed as discussed in Chapter 2. As described above, this can further be simplified as two separate steps: select on P_2 to return the bindings of B. For each constant bindings of B, query $v(b/B, f/Y, f/Z)$. The results are then combined with the current view table using set union or set difference, depending on whether the update is an insertion or deletion. For example, if $p_1(l, m)$ is inserted, the resulting tuples will be unioned with V's that are affected. If the $p_1(l, m)$ is deleted, the above algebra will return a set of tuples that should be deleted from the V's that are affected. This can be accomplished by applying the difference operator to the V's and the results of the above expression.

Note that, since the screening test can be potentially pre-determined and the view definition is static, the update procedure can also be pre-compiled similar to the compilation of rules in Chapter 2. For example, updates on P_2 will always trigger updates on the view while updates on P_1 may affect the view but are dependent on the binding of X. Hence, for each potential base updates, it is possible to generate a compiled program for the screening test. Conceptually, for each view definition, there are two sets of pre-compiled procedures. One contains the compiled procedures for each adornment pattern and is used for query purposes. The other set contains the compiled procedures for each potential update of the base tables and is used when the screening test indicates that the update is affecting the view table. The details to implement the compiled update request will be one of the future research directions of this dissertation.

4. Heuristics to Improve Base Updates

The approach discussed above suffers from two shortcomings. First, if the update variables are not view variables, the current screening test always fails (hence the view table has to be updated.) Second, if the update is a deletion, executing the expression with the incremental join operator finds all affected view tuples that will later be deleted by applying the difference operator. Since all these affected tuples have previously been derived (otherwise, they would not be in the current view table), this is obviously a redundant step. With some extra manipulation, as described below, the performance of deletion can be improved.

4.1. The Complete Table Method

To improve on the the first shortcoming, we modify the view mechanism by maintaining all variables without projecting out the unwanted ones. If all variables (including input variables and intermediate variables in the body of the rule) are kept without projecting out any unwanted ones, the screening test becomes more

```
(1):  if update constants are in the Complete view table
          update the Complete view table
(2):  if update constants are NOT in Complete view table
          do nothing (view unaffected)
```

Figure 14

Screening Test With Complete Table

efficient. In the original screening, if the update variables are not view variables, the screening test has to fail since there is not enough information to decide if the view is affected or not. However, if all variables are maintained, even though the update variables are not input variables, the screening test is able to decide if the deletion will affect the view by checking the existence of the constant bindings of the deletion. To illustrate the method, let's consider the same view definition again:

$$v(X,Y,Z) : -p_1(X,A), p_2(A,B), v(B,Y,Z).$$

To query the view with our incremental approach will, at a certain time, need to execute the pre-compiled program of the form:

$$\pi_{X,Y,Z}\{\sigma P_1 \bowtie P_2 \bowtie^I V\}$$

That is, the results from the joins will be projected into the view table by retaining only the attributes X, Y, and Z. Since other attributes (A and B in this case) are"projected out" from the view table, it becomes uncertain whether updates of P_2 could affect V and hence it is necessary to update V.

Our approach is to keep all attributes from the join operation. For each possible view adornment, instead of keeping the view table, a *"Complete"* table which contains all attributes is maintained. For instance, the Complete view table of the above view V becomes $Complete_V(X, A, B, Y, Z)$. The Complete view table is transparent to the user. A query on the view is then translated as a selection and a projection operation on the Complete view. For instance, the query *v(a, Y,Z)* is translated into the following relational algebra expression:

$$\pi_{X,Y,Z}\sigma_{x='a'}Complete_V$$

The expression says to select tuples in the Complete table of view V where the first attribute equals constant a. The results of the selection are then projected into the X, Y, and Z columns.

Since there are no view tables but Complete view tables, for each adornment pattern, the Complete view table is adorned. For instance, a Complete view table of V may be adorned as $Complete_V(b/X, A, B, f/Y, f/Z)$. The adornments are identical to the adornments before. For example, the adornment of this Complete view table implies that X is bound to a constant, and columns Y and Z are free. Note that we do not adorn the intermediate variables A and B since they are not input variables.

Since a view can have multiple definitions (several rules define the same view) and every definition may have different intermediate variables, the final Complete view table contains input variables plus all intermediate variables. (Since intermediate variables can be always renamed, we assume intermediate variables are all distinct.) Therefore, it is *not* necessary that each tuple in the Complete view table will have values for every attribute. For instance, the view v might have an exit clause that contains no intermediate variables as follows:

$$v(X,Y,Z) : -t(X,Y,Z).$$

The Complete view table of v has the input variables (X, Y, and Z) and the intermediate variables (A and B) and is depicted in Figure 15. Note that if the tuple is generated via the exit clause, it does not have the values for A and B.

View V	X	A	B	Y	Z
1.	a	e	f	b	c
2.	a	e	f	b	d
3.	a	g	h	b	c
4.	a	-	-	b	c

Figure 15

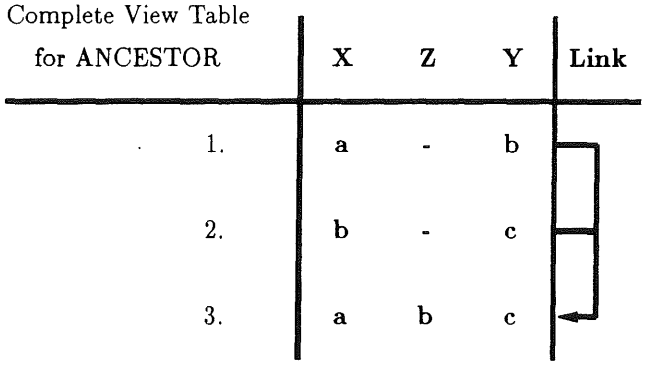
An Example of A Complete View Table

The screening test algorithm can be improved for deletion requests as shown in Figure 14. The new screening test improves mainly on the lines 1 and 2 from the old one in Figure 12. In the original test, if none of the update attributes are view variables, we have no choice but to update the view. Therefore, if the update is to delete a tuple with update attributes that are not view variables, the view still needs to be updated. Since the Complete table includes all variables, this step is modified as shown in line 1 of Figure 14. In the new algorithm, we only check if the update constants (note, not update attributes) are in the Complete view table or not. If they are in the Complete view table, then the view has to be refreshed; otherwise, the view is unaffected. The advantage of this method lies in the fact that, in the case where the update is a deletion, and if the update constants are not in the Complete view table, the view is unaffected. This is not possible in the old screen test since the intermediate variables have been projected out. Note that, for insertion request, there is no such improvement.

To demonstrate the advantage of using the Complete view tables, let's assume that a query is processed successfully for the view $v(a, Y, Z)$. Therefore, a partial view table of $V(b/X, f/Y, f/Z)$ is retained. With the Complete table method, let's assume that the $Complete_V(b/X, A, B, f/Y, f/Z)$ contains tuples as shown in Figure 15. The fourth tuple is a result of the exit clause above and does not have values for the attributes A and B. To derive the view table $V(b/X, f/Y, f/Z)$, it only requires to project out the A and B attributes to obtain the tuples $\{(a,b,c), (a,b,d)\}$ (after the bindings (e,f), (g,h) are projected out, duplicates are eliminated.) Furthermore, consider that an update request to delete a tuple $p_2(l, m)$ succeeds. The old screening test in Figure 12 fails since l and m are values of the intermediate variables. Hence the view has to be updated. However, if the Complete view table is used, the new screening test as shown in Figure 14 will detect that the view is not affected because the table does not contain tuples with bindings of l and m as the attribute values.

4.2. Lemma Dependency Link

The second shortcoming in the original scheme is the implementation of the deletion of view tuples. In order to delete all affected view tuples that have previously been derived, it is necessary to carry out the operation similar to a query to identify these tuples. In this section, we augment the Complete view table to further improve the performance if the update is a deletion. (Note that, however, the lemma dependency described in this section can be implemented without using the Complete view table.) Our method is based on the observation that there exists some data dependency among the view tuples (lemmas) when they are being derived. Therefore, by recording these dependencies in the view table, it is not necessary to call on the pre-compiled program to re-generate the affected tuples and then delete them from the current view table.



Complete View Table for ANCESTOR	X	Z	Y	Link
1.	a	-	b	
2.	b	-	c	
3.	a	b	c	

Figure 16

An Example of Dependency Link

To illustrate this method, consider the resolution tree in Figures 3, 4, and 5 of Chapter 2. There exist two types of view tuples (lemmas): *primary* and *secondary*. Primary view tuples are those generated by means of the exit conditions and hence are not dependent on other tuples in the same view. For example, the tuples *an(a,b)*, *an(b,c)* are primary view tuples. On the other hand, secondary view tuples are those dependent on some tuples of the same view. For example, the view tuple *an(a,c)* depends on the existence of the tuple *an(a,b)* and *an(b,c)*. With this dependency relationship, deletions in the view table can further be improved by augmenting the Complete view table with an extra field to record these dependencies. For example, after the query *an(X,Y)* is materialized (as shown in Chapter 2), the Complete table of ANCESTOR is depicted in Figure 16. Note that the table is augmented by recording the dependency relationship between view tuples in a separate field. For example, the third view tuple is dependent on view tuples 1 and 2, which is recorded by the link as shown. This link is created while tuples are being generated. With this extra link, deletion from the view table becomes simple: upon a successful deletion request on a base table, and provided the screening test indicates that the deletion would affect the current view table, we can simply select all tuples with the binding constants, instead of calling the precompiled program to re-generate all derivable tuples. Then, for each selected tuple, we delete the tuple and follow the link to delete all other dependent tuples. For example, suppose the deletion of $par(a, b)$ succeeds. Instead of calling the pre-compiled program to re-generate the tuples, a simple selection on the complete view table with $(X = a, Y = b)$ will select the tuple 1. This tuple and all tuples reachable via the dependency links (e.g. tuple 3) are candidates to be deleted. However, the dependency of a lemma can be many-to-one, i.e. a secondary lemma can be derived from more than one lemmas. Therefore, it is necessary to keep a reference count for every secondary lemma. A lemma tuple can only be deleted when its reference count is reduced to zero.

There are two disadvantages of adopting the Complete table. First, it is obvious that the Complete table is larger than the actual view table. Second, the query process against the view must perform an additional projection operation on the Complete view table. The main advantage of this method is that updating the Complete view table is much faster when the update request is a deletion. In this case, we don't need to call upon the pre-compiled program to generate all affected tuples. Instead, the process is simplified to delete all tuples with the particular update constants.

5. Chapter Summary

In this chapter, we have discussed how lemma resolution and rule compilation are used in the actual query/update situation. Both queries and updates are processed *incrementally*. In particular, instead of maintaining an entire view table, we proposed that a set of smaller, partially materialized view tables are to be retained and maintained. To answer a query, instead of invoking the compiled program, the smaller and partially materialized view table is first searched. Only if the query cannot be satisfied should the compiled program be invoked. A query process such as ours can improve the performance if there exists some kind of user query patterns, in which users tend to pose repeatedly the same questions. In order to guarantee the partial table method to be complete, we define the concept of table subsumption that can eliminate redundant tables. We further demonstrated that it is only necessary to maintain one physical view tables but with many indexed files that correspond to each of the unique adornment patterns of the view.

Materialization of views can improve query performance but, at the same time, the cost to maintain them can be high. If the view maintenance costs are higher, then the improvement of query performance may not be justifiable. In particular, we define two distinct types of updates: *base updates* and *view updates.* In this chapter, we define what base updates are and how they affect the materialized view tables. We designed a simple but effective screening test to determine if an update request affects the views or not. We also demonstrated how to generate the view tuples that are affected by the base updates so that they can be inserted to or deleted from the view tables.

Finally, we described two methods to improve the update process should the update be a deletion. The first method is to retain all variables (input variables and intermediate variables) in a Complete View table. By doing so, the screening test is improved and is more efficient to determine if the deletion requests affect the view or not. The second method is to augment the query process by linking the dependency of primary and secondary lemmas (view tuples) in the Complete View table. Should the deletion of a base tuple affect the the view, those affected view tuples must exist currently in the view table. It will be redundant to re-generate all these view tuples while they can be linked together via their dependencies. Once the primary lemma is identified to be deleted, all other lemmas that are solely dependent on this primary lemma can also be deleted without any further computation. We believe that these two methods would indeed improve the performance when update requests are to delete tuples in the base tables.

CHAPTER 4
Empirical Studies on Query Processing and Base Updates

As we have seen in Chapter 3, query processing and base updates are closely related. One way to improve query processing time is to retain previously derived answers (hence, materialized views). However, if updates are allowed, maintaining materialized views becomes a major issue. In general, one cannot improve the overall performance by studying and designing algorithms to resolve these two problems *separately* as we have seen it in most of the past studies. All compilation methods such as top-down and bottom-up evaluations were designed with a single-minded objective: to improve the query processing performance, without considering the effects of updates. For example, there are two major approaches to improve query processing performance. The first one is to materialize a complete view table when each view is defined. We shall call this the *totally materialized approach*. On the other hand, the second approach does not materialize any view tables at all. Instead, view tuples are derived only when they are needed by executing its corresponding compiled program. It is called the *on-the-fly* or the *query modification* approach. Each of these two approaches may perform better than the other one in certain situations. One of the major factors is the frequency of updates. For example, if the update ratio is higher, the query modification performs much better than the totally materialized method because it does not need to maintain the consistency of the view tables. On the other hand, if the update ratio is low (or none), the totally materialized method is better because the view table is materialized at view definition time. Any queries on the view become simple table selection operations without computing the view at run time.

These two approaches are the two extremes that are based on the premises either to have faster query response time (totally materialized method) or to have faster update response time (the on-the-fly method). We believe that, to improve the overall performance (average of query and update response time), we should design new algorithms based on the trade-off and inter-relationship of these two processes. Our incremental query and update approach introduced in Chapter 3 provides the solution in this direction. To be more specific, the query processing strategy by maintaining smaller view tables as introduced in Chapter 2 is designed to be integrated with the base update strategy as discussed in Chapter 3 into one unified solution.

In this chapter, we shall study the behavior of our incremental approach in comparison with the totally materialized approach and the on-the-fly approach. It is obvious that there will not be one single approach that outperforms the others. Rather, each of these three approaches will have their applications in different situations. We would like to identify the situations where our approach is more desirable than the other two.

1. Cost Components of the Three Approaches

In this section, we describe how these three approaches are different in processing queries and update requests. In particular, we show where major computational costs occur.

Given a set of requests that contains a mixture of queries on the view tables and updates to the base tables, we identify the different cost components of the totally materialized view approach, on-the-fly approach and our incremental update approach.

1.1. Costs of the Totally Materialized View Approach

The Totally Materialized View Approach (MA for short) is to materialize a single complete view table at view definition time and maintain the view throughout base updates. The following cost components are identified:

(1) (MA1): the cost to materialize the complete view table – this cost is basically the execution time of a compiled program that is based on a certain algorithm, e.g. top-down, bottom-up, magic sets, and so on. The efficiency of these methods have been studied thoroughly in the past. As shown in [BR86], the time is dependent on the selectivity of the supportive base tables which, in turn, determines (a) the depth of recursion and (b) the size of each intermediate resulting table. The time (MA1) is the major overhead of the total computation; it occurs only once when the view is defined.

(2) (MA2): the cost to query the view table – this cost is a straightforward implementation of relational selection on the view table; the time (MA2) is the variable cost of the total computation time; it occurs every time a query on the specific view is processed.

(3) (MA3): the cost to update the base tables – this cost is identical to inserting or deleting a tuple to a base table in a relational system; the time (MA3) is another variable cost of the total computation time.

(4) (MA4): the cost to maintain the view table integrity when the supportive base tables get updated; the time is determined by (a) the efficiency of the screening test; (b) the view_ maintenance algorithm as discussed in Chapter 3. The time (MA4) is also a variable cost of the total computation time; it is triggered every time when an update of a supportive base table is successful.

Apparently, the major costs of this approach are to materialize the view table (i.e. MA1) and to maintain the view table (i.e. MA4). The materialization cost (MA1) is significant because it is the most expensive computation especially for a recursive view. The maintenance costs (MA4) is significant since any base update may trigger the view maintenance mechanism to validate the view table. If the effects of update cannot be delayed (e.g. updates have to be done in real time), then the validation of the view table becomes significant.

However, the potential gain of this approach is the speedup on processing queries. Therefore, intuitively, if the application involves infrequent updates or updates that can be done in batch, then the speedup of processing queries of this approach becomes attractive.

1.2. Costs of the On-The-Fly Approach

The On-The-Fly Approach (FLY for short) does not materialize any view tables. Therefore, there are no view maintenance costs. Instead, the view definition is *compiled* by a certain method such as top-down, bottom-up, magic sets, and so on. We identify the following cost components:

(1) (FLY1): the cost to query the view – the time is identical to the query evaluation which is, basically, the execution time of the compiled program of the query. For example, query on the view *ancestor(a, X)* is processed by executing the compiled program of ancestor for the adornment pattern of *(b, f)*. Similar to (MA1), it is related to the selectivity and the recursion structure.

(2) (FLY2): the cost to update the base tables – the time is the same as (MA3).

In this approach, there is *no* fixed overhead cost. The major cost component is the query processing time (i.e. FLY1). Base updates are simply the time to spend on updating the affected base tables, which is identical in all three approaches since there is no need to propagate the update effects. The on-the-fly approach and the totally materialized approach are two extremes of a query processing strategy. In situations where update activities are heavy, the on-the-fly approach is apparently more attractive. As shown in [HAN87], if the selectivity and update ratio are high, it is better not to materialize views. However, if no base update is allowed or if batch processing is allowed to process update requests and if the recursion of the view is shallow, the totally materialized approach may be better, especially when some of the queries are repetitive. As discussed in Chapter 3, there exists a *locality* of query processing, i.e. repetition of queries. We will investigate how these factors (e.g. selectivity and update ratio) will affect these two methods along with our incremental method.

1.3. Costs of the Incremental Query and Update Approach

Our incremental approach (INC for short) as described in Chapter 3 is a compromise between the totally materialized method and the on-the-fly method. When a view is defined, it is not materialized at once. Only when query on the view is posed by the user and is successfully retrieved, are partially materialized view tables created and maintained. We identify the following cost components:

(1) (INC1): the cost to process view queries – the time parameter has two components. First, if the related view table exists, the view is processed identically to (MA2); if the answers are found then the query processing is finished. If there is no relevant view tables or the answers cannot be found, the query will be processed as in (FLY1). That is, the related compiled program for the query is executed. After the query is processed, certain partially materialized tables are generated by retaining the bindings of the query.

(2) (INC2): the cost to update the base tables – this time is the same as (MA3).

(3) (INC3): the cost to maintain the view table integrity when the supportive base tables get updated. This is dependent on the screening test and the view maintenance algorithm as discussed in Chapter 3.

Our incremental query and update processing is designed based on the existence of the so-called database locality in which users tend to ask certain queries repeatedly. It suggests that maintaining these often repeated views is justifiable. Neither the totally materialized approach nor the on-the-fly approach takes advantages of this locality property. Our approach thus becomes a compromise between them because it retains only those queries users have asked before without the heavy costs of materializing and maintaining the entire view tables as in the totally materialized view approach. On the other hand, repeated queries can be simply answered via a table lookup, which may show significant improvement over the on-the-fly approach which does *not* take advantage of repeated queries. In this chapter, we will verify this intuition and identify the situations where our method renders better performance than the other two.

2. Related Work

As noted in the introduction of this chapter, research in the past seems to concentrate on studying query processing strategies and base update strategies *separately.* Query processing strategies focus on different algorithms to compile queries [BR86]. Base update strategies concentrate on the differences among various view maintenance algorithms. Furthermore, all these view maintenance algorithms are designed with the select-project-join (SPJ) views in mind and thus are not recursive [HAN87, BLT86, BC79]. Even so, as shown in [HAN87], one conclusion that can be drawn is that if updates tend to be the major operation and the selectivity is high, it is better to use the on-the-fly approach instead of materializing and maintaining the views. Recursive views will surely increase the computational costs. The number of levels of recursion is directly related to the selectivity between the tables defining the view. Therefore, it can be speculated that the same conclusion still holds in a recursive view.

To our knowledge, this is the first study that takes query repetition into consideration within a framework unifying query and update processes together to achieve a general performance improvement.

3. Experiment Design

In order to test our intuitions on these three approaches, we experiment by studying their behaviors empirically. The major objectives of the experiment are to:

(1) compare *only* the query processing performance of our incremental method against the totally materialized method and the on-the-fly method;

(2) compare these three methods in a more realistic situation where updates and queries are mixed.

Rather than implementing the three methods to handle general rules, we decided to apply them to the ancestor view of Figure 9 in Chapter 3. The reason of picking the ancestor view is that it is a single recursion and is believed to have the most applications in practice. (For example, in [NS88] the Laguna Beach Database Report, the panelists tended to believe that double recursion or any other complex recursion have little to no practical use while the single recursion is significant in any deductive database applications.)

3.1. The Query Process

The ancestor view, as discussed in Chapter 2 contains one recursive view definition that can be expressed in terms of the incremental join operator as follows:

$$PARENT \bowtie^{I} ANCESTOR$$

For our incremental method, we derived four procedures according to the four adornments of the ancestor view. They are, *ancestor(f/X, f/Y)*, *ancestor(f/X, b/Y)*, *ancestor(b/X, f/Y)*, and *ancestor(b/X, b/Y)*. As the query process begins, partially materialized tables ANCESTOR(f/X, b/Y), ANCESTOR(b/X, f/Y), ANCESTOR(f/X, f/Y), and ANCESTOR(b/X, b/Y) are generated. The time (INC1) to process a query is either a table lookup in the related partially materialized table or, if it cannot be found, the time to invoke the compiled program corresponding to the query's adornment. Answers are incrementally accumulated. However, since *ancestor(f/X, f/Y)* would materialize the entire ANCESTOR table and thus reduce the case to the totally materialized approach, we decided not to allow this kind of *for_all* query in our test.

For the totally materialized method and the on-the-fly method, the query processing times (MA1) and (FLY1) are basically the times to execute some compiled programs. However, as mentioned in Chapter 2, there are different methods to compile a recursive query. For example, the Henchen and Naqvi's linear compilation has the best performance of all approaches (but is restricted to linear recursion; otherwise it has a termination problem.) In view of this, we decided to compute (MA1) and (FLY1) using this method which can be shown as in the following formula [HL86]:

$$\sigma_a PARENT \bowtie^k PARENT$$

where $\sigma_a PARENT$ denotes a selection of a on the corresponding attribute of the relation PARENT, and $\bowtie^k PARENT$ is the join operations on corresponding join attributes of the relation PARENT $k-1$ times, where k ranges from 0 to n and n is the number of iterations up to the termination point. The termination point is reached when the formula generates no more new results. This is always guaranteed since, first, it is a single recursion and second, data are all non-cyclic in our test. To be specific, the formula is expanded as follows:

n=0	$\sigma PARENT$
n=1	$(\sigma PARENT) \bowtie PARENT$
n=2	$(\sigma PARENT \bowtie PARENT) \bowtie PARENT$

n=3 $(\sigma PARENT^2 \bowtie PARENT) \bowtie PARENT$

......

n=k $(\sigma PARENT^{k-1} \bowtie PARENT) \bowtie PARENT$

Note that the results of operations in parentheses are retained from each previous step; the strategy is called a *single wavefront* in [HL86].

The time (MA1) for the totally materialized method is basically the time to execute this compiled program where the selection operation σ_a is not needed (because it will select all tuples.) Results are retained as the view ANCESTOR table; this is done only once when the view is defined. Any future queries on the ANCESTOR table (hence time (MA2)) are simple table lookups implemented as $\sigma_y ANCESTOR$ where y is the binding constants of the query.

Similarly, for the on-the-fly method, each query is a call to this program with the specific selection operation corresponding to the query's bindings, hence the time (FLY1). However, the difference between the totally materialized method and the on-the-fly method is that the intermediate results are not retained permanently for the on-the-fly method. They are discarded once the query is answered.

3.2. The Update Process

Since we are only concerned with base updates in this study, updates are on the PARENT table only. The cost to process an update request has two parts. The first one is the time to update the base table PARENT. This may be a straightforward insertion or deletion operation on PARENT. Therefore, it will be the same for all three methods. Hence, (MA3), (FLY2) and (INC2) are identical. The second one is the time to maintain the view tables that are affected by the base update. This only occurs for the totally materialized method and our incremental method; it is not applicable to the on-the-fly method since the latter does not retain any physical view tables.

For the totally materialized method, an update request will definitely affect the ANCESTOR table if the base update is valid (i.e. if it causes tuples to be added to or deleted from the PARENT table successfully.) Therefore, the screening test is trivial and does not add on to the total cost: each valid update on PARENT table will trigger the maintenance of the ANCESTOR table. In this case a procedure similar to the query process discussed in section 3.1 is invoked. For example, if {add PARENT(123,456)} is a valid base update, then the compiled program is invoked where σ_a of the formula is to select those tuples whose column 1 and column 2 are equal to *123* and *456*, respectively. The resulting tuples are then appended to the ANCESTOR table if they are unique. The same procedure is applied if the update is a deletion except that the resulting tuples are then eliminated from the ANCESTOR table. This constitutes the time (MA4).

For our incremental method, a valid base update may *not* necessarily trigger the view maintenance process. However, the screening test is simple. If a base update is valid, a selection with the binding of the first attribute of the base update is invoked and applied to each of the currently existing partial view table. Then the view maintenance procedure described in section 3.2 of Chapter 3 is applied. This forms the time (INC3). Furthermore, tuples in these partial ANCESTOR tables are linked by the dependency pointers and reference count between them (as discussed in section 4.2 of Chapter 3), which improves the performance when the update request is a deletion.

To summarize, if n is the total number of queries and updates in a single run of our experiment, the *average* performance for the three methods can be expressed as:

$$\begin{aligned} t_{MA} &= (MA1 + MA2 + MA3 + MA4)/n \\ t_{FLY} &= (FLY1 + FLY2 + FLY3)/n \\ t_{INC} &= (INC1 + INC2 + INC3)/n \end{aligned}$$

3.3. Empirical Study Parameters

By analyzing the cost components in section 3.2, we identify three major factors that affect the performance of the three approaches. They are (1) the selectivity between supportive base tables, (2) the repetition of queries, and (3) the frequency of updates.

Selectivity refers to the *join selectivity* [HL86] between tables which, basically, defines the depth of the recursion. In the case of the ancestor view definition, the selectivity is the join selectivity of the PARENT table. Since the PARENT table has only two columns, selectivity is further defined as a percentage of attribute values in column two that also appear in column one. The higher the selectivity, the deeper of recursion and more costly the computation will be.

Repetition of queries is the measure of how many queries are repeated. The repetition is not significant to the on-the-fly approach but can be very significant for the totally materialized approach and our incremental approach. Update frequency refers to how often the base tables are updated.

The experiment was conducted as follows:

(1) a PARENT table containing 500 tuples of two integers was generated with a specific selectivity;

(2) a query BATCH containing 1000 queries (e.g. Ancestor(100, Y)) and/or updates (e.g. {Add Parent(100, 312)} Or {Delete Parent(200,167)}) was generated according to the specific update ratio and/or repetition ratio.

(3) for each test, the same PARENT table was used in each of these three methods. The actual execution time to finish each BATCH was recorded. Since an empirical study such as this will suffer from random noise, for each test, ten BATCHes were generated and run against the same PARENT tables. The averages of these ten results were computed and became our data points.

More specifically, the query BATCH (B) is expressed as $B = (Q + U)$ where Q and U are the number of queries and updates, respectively. Update ratio is expressed as $U/(Q + U)$.

Repetition ratio determines how many queries are repeated. If the number of repeated query is q, then the repetition ratio is defined as q/Q. For example, a 10% repetition of 1000 queries means that 100 queries are repeated. Of these one hundred queries, however, it makes a big difference whether a *single* query gets repeated 100 times or 50 queries repeat one time each. Apparently, the latter is the worst case scenario. We decided to use the worst case in our experiment. Therefore, a 10% repetition means that for every 10 tuples the first query is repeated; 15% repetition means that every 20 tuples the first three queries are repeated, and so on. All repetitions are distributed evenly in the query BATCH.

Furthermore, the timing of updates is significant. If the updates are all done at the beginning or at the end of the query BATCH, it may benefit a certain method over the others. To make the comparisons unbiased, updates are distributed evenly throughout the query BATCH and updates are divided equally into insertion and deletion.

The compiled programs described in section 3.1 were implemented in "C" language. Two index files on the first and second columns of the PARENT table were generated and maintained with B+ index trees. Experiments were run on an i386 33MHz machine without cache in a single user mode.

4. The Empirical Study and Its Results

We carried out three separate tests in this study:

(1) a comparison of all three methods by varying selectivity and repetition ratio;

(2) a comparison of all three methods by varying selectivity and update ratio while holding the repetition ratio constant;

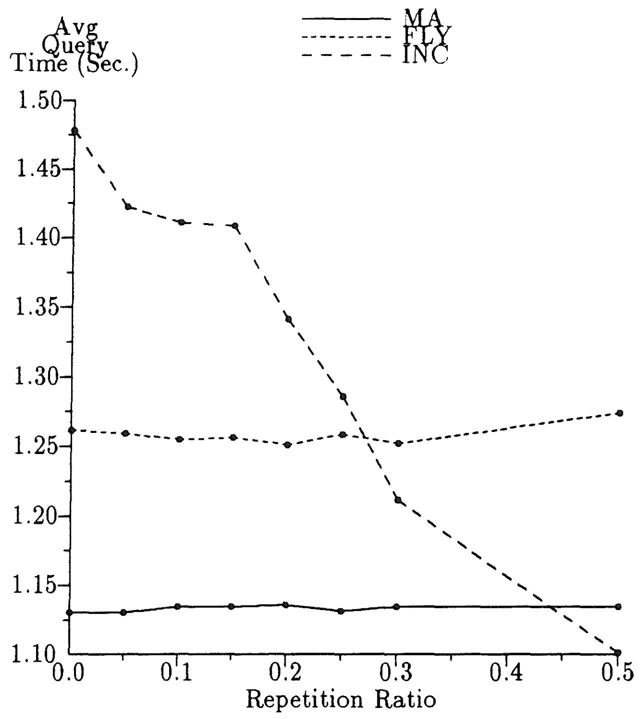


Figure 17

Selectivity= 0.00 With No Updates

(3) a comparison of only the on-the-fly approach to our incremental method as in (2) but with a wider spectrum of update ratios;

4.1. Comparisons of MA, FLY and INC With No Updates

The first study is the most fundamental test to compare the performance of query processing of the MA, FLY and our INC methods. Since the main objective is to investigate how well these three approaches behave in terms of answering queries, the query BATCH contains only queries without any update requests. The parameters to be tested are the selectivity and repetition ratio. The selectivity ranges from 0% to 15% while the repetition ratio is between 0 to 50%. The results are plotted in Figures 17,18, 19, 20 and 21. We make the following observations:

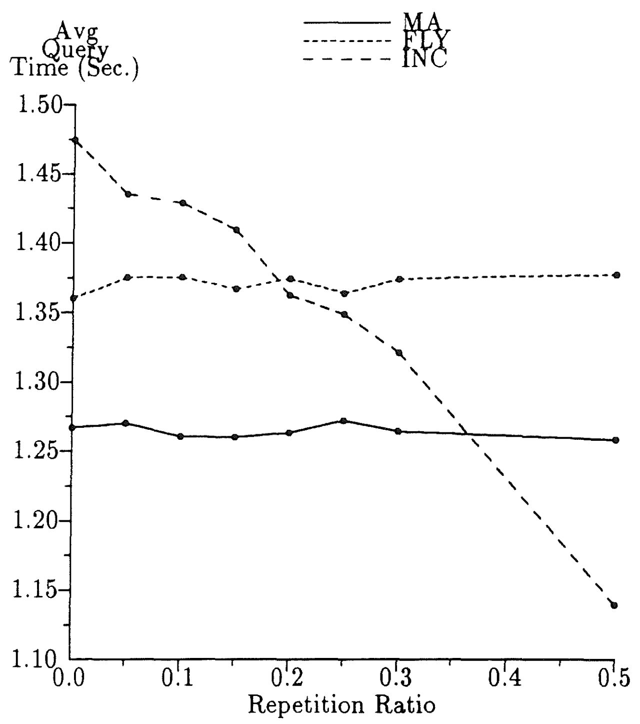


Figure 18

Selectivity= 0.05 With No Updates

(1) No method is superior to the others in all circumstances.

(2) MA is better than FLY for low selectivity (up to 0.15) as shown in Figures 17, 18, 19 and 20. It becomes worse for higher selectivities (Figure 21).

(3) INC is worse than either MA or FLY when selectivity and repetition ratio are low *but* surpasses both methods as selectivity and/or repetition ratio increase.

To capture these observations quantitatively, we plot the data as pair-wise comparisons between the three methods in Figures 22 and 23. In each figure, the rectangular area represents the possible space of selectivity and repetition ratio pairs. The curve dividing the area into two represents the cut-off points between

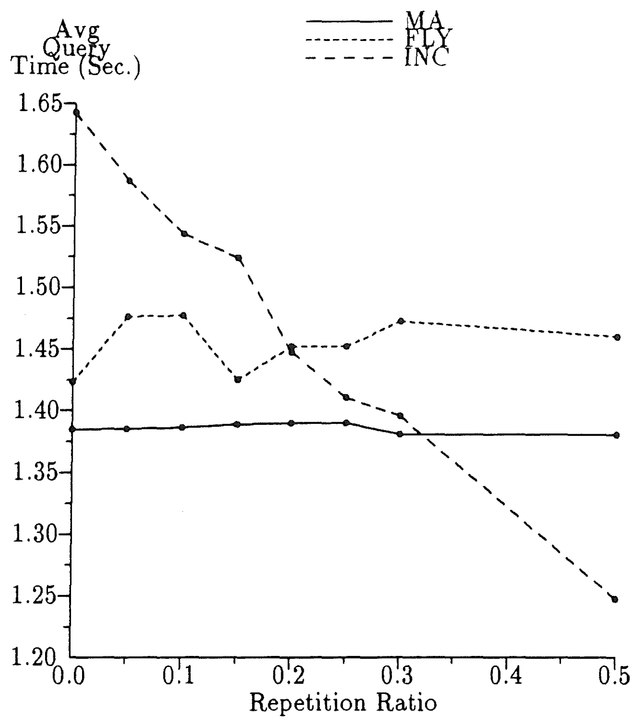


Figure 19

Selectivity= 0.075 With No Updates

the two respective methods. In particular, Figure 22 shows that FLY is superior to MA for selectivity 0.15 or higher, regardless of the repetition ratio.

When comparing the totally materialized to our incremental method, we obtain the graph as depicted in Figure 23. In this graph, we notice the importance of both selectivity and repetition ratio for our incremental method. Notice that the curve divides the area almost diagonally. This implies that INC improves steadily over MA as either selectivity or repetition (or both) increase.

Similarly, we compare our incremental method to the on-the-fly method and depict the results in Figure 24. As in the previous case, INC improves over FLY with increase in selectivity and repetition ratio. The former, however, is much

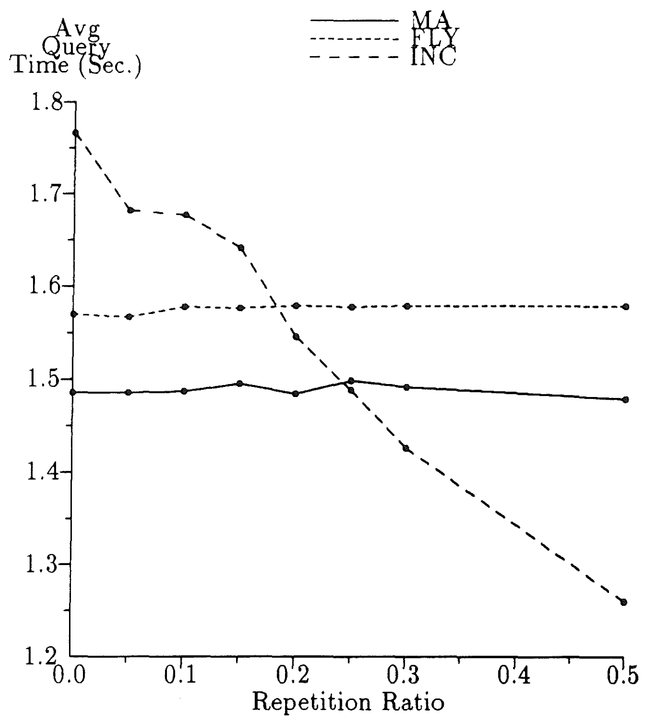


Figure 20

Selectivity= 0.10 With No Updates

more significant, since even a small change in selectivity can tilt the scales in favor of one or the other method.

These observations confirm some of our beliefs about the three methods. First, selectivity plays an important role with all three methods. However, the totally materialized method is affected by the selectivity more severely than the other two. For instance, examining Figures 17, 18, 19, 20, and 21, we notice that the gap between the totally materialized method to the incremental and on-the-fly methods closes up more rapidly as the selectivity increases. As the selectivity increases up to a certain point, the totally materialized method actually performs worse than the other two. This confirms our belief that not all view tuples are queried by the users throughout the lifetime of the view. In this case, since there

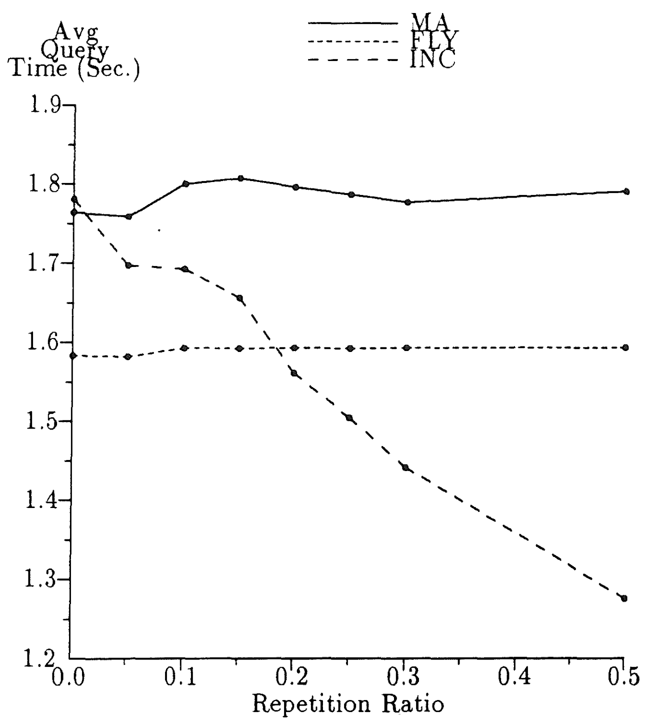


Figure 21

Selectivity= 0.15 With No Updates

are only one thousand queries in our test, it simply implies that many tuples generated and kept in the view table have never been used to answer these queries. The higher the selectivity, the deeper the recursion and hence the more costly the computation.

Second, repetition does not significantly affect either the on-the-fly or the totally materialized methods. On the other hand, our incremental method benefits tremendously from an increase of repetition ratio. As mentioned in section 1, a primary motivation for designing the incremental method was to take advantage of the repetition of queries. A partially materialized view table is, in fact, very much like a *cache* when answering queries. The higher the repetition ratio, the higher

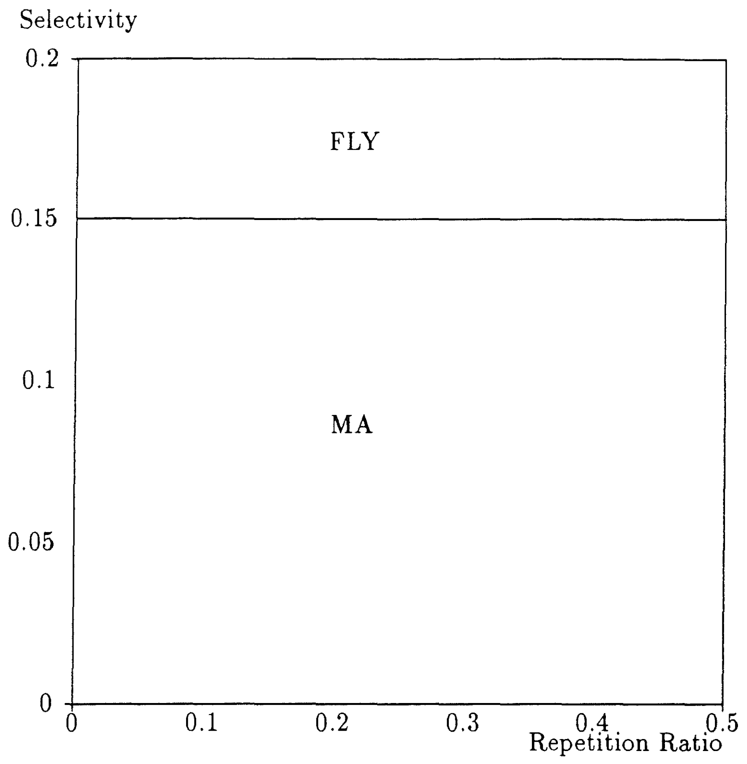


Figure 22

FLY vs MA With Selectivity

and Repetition (No Updates)

the hit ratio will be. Hence, the incremental method satisfies this important design goal.

Lastly, both the totally materialized and on-the-fly methods are using the Henchen and Naqvi's linear compilation method that has been shown to be the most efficient way to answer single recursive queries. However, we demonstrated here that, when the repetition ratio factor is introduced, the linear compilation method is no longer superior to other approaches. Specially, with any non-trivial repetition ratio, our incremental method actually outperforms the other two methods that use linear compilation.

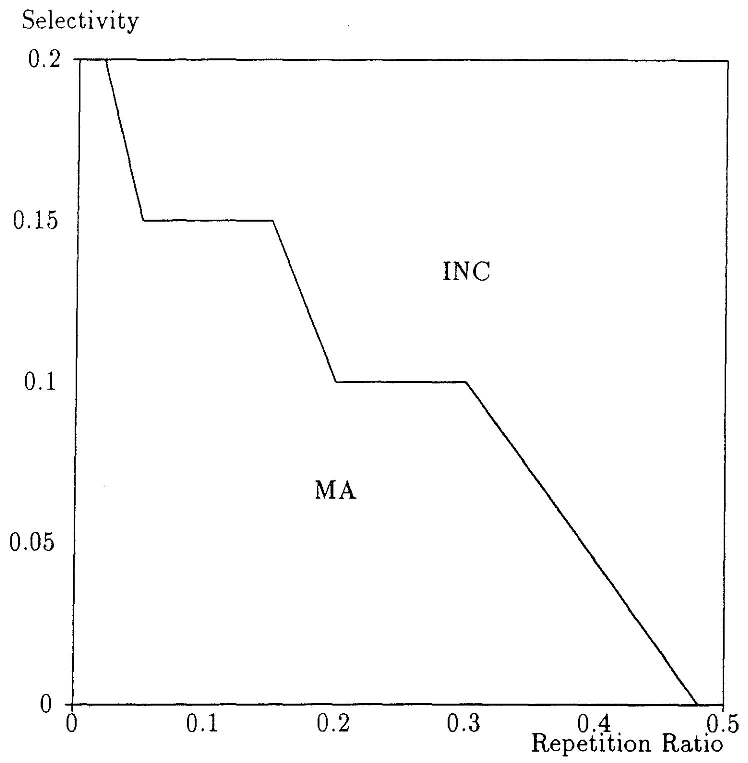


Figure 23

INC vs MA With Selectivity

and Repetition (No Updates)

4.2. Comparisons of MA, FLY and INC With Updates

The second test was to compare these three methods with respect for both queries and base updates. Among the three major factors, namely, selectivity, update ratio and repetition ratio that affect the performance of the three methods, and in section 4.1, selectivity is identified to be very significant while repetition factor affects only our incremental method. Therefore, in this experiment, we held the repetition factor as a constant in order to study how selectivity and update ratio would affect the performance.

From the data obtained in section 4.1, we discovered that when the repetition factor was about 20% and the selectivity was about 10%, these three methods

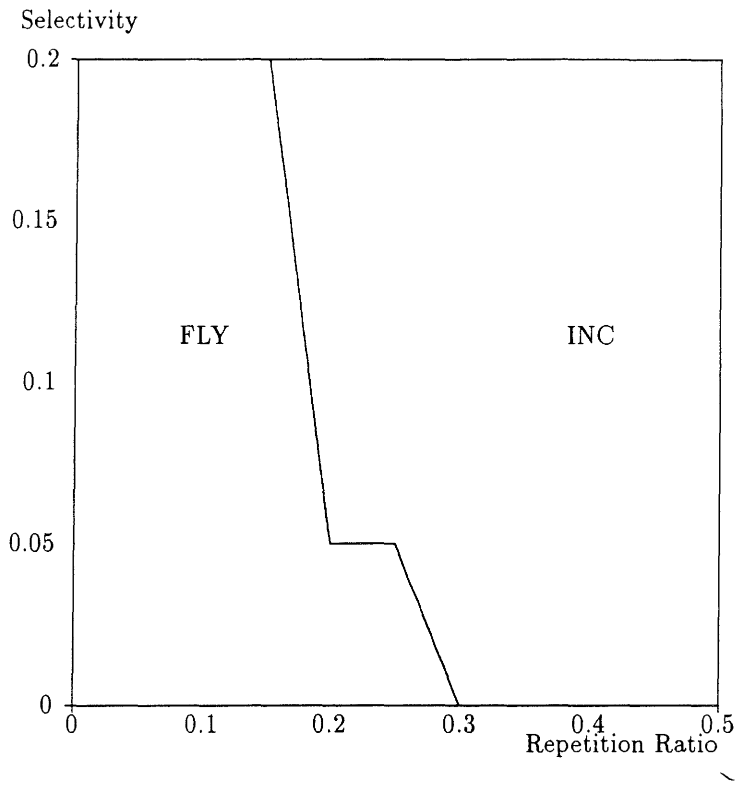


Figure 24

INC vs FLY With Selection

and Repetition (No Updates)

displayed performance very much close to the others, with a slightly advantage for the totally materialized method. With these two factors, we carried out the experiment by varying the update ratio and selectivity while holding the repetition ratio as a constant of 20%. The selectivity was varied between 10% to 30% while the update ratio ranged from 0% to 10% which is on the lower side of most database applications.

The results of this experiment are depicted in Figures 25, 26, 27, 28, and 29. It is obvious that the totally materialized method did not stand up to the test. It performed very much like the other two methods when there were no updates. However, even a small number of updates degrades this method to the worst

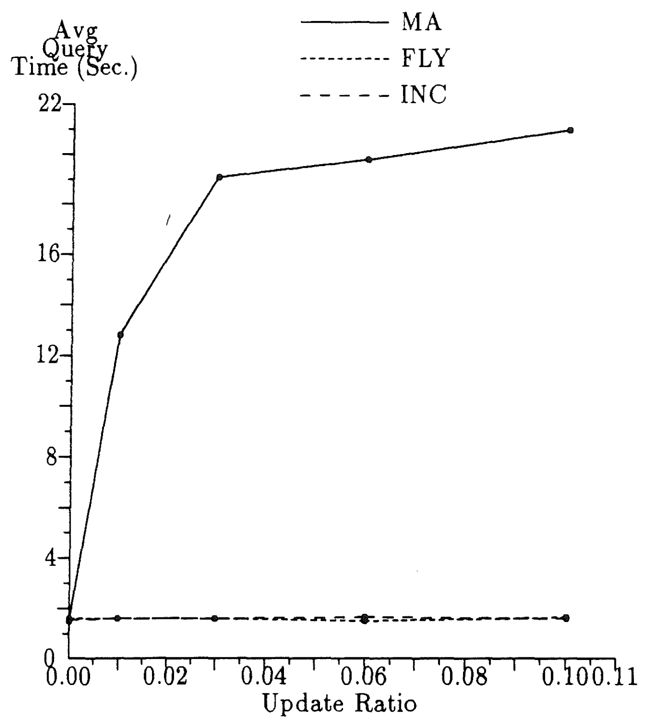


Figure 25

Selectivity= 0.10 With 20% Repeated Queries

position. Relatively, both our incremental method and the on-the-fly method maintained a comparable performance.

The results actually confirm our intuition that when updates are allowed, the totally materialized method suffers the most. There are two major factors that work against this method. First, as discussed in the previous section, the method's performance deteriorates as selectivity increases. Second, any base updates in the *ancestor* view will trigger the view maintenance procedure because the view is materialized in its entirety. Of course, with a more efficient screening test and view maintenance algorithms, this method could be improved. Unfortunately, most of the past studies focused on the improvement for the SJP views that are not recursive. Improving the screening test and view maintenance algorithms for

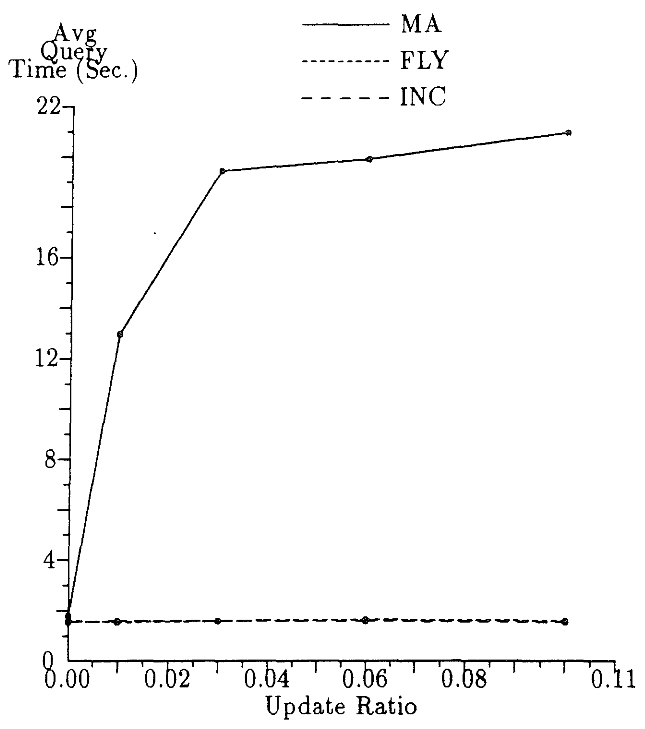


Figure 26

Selectivity= 0.15 With 20% Repeated Queries

recursive queries (of which, again, the single recursive has special interest) is still an open problem.

As noted in section 1.2, update ratio does not significantly affect the on-the-fly method since it does not maintain any view tables. On the other hand, our incremental method *does* react to any increase in updates. When examining the data, we diseover that, with the rising update ratio and lower selectivity, our incremental method performs worse than the on-the-fly method. However, if the selectivity increases, our incremental method outperforms the on-the-fly method. In view of this behavior, we decided to carry out the next experiment that concentrates on the comparison between our incremental method and the on-the-fly method.

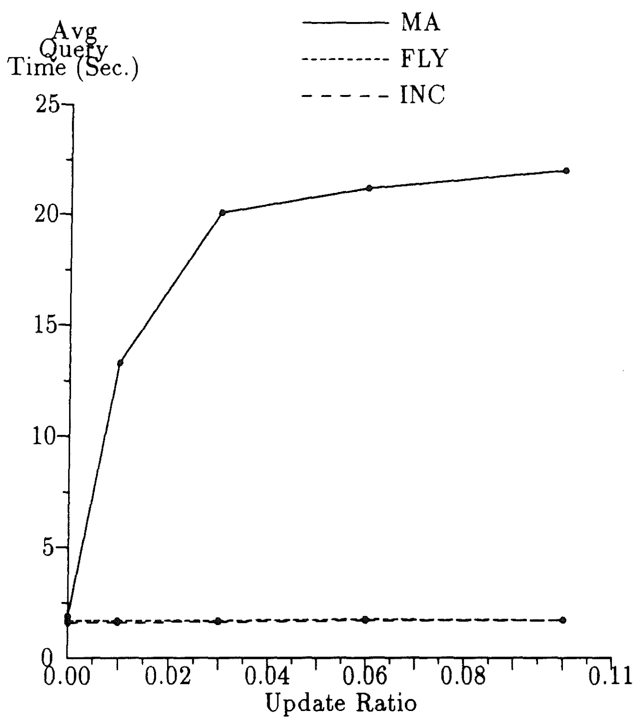


Figure 27

Selectivity= 0.20 With 20% Repeated Queries

4.3. Comparisons of FLY and INC With Updates

In this experiment, we studied how selectivity and updates would affect the performance of our incremental method and the on-the-fly method. From the previous experiment, we noticed that update ratio affected our incremental method while selectivity affected the on-the-fly method. However, the range of update ratios in that study was relatively small (e.g. from 0% to 10%.) In this study, we still kept the selectivity from 10% to 30% but the update ratios were varied from 0% to 100%. Again, the experiment used a 20% query repetition ratio as before. The results are plotted in Figures 30, 31, 32, 33, and 34, and are summarized in Figure 35.

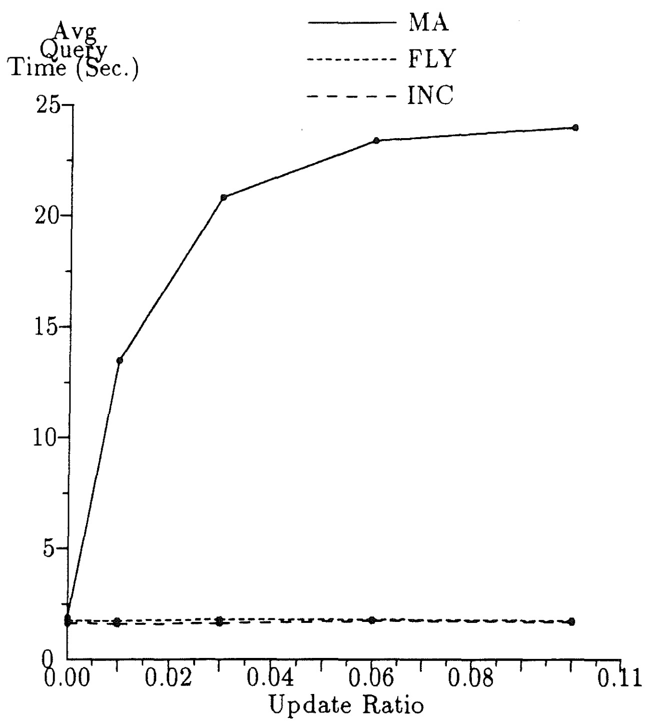


Figure 28

Selectivity= 0.25 With 20% Repeated Queries

With these results, we notice that our incremental method performs better when the selectivity is high and the update ratio is low. We further notice that the on-the-fly method is more sensitive to selectivity than our incremental method. For instance, even the update ratio is high (e.g. 25%), our method is better than the on-the-fly method if selectivity is high. This observation is quite intuitive. Since our incremental method does not generate the entire view table, not every update request triggers the view maintenance procedure. Therefore, even when the update ratio gets higher, it does not necessarily increase the time to process the whole BATCH. One can further observe from Figures 30, 31, 32, 33, and 34 that, the higher the update ratio, the less time it will take to finish the BATCH.

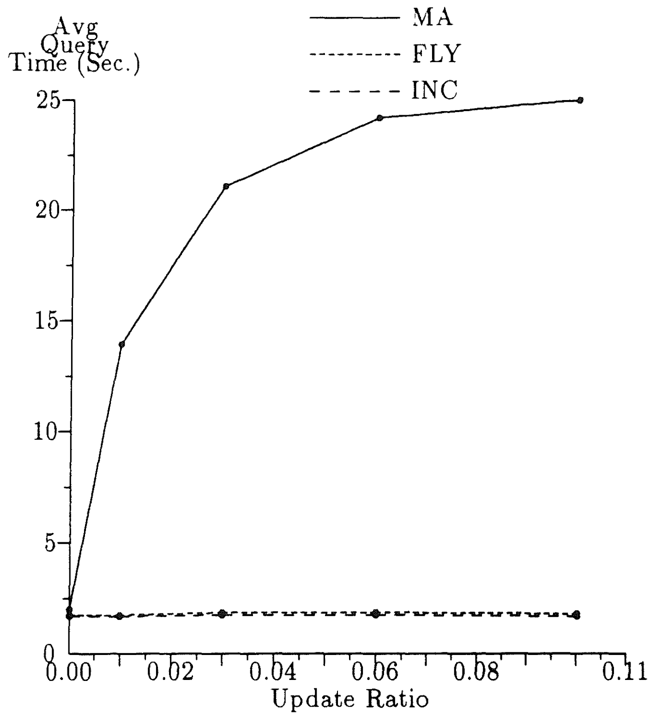


Figure 29

Selectivity= 0.30 With 20% Repeated Queries

This is because if there are more updates than queries, it has a higher chance that these updates won't affect those few partially materialized view tables.

Since we held the repetition factor to 20% this experiment, we can further infer that a higher repetition ratio will be definitely in favor of our incremental method.

5. Chapter Summary

In this chapter, we implemented and compared three methods to study query and base update performance. Two of these methods, the totally materialized and the on-the-fly method, have been studied in the past in the relational database research. They represent two extreme approaches to improving performance of

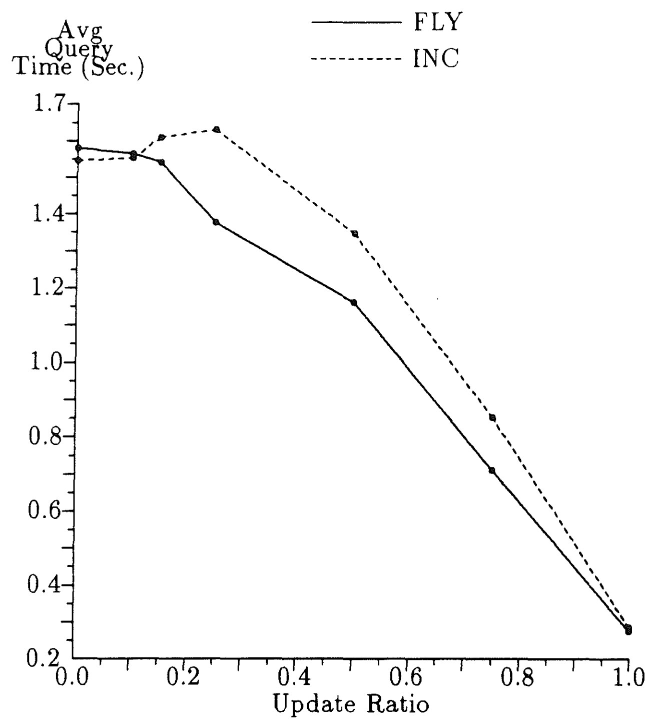


Figure 30

Selectivity= 0.10 (FLY vs INC)

both queries and updates. As demonstrated in [HAN87], when the selectivity and update is higher, views are better off not materialized. On the other hand, when the selectivity and update ratio are low, the materialized method becomes better. For example, a historical database would be an excellent candidate for the totally materialized method since update is rare. Furthermore, when it needs to update, the updates can be done in batch instead of real time. On the contrary, an airline reservation database may call for the on-the-fly approach since updates are very frequent in such an environment.

We believe that our incremental method is a compromise between the totally materialized and on-the-fly methods. The motivation is the database locality of repeated user queries. It makes much sense to retain and maintain not every view

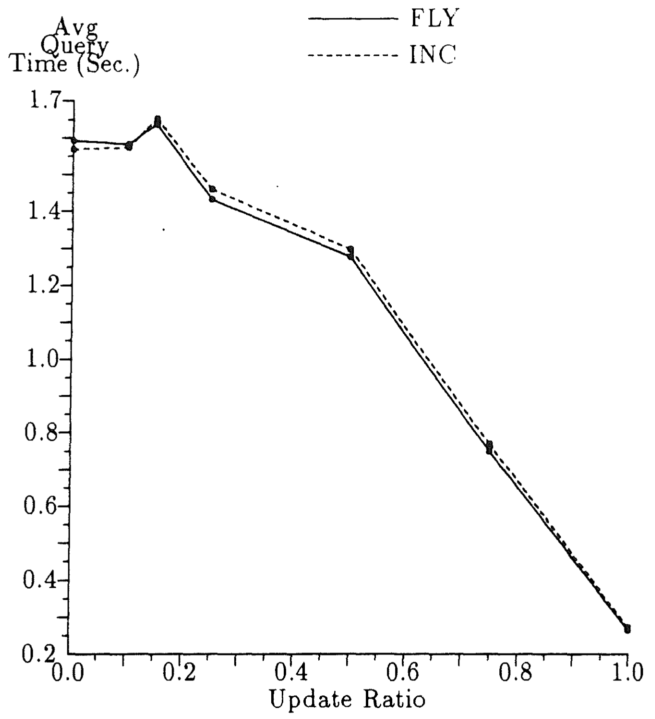


Figure 31

Selectivity= 0.15 (FLY vs INC)

tuples as in the totally materialized method but only those that have been asked previously. None of the past studies has addressed this issue.

The empirical results confirm that our method is indeed a compromise between the on-the-fly and the totally materialized method. Our method is more attractive when there are sufficient repetitions of queries and high selectivity while the update is relatively less frequent. There are applications that fit into this type of environment. For example, in a manufacturing database application (such as MPS (master production scheduling)), the bill of materials may contain a single recursive part-subpart relation. It also needs updates on base tables but not too often. Queries regarding some derived views may repeat very often for a specific manufacturing project. Therefore, instead of creating an entire table as in the

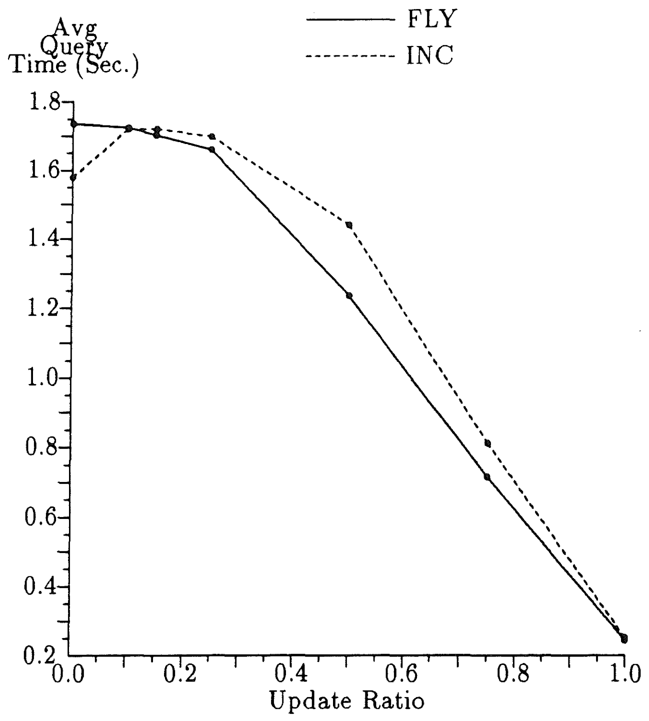


Figure 32

Selectivity= 0.20 (FLY vs INC)

totally materialized method or computing the view tuples on the fly, we could take advantage of the incremental method to achieve better performance.

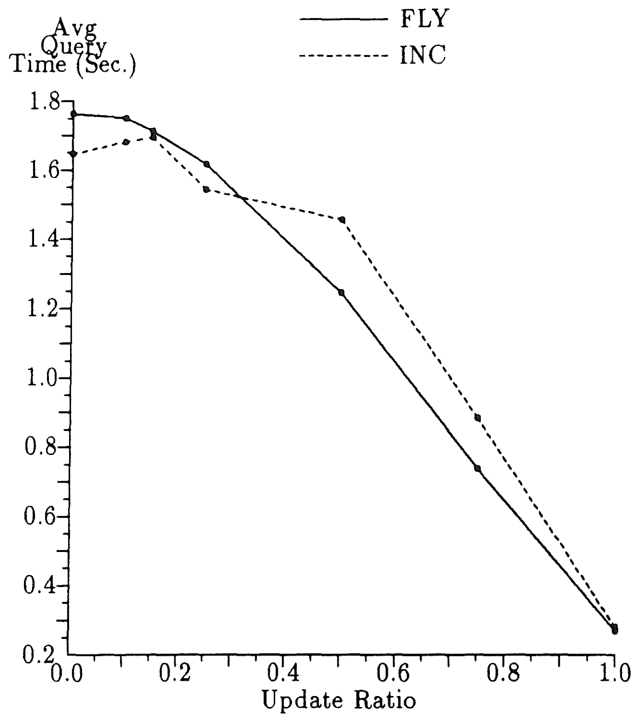


Figure 33

Selectivity= 0.25 (FLY vs INC)

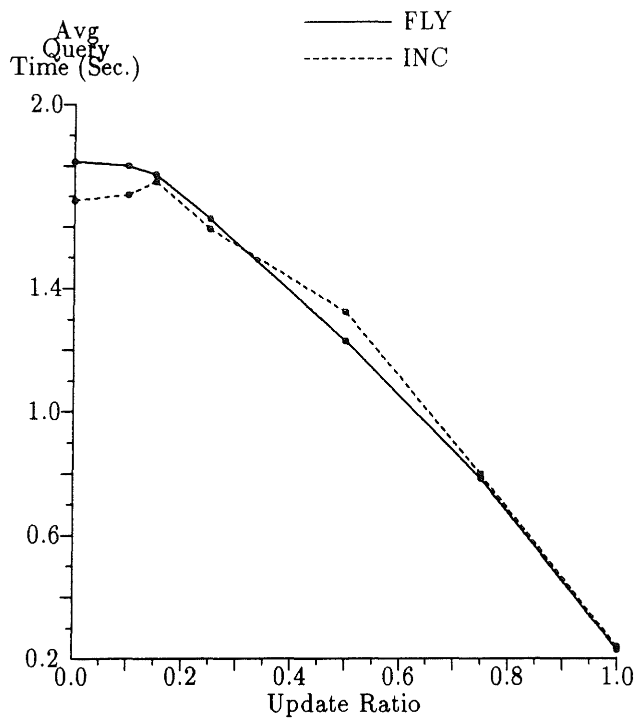


Figure 34

Selectivity= 0.30 (FLY vs INC)

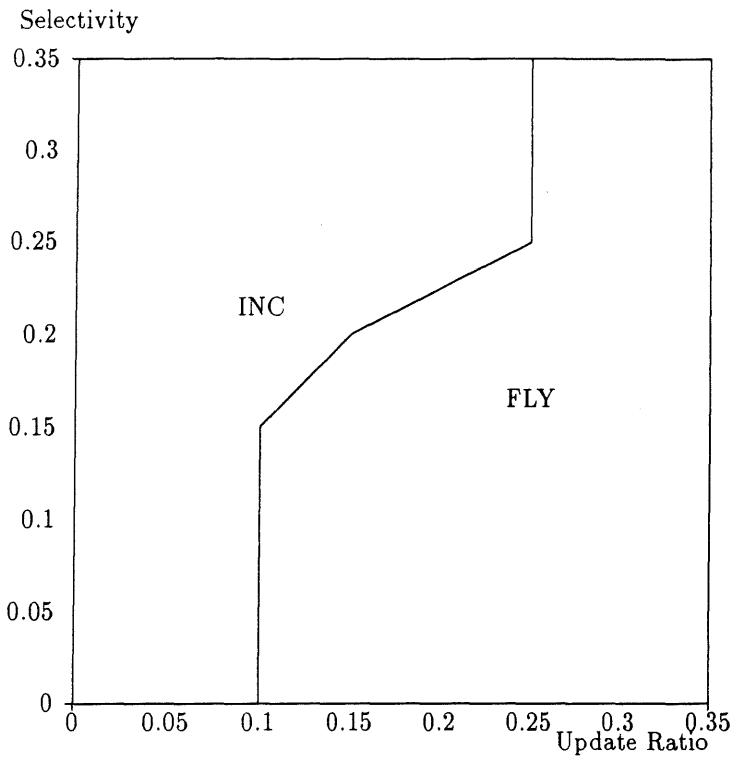


Figure 35

INC vs FLY with Selectivity and Update Ratio

CHAPTER 5
Recursive View Updates

1. Introduction

In Chapter 3, we have shown that there are two kinds of updates: base updates and view updates. The details of base updates were also discussed. Furthermore, in Chapter 4, we demonstrated that our incremental method performs better than the totally materialized and on-the-fly methods for certain cases. It is very clear that maintaining materialized views (incrementally) is justifiable.

In his controversial paper [CODD85], Codd defined twelve rules that test whether a database system is truly relational. The sixth rule is the so-called *view updating rule*. It states that all views that are theoretically updatable must be updatable by the system, i.e. the update effects should be automatically propagated to the supportive base tables. Thus, if views are materialized, users should be able to treat the view tables as if they were base tables and should be able to update them directly. We believe that view updates are important in any intelligent database systems for the following reasons:

(1) Similar to their base table counterparts, view tables should be updatable as users' knowledge of the views is changed. For example, let us consider the *ancestor* view again. If a view table *ANCESTOR* is retained, and if the user ascertains that the relation *ancestor(bill, john)* is no longer true, the tuple should be deleted from the *ANCESTOR* table. Similarly, if the user wants to assert the fact *ancestor(bill, mary)*, the tuple should be inserted in the *ANCESTOR* table. This is especially important since most views hide

information from the users, e.g. the users would not know the underlying tables used to derive the view. Without a view updating capability, the user will not know how to assert facts that can help deriving the desired view tuple.

(2) Interestingly, the view update mechanism can also be used to answer "what-if" questions. That is, it can provide an explanation mechanism that is highly desirable in any intelligent systems. Again, consider the same *ancestor* example. A curious user may want to find out what would happen if, for example, *ancestor(bill, mary)* were not true. By deleting this tuple in the *ANCESTOR* table, it may trigger deletions in the *PARENT* table and/or even further deletions in the *ANCESTOR* table. These deletions need not take effect immediately in the database. Rather, they can be retained only temporarily and used to explain that, if *bill* were not the ancestor of *mary*, then for instance, *john* would not be the parent of *mary*, etc.

In this chapter, we shall address the problem of updating views and shall also propose a recursive view update method that is complete, and also practical for integrating with relational database systems.

2. Issues of View Updates

There are essentially two major issues in view updates: first, why is it necessary to propagate view updates and second, why are view updates inherently ambiguous, and, finally, what would happen if the view is recursively defined?

2.1. Need to Propagate View Updates

If a view gets updated, its effect should be propagated to its supportive tables. First, let us examine why it is necessary to propagate view updates by considering the example of the views *parent* and *coach* that can be defined as follows:

(1) parent(X,Y):- father(X,Y).

(2) parent(X,Y):- mother(X,Y).
(3) coach(X,Y):- father(X,Y).

Let's assume that *father(bill, john)* is currently not in the database and is to be added while the tuple *parent(mary,susan)* is to be removed from the database. As most experienced Prolog programmer would do, the update {add parent(bill, john)} may be satisfied by simply adding a new fact *parent(bill, john)* to the database and the update {delete parent(mary, susan)} may be satisfied by adding the fact ¬ *parent(mary, susan)* to the database. Hence, the database become:

(1) parent(X,Y):- father(X,Y).
(2) parent(X,Y):- mother(X,Y).
(3) coach(X,Y):- father(X,Y).
(4) parent(bill, john).
(5) ¬ parent(mary, susan).

However, this approach is not desirable. First, this implies that queries to the relation *coach* will not see the effects of the updates to the relation *parent* since the update to the view does not propagate down to its underlying supportive base table. For example, assume that the database currently contains the facts *parent(bill john)* and *coach(bill, john)*. Later, if it is found that *bill* is not the parent of *john*, the tuple is deleted from the PARENT table. The action should be propagated to the FATHER table (and should fail in updating the MOTHER table because of the integrity rule built-in for the MOTHER table. More details on integrity will be discussed in the following section.)

Second, as discussed in section 1 of Chapter 1, a deductive database should contain two distinct components: the extensional database (EDB) or simply, the facts, which describe the world; and the intensional database (IDB), or the views (i.e. rules) that are used as reasoning mechanisms over facts to derive new facts. Since different users share the same database, we would like to change the facts but *not* the rules for deriving views by adding ground rule predicate as clauses (4) and (5) above. Similarly, different users should be able to share the updates defined through these views.

Therefore, it is necessary to propagate the view updates so that the state of the database can remain consistent through view updates from different users.

2.2. Semantic Ambiguity of View Updates

View update effects have to be propagated to their supportive tables, but the semantics of the propagation is ambiguous. The ambiguity can be categorized into two types, namely, *intra-rule* and *inter-rule* ambiguities.

Inter-rule ambiguity occurs when there are multiple view definitions. For example, the *parent* view in the previous section was defined either by rule (1) (i.e. *father(X,Y)*) or by rule (2) (i.e. *mother(X,Y)*). If the view update of *parent* has to be propagated, then which of these two definitions (or both) should be considered? That depends on the type of update. For instance, consider the view update request: {delete parent(a,b)}. In order to guarantee that the database can never derive the relation *parent(a,b)* again in the future, both rules (1) and (2) must be used to propagate the effect to delete the relevant base tuples so that it will *never* be able to derive the relation *parent(a,b)*. Therefore, for deletion requests, the update effect must propagate to each view definition and hence is *not* ambiguous. However, if the view update is an insertion such as { add parent(a,b)}, should we attempt to update only the FATHER table, or only the MOTHER table, or both FATHER and MOTHER tables?

We distinguish two situations: *weak propagation* and *strong propagation*. In a weak propagation, the view update effect is propagated to any one of its view definition as long as the propagation is successful. If the view has to be derived from scratch, it only requires one successful branch in the SLD-tree (refer to Chapter 2) to define the bindings of the view. However, since there are multiple view definitions, to decide on which definition should be used to propagate the effect becomes ambiguous. *Strong propagation*, on the other hand, refers to the situation where the view update effect is propagated to *all* view definitions. The justification is as follows: similar to the deletion, an insertion of a view tuple should be supported by as many paths as possible. If a view is defined by a set of rules, a strong propagation will attempt to propagate the view update into all rules. The rationale is that if the view tuple is believed to be true, it should be supported by all available supportive base tables. Therefore, in the future, if a certain branch of the SLD-tree gets pruned off due to the deletion of a certain base tuple, the view tuple is still supported by other alternate paths.

It may seem that the strong propagation would generate a lot of base updates. However, this is not necessarily true since each of these updates is still subject to the integrity constraints (ICs) specified by the database administrator (DBA). For instance, consider the view update request {add parent(bill, john)}. With strong propagation, both rules (1) and (2) are used to propagate the update effect. However, the integrity constraint for the MOTHER table will fail the update propagation since *bill* is not female. In general, it is believed that, with integrity constraints for each domain and view, strong propagation will not increase the database dramatically by inserting too many tuples.

Intra-rule ambiguity, on the other hand, refers to the following situation: after it has been decided to propagate a view update effect into one specific definition, it is ambiguous which of the underlying supportive base tables should be updated. For example, consider the following view definition:

$$v(X,Y,Z) : -p_1(X,A), p_2(A,B), p_3(B,Y,Z).$$

The view V is a many-to-many relation of the supportive base tables P_1, P_2 and P_3. To process a view update request such as {*add v(e,f,g)*}, the update effect has to propagated to the body of the view's definition. Should the effects be propagated to P_1, or P_2 or P_3 or any combinations of these three tables? Furthermore, the semantics of the update effects is not well understood if the view v is recursively defined, as, for example, the following:

$$v(X, Y, Z) : -p_1(X, A), p_2(A, B), v(B, Y, Z).$$

In this chapter, we provide recursive view updates, based on strong propagation, with proper semantics and a practical method to process them.

2.3. Effects of Recursive View Updates

If a view is recursively defined, updating the view has two effects: *anterior view update*, and *posterior view update*. Anterior view update implies that, in order to support the new database resulted from the recursive view update request, some underlying tables have to be updated so that the update request will be supported. *Posterior view update* refers to the situation that, if the update request is supported, the current database will be updated with these newly inserted or deleted view tuples resulted from the update request. Posterior view update are just the same as *base updates*, in which the view table is treated as if it was a base table. To illustrate these two effects, let's consider the ancestor view in Figure 9. Assume that the user wants to update the view table *ANCESTOR*. The anterior view update is to propagate the update effects to the underlying tables as discussed in section 2.1 and 2.2, in order to support the update request. If these supports can be generated (as we shall see later on that not every view update is satisfiable), the effect of updating *ANCESTOR* may affect other *ANCESTOR* tuples. For instance, if the update is to delete a tuple in *ANCESTOR* and the tuple is a primary lemma to some other tuples, these tuples should also be deleted. Similarly, if the update is to insert a tuple in *ANCESTOR*, the new tuple may trigger derivation of some other tuples to be included int *ANCESTOR*. This posterior view update is identical to the base update discussed in Chapter 3. They are also subject to the screening test. On the other hand, an anterior view update is to propagate the update effects so that the update request can be supported. To put it in a different way, posterior view update works its way from the right hand side of the rule up to the head while anterior view update propagates the view update effect from the head of the rule to its body. These two updates have to be done separately: anterior view update first, followed by the posterior view update.

In this chapter, we address the anterior view update problem. That is, how to propagate the update effects from the head to the body of its rules.

2.4. Related Work

There are two distinct ways to update views: (1) the heuristic approach, and (2) the theoretic approach. The former is typified by [KEL85] and [MW88] while the latter can be found in [FUV83] and [RN88].

The theoretic approach argues that, since the mapping from a view to the underlying base tables is not unique (ambiguous), we should generate *all* possible alternatives that entail the update. Accordingly, they proposed a method that defines an update as a mapping from the old "theory" to a disjunction of all possible new "theories".

For example, consider the following tail recursive view v:

(1) v(X) :- p(X), v(X).
(2) v(X) :- q(X).

In this approach [RN88], the update request *add v(a)* is translated into $(p(a) \wedge v(a)) \vee q(a)$; and the update request *delete v(b)* is translated into $(\neg\, p(b) \vee \neg\, v(b)) \wedge \neg\, q(b)$. Therefore, the database is transformed from the "old" theory into the following "new" theory which, unfortunately, is non-Horn:

(1) v(X) :- p(X), v(X).
(2) v(X) :- q(X).
(3) $(p(a) \wedge v(a)) \vee q(a)$
(4) $(\neg p(b) \vee \neg v(b)) \wedge \neg q(b)$

While this approach does present a semantically consistent picture of view updates, it suffers the following drawbacks:

(1) In [FUV83], the authors noted that the approach is intractable. Even in [RN88], the improved method still suffers from the same problem. The new update theories generated by this approach (generally, non-Horn) are exponential to the number of definitions of the view. Hence, future query processing time deteriorates (there are more rules to process). Furthermore, a view deletion will generate a set of new update theories that contain mostly negation, which makes a database work harder while processing a query.

(2) Since the update theories are mostly non-Horn, and they can only be resolved with the special type of resolution method, they will be difficult to integrate with any relational system.

(3) If every update generates a new set of "rules", then it is necessary to re-generate the compiled query program as discussed in Chapter 2. This defeats the original motivation to compile queries (views) so that once the compilation is done at view definition time, we don't need to reference the inference engine again for future query or update operations

In the heuristic approach, on the other hand, it is argued that a database administrator (DBA) should know what to update (by user requirements analysis, heuristics, experience, etc, similar to setting up integrity constraints.) Hence it is desirable for the DBA to designate the relations that will be updated when the view is updated. For example, in [KEL85], the author proposes a set of five criteria that should be satisfied by view updating algorithms. Through a structured interactive dialogue with the DBA, he designs different algorithms to choose a view translator at view definition time. In [MW88], a similar approach is proposed, except that it is treated in a more formal manner. In this approach, *view translators* are defined based on dynamic logic programming techniques. Then each view update request is processed by its relevant view translator that acts like a procedure to implement the update request. The procedural semantics of update translators can be easily incorporated in a relational system.

2.5. Our Contributions

As noted in the introduction, our primary motivation is to improve recursive query processing time by retaining materialized view tables. The heuristic approach is more appealing to us in this respect, since the update translators can be easily incorporated into a relational system. However, the approach proposed by [MW88] suffers from the following drawbacks:

(1) no multiple rule definitions are allowed.
(2) no recursive relations are allowed (views in their system are restricted to be non-recursive.)
(3) all variables have to appear in the head of a rule.

In this chapter, we extend the heuristic approach such that it overcomes these drawbacks. In our approach, we use strong propagation to counter inter-rule ambiguity. This is necessary since recursive views must have, by definition, at least two rules: one to define the view recursively and the other as the *exit* clause that does not contain any recursion. Therefore, to implement any recursive view update, it is necessary to deal with multiple view definitions.

3. The Heuristic Approach Based on Dynamic Logic Programming

There are four situations that change the contents of a view table: (1) newly found view tuples resulting from our query process; (2) an indirect effect from inserting or deleting a base table, which triggers updates on the view table; the so-called base updates; (3) a direct ADD request to the view table from the user, and finally, (4) a direct DELETE request to the view table by the user. The latter two situations are essentially the view update problems. As discussed in section 2.3, for each of these two update requests, it triggers an anterior view update and, if this is successful, it further triggers a posterior view update. Since posterior view update is identical to base update, we shall concentrate on anterior view update in this chapter.

Given a view update request from the user, it will be desirable to update the "appropriate" underlying tables (the so-called *supportive* tables) so that the requested view tuple is or is not supported. However, since a view is a many-to-one relationship, it is inherently ambiguous to decide which of the supportive tables (or all of them) are to be updated. As already mentioned in section 2.3, one way to solve the ambiguity problem is to let the DBA decide which of the supportive tables should be updated. We believe that this is very reasonable since the designation of supportive tables to be updated is very similar to the setting up of integrity constraints, which can be done only thorough user requirements analysis, heuristics from experience and understanding of the application domain. We abandon the informal approach by [KEL85] but adopt the approach of [MW88]. Since this approach is based on the dynamic logic programming technique, we shall, in the following section, describe how dynamic logic programming works.

3.1. The Basics of Dynamic Logic Programming (DLP)

Dynamic logic [HA79] is logic for reasoning about programs that test and change an environment. If update requests are to be translated into some form of procedures, it has been shown [MW88] that it is possible to construct a dynamic logic of update programs. That results in the so-called dynamic logic programming (DLP). In DLP, a database D is a triple (EDB, IDB, U) where EDB and IDB are the extensional and intensional databases, analogous to those of the deductive database, which we have been discussing in the previous chapters. U is a set of update rules that are called *update translators*. Update translators define the semantics and also the *procedure* to propagate the update effects.

In dynamic logic programming, there are three types of literals (relations), namely, base literals, view (virtual) literals, and dynamic literals. The base and view literals are identical to those we have been using so far. A dynamic literal is to signal an insertion into or a deletion from some tables. It is enclosed in angle brackets and has a plus or minus sign to indicate the action to be taken. For example, consider the following query:

$$: -father(bill, john), \langle -coach(bill, john)\rangle.$$

Let's assume that *father* is a base relation and *coach* is a view, as in the example of section 2.1. The meaning of the query is as follows: if *bill* is the father of *john*, then delete the tuple *(bill, john)* in the view *coach.* The literal $\langle -coach(bill, john)\rangle$ is called a *dynamic literal* since it will change the state of the database. In this example, the dynamic literal $\langle -coach(bill, john)\rangle$ has a dual meaning. First, it means to physically delete the tuple from the view table if the view *coach* is materialized. Second, all supportive base tables that can regenerate the tuple *coach(bill, john)* should be updated such that it will not be possible to derive the tuple in the future. Similarly, if the dynamic literal is $\langle +coach(bill, john)\rangle$, the update procedure will first insert the tuple *(bill, john)* in the view table if it is not already there; then it will propagate the effects to the relevant underlying tables that support the newly added view. The reason for propagating the update effects was discussed in section 2.1. If the dynamic literal is to insert or delete a base tuple, then it is nothing more than a base update that we have discussed in Chapters 3 and 4. If the dynamic literal is to insert or delete a view tuple, it is referred to as a *view update.*

For example, consider the same database of parent and coach relations as in section 2.1. A complete dynamic database may look like the following:

(1) parent(X,Y):- father(X,Y).

(2) parent(X,Y):- mother(X,Y).
(3) coach(X,Y):- father(X,Y).
(4) ⟨ +parent(X,Y)⟩:-father(X,Y).
(5) ⟨ +parent(X,Y)⟩:-mother(X,Y).
(6) ⟨ -parent(X,Y)⟩:-⟨ - father(X,Y)⟩.
(7) ⟨ -parent(X,Y)⟩:-⟨ - mother(X,Y)⟩.
(8) ⟨ +coach(X,Y) ⟩ :- ⟨ +father(X,Y)⟩.
(9) ⟨ -coach(X,Y) ⟩ :- ⟨ -father(X,Y)⟩.

The first three rules are the original view definitions. Rules (4) to (9) are the update rules that (based on the DBA's point of view) define the semantics of the updates. For example, rules (4) and (5) state that if one wants to assert the relation of *parent(X, Y)*, either the relation *father(X, Y)* or *mother(X, Y)* has to be true. Similarly, if a tuple of *parent(X, Y)* is to be deleted, then all supportive tuples have to be deleted so that the tuple of *parent(X, Y)* can no longer be derived (rules 6 and 7.) Another way to interpret the update request is as follows: if the head of the update rule is an insertion, it can be interpreted as "update the database such that this fact is satisfiable". If the fact can be satisfied with many alternatives, let all those alternatives be known (strong propagation). If the head of the update rule is a deletion, then "update the database such that this fact becomes unsatisfiable by all means". The update rules that can guarantee these are *semantically correct.* (They are, in this case, the weak correctness conditions of the update rules.) Furthermore, if there are updates to insert tuples already in the view, or to delete tuples not in the view, and if the update rules *do not* change the database, the update rules are said to be *semantically acceptable* [BS81].

Specifying the update rules is the job of the DBA who, based on his experience and user requirements, generates these rules. Unfortunately, not every update rule specified by the DBA is correct and acceptable. For example, the update rule

$$\langle +coach(X,Y)\rangle : -\langle +friend(X,Y)\rangle.$$

(1)	$\lambda(q_b(\bar{t}))$	$= q_b(\bar{t})$ or $\langle +q_b(\bar{t})\rangle(true)$
(2)	$\lambda(q_v(\bar{t}))$	$= q_v(\bar{t})$ or $\langle +q_v(\bar{t})\rangle(true)$
(3)	$\lambda(q_r(\bar{t}))$	$= q_r(\bar{t})$

Figure 36

The Add Translator

where *friend* is an existing base relation. This update rule is neither correct nor acceptable. It is not correct because inserting a new tuple to the relation *friend* cannot derive the *coach* tuple since *coach* is defined only by the *father* relation (i.e. rule (3)). Similarly, it is not acceptable because, even if the relation of *coach(X, Y)* were already in the view table, this update rule would still insert a tuple in the relation *friend*. This is a contrived example to illustrate the point, since updating the *coach* relation should have nothing to do with the *friend* relation; *coach* is not defined on *friend*. As a rule of thumb, to guarantee the correctness and acceptability of the update translator, the entire body of the view definition is included in the translator.

In the following sections, we shall describe the add and delete translators separately. We follow the same terminology as in [MW88] to define these translators.

3.2. The ADD Translator

In this section, we summarize the semantics of the Add Translator defined by [MW88].

A view definition always has the following form:

$$v(\bar{u}) : -q_1(\bar{t}_1), \ldots, q_p(\bar{t}_p)$$

where the view v has a vector of variable names $\bar{u}$ and the q_is literals have their own vectors of variable names. In [MW88], the q_i is further distinguished as either a *base* literal, a *rule* literal, or a *view* literal.

An update rule for insertion based on this view can be expressed as:

$$\langle +v(\bar{u})\rangle : -\lambda\{q_1(\bar{t}_1), ..., q_p(\bar{t}_p)\}$$

where λ is a mapping from the conjunction of $q_1(\bar{t}_1), ..., q_p(\bar{t}_p)$ to a corresponding dynamic conjunction. The mapping λ is distributive and has different transformations for different types of literals. Let q_b, q_v, and q_r represent the base, the view, and the rule literals, respectively. Manchanda et al [MW88] define the semantics of the Add translator as shown in Figure 36.

Rule (1) states that if the add translator is applied to a base literal, it succeeds if the base literal is already satisfiable (i.e. it already exists in the base table). Otherwise, the tuple is added to the base table. If the insertion into the base table is successful, then the entire update request becomes successful (i.e. $\langle +q_b(\bar{t})\rangle(true)$). Similarly, rule (2) describes the add translator applied to a view literal. In this case, it will first check if that view tuple already exists. If not, it will try to add it. Finally, rule (3) states that when the add translator is applied to a *rule* literal, it simply checks if the rule literal is satisfiable (i.e. can be derived when queried), without inserting any tuple to make it satisfiable. Note that the view literals, as defined in [MW88], are basically the non-recursive view, while the so-called *rule* literal is actually a recursive view.

To illustrate the Add translator, consider the following rules:

(1) *parent(X, Y) :- father(X, Y).*
(2) *parent(X, Y) :- mother(X, Y).*
(3) *grandparent(X, Y) :- parent(X, A), parent(A, Y).*
(4) *ancestor(X, Y) :- parent(X, Y).*
(5) *ancestor(X, Y) :- parent(X, A), ancestor(A, Y).*

where *father* and *mother* are *base* literals; *parent* and *grandparent* are *view* literals and *ancestor* is a *rule* literal. To derive the Add translator for *parent*, rule (2) of Figure 36 is applied. The resulting translator has the form: $\lambda(parent(X, Y)) =$

$parent(X,Y)$ or $\langle +parent(X,Y)\rangle(true)$. Suppose the transformation is applied to *parent(bill, john)*. According to this rule, the Add translator will first examine if the tuple *parent(bill, john)* already exists in the view. If it does, then the insertion of the tuple succeeds. Otherwise, the Add translator will try to evaluate $\langle +parent(bill,john)\rangle(true)$, which will try to insert the tuple *parent(bill, john)* into the view. Note that, for the purpose of brevity, the λ mapping is simplified as $\lambda(parent(X,Y)) = \langle +parent(X,Y)\rangle$. It has the identical meaning: if *parent(X, Y)* exists, then succeed; otherwise, try to insert the tuple of *parent(X, Y)*. In order to support this newly added tuple, the update effect has to propagate to its supporting tables. The DBA must designate the tables affected when the view is defined. For example, the DBA may specify the following two update rules:

(1) $\langle +parent(X,Y)\rangle : -\langle +father(X,Y)\rangle$.

(2) $\langle +parent(X,Y)\rangle : -\langle +mother(X,Y)\rangle$.

The effect of $\langle +parent(bill,john)\rangle$ is propagated to the body of both rules, i.e. $\langle +father(bill,john)\rangle$ and $\langle +mother(bill,john)\rangle$. In turn, the Add translator will be applied to these two dynamic literals. The request of $\langle +parent(bill,john)\rangle$ becomes successful, if any one of these actions succeeds. If neither one of these two actions succeeds, the request is failed. Therefore a view update request is not always successful. Note also that adding a tuple to *father* or to *mother* is still subject to the update integrity constraints.

The original transformations, as proposed in [MW88], are restricted to non-recursive views such as the *parent* view. For recursive views, such as the *ancestor* view, the transformation rule (3) in Figure 36 simply checks if the tuple is currently supported by the database. If so, then the update becomes successful; otherwise, it fails, i.e. no insertion is performed. For the above example, $\langle +ancestor(bill,john)\rangle$ would simply be converted to a query to check if *ancestor(bill, john)* can be derived or not. This is a major drawback of such a translator. To correct the problem, the update effect on the recursive view should be propagated to the body of the original rule, similar to the non-recursive ones. For example, the DBA might specify the following Add translator for the *ancestor* view:

(1) $\langle +ancestor(X,Y)\rangle$:- $\langle +parent(X,Y).\rangle$

(2) $\langle +ancestor(X,Y)\rangle$:- $\langle +parent(X,A)\rangle$, *ancestor(A,Y).*

An update request such as ⟨ *ancestor(bill, john)* ⟩ will trigger both of these rules. From rule (1), the effect is propagated to $\langle +parent(bill, john)\rangle$. From rule (2), the update is translated to the body ⟨ *+parent(bill, A)* ⟩, *ancestor(A,john).* Syntactically, the translator says to insert the tuple *parent(bill, A)* to the PARENT table where A is a free variable. Then the next step is to query on *ancestor(A, john)* in the current database.

Note that such recursive view updates are not possible in the original proposal of [MW88] simply because they could not solve the termination problem that exists in any recursive query process. It is different in our approach. Our method is based on the lemma resolution and will terminate for any recursive query. Therefore, propagating the update effects into the body of a recursive view becomes possible.

Note further that the order of the two literals in the body of rule (2) above is significant. In the original rule, the AND connective between *parent(X,A)* and *ancestor(A, Y)* is commutative. However, in the update translator, it is not. In this example, it means "insert tuples *parent(bill, A)* first then, after the insertion, see if *ancestor(A, john)* can be satisfied or not"; if it can be satisfied, then the constant bindings of A will be combined with *bill* and be inserted in the PARENT table. Otherwise, the insertion on *ancestor(bill, john)* is failed. Note also that the free variable A in this case is short-lived. It is substantiated after the query *ancestor(A, bill)* is satisfied.

In this chapter, we extend the original Add translator from [MW88] to cover recursive view updates. Our system has only two types of rules in the Add translator:

(1) $\mu(q_b(\bar{t}))$ $= true$ or $\langle -q_b(\bar{t})\rangle(true)$

(2) $\mu(q_v(\bar{t}))$ $= true$ or $\langle -q_v(\bar{t})\rangle(true)$

(3) $\mu(q_r(\bar{t}))$ $= true$

Figure 37

The Delete Translator

(1) $\lambda(q_b(\bar{t}))$ $= q_b(\bar{t})$ or $\langle +q_b(\bar{t})\rangle(true)$

(2) $\lambda(q_v(\bar{t}))$ $= q_v(\bar{t})$ or $\langle +q_v(\bar{t})\rangle(true)$

The first rule is applied to base literals while the second rule is for view definitions, both recursive and non-recursive ones. The main problem with this extension is the need to guarantee completeness. Fortunately, this can be achieved using lemma resolution for a restricted class of update rules, the *safe* update rules. Details will be discussed in section 4.

3.3. The Delete Translator

The Delete translator is defined in [MW88] as shown in Figure 37. Similar to its Add translator counterpart, the μ is the mapping of the Delete translator and q_b, q_v, and q_r are base, view, and rule literals, identical to those of the Add translator. For instance, the first rule in Figure 37 means that the mapping of the deletion request for a base dynamic literal is to first check if the base tuple currently exists. If it does not, the deletion request is complete. If the base tuple currently exists, then try to delete it from the base table. The mapping for the view is similar to that of the base literal. The mapping of the rule literal simply checks if the rule tuple is not supported (i.e. it cannot be derived.) Similarly, the original translator is restricted to non-recursive views. In this chapter, we extend this to cover recursive views by defining the Delete translator as follows:

(1) $\mu(q_b(\bar{t}))$ $= true$ or $\langle -q_b(\bar{t})\rangle(true)$

(2) $\mu(q_v(\bar{t}))$ $= true$ or $\langle -q_v(\bar{t})\rangle(true)$

The extended delete translator now propagates the deletion effects to recursive rules as well. For instance, the DBA may specify the following deletion operations for the *parent*, *grandparent*, and *ancestor* views:

(1) $\langle -parent(X,Y)\rangle : -\langle -father(X,Y)\rangle$.

(2) $\langle -parent(X,Y)\rangle : -\langle -mother(X,Y)\rangle$.

(3) $\langle -grandparent(X,Y)\rangle : -\langle -parent(X,A)\rangle$, *parent(A, Y).*

(4) $\langle -ancestor(X,Y)\rangle$:- $\langle -parent(X,Y).\rangle$

(5) $\langle -ancestor(X,Y)\rangle$:- $\langle -parent(X,A)\rangle$, *ancestor(A,Y).*

For brevity purpose, the μ mapping is also simplified as $\mu(q(\bar{t})) = \langle -q(\bar{t})\rangle$, with the same meaning: if $q(\bar{t})$ does not exist, the deletion operation succeeds. If $q(\bar{t})$ exists, then try to delete it. In this example, the DBA specifies that if a tuple in the PARENT table is to be deleted, supportive tuples in both FATHER and MOTHER tables must be deleted so that the PARENT tuple will never be derived in the future (strong propagation requires all possible tables to be updated.) Similarly, to delete a tuple *(bill, john)* from the GRANDPARENT table, first deletion of tuples *(bill, A)* in PARENT is attempted, then *parent(A,john)* is queried. If it is satisfiable, the bindings of A will be combined with *bill* and deleted from the PARENT table. To delete a tuple from the ANCESTOR table, the supportive tuples in PARENT must be deleted. Both deletion rules (rules (4) and (5)) must be activated to eliminate all supportive tuples.

4. Dynamic SLD Resolution (DSLD)

In [MW88], the semantics of view updates are defined. However, the update process is treated separately from the query process. Because of this separation, they fail to define the proper semantics for recursive view update. In the previous section, we extended the translators to recursive view update. In this section, we describe a modified resolution method, the *dynamic SLD Resolution (DSLD)* by combining the dynamic logic programming with the lemma resolution to describe the update semantics. We demonstrate how this scheme works. We further show that the DSLD is *incomplete* for an arbitrary function-free Horn database.

The Add and Delete translators described in the previous section can be incorporated into the SLD (with lemmas) resolution. The method based on both SLD (with lemmas) and the update translators is called *dynamic SLD resolution (DSLD).* The resulting tree is called Dynamic SLD resolution tree. The dynamic SLD resolution is similar to the SLD resolution with lemmas except that when the goal is a dynamic literal, the relevant update rule is called, instead of the regular rule, to resolve the goal.

Assuming that all update requests are ground literals (i.e. no free variables) and the goal is a dynamic literal, the procedure of the DSLD resolution, similar to that of the lemma resolution from Chapter 2, can be described as follows:

(1) find relevant update rule for the update request in a top-down manner. If such rule is found, then expand the update rule.

(2) all possible paths of the search tree are expanded until one of the following situations occurs: (a) all literals are resolved, i.e. the update request succeeds; (b) the path fails, i.e. update request cannot be satisfied; (c) a dynamic literal that is identical to a previously resolved dynamic literal is encountered; (d) a dynamic recursive literal is encountered; (e) a regular recursive literal that is an instance of a previously solved goal (i.e. lemma resolution) is encountered.

When any of the cases (b) through (e) occurs, the node is suspended. Then the process backtracks and tries other available rules.

When all available branches are suspended, we have finished processing the current stage.

(3) If the resulting resolution tree from (2) contains any suspended nodes, use the new database and view (lemma) tables to expand these nodes in the same manner as in steps (1) and (2).

(4) The process is terminated when the current stage does not insert or delete any more tuples.

This resolution method provides a method of inserting facts to and deleting facts from the database. As we have discussed before, the update process may not always succeed. Therefore, it would be unwise to update the tables immediately while the resolution is still in progress. One way to handle this is to maintain two *intentions lists*, as suggested in [MW88]: the add set (AddSet) and the delete set (DeleteSet). These two sets are initially empty. As the resolution proceeds, tuples are inserted into these two sets dynamically. After the update request is performed successfully, these two sets are inserted or deleted from the database. To be specific, the unification process can be described as follows:

(1) If $q_1, q_2 ..., q_n$ is the current resolvent and q_1 is a base literal, then search AddSet or the EDB for a base fact that unifies q_1 with substitution s. If this can be done, then the new resolvent becomes $[q_2, ..., q_n]s$; otherwise, fail the node.

(2) If q_1 in (1) is a view literal, use lemma resolution to search the IDB for a rule that unifies q_1 with substitution s. Then the new resolvent becomes $[q_2, ..., q_n]s$; otherwise, fail the node.

(3) If the resolvent is $\langle +q_1 \rangle, q_2, ..., q_n$ where q_1 is a dynamic base literal, then first search DeleteSet to check if it contains q_1. Remove it from DeleteSet if it exists. Next, check if it exists in the EDB. If not, add it to the AddSet. The new resolvent becomes $q_2, ..., q_n$.

(4) If the resolvent is $\langle -q_1 \rangle, q_2, ..., q_n$ where q_1 is a dynamic base literal, then first search AddSet to check if it contains q_1. Remove it from AddSet if it

(1) *parent(X, Y) :- father(X, Y).*

(2) *parent(X, Y) :- mother(X, Y).*

(3) *grandparent(X, Y) :- parent(X, A), parent(A, Y).*

(4) *ancestor(X, Y) :- parent(X, Y).*

(5) *ancestor(X, Y) :- parent(X, A), ancestor(A, Y).*

(6) $\langle +parent(X,Y)\rangle : -\langle +father(X,Y)\rangle.$

(7) $\langle +parent(X,Y)\rangle : -\langle +mother(X,Y)\rangle.$

(8) $\langle +grandparent(X,Y)\rangle : -\langle +parent(X,A)\rangle,$ *parent(A, Y).*

(9) $\langle +ancestor(X,Y)\rangle$:- $\langle +parent(X,Y).\rangle$

(10) $\langle +ancestor(X,Y)\rangle$:- $\langle +parent(X,A)\rangle,$ *ancestor(A, Y).*

(11) $\langle -parent(X,Y)\rangle : -\langle -father(X,Y)\rangle.$

(12) $\langle -parent(X,Y)\rangle : -\langle -mother(X,Y)\rangle.$

(13) $\langle -grandparent(X,Y)\rangle : -\langle -parent(X,A)\rangle,$ *parent(A, Y).*

(14) $\langle -ancestor(X,Y)\rangle$:- $\langle -parent(X,Y).\rangle$

(15) $\langle -ancestor(X,Y)\rangle$:- $\langle -parent(X,A)\rangle,$ *ancestor(A, Y).*

Figure 38

An Example of Query and Update Rules

exists. Next, check if it exists in the EDB. If not, add it to the DeleteSet. If q_1 is neither in AddSet nor in the EDB, then ignore it since it is not derivable from the database anyway. The new resolvent becomes $q_2, \dots, q_n$.

After the update process is done, the AddSet and DeleteSet are used to update the actual EDB by applying the union and difference operators. These AddSet and DeleteSet can be used as *intentions lists* to answer the *what-if* questions without actually updating the EDB.

To illustrate the DSLD resolution, consider the rules with their update translators as shown in Figure 38 and also the current database contents as in Figure 39. The resolution with the dynamic literals works as follows: in order to satisfy a dynamic literal (i.e., to update the table), we propagate the update effects according to the translator. If there are regular literals, the SLD with lemma is called to

FATHER

john	*bill*
john	*mary*
bill	*david*

MOTHER

susan	*david*

PARENT

john	*bill*
john	*mary*
bill	*david*
susan	*david*

GRANDPARENT

john	*david*

ANCESTOR

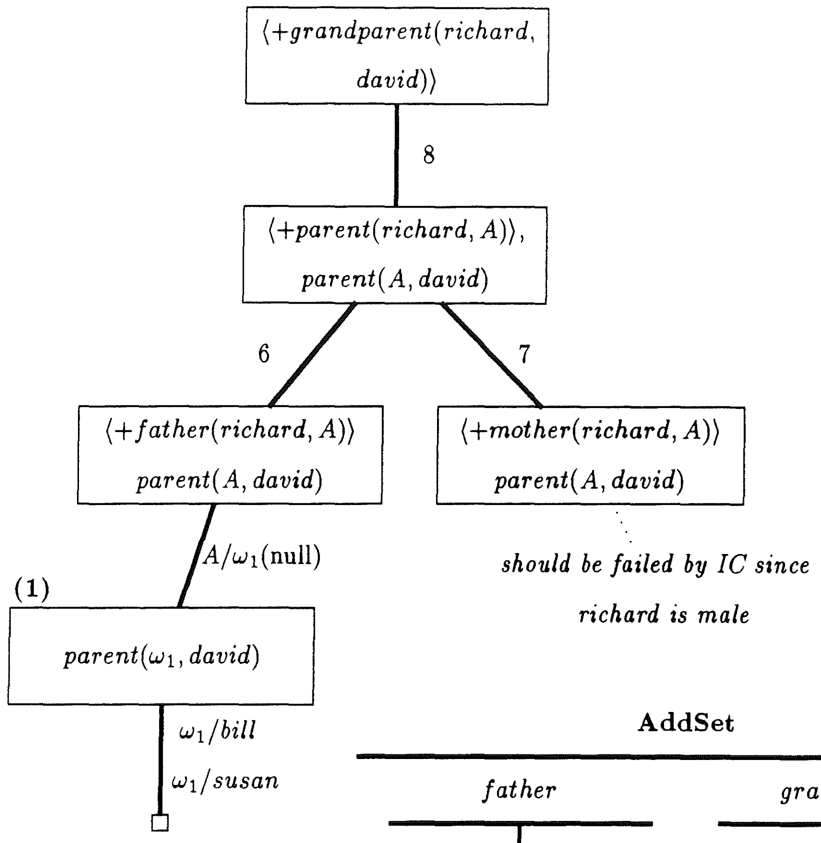
john	*bill*
john	*mary*
john	*david*
bill	*david*
susan	*david*

Figure 39

Tables for the Family Example in Figure 38

resolve these literals until it derives an empty clause. Let us look at two examples, one with the Add translator and the other one with the Delete translator.

Consider the update request $\langle +grandparent(richard, david)\rangle$. The DSLD resolution tree is depicted in Figure 40. For instance, the update request is expanded by using rule 8. The next goal becomes $\langle +parent(richard, A)\rangle$ which is, in turn expanded with rule 6. Note that the unbound variable A is unified to the body of the update rule identical to the usual resolution process. The next goal becomes $\langle father(richard, A)\rangle$ which is a base dynamic literal. According to the Add translator, this goal is true (satisfiable) if there exists an instance *father(richard, A)*; otherwise, if the tuple *father(richard, A)* can be asserted, then the goal can also be asserted true. This dynamic base literal contains an unbound variable A. This variable can be temporarily assigned a distinct *null* value such as ω_1. Then the literal *parent*(ω_1, *david)* becomes the next goal. Similarly, to satisfy this goal, ω_1 can be bound to the constants *bill* and *susan* as shown in Figure 40. The leftmost


⟨+grandparent(richard, david)⟩
8
⟨+parent(richard, A)⟩, parent(A, david)
6
7
⟨+father(richard, A)⟩ parent(A, david)
⟨+mother(richard, A)⟩ parent(A, david)
A/ω1(null)
(1)
should be failed by IC since richard is male
parent(ω1, david)
ω1/bill
ω1/susan

AddSet

father

richard	*bill*
richard	*susan*

grandfather

richard	*david*

parent

richard	*bill*
richard	*susan*

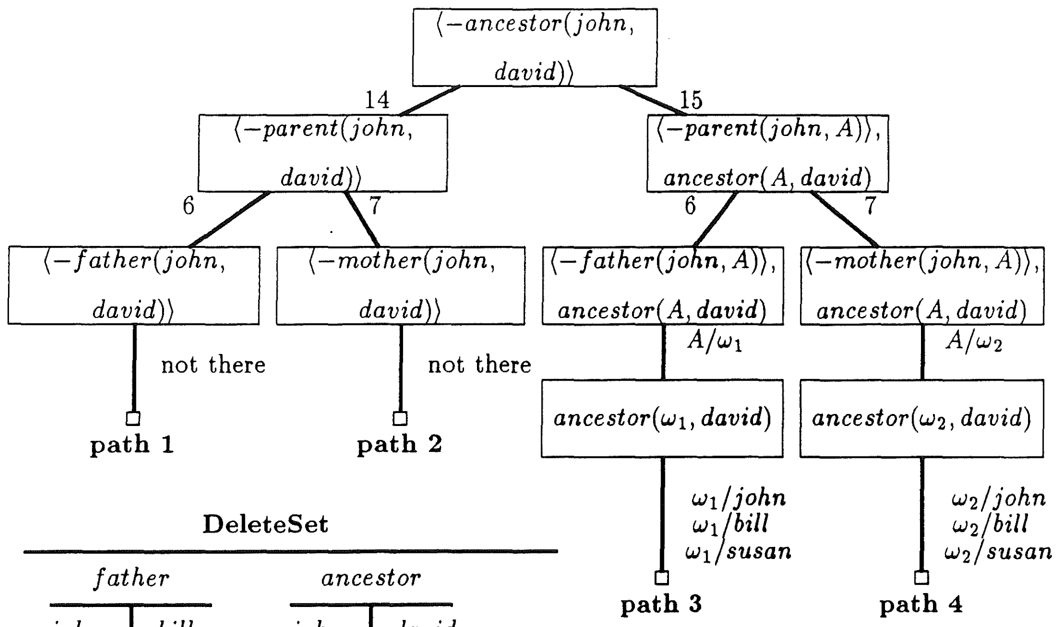
Figure 40

Example of The DSLD Resolution for An Add Translator

branch thus terminates and returns a set of bindings, in this case, *father(richard, bill)*, *father(richard, susan)*, *parent(richard, bill)*, and *parent(richard, susan)*. This means that, if we like to insert the tuple *grandfather(richard,david)*, we need to insert these new tuples. After the insertion, the query *grandfather(richard,david)* is supported and will derive an empty clause in a resolution. Should the goal *parent*(ω_1, *david)* fail, the process backtracks and tries other alternatives. The backtracking will undo those tuples in the AddSet before branching out to other paths. Therefore, the actual inclusion of the dynamic literals in the AddSet takes place only for the successful branches.

The rightmost branch will be processed in a similar fashion. Note that if there is *no* other information such as integrity constraints, then the translator will add facts that make *bill* and *susan* siblings! Similarly, *richard* might become the mother of someone (from the rightmost branch). This shows that, in any update situation, integrity constraints are extremely important to guarantee that no nonsense data ever enters the database. If the integrity constraint specifies that no parents can be siblings then the facts *parent(richard, bill)* and *parent(richard,susan)* become mutually exclusive. The present case implies that the integrity constraints are not sufficient enough and the process becomes ambiguous. One way to handle the ambiguity is to interact with the user for additional information. Another way is to generate different branches for each of these exclusive bindings. For example, at the node marked (1) in Figure 40, the resolution could separate into two branches: one for the case where *richard* is the parent of *bill*, the other one for the case where *richard* is the parent of *susan*. Thus, the AddSet becomes { *father(richard, bill), parent(richard, susan)*} OR { *father(richard, susan), parent(richard, susan)* }. These two sets are the intent lists. Eventually, the user has to select one over the other. However, if cases such as this keep occurring, it implies that the integrity constraints have to be strengthened to preclude the ambiguity. In this chapter, we assume that integrity constraints are complete and sufficient enough to eliminate this type of ambiguity.

As for the delete translator, consider the update request to delete a view tuple in the ancestor view, e.g. $\langle -ancestor(john, david) \rangle$. The DSLD resolution tree is



DeleteSet

father

john	*bill*
john	*susan*
john	*john*
john	*david*

ancestor

john	*david*

mother

john	*bill*
john	*susan*
john	*john*
john	*david*

Figure 41

Example of The DSLD Resolution for A Delete Translator

depicted in Figure 41. In this case, the delete rules were called to propagate the update effects. Paths 1 and 2 terminate because the tuples *father(john, david)* and *mother(john, david)* are not in the base tables. Paths 3 and 4 also terminate with the short-lived null value ω bound to the constants *john, bill,* and *susan.* The final DeleteSet contains the three tables with their tuples as shown. Note that many of these tuples to be deleted from the original tables are not even there. Furthermore, if the database has a more comprehensive integrity constraint system, many of these tuples won't even enter the DeleteSet. For example, the MOTHER table would not be affected and the tuple *father(john, john)* would not be generated.

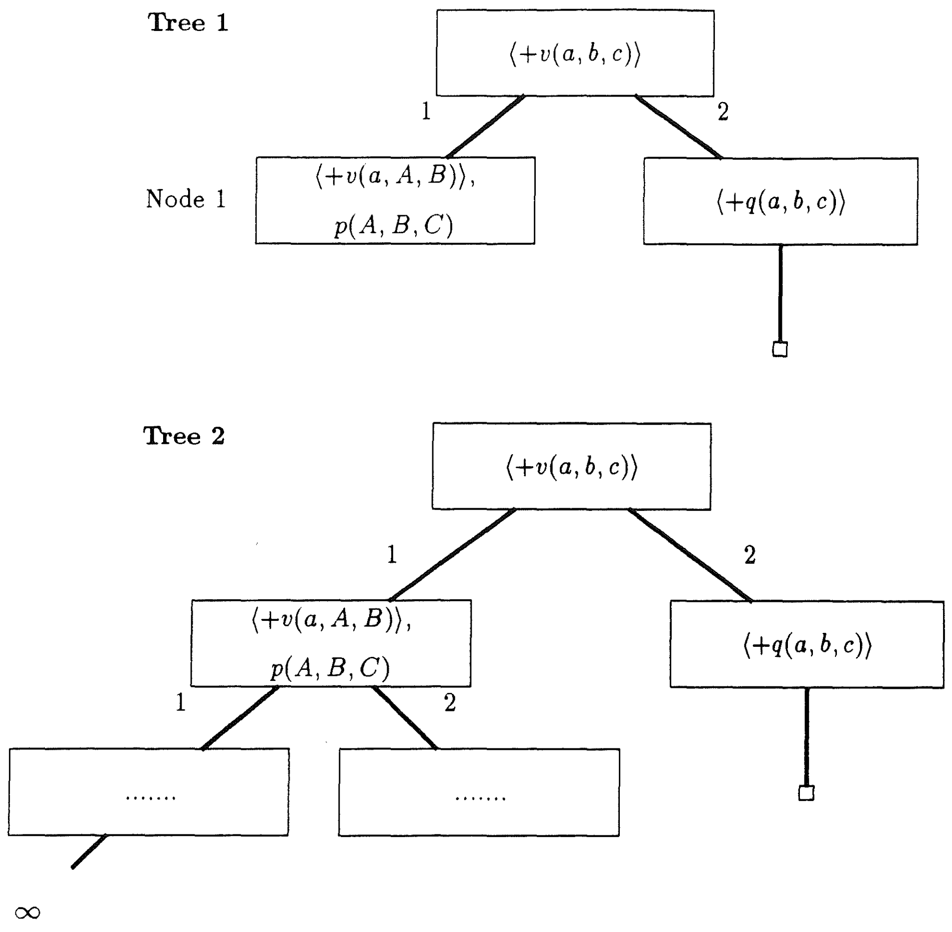
4.1. Incompleteness of the DSLD

The DSLD method described in the previous section is, unfortunately, incomplete. The completeness here refers to the *database completeness* as defined in [VIE87]. The database completeness or *db-completeness* states that if there are n successful branches in a resolution tree, the resolution process should be able to retrieve all n sets of bindings and the process should also terminate. In this section, we show what makes the method incomplete.

In [MW88], the Add and Delete translators were shown to be semantically acceptable, correct, and complete. However, as we extended these to propagate the update effects to recursive rules, the resolution becomes incomplete. It is incomplete because of the possible infinite expansion of the search tree while there are alternative branches that can satisfy the update request. We can further distinguish two cases of infinite expansion. One results from expanding a regular recursive literal while the other comes from expanding a dynamic literal. To illustrate the former, consider the following update rule:

$$\langle +v(\bar{t})\rangle : -v(\bar{u}), \langle +p(\bar{w})\rangle.$$

This rule states that to update the view v, the query $v(\bar{u})$ has to be satisfied first before updating p. The query v in this case is recursively defined and suffers from the same termination problem as described in Chapter 2. However, as also shown in Chapter 2, the lemma resolution is able to terminate the recursion and returns all possible bindings. Therefore, this termination problem does not pose any threat to the DSLD resolution, since the expansion of the recursive literal can be handled by the lemma resolution.



Tree 1
$\langle +v(a,b,c)\rangle$
1
2
Node 1
$\langle +v(a,A,B)\rangle$, $p(A,B,C)$
$\langle +q(a,b,c)\rangle$
Tree 2
$\langle +v(a,b,c)\rangle$
1
2
$\langle +v(a,A,B)\rangle$, $p(A,B,C)$
$\langle +q(a,b,c)\rangle$
1
2
....
....
∞

Figure 42

Vertical Infinite Loop of DSLD Resolution Tree

The second source of infinite loops in the search tree is the expansion of dynamic literals. -We can further distinguish two cases: (1) a non-terminating branch resulting from expanding the dynamic literal (vertical loop), and (2) an infinite branching (horizontal infinite branching) as a result of newly added tuples. We shall discuss these two types of loop (or branching) in detail next.

4.2. Vertical Loop

Vertical loop refers to the case where the DSLD resolution keeps expanding a dynamic literal without terminating. For example, consider the following Add translator:

(1) $\langle +v(X,Y,Z)\rangle :- \langle +v(X,A,B)\rangle, p(A,B,Z).$

(2) $\langle +v(X,Y,Z)\rangle :- \langle +q(X,Y,Z)\rangle.$

An update request $\langle +v(a,b,c)\rangle$ may contain a non-terminating branch as shown in Figure 42. At each stage, expanding the suspended nodes will always change the state of the database. In this example, the node marked (1) is suspended the first time and hence the process backtracks and inserts *q(a,b,c)*. Then at the next stage, node (1) is expanded by using update rule (1). Variables A and B are temporarily assigned some null values. This process keeps on forever without terminating. This is also true even when breadth-first search is used. A DSLD resolution tree containing a non-terminating path such as this is said to have a vertical loop that makes the resolution incomplete. Vertical loops can occur for both Add or Delete requests.

4.3. Horizontal Infinite Branching

Horizontal infinite branching occurs only when the update request is an insertion. The DSLD resolution trees are generated in stages. At stage i of the resolution, the newly added tuples from stage $i-1$ may expand a certain node, which results in adding yet more new tuples. In the worst case, a branch can be *self-producing*, where the branch produces tuples that can be fed back to the branch and thus generate new tuples indefinitely.

For example, consider the following update translators and rules:

(1) $\langle +v(X,Y)\rangle :- \langle +p(X,Y).\rangle$

(2) $\langle +v(X,Y)\rangle :- l(A,B), \langle +v(X,C)\rangle.$

(3) $\langle +v(X,Y)\rangle :- v(X,A), \langle +l(B,Y)\rangle.$

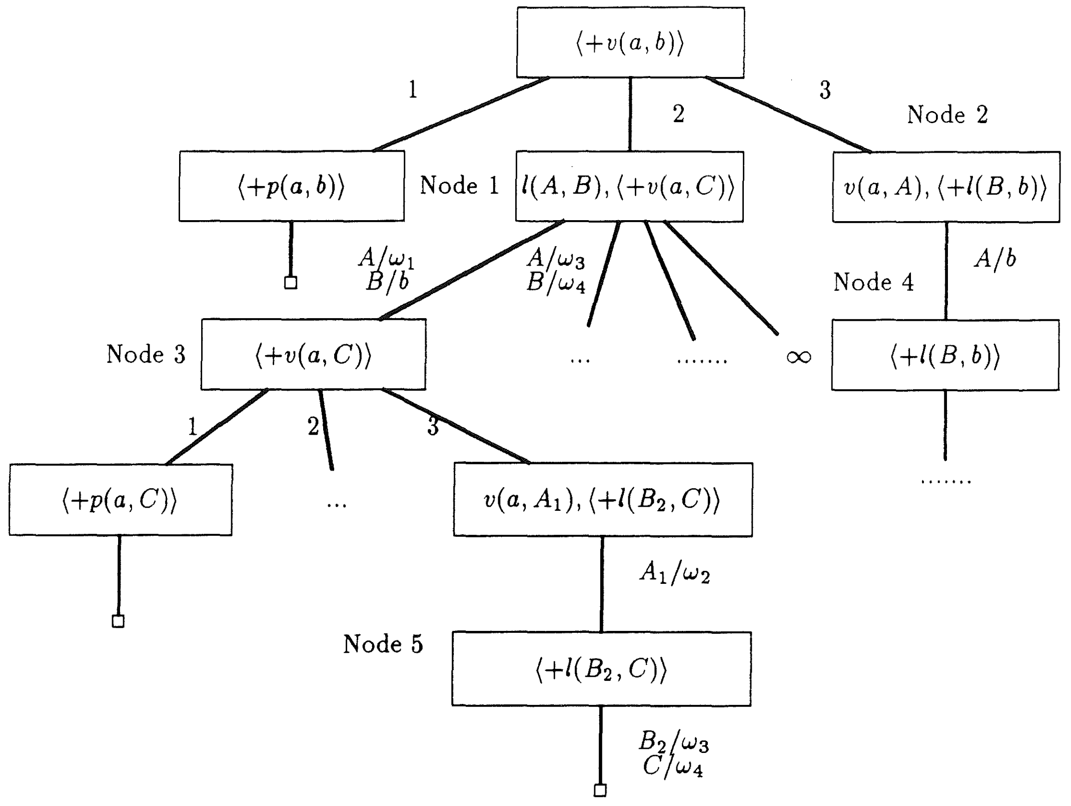


Figure 43

Horizontal Infinite Branching of DSLD Resolution Tree

Assume that l and p are base literals while v is the recursive view. The tables of l, p, and v are currently empty. Let us consider the update request $\langle +v(a,b)\rangle$. The resolution trees are depicted in Figure 43. (Since each resolution tree is a subject of the next one, only the final tree is shown explicitly here.) In the first tree, node (1) is suspended since $l(A,B)$ fails (because table l is empty). The process backtracks to node (2) and the literal $v(a,A)$ is unified with the newly established fact $v(a,b)$ from the first branch. Then, the new goal becomes node (4): $\langle +l(B,b)\rangle$. This dynamic literal can be satisfied by simply asserting the fact (ω_1, b) in the L table where ω_1 is a null value. At stage 2, node (1) is expanded since the literal *l(A,B)* is unifiable with the new fact $l(\omega_1, b)$. The new resolvent becomes $\langle +v(a,C)\rangle$ and the resolution carries on. At some later point, new tuples may again be inserted into the L table; for instance, at node (5), a new tuple (ω_3, ω_4) is added to L. Note that this newly added tuple will be used in the next stage to unify with node (1) again and to create another new branch. Hence, it is possible that there is an infinite number of new tuples added to L (if null values are allowed) and therefore, an infinite number of branches from node (1). (Note that, in this example, there is also an infinite vertical branch.)

5. Safe DSLD Resolution

When examining the vertical and horizontal branching, we discover that there is one common characteristic: there are *escaping variables* in the dynamic recursive literals. Escaping variables are simply free variables. They are "escaping" if they are still unbound by the time the dynamic literal is to be expanded. For example, consider the following update rules:

(1) $\langle +ancestor(X,Y)\rangle : -\langle +ancestor(X,A)\rangle, ancestor(A,Y).$

(2) $\langle +ancestor(X,Y)\rangle : -\langle +parent(X,Y)\rangle.$

In this case, suppose the update request is to add *ancestor(bill, john)*. Expanding the first rule is not terminating as explained in section 4. This is because the dynamic recursive literal contains an escaping variable A that is not bound when the literal is being expanded. In order to guarantee the completeness of the DSLD, it is necessary to require that there be no escaping variables in the dynamic literals of the translator. DSLD restricted in such a manner is called *safe* DSLD resolution. There are two approaches to make sure that there are no escaping variables: (1) as proposed in [MW88], all variables have to appear in the head; (2) the DBA designates the dynamic literals such that there are no escaping variables; this is verified by the system. The first approach is not desirable since one major reason to have views is to hide irrelevant information from users. To include all variables in the view definition violates this principle. Hence, we are interested in option (2).

To illustrate this, consider again the above ancestor view. The DBA could specify the following Add translator for the same view, in which the dynamic literal does not contain any escaping variable. Therefore, the DSLD resolution tree won't contain any non-terminating branches.

(1) $\langle +ancestor(X,Y)\rangle : -ancestor(X,A), \langle +ancestor(A,Y)\rangle.$

(2) $\langle +ancestor(X,Y)\rangle : -\langle +parent(X,Y)\rangle.$

In this case, the variable A in the dynamic recursive literal in rule (1) is not escaping any more. The query *ancestor(X,A)* will be evaluated first using the lemma resolution method as shown in Chapter 2. If the query is successful, then A is bound to a set of constants. Therefore, the next goal ($\langle +ancestor(A,Y)\rangle$.) will become a concrete update request since both A and Y are constants. (We require that all user update requests be ground literals, i.e. no free variable in the literal such as *ancestor(bill, Everybody)* is allowed.) Hence, this update translator is safe.

5.1. Completeness of the Safe DSLD Resolution

Based on the discussion in the previous sections, it is relatively straightforward to show the completeness of the safe DSLD resolution. There are several restrictions necessary to guarantee the safe DSLD resolution to be complete.

(1) update requests must be all ground literals;

(2) Add and Delete translators are all separate in the following sense: an Add translator can contain only insertion requests and a Delete translator can contain only deletion requests. A translator cannot contain both insertion and deletion requests (neither in the same rule nor in different rules.)

With these assumptions, we further identify the following *two* types of recursive rules that can guarantee its dynamic literals free from escaping variables:

(1) the body of a recursive rule that contains a dynamic literal that is mutually recursive with the head, with variables identical to all input variables

(however, they may be in different positions). For example, the rule: $\langle +q(X, Y, Z)\rangle : -\langle +q(Y, Z, X)\rangle$ is free from escaping variables and is safe.

(2) Within the body of a recursive rule, there are some regular literals that behave as *pre-conditions* for the dynamic recursive literals so that by the time these regular literals are satisfied, all variables in the dynamic recursive literals are bound.

Any update translators fall into these two types are *safe* translators. The DSLD resolution method described in section 4 applied to these translators are called *safe DSLD* and always terminates. If there are no escaping variables in a translator rule, there will be no escaping variables in the derivation of this rule. Since the bindings of variables in an update translator are static, detecting the existence of escaping variables is trivial. Next, we need to show how the safe DSLD always terminates.

Lemma 1: If a repeated dynamic literal is identical to one of its predecessors in the resolution tree, it can be terminated.

Proof: Obvious.

Theorem 1: There is no infinite vertical branching in safe DSLD.

Proof: The proof is straightforward. Let us consider the above two possible situations where dynamic literals are guaranteed to be free from escaping variables.

Case 1: There is a dynamic literal that is mutually recursive to the head, and all variables of that dynamic literal are from the input variables.

In this case, the original rule is either a tautology or a tautology loop as identified in [WONG86, WONG87]. A tautology rule is a rule that the head and a literal in the body are identical. For example,

$$q(X,Y) : -q(X,Y).$$

is a tautology. The Add translator of this rule may be defined as below:

$$\langle +q(X,Y)\rangle :- \langle +q(X,Y)\rangle$$

A tautology can simply be terminated by Lemma 1.

A tautology loop is defined as a rule that contains a recursive literal that has variables coming directly from input variables but they are in different positions in the literal.

For example, the following update rules form a tautology loop:

(1) q(X,Y,Z) :- p(X,Y,Z). (exit clause)
(2) q(X,Y,Z) :- q(Y,Z,X).

In this case, rule (2) forms a tautology loop, in which, the recursive literal in the body contains exactly the same variables as the head but in different positions. The different positions of the variables actually form a permutation cycles. A tautology loop has first been identified in [WONG86, WONG87]. It has also been shown that, by expanding the rule n times the next goal is identical to the original goal, where n is order of the permutation. For instance, the permutation order of rule (2) above is 3. Therefore, by expanding the rule three times, the next goal is identical to the original goal. Hence, a update translator defined on a tautology loop always terminates (by lemma 1.)

For instance, the Add translator of the above tautology loop may be defined as follows:

(1) $\langle +q(X,Y,Z)\rangle :- \langle +q(Y,Z,X)\rangle.$
(2) $\langle +q(X,Y,Z)\rangle :- \langle +p(X,Y,Z)\rangle.$

To illustrate this, let's assume that the update request is $\langle +q(1,2,3)\rangle$ where *1, 2, 3* are constants. The DSLD tree is depicted in Figure 44. After expanding rule (1) three times, the next goal becomes identical to the original goal and hence, by using lemma 1, the goal can be terminated.

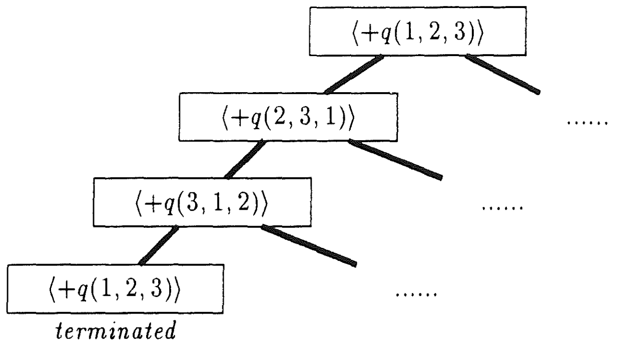


Figure 44

Example of Terminating A Tautology Update Translator

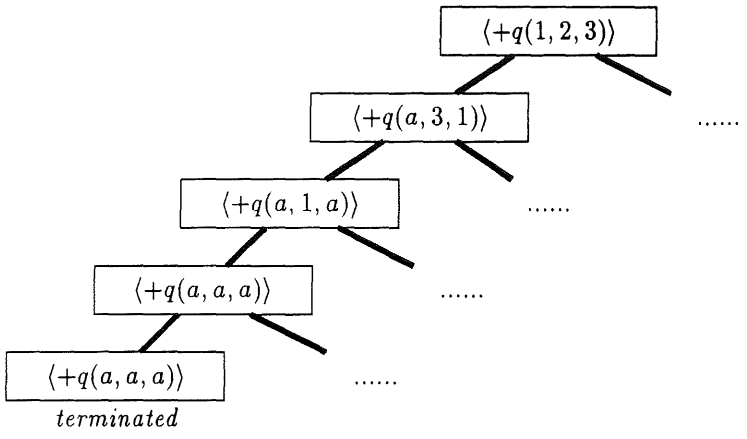


Figure 45

Another Example of Terminating Tautology Update Translator

Another variant tautology loop translator is when the dynamic recursive literal contains some constants but the variables are directly from the input variables. For instance, consider the update translator below:

(1) $\langle +q(X, Y, Z)\rangle : -\langle +q(a, Z, Y)\rangle.$

(2) $\langle +q(X, Y, Z)\rangle : -\langle +p(X, Y, Z)\rangle.$

In this case, the dynamic recursive literal $\langle +q(a, Z, Y)\rangle$ does not contain any escaping variables. Similar to tautology loop translator above, this type of update

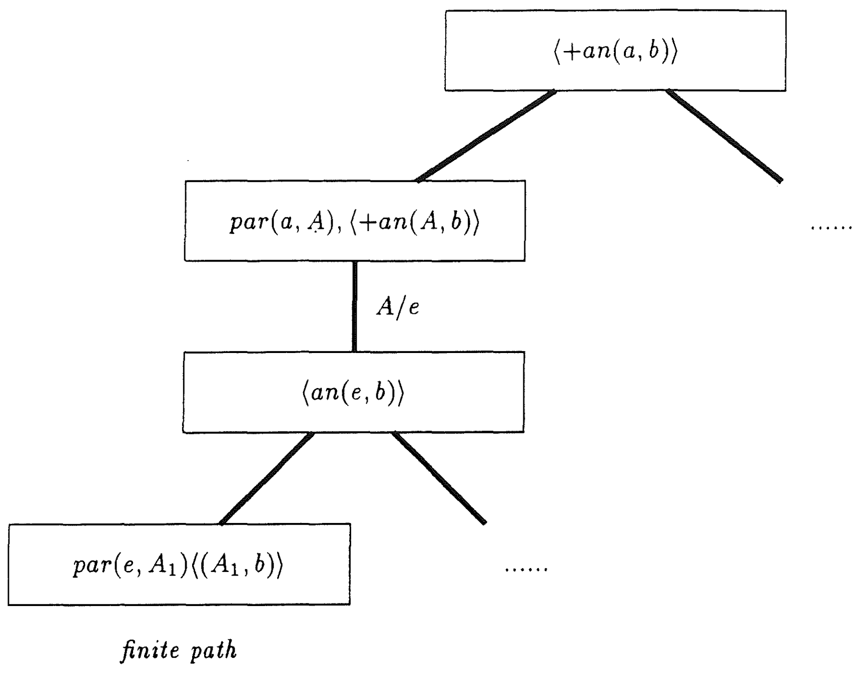


Figure 46

Terminating Vertical Loop in Safe Update Translator

rule always terminates. The expansion of the update request $\langle +q(1,2,3)\rangle$ is shown in Figure 45. In this example, the first term of the dynamic literal in the body is a constant. However, the other two terms are variables directly from the input variables. In a finite number of expansion, the goal is repeated and is identical to one of the predecessor goals in the resolution tree.

Case 2: The dynamic recursive literals are preceded by regular literals that bind all free variables in the dynamic literals.

Since there are some regular literals preceding the dynamic literals, if all variables in the dynamic literals will become constants before they are expanded, then the translator is safe and will terminate. The translator always terminates because the bindings of the dynamic literals are drawn from the finite domain of those preceding regular literals.

To illustrate the point, consider the following update translator:

(1) $\langle +ancestor(X,Y)\rangle : -\langle +parent(X,Y)\rangle.$

(2) $\langle +ancestor(X,Y)\rangle : -parent(X,A), \langle +ancestor(A,Y)\rangle.$

The dynamic literal $\langle +ancestor(A,Y)\rangle$ in rule (2) does not contain any escaping variables since variable A will be bound if the regular literal *parent(X,A)* succeeds. Since there are no escaping variables, there will be no *null* values generated in the expansion (as in the case discussed in sections 4.2 and 4.3) which could cause an infinite expansion of the dynamic literal. The literal *parent(X,A)*, in this example, is the pre-condition to insert a tuple in the *ancestor* view. To expand the dynamic literal, the pre-condition has to be satisfied first. Note that there also exists a *safety condition* defined in [ULL89] that is applied to Horn database to guarantee the completeness. The condition states that all variables that occur in the head of a rule also occur in the body of that rule. This safety condition, combined with the requirement that each fact in the database has to be a ground fact, has been proven to guarantee that only a finite number of facts can be deduced from the database [ULL89]. The requirement of escaping variable free is, as a matter of fact, a stronger safety condition than the one proposed in [ULL89]. In other words, because the number of domain values in a database is finite, the number of possible tuples of a specific relation is also finite. Therefore, the test of the pre-condition *parent(X,A)* will eventually fail and hence terminate.

For example, the DSLD resolution tree of $\langle +ancestor(a,b)\rangle$ is shown in Figure 46. The leftmost branch of the tree contains the pre-condition *parent(b/X, f/A)* which is the adornment pattern with a bound and a free variable. This pre-condition will eventually terminate since the domain of *parent(b/X,f/A)* is finite.

Therefore, a safe DSLD resolution tree always terminates.

Theorem 2: There is no infinite horizontal branching in safe DSLD.

Proof: Theorem 1 states that there is no infinite vertical branching in safe DSLD. This implies that every branch will eventually terminate. From any expandable node, a new branch could start *only* if the preceding literals (the pre-condition) were satisfied with some newly generated tuples. Since the number of domain values for a specific relation is finite in the case of safe DSLD, there is no infinite horizontal branching in safe DSLD.

Theorem 3: The safe DSLD is db-complete.

Proof: Lemma resolution (Chapter 2) is db-complete. The dynamic logic programming in [MW88] is also complete. DLP becomes incomplete only when it is extended to define recursive dynamic literals because of the possibility of vertical and horizontal loops. Theorems 1 and 2 show that safe DSLD always terminates without any vertical and horizontal loops. Therefore, the safe DSLD is db-complete

5.2. Termination of Anterior View Update and Posterior View Update Loop

If a view is arbitrarily defined by multiple rules, base update (e.g. posterior view update to derive affected view tuples) and view update (e.g. anterior view update to generate supports of view update) may form a non-terminating loop. In this section, we shall show that this loop always terminates for safe update translators.

To illustrate how anterior and posterior view updates may form a loop, consider the following set of rules:

(1) $v(X,Y,Z) : -p_1(X,A), p_2(A,Y,Z).$

(2) $v(X,Y,Z) : -p_2(X,Y,A), p_3(A,Z).$

Assume that the Add translator is defined as below:

(1) $\langle +v(X,Y,Z)\rangle : -\langle +p_1(X,A)\rangle, p_2(A,Y,Z).$

(2) $\langle +v(X,Y,Z)\rangle : -\langle +p_2(X,Y,A)\rangle, p_3(A,Z).$

If a base update to $p_2(A,Y,Z)$ in rule (1) succeeds and if the view v is affected, the posterior view update may generate a set of tuples to be inserted to the view v. However, to insert each of these tuples in the view may need to generate all possible support tuples (due to strong propagation), which may trigger an anterior view update, for example, propagating the request to the update rule (2) above. In this case, a new tuple has to be added to p_2 in order to support this action. If the update rule succeeds (after $\langle +p_2(X, Y, A)\rangle$ succeeds and the literal $p_3(A, Z)$ is satisfied), then the newly added tuple in p_2 takes effect and initiates another posterior update. The loop can be infinite if null values are allowed.

Note that another potential loop occurs when the posterior view update generates *unsafe* view tuples (e.g. containing null values). In this case, the anterior view update is also not safe because it suffers from vertical and horizontal infinite branching as discussed above.

Fortunately, under the same assumptions as in section 5.1 and assuming that update translators contain only safe rules (e.g. no escaping variables), these two potential loops do not occur.

First, since all update requests are assumed to be ground literals, the posterior view update will generate a finite set of tuples to be updated in the view table. These tuples contain all bound terms without any null values. Therefore, each anterior view update triggered by these tuples fall into the safe DSLD clauses described above and will terminate as shown.

Similar to the proof of Theorems 1 and 2, there are also a finite number of possible bindings generated by the anterior view update. For example, given a set of safe DSLD clauses and all update requests being ground literals, the number of tuples generated by $\langle +p_2(X, Y, A)\rangle$ in rule (2) of the above update translator is finite. Hence, the loop between posterior view update and anterior view update always terminates.

6. Chapter Summary

In this chapter, we recognized that updating views is a desirable feature of any intelligent database systems. However, view updates are inherently ambiguous.

To solve this problem, we adopted the heuristic approach that allows the database administrator to designate the propagation of update effects. The theoretic approach is not practical since it generates mostly non-Horn clauses which make the problem intractable. Furthermore, generating new "rules" implies that query must be recompiled every time there is an update request. The heuristic approach is more attractive because it does not generate any new rules and can be integrated into a relational system easily. However, the system proposed earlier in [MW88] fails to define the update translators for recursive view updates. The major problem, we believe, is the fact that termination of neither regular nor dynamic literals can be guaranteed with arbitrary update rules. In this chapter, we proposed a new resolution method that is based on the lemma resolution and is combined with the dynamic logic programming method. Our approach identifies a subclass of recursive update requests that are safe in terms of termination and completeness.

Our approach to view updates is superior to [MW88] in the following ways:

(1) We defined the semantics for recursive view updates;

(2) We extended the SLD resolution to accommodate dynamic logic programming to define the dynamic SLD resolution;

(3) We defined the strong propagation of update effects and hence are able to use multiple rules;

(4) We showed that the dynamic SLD resolution is incomplete for an arbitrary function-free Horn database;

(5) We further identified a subclass of *safe* recursive view update translators which, when restricted to dynamic literals free from escaping variables, always terminate and hence are complete;

(6) Finally, we proved that the safe DSLD terminates and hence is complete.

CHAPTER 6
Conclusions and Directions for Future Work

This dissertation has presented a unifying approach to solve both query and update processing when logic programming is integrated (via query compilation) with relational databases. In the studies of deductive databases, these two issues have been investigated, most of the time, separately. However, when performance is of concern, it is necessary to have a general strategy that encompasses both query and update processing in a way that would improve the overall performance. Fortunately, using the principles of database locality and lemma resolution, we were able to develop our incremental methods for queries and updates. We believe that our study is the first to address these two issues within the same framework while at the same time, the overall performance is being considered.

In this chapter, we first summarize our accomplishments. We then present several open issues that should be addressed in future research.

1. Summary

The work in this dissertation embodies the following major accomplishments:

(1) *Query Compilation.* Based on lemma resolution, we designed a general algorithm that will compile any recursive query into an iterative program containing relational algebra operators. Once a query has been compiled, it can be stored within the relational database as a procedure. Any future references to the deductive rules are then treated as invocations of this procedure. Hence, there is no need to have a separate inference engine; the inference rules are embedded in the relational system. This has been described in Chapter 2.

(2) *Partially Materialized Views.* We proposed to keep the bindings generated by processing queries (view materialization). However, in our approach, views are materialized *incrementally.* They are materialized as the query progresses, instead of materializing the complete view table at once when the view is defined. This is discussed in Chapter 3.

(3) *Enhanced Query Processing.* Based on the query compilation and partially materialized views, we designed a strategy that enhances the query process by searching the partially materialized views first before invoking the compiled query procedures. The modified query process is designed with the principle of database locality in mind. If locality is present, the modified query process takes full advantage of it, since repeated queries are all retained. Therefore, answering a repeated query is reduced to a simple selection operation on these partially materialized tables. This is illustrated in Chapter 3.

(4) *Base Update Processing.* If views are materialized, updates will affect the validity of views. We first addressed the problems of base updates, i.e., updates on base tables. We designed algorithms for an efficient screening test to identify those base updates that affect the view tables, and, hence require immediate updating of the view tables. With the incremental update and query processing methods in (2) and (3) above, we then demonstrated that base updates can be implemented efficiently. We further showed how heuristics can be applied to improve query and base updates. In particular, we demonstrated that the proposed Complete View table method, along with recording the lemma dependency links when views are materialized, can improve the performance of deletion requests on base tables. This is demonstrated in Chapter 3.

(5) *Empirical Studies on Queries and Base Updates.* We have implemented our incremental methods of base update and query processes for a specific application: a transitive closure relationship. We compared our methods against the totally materialized method and the on-the-fly method. We concluded that our method is a compromise between these two methods. It performs better when there are moderate levels of update activities while the repetition ratio and selectivity is high. This is demonstrated in Chapter 4.

(6) *Recursive View Updates and Dynamic Lemma Resolution.* We have developed a solution for the recursive view update problem. We first defined the dynamic SLD resolution (DSLD) by combining the lemma resolution method with dynamic logic programming. With the DSLD, we defined the semantics of recursive view updates and showed how update translators can be defined with the DSLD. Unfortunately, the DSLD is still incomplete for arbitrary translators. Hence, we identified a subclass of *safe* update translators that contain no escaping variables in the dynamic literals. We proved that view updates are complete and will terminate with these safe translators. These translators are procedurally oriented and hence can be easily integrated with a relational system.

2. Directions for Future Work

Despite its many virtues, the approach presented in this dissertation suffers from a few limitations. We list several issues that should be addressed in the future to make the current approach more desirable.

(1) *Compilation of Base Update Requests*

As mentioned in Section 3.2 of Chapter 3, updates on a base table may be statically analyzed. Hence, the next logical step should be to compile the possible base updates into procedure calls. For example, in Figure 13 in Chapter 3, updating

P_2 always triggers updates of the view v. However, updating P_1 depends on the constants of the update attributes and the current view table contents. In any case, if the view has to be updated, it is analogous to answering a query. Hence we could use the updated base tuple to derive the affected view tuples. Therefore, for each potential base update, it is possible to generate a compiled procedure to implement all the necessary update activities.

(2) *Compilation of View Update Requests*

The update translators presented in Chapter 5 can also be compiled. Since the update translators are already algorithms indicating how and what to update, they can also be compiled.

(3) *Effects on Integrity Constraints*

As shown in section 4 of Chapter 5, integrity constraints are the most important vehicle to prevent non-determinism in the update translators. The full effect of the interactions of update translators and integrity constraints have to be studied in the future. Studies have shown that integrity constraints are not only useful in maintaining the data integrity during updates, but they also help to aid the search process and, hence, improve performance [CGM88].

(4) *Multiple Versions of Update Rules*

In Chapter 5, we show that the DBA can only designate a single form of each translator. However, the system would be more flexible if the DBA could specify *multiple versions* of a translator. For example, for the same rule $v(\bar{t}) :- p_1(\bar{u}), p_2(\bar{k}), v(\bar{w})$, the DBA may designate the following two versions of an Add translator:

(1) $\langle +v(\bar{t})\rangle :- p_1(\bar{u}), p_2(\bar{k}), \langle +v(\bar{w})\rangle$, OR

(2) $\langle +v(\bar{t})\rangle :- p_1(\bar{u}), \langle +p_2(\bar{k})\rangle, v(\bar{w})$.

Note that the multiple versions are different from multiple rules. Multiple versions imply that the update requests can be done by either (1) or (2) or both. The main questions are: How will this complicate our DSLD resolution, and, would it still be safe with multiple versions?

(5) *Complexity Analysis of Recursive View Updates*

The complexity of implementing recursive view update would be an interesting research problem. Since the theoretic approach of view updates proposed by [FUV83] and by [RN88] is intractable, it would be important to analyze the complexity of our current approach. It would also be interesting to perform some empirical studies of recursive view updates to reveal the more practical problems with this approach.

(6) *Parallel Query Processing*

As noted in [RS90], if recursive query evaluation is to be of practical use, it is important that the queries be evaluated efficiently against large database tables. From the empirical studies in Chapter 4, we notice that it is very costly to process a recursive query. One possible improvement of our current approach is to utilize multiple processors to evaluate recursive queries. Several proposals have already been made on how to evaluate the transitive closure with a bottom-up evaluation method. For example, in [AJ88], the following algorithm has been given:

```
1.   $R^p_f \leftarrow R^p_0$
2.   $R^p_\Delta \leftarrow R^p_0$
3.   while $R^p_\Delta \neq \emptyset$ do
4.           $R^p_\Delta \leftarrow R^p_\Delta \bowtie R_0$
5.           $R^p_\Delta \leftarrow R^p_\Delta - R^p_f$
5.           $R^p_f \leftarrow R^p_f \cup R^p_\Delta$
```

In this scheme, R^p_0 is initially partitioned on some attributes and is assigned to each processor. The superscript p indicates that the relation has been partitioned and assigned to the p^{th} processor. The R^p_f is the resulting table of each processor p and the R^p_Δ is the immediate result of the bottom-up evaluation. The same algorithm is being executed on every processor independent from each other. The results of each R^p_f are then unioned together to become the final results of the entire transitive closure. The advantage is that there is no communication or synchronization required between processors. With a good partition (i.e. making each processor is equally busy), it will perform well. However, it has to access the entire R_0 relation in each iteration (line 4 above) and there may be redundancy in the computations of all the processors.

Another similar approach was proposed in [VK88], in which the relation R_0 is horizontally partitioned and each processor computes the bottom-up evaluation, corresponding to its partitions of R_0 and R_Δ, without accessing to the original R_0 relation. However, the results of each processor must still be *merged* with one another. The two-way merges, unfortunately, perform worse than the above algorithm.

Recently, in [RS90], a *parallel pipelined strategy* based on the top-down evaluation method was proposed and shown to have better performance than the above algorithm. In this method, the recursive query is evaluated in the top-down manner similar to the example of expanding the formula in Section 3.1 in Chapter 4. For instance, the ANCESTOR view can be generated by the formula:

$$\sigma_a PARENT \bowtie^k PARENT$$

The formula is expanded as shown below.

n=0	$\sigma PARENT$
n=1	$(\sigma PARENT) \bowtie PARENT$
n=2	$(\sigma PARENT \bowtie PARENT) \bowtie PARENT$
n=3	$(\sigma PARENT^2 \bowtie PARENT) \bowtie PARENT$
	......
n=k	$(\sigma PARENT^{k-1} \bowtie PARENT) \bowtie PARENT$

The results of operations in parentheses are identified as *common* structures, which are basically the same *wavefront* concept as in [HL86]. With these common structures, they identify possible parallelism in executing these operations and assign each operation to a processor for evaluation.

We believe that the lemma resolution, especially, the dependency between lemmas as mentioned in Chapter 3, is a better way to identify the common structures. Moreover, since we retain results from previous operations, the performance will be better than the above method, which still requires the common structures to be evaluated more than once. We also believe that the different adornment patterns in our partially materialized view tables will provide us with hints on how the views or base tables should be partitioned in order to facilitate future query processing. Most importantly, we believe that both our incremental update and query processes can be implemented in a multiprocessor environment in which queries and updates can be processed in parallel with fewer concurrency control problems.

References

[AJ88] AGRAWAL, R., AND H. V. JAGADISH Multiprocessor Transitive Closure Algorithms. In *Proc. of the 1988 Intl Symposium on Databases in Parallel and Distributed Systems*, 1988.

[APVA82] APT, K.R. AND M.H. VAN EMDEN Contributions to the Theory of Logic Programming. *JACM 29* (1982), 841–862.

[AU79] AHO, A., AND J.D. ULLMAN Universality of Data Retrieval Language. In *Proc of the Sixth Annual ACM Symposium on Principles of Programming Languages*, 1979, pp. 110–120.

[BC79] BUNEMAN, O.P., AND E. CLEMONS Efficiently Monitoring Relational Databases. In *ACM Trans. on Database Systems*, Vol. *4:3*, 1979, pp. 368–382.

[BEER87] BEERI, C. ET AL Sets and Negation in a Logical Database Language (LDL1). In *Proc Sixth ACM SIGMOD-SIGART Symp. on Principles of Database Systems*, San Diego, CA, 1987.

[BIR80] BIRD, R.S. Tabulation Techniques for Recursive Program. *ACM Computing Survey 12 4* (December, 1980), 4403–418.

[BLT86] BLAKELEY, J. A., P. LARSON, AND F.W. TOMPA Efficiently Updating Materialized Views. In *Proc. of 1986 ACM SIGMOD Conference on Management of Data*, Washington DC, 1986, pp. 61–71.

[BMSU86] BANCILHON, F., D. MAIER, Y. SAGIV, AND J. D. ULLMAN Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc ACM SIGMOD-SIGART Symp. on Principles of Database Systems*, Cambridge, MA, 1986.

[BR86] BANCILHON, F., AND RAMAKRISHNAN, R. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of SIGMOD '86 International Conference on Management of Data*, ACM, 1986, pp. 16–52.

[BS81] BANCILHON, F., AND N. SPYRATOS Update Semantics of Relational Views. In *ACM Trans. on Database Systems*, Vol. *6(4)*, 1981, pp. 557–575.

[BW84] BROUGH, D.R. AND A. WALKER Some Practical Properties of Logic Programming Interpreters. In *Proceedings of the Int'l Conf on Fifth Generation Computer Systems*, Institute of New Generation Computing, Tokyo, Japan, 1984, pp. 149–156.

[CG85] CLARK, K. AND S. GREGORY Notes on the Implementation of Parlog. *Journal of Logic Programming 1* (1985), 17–42.

[CGM88] CHAKRAVARTHY, U. S., J. GRANT, AND J. MINKER Foundations of Semantic Query Optimization for Deductive Databases. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed., Morgan Kaufmann, Los Altos, Calif, 1988.

[CGT90] CERI, S., G. GOTTLOB, AND L. TANCA *Logic Programming and Databases*, Springer-Verlag, Berlin Heidelberg, 1990.

[CH82] CHANDRA, A.K., AND D. HAREL Horn Clauses and the Fixpoint Query Hierarchy. In *ACM SIGMOD*, 1982, pp. 158–163.

[CHA78] CHANG, CHIN-LIANG DEDUCE 2: Further Investigations of Deduction in Relational Data Bases. In *Logic and Data Bases*, Gallaire, H. and Minker, J., Ed., Plenum Press, New York, 1978, pp. 201–236.

[CHAN73] CHANDRA, A.K. Efficient compilation of linear recursive programs.. In *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, Iowa City, Iowa, 1973, pp. 16–25.

[CHANG81] CHANG, CHIN-LIANG On Evaluation of Queries Containing Derived Relations in a Relational Data base. In *Advances in Data Base Theory volume 1*, Gallaire, H, Minker, J and Nicolas, Jean Marie, Ed., Plenum Press, New York, 1981, pp. 235–260.

[CL73] CHANG, CHIN-LIANG AND LEE, RICHARD CHAR-TUNG *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[CM81] CLOCKSIN W.F. AND MELLISH C.S. *Programming in PROLOG*, Springer-Verlag, 1981.

[CODD70] CODD, E. F. A Relational Model of Data for Large Shared Data Banks. *CACM 13*, 6 (June, 1970).

[CODD72] CODD, E. F. Relational Completeness of Data Base Sublanguages. In *Data Base Systems, Courant Computer Science Symposia Series 6*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[CODD85] Codd, E. F. Is Your DBMS Really Relational?. In *Computerworld*, 1985.

[COH79] Cohen, N.H. Characterization and Elimination of Redundancy in Recursive Programs. In *ACM Sixth Annual Conf. on Principles of Programming Languages*, 1979, pp. 1443–157.

[COH83] Cohen, N. H. Eliminating Redundant Recursive Calls,. In *ACM TOPLAS*, Vol. *5*, 1983, pp. 265–299.

[DAHL82] Dahl V. On Database Systems Development Through Logic. *ACM Transactions on Database Systems 7*, 1 (March, 1982), 102–123.

[DATE90] Date, C.J. *An Introduction to Database Systems - Fifth Edition*, Addison Wesley, Vol. *Volume 1*, 1990.

[DKO84] DeWitt, D.J., R.Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood Implementation Techniques for Main Memory Database Systems,. In *Proc. ACM SIGMOD Conf*, 1984, pp. 1–8.

[DOY79] Doyle, J. A Truth Maintenance System. *Artificial Intelligence 12* (1979), 231–272.

[DW85] Dietrich, S.W and D.S. Warren Dynamic Programming Strategies for the Evaluation of Recursive Queries. Technical Report Number 85-31. Computer Science Department, SUNY at Stony Brook (1985).

[DW86] Dietrich, S.W and D.S. Warren Extension Tables: Memo Relations in Logic Programming. Technical Report Number 86-18. Computer Science Department, SUNY at Stony Brook (1986).

[EAR70] Earley, Jl An Efficient Context-free Parsing Algorithm. In *CACM*, Vol. *13:2*, 1970, pp. 354–381.

[FiHe77] Fikes, R.E. and Hendrix, G.G. A Network-Based Knowledge Representation and Natural Deduction System. In *Porceedings of 5th International Joint Conference on Artificial Intelligence*, 1977, pp. 235–246.

[FUV83] Fagin, R., J. D. Ullman, and M. Vardi On the Semantics of Updates in Databases. In *Proc. 2nd ACM PODS*, 1983.

[GM78] Gallaire, Herve and Minker, Jack *Logic and Data Bases*, Plenum Press, New York, 1978.

[GMN78] GALLAIRE, H., J. MINKER, AND J. M. NICHOLAS *Advances in Database Theory, Vol II*, Plenum Press, 1984.

[HA79] HAREL, D. *First-Order Dynamic Logic, Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[HAN87] HANSON, E. N. A Performance Analysis of View Materialization Strategies. In *ACM SIGMOD Conference Proceeding*, 1987.

[HL86] HAN, J. AND H. LU Some Performance Results on Recursive Query Processing in Relational Database Systems. In *Proc. of the IEEE Conf. on Data Engineering*, 1986.

[HN84] HENSCHEN, LAWRENCE J. AND NAQVI, SHAMIN A. On Compiling Queries in Recursive First-Order Databases. *Journal of the ACM 31*, 1 (January, 1984), 47–85.

[ITOH86] ITOH, H. Research and Development on Knowledge Base Systems at ICOT. In *Proc of 12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, 1986.

[JAR77] JARDINE, D. The ANSI/SPARC DMBS Model. In *Proc of the 2nd SHARE Working Conf. on Data Base Management Systems*, Montreal, Canada, 1976.

[KDC87] KAMEL, M. N., S. DAVIDSON, AND E. CLEMONS Semi-Materialization: A Technique for Optimizing Frequently Executed General Queries. Tech. Report MS-CIS-87-66. Dept of Computer and Information Science, University of Pennsylvania (August, 1987).

[KEL85] KELLER, A. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. In *Proc. of the 4th ACM PODS*, 1985, pp. 154–163.

[KENT89] KENT, W. *Data and Reality - Basic Assumptions in Data Processing Reconsidered*, North-Holland, 1978.

[KER87] KERSCHBERG, L. *Expert Database Systems*, The Benjamin Cummings Publishing Company, Inc., 1987.

[KK71] KOWALSKI, R.A. AND KUEHNER, D., Linear Resolution with Selection Function. In *Artificial Intelligence, 2(1971)*, pp. 227-260.

[KOW79A] KOWALSKI, R. Algorithm Logic + Control. *communications of the ACM 22*, 7 (July, 1979), 424–436.

[KOW79B] KOWALSKI, R. *Logic for Problem Solving*, North-Holland, New York, 1979.

[KS86] KELLER, R.M. AND M.R. SLEEP Applicative Caching. *ACM TOPLAS 8 1* (January, 1986), 88–108.

[LHM86] LINDSAY, B, L. HAAS, C. MOHAN, H. PIRAHESH, AND P. WILMS A Snapshot Differential Refresh Algorithm,. In *ACM SIGMOD*, 1986, pp. 53–60.

[LLO84] LLOYD, J.W. *Foundations of Logic Programming*, Springer-Verlag, New York, 1984.

[LOV78] LOVELAND DONALD W. *Automated Theorem Proving: A Logical Basis*, North-Holland, New York, 1978.

[LR81] LOVELAND, D. W. AND C. R. REDDY Deleting Repeated Goals in the Problem Reduction Format. *JACM 28*, 4 (October, 1981), 646-661.

[MAN74] MANNA, ZOHAR *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[MBW80] MYLOPOULOS, J., BERNSTEIN, P.A., AND WONG, H.K.T. a Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems 5*, 2 (June, 1980), 185–207.

[MD80] McDERMOTT, D., AND J. DOYLE Non-monotonic Logic. *Artificial Intelligence 13* (1980), 41–77.

[MIC68] MICHIE, D. 'Memo' Functions and Machine Learning. *Nature 218* (April, 1968), 19–22.

[MIN88] MINKER, J. *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, Calif, 1988.

[MS81] McKAY, D. AND SHAPIRO, S. Using Active Connection Graphs for Reasoning with Recursive Rules. In *Proc. 7th Int'l Joint Conference on Artificial Intelligence*, 1981, pp. 368–374.

[MW88] MANCHANDA, S., AND D. S. WARREN A Logic-based Language for Database Updates. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed., Morgan Kaufmann, Loas Altos, Calif, 1988, pp. 363–394.

[NA88] NAUGHTON, J. F. Compiling Separable Recursions. In *ACM SIGMOD*, 1988, pp. 312–319.

[NS88] Future Directions in DBMS Research. TR-88-001. International Computer Science Institue (Feb, 1988).

[REI78] REITER, RAYMOND Deductive Question-Answering on Relational Data Bases. In *Logic and Databases*, H. Gallaire and J. Minker, Ed., Plenum Press, New York, 1978, pp. 149–177.

[RLK86] ROHMER, J, R. LESCOEUR, AND J. M. KERISIT The Alexander Method – A Technique for the Processing of Recursive Axioms in Deductive Databases. In *New Generation Computing*, OHMSHA, LTD. and Springer-Verlag, Vol. *4*, 1986, pp. 273-285.

[RM75] ROUSSOPOULOS, N. AND MYLOPOULOS, J. Using Semantic Networks for Data Base Management. In *Proceedings First International Conference on Very Large Data Bases*, ACM, 1975, pp. 144–172.

[RN88] ROSSI, F., AND S. NAQVI Contributions to the View Update Problem (Extended Abstract). DB-213-88. Microelectronics and Computer Technology Corporation (MCC), Austin, TX (1988).

[RS90] RASCHID, L., AND S. SU A Parallel Pipelined Strategy for Evaluating Linear Recursive Predicates in a Multiprocessor Environment. Submitted for Publication. Dept of Information Systems and Institute for Advanced Computer Studies, University of Maryland (1990).

[SSH87] STONEBRAKER, M., T. SELLIS, AND E. HANSON An Analysis of Rule Indexing Implementation in Data Base Systems. In *Expert Database Systems*, L. Kerschberg, Ed., The Benjamin-Cummings, 1987.

[STON75] STONEBRAKER, M. R. Implementation of Integrity Constraints and Views by Query Modification. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, San Jose, Calif, 1975.

[STZ87] SHMUELI, O., S. TSUR, AND H. ZFIRA Rule Support in Prolog. In *Expert Database Systems*, L. Kerschberg, Ed., The Benjamin-Cummings Publishing Company, Inc., 1987, pp. 247–269.

[SZ87] SACCA, D., AND C. ZANIOLO Magic Counting Methods. In *Proc of the ACM-SIGMOD Conference*, 1987.

[SZET86] SHMUELI, O., H. ZFIRA, R. EVER-HADANI Dynamic Rule Support in Prolog. In *Fifth Generation Computer Architectures*, J. V. Woods, Ed., Elsevier Science Publishers B.V. (North-Holland), 1986.

[TS86] TAMAKI, H. AND T. SATO OLD Resolution with Tabulation. In *Proceedings of the Third Int'l Conf on Logic Programming*, Springer-Verlag, 1986, pp. 84–98.

[TSUR86] TSUR, S. AND C. ZANIOLO LDL: A Logic-Based Query Language. In *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, Japan, 1986.

[ULL85] ULLMAN, J. D. Implementation of Logical Query Languages for Databases. In *ACM Trans. on Database Systems*, Vol. *10:3*, 1985.

[ULL89] ULLMAN, J. D. *Principles of Database and Knowledge-Base Systems, Vols I and II*, Computer Science Press, Rockville, MD, 1989.

[VAKO76] VAN EMDEN, M.H. AND R.A. KOWALSKI The Semantics of Predicate Logic as a Programming Language. *JACM 23 4* (1976), 733–742.

[VIE87] VIEILLE, L., A Database Complete Proof Procedure Based on SLD Resolution. In *Proc. 4th Int. Conf. on Logic Programming ICLP '87*, Melbourne, Australia, 1987.

[VIE88] VIEILLE, L. From QSQ towards QoSaQ: Global Optimization of Recursive Queries. In *Proc of the 2nd Intl Conf. on Expert Database Systems*, Kerschberg, L., Ed., Benjamin-Cumming, 1988.

[VK88] VALDURIEZ, P. AND S. KHOSHAFIAN Transitive Closure of Transitively Closed Relations. In *Proc. of the 2nd Intl Conf. on Expert Database Systems*, 1988.

[WAR81] WARREN, D. Efficient Processing of Interactive Relational Database Queries Expressed in Logic. In *Proceedings Seventh International Conference on Very Large Data Base*, ACM, 1981.

[WAR84] WARREN, D. S. Database Updates in Pure Prolog. In *Proc. of the Intl Conf. on Fifth Generation Computer Systems*, ICOT, Tokyo, Japan, 1984, pp. 244–253.

[WHA88] WHANG, K. Y. AND S. BRADY A Framework for Optimization in Expert System - DBMS Interface. In *Proc. of the Intl Conf. on Data Engineering, IEEE*, 1988, pp. 126–133.

[Wis86] Wise, Michael *Prolog Multiprocessors*, Prentice-Hall, 1986.

[WOLB84] Wos, Larry, Overbeek, Ross, Lusk, Ewing and Boyle, Jim *Automated Reasoning, Introduction and Applications*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1984.

[WONG83] Wong, E. Semantic Enhancement through Extended Relational Views. In *2nd International Conference on Databases(ICOD-2)*, Cambridge, U.K., 1983, pp. 169–178.

[WONG86] Wong, W. C. and L. Bic Efficient Recursion Termination For Function-Free Horn Logic. Technical Report 86-26. ICS Dept, University of California, Irvine (1986).

[WONG87] Wong, W. C. and L. Bic A Tagging Scheme to Prevent Infinite Recursion in First-Order Databases. In *Proceedings of the Second Int'l Conference on Computers and Applications, IEEE*, 1987.

[WONG87a] Wong, W. C. On Terminating Infinite Recursion for Function-Free Horn Logic. Parallel Computation Notes, No.4 (unpublished research group notes). Parallel Computing Group, ICS Dept, University of California, Irvine (August, 1987).

[WONG87b] Wong, W. C. On Compilation of Recursive Rules. Parallel Computation Notes, No.5 (unpublished research group notes). Parallel Computing Group, ICS Dept, University of California, Irvine (Sept, 1987).

[WSB87a] Wong, W. C., T. Suda, and L. Bic Performance Analysis of A Message-Oriented Knowledge-Base. TR 87-11. ICS Dept, University of California, Irvine (1987).

[WSB87b] Wong, W. C., T. Suda, and L. Bic Performance Analysis of A Message-Oriented Knowledge-Base. In *Int'l Seminar on Performance of Distributed and Parallel Systems IFIP WG 7.3*, Yutaka Takahashi, Ed., North-Holland, Kyoto, 1988.

[WSB90] Wong, W. C., T. Suda, and L. Bic Performance Analysis of A Message-Oriented Knowledge-Base. *IEEE Transactions on Computers 39*, 7 (July, 1990).

[YSI86] Yokota, H., K. Sakai, and H. Itoh Deductive Database System based on Unit Resolution. In *Proc. Conf. on Data Engineering, IEEE*, 1986, pp. 228–235.

[ZAN87] Special Issue on Database and Logic,. In *IEEE Data Engineering*, Zaniolo, C., Ed., Vol. *10:4*, 1987.

[ZLO75] Zloof, M.M. Query by Example. In *Proceedings of National Computer Conference*, Anaheim, CA, 1975, pp. 431–437.