EFFICIENT SUPPORT FOR RULES AND DERIVED OBJECTS IN RELATIONAL DATABASE SYSTEMS

by

Eric N. Hanson

Memorandum No. UCB/ERL M87/70

24 August 1987

EFFICIENT SUPPORT FOR RULES AND DERIVED OBJECTS IN RELATIONAL DATABASE SYSTEMS

by

Eric N. Hanson

Memorandum No. UCB/ERL M87/70

24 August 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

This research was supported by the National Science Foundation, Grant DMC-8504633 and by the Defense Advanced Research Projects, Contract N00039-84-C-0089.

EFFICIENT SUPPORT FOR RULES AND DERIVED OBJECTS IN RELATIONAL DATABASE SYSTEMS

by

Eric N. Hanson

Memorandum No. UCB/ERL M87/70

24 August 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

This research was supported by the National Science Foundation, Grant DMC-8504633 and by the Defense Advanced Research Projects, Contract N00039-84-C-0089.

Efficient Support for Rules and Derived Objects in Relational Database Systems

Eric N. Hanson

Abstract

This thesis presents the design and analysis of a collection of algorithms to support triggers, inference rules, and derived data objects (e.g. views) in relational database systems. A basic component of algorithms for testing rule conditions is known as *rule indexing*. Given a collection of rule conditions and a database record, a rule indexing algorithm finds all the conditions that are satisfied by that record. A rule indexing technique called *basic locking* has been previously proposed. Basic locking is known as a *lock-based* algorithm because it places special locks on data records and in conventional indexes. Two other lock-based rule indexing methods, *reduced basic locking* and *mark intersection* are proposed here, and the performance all three algorithms is analyzed.

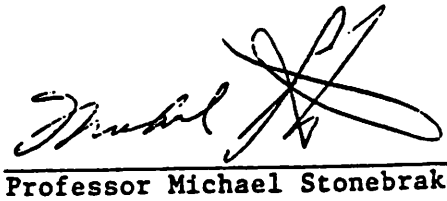
A *view maintenance algorithm* is a method for maintaining and incrementally updating a physically stored copy of a database view. A new view maintenance algorithm called *Rete view maintenance* (RVM) is proposed in this thesis. RVM is based on the Rete Network, a type of discrimination network used to test rule conditions in forward-chaining rule interpreters. Methods are discussed for improving the performance of view maintenance algorithms by utilizing rule indexing techniques. A collection of algorithms is also proposed to allow maintenance of materialized aggregates and aggregate functions.

By keeping a stored copy of a view up-to-date using a view maintenance algorithm, it is possible to process view queries directly using the copy. The conventional way to process queries against views is to use *query modification*, whereby a view query is translated into an equivalent

query that refers only to the base relations. A performance analysis is presented which compares the average cost of a view query for these two alternatives for different view types, including a simple selection from one relation, the join of two relations, and an aggregate over one relation.

A related performance analysis is also presented comparing the costs of different algorithms for querying *database procedures*. The database procedures analyzed are made up of one or more database queries stored in the field of a record. The value of a database procedure is the result of executing the query or queries in its definition. Three different algorithms for processing queries against database procedures are evaluated. The first algorithm is to always execute the queries in the procedure. The second algorithm requires caching the last value returned by executing the queries in the procedure; if the cached value is valid when the procedure is queried, the value from the cache is returned. Otherwise, the procedure value is recomputed, and written to refresh the cache. The third algorithm is to use a view maintenance method to keep a stored copy of the procedure result up-to-date at all times, and return the result whenever it is requested. As in the case for views, the average query cost for each algorithm is compared.

Finally, enhancements to the rule sublanguage of the POSTGRES database management system are proposed to increase the power of the language and to simplify implementation of rule-based applications. Methods are presented for implementing the new language features efficiently using rule indexing and view maintenance techniques.



Professor Michael Stonebraker
Committee Chairman

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Michael Stonebraker, for giving me the opportunity to work in the field of database management, and for providing indispensable guidance for this research. I also wish to thank the other members of my thesis committee, Professors Lawrence A. Rowe and William Cooper, for reviewing this dissertation. The other graduate students in the database research group at UC Berkeley, including Margaret Butler, Brad Rubenstein, Timos Sellis and Yiannis Ioannidis provided helpful comments on this work. Careful editing by Kaaren Bock greatly improved chapter 1, and her stylistic comments helped me present the other chapters more effectively. Finally, I thank Jeff Anton, our system manager, for keeping our computers running smoothly throughout the course of my research.

Table of Contents

List of Figures

CHAPTER 1

INTRODUCTION

1.1. Background

Many different applications require the ability to specify rules about information stored in a database management system. In partial response to this need, commercially available database systems contain several specialized rule mechanisms. Two examples are *protection subsystems*, which support rules that allow or dissallow certain database actions such as reading or updating part of a relation, and *integrity control subsystems*, which permit the user to define logical assertions that a database is required to satisfy. Proposals for future database systems include more sophisticated rule systems, including *triggers* and *inference rules*. A trigger is a condition and an associated action to be executed if database updates cause the condition to become true. An inference rule specifies how to derive new information from existing facts when querying the database.

Another important class of database facilities involves support for *derived objects*. Derived objects are computed from other data stored in the database. A simple example of a derived object is a database view. Another type of derived object is the result of a collection of database commands, also known as a *database procedure* [SAH84, SAH85]. A third type of derived object is an *aggregate*, which is a function that takes a set of values as input, and returns a single value. Two commonly used aggregates are *sum* and *average*. The use of views and database procedures can simplify access to data because they free the user from specifying complicated queries. Using derived objects can also improve performance of applications if the values of frequently accessed objects are computed in advance and stored. Rules and derived objects are closely related because the condition of a rule (e.g., a trigger) has a structure similar to the definition of a derived object (e.g., a view).

Facilities to support rules and derived objects in a database management system (DBMS) should be efficient, and the DBMS should still provide good performance for queries and updates. Therefore, the design and implementation of a system to support rules and derived objects in a DBMS deserves careful attention. The subject of this thesis is efficient support for rules and derived objects in a database environment. The remainder of this chapter presents a review of previous research related to rules and derived objects, and an outline of the dissertation.

1.2. Research on Rule Systems

In the past 15 years, there has been a large amount of research related to rule-base systems. This work is divided between the fields of database management and artificial intelligence. Section 1.2.1 is a brief review of the work done in the database community. Section 1.2.2 covers the related work from artificial intelligence.

1.2.1. Database Rule Systems

There have been several proposals for increasing the power of the rules systems in database managers. The most important work relative to this thesis focuses on database triggers and *alerters* (an alerter is a trigger that sends a message to a user or application program as its action). The flow of execution of triggers is called *forward chaining*, because the action of one rule can change the database, causing another rule to execute. A separate area of research focuses on *deductive inference rules*. Deductive inference follows a pattern of execution known as *backward chaining* since rules are processed in reverse. For example, consider the following collection of facts and rules, where "$\rightarrow$" represents logical implication.

$$A$$
$$A \rightarrow B$$
$$B \rightarrow C$$

To see if C is true using backward chaining, the system attempts to show that B is true. This in

turn leads to an attempt to show that A is true. Since A is stored in the database, it is true, so the system concludes that C is true.

Section 1.2.1.1 discusses previous work on database triggers and alerters and section 1.2.1.2 covers research on deductive inference in databases.

1.2.1.1. Triggers and Alerters

The first work on database triggers was Eswaran's consideration of the impact of such rules on other database functions, including authorization and concurrency control [Esw76]. He concluded that since triggers can be defined to monitor database activity and copy data, they should execute with no more privileges than the person who created them and they should be carefully monitored by the database administrator (DBA). He also showed that to maintain serializability of transactions, the database concurrency control mechanism must treat reads and writes performed by triggers as part of the transaction that caused the trigger to execute.

Later work by Buneman and Clemons discusses two classes of triggers identified as *simple* and *complex* [BuC79]. Simple triggers are those with conditions based on a single tuple from one relation. For example, consider an employee-department database with the following schema:

EMP(name, age, salary, dept, job)
DEPT(dname, building, floor)

An example of a simple trigger is the following:

If an employee's salary is greater than 100,000 dollars
then append his name to the HIGH-ROLLER relation

Complex triggers have conditions based on two or more relations. These conditions have the same structure as relational algebra expressions. One can also think of a complex trigger condition as the definition of a database view. The condition becomes true if the database changes so that the view contains a new tuple. An example of a complex trigger is:

if a new name enters the view B23EMPS, which has the definition

```
define view B23EMPS (EMP.name)
where EMP.dept = DEPT.dname
and DEPT.bldg = "B23"
```

then send the employee with that name the fire safety instructions for building B23.

Buneman and Clemons envision *add* and *delete* triggers that are awakened when a tuple logically enters or leaves a database view, respectively [BuC79]. The complex trigger above is an add trigger. The triggering mechanism they propose requires recomputing the view after each update. The new value of the view is then compared to the old one to see whether or not to execute the rule. Because recomputing the view is expensive, they developed a theorem-prover that attempts to show when an update command cannot cause a view to change by examining the command *before* execution. The view is recomputed only if the theorem-prover determines that an update might have caused it to change. The theorem-prover is based in part on the notion of *readily ignorable updates* or RIU's. An RIU is an update that can be determined in advance not to affect the view used as the condition of a rule. For example, any update to the age field of EMP is an RIU with respect to a rule condition referring only to the salary field.

Most other work on triggers focuses on rules of the simple type. Chang proposes a way to use chains of two or more simple alerters to achieve the affect of a complex alerter [Cha82]. Simple triggers were proposed for the experimental relational DBMS *System R,* but never implemented [ABC76, CAE76]. A new commercial system from Sybase provides a full implementation of a restricted form of simple triggers [How86]. As opposed to complex triggers, Sybase trigger conditions can specify only whether a record was inserted into or deleted from a relation. Yet the action of a Sybase trigger can be a general program written in TRANSACT-SQL, an extended version of IBM's SQL query language [CAE76] provided by Sybase. TRANSACT-SQL programs can contain any SQL statement, as well as commands to support the following:

- control flow (WHILE, IF-ELSE, GOTO)
- transaction handling (BEGIN TRAN, COMMIT TRAN, ROLLBACK TRAN, SAVE TRAN)
- output and error handling (PRINT, RAISERROR)
- setting an interval or absolute timer (WAITFOR)

Stonebraker proposed a form of complex triggers quite different from those envisioned by Buneman and Clemons [Sto85]. These rules can be formed by tagging any command in the QUEL query language [HSW75] with the keyword **always**. Commands tagged with **always** logically appear to run indefinitely. A trigger to force Fred's salary to always be equal to Sam's salary can be expressed as follows:

```
always replace EMP (salary = E.salary)
from E in EMP
where EMP.name = "Fred"
and E.name = "Sam"
```

(The **from** clause above defines a tuple variable E over EMP in the new syntax used by the POSTGRES system [StR86].) Given this rule, whenever a command such as

```
replace EMP (salary = 1000) where EMP.name = "Sam"
```

is processed, the trigger should be awakened to update Fred's salary.

In general, the system must store a collection of triggers:

T_1: **always** <*command*> relname-1 (Target-list-1) **where** PREDICATE-1
.
.
.
T_n: **always** <*command*> relname-n (Target-list-n) **where** PREDICATE-n

When a user update U is processed, the system must find all triggers T_i for which there exists a tuple t modified or inserted by U that might cause T_i to have an effect. A trigger T_i definitely does not have to execute unless the following conditions are satisfied:

t satisfies PREDICATE-i
and
the fields of t changed by the update command contain an attribute
that appears in Target-list-i or PREDICATE-i

Under some circumstances, the system may awaken a trigger T_i even when the condition above is not satisfied. Unnecessary rule activations of this form are called *false drops*. This is not a problem because the semantics of **always** rules are designed so that they can be executed more often than necessary without changing the database. For example, the rule for setting Fred's salary equal to Sam's salary that was presented previously can be run any number of extra times since it will overwrite Fred's salary field with an equal value, leaving the database in the correct state. The only penalty for finding false drops is that time is wasted processing a trigger which does not actually do anything. A implementation of **always** rules is being undertaken as part of the POSTGRES system [SHP87].

An important class of rules are known as *integrity constraints*. Integrity constraints are a special case of triggers. The purpose of an integrity constraint is to ensure that the database meets some condition. A simple type of integrity constraint called *value integrity* specifies that the data values in a relation must meet some condition. An example of a value integrity constraint is the following:

> The value of EMP.salary in each record must be between 10,000 and 50,000 dollars.

A more complex type of integrity constraint is *referential integrity* [Dat81a]. A referential integrity constraint specifies that if a domain D supplies primary key values for a relation R_1, and values from D appear in attribute A of a relation R_2, then any value, say d, from domain D which appears in R_2 must be the primary key of some record in R_1. A typical example of a referential integrity constraint is the following:

> No EMP record may have a dept field value that is not the dname value of some record in DEPT (in other words, no employee may work in a department that does not exist).

A rule to enforce an integrity constraint must determine that the constraint has been violated, and then take some action, such as aborting the current command, or refusing to accept a partic-

ular record. The fact that integrity constraints must act in response to changes to the database made by updates explains why integrity constraints are a special case of triggers.

1.2.1.2. Deductive Databases

A growing body of research focuses on extending databases with a logical inferencing capability similar to that found in the logic programming language PROLOG [Bra86, GaM78, GMN81]. Addition of inference rules to the database in these proposals is normally presented as an extension of the database view system, allowing views to be defined using a collection of possibly recursive rules called Horn clauses [End72]. Processing a collection of inference rules for a large database is potentially very expensive. Past research has considered methods to optimize recursive queries that arise in databases which have been extended with recursive inference rules [Ioa85, Ull85, Zan85, Zan86].

As an example of recursive inference, consider the relation

parent-of (parent, child)

Given parent-of, a view

ancestor (name, descendent)

can be defined recursively using the following rules:

```
define view ancestor (name=parent-of.parent, descendent=parent-of.child)

define view ancestor (name=ancestor.name, descendent=parent-of.child)
where ancestor.descendent=parent-of.parent
```

Given this definition one can query the ancestor view directly. For example, the following query retrieves the names of all the ancestors of "Bob."

```
retrieve (ancestor.name)
where ancestor.descendent = "Bob"
```

This query is implicitly recursive because ancestor is defined using a recursive rule.

Other research has focused on productive ways to couple databases with logic programming systems. Jarke et al. developed an optimizing PROLOG front-end to an SQL-based relational database system [CAE76, JCV84]. This system allows users to query the underlying relational database using PROLOG. It attempts to optimize performance by

1. caching the results of previous SQL queries in the front-end,
thus limiting the number of query evaluations performed by the back-end.

2. allowing the back-end to execute all operations it is capable of performing,
such as testing simple predicates.

3. using semantic query optimization techniques [ASU79].

Additional semantic optimization methods that can be applied in the front-end are discussed in [Jar86]. Sciore and Warren advocate a tighter coupling between PROLOG and a DBMS to improve efficiency and facilitate sharing of data [ScW86]. They propose building into the PROLOG runtime system components of the DBMS including disk and buffer management, indexing, query optimization, and concurrency control.

Another form of inference rule is described in the proposal for **always** rules in POSTGRES [Sto85, SHP87]. The POSTGRES rule manager chooses as an optimization whether or not to use forward or backward chaining to process an **always** rule. If forward chaining is selected, an **always** rule is a trigger (forward chaining is called *early* evaluation). If backward chaining is selected, it is an inference rule (backward chaining is called *late* evaluation). Early rules are processed as described in the discussion above on the use of **always** rules as triggers. Late rules are processed by modifying the query when data is retrieved. For example, consider the following rule, which is assumed to be designated late by the system:

always replace EMP (salary = E.salary)
from E **in** EMP
where EMP.name = "Bob" **and** E.name = "Jim"

Suppose the following query is submitted:

```
retrieve (EMP.salary) where EMP.name = "Bob"
```

When this query executes, the system notices that the rule above might affect the salary of Bob. The system then substitutes Bob's tuple into the rule, forming the following subquery:

```
retrieve (salary = E.salary)
from E in EMP
where "Bob" = "Bob" and E.name = "Jim"
```

This subquery is run, and the value it returns is used as Bob's salary.

The proposal for **always** rules supports the use of rule priorities to resolve conflicts between rules. Any **always** rule can be given a priority, which is a real number in the range [0,1]. The effective value of any field is the one assigned by the highest priority rule. This is accomplished by attaching a priority to each field of a tuple written by a rule. For example, a possible collection of prioritized rules is:

```
always replace .5 EMP (salary = 1000)
where EMP.name = "Sam"

always replace .5 EMP (salary = E.salary)
from E in EMP
where EMP.name = "Fred"
and E.name = "Sam"

always replace .7 EMP (salary = 2000)
where EMP.status = "mgr"
```

Given these rules, the salary of Fred will normally be 1000. However, if Fred is promoted from a worker to a manager the priority mechanism causes his salary to be changed from 1000 to 2000.

Priority situations are common in AI applications and are straightforwardly supported in the scheme of [Sto85,SHP87]. Priorities are difficult to handle in rule systems that utilize the view processing system for inference (see [Ioa86,Ull85]). This difficulty arises since if multiple rules are used to define a single view, the view by definition contains the *union* of all tuples derived using those rules. In priority situations, the union of all tuples is not the desired outcome. The value of

each field of a tuple should be the one designated by the highest priority rule. Any attempt to add priorities to a view mechanism would require some yet-to-be-devised *ad hoc* method for merging multiple tuples retrieved using the view definition.

1.2.2. Rule Systems In Artificial Intelligence

A subset of the research done on AI programming environments focuses on production rule systems, which are closely related to database triggers. An early production rule system known as OPS5 [For81] combines a forward chaining rule execution engine with a database, called the *working memory,* that consists of relations with named attributes. The database resides entirely in virtual memory, and is valid only during the execution of a single OPS5 program. In OPS5, relations are created using a **literalize** statement, e.g.

```
(literalize emp name age salary mgr)
```

creates a relation "emp" with the attributes shown. Rules are defined using the **p** (production) statement. Each rule has a name, a condition, and an action. For example, an OPS5 rule "zerorule" that will zero the salary of any employee currently earning more than 50,000 dollars is specified as follows:

```
(p zerorule
  (emp ^salary > 50000)
  -->
  (modify 1 ^salary 0))
```

The effect of statement (**modify** 1 ^salary 0) is to write 0 into the salary field of the data item matching condition element number 1. In general, rule conditions can contain one or more negated or non-negated elements. Negated conditions are signified by placing a "-" in front of a term. Rule actions can contain one or more of the following statments:

modify update one or more fields of a data element matching a condition term

remove	delete a data element matching a condition term
make	insert a new data element into working memory
call	call a user defined procedure

An example of a more complex rule is the following, which deletes employees who earn more than their manager, and places their names in the table "delemp":

```
(p salaryrule
 (emp ^salary <x> ^mgr <y> ^name <z>)
 (emp ^salary < <x> ^name <y>)
 -->
 (make delemp ^name <z>)
 (remove 1))
```

Joins between more than one tuple in the condition are specified with pattern-matching variables such as <x>, <y> and <z> above. The first appearance of a variable will match any value, and that value is bound to the variable. When the variable appears later, it has the value that was bound to it when it first occurred. When the above rule fires, the emp record of an employee earning more than his or her manager matches condition element number 1, and the manager's emp record matches condition element number 2. The effect of the **make** statement in the action of the rule is to create a "delemp" record with a name field containing the value currently bound to the variable <z> (the employee's name). The **remove** statement then deletes the record matching condition element number 1 (the employee's record).

In OPS5, rules are *tuple-oriented,* meaning that they are awakened and run for each unique combination of data elements that match the condition. With the example rule above, the net outcome would be affected by the order in which tuples were deleted. In a database environment, it would be preferred to process a set of tuples during execution of a single rule. Processing a set of tuples at one time would be more efficient that processing tuples individually since database query processors are tuned to process large sets of tuples. It would also avoid order-dependent outcomes that can occur when rules are fired once for each qualifying tuple.

Frame-based AI languages including FRL [RoG77], KRL [BoW77], KEE [FiK85], and ART [Sho87] also provide rule processing capability. A *frame* is simply a record with a collection of named fields or *slots*. In frame-based systems, an *is-a* hierarchy connecting the frames provides a simple but very efficient form of inference, with a built-in priority mechanism. To determine the value of a slot in a particular frame, one first looks at that frame. If it contains a value in that slot, that value is returned. If it does not contain a value, the frames above the current frame in the is-a hierarchy are examined. For example, one frame for "vehicle" might have a slot "has-wheels" containing the value TRUE. Another frame for "truck" containing no value for "has-wheels" could be connected to "vehicle" using an is-a link. A third frame for "boat" (also connected to "vehicle") might have a "has-wheels" slot containing FALSE. One could infer that a truck has wheels by simply following the link to the vehicle frame. However, the value "has-wheels" = FALSE in the "boat" frame would over-ride the default value TRUE found in the "vehicle" frame.

Frame-based systems also provide *procedural attachments* that allow triggering of a *demon* procedure when a frame slot is read or written [Min75]. The more advanced frame-based systems including KEE and ART have built-in forward and backward chaining rule processors that use the frame hierarchy as the database of facts. The forward chaining mechanism is otherwise similar to OPS5, and the backward chaining mechanism is similar to that found in the logic programming language PROLOG [Bra86].

Some recent work in AI has focused on issues of large knowledge bases, and the storage of data objects on secondary storage. Examples are work on persistent LISP [But86,Mis84], and methods for storing AI reasoning knowledge in a database system [DeF86,FWA86].

1.3. Rule Indexing

Rule-based applications require the ability to determine efficiently which rule conditions match a given tuple or collection of tuples. Algorithms for performing this task are called *rule indexing* methods [SSH86]. Rule indexing can be applied in a database system to determine

1. when to awaken a trigger,
2. when an inference rule should be applied, and
3. when the precomputed value of a derived object (e.g., a database procedure) should be invalidated or refreshed.

Below, in section 3.1, a collection of rule indexing techniques are discussed which work for rule conditions that are selections from a single relation. Section 3.2 discusses rule indexing techniques for handling conditions that contain joins.

1.3.1. Rule Indexing for Selection Predicates

Selection predicates on a single relation R are typically boolean combinations of terms of the form

expression **relop** *constant*

where *expression* is of the form *R.attribute*, or is a function of one or more attributes of R, and **relop** is one of $\{<,>,\leq,\geq,=,\neq\}$. The following are some examples of selection predicates of this type on the relation EMP:

EMP.name = "Bob"
EMP.salary $\leq$ 50000 **and** EMP.salary $\geq$ 30000
EMP.salary / EMP.age > 1000

In general, the problem of rule indexing for selection predicates is to determine for each predicate P_i, for $1 \leq i \leq M$, the subset S_i of the tuples in a relation R such that each tuple in S_i matches P_i, and no other tuples in R match P_i. Several approaches to solving this rule indexing problem have been suggested. Using a brute force solution, every inserted or deleted data element must be interpreted against all predicates $P_1 \cdots P_M$. Obviously, if M is not small this algorithm has

serious performance problems. A slightly more sophisticated rule indexing scheme combines interpretation with indexing on one or more terms of the predicate. In this scheme, an index (e.g., a hash table) is created using terms extracted from the predicates as keys. An example of a term that might be extracted is one of the form <attribute=constant>. For each data element inserted into or deleted from the database, the index is searched to find potentially matching predicates. The element is then interpreted against all predicates found by the search. This algorithm performs reasonably well when the number of rules indexed under most keys is small. If the number becomes large, performance degrades since many predicates will have to be interpreted after each insertion or deletion of a data element.

Another method for rule indexing is called *predicate indexing* [SSH86]. Predicate indexing is based on an R-tree storage structure, a multi-dimensional extension of the B-tree designed for indexing spatial data (which must be represented as rectangular regions in an N-dimensional space). For a full discussion of algorithms for manipulating and searching an R-tree, the reader is refered to the original paper on R-trees [Gut84]. One property of the R-tree is that makes it possible to find all the rectangular regions indexed that contain a particular point efficiently. In the predicate indexing algorithm, the predicates to be indexed must be represented as rectangular regions in an N-dimensional space. This space is defined by a relation A with N attributes (the predicates are defined on A). Predicate indexing operates by first building an R-tree index on the set of predicates $P_1 \cdots P_M$ in question. Note that a tuple t in relation A represents a point in the N-dimensional space defined by A. Hence, finding all the predicates that match t can be done quickly by searching the R-tree. For example, consider a relation

R(x,y)

and the following predicates on R:

P_1: $0 \leq x \leq 7$ and $0 \leq y \leq 5$
P_2: $3 \leq x \leq 10$ and $3 \leq y \leq 8$
P_3: $5 \leq x \leq 9$ and $1 \leq y \leq 10$

These predicates can be represented rectangles, as shown in figure 1.1. Consider a tuple t in R with the value <x=6,y=2>. By examining figure 1.1, it can bee seen that the rectangles for P_1 and P_3 contain the point for t, but P_2 does not overlap the point. Hence, t satisfies P_1 and P_3 but not P_2. The R-tree provides an efficient rule indexing mechanism for predicates that can be represented as rectangular regions because it makes it possible to determine quickly which rectangles contain a point.

A rule indexing method known as *basic locking* [Sto85] is related to locking methods used for database concurrency control [Gra78]. Basic locking utilizes special persistent locks, called *trigger*

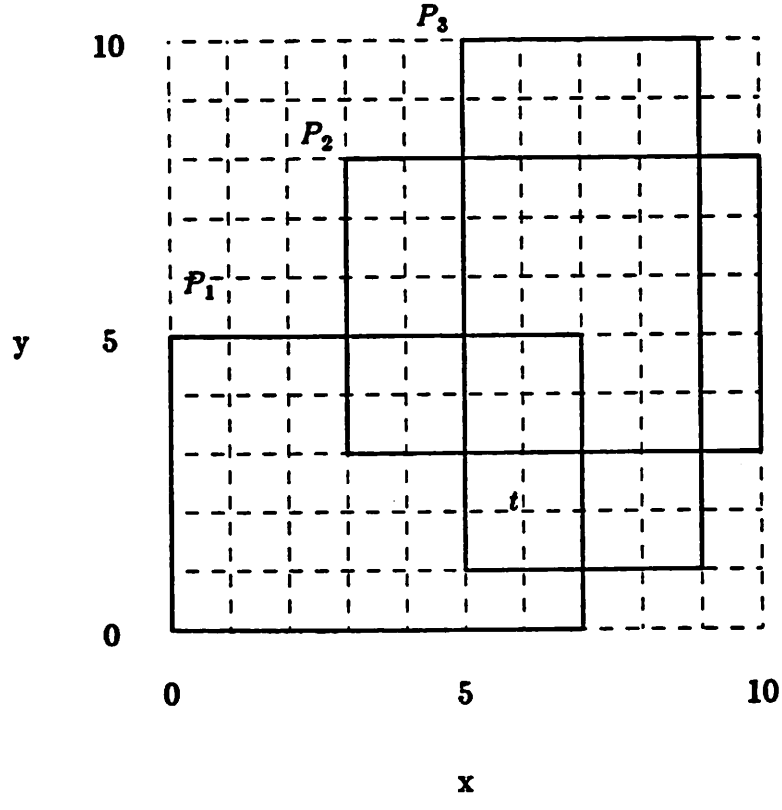


Figure 1.1. Representation of predicates as rectangles

locks or t-locks, which are placed on data records and in conventional indexes. The scheme utilizes a relation

RULES(id, name, rule-def)

to hold the rule base. The "id" field contains a unique identifier for the rule, "name" contains an optional user-supplied name for the rule, and "rule-def" contains the rule definition, including its predicate.

For each rule defined, an access plan is constructed using a conventional query optimizer [Sel79]. This plan is executed and each tuple it reads is marked with a t-lock that contains the "id" of the predicate. If a sequential scan of the relation is used, then all tuples in the relation will be marked. In this case, conventional lock escalation will convert record locks to a relation lock. Otherwise, an index will be used for access and t-locks will be set on data records *and* on the key interval inspected in the index. Such index interval locks are required to deal correctly with insertion of new records, as explained momentarily.

If a tuple x is inserted, then the collection of markers must be found for the new tuple. As a result of the insertion, values will be inserted into all indexes on the relation. If such a value is covered by a key-range lock, then a corresponding t-lock will be added to the data tuple containing the value.

To find the collection of predicates that cover a tuple x, one first collects all the t-locks on x. The locks include the id's of rules that might match the tuple. Since these t-locks represent a superset of the predicates that actually match the tuple, relevant tuples in the RULES relation must be checked to determine whether x actually satisfies each predicate.

For example, assuming that there is a B-tree index on EMP.salary and no index on EMP.age, the qualification:

$$Q\text{: EMP.salary} = 1000 \text{ and EMP.age} > 30$$

will set t-locks in the salary index and on all data records that it reads (i.e., those with salary = 1000). Supposed the following tuple is inserted into EMP:

x: <name="Jane", age=35, salary=1000, job="Salesperson", dept="Sales">

When x is inserted, the salary index must be updated to point to x. Since the salary field has value 1000, x will conflict with the t-lock set in the salary index for Q. A copy of this t-lock (and possibly others) will be stored directly on x.

Not all predicates indicated by the t-locks a tuple will necessarily match the tuple. The reason that a superset of the tuples that match each predicate must be locked is that a non-indexed attribute may be modified so that a record matches a predicate it did not match before the update. For example, the age field of an employee record may be updated from 30 to 31. Since there is no secondary index on age, the basic algorithm would have no way of discovering that it should now be marked without searching the salary index. This search should be avoided if only age is updated (in database systems such as INGRES [SWK76] and System R [ABC76], it is not necessary to read or write any index pages to update a non-indexed field correctly). Because of this problem, t-locks must be set on all tuples that *potentially* satisfy a predicate based on the interval locks the predicate has set in *one* index. In effect, key interval locks in the indexes are used to implement predicate locks to determine whether new tuples conflict with existing rules. Setting t-locks on index intervals is analogous to the use of predicate locks to solve the problem of *phantoms* in concurrency control [EGL76].

This strategy is called basic locking because it sets t-locks on all objects for which a normal query would set read or write locks. It requires no changes to conventional execution of access plans, so it can be properly called a locking mechanism. The advantage of this scheme is that it is closely coupled to normal query processing. New qualifications can be added using normal facilities, and locks for new tuples are found as byproducts of normal update processing.

Both R-tree-based predicate indexing and basic locking are designed for predicates on only one relation. Both algorithms can be extended to partially index predicates that contain joins. This is done by indexing separately all single-relation selection predicates that appear in a join predicate. However, this is only a partial solution to the problem of indexing join predicates. In the next subsection, some rule indexing algorithms are described which support predicates that contain joins.

1.3.2. Rule Indexing Techniques for Join Predicates

A form of indexing used in production rule systems for AI programming is the *Rete Match Algorithm* [For82]. Rete Match was invented for OPS5, and is also used in OPS83 [For84] and ART [Gev87,Sho87]. The algorithm utilizes a data structure called a *Rete network* that eliminates the interpretation step of brute force and indexing methods, and also handles join predicates naturally. Direct interpretation of patterns is avoided by compiling all the patterns together in advance to form a network. The Rete Match Algorithm maintains a *conflict set* showing each pattern and the data elements that match it. Changes to the database are represented by *tokens* which are tuple values tagged with a "+" or a "-" to show whether the tuple was inserted or deleted, respectively. Modifications are treated as deletions followed by insertions. The network can be viewed as a black box that receives tokens as input, and outputs changes to the conflict set, as shown in Figure 1.2.

The output of the Rete network compiler is a discrimination network containing the following types of nodes:

- **root** node: The single root node receives all tokens input to the net, and broadcasts the tokens to all successors.
- **T-const** nodes: These nodes test input tokens for simple conditions of the form

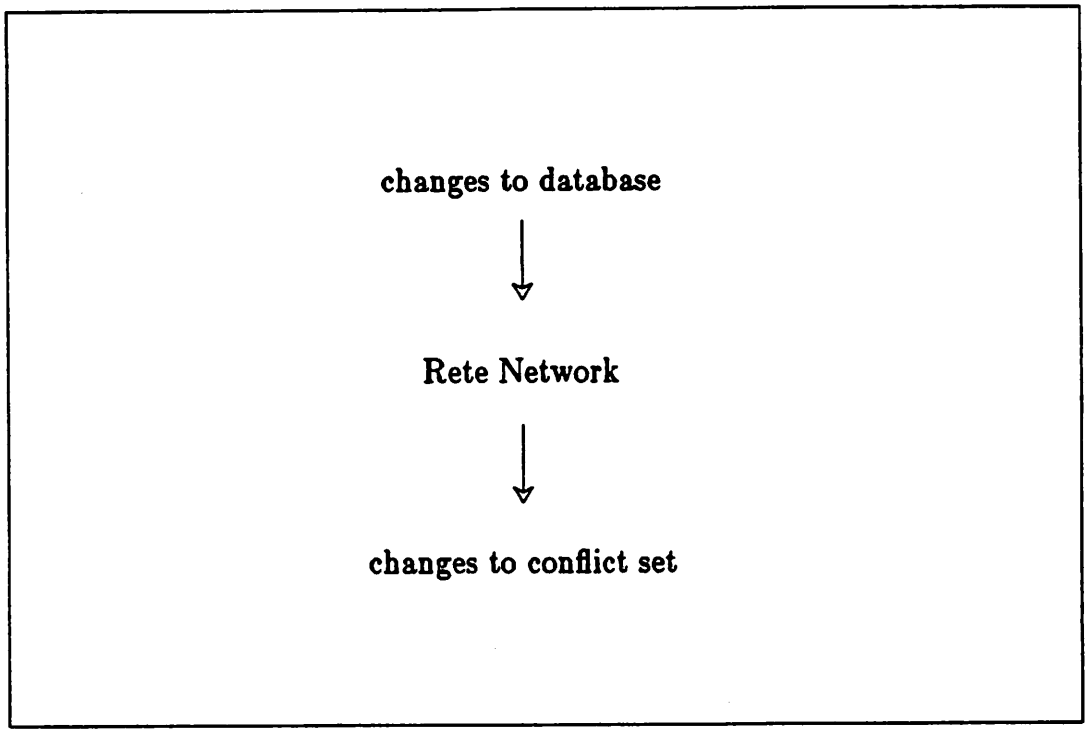


Figure 1.2. Function of the Rete Network

attribute *operator* constant

where the operator can be one of $\{<,>,\leq,\geq,=,\neq\}$. All tokens that pass the test are passed on to the successors of the **T-const** node. Tokens that do not pass the test are discarded.

- **α-memory** nodes: These nodes serve to hold the output of **T-const** nodes. A token input to an **α-memory** node containing a "+" tag is added to the memory. A token with a "–" tag is deleted from the memory.
- **and** Nodes: These nodes specify joins of the form

  left-input.attribute *operator* right-input.attribute

The left and right inputs of an **and** node are memory nodes.

• **not** nodes: These nodes are used to implement negated conditions. They are similar to **and** nodes, except that they keep reference counts with tokens in the left memory showing how many tokens they match in right memory. If the reference count is zero, the token is forwarded to the successor nodes of the **not** node.

• **β-memory** nodes: These nodes hold the output of **and** nodes and **not** nodes.

• **P** nodes: One of these nodes is associated with each rule. If a token with a "+" tag makes it to a **P** node, a pair consisting of that rule and token is added to the conflict set.

The compiler recognizes common subexpressions in rule conditions and generates shared nodes to improve efficiency. Consider as an example a Rete network constructed for the following two OPS5 rules:

```
; delete Bob if he works on the first floor
(p rule1
  (emp ^name Bob ^dept <x>)
  (dept ^dname <x> ^floor 1)
  -->
  (remove 1))

; delete every programmer who works on the first floor
(p rule2
  (emp ^job Programmer ^dept <x>)
  (dept ^dname <x> ^floor 1)
  -->
  (remove 1))
```

The network for these two rules is shown in Figure 1.3. This network has shared nodes for the condition term

```
(dept ^dname <x> ^floor 1)
```

that appears in both rules.

As an example of how the Rete network operates, suppose that the following tuple is inserted into EMP:

T: <name="Jack", age=28, job="Programmer", salary = 30000, dept="Engineering">

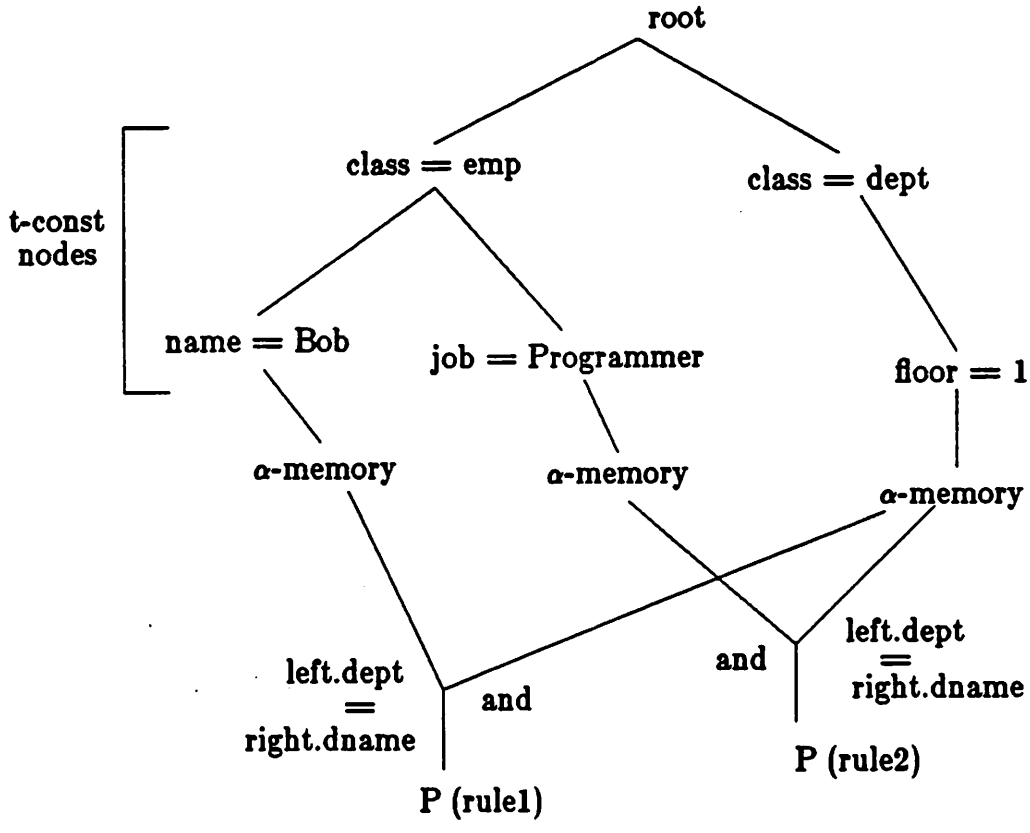


Figure 1.3. Example Rete Network

The tuple T is first placed at the root node of the network. Assume T is first passed to the node labeled "class=dept." T does not meet the qualification of that node, so T is discarded there. T is then passed to the node labeled "class=emp." T meets this qualification since it is in the EMP relation, so T is passed on to both successors. Suppose T is first passed to the node labeled "name=Bob." T does not meet this qualification, so it is discarded at that node. T is then passed to the node labeled "job=Programmer." T meets this condition, so it is passed on to the α-memory following the node labeled "job=Programmer". A copy of T is stored in the α-memory, and T is passed to the subsequent and node. The opposite α-memory is then consulted to see if T joins with a tuple there. Assuming that there is a tuple

S: <dname="Engineering", floor = 1>

in the opposite α-memory, a new token T' is constructed that contains both S and T. T' is passed on to the P node for rule2. In OPS5, this causes an *instantiation* of rule2 to be made eligible for execution. This instantiation consists of a pair containing rule2 and the token T' (other instantiations of rule2 for different tokens may be also be eligible for execution).

It is important to note that the Rete network has the same structure as a relational database query plan [Sel79]; the **T-const** nodes correspond to scans, and the **and** nodes correspond to joins. The discrimination network output by a Rete network compiler contains a fixed query processing plan for all the patterns. In current expert system shells, including OPS5, OPS83 and ART, this plan is constructed using heuristics, and is not optimized with regard to the database.

The Rete network described above is a fairly sophisticated and efficient form of rule indexing for predicates that may contain joins. Using the Rete algorithm, only an incremental amount of computation is necessary each time a tuple is inserted or deleted. The work on database triggers by Buneman and Clemons describes a more primative type of rule indexing for predicates that may contain joins [BuC79]. This method has efficiency problems for two reasons:

1. before *every* database update command the system must verify whether each rule will be affected, and

2. if a rule condition based on some view V *is* affected, V must be recomputed, even though only a small fraction of it may have changed.

Another algorithm proposed for incremental update of materialized database views can also be used for rule indexing, and it automatically handles predicates with joins [BLT86]. This algorithm will be called *algebraic view maintenance* (AVM). One can think of a relational algebra expression defining a view as a database predicate. Whenever a new tuple enters a view, it means that the tuple satisfies the equivalent predicate. AVM works by maintaining a materialized copy of the view. As updates change the database, the algorithm incrementally alters the stored view

to reflect the current state of the database. AVM can be used for rule indexing in the following way. Consider a trigger T whose condition is defined by a view V. If V is maintained using AVM, then if a new tuple enters V, T can be made eligible to run.

Input to the AVM algorithm consists of the following sets of tuples:

$R_1, R_2, \ldots R_N$	the N base relations
$A_1, A_2, \ldots A_N$	the N sets of tuples inserted into the base relations by the current transaction
$D_1, D_2, \ldots D_N$	the N sets of tuples deleted from the base relations by the current transaction

The sets $A_1 \cdots A_N$ and $D_1 \cdots D_N$ must contain the *net* changes to the database made by one transaction. Hence, the following conditions must hold:

1. $A_i \bigcap D_i = \phi$
2. $A_i \bigcap R_i = \phi$ for $1 \leq i \leq N$
3. $D_i \subseteq R_i$

Condition 1 must be true since a tuple that is inserted and later deleted is not part of the net change to the relation. Condition 2 must hold since appending a tuple to a relation that already contains an identical tuple does not constitute a net change. Condition 3 must be true because a tuple cannot be part of the net deletions from a relation unless the tuple was previously in the relation.

The definition of a view V can be represented by a *select-project-cross-product* expression, where σ_X represents selection based on a predicate X, π_Y represents projection of the set of attributes Y, and $\times$ represents cross-product, as follows.

$$V = \pi_Y(\sigma_X(R_1 \times R_2 \times \cdots \times R_N))$$

Consider an example with two relations, $R_1(a,b)$ and $R_2(b,c)$, and a view V defined as follows, where $Y=\{a,c\}$ and $X=(R_1.a=5 \text{ and } R_1.b=R_2.b)$:

$$V = \pi_Y(\sigma_X(R_1 \times R_2))$$

The value of V given the original contents of the database will be called V_0. The following expression shows the value of V after an append-only transaction that updates both R_1 and R_2.

$$V_1 = \pi_Y(\sigma_X((R_1 \cup A_1) \times (R_2 \cup A_2)))$$

Selection and projection both distribute over union, so the above expression simplifies as follows:

$$\begin{aligned} V_1 &= \pi_Y(\sigma_X(R_1 \times R_2 \cup A_1 \times R_2 \cup R_1 \times A_2 \cup A_1 \times A_2)) \\ &= \pi_Y(\sigma_X(R_1 \times R_2)) \cup \pi_Y(\sigma_X(A_1 \times R_2)) \cup \pi_Y(\sigma_X(R_1 \times A_2)) \cup \pi_Y(\sigma_X(A_1 \times A_2)) \\ &= V_0 \cup \pi_Y(\sigma_X(A_1 \times R_2)) \cup \pi_Y(\sigma_X(R_1 \times A_2)) \cup \pi_Y(\sigma_X(A_1 \times A_2)) \end{aligned}$$

This algebraic simplification shows that V can be refreshed by computing the value of the last three expressions shown above and unioning the results to the stored copy of V (V_0). In practice, the query optimizer can be used to find the most efficient method available for computing these subexpressions.

AVM becomes slightly more complicated when both deletions and insertions occur in transactions. One problem is that tuples in V may have been contributed by more than one source, since the projection operation can map multiple input tuples to the same value. If it appears that a tuple should be deleted from V, but V is stored with duplicates removed, it is impossible to decide what action to take without recomputing V from the base relations. To overcome this difficulty without wasting disk space by physically storing duplicates, each tuple in V must contain a *duplicate count* indicating how many sources contributed the tuple. If an identical value is already stored when a tuple is inserted into V, its duplicate count is incremented. Otherwise the tuple is inserted with a duplicate count of 1. Conversely, the duplicate count of the stored value is decremented when a tuple is deleted. If the count becomes 0, the tuple is physically removed from V.

Extending the previous example, consider a transaction that inserts *and* deletes tuples from both R_1 and R_2. The new version of the view, V_1, is represented as follows:

$$V_1 = \pi_Y(\sigma_X(((R_1 - D_1) \cup A_1) \times ((R_2 - D_2) \cup A_2)))$$

Using

$$R_1' = (R_1 - D_1)$$
$$R_2' = (R_2 - D_2)$$

we can rewrite the formula above as

$$V_1 = \pi_Y(\sigma_X((R_1' \cup A_1) \times (R_2' \cup A_2)))$$

Multiplying out this expression yields

$$V_1 = \pi_Y(\sigma_X(R_1' \times R_2' \cup R_1' \times A_2 \cup A_1 \times R_2' \cup A_1 \times A_2))$$

Expanding the $R_1' \times R_2'$ term of the above gives the following (the remaining terms are indicated by ellipsis):

$$\begin{aligned} V_1 &= \pi_Y(\sigma_X((R_1 - D_1) \times (R_2 - D_2) \cup \cdots)) \\ &= \pi_Y(\sigma_X(R_1 \times (R_2 - D_2) - D_1 \times (R_2 - D_2) \cup \cdots)) \\ &= \pi_Y(\sigma_X(R_1 \times R_2 - R_1 \times D_2 - D_1 \times (R_2 - D_2) \cup \cdots)) \end{aligned}$$

Re-writing the second occurrence of R_1 as $(R_1' \cup D_1)$ gives

$$V_1 = \pi_Y(\sigma_X(R_1 \times R_2 - (R_1' \cup D_1) \times D_2 - D_1 \times (R_2 - D_2) \cup \cdots))$$

Multiplying the second term through, and substituting R_2' for $(R_2 - D_2)$ leaves

$$V_1 = \pi_Y(\sigma_X(R_1 \times R_2 - R_1' \times D_2 - D_1 \times D_2 - D_1 \times R_2' \cup \cdots))$$

The projection operation π has the distributive property for *both* "$-$" and "$\cup$" when these operations are implemented using duplicate counts [BLT86]. Applying these distributive properties to the expression above, we are left with

$$\begin{aligned} V_1 &= \pi_Y(\sigma_X(R_1 \times R_2)) - \pi_Y(\sigma_X(R_1' \times D_2)) \cdots \\ &= V_0 - \pi_Y(\sigma_X(R_1' \times D_2)) - \pi_Y(\sigma_X(D_1 \times R_2')) - \pi_Y(\sigma_X(D_1 \times D_2)) \\ &\quad \cup \pi_Y(\sigma_X(R_1' \times A_2)) \cup \pi_Y(\sigma_X(A_1 \times R_2')) \cup \pi_Y(\sigma_X(A_1 \times A_2)) \end{aligned}$$

As expected, the first term of this expression is V_0, the previous stored value of V. To update the stored copy of V so that its value becomes V_1, the remaining expressions must be evaluated and

either inserted into or deleted from V as required, maintaining the correct duplicate counts.

The method presented by Blakeley et al. for determining how to refresh the view when both deletions and insertions occur is slightly different than the one shown here, and is in fact not always correct [BLT86]. Using that scheme, the expression below would be used to refresh the view:

$$V_1 = \pi_Y(\sigma_X(R_1 \times R_2 \cup A_1 \times A_2 \cup A_1 \times R_2 \cup R_1 \times A_2 - D_1 \times D_2 - D_1 \times R_2 - R_1 \times D_2))$$

Using this expression can cause improper update of the duplicate counts. For example, suppose tuples t_1 in R_1 and t_2 in R_2 joined together to produce a result tuple in V_0. If a transaction deleted both t_1 and t_2, then the result of joining t_1 to t_2 would be deleted from V_0 three times, not just once as it should be. This erroneous deletion happens since t_1 is in both R_1 and D_1, and t_2 is in both R_2 and D_2. The formulation given in this paper (using $R_1' = R_1 - D_1$ and $R_2' = R_2 - D_2$) does not have this problem.

As originally described by Blakeley, the algorithm requires that every view definition must be interpreted for every inserted or deleted tuple to see if the view might change. Interpreting every view definition against every tuple may require a significant amount of CPU time if there are many views. Alternatively, to eliminate this costly interpretation step it is straightforward to extend the algebraic view maintenance algorithm with a mechanism similar to basic locking.

Blakeley's original AVM algorithm maintains copies of views just after each update transaction. This procedure, whereby views are maintained as soon as possible, will be called *immediate view maintenance,* or simply *immediate.* An alternative is possible, in which views are maintained only before queries that read data from the view. This scheme will be called *deferred view maintenance,* or *deferred.* This method has been implemented in the ADMS± system [RoK86] for materializing copies of views on workstations attached to a mainframe. In that scheme, the mainframe maintains a shared global database, and workstations update local copies of views when

they process queries. Deferred view maintenance will be discussed in more detail in chapter 4.

This section has discussed rule indexing methods for both single-relation selection predicates and join predicates. Rule indexing and the maintenance of derived database objects are closely related. As an example of this relationship, the rule indexing method proposed by Buneman and Clemons maintains materialized views as an intermediate step [BuC79]. In general, any efficient algorithm for maintaining materialized copies of derived objects (e.g., as algebraic view maintenance) can also serve as a good rule indexing scheme. Support for derived objects is discussed further in the next section.

1.4. Derived Objects

A *derived object* is a value returned by a function applied to the database. The most commonly used type of derived object in relational database systems is the *view*. The standard method for retrieving data from a view is *query modification* [Sto75]. Using this technique, queries to the view are translated into queries on the underlying base relations. A complete materialized copy of the view is never formed. For example, consider the following view definition:

```
define view ED ( EMP.all , DEPT.all )
where EMP.dept = DEPT.dname
```

Suppose the query

```
retrieve (ED.name) where ED.floor = 1
```

is submitted. Query modification would translate this query into the following one that depends only on the base relations EMP and DEPT:

```
retrieve (EMP.name)
where EMP.dept = DEPT.dname
and DEPT.floor = 1
```

The query optimizer will find an efficient method for processing this modified query. AVM is

another method for processing views that was discussed previously in the context of rule indexing [BLT86]. Using this method, views are materialized in advance and maintained incrementally. View queries are processed against the stored copy of the view.

Some other algorithms for maintaining derived objects have also been proposed. A differential view update algorithm similar to AVM is described in [HoT86]. A method for maintaining materialized views that is less general than AVM was presented by Shmueli and Itai [ShI84]. An algorithm that allows maintenance of *database snapshots*, which are copies of views consisting of selections and projections of a single base table, is presented in [AdL80,LHM86].

An algorithm called *caching* allows storage of derived objects [Sel86b,StR86]. To maintain the answer to a QUEL **retrieve** command, the system processes the command normally, locks each record read with a persistent *invalidate lock* or *I-lock* and writes the retrieved value of the object to disk. If a conventional write lock later conflicts with an I-lock, the object for which the I-lock was set is marked as invalid. When an object is read, it will be recomputed and written back to disk only if it was previously invalidated. A possible optimization is to use idle CPU and disk resources to recompute invalidated objects.

Another important class of derived objects are *aggregates*. In QUEL, built-in aggregates can be invoked either as *scalar aggregates* or *aggregate functions* [Eps79,HSW75]. A scalar aggregate can be computed independently from the rest of the query containing it, and will yield a single scalar value as a result. The following is the general form of a scalar aggregate*:

aggregate-operator (*aggregate-expression* [**where** *qualification*])

The *aggregate-operator* may be any one of the built-in aggregate operators (e.g., avg, sum, count, min, and max), *aggregate-expression* may be any legal QUEL expression yielding a scalar value, and *qualification* is a legal QUEL qualification. For example, suppose the relation EMP con-

*In descriptions of syntax, keywords are shown in bold, place holders for expressions are indicated in italics, items enclosed by "[" and "]" are optional, and items are surrounded by "{" and "}" can be repeated zero or more times.

tained the following tuples:

name	dept	salary
Bob	Toy	10,000
Jim	Toy	20,000
Al	Fire	10,000
Susan	Fire	12,000

The scalar aggregate

avg(EMP.salary **where** EMP.dept = "Toy")

returns 15,000 in this case.

Unlike scalar aggregates, aggregate functions return a *set* of values. When an aggregate function is computed the tuples to be aggregated are partitioned on the value of one or more attributes. By convention, the value being aggregated in an aggregate function is called the aggregate expression, and the value determining the partition is called the *by-list*. The general form of an aggregate function is:

aggregate-operator (*aggregate-expression* **by** *by-list* [**where** *qualification*])

The set returned by an aggregate function is represented by a temporary relation with two attributes, one containing an aggregate value, and one containing a by-list value. Continuing the example above, the aggregate function

avg(EMP.salary by EMP.dept)

returns the following temporary relation:

dept	avg
Toy	15,000
Fire	11,000

Scalar aggregates and aggregate functions as described here will be called *general aggregates* because they can be used to compute an aggregate over the result of any query that can be expressed in relational algebra [Cod70].

Aggregation often involves a large amount of computation due to the volume of data that must be processed. Since aggregates are often expensive to compute, but normally require little space to store, they are attractive candidates for caching or differential maintenance. In the context of relational database systems, previous research has analyzed ways to precompute aggregate results and save previous results for future use [BBD82]. These methods do not handle general aggregates. Rather, they can only be used to cache aggregates over a whole relation. Other research has concentrated on ways to maintain derived data, including aggregates, in a functional/binary association data model [KoP81,Pai80]. This approach is based on a program-transformation technique that takes the program text of a transaction procedure as input, and outputs a *derivative procedure.* The derived object is incrementally updated to the correct state after the transaction by executing the derivative procedure. Using this technique, aggregates similar in structure to general aggregates can be maintained. However, the method is not applicable to the relational data model and high level query languages like QUEL and SQL.

1.5. Thesis Overview

The subject of this thesis is efficient support for rules and derived objects in a database management system. Chapter 1 has presented an overview of previous research on rules and derived objects. Chapter 2 presents a collection of physical locking algorithms for rule indexing, and analyzes the time and storage requirements of these schemes. Chapter 3 proposes several variations of AVM, and also a new view maintenance algorithm called *Rete view maintenance* (RVM). It then discusses how any view maintenance algorithm can be applied to the task of maintaining several types of materialized objects, including general aggregates, database procedures, and views and procedures containing aggregates. Chapters 4 and 5 present two separate performance studies of techniques for materializing derived objects. The two chapters focus on views and database procedures, respectively. Chapter 6 discusses the semantics of forward-chaining rules in a DBMS and proposes extensions to the rules system described in [SHP87] to

allow rules with more powerful conditions and actions. Chapter 7 summarizes the results of this work and presents conclusions.

CHAPTER 2

LOCK-BASED RULE INDEXING

2.1. Introduction

A previous paper on rule indexing compared the performance of basic locking and R-tree-based predicate indexing [SSH86]. The paper recognized that when supporting an inference mechanism two alternative strategies are possible: *early* and *late*. The early strategy finds all rules that match a tuple at the time the tuple is inserted or modified and then stores the identifiers of those rules on the tuple. The late strategy finds matching rules when the tuple is retrieved by a query. If updates are frequent, late matching is most efficient. Early matching is preferred when retrievals outnumber updates. Note that the late strategy is only applicable in situations where predicate matching can be delayed until retrieval time. For example, triggers must run immediately after updates that activate them. Thus, early matching must be used to support testing trigger conditions. The issue of whether to perform early or late matching for inference rules depends only on the fraction of operations that are updates. The choice of whether to do matching early or late is independent of the particular rule indexing algorithm used. Thus, the early versus late issue is not considered here.

This chapter presents two lock-based rule indexing methods in addition to basic locking and analyzes the performance characteristics and storage requirements of all three methods. Support for triggers and inference rules is emphasized in the discussion, although rule indexing techniques are also applicable to maintaining materialized views and database procedures. The chapter is organized as follows. Section 2.2 describes the proposed rule-indexing methods. Section 2.3 analyzes the performance of the algorithms based on a simplified model of the database and rules. Section 2.4 gives the results of the performance analysis. Section 2.5 analyzes the memory

requirements of each algorithm. Section 2.6 presents the results of the memory usage analysis. Finally, Section 2.7 summarizes the chapter.

2.2. Rule Indexing Algorithms

The key operation that must be performed when processing triggers or inference rules in a database system is

(A)

> Given a tuple and a set of rule conditions, determine the subset of the conditions that match the tuple.

Since rules are in one-to-one correspondence with conditions, this matching process determines the set of rules that apply to the tuple. Basic locking (BL) is one way to perform operation (A). Two other locking algorithms are presented below. The first, called *mark intersection* (MI), is a generalization of BL that is more efficient in some circumstances because it reduces the number of rules that must be read from the RULES relation. The second, called *reduced basic locking* (RBL), is a modified version of basic locking that requires less memory for locks.

2.2.1. Motivation for Mark Intersection

To simplify the problem of locking data covered by a rule, the basic locking algorithm locks only one term from the rule predicate. For example, suppose that the relation

EMP(name, age, salary, dept, job)

has B-tree indexes on attributes name, dept and job, and no indexes on the other attributes. The following rule has two predicate terms, both of which have indexes:

(B)

```
always replace emp (salary = 30000)
where emp.dept = "Accounting" and  emp.job = "Programmer"
```

However, the basic locking algorithm will choose only one index in which to place locks. As an example, assume that there are 10 different departments and 100 different jobs. Since JOB has the most selective index, JOB will be scanned, and *t*-locks will be set in the JOB index and on all data records with a job title of "Programmer." Now, consider the following update:

(C)

```
append emp (name = "Robert", job = "Programmer",
    dept = "Records", salary = 20000, age = 35)
```

When this update is processed, the index on JOB will be updated to insert the value "Programmer", which will break a *t*-lock for the rule (B) above. At this time, the system knows that the new tuple matches a specific rule on the JOB attribute. In basic locking, the system now has no choice but to retrieve rule (B) from the RULES relation, and check the new tuple against the rule predicate to see if it applies. This "false drop" will incur a cost in both CPU and disk I/O. Since Robert is not in the Accounting department, the rule does not match.

It would be best if the false drop could be avoided in this case. From the definition of the predicate of (B), the system has the information that a tuple must match (B) on *both* JOB *and* DEPT for (B) to be triggered. If it could somehow take advantage of this knowledge, the false drop could be avoided. For example, rather than locking only the JOB index for the value "Programmer," the system could lock the DEPT index for "Accounting" as well. Then, when (C) was executed, a *t*-lock for rule (B) would be broken in the JOB index, but not the DEPT index because the inserted tuple does not have DEPT = "Accounting". If the system knew that any tuple must break a *t*-lock on DEPT *and* JOB to trigger (B), then it could avoid searching into the RULES relation. One way to make this information available is to store it with all *t*-locks for the rule (B). Using this method, a *t*-lock for (B) has the form

<rule-id = A; attributes-to-match = JOB, DEPT>

Then, if a new tuple did not break a *t*-lock for rule (B) on *every* attribute in the *attributes-to-*

match set, the system would know that (B) could not be triggered. The false drop would thus be avoided.

2.2.2. The Mark Intersection Algorithm

This section formally describes the mark intersection algorithm. The algorithm operates as follows.

Rule Qualifications: All rule qualifications (predicates) P are assumed to be conjunctions of simple restriction terms, p_i, for $1 \leq i \leq k$ as follows

$$P \equiv p_1 \text{ and } p_2 \cdots \text{ and } p_k$$

The assumption that all conditions are conjunctions does not limit the generality of mark intersection. Predicates containing a mixture of **and** and **or** operators can be broken down into disjunctive normal form (an **or** of **and**'ed clauses) and the clauses can be indexed separately.

Placement of Locks: If a predicate term is on an attribute with an index, t-locks for the term are set in that index. For example, the predicate

(D)

EMP.salary $\leq$ 10000
and EMP.dept = "Engineering"
and EMP.job = "Technician"

would set t-locks on dept and job because they have indexes, but not on the unindexed attribute salary.

If insertion of a record breaks a t-lock in the index on attribute A, a copy of the t-lock is placed on the record on attribute A. As an example, consider the following tuple in EMP:

<James, 28, 15000, Engineering, Draftsman>

This tuple would have a t-lock for (D) on DEPT, but not on SALARY or JOB.

Lock Format: As previously shown in an example, t-locks in the mark intersection scheme are pairs of the form

<rule-id, set of attributes to match>

The rule-id is a four-byte integer. The set of attributes to match can be represented by a bit map. Each bit position corresponds to an attribute of the relation. If there is a "1" in the bit position for an attribute, then that attribute is in the set of attributes to match for the rule. For example, assuming that attributes are in the same order shown in the definition of the EMP relation, a rule with predicate (D) would have the bit map 00111.

Intersecting the Locks on a Tuple: When given a tuple with its set of t-locks, the problem the system must solve is to determine which of those locks require a search into the RULES relation. For example, suppose a pair of rules with the following predicates are given:

(E) emp.dept = "Toy" and emp.job = "Clerk"
(F) emp.dept = "Records" and emp.job = "Clerk" and emp.salary $\geq$ 20000

Consider the tuple

<Richard, 30, 15000, Toy, Clerk>

This tuple is shown in Figure 2.1 with the t-locks it has from the predicates (E) and (F). It can be determined as follows by examining the t-locks shown in the figure that the tuple does not match

Attribute	Value	T-Locks
name	Richard	none
age	30	none (no index on salary)
salary	15000	none (no index on salary)
dept	Toy	<E; DEPT, JOB>
job	Clerk	<E; DEPT, JOB>, <F; SALARY, DEPT, JOB>

Figure 2.1. Example tuple with locks set by the Mark Intersection algorithm

(F), but it does match (E). Looking at the lock for (F) on JOB, it can be seen that there must be locks for (F) on SALARY, DEPT *and* JOB for (F) to match the tuple. There are no locks for (F) on SALARY and DEPT, so the tuple definitely does not satisfy the qualification of (F). However, (E) has locks on both the necessary attributes, DEPT and JOB, so (E) might match the tuple. Thus, only (E) must be fetched from the RULES relation and interpreted against the tuple to see if the tuple matches (E).

An algorithm for determining exactly which rules must be retrieved from the RULES relation based on the *t*-locks on a tuple is described below. The variable SetIndexedAttrs is a set containing the attributes of the relation that have indexes. For example, for the EMP relation, SetIndexedAttrs = {name, dept, job}.

T-Lock Screening Algorithm:

Input:

A tuple with its collection of *t*-locks.

Data Structures:

The primary data structure used will be a hash table that will contain triples of the form <RuleID, SetToMatch, SetMatched>. SetToMatch is the set of attributes that *must* have a *t*-lock for RuleID for a search into RULES to be necessary. SetMatched is the set of attributes for which a *t*-lock from rule RuleID has been found so far.

Algorithm
1. initialize an empty hash table

2. **for** each attribute A of the tuple **do**
   for each t-lock <RuleID, SetToMatch> on A *do*
   compute the hash function of RuleID
   if no t-lock for RuleID is in the hash table
   IndexedSet := SetToMatch $\cap$ SetIndexedAttrs
   store <RuleID, IndexedSet, $\{A\}$> in hash table
   else (a t-lock for RuleID was previously found)
   add A to the SetMatched associated with RuleID
   end if
   end
   end

3. SurvivingRulesSet = ϕ
   for each entry <RuleID, SetToMatch, SetMatched> in the hash table **do**
   if SetToMatch = SetMatched
   add RuleID to SurvivingRulesSet
   end

Output

When the algorithm finishes, the SurvivingRulesSet contains identifiers for all the rules that must be retrieved from RULES.

Processing the Surviving Rules: The rules with RuleID's in SurvivingRulesSet are processed in the same way as rules that have a lock broken when using the Basic Locking algorithm.

2.2.3. Reduced Basic Locking

Both the basic locking and the mark intersection algorithms required that t-locks be stored directly on data records. However, it is not strictly necessary to store the locks directly on the data records because they can be derived when needed by searching the indexes. Electing to derive t-locks when necessary rather than storing them saves disk space at the expense of requiring more computation and I/O time.

The idea above leads to the reduced basic locking algorithm which is very similar to basic locking. The only difference is that when one wishes to find the rules matching a tuple using RBL, *all* indexes on the relation must be searched to derive the locks for the tuple. These are the t-

locks that would normally be on the tuple in BL.

As an example, consider the tuple

<Jessica, 22, 30000, Accounting, Manager>

Suppose that the JOB field of this record was updated to "Vice President" by a **replace** statement. In BL and MI, only the JOB index would be consulted to derive any new t-locks on the value "Vice President"; the t-locks stored on the NAME and DEPT fields would still be valid. Since the JOB index must be updated anyway, no extra I/O is required. However, if RBL were being used and it was necessary to find the locks for the tuple at the time of the update, the indexes on NAME and DEPT would also have to be searched to collect t-locks for all the indexed attributes.

2.3. Performance Characteristics

This section presents a performance analysis of the three lock-based rule indexing schemes: BL, MI, and RBL. The parameters used in the analysis are the following:

Parameter	*Description*
C_1	The cost of evaluating a predicate for a given tuple in ms
C_2	The cost of reading a page in ms
B	The size of the page in bytes
N	The number of tuples in the relation
F	The number of fields in the relation
F_I	The number of fields in the relation with an index
S	The width of individual fields in the relation in bytes
4	The assumed width of pointers in bytes
LSIZE	The assumed width of t-locks in bytes
t	The number of rules
M	The number of terms in a rule predicate
Q	The fraction of records matching a single term of a predicate

These parameter settings are used by default unless otherwise specified:

Parameter	*Default*
C_1	1
C_2	30
B	4000
N	1,000,000
F	6
S	10
F_I	3
t	10,000
Q	0.0001
M	3
LSIZE	4 in BL, RBL 8 in MI

The parameters chosen simplify the situation that would occur in reality. In particular, M and Q would vary for each predicate and predicate term, respectively. An attempt is made to minimize the affect of this simplification by selecting appropriate values for M and Q.

2.3.1. The Predicate Model

In the subsequent analysis, it is assumed that each predicate is a conjunction of M simple restriction terms. These terms may be either equality restrictions or simple range restrictions based on the operators $\{<,>,\leq,\geq\}$. The following is a list of legal terms:

EMP.name = "Fred"
EMP.name $\geq$ "Q"
20000 $<$ EMP.salary $\leq$ 30000

Recall that at least one term of every predicate must be on an indexed attribute so that an index interval can be locked. Thus, it is assumed that terms are distributed such that the first term of every predicate is selected at random from among the F_I indexed attributes. The remaining $M-1$ predicate terms are selected at random from among the remaining $F-1$ attributes.

In the performance analysis that follows, the total cost of determining which predicates match a single tuple is estimated for each of BL, RBL and MI.

2.3.2. Performance of Basic Locking

In this algorithm, a tuple insert incurs zero I/O overhead since new t-locks are derived during the standard index updates. The extra CPU cost is negligible. The only significant cost occurs in finding covering predicates. The predicates corresponding to all t-locks must be accessed (at cost C_1 each) and then checked (at cost C_2) to find the ones that actually cover the tuple. The expected number of t-locks on a tuple in BL is simply the total number of rules, t, times the probability Q that a rule matches a tuple. Therefore, the expected total cost of finding predicates matching a tuple is

$$\text{TOTAL} = t \cdot Q \cdot (C_1 + C_2)$$

2.3.3. Performance of Mark Intersection

The Mark Intersection algorithm determines which predicates match a specific rule in two steps. The first step screens the locks initially on the tuple. The second step retrieves rules from the RULES relation for each lock that survives the screening, and tests these rules against the tuples directly. Thus, the expected total cost of determining which predicates match a tuple is

TOTAL = (cost of screening locks) +
(C_1+C_2)(the number of locks that survive the screening)

2.3.3.1. Cost of Screening the Locks

Screening the locks can be done in time linear in the number of locks using the hashing-based algorithm presented earlier. To get a reasonable estimate of the cost of screening the locks it is assumed that if there are F locks on the tuple then the cost of screening them all is C_1. Using this estimate, there is a cost of $\frac{C_1}{F}$ per t-lock to screen the locks. Thus, if there are T t-locks on a tuple, the cost to screen them is:

$$T(\frac{C_1}{F})$$

The expected number of t-locks on the tuple, N_{TLOCKS}, is the sum over the number of attributes, F, of the following:

(the expected number of rules that have exactly i t-locks on a tuple) $\cdot i$

The first term of the above will be called $S(i)$. The total number of t-locks per tuple is expressed using by the following sum:

$$N_{TLOCKS} = \sum_{i=1}^{F} i\ S(i)$$

The expected number of rules $S(i)$ for which there are i identical t-locks on a tuple is

$$S(i) = \sum_{j=i}^{F} \begin{array}{l}(\#\text{ of rules with exactly j indexed attributes}) \cdot \\ (\text{probability that exactly i of them match the tuple})\end{array}$$

The expected number of rules with exactly j indexed attributes is simply the probability that a rule has j indexed attributes, which will be called $P(j)$, times t, the total number of rules. Application of standard techniques of discreet probability yields the following expression for the function P:

$$P(j) = \begin{cases} 0 & \text{if } j>M \text{ or } j>F_I \\ & \text{or } j<(F_I+M)-F \text{ or } j<1 \\ \dfrac{\binom{F_I-1}{j-1}\binom{F-F_I}{M-j}}{\binom{F-1}{M-1}} & \text{otherwise} \end{cases}$$

The probability that exactly i out of the j indexed attributes of a rule actually match the tuple is $Q^i(1-Q)^{j-i}$. This yields the following final expression for $S(i)$:

$$S(i) = \sum_{j=i}^{F} (t\ P(j)) \cdot (Q^i\ (1-Q)^{j-i})$$

2.3.3.2. Number of Locks That Survive Screening

An estimate for the number of rules that "survive" the initial screening is needed. By the definition of the Mark Intersection algorithm, a rule that has exactly i indexed attributes survives the screening for a tuple only if the tuple has t-locks from that rule on all i of those attributes. Thus, the number of rules with exactly i indexed attributes that survive the screening is

(the expected number of rules with i indexed attributes) ·
(the probability that all i indexed terms of a rule match a tuple)

The expected number of rules with i indexed attributes is $t \cdot P(i)$. The probability that all i indexed terms of a rule match a tuple is simply Q^i. This leads to the following expression for the expected total number of rules that will survive the screening for a given tuple:

$$N_{SURVIVE} = \sum_{i=1}^{F} (P(i)\, t) \cdot Q^i$$

The value of $N_{SURVIVE}$ is dominated by the size of the first non-zero term in this sum since $Q \ll 1$. Sometimes, the first few terms of the sum are zero because $P(i)$ is zero if there is no way for a given rule to have only i indexed attributes. For example, if there are 5 total attributes ($F = 5$), 3 indexes ($F_I = 3$), and 4 terms per predicate ($M = 4$) then at least two of the predicate terms must lie on an indexed attribute, so $P(1)=0$. The estimates for the cost to screen locks and the number of rules that survive the screening yield the following formula for the over-all expected cost of the Mark Intersection algorithm:

$$TOTAL = \frac{C_1}{F} N_{TLOCKS} + (C_1 + C_2) N_{SURVIVE}$$

2.3.4. Performance of Reduced Basic Locking

In RBL, an update in place must be implemented as a delete followed by an insert so that the t-locks covering the modified tuple can be rederived. If the system could otherwise actually do the update in place, it could usually avoid extra I/O to modify the indexes. The overhead *per*

update to perform rule processing using RBL is thus the same as that for basic locking, plus the cost of this extra index I/O. It is assumed that the fraction of updates that are modifications in place is P_{IP}, and the default for P_{IP} is 0.5. All other updates insert a new tuple. The cost to find which rules match a tuple using basic locking will be called C_{BL}. An estimate of average cost per update to determine which rules match the updated tuple is equal to that for basic locking for a fraction $1-P_{IP}$ of the updates. For a fraction P_{IP} of the updates (those in place), F_I indexes must be consulted. It is assumed as a simplification that B-tree index pages are packed full, and that index records are composed of <key,pointer> pairs of size $S+4$ bytes. Since there are B bytes per page, the height of an index, H_I, is the following:

$$H_I = \left\lceil \log_{\left\lfloor \frac{B}{S+4} \right\rfloor} N \right\rceil$$

Thus, an estimate of the average cost to determine the predicates matching a tuple in RBL is as follows:

$$TOTAL = (1-P_{IP})C_{BL} + P_{IP}(C_2 F_I H_I + C_{BL})$$

The overhead of this method for updates in place may seem too high to justify the space savings it provides. However, in a system that implements updates as deletes followed by inserts, the cost of RBL is identical to that for BL, so RBL may be preferred because it saves storage.

2.4. Performance Results

In this subsection the cost functions for the different algorithms are plotted to allow comparison of their performance characteristics. All algorithms discussed are sensitive to the product of the total number of rules, t, and the probability, Q, that a predicate term matches a tuple. Figure 2.2 shows the cost of BL, RBL, and MI versus tQ (the average number of t-locks per tuple in BL). This graph was created by holding t fixed, and varying Q from 0 to .0020.

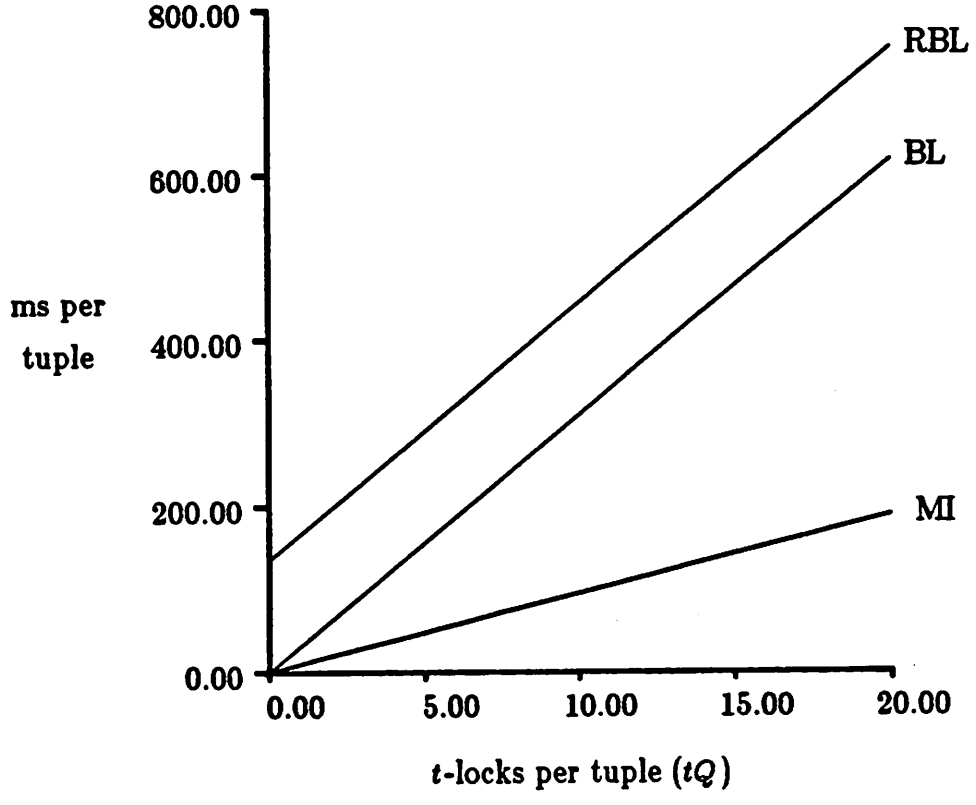


Figure 2.2. Cost of BL, RBL and MI versus tQ

The mark intersection algorithm is also sensitive to the fraction of attributes that have an index. To indicate this sensitivity, Figure 2.3 shows a family of 4 curves obtained by fixing F at 6, M at 3, and varying F_I from 1 to 4. The curves plot the total overhead for rule processing per tuple versus tQ, holding t fixed and varying Q as before. The curve for MI using j indexed attributes is labeled "MI-j" for $j = 1$ through 4. The cost of basic locking is also shown. Notice that the cost of basic locking is almost identical to that for MI-1. This indicates that most of the cost incurred by BL is to read rule definitions from the RULES relation. MI-1 reads the same number of pages from RULES as BL, while MI-2, MI-3, and MI-4 perform successively fewer reads because they are able to successfully screen out some of the t-locks on each tuple.

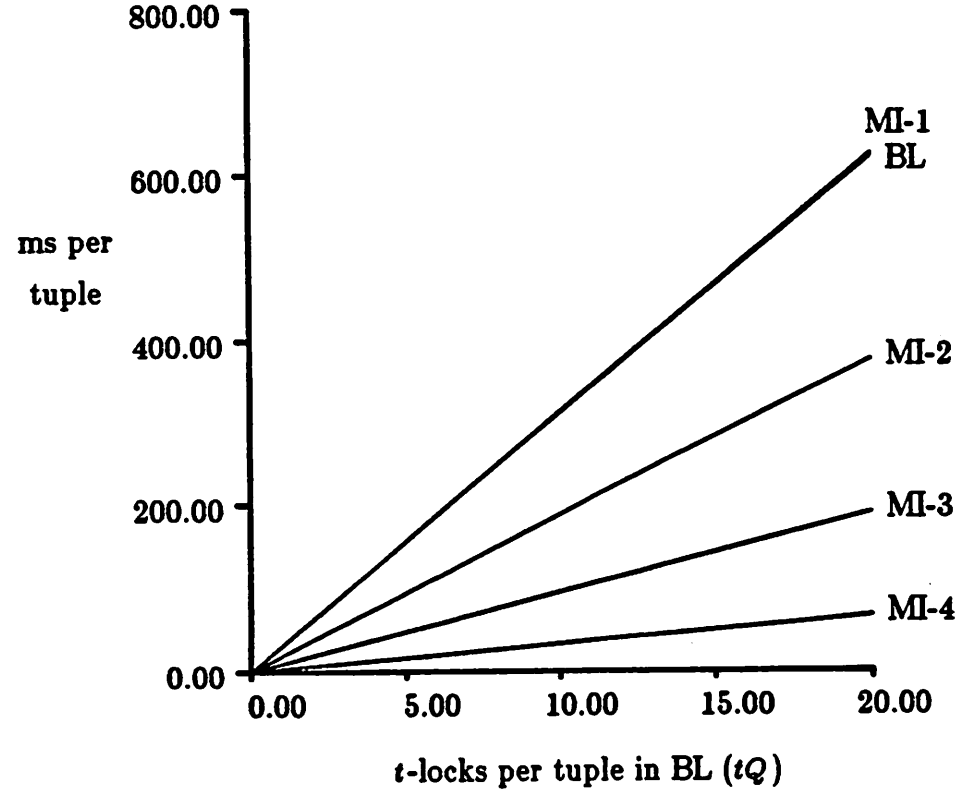


Figure 2.3. Cost of MI versus BL as number of indexed attributes varies

2.4.1. Simplified Analysis of Mark Intersection

The preceding analysis of the Mark Intersection Algorithm assumed that the terms of predicates were uniformly distributed over the collection of F attributes, with the restriction that at least one term must lie on an indexed attribute. However, in reality, it is reasonable to speculate that terms would be more likely to lie on indexed attributes than non-indexed ones. Hence, the performance of MI might be better than previously indicated. In an attempt to measure the effect of the bias toward placing terms on indexed attributes, an analysis is performed here in which there are only two types of predicates, some with one indexed attribute, and others with two.

It is unlikely that both terms will have the same selectivity in practice. To account for differences in term selectivity, it is assumed that the first term of every predicate has selectivity Q, and the second term has selectivity Q'. The default selectivity of Q' will be .01, which is much higher than the default value .0001 for Q.

The cost formula derived for BL still applies for this comparison, but the formula for the cost of MI can be simplified. In the new cost formula for MI, the cost to screen the t-locks on a tuple is the same as previously derived. If X is the fraction of rules that have a single indexed term, and $1-X$ is the fraction that have two, then the expected number of t-locks per tuple, N_{TLOCKS}, is as follows:

$$N_{TLOCKS} = XtQ+(1-X)t(Q+Q')$$

The number of locks that survive the screening, $N_{SURVIVE}$ is the following in this case:

$$N_{SURVIVE}=XtQ + (1-X)tQQ'$$

The formula for the total cost of MI in this situation is still the same as derived before; only N_{TLOCKS} and $N_{SURVIVE}$ change. By plugging the above values for these parameters into the total cost formula form MI, it can be seen that as X approaches 1 (most predicates have 1 indexed term), the total cost for MI becomes exactly that for BL, plus a small amount of overhead to screen the locks. However, as X approaches zero (most predicates have two indexed terms) the cost drops rapidly because most t-locks will be successfully screened, avoiding much access to the RULES relation.

The results of this simplified analysis are shown in Figure 2.4, which plots the total cost of rule processing versus X for both BL and MI. The figure shows that in the case of MI, cost decreases linearly as the fraction of rules with two indexed terms increases. The cost of BL stays constant because only one indexed term is ever used. If every rule has two indexed terms, the cost of MI is smaller because the majority of reads from the RULES relation are avoided by intersecting locks. However, MI does have to pay a cost to screen the locks, and there are many more

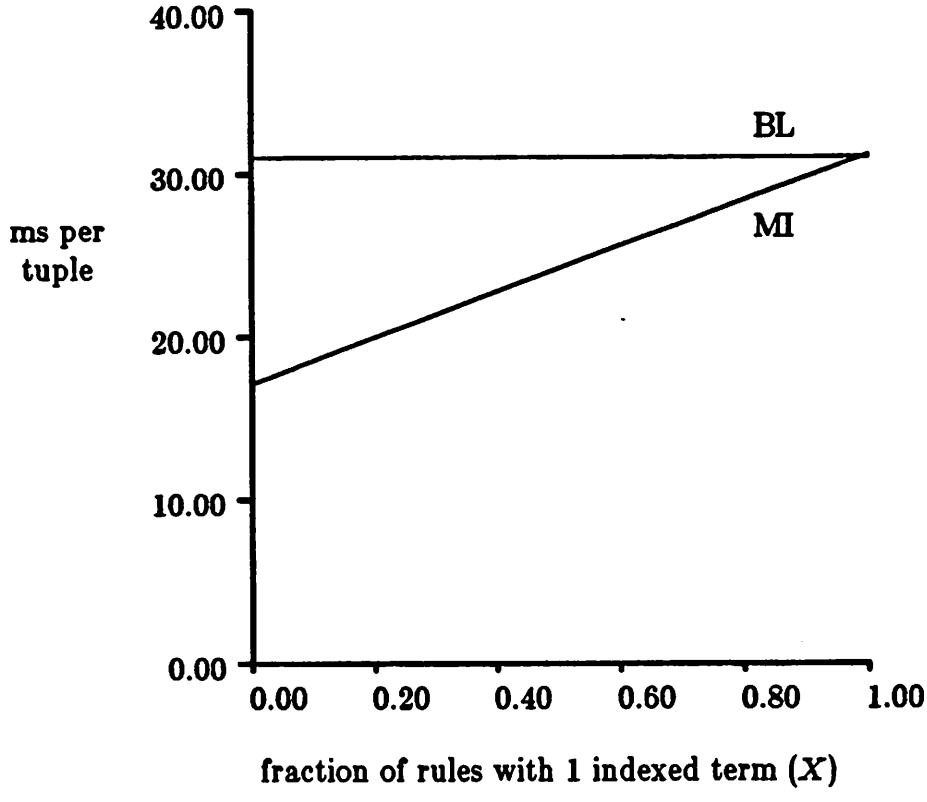


Figure 2.4. Cost of MI versus Cost of BL Assuming Only Two Predicate Types

locks set by MI than by BL since Q' is much larger than Q. If there were no extra cost to screen locks, MI would be a factor of 100 better than BL for X=0. When Q'=0.1 the overhead of screening locks balances out the benefits of saving reads to the rules relation, so BL and MI have approximately the same cost for all values of X.

2.5. Storage Utilization

The storage requirements of the three algorithms vary significantly. RBL has the smallest storage requirements since it sets locks for a single predicate term in the indexes only, not on the data. BL requires somewhat more storage because it sets the same index locks as RBL, and also

puts 4-byte locks on each tuple that conflicts with an index lock. MI requires the most storage, using interval locks in more than one index, and placing 8-byte locks on each tuple that conflicts with an index lock.

2.5.1. Size of the RULES Relation

All three algorithms use an identical RULES relation containing one tuple for each rule. Recall that the formant of RULES is

RULES(id, name, rule-def)

The size of RULES is estimated as follows. The id field requires four bytes and name requires 16 bytes. The rule-def attribute has subfields containing the number of bytes indicated below:

field	*definition*	*bytes*
text	text of rule	100
predicate	compact representation of rule predicate	$M(2S+4)$

The estimate above for the size of the predicate field is based on the fact that each of the M rule terms will have up to 2 constants of length S bytes each, plus another 2 bytes to indicate the attribute on which the term lies and 2 bytes to represent the operators for the term. The complete expression for the size of a single tuple in RULES is

$$Y = 4+16+100+M(2S+4)$$

Given this tuple size, the number of bytes occupied by RULES, $SPACE_{RULES}$, is

$$SPACE_{RULES}=tY$$

2.5.2. Storage Use in Reduced Basic Locking

The only storage used in RBL is that for the locks set in the indexes, plus the size of the RULES relation. In general a rule predicate term can have the form

$$constant_1 \text{ lower\_op } attribute \text{ upper\_op } constant_2$$

where lower_op and upper_op are one of the relational operators $\{ <, \leq \}$. To lock an interval of this form in a B-tree, a hierarchical locking scheme is employed (see Appendix 1 for a complete description of the algorithm for placing interval locks). The structure of an interval lock is

$$t = [\text{RuleID, RuleType, } constant_1\text{, lower\_op, } constant_2\text{, upper\_op}]$$

Locks of this form require LSIZE bytes for the RuleID, 1 byte for the RuleType, lower_op and upper_op, and S bytes each for $constant_1$ and $constant_2$. The size of an index interval lock, ILSIZE, is the following:

$$ILSIZE = 3+LSIZE+2S$$

To get an estimate of the amount of space used to set interval locks, observe that for Q=.0001 and N=1,000,000, each interval lock covers 100 tuples. To get a more accurate count of the total number of pages used, it is assumed here that B-tree pages are packed 69% full as derived in [Yao78]. Index records contain one data value and a 4-byte pointer, so leaf pages in the index contain $.69B/(S+4)$ records. For the default values of B and S, the number of records in a leaf index page is approximately 200. The probability that a lock will span a page boundary is 100/200=1/2, so the expected number of locks, $TERMLOCKS$, set by a single predicate term in a B-tree is

$$TERMLOCKS = \frac{1}{2}\cdot 1+\frac{1}{2}\cdot 2 = 1.5$$

The amount of space used by the range locks set by all t rules is

$$SPACE_{RGLOCKS} = t \cdot TERMLOCKS \cdot ILSIZE$$

This yields the following expression for the total amount of storage space used by the rule subsystem in RBL:

$$SPACE_{RBL} = SPACE_{RGLOCKS}+SPACE_{RULES}$$

To determine the amount of storage used by the data and indexes using no rule processing, it is assumed that all F_I indexes are secondary. In the data relation itself, the N tuples have F

fields, each of width S bytes. There are F_I indexes, each with a total of N <key,pointer> pairs, and each pair contains $S+4$ bytes. Only the leaf-level pages are counted because there are far more than contained in the upper levels. The total storage occupied by the data is thus as follows:

$$SPACE_{DATA} = N.S.F + F_I \frac{N(S+4)}{.69}$$

2.5.3. Storage Use in Basic Locking

The Basic Locking algorithm uses the same amount of storage space to mark the indexes as does RBL, plus extra storage to mark the data records. The expected number of rules that match a tuple, and thus the expected number of t-locks per tuple, is simply tQ. Since there are 4 bytes per t-lock in BL, each tuple will have, on the average, $4tQ$ extra bytes. Including the size of the rules relation, the total storage used by the rule subsystem in BL is the following:

$$SPACE_{BL} = SPACE_{RULES} + SPACE_{RGLOCKS} + N(4tQ)$$

2.5.4. Storage Use in Mark Intersection

In MI, the average number of bytes of t-locks per tuple is the expected total number of indexed rule terms times the size of a t-lock times the probability that a term matches a tuple. The expected total number of indexed rule terms, which will be called T, is the sum of j times the expected number of rules with j indexed terms, $t \cdot P(j)$, for j=1,2, $\cdots$ F. The simplified expression for T is

$$T = t \sum_{j=0}^{F} jP(j)$$

There are 8 bytes per t-lock on data tuples in MI so the expected number of bytes occupied per tuple by t-locks is $8 \cdot T \cdot Q$. Extra data is also required to place locks in the indexes. The ratio of the total number of indexed terms in RBL to the number in MI is T/t, so the total amount of

data used for index locks in MI is the following:

$$\frac{T}{t} SPACE_{RGLOCKS}$$

This yields the following expression for the total number of bytes used in MI:

$$SPACE_{MI} = SPACE_{RULES} + \frac{T}{t} SPACE_{RGLOCKS} + 8 \cdot T \cdot Q \cdot N$$

2.6. Storage Analysis Results

To compare the amount of space used by the three algorithms, Figure 2.5 shows the space occupied by the data and the RULES relation and the amount of space required by the three different rule indexing algorithms. As expected, the figure shows that MI requires more storage than BL and RBL because MI locks more than one attribute per rule, and t-locks in MI contain 8 bytes instead of 4. RBL is clearly the most economical user of storage among the three algorithms. The line showing the amount of space occupied by the RULES relation demonstrates that RBL uses very little space for locks. Also, range locks in the indexes occupy a small amount of space compared with the actual locks on tuples. The relative amount of space used for locks in indexes and on data records is illustrated by the difference in the amounts of storage used by BL and RBL (BL puts t-locks on tuples in the database and RBL does not).

2.7. Discussion

The choice between the three lock-based rule indexing methods evaluated in this paper depends on the database and rule environment. In general, basic locking appears to be the method of choice because: (1) it is easy to implement as a byproduct of normal query processing, (2) it performs well with a small to moderate number of rules, and (3) it requires only slightly more disk space than occupied by the data itself. However, if the RULES relation is based primarily on disk, and a significant fraction of all rules have more than one indexed predicate term, mark intersection can be much more efficient than the other methods. Mark intersection

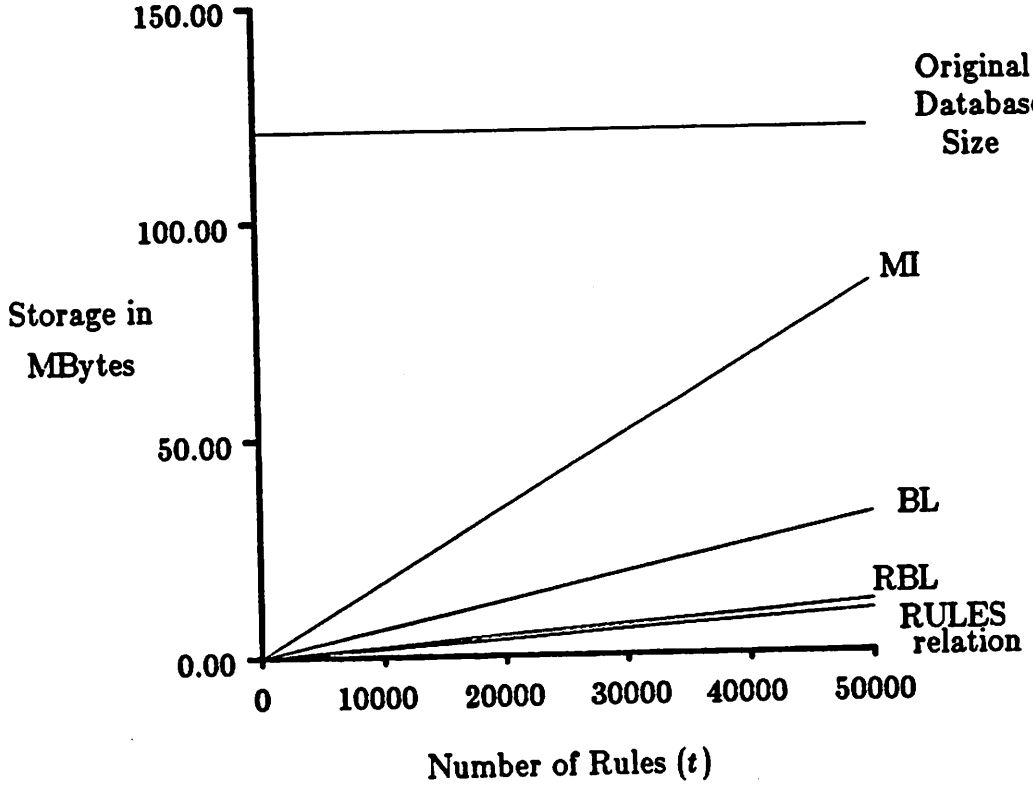


Figure 2.5. Amount of storage used by lock-based rule indexing algorithms

provides this speed-up because it uses t-locks that contain more information than in basic locking. This extra information allows many rules to be screened by intersecting locks, thus avoiding some reads from the RULES relation. Due to its high index I/O requirements, reduced basic locking is only suitable for trigger processing in database systems where updates in place are performed as deletes followed by inserts. This allows RBL to "piggy-back" its searches for t-locks in the indexes with the index updates performed when the new tuple value is inserted. RBL is not suitable for processing inference rules because it would cause one or more index lookups for each tuple on a read-only query, which is an unacceptable performance burden.

Unfortunately, if the RULES relation is on secondary storage, there appears to be a practical limit of only a few rules with a t-lock on each tuple using any of these schemes. For example, using BL with $tQ=10$ (i.e. approximately 10 rules have a t-lock on each tuple) Figure 2.2 shows that about 300 ms are required per tuple to do rule processing, which is a very large amount of time to spend processing one tuple. This observation makes it clear that to achieve acceptable performance for a larger number of rules, access to the RULES relation must be speeded up by keeping some or all of it in high-speed memory. With the advent of large memories, maintaining most of RULES in high speed memory should prove feasible. A standard LRU replacement algorithm using a very large buffer pool would likely keep a sufficiently large portion of RULES in memory to avoid most of the I/O observed in the analysis in this paper. Because MI spends extra CPU time intersecting locks to avoid probes into the RULES relation, the difference in performance between MI and BL will narrow if these probes are made less expensive, making BL even more attractive.

CHAPTER 3

MAINTAINING DERIVED OBJECTS

This chapter presents a collection of algorithms for processing queries against views and other derived data objects. When a user wishes to retrieve part or all of a view, the conventional way to process the request is to construct the result from the base relations using query modification [Sto75]. Another method for answering queries against views is to keep a stored copy of the view and read it directly during query processing. The simplest way to maintain the view is to use the caching procedure described in chapter 1, whereby the view is recomputed before a query if it has been invalidated by a preceding update. A more sophisticated version of this method is to use a differential view maintenance algorithm (e.g., AVM) to keep the stored copy of the view up to date. Several other differential view maintenance algorithms are proposed in this chapter. In addition, methods are presented for maintaining materialized copies of other types of derived objects, including:

1. database procedures
2. aggregates
3. views or procedures containing aggregates

Before describing a specific view maintenance algorithm, it is useful to develop a taxonomy for classifying them. One type of classification already introduced is the distinction between immediate and deferred view maintenance. Recall that immediate algorithms maintain views after each update transaction, while deferred algorithms maintain views before queries that read views. A second type of classification involves the time at which the compilation and optimization step is performed in a view maintenance algorithm. If this step is delayed until just before computing the expressions required to refresh the view, the algorithm is called *dynamicly optimized* or simply *dynamic*. If compilation and optimization is performed in advance, the algorithm

is described as *staticly optimized* or *static*. The third classification regards whether or not the algorithm factors out shared subexpressions. Algorithms that use common subexpression elimination techniques are called *shared* and those that do not are called *non-shared*. In summary, view maintenances algorithms can be classified according to the following criteria:

1. immediate versus deferred
2. dynamic versus static
3. non-shared versus shared

The original AVM algorithm described in [BLT86] is immediate, dynamic, and non-shared. A staticly optimized version of algebraic view maintenance called *static AVM* (SAVM) is proposed in this chapter. In general, versions of both AVM and SAVM are possible which are either immediate or deferred, or either shared or non-shared. A view materialization algorithm called *Rete view maintenance* (RVM) is also proposed in this chapter. RVM utilizes a Rete network to perform view maintenance. RVM is staticly optimized because it takes a collection of view definitions in advance and builds an optimized structure for maintaining the views. Possible optimization strategies are to share common subexpressions between views, and arrange join orderings in an efficient way. Because common subexpressions are combined when the Rete network is built, RVM is classified as a shared algorithm. A non-shared version of RVM could be constructed, but this possibility will not be discussed since sharing subexpressions in RVM is straightforward and clearly preferable. Both immediate and deferred versions of RVM are possible.

Static AVM and Rete view maintenance are presented in detail below. Methods are then given for maintaining aggregates and database procedures, as well as views and procedures containing aggregates. The chapter concludes with a discussion of the different alternatives available for materializing derived database objects.

3.1. Staticly Optimized View Maintenance Algorithms

3.1.1. Algebraic View Maintenance

It is possible to extend the algebraic view maintenance algorithm to avoid the compilation overhead incurred in dynamic AVM, and also to take advantage of shared subexpressions in a way similar to Rete view maintenance. An algorithm is presented here that makes use of a pre-compiled execution plan to perform algebraic view maintenance. The algorithm has two main components. The first component takes as input a collection of view definitions $V_1, V_2, \cdots V_M$, and produces as output a collection of plans for maintaining the views. The plans are represented as a directed acyclic graph. In a non-shared version of SAVM, the graph is a collection of disjoint trees. In a shared version of SAVM, the graph is a collection of trees that may share some subtrees. The leaf nodes of the graph represent scans of relations, and internal nodes represent joins. Each node is assigned a *level number*. Base relations appear at level 0, the results of selections from a single base table appear at level 1, two-way join nodes are at level 2, and so on up to level J. In general, a node at level i (where $i > 1$) joins one node from level $i-1$ with another node at a level less than i. An update transaction may append or delete tuples from each of the base relations. In general, after an update transaction, the following sets of tuples may be present:

$R_1, R_2, \ldots R_N$ – base relations
$A_1, A_2, \ldots A_N$ – appended tuples
$D_1, D_2, \ldots D_N$ – deleted tuples

Given these sets as input, the algorithm must find the sets of tuples to be inserted into and deleted from the views $V_1, V_2, \cdots V_M$. These insertion and deletion sets are denoted as follows:

$A_{V_1}, A_{V_2} \ldots A_{V_N}$
$D_{V_1}, D_{V_2} \ldots D_{V_N}$

The value of these sets is found using the following algorithm:

```
for i = 1 to J do
  for all nodes n at level i do
    compute A_n and D_n
  end
end
```

At any level i the A and D sets for all nodes at levels less than i have been computed, so they can be used to help compute the A and D sets for level i nodes. Suppose that a view V_i has another view V_j as a subexpression. Then in a shared version of SAVM, the sets A_{V_j}, D_{V_j} and V_j are used to help compute A_{V_i} and D_{V_i}.

As an example of the benefits that can be obtained by combining shared subexpressions, consider the following pair of views:

```
define view V1 (R1.all, R2.all, R3.all)
where R1.a = R2.b
and R2.c = R3.d

define view V2 (R2.all, R3.all)
where R2.c = R3.d
```

Using the standard dynamic AVM algorithm, both these views are maintained separately. Suppose a set of tuples A_1 is appended to R_1. To find the sets of tuples to append to V_1 in this situation, the following query must be run:

```
retrieve (A1.all, R2.all, R3.all)
where A1.a = R2.b
and R2.c = R3.d
```

A possible execution plan for the above query is shown in Figure 3.1. (The plan is drawn with the leaves at the top and internal nodes at the bottom to so that the style of presentation is the same as for Rete networks.) By analyzing the definitions of the two views in advance, it is possible to identify that V_2 is a subexpression of V_1. This makes it possible to find the set of tuples that need to be appended to V_1 using the following query:

```
retrieve (A1.all, V2.all)
where A1.a = V2.b
```

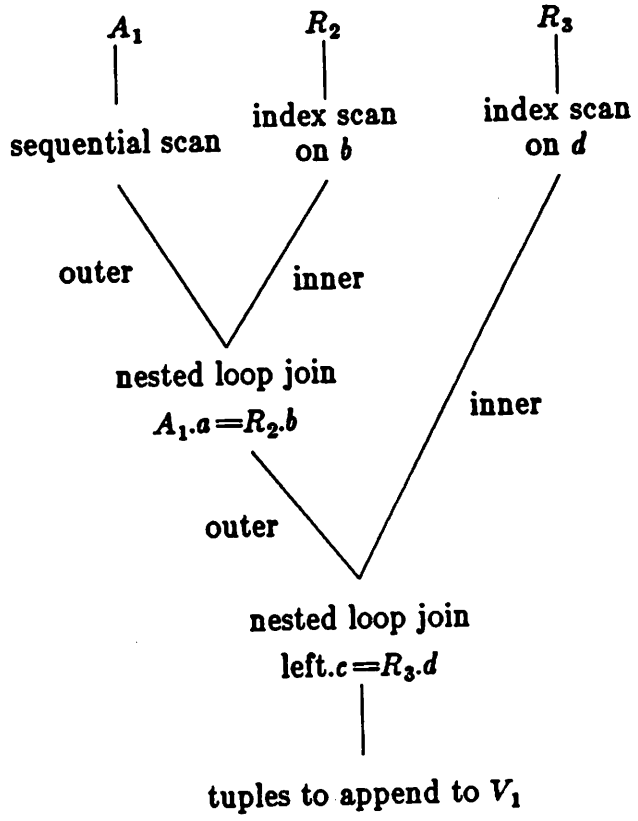


Figure 3.1. Standard algebraic view maintenance (no sharing)

A plan for executing this query is shown in Figure 3.2. This plan requires computation of only a two-way join, as opposed to the three-way join required by the original plan. This could provide a substantial cost savings.

The previous example showed how shared subexpressions can be useful because they reduce the number of joins that are performed. Sharing subexpressions can also improve performance by eliminating redundant computation. For example, consider the views V_1 and V_2 previously defined. Suppose that tuples A_2 and D_2 are inserted into and deleted from R_2, respectively. The algorithm will first find the net changes to V_2 (A_{V_2} and D_{V_2}). This will be done by executing the following queries:

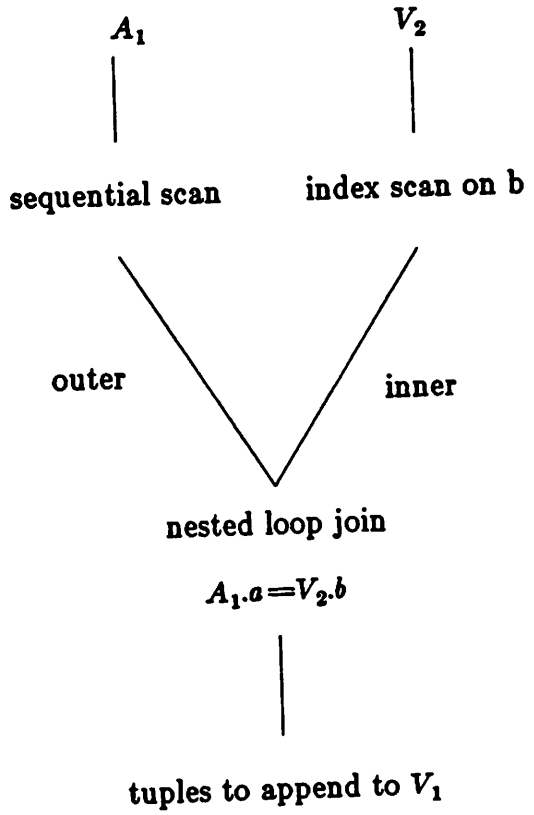


Figure 3.2. Algebraic view maintenance using shared subexpression

to find A_{V_2}:

```
retrieve (A2.all, R3.all)
where A2.c = R3.d
```

to find D_{V_2}:

```
retrieve (D2.all, R3.all)
where D2.c = R3.d
```

A_{V_2} and D_{V_2} will be used at the next level to compute the changes to V_1 (A_{V_1} and D_{V_1}) as follows:

to find A_{V_1}:

retrieve (R_1.all, A_{V_2}all)
where R_1.a = A_{V_2}b

to find D_{V_2}:

retrieve (R_1.all, D_{V_2}all)
where R_1.a = D_{V_2}b

The effort for computing A_{V_2} and D_{V_2} was spent only once, and shared between V_1 and V_2. After A_{V_1}, D_{V_1}, A_{V_2} and D_{V_2} have been found, they are used to update the stored copies of V_1 and V_2.

Note that of the two parts of this algorithm, only the second part has been described completely here. The first stage in the algorithm (constructing a pre-compiled execution plan for the views) is a complex optimization problem. A heuristic algorithm that merges individually optimized execution plans for each view into a global execution plan is straightforward to construct. Such an algorithm will not, however, produce a globally optimal plan. Previous work on multiple-query optimization (e.g. [Sel86a, Sel86b]) can serve as a starting point for future research into methods for constructing an optimized merged execution plan for maintaining a collection of views.

3.1.2. Maintaining Views Using a Rete Network

The Rete network was designed to find combinations of tuples that match production-rule predicates. Because relational database views have the same structure as the rule predicates used in OPS5, a Rete network can be used to find new tuples that satisfy a view qualification. For example, consider the following view, which lists all the technicians that work in building B23:

```
define view AP (EMP.all, DEPT.all)
where EMP.job = "Technician"
and DEPT.building = "B23"
and EMP.dept = DEPT.dname
```

The equivalent OPS5 rule condition has the form

(EMP ^job Technician ^dept <x>)
(DEPT ^dname <x> ^building B23)

This condition would be represented by the Rete network shown in Figure 3.3. The bottom node in the network is a P-node containing the rule associated with the condition. However, if this P node is replaced by a β-memory node, as shown in Figure 3.4, all tuples contained in that node will match the qualification of the view AP.

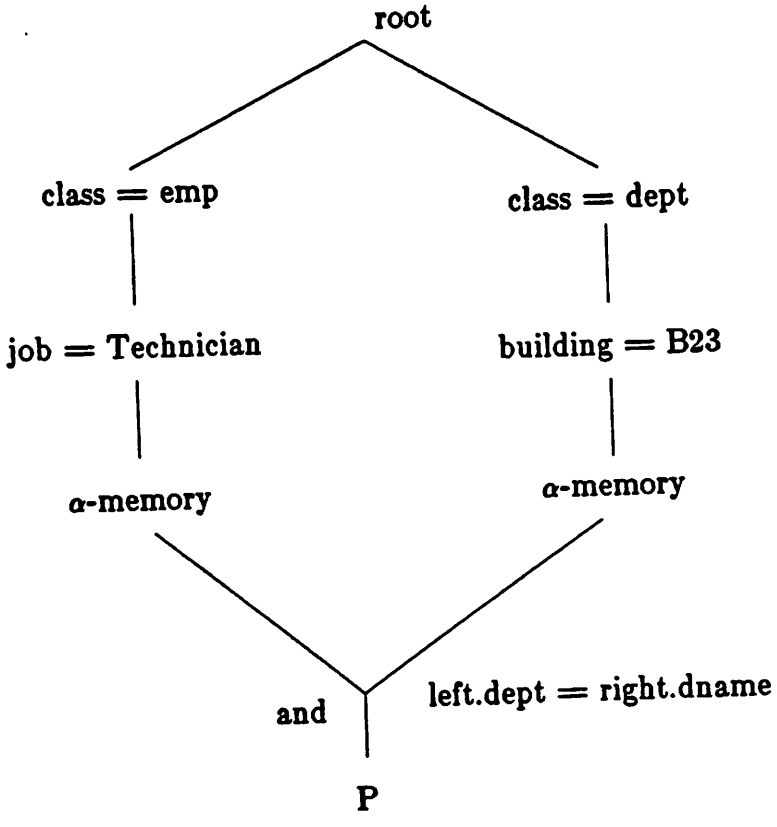


Figure 3.3. Rete network for example rule

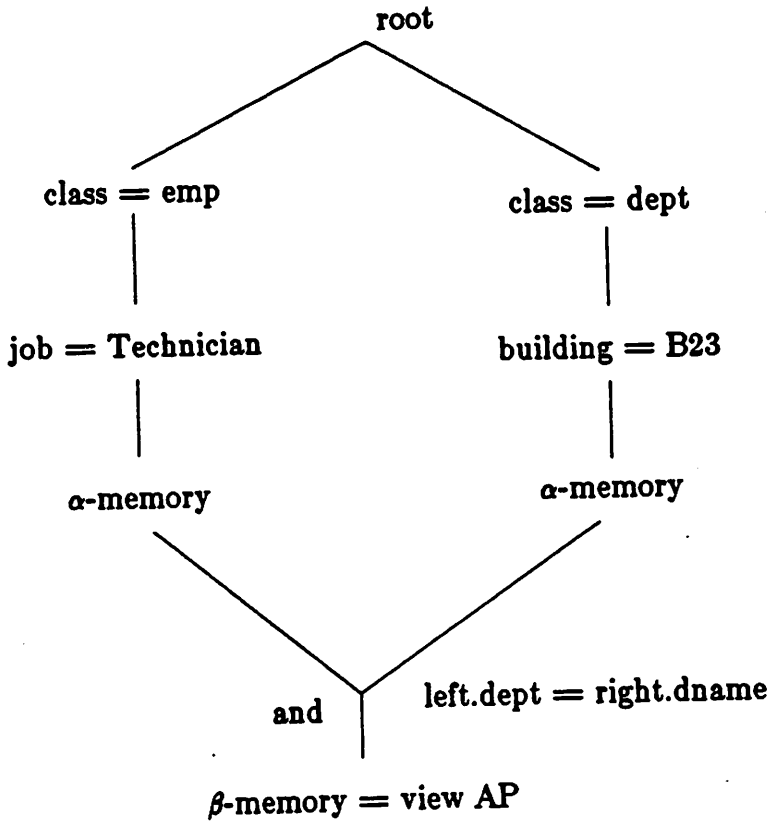


Figure 3.4. Rete network used for view maintenance

3.1.3. The Rete View Maintenance Algorithm

In general, given a collection of views, a Rete network can be constructed that has a memory node corresponding to each view. The desired semantics are that each memory node associated with a view V should contain the current value of V. The meaning of "current value" is defined as the value of V that would be retrieved by the command

retrieve (V.all)

given the current contents of the base relations. Although all tuples in a memory node will match the qualification of the view associated with the node, it must be demonstrated that the set of

tuples in each memory node is exactly equal to the current value of the corresponding view.

Consider the following algorithm for maintaining a collection of views, $V_1, \cdots V_N$ using a Rete network R_N.

Algorithm A

input:

1. A version of Rete network R_N where each memory node associated with one of the views $V_1, \cdots V_N$ has the correct current value.

2. A database transaction X represented by Rete Network tokens $t_1, t_2 \cdots t_m$. This can be a list of + or − tokens in any order. Updates in place are represented by a − token for the old value followed by a + token for the new value.

output:

A version of R_N where each memory node associated with $V_1, \cdots V_N$ has the correct current value for the database after transaction X has been executed.

method:

for $i = 1$ to m **do**
pass t_i through R_N, allowing its effects to propagate as far as possible using a depth-first traversal through the network.
end

Theorem:

Algorithm A is correct. In other words, after it executes, every memory node associated with a view V contains the current value of V with respect to the database after execution of transaction X.

Proof:

This can be shown by induction on the number of tokens m in transaction X. To simplify the proof, it is assumed that every memory node corresponds to a unique view. This does not affect the results, since for a given collection of views and corresponding Rete network, extra "dummy" views can be added for the memory nodes that do not already correspond to a view in the original collection.

Base case: $m=1$

Here, the transaction X consists of a single token t_1, representing insertion or deletion of a single base relation tuple Y. It is necessary to show the following:

> If insertion of Y causes a tuple t to enter the view corresponding to a memory node M, Algorithm A causes one and only one instance of t to be added to M. Similarly, if deletion of Y causes a tuple t to leave the view corresponding to M, the algorithm removes one and only one instance of t from M.

Suppose M is an α-memory node. Then if t_1 reaches M, t_1 matched the qualification for M. To move M to the correct state, if the tag of t_1 is $-$, t_1 should be deleted from M, and if it is $+$, it should be inserted. This is exactly what the Rete network algorithm will do. Thus, views corresponding to α-memory nodes are left in the correct state after passing a token through the Rete network.

The *depth* of a memory node M is defined as the maximum number of memory nodes between M and the root. Assume that all views corresponding to memory nodes of depth $\leq k$ are updated correctly using Algorithm A (the base case has been proven above for depth=0, in the case of α-memory nodes). Suppose the token of t_1 is $+$. Consider a β-memory M of depth $k+1$. Let S be the set of new elements that need to be inserted into M to bring M to the correct state, given the insertion of Y. Every tuple s in S must correspond to the join of two tuples, t_{left} and t_{right}, from the input memories of M (if not, then the input memories are

incorrect, but since they are at depth $< k$, they must be correct). Tuples t_{left} and t_{right} may be old values that already existed in the input memories, or new ones that entered those memories because of insertion of Y. The following combinations of new and old tuples may occur in S:

t_{left}	t_{right}
new	old
old	new
new	new

The (old,old) combination is not possible since such a tuple would not be added to M by insertion of Y. If a tuple s in S is of the form (new,old), then a single "new" t_{left} tuple will be inserted into the left input memory of M (this follows since the depth of that input memory node is $\leq k$). If this t_{left} tuple joins to an "old" tuple t_{right} in the right input memory for M, then a single token $[+,<t_{left},t_{right}>]$ will be passed to M. Hence, exactly one corresponding tuple will be inserted into M. Using a similar argument, if a new tuple arrives at the right input memory of M, one and only one instance of each (old,new) tuple in S will be added to M.

The case of (new,new) tuples in S is somewhat more complicated. Clearly, if a new t_{left} value arrives before the new t_{right} value, the t_{left} value will not combined with the t_{right} value to form a token to pass to M. However, when the new t_{right} value arrives later, it will join to the t_{left} value, and a single token $[+,<t_{left},t_{right}>]$ will be passed on to M. This will cause one and only one tuple $<t_{left},t_{right}>$ to be inserted into M. A symmetric argument holds for the case where the new t_{right} arrives before the new t_{left}. Hence, every tuple s in S is inserted into M exactly once, and no other tuples are inserted. Algorithm A thus leaves all memory nodes of depth $k+1$ in the correct state when a + token is passed through the network. By induction, all memory nodes are left in the correct state after insertion of a + token. The proof for - tokens is similar, and will not be shown. Hence, Algorithm A leaves all memory nodes in the correct state for transactions of length $m=1$.

Induction Step:

Assume that Algorithm A is correct for all transactions of length j or less. The affect of a transaction of length $j+1$ is simply the affect of two transactions run serially, one of length j (which is assumed correct) followed by one of length 1 (which was shown to be correct in the base case). Hence, applying Algorithm A for a transaction of length $j+1$ must leave all memory nodes (i.e., views) in the correct state. By induction, this shows that Algorithm A is correct for all transactions of length $m \geq 1$. □

It is important that the effects of tokens are propagated in depth-first order through the network in Algorithm A. For example, consider the view JP consisting of all the programmers working in the same department as John:

```
define view JP (EMP.all, E.all) using E in EMP
where EMP.name = "John"
and E.job = "Programmer"
and EMP.dept = E.dept
```

A Rete network for maintaining JP is shown in in Figure 3.5. Suppose that John is a programmer. Since John is a programmer, there is initially a tuple T in JP that consists of John's EMP tuple joined with itself. Also, both the left and right α-memories will contain John's tuple. Now, suppose John's tuple is deleted from EMP, and a $-$ token for it is placed at the root of the network. Using depth-first propagation, John's tuple would be deleted from one α-memory (say the left), and that would cause T to be deleted from the β-memory JP. Then, John would be deleted from the right α-memory, leaving JP in the correct state. However, using non-depth-first propagation, the following anomaly could occur. John's tuple could be deleted from the left α-memory, then the right one. Then the system would test whether John's tuple that was just deleted on the left joined to any tuples on the right. The answer would be no, so T would remain in JP, which is incorrect. This illustrates that the depth-first propagation used in Algorithm A is essential to

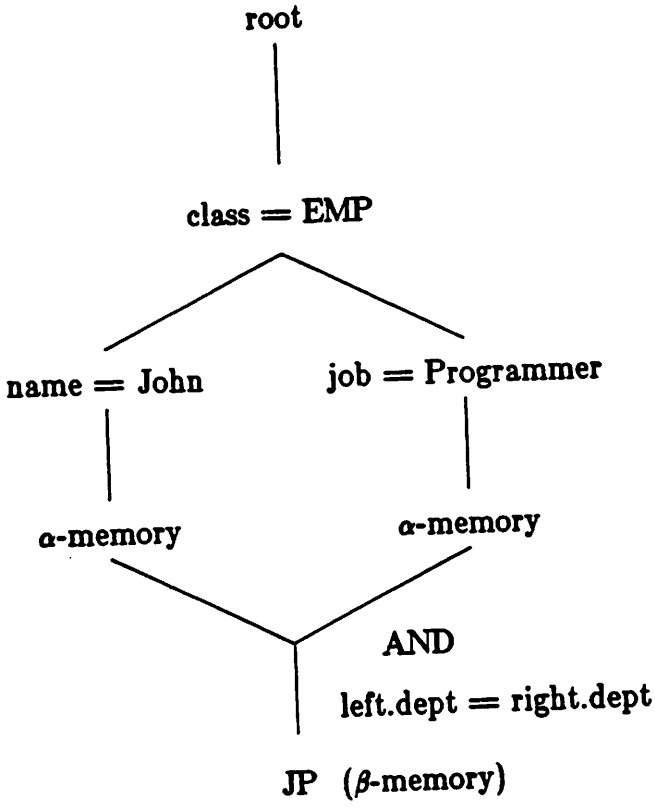


Figure 3.5. Error using non-depth-first propagation

maintaining the views in the correct state.

A potential performance problem with the Rete network as previously described is that tokens are "broadcast" from the root to all t-const nodes. If the number of t-const nodes is large, as would be expected in a database system with a large number of rules, testing the predicates at all t-const nodes will be very expensive. Fortunately, it is straightforward to apply lock-based rule indexing to reduce this cost. Consider the general Rete network shown in Figure 3.6. All parts of the network above the dotted line in the figure can be encoded by setting locks in the catalogs, and conventional indexes. Rather than a rule identifier, each lock would contain a pointer to a place in the network where a deleted or inserted tuple should be deposited if it

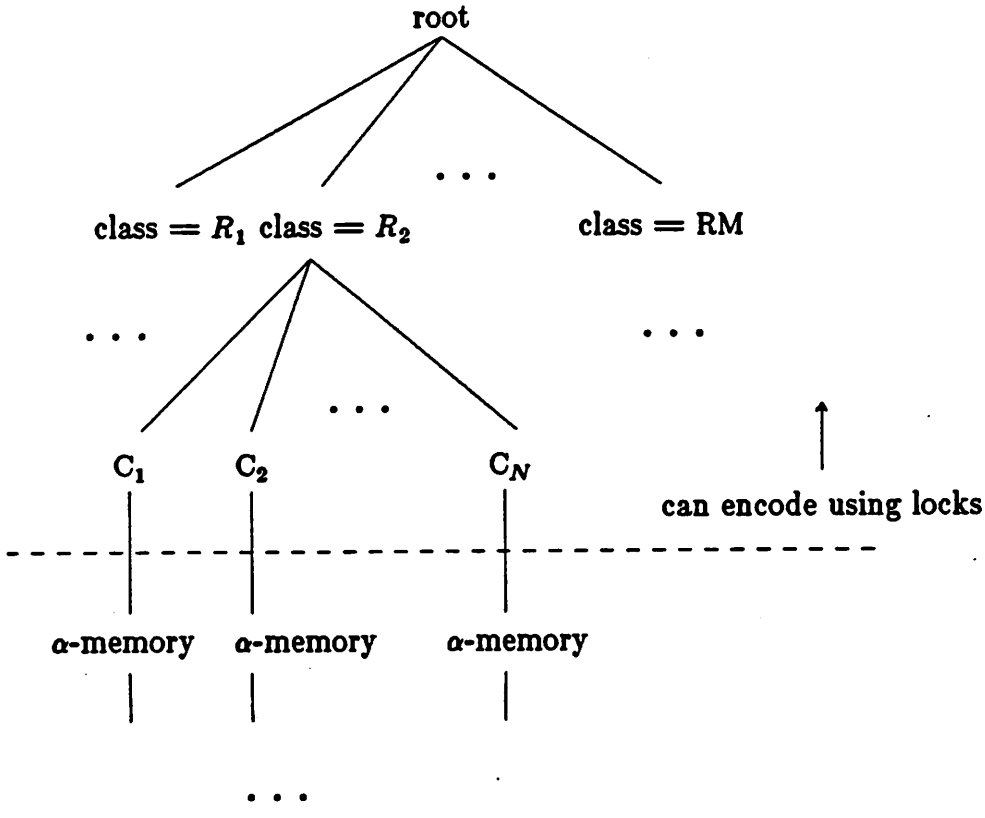


Figure 3.6. Augmenting Rete network with locking

conflicted with the lock. For example, if a rule had the condition

EMP.dept = "Toy"

then t-const nodes for "class = EMP" and "dept = Toy" would be placed in a normal Rete network. Assuming there is an index on the EMP.dept field, these t-const nodes can be replaced by putting a lock record containing the value "Toy" in the index. This record would point to the α-memory node that would normally come after the t-const node "dept = Toy".

3.2. Dynamicly Optimized View Maintenance and Sharing

An important feature of the shared version of SAVM as well as RVM is that the cost of processing any shared subexpression is paid only once. As originally described, the standard (dynamic) AVM algorithm does not take advantage of shared subexpressions. In general, dynamic AVM must compute the result of k relational-algebra expressions $Q_1, Q_2, \cdots, Q_k$ to maintain views after each transaction. It is possible to extend dynamic AVM to factor out common-subexpressions in $Q_1, Q_2, \cdots, Q_k$, using techniques described in [Sel86a, Sel86b]. Clearly, this shared, dynamic version of AVM would have a large run-time optimization cost.

3.3. Database Procedures

As described in Chapter 1, database procedures are collections of queries stored in the database. Since database procedures are collections of one or more queries, they have the same structure as a collection of views. This allows any view materialization method to be used to materialize the results of stored procedures. The most straightforward method to allow the result of a stored procedure to be retrieved is to simply run the queries in the procedure definition. This approach is analogous to query modification, and was used in the system described in [SAH84, SAH85].

Another method for maintaining procedures is *caching* [Sel86b, Sel87, StR86]. Caching involves storing the result of a procedure in the database. If the database is updated in a way that would change the value of the result, then the cached value is invalidated. When retrieving a cached procedure value, if the cache is valid, the value is returned. Otherwise, the value is recomputed and written to refresh the cache.

Alternatively, any incremental view maintenance algorithm can be employed to maintain up-to-date copies of the results of database procedures. To accomplish this using the standard AVM algorithm, each query inside a procedure is maintained individually. Using Rete view

maintenance, all queries in database procedures are encorporated into a single Rete network. The result of each query is maintained as a memory node in the network. Using shared, staticly optimized AVM, a shared execution plan is produced for all the queries in all procedures present in the system. Changes to the value of each stored query are found using this plan.

3.4. Aggregate Maintenance

The aggregate maintenance procedures presented here allow any general aggregate to be materialized. In order to maintain consistency, the algorithms are designed so that the value of a materialized aggregate will be identical to the result of processing the same aggregate conventionally. The algorithms presented by Epstein for processing aggregates in QUEL queries are used as a guideline* [Eps79,HSW75]. These are briefly reviewed below.

3.4.1. Basic Aggregate Processing Algorithms

To compute a scalar aggregate, a *state* data structure is used to keep track of the calculations made up to the current time. For example, the state of the computation of the **avg** (average) aggregate can be represented as a pair of numbers, one equal to the average of the values seen so far and the other equal to the count of those values. By convention, the state of an aggregate is required to contain the current value of the aggregate at all times. The following simple algorithm is used in INGRES to compute the value of a scalar aggregate:

1. Initialize variables to hold the aggregate state
2. For each tuple meeting the qualification of the aggregate,
   update the state.

At the end of this procedure, the value of the state contains the result.

* The aggregate maintenance algorithms presented in this chapter are discussed in terms of the QUEL query language [HSW75]. However, the techniques presented here can also be used to maintain aggregates expressed in other query languages, including SQL [CAE76] and POSTQUEL [StR86] (see appendix 2 for a description of the POSTQUEL aggregate syntax).

To compute aggregate functions, for each unique value of the by-list a pair of the form <state,by-value> is maintained . This set of values can be put in a temporary relation with attributes for the state, and each component of the by-list. For example, the aggregate

avg(EMP.salary by EMP.dept, EMP.job)

accumulates its results in the relation

TEMP(count, average, dept, job)

The state of the aggregate is represented by the count and average attributes, and the by-list entry is composed of the dept and job attributes.

The algorithm to compute an aggregate function is:

1. Create a temporary relation with the required attributes.
2. For each tuple satisfying the qualification
   a. if the temporary relation already contains a tuple with the same by-list value, update the state value of the tuple
   b. otherwise, add a tuple to the temporary relation with the new by-list value and the correct initial state

When this algorithm finishes, the temporary relation contains the result of the aggregate function.

There are a few variations on standard scalar aggregates and aggregate functions discussed above. Aggregates can be classified as either *unique* or *non-unique*. With a unique aggregate, duplicates are removed from the input before the result is computed. Duplicates are not removed in non-unique aggregates. Consider the following sample database:

name	dept	salary	job
Bob	Toy	10,000	Clerk
Jim	Toy	20,000	Buyer
Al	Fire	10,000	Fire-fighter
Susan	Fire	12,000	Fire-fighter

The count-unique (countu) aggregate below returns the value 3 because there are only 3 unique values of salary in this table.

countu(EMP.salary)

The non-unique count aggregate returns 4 when used in the same fashion.

Also, there are some important differences in the way aggregate functions are processed, depending on whether or not they have a qualification. If an aggregate function has a qualification, there is a problem when using the basic algorithm just described. For example, suppose one wished to determine the number of programmers working in each department. The following aggregate function performs the desired computation:

count (EMP.name by EMP.dept **where** EMP.job = "programmer")

If a department has no employees with the job title "programmer," then that department will not appear in the result. To be correct, the result should contain these departments and show that they have zero programmers. To handle situations like the above, if an aggregate function has a qualification clause, then before beginning the computation, the temporary relation to hold the result of the aggregate must be initialized with a tuple for *every* by-list value that could take part in the aggregate. To perform the initialization, the system projects the attributes of the by-list into the result relation along with an initialized state value. The normal algorithm is then used to compute the aggregate function, and it will only be required to update tuples in the result relation, not create new ones.

In general, aggregates can be divided into the following classes:

scalar aggregates

non-unique
unique

aggregate functions

unqualified, non-unique
unqualified, unique
qualified, non-unique
qualified, unique

This chapter describes methods to maintain aggregates in each of these classes.

3.4.2. Fundamentals of Aggregate Maintenance

Before a materialized copy of any aggregate can be maintained, it must be initialized to the correct value for the current contents of the database. When an aggregate is installed, its value will be computed using conventional aggregate processing algorithms. From that point onward, the aggregate state will be updated as necessary when the database changes.

Essential to maintaining any materialized aggregate is a pair of incremental update functions (IUFs) to modify the aggregate on inserts and deletes, respectively. These functions will be called f_{insert} and f_{delete}. Thus, if V is the current state of the aggregate, and a value T is inserted into the set of values being aggregated, V is updated as follows:

$$V := f_{\text{insert}}(V,T)$$

Similarly, if a value T is deleted from the set being aggregated, V is updated as shown below:

$$V := f_{\text{delete}}(V,T)$$

For example, a simple pair of IUFs exist for the *average* aggregate. The state of the aggregate can be represented as a pair containing a count N and current average A. Thus, the incremental update functions are:

$f_{\text{insert}}([N,A], T)$
{
    return ([$N+1, (N \cdot A+T) / (N+1)$])
}

$f_{\text{delete}}([N,A], T)$
{
    return ([$N-1, (N \cdot A-T) / (N-1)$])
}

Update functions of this sort can be found for most of the common aggregates that are built into current database systems (e.g. **sum** and **count**) as well as many user-defined aggregates [Han84]. It is more difficult to find update functions for certain aggregates, such as **min** and **max**. The f_{insert} function for these aggregates is simple to create. However, a problem arises with deletion. Consider, for example, maintenance of a **max** aggregate. Suppose that the state of this aggregate

is represented by the current maximum value. If a value less than the current maximum is deleted, the f_{delete} function does not change the aggregate state. However, consider the case where a value equal to the current maximum is deleted. The aggregate state may change if there is no other value in the set being aggregated that is equal to the deleted value. If the aggregate state contains only one value, then there is no way of knowing what the new maximum is without completely recomputing the aggregate. A better way to represent the state of the aggregate **max** is to maintain a *state set* of up to k of the current largest values in the set being aggregated (e.g. a value k=10 might be chosen). In this way, some deletions of the maximum value can be tolerated. If there are many deletions of large values, the aggregate state set may become empty. When this sort of "underflow" occurs, it becomes necessary to recompute the aggregate and refresh the state set with k new values.

To make it possible to maintain aggregates like **min** and **max**, where error conditions such as underflow may occur, an error code INVALIDATE may be returned by f_{insert} and f_{delete}. If INVALIDATE is returned, the system marks the aggregate invalid, and will re-initialize the aggregate state at the end of the current transaction.

Update functions do not exist at all for some aggregates, particularly those that are sensitive to the order in which the values are processed. An example of such an aggregate is a checksum of a stream of bytes. In such cases, there is no hope of incrementally updating the aggregate – it must be completely recomputed. Thus, aggregates for which no incremental update functions exist are not considered here.

3.4.3. Scalar Aggregate Maintenance

3.4.3.1. Non-Unique

The general form of a non-unique scalar aggregate is

aggregate_operator (*aggregate_expression* **where** *qualification*)

Since this aggregate is non-unique, it is a function of all the tuples returned by the following query, without removing duplicates:

retrieve (value = *aggregate_expression*)
where *qualification*

The value of the relation retrieved by the query above will be called AggInput.

In general, the value retrieved by the command above is the same as that of the following relational algebra expression:

$$\pi_X(\sigma_Y(R_1 \times \cdots \times R_N))$$

where X is the target list, and Y is the qualification, i.e.,

$$X \equiv \{ \text{value} = \textit{aggregate\_expression} \}$$

and

$$Y \equiv \textit{qualification}$$

The semantics of the project operation π are modified slightly so that duplicates are not removed.

Suppose that sets $A_1 \cdots A_N$ are appended to each of relations $R_1 \cdots R_N$, and similarly, $D_1 \cdots D_N$ are deleted from $R_1 \cdots R_N$. Using an algorithm for incremental view update (either AVM or RVM) one can determine the values to be inserted into AggInput, and the values to be deleted from AggInput solely from $A_1 \cdots A_N$, $D_1 \cdots D_N$, and the contents of the base relations, $R_1 \cdots R_N$. This will yield two relations, A_{net} and D_{net}, containing the net changes (insertions and deletions respectively) to AggInput. A materialized copy of AggInput does *not* have to be maintained. This is extremely important, since it can save a large amount of storage, and it also significantly reduces the time required to maintain the aggregate.

To update the state V of a non-unique aggregate based on the values of A_{net} and D_{net}, the following steps are performed:

for each value a in A_{net}
  $V := f_{insert}(V,a)$

for each value d in D_{net}
  $V := f_{delete}(V,d)$

Performing this procedure after each base relation update that affects the aggregate will keep the aggregate in the correct state.

3.4.3.2. Unique

Aggregates that require duplicate removal can be maintained in a way similar to non-unique aggregates. To do this, a method is needed to determine when a unique value enters or leaves AggInput. This can be accomplished by maintaining a duplicate-free copy of AggInput at all times using a view-maintenance algorithm. Unfortunately, this makes the cost to maintain a unique aggregate significantly higher for a non-unique one. The version of AggInput maintained contains a set of <*value, count*> pairs, where *value* is a unique tuple value, and *count* is a duplicate count. The net change sets, A_{net} and D_{net}, used to update AggInput also contain <*value, count*> pairs. Furthermore, it is known that $A_{net} \cap D_{net}$ is empty. Duplicate-free versions of the net change sets, which will be called A_{net}' and D_{net}', are needed to correctly update the aggregate state. These can be produced while AggInput is being maintained. The algorithm for producing A_{net}' and D_{net}' during the process of incrementally updating AggInput is shown below:

```
for each value <a, count> in A_net do
      if a tuple <a, count'> exists in AggInput then
            set count' := count' + count
      else do
            add <a, count'> to AggInput
            add a to A_net'
      end
end
```

```
for each value <d, count> in D_net do
    if a tuple <d, count'> exists in AggInput then
        set count' := count' - count
        if count' becomes 0 then do
            remove <d, count'> from D_net
            add d to D_net'
        end
    else
        /* this should never happen */
        signal an error
    end
end
```

The values of A_{net}' and D_{net}' can be used to update the state of the unique aggregate (V) just as A_{net} and D_{net} are used for non-unique aggregates.

3.4.4. Aggregate Function Maintenance

It is assumed below that the value of the aggregate function to be maintained has already been initialized. The resulting relation, which will be called AggResult, is stored in the database. AggResult has the following schema:

AggResult (state, count, byvalue)

Here, *state* contains the current aggregate state for all tuples with a by-list value *byvalue*. The *count* field tells how many tuples in AggInput had a particular *byvalue*. The count is required so that tuples can be deleted from AggResult when the count goes to zero. As a simplification, the discussion below treats *state* and *byvalue* as individual attributes; in reality, they might be composed of more than one attribute each.

3.4.4.1. Unqualified, Non-unique

Unqualified, non-unique aggregate functions have the form

aggregate_operator (*aggregate_expression* by *by_list*)

The following retrieve command defines the input relation (AggInput) for this aggregate func-

tion:

retrieve (value = *aggregate_expression* , byvalue = *by_list*)

The net changes (A_{net} and D_{net}) to AggInput are determined using a view maintenance algorithm. The system does not have to maintain a materialized copy of AggInput since it is not needed to compute A_{net} and D_{net}. Again, this provides substantial cost savings.

Given A_{net} and D_{net}, AggResult can be updated by the following procedure, where INITIAL represents the initial state value for the aggregate:

```
for each tuple <a, by_value> in A_net do
  if a tuple <V, count, by_value> exists in AggResult then
    replace it with <f_insert(V,a), count + 1 , by_value >
  else
    insert into AggResult a tuple
       <f_insert( INITIAL , a), 1 , by_value>
  end if
end

for each tuple <d, by_value> in D_net do
  if a tuple <V, count, by_value> exists in AggResult then
    if count = 1 then
      there will be no tuples left in AggResult with this by_value, so
        delete <d, count, by_value> from AggResult
    else
      replace the tuple in AggResult with
        <f_delete(V,a), count - 1 , by_value>
    end if
  else
    There is no tuple with the same by_value in AggResult.
    This should never happen, so signal an error.
  end if
end
```

3.4.4.2. Unqualified, Unique

Unqualified, unique aggregates are maintained using the same algorithm for unqualified, non-unique aggregates, except that A_{net} and D_{net} are replaced by their duplicate-free counterparts, A_{net}' and D_{net}'. The duplicate-free change sets are found in the same way presented for unique scalar aggregates. This requires keeping a materialized copy of AggInput.

3.4.4.3. Qualified, Non-unique

Recall that the general form of an aggregate function is

aggregate-operator (*aggregate-expression* **by** *by-list*
[**where** *qualification*])

The semantics of aggregate functions require that a tuple be present in AggResult for each possible unique by-list value, even if no tuples for that value match the qualification. Thus, the system for maintaining a qualified aggregate has the following two parts:

1. a procedure to maintain the correct set of by-list values in AggResult
2. a procedure to update the aggregate state for existing by-list values in AggResult

When a database update occurs that causes the value of AggResult to change, procedure 1 is performed before procedure 2. These procedures are implemented as follows:

Procedure 1

The incremental view update algorithm is used to maintain a materialized copy of the answer to the following query:

retrieve unique (*by-list*)

The result of this query will be called ByValues. If a tuple $t = <B>$ enters ByValues, then the following tuple is created, and inserted into AggResult:

<state = INITIAL, count = 0, byvalue = B>

If a tuple $t = <B>$ leaves ByValues, the tuple with byvalue = B is deleted from AggResult.

Procedure 2

As database updates occur, the incremental view maintenance algorithm is used to find the net change sets, A_{net} and D_{net}, for the relation AggResult retrieved by the following query:

retrieve (value = *aggregate-expression*, byvalue = *by-list*)
where *qualification*

AggResult is not kept physically materialized. A_{net} and D_{net} are used as input to the algorithm described below. The only difference between this algorithm, and the one for unqualified aggregates is that here it is not necessary to insert or delete AggResult tuples since that has been handled in procedure 1.

for each tuple $<a$, by_value$>$ in A_{net} **do**
  if there is a tuple $<V$, count, by_value$>$ in AggResult
  replace it with $<f_{insert}(V,a)$, count + 1 , by_value $>$
end

for each tuple $<d$, by_value$>$ in D_{net} **do**
  if there is a tuple $<V$, count, by_value$>$ in AggResult **then**
    replace the tuple in AggResult with
      $<f_{delete}(V,a)$, count - 1 , by_value$>$
end

3.4.4.4. Qualified, Unique

Qualified, unique aggregate functions are handled using an algorithm identical to that for qualified, non-unique aggregates, except that the duplicate-free sets A_{net}' and D_{net}' are used in place of A_{net} and D_{net}. A_{net}' and D_{net}' are determined using the algorithm previously described. This requires a duplicate-free copy of AggInput to be maintained using a view maintenance algorithm.

3.5. QUEL Commands Containing Aggregates

Using QUEL that contains aggregates, more general database objects or views can be retrieved than using aggregate-free QUEL. QUEL retrieve commands (or view definitions) containing aggregates can be divided into several classes. These include

class 1: queries containing scalar aggregates, and no tuple variables at the top level (i.e., outside the scope of any aggregate)

class 2: queries containing scalar aggregates, and one or more tuple variables at the top level

class 3: queries containing aggregate functions

Methods for maintaining objects containing aggregates in each of these classes are discussed below.

Class 1

The general form of a class 1 query is the following:

retrieve $(l_1=f_1, \cdots, l_n=f_n)$
where $Q(f_{n+1}, \cdots, f_m)$

Above, all the expressions $f_1 \cdots f_m$ can contain any combination of constants and scalar aggregates. This type of query always retrieves either one tuple, or none. The following is an example of such a query:

retrieve (a=avg(EMP.salary))

A mechanism to materialize the results of such retrieve commands is a small extension of the schemes presented earlier for maintaining the results of scalar aggregates (both qualified and unqualified). To maintain the result of a class 1 query, every aggregate in it is maintained separately using techniques previously presented. Whenever one or more of the aggregates changes due to a database update, then the command is re-run using the currently materialized values of all the aggregates. The value returned replaces the previous stored result for the command.

Class 2

The next class of queries to consider are those containing simple aggregates, and one or more tuple variables at the top level. In general, class 2 queries have the same form as class 1 queries except that the functions $f_1 \cdots f_m$ can be functions of attribute.value pairs (e.g.

EMP.salary). An example is the query that retrieves all employees who earn the average salary:

retrieve (EMP.all) **where** EMP.salary = avg(EMP.salary)

The same algorithm for maintaining class 1 aggregates works for class 2.

The main problem with maintaining the result of a class 2 query is that any small change in the aggregate can require a large amount of recomputation to be done to update the result. For example, changing the salary of one employee can change the average salary, and thus completely change the value retrieved by command above. This property makes it unattractive to always maintain a materialized copy of such a command. An alternative that is likely to perform better is to always maintain the aggregates and use them to compute the final value of the object on demand. For example, if the materialized average salary is 20K, then when a user wants to access the value of the command above, the following command will be run instead:

retrieve (EMP.all) **where** EMP.salary = 20K

Class 3

The extent of change to an object whose definition contains an aggregate function is usually not as great as in the case of class 2 objects. Consider a general retrieve command

retrieve $(l_1{=}f_1, \cdots, l_n{=}f_n)$
where $Q(f_{n+1}, \cdots, f_m)$

where one or more of the f_i is an aggregate function. When processing a query of this form using conventional techniques, an aggregate function is computed separately, forming a temporary relation (AggResult). AggResult is then bound to the rest of the command by query modification. The modified query has the following structure:

retrieve $(l_1{=}f_1', \cdots l_n{=}f_n')$
where $Q(f_{n+1}', \cdots, f_m')$
and AggResult.byvalue = R.byvalue

Here, $f_1' \cdots f_m'$ are the same as $f_1 \cdots f_m$ except that all occurrences in $f_1 \cdots f_m$ of an

aggregate function are replaced by the expression

AggResult.Result

This represents the field of AggResult that contains the result of the aggregate. The modified command is just an ordinary query containing a collection of selects, projects and joins. To maintain the answer to this query, the system must perform the following steps:

1. find the net changes (A_{net} and D_{net}) for AggResult and any other relations participating in the query
2. apply any incremental update algorithm (e.g. AVM or RVM) to bring the stored result of the query to the correct state.

The net changes to AggResult are found as a byproduct of the aggregate function maintenance algorithm. Old and new AggResult tuples are saved by the algorithm instead of discarded. The net changes to base relations appearing in the command are found as usual.

3.6. Discussion

In this chapter a collection of algorithms has been presented for maintaining derived objects in relational database systems. The following is a summary of the view maintenance algorithms that were discussed:

1. dynamic, non-shared AVM
2. dynamic, shared AVM
3. static, non-shared AVM
4. static, shared AVM
5. RVM (static, shared)

Immediate and deferred versions of all the above are possible.

The main difference between the staticly optimized view maintenance algorithms and the dynamicly optimized ones is in planning and optimization cost. Staticly optimized algorithms pay a large planning cost once, and have no planning overhead at run time (i.e., at the time view maintenance is performed). Dynamicly optimized algorithms pay a large planning cost at run

time. A disadvantage of static algorithms is that they produce a fixed execution plan for maintaining views which will not necessarily be optimal at run time. An interesting topic for future research would be to find ways to construct an efficient Rete network for RVM or compiled execution plan for SAVM given statistics about the structure of the base relations and the frequency of database updates.

Shared algorithms have an advantage over non-shared algorithms because they avoid redundant computation. However, there is a relationship between planning cost and sharing – it costs more to construct a shared plan. This is particularly significant for dynamic algorithms because they pay a planning cost each time the view is updated. For static algorithms, it is less significant since planning cost is paid only once, and amortized over many changes to the view.

Deferred view maintenance algorithms may gain an advantage over immediate ones because using a deferred strategy, large sets of tuples may be processed a few times (i.e., after each query), rather than processing small sets of tuples many times (i.e., after each update). However, there is overhead in a deferred strategy for maintaining the net changes to the base relations in a data structure between transactions. These issues are discussed more fully in chapter 4.

The RVM algorithm has the same advantages and disadvantages as other shared, static view maintenance algorithms. A potential drawback of RVM is the time and storage required to maintain the memory nodes in the Rete network. A potentially serious drawback of RVM given the advent of high-speed multiprocessor hardware (e.g. SPUR [Hil86]) is that RVM cannot exploit parallelism easily because tokens must be propagated depth-first through the network. Using AVM, each expression to be computed could be assigned to a different processor.

This chapter also presented algorithms for maintaining database procedures and aggregates. Because database procedures are simply collections of queries, any view maintenance algorithm can be applied to maintain them. Algorithms for maintaining scalar aggregates and aggregate functions can be built on top of any view materialization algorithm. Non-unique scalar aggregates

(those that do not require duplicate removal) are very promising candidates for materialization because they do not require maintenance of the complete view defined by the qualification of the aggregate; only the result of the aggregate computation must be stored. Unqualified, non-unique aggregate functions also have this attractive property.

It is not essential to keep derived objects materialized. However, the decision of whether or not to materialize an object can have a large impact on performance. For materializing objects, the costs of query modification, algebraic view maintenance, Rete view maintenance, and caching are all different. Each may perform best, depending on environmental factors, including the frequency of queries and updates, and the structure of the database and the objects.

The relative performance of different algorithms for materializing derived database objects is analyzed in the next two chapters. The focus of chapter 4 is the performance of different methods for processing queries against views. Chapter 5 analyzes the performance of different algorithms for processing queries that retrieve the results of database procedures. In general, the following strategies are possible for processing queries against derived objects:

1. query modification
2. caching
3. differential maintenance

All the view maintenance algorithms presented in this chapter fall into the differential maintenance category. The following table shows the topics covered in chapters 4 and 5:

algorithm	ch. 4 (views)	ch.5 (procedures)
query modification	x	x
caching		x
differential maintenance	x	x
immediate, non-shared, static (AVM)	x	x
deferred, non-shared, static (AVM)	x	
immediate, shared, static (RVM)		x

Only staticly optimized algorithms are considered in chapters 4 and 5. Dynamicly optimized

algorithms are not considered because of their high run-time planning and optimization costs. The main goal of chapter 4 is to explore the differences between query modification, and deferred and immediate differential view maintenance techniques. In chapter 4, the model analyzed consists of a single large view. Algorithms that take advantage of shared subexpressions are not considered in chapter 4 because they would provide no advantage in this environment. Also, chapter 4 does not consider caching as an alternative since it is an inefficient technique for large views.

In chapter 5, caching is analyzed because it may be a worthwhile option for processing queries that retrieve the results of database procedures. Caching is more promising in this environment for the following reasons:

> (1) procedures are typically much smaller than views and hence less likely to be invalidated by updates, and
>
> (2) queries that access a database procedure read the entire result returned by the procedure, not just part of it, as is usually the case with queries against views.

Because there will typically be a large number of database procedures, many of which contain shared subexpressions, the use of a shared view maintenance algorithm may be advantageous. Hence, RVM is analyzed in chapter 5 as an example of a shared, static view maintenance algorithm. No deferred view maintenance algorithms are explored in chapter 5 because the deferred versus immediate issue is explored thoroughly in chapter 4. Chapter 4 shows that the immediate strategy is usually superior to deferred, although the performance difference between the two is small.

CHAPTER 4

VIEW MATERIALIZATION PERFORMANCE *

4.1. Introduction

In order to process queries against views, some sort of view materialization strategy is required. Conventional systems process queries against views using query modification, as described in Chapter 1 [Sto75]. This procedure translates queries referring to views into queries involving only the base relations. Recently, algorithms have been proposed for maintaining materialized copies of views (see [BLT86] and chapter 3). Given a materialized view, a query can be processed using the stored view directly, as if it were an ordinary base relation.

This chapter will analyze and compare the performance of the following algorithms for processing view queries:

1. query modification
2. differential maintenance
   a. immediate, non-shared, static (AVM)
   b. deferred, non-shared, static (AVM)

In the rest of this chapter, 2.a. will be called "immediate view maintenance" or simply "immediate," and 2.b. will be called "deferred view maintenance" or "deferred." The performance analysis will consider the differences between query modification and differential view maintenance in general, and the differences between immediate and deferred view maintenance in particular.

The types of views considered are those that can be defined using only the SELECT, PROJECT and JOIN operations. The cost of the various methods when applied to processing aggregate queries is also analyzed.

* A version of the material presented in this chapter has been published as a separate paper [Han87].

An important way to improve the performance of algorithms that maintain physically stored copies of views is to use a *screening* algorithm to test each tuple inserted into or deleted from the base relations. If a tuple passes the screening test, then its insertion or deletion may cause the state of the view to change, so the tuple must be used to try to update the view. If the tuple fails the screening test then it cannot cause the view to change, so it does not need to be used to refresh the view. In the scheme described in [BLT86] screening is done by substituting a tuple into a view predicate, which is then tested to see if it is still satisfiable. If so, the tuple passes the screening test, otherwise it fails. This test is performed for every tuple inserted into a relation, incurring a significant CPU cost, especially if there are many views.

An alternative screening mechanism that will usually be more efficient can be built using rule indexing [SSH86]. Using rule indexing, the index intervals covered by one or more clauses of the view predicate are locked using *t-locks*. When a tuple is inserted into the relation, if an index record containing a t-lock is disturbed, then the tuple passes the screening test. Otherwise, the tuple fails the test implicitly. Since this screening test can produce "false drops" (i.e., tuples which pass the screening test but do not satisfy the view predicate), a second stage screening test, substituting the tuple into the view predicate, is required. This strategy is assumed for both immediate and deferred view maintenance in the performance analysis of this chapter.

This chapter is organized as follows. The implementation of deferred view maintenance analyzed will be briefly described in Section 4.2. In Section 4.3, cost formulas for each of the algorithms are derived for three different view models:

1. selection-projection views
2. two-way natural join views
3. aggregates over selection-projection views

The performance of the algorithms is compared for each model. Finally, Section 4.4 presents conclusions, and suggests directions for future research on view materialization methods.

4.2. Deferred View Maintenance

To implement deferred view maintenance, a method must be found to save the net changes (A_i-net and D_i-net) for each base relation, R_i, for $1 \leq i \leq N$, over a period encompassing more than one transaction. Given mechanism to save the net changes, differential view maintenance can be done whenever desired (hence the name *deferred* view maintenance). It is assumed in this chapter that to refresh the materialized view on a deferred basis, A_i-net and D_i-net are calculated and then input to the static, non-shared AVM algorithm.

A previously developed technique called *hypothetical relations* [WoS83] can be adapted to the purpose of computing A_i-net and D_i-net. The basic algorithm for implementing hypothetical relations is briefly described below. Efficient implementation of hypothetical relations to support deferred view maintenance will be discussed after the basic algorithm is presented.

4.2.1. Hypothetical Relations

The hypothetical relation (HR) scheme uses three tables for each relation rather than one. Each relation has associated with it tables R, D and A, for base tuples, deletions and insertions, respectively [AgD83]. The data value of a tuple will simply be called "value." Each tuple will also have a unique identifier field "id." This yields the following schema for each relation:

R(id, value)
D(id, value)
A(id, value)

The true value of the relation (R_T) is $(R \bigcup A) - D$. The set difference operation "$-$" above has the normal meaning, based on all fields of the tuple, including id.

To append a tuple to R_T, a transaction inserts that tuple in A, placing the value of the system clock or other monotonicly increasing source in the id field. If duplicate-free semantics are desired, the system must ensure that the tuple is not already in $(R \bigcup A)-D$ before appending it to A. To delete the tuple from the relation, a copy of its value, including the id it had in R or

A, is placed in D. To modify an existing tuple, its old value will be put in D, and its new value in A. When retrieving data from R_T, queries are processed against both R and A, and any tuples found are checked to make sure they are not already in D (if they are, they are ignored).

Given this structure of the HR, the expressions for computing A-net and D-net from R, A and D are the following:

$$A\text{-net} := A - D$$
$$D\text{-net} := D - A$$

After a view refresh that uses A-net and D-net, the files used to store the hypothetical relation will be reset as follows:

$$R := (R \cup A) - D$$
$$A := \phi$$
$$D := \phi$$

4.2.2. Efficient Implementation of Hypothetical Relations

The problem with the most straightforward implementation of hypothetical relations is that retrieving a tuple from R requires three disk accesses rather than just one. To retrieve a tuple t from R_T using the HR scheme this way, an attempt must be made to read t from both R and A, and then D must be read to make sure that t has not been deleted.

Fortunately, a method developed in [SeL76] can be used to eliminate most page accesses when using a differential file. In this method, a *Bloom filter* [Blo70] consisting of an array of bits $B[1..m]$, with each entry initially zero, is used for each differential file. It is assumed that some subset of the fields of each record called the *key* uniquely identifies the record. For each record in the differential file, a hash function h mapping the key of a record to an integer in the range 1 to m is computed, and the corresponding entry in B is set to 1. Then, to test whether a record t is in the differential file, if $B[h(t.key)]=0$, t is not present; otherwise, if $B[h(t.key)]=1$, it might be present, so the differential file must be searched to see if it is there. Using the method pro-

posed in [SeL76] one can design a Bloom filter with any desired ability to screen out accesses to records not present in the differential file given a sufficient number of bits m.k

As another measure to help speed up accesses to the differential file, A and D for each relation R will be combined into a single file, AD. An extra attribute "role" will be added to tuples in AD to indicate whether they are appended or deleted tuples. This storage structure will speed up the majority of updates, which modify existing records without changing the key. For example, if AD is maintained using a clustered hashing access method on the key, then when a tuple t is updated to t' without having its key changed, t' will hash to the same page as t. Thus, a maximum of only three disk I/Os will be required to update a single tuple t in R given the key for t. The procedure to perform this update is the following:

I/O #1:
Read the tuple. (Check the Bloom filter to see if t could be in AD. If not, read t from R. Otherwise, read AD to see if it is there. If t is not in AD, read R. This might require 2 I/O's, but the probability can be made arbitrarily small by increasing m. Hence, only one I/O is counted here for simplicity.)

I/O #2:
Read the page where the new value of t (t') will lie in AD. (Place both t and t prime on the page. The role values of t and t' are "deleted" and "appended" respectively.)

I/O #3:
Write this page back to disk.

Three I/O's is only one more I/O than necessary to perform a one-tuple update using a single file to store the relation. If separate files for A and D were used, at least *five* I/O's would be required rather than three since R must be read, and A and D must both be read and written.

In the remainder of the chapter, the sets of inserted and deleted tuples will still be referred to as A and D, even though they are stored in the AD table. It is assumed that AD will be partitioned to form A and D when necessary.

4.3. Performance Comparison

Each of the view materialization methods presented will have different performance characteristics. For example, because query modification pays no overhead for base relation updates, it will clearly be the algorithm of choice if the ratio of updates to view queries is very high. On the other hand, if this ratio is low, then a view maintenance algorithm will probably perform best. This section discusses in detail the factors affecting performance and derives cost functions for each view materialization algorithm for view models 1-3.

4.3.1. Description of View Models

The structure of view models 1-3 is as follows:

model	*view structure*
Model 1	selection and projection of a single relation R
Model 2	natural join of two relations, R_1 and R_2, on a key field
Model 3	aggregates (e.g. sum, average) over a Model 1-type view

The views have the following definitions:

Model 1:

retrieve (R.fields)
where $C_f(R)$

Here, the target list projects exactly one half of the attributes of R, and the qualification clause $C_f(R)$ restricts relation R with selectivity f.

Model 2:

retrieve (R_1.fields, R_2.fields)
where $C_f(R_1)$
and $R_1.a = R_2.b$

For Model 2, the target list projects one half of the attributes of both R_1 and R_2, and the clause $C_f(R_1)$ restricts R_1 with selectivity f.

Model 3:

retrieve (a = agg(R.b where $C_f(R)$))

For Model 3, the clause $C_f(R)$ again restricts R with selectivity f.

Only two types of operations will be considered in the models: updates to the base relations, and queries to the view. No other operations are relevant to the performance issue being studied. It is assumed that exactly k update operations, and q queries to the view will be run. For each model, a formula for the *average* cost per query, over all k updates and q queries, will be derived.

The access methods of the relations involved are shown in figure 4.1. Generous assumptions will be made for all view materialization schemes regarding how queries and other operations are performed using these clustered indexes. Since these performance benefits will be given to all algorithms, the results should not be biased toward any one scheme.

The parameters important to the analysis are shown in Figure 4.2. The default values of these parameters, which will be used unless stated otherwise, are shown in Figure 4.3.

relation(s)	*access method*
R, R_1	B^+-tree primary index on field used in view predicate terms $C_f(R)$ and $C_f(R_1)$
R_2	hashed primary index on join field (b)
materialized view (V)	B^+-tree primary index on field used in view predicate terms $C_f(R)$ and $C_f(R_1)$
differential file (AD)	hashed primary index on a key field

Figure 4.1. Access methods of relations in performance model

parameter	*definition*
N	number of tuples in relation
S	bytes per tuple
B	bytes per block
b	total blocks ($b = NS/B$)
T	number of tuples per page ($T=B/S$)
d	number of bytes in a B^+-tree index record
k	number of update transactions on base relation
l	number of tuples modified by each update transaction
q	number of times view queried
u	number of tuples updated between view queries ($u=kl/q$)
P	probability that a given operation is an update ($P=k/(k+q)$)
f	view predicate selectivity for Model 1
f_v	fraction of view retrieved per query
f_{R_2}	size of R_2 as a fraction of R_1
C_1	CPU cost to screen a record against a predicate in milliseconds (ms)
C_2	Cost in ms of a disk read or write
C_3	Cost in ms per tuple per transaction to manipulate A and D data structures in immediate view maintenance

Figure 4.2. View Materialization Cost Parameters

N	100,000	f	.1
S	100	f_v	.1
B	4,000	f_{R_2}	.1
k	100	C_1	1
l	25	C_2	30
q	100	C_3	1
d	20		

Figure 4.3. Default Parameter Values

4.3.2. Model 1 Cost Analysis

In Model 1, the view is formed by projecting exactly one half of the attributes of tuples from

R, and applying a predicate with selectivity f. Thus, the result will contain f times N tuples. The value that will be measured for each view maintenance scheme is the average cost of a query that retrieves a fraction f_v of the tuples in the view.

4.3.2.1. Cost of Deferred View Maintenance Assuming Model 1

In deferred view maintenance, it is assumed that the view is refreshed every time it is queried. After the refresh is finished, the result of the query is computed. The average cost of a query to the view, which will be called $\text{TOTAL}_{\text{deferred1}}$, has several components. The first is the cost to read the result of the query from the copy of the view stored on disk. The second is the cost to refresh the view. The third is the cost to screen incoming and deleted tuples to see if they might affect the state of the view. Finally, the fourth is the cost to maintain the hypothetical relation(s). The average value of each of these costs are added together to get the average cost per query, $\text{TOTAL}_{\text{deferred1}}$. In summary,

$\text{TOTAL}_{\text{deferred1}} =$
(cost to retrieve result of query from stored copy of view)
+ (cost to refresh the view)
+ (average cost per query to screen tuples to see if they affect view)
+ (average cost per query to maintain hypothetical relation)

It is assumed that no duplicates are formed by projecting half the attributes, so the view has fN tuples and $fb/2$ pages. A fraction f_v of the view is read during each access, requiring $ff_vb/2$ page reads, at a cost of C_2 each. One search of the B^+-tree will also be necessary to locate the position in the view to begin scanning. Since there are d bytes per index record, the height of the B^+-tree, not including the data pages, is determined as follows. The number of index records per page, and thus the index fanout, is B/d. There is one index record for each of the fN tuples in the view. Assuming as a simplification that all pages are packed full, the height of the view index (H_{vi}) is thus

$$H_{vi} = \lceil \log_{\lfloor B/d \rfloor} fN \rceil$$

Additionally, each tuple read from the view must be screened against the query predicate, at a

cost of C_1, for a total cost per view access of $C_1 f_v f N$. Thus, the total cost C_{query1} to query a materialized view is

$$C_{query1} = C_2 \frac{f f_v b}{2} + C_2 H_{vi} + C_1 f f_v N$$

The next cost to consider is that for the hypothetical relation overhead. It is only necessary to measure the cost in excess of that required to perform normal base relation updates. As a simplification, the assumption is made that only tuples in R are updated, and never tuples in AD. The cost to maintain the HR for a single insertion into R in this situation is the following:

1. read the original tuple from R
2. read the page in AD where the modified tuple will be placed
3. write this page in AD

Step (2) is the only extra I/O required compared with keeping R in a single file and updating it directly. The normal cost to update R would be one read and one write, or $2C_2$, per tuple updated. If the cost of step (2) is averaged over all queries and updates, the cost per query to maintain the HR is at most the cost of one I/O (C_2) times the number of tuples update per view query (u). The total cost is likely to be somewhat less than this, however, since AD often has a small number of pages, and there are l tuples modified per transaction. The cost can be modeled more accurately using a function for estimating the number of pages touched when accessing k out of n records in a file occupying m disk pages. This function, which will be called $y(n,m,k)$, has been previously derived [Yao77]*. The number of tuples in AD will be twice the number of

*Given that there are n total records on m blocks, a formula giving the expected number of blocks that will be accessed to modify k records is known as the Yao function, denoted by $y(n,m,k)$ [Yao77]. Let C_a^b be the number of ways that b items can be selected from a items ($a \geq b$). If the number of records per block is $p = n/m$, then the formula giving the expected number of block accesses is C_k^{n-p}/C_k^n. An alternative to the above called *Cardenas' approximation* that is very close if the blocking factor is large (e.g. $n/m > 10$) is $m(1-(1-1/m)^k)$ [Car75]. Cardenas' approximation gives good results unless m approaches 1. Clearly, any stored object must occupy at least one page. The approximation used in this thesis is that if $k \leq 1$, the expected number of pages touched is k. If k is greater than 1, and m is less than 1, the expected number of pages touched is 1. Otherwise, if m is less than some upper bound U ($U=2$ is used) and k is more than 1, the minimum of k and m is returned. If none of the above conditions apply, Cardenas' approximation is

tuples updated per view query ($2u$). The number of pages in AD will thus be $2u$ divided by the number of tuples per page (T). The number of pages in AD touched per transaction is thus $y(2u,2u/T,l)$. Averaged over q queries and k updates, the total cost of the extra accesses to AD is thus the following:

$$C_{AD} = C_2 \frac{k}{q} y(2u, \frac{2u}{T}, l)$$

Consider now the cost to refresh the view V once. This first involves the cost to read all of AD. Since u tuples are updated per view query, AD has approximately $2u$ elements. There are T records per page, so AD has $2u/T$ pages. Thus, the cost C_{ADread} of reading AD is

$$C_{ADread} = C_2 \frac{2u}{T}$$

Another cost is incurred to screen updates to see whether they have a chance of affecting the view. Recall that to screen incoming tuples to see whether they can affect a view, rule indexing is used in combination with a more stringent satisfiability test. For the view maintenance methods analyzed, it is assumed that the screening is performed as follows:

> **if**
> (1) a tuple breaks a t-lock for the predicate of view V, and
> (2) the predicate for V with t substituted into it is still satisfiable,
> **then**
> a marker indicating this is placed on t.

In both the deferred and immediate view update algorithms, a tuple will be used to update a stored view V only if the tuple has a marker for V. A fraction f of the u tuples inserted into R per query will conflict with a t-lock set for V in step (1) above, and thus must be passed on to step (2). Step (1) has essentially no overhead, and step (2) costs C_1. Thus, the average overhead per query to screen tuples to see if they affect V is:

---

used. This approach gives an accurate estimate of the expected number of pages touched for a wide range of parameter settings.

$C_{screen} = C_1 fu$

Also, approximately fu tuples per query will be inserted into and deleted from the view, respectively, for a total of $2u$ tuple updates. Each insertion or deletion from the view requires reading the B^+-tree view index, and reading and writing a data block. However, somewhat less than $2fu$ *pages* of the view may actually have to be updated during a refresh, since there may be more than one record per block in the view. Using the Yao function, since there are fN tuples and $fb/2$ blocks in the view, the number of view blocks accessed (X_1) is approximately

$$X_1 = y(fN, \frac{fb}{2}, 2fu)$$

Each access requires reading the index, reading and writing a data block, and writing a leaf-level index block (splits of internal index pages are infrequent, so their cost will be ignored as a simplification). This requires 3 I/Os, plus a number of I/Os equal to the height of the index on V (H_{vi}). Thus, the cost to refresh the view, $C_{def\text{-}refresh1}$, is as follows

$$C_{def\text{-}refresh1} = C_2 (3+H_{vi}) X_1$$

The following is the final expression for the cost per query to the view V using deferred refresh:

$$TOTAL_{deferred1} = C_{AD}+C_{ADread}+C_{query1}+C_{def\text{-}refresh1}+C_{screen}$$

4.3.2.2. Cost of Immediate Assuming Model 1

The cost per view access of performing immediate view maintenance, $TOTAL_{immediate1}$, is as follows:

$TOTAL_{immediate1} =$
(cost to query view)
+ (total cost to modify stored view) / (# of view accesses)
+ (total cost to screen tuples inserted into R to see if they should enter view) / (# of view accesses)
+ (overhead per query to maintain A and D sets in a data structure during transaction processing)

The cost C_{query1} to query the view is the same as for deferred view maintenance. The cost to

update the stored view when a transaction modifies R, which will be called $C_{imm\text{-}refresh1}$, is computed much like $C_{def\text{-}refresh1}$. The difference is that approximately $2fl$ tuples in the view must be modified *once per transaction*, rather than modifying $2fu$ view tuples once per query. Since some of these $2fl$ tuples may lie on the same page, the number of view pages touched (X_2) can be estimated using the Yao function as follows:

$$X_2 = y(fN, \frac{fb}{2}, 2fl)$$

Similar to the case for deferred view maintenance, updating a tuple in V requires a B^+-tree search, the read and write of a data block, and the write of an index block. This requires $(3+H_{vi})$ I/Os for each view page touched, as before. Since there are k updates for every q queries, the average cost per query to update the view is:

$$C_{imm\text{-}refresh1} = \frac{k}{q} C_2 (3+H_{vi}) X_2$$

The cost C_{screen} to screen the kl tuples inserted into R is unchanged.

Finally, since immediate view maintenance must update the view after every transaction, the data structures used to maintain the A and D sets must be reset once per transaction. The overhead per query to do this, which will be called $C_{overhead}$, will be estimated as C_3 for each of the fl tuples in A and D, multiplied by the number of updates per query (k/q), i.e.,

$$C_{overhead} = (C_3 2fl)\frac{k}{q}$$

This gives the following expression for the total cost of immediate view maintenance:

$$TOTAL_{immediate1} = C_{query1} + C_{imm\text{-}refresh1} + C_{screen} + C_{overhead}$$

4.3.2.3. Cost Using Query Modification Assuming Model 1

The cost of using query modification rather than materializing the view in advance is considered here (this option will perform best in some circumstances, e.g. if the ratio of updates to queries is high). Three different methods for retrieving the view from R will be considered:

(1) a clustered (primary) index scan for which no extra tuples must be read (**clustered**)

(2) an unclustered (secondary) index scan (**unclustered**)

(3) a sequential scan of the entire relation (**sequential**)

Using a clustered index scan, the number of pages that must be read from R is equal to the size of the view, which is fb, times the fraction of the view retrieved, f_v. The number of tuples retrieved is ff_vN, and each of these tuples must be tested against the view predicate at a cost of C_1. Also, to find the point to begin the scan, a search of the B^+-tree index on R is required. The height of the index is

$$H_i = \lceil \log_{\lfloor B/d \rfloor} N \rceil$$

Thus, for the clustered scan (1), the total cost to retrieve the view per access is

$$\text{TOTAL}_{\text{clustered}} = C_2 bff_v + C_1 Nff_v + C_2 H_i$$

Using an unclustered scan (2), a larger number of pages must be read from R. Searching for ff_vN tuples out of a total of b pages will require approximately $y(N,b,Nff_v)$ reads. The system must still do an index search and test Nff_v tuples against the view predicate. Thus, the total cost for case (2) is

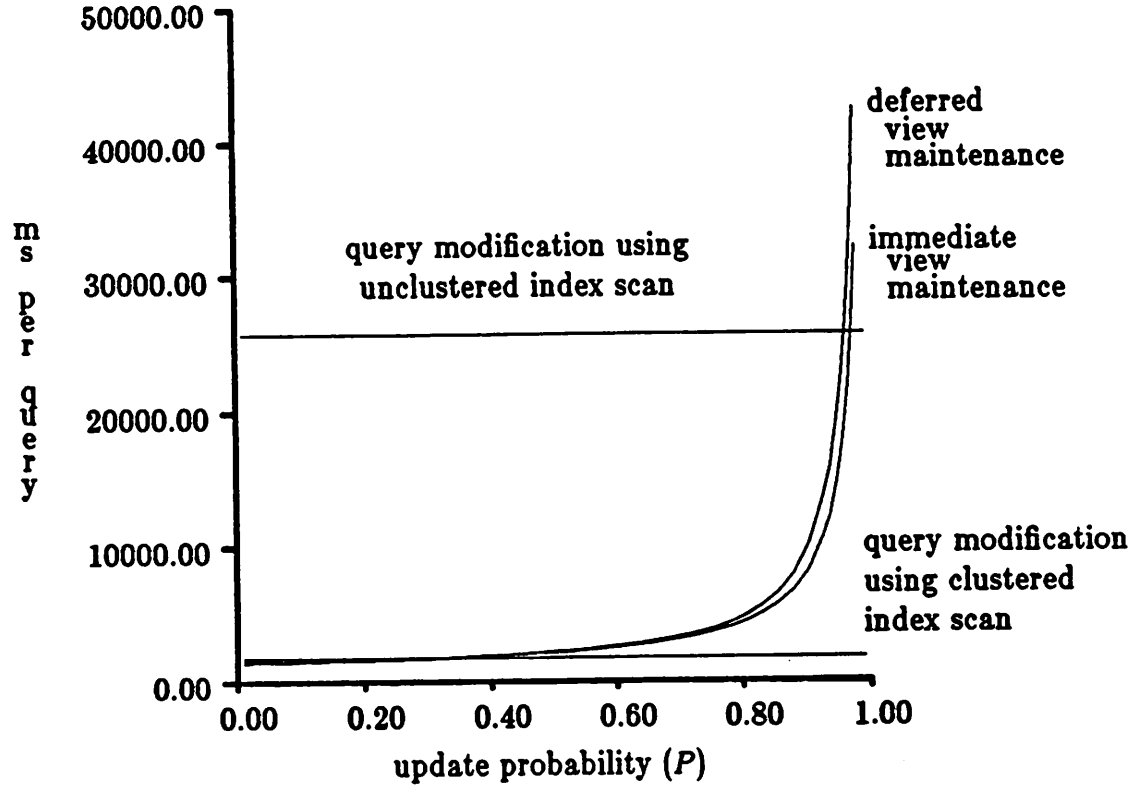
$$\text{TOTAL}_{\text{unclustered}} = C_2 \cdot y(N,b,Nff_v) + C_1 Nff_v + C_2 H_i$$

Using a sequential scan of the entire relation (3), all b pages must be read, and all N tuples must be screened against the view predicate, resulting in the following total cost:

$$\text{TOTAL}_{\text{sequential}} = C_2 b + C_1 N$$

4.3.3. Performance Results for Model 1

To indicate the differences in cost with respect to the probability P that an operation is an update, Figure 4.4 plots the total cost of deferred, immediate, clustered and unclustered versus P for the standard parameter settings (sequential is not shown since it is off the scale). This setting of the parameters models a situation where the view contains 10,000 tuples, and each query



Model 1 (1 table, simple selection/projection view): Total cost of a view query versus update probability P.

Figure 4.4. Model 1: Query Cost

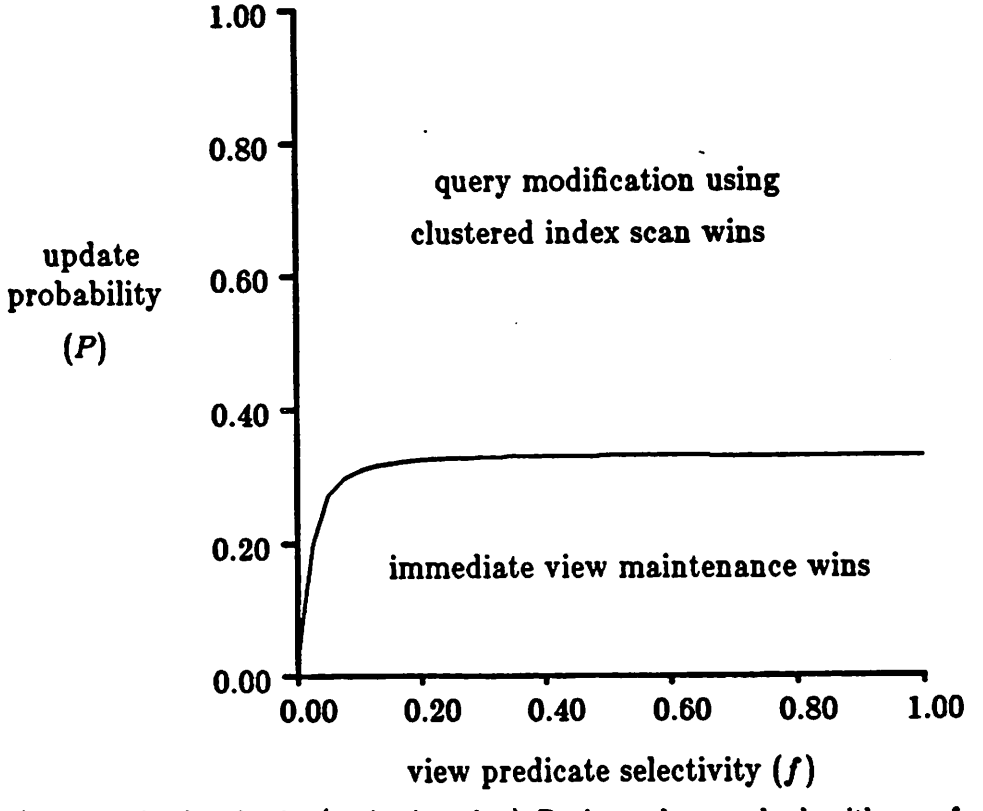
retrieves 1,000 tuples. In this situation, query modification using a clustered access path has performance equal or superior to deferred and immediate. One would expect that clustered would perform well here since the number of pages that must be read is small when using a clustered index. The only advantage that deferred and immediate have over clustered is that there are twice as many tuples per page in the view compared with the base relation. However, the extra overhead paid by deferred and immediate to maintain the materialized copies of the view offsets this. Deferred and immediate would perform even less well compared to query modification if the view projected all the attributes of relation R instead of only half of them.

It is surprising that deferred and immediate view maintenance have almost identical cost under these circumstances. Once reason for this is that for low values of P, materialization methods have nearly equal cost for virtually any parameter setting. This occurs since for low update probability, a large fraction of the cost of both algorithms is for processing queries against the materialized view, and both algorithms do this the same way. The fraction of the cost that is for updating views is inconsequential for small P, regardless of the view maintenance algorithm used. Another cause of the close match is that the hypothetical relation overhead in deferred view maintenance counteracts the other advantages it holds over immediate view maintenance. If more than one disk is available, and I/O operations can be issued concurrently by a program, then it would be possible to significantly decrease the cost of maintaining hypothetical relations (e.g. by putting R, A and D on separate disks and reading from them simultaneously). This would give deferred maintenance an advantage over the immediate scheme for a wider range of parameter settings. However, these assumptions are not made here since they would require extra hardware, and operating system functionality not readily available in all computer systems.

Assuming the view is maintained with a clustered index on a commonly used access path, the view materialization methods are significantly superior to query modification when only an unclustered access path is available on the base relation. This has implications for physical database design, since a materialized view could be clustered on one attribute, and the base relation on another. In this situation, the query optimizer could chose to process a view query in one of two ways, depending on the query predicate. If the predicate could be processed most efficiently using the clustered index on the base relation, query modification would be chosen to execute the query. Otherwise, the query could be processed against the materialized view, using the clustered view index as an alternate access path.

An interesting tradeoff among the algorithms centers around the parameters f, P, and f_v. To illustrate the relationship between these parameters, Figure 4.5 plots the region where each

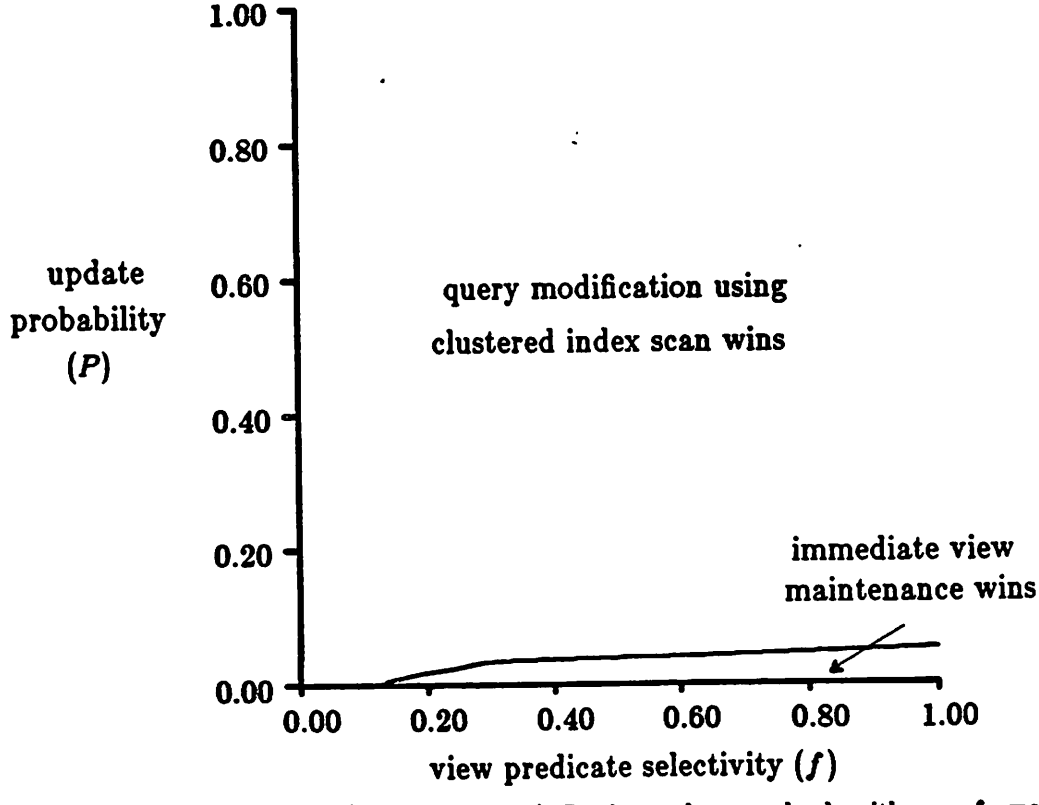
algorithm has lowest cost for different values of P and f, with f_v fixed at .1. Although deferred is never the most efficient algorithm under these parameter settings, larger values for f improve the performance of deferred relative to immediate view maintenance. This occurs due to the nature of the Yao function, combined with the fact that increasing f increases the size of A and D proportionately. Larger values of P tend to favor the algorithm with the least overhead per update transaction (i.e., query modification). Reducing the total fraction f_v of the view retrieved also tends to favor using query modification, since the overhead of the view maintenance schemes



Model 1 (1 table, simple selection/projection view): Regions where each algorithm performs best for f versus P (fraction of view retrieved (f_v) = .1).

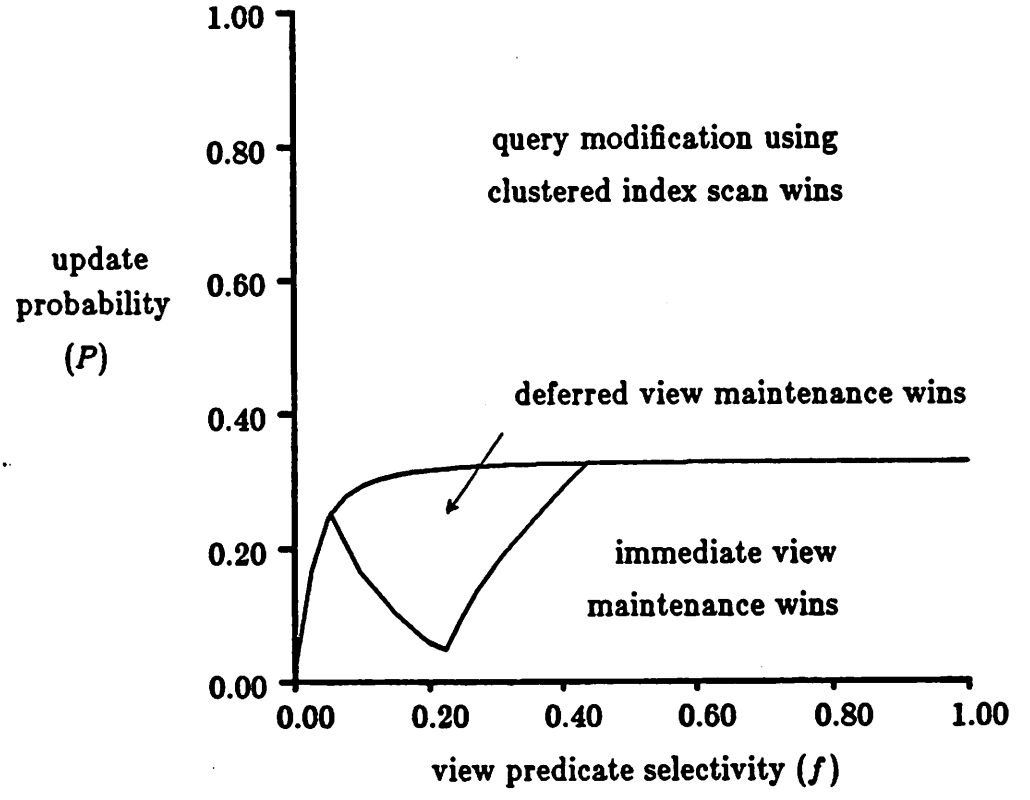
Figure 4.5. Model 1: Algorithm Comparison

is independent of f_v, but the cost per query decreases with f_v. When the value of f_v is lowered to .01, as shown in Figure 4.6, query modification using a clustered index scan performs best over an even larger area. In Figure 4.7, C_3, the overhead per tuple for maintaining the main-memory-based A and D sets in immediate view maintenance was increased from 1 to 2 ms, while setting f_v=.1. The affect of this change can be seen by comparing Figure 4.5 and Figure 4.7. The fact that deferred view maintenance now performs best in part of Figure 4.7 shows that the cost of the view materialization methods is very sensitive to the overhead for maintaining the A and D sets.



Model 1 (1 table, simple selection/projection view): Regions where each algorithm performs best for f versus P (fraction of view retrieved (f_v) = .01).

Figure 4.6. Model 1: Algorithm Comparison



Model 1 (1 table, simple selection/projection view): Regions where each algorithm performs best for f versus P (overhead per tuple for maintaining A and D data structures in immediate view maintenance (C_3) = 2, fraction of view retrieved (f_v) = .1).

Figure 4.7. Model 1: Algorithm Comparison

4.3.4. Model 2: 2-Way Join View

In this section, the performance of the different view maintenance algorithms is compared for view model consisting of a two way join. It is assumed that every tuple of R_1 that matches restriction clause C_f in the view definition joins to exactly one tuple in R_2, so V has $f \cdot N$ tuples total. Also, both R_1 and R_2 contain tuples of size S bytes, and only half the attributes of each relation are projected in the target list of the view definition. Thus, the tuples in V also contain S bytes each. The query and update activity assumed is the same as for Model 1, except that all

updates are to R_1 rather than R (R_2 is never updated).

4.3.4.1. Cost of Deferred Assuming Model 2

For Model 2, the cost per query of doing deferred view maintenance is determined as follows:

$\text{TOTAL}_{\text{deferred2}}$ = (cost to read AD)
+ (cost to refresh view)
+ (cost to query view)
+ (cost per query to screen new tuples against view predicate)

The costs C_{AD} and C_{ADread} of updating and reading the HR, respectively, from Model 1 are unchanged for Model 2. The cost to refresh the view before it is queried (using deferred view maintenance), which will be called $C_{def\text{-}refresh2}$, will be determined as follows. To refresh V, the value of the following expression must be computed (the notation $V(X,Y)$ means the expression for V evaluated with X and Y in place of R_1 and R_2, respectively:

$$V(R_1,R_2) \bigcup V(A_1,R_2) - V(D_1,R_2)$$

The $V(R_1,R_2)$ term is already computed and stored as the previous version of the view (V_0). No terms containing A_2 and D_2 are shown since R_2 is never updated. Thus, only $V(A_1,R_2)$ and $V(D_1,R_2)$ must be computed. Recall that there is a clustered hashing index on R_2 that can be used as an access path to join tuples in A_1 and D_1 to R_2. The cost to join the A_1 and D_1 sets to R_2 is determined as follows: R_2 has $f_{R_2}N$ tuples and $f_{R_2}b$ pages, and there are u tuples in each of A_1 and D_1 at refresh time. Thus, the total number of pages that must be read from R_2 to perform these two joins is

$$X_3 = y(f_{R_2}N, f_{R_2}b, 2fu)$$

It is assumed that pages read for the first join stay in the buffer pool for the second.

There is also a CPU cost of C_1 for matching each of the $2u$ tuples in A_1 and D_1 with the joining tuple in R_2. Furthermore, for each joining tuple, a page must be read and written from the stored view. Using the Yao function, since the view has fN tuples of size S bytes, and a

fraction f of the tuples in A_1 and D_1 join to exactly one tuple in R_2, the actual number of view pages that will be updated is approximately

$$X_4 = y(fN, fb, 2fu)$$

Each page update requires reading the B^+-tree index on the view, as well as reading and writing the data page, and writing the index leaf page (i.e., $3+H_{vi}$ I/Os). Thus, the total cost $C_{def\text{-}refresh2}$ to update the view every time it is queried is:

$$C_{def\text{-}refresh2} = C_2X_3 + C_1 2u + C_2(3+H_{vi}) \cdot X_4$$

When the view is queried, both deferred and immediate view maintenance pay the same cost, C_{query2}. This consists of searching the view index to find the starting point, and then performing a clustered index scan to retrieve a fraction f_v of the view. This costs C_2 per page, and C_1 per tuple scanned. Summing the cost of the index search and scan yields the following expression for C_{query2}:

$$C_{query2} = C_2H_{vi} + C_2f_vfb + C_1f_vfN$$

Both deferred and immediate view maintenance pay an average screening cost of C_{screen} per query to the view. Given $C_{def\text{-}refresh2}$, C_{query2}, and C_{screen}, the expression for the total cost using deferred view maintenance assuming Model 2 is

$$TOTAL_{deferred2} = C_{ADread} + C_{def\text{-}refresh2} + C_{query2} + C_{screen}$$

4.3.4.2. Cost of Immediate View Maintenance Assuming Model 2

The cost $TOTAL_{immediate2}$ of doing immediate view maintenance combined with rule indexing in Model 2 is

$TOTAL_{immediate2} =$
(cost per query to update view)
+ (cost to query view once)
+ (total overhead per query to maintain A and D sets)
+ (cost to screen new tuples against view predicate)

To find the cost per query $C_{imm\text{-}refresh2}$ of maintaining the materialized view, the cost to refresh

the view after each transaction must first be found. The components of this refresh cost are the I/O cost of reading the pages of R_2 to which tuples in A_1 and D_1 join and reading and writing modified pages of V, plus the CPU cost of handling each tuple in A_1 and D_1. Since A_1 and D_1 both contain l tuples at the end of each transaction, and a fraction f of these match the view predicate and must be joined to R_2, the number of pages that must be read from R_2 is

$$X_5 = y(f_{R_2}N, f_{R_2}b, 2fl)$$

Each tuple in A_1 and D_1 joins to some tuple in R_2, so each causes one tuple to enter or leave V. The number of modified pages of V is

$$X_6 = y(fN, fb, 2fl)$$

Again, for each of these pages, the index on V must be read, the page must be read and written, and an index leaf page is written, requiring $3+H_{vi}$ page I/Os. There is also a CPU cost of C_1 for handling each of the $2l$ tuples in A_1 and D_1. Averaging the per-transaction cost of updating V over k transactions and q queries, the estimated cost per query is as follows:

$$C_{\text{imm-refresh2}} = \frac{k}{q}(C_2X_5 + C_2(3+H_{vi})X_6)$$

Given $C_{\text{imm-refresh2}}$ and C_{query2}, the following expression shows the total cost of immediate view maintenance using rule indexing, assuming Model 2:

$$\text{TOTAL}_{\text{immediate2}} = C_{\text{imm-refresh2}} + C_{\text{query2}} + C_{\text{overhead}} + C_{\text{screen}}$$

4.3.4.3. Cost Using Query Modification Assuming Model 2

Another important cost to measure is that to materialize a view directly from the base relations. A frequently used join strategy called *nested-loops* (or *loopjoin*) involves scanning one (outer) relation, and for each of its elements, searching the other (inner) relation to find all joining tuples. If an index is present on the join field of the inner relation, it can be used for the search.

It is assumed that the nested-loops join algorithm is used to join R_1 and R_2 in Model 2. R_1 will be the outer relation, and R_2 will be the inner one. Since there is a hash index on the join

field of R_2, it will be used for the inner search. The assumption is made that pages of R_2 stay in the buffer pool throughout the computation of the join after they are read the first time. With the advent of very large main memories, this is reasonable since R_2 contains only $f_{R_2}NS$ bytes, which is approximately 1 Mbyte using the standard parameter settings. Under these assumptions, nested loop join has the following cost components, with the actual costs shown below:

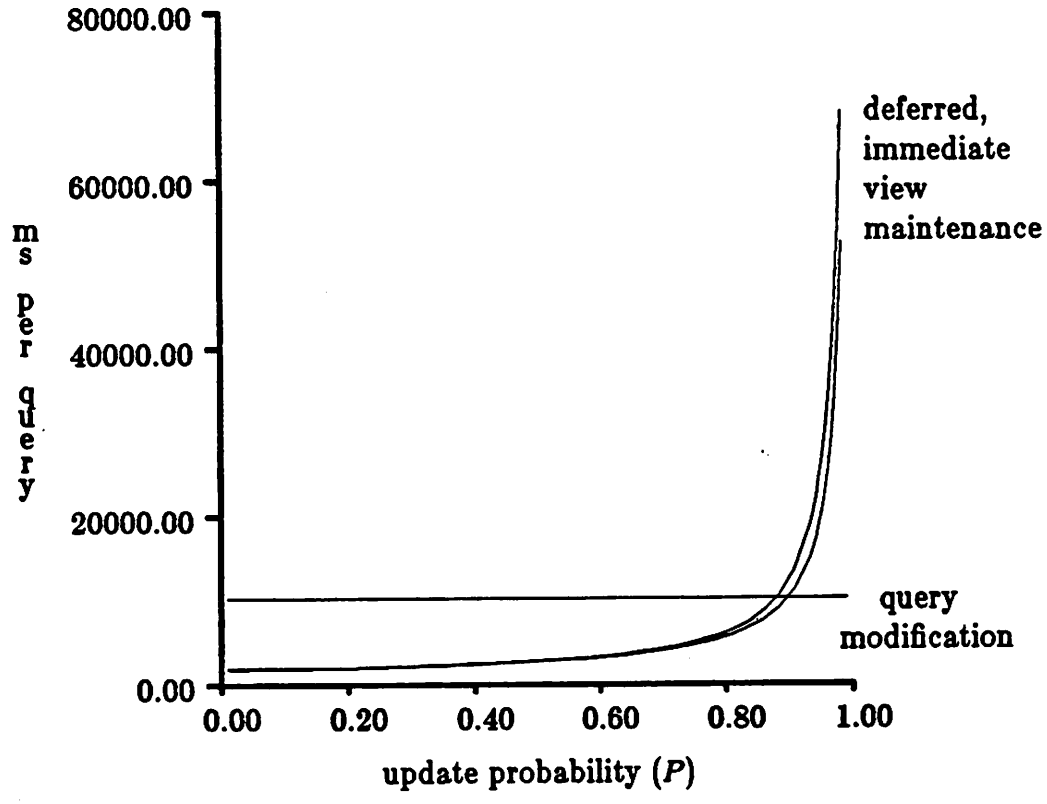
cost component	*actual cost*
read B^+-tree on R_1	C_2H_i
read part of R_1 using clustered scan	C_2ff_vb
CPU cost to screen R1 tuples scanned	C_1ff_vN
read pages from R_2 using hash index	$C_2y(f_{R_2}N, f_{R_2}b, ff_vN)$
CPU cost to match R_1 tuples to R_2 tuples	C_1Nff_v

Summing the above cost components gives the following formula $\mathrm{TOT}_{\mathrm{loopjoin}}$ for the total cost to compute the join using nested loops:

$$\mathrm{TOT}_{\mathrm{loopjoin}} = C_2\lceil \log_{\lfloor B/b \rfloor} N \rceil + C_2ff_vb + C_2y(f_{R_2}N, f_{R_2}b, ff_vN) + 2C_1Nff_v$$

4.3.5. Performance Results for Model 2

The actual cost per query for deferred view maintenance, immediate view maintenance, and query modification using a nested loop join with an index on the inner relation are plotted in Figure 4.8 using the standard parameter settings. This figure indicates that the results for Model 2 are significantly different than to those for Model 1. When the view joins data from more than one relation, differential view maintenance algorithms (deferred and immediate) perform better relative to query modification. By maintaining a materialized copy of the view, the query cost is greatly reduced, since each result tuple is stored on exactly one page. In effect, maintaining the view serves as an effective way of *clustering* related data on the same page. However, as the update probability P increases, the overhead for maintaining the materialized view overwhelms the advantage gained by clustering, so query modification becomes more attractive. Also, similar



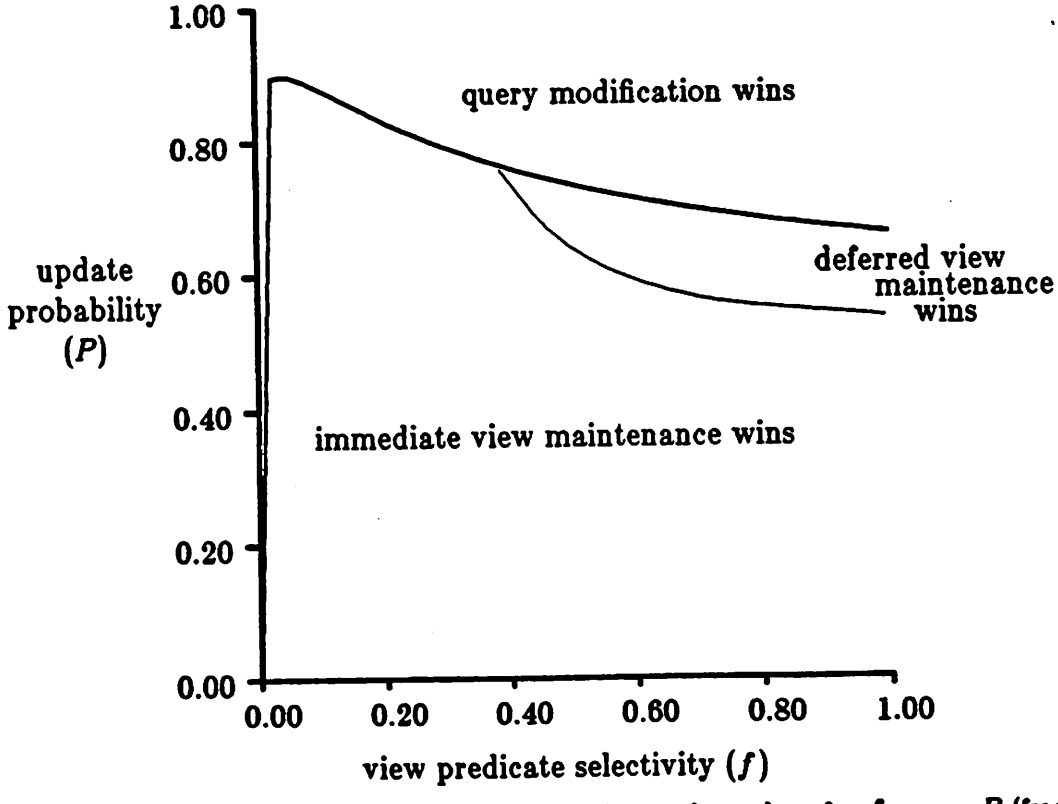
Model 2: Total cost per query using deferred view maintenance, immediate view maintenance, and query modification (fraction of view retrieved (f_v) = .1).

Figure 4.8. Model 2: Query Cost

to Model 1, as the fraction of the view retrieved (f_v) is decreased, the advantage of query modification grows. Query modification performs better for smaller values of f_v because making f_v smaller reduces the query cost, while the amount of overhead paid by deferred and immediate algorithms for updating the view stays the same.

An important special case to consider is when the view is large, and the queries read a small amount of data. For example, this special case arises using the standard EMP and DEPT relations, and view ED joining the two. The majority of queries in this situation might retrieve only a single tuple from ED. Also, updates usually change only one EMP tuple. This example was

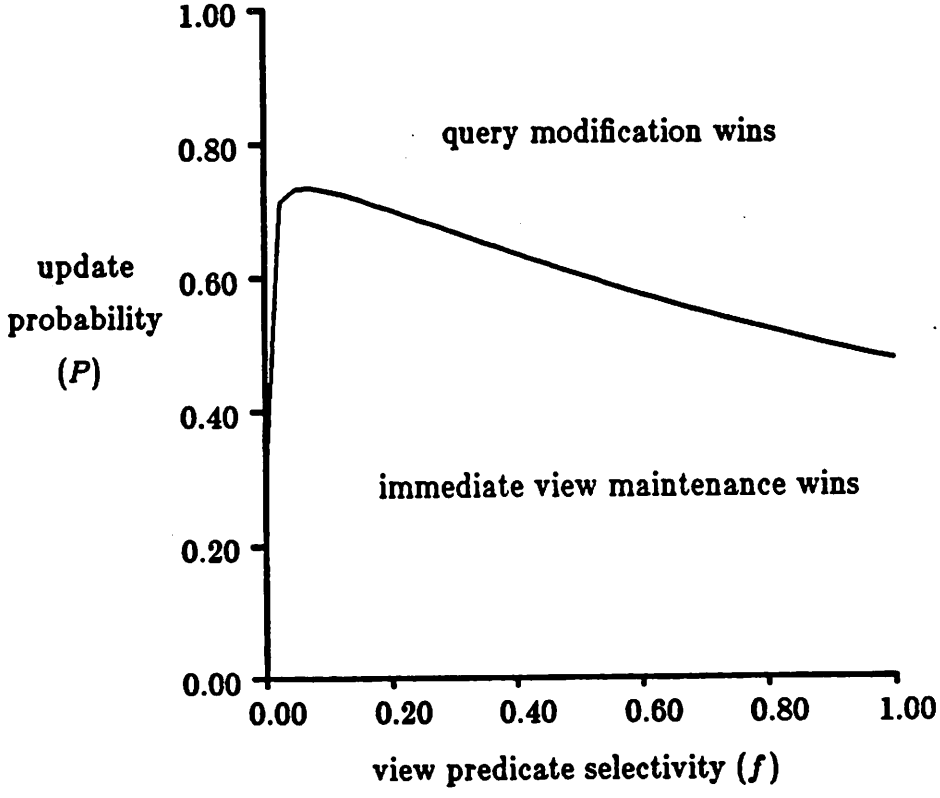
modeled by setting f=1, f_v=$1/N$ and l=1, and the results showed that query modification is superior to deferred and immediate view maintenance under these circumstances for all values of $P \geq .07$. Thus, query modification is almost always the preferred method for answering small queries against large views. Other effects of varying f_v are shown using two figures. Figure 4.9 plots the areas where deferred view maintenance, immediate view maintenance and query modification using nested loops each have best performance for different values of P and f, with f_v set to .1 (recall that the nested loop join uses an index on the inner relation). Figure 4.10



Model 2 (2-way join view): Regions where each algorithm performs best for f versus P (fraction of view retrieved (f_v)=.1).

Figure 4.9. Model 2: Algorithm Comparison

shows the same information with f_v set to .01. The view materialization methods perform better than query modification over a much wider area in Model 2 than in Model 1. In particular, view materialization is superior to query modification in Model 2 for much higher update probability. For example, for a view with selectivity f=0.5, the trade-off between query modification occurs in Figure 4.5 (Model 1) at around P=0.3, yet it happens in Figure 4.9 (Model 2) at approximately P=0.7.



Model 2 (2-way join view): Regions where each algorithm performs best for f versus P (fraction of view retrieved (f_v)=.01).

Figure 4.10. Model 2: Algorithm Comparison

In Model 2, the assumption is made that exactly half of the attributes from relations R_1 and R_2 are projected in the view. Projecting a subset of the attributes in the view gives some advantage to view materialization since when the view is queried, less total I/O is necessary than if all attributes of R_1 and R_2 are projected. However, even if all attributes of R_1 and R_2 are projected in the view, the results are similar to those described above (query modification does not perform markedly better). The reason the results do not change significantly is that the clustering of view tuples on a single page achieved by view materialization provides the primary performance advantage. Even when all attributes of R_1 and R_2 are projected in the view, the advantage of clustering dominates the extra I/O cost required for reading data from the larger materialized view.

4.3.6. Model 3: Aggregates Over Model 1 Views

Aggregates such as sum, count and average are an often-used feature of database systems. As discussed in detail in Chapter 3, many aggregates can be incrementally updated as changes occur to the data from which they are computed. Incremental maintenance of aggregates is done by defining a *state* for the aggregate, functions for updating it in case of deletion or insertion of values in the set being aggregated, and a function for finding the current value of the aggregate given the state. The notion of incrementally maintaining aggregates is extremely attractive since the aggregate state can be read quickly because it normally requires less than one disk block of storage, while it often takes a large amount of I/O to recompute the aggregate from scratch. Thus, it would appear that an aggregate need not be used often to justify the expense of maintaining a materialized version of it.

To compare the performance of maintaining aggregates versus computing them from scratch, Model 3 is is analyzed. In this model, the tuples for which the aggregate is computed do not need to be kept in a separate materialized view. Only the aggregate state must be stored.

For this model, a query to the view consists of simply reading the state of the aggregate. Using the deferred view maintenance scheme in Model 3, the cost $\text{TOTAL}_{\text{deferred3}}$ per query to the view is

$\text{TOTAL}_{\text{deferred3}} =$
(cost to read hypothetical database)
+ (cost to read the aggregate state)
+ (cost per query to update the aggregate state if necessary)
+ (cost per query of screening tuples to see if aggregate is affected)

The cost to read the hypothetical database is C_{ADread}, unchanged from Model 1. The cost to query the aggregate is the cost to read a single page, i.e.,

$$C_{\text{query3}} = C_2$$

The cost to update the aggregate is the cost of one write times the probability that at least one tuple modified since the last query to the view lies in the set being aggregated (no read is necessary since the aggregate must be read to answer the query). There are $2u$ modified tuples in the hypothetical database per query to the view, and each has probability f of lying in the aggregated set. The probability that at least one of these tuples will lie in the aggregated set is equal to 1 minus the probability that none of the tuples lie in the set. Thus, the probability that at least one of the tuples lies in the set is $(1-(1-f)^{2u})$. This yields the following expression for the cost per query to update the view:

$$C_{\text{def-refresh3}} = C_2(1-(1-f)^{2u})$$

The final value of $\text{TOTAL}_{\text{deferred3}}$ is the following:

$$\text{TOTAL}_{\text{deferred3}} = C_{\text{ADread}}+C_{\text{query3}}+C_{\text{def-refresh3}}+C_{\text{screen}}$$

Using the immediate view update algorithm, the cost per query to maintain the aggregate is

$\text{TOTAL}_{\text{immediate3}} =$
(cost to read the aggregate state)
+ (cost per query to update the aggregate state if necessary)
+ (cost per query of screening tuples to see if aggregate is affected)

The cost to read the aggregate state is C_{query3}. The cost *per transaction* to update the aggregate

state is C_2 times the probability that at least one tuple modified by the transaction matches the qualification of the aggregate. This probability is $(1-(1-f)^{2l})$. The cost per query to update the aggregate state is thus as follows:

$$C_{\text{imm-refresh3}} = \frac{C_2 k}{q}(1-(1-f)^{2l})$$

The cost of screening tuples is again C_{screen}, yielding the following expression for $\text{TOTAL}_{\text{immediate3}}$:

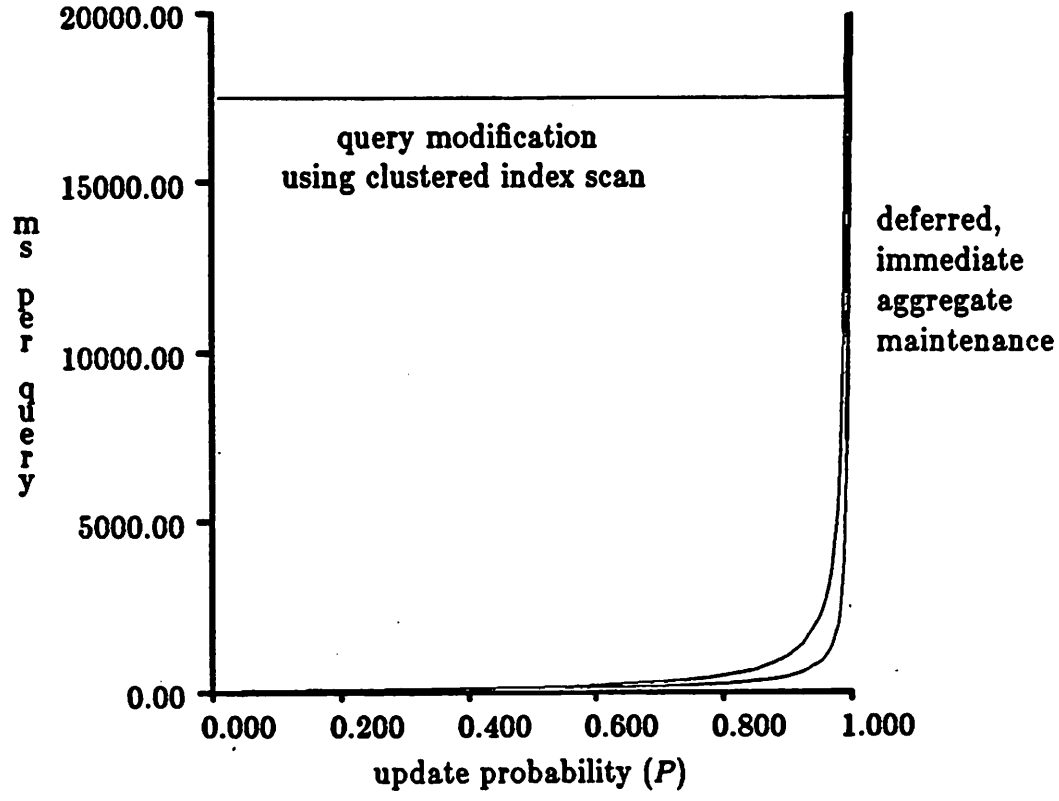
$$\text{TOTAL}_{\text{immediate3}} = C_{\text{query3}}+C_{\text{imm-refresh3}}+C_{\text{screen}}$$

The actual cost of recomputing the aggregate for each query using a clustered index scan is the same as the cost of query modification in Model 1, which is $\text{TOTAL}_{\text{clustered}}$. This cost will be compared to $\text{TOTAL}_{\text{immediate3}}$ and $\text{TOTAL}_{\text{deferred3}}$.

4.3.7. Performance Results for Model 3

To compare the total cost of using deferred view maintenance, immediate view maintenance, and a clustered index scan to compute an aggregate, the total cost of all three is plotted versus P in Figure 4.11. In this figure, f=.1 and f_v=1, so the fraction of the data being aggregated is 0.1. Even when the update probability is very high, it still pays to maintain a large aggregate like this one. For example, when P=.95 the cost per query when keeping the aggregate materialized is only about 5% of that to compute the aggregate completely. For small update probabilities, the difference is even more dramatic. When P=.20, the average cost per query is only slighly more than C_2 (30 ms) if the aggregate is materialized. Approximately 17 seconds are needed to completely recompute the aggregate.

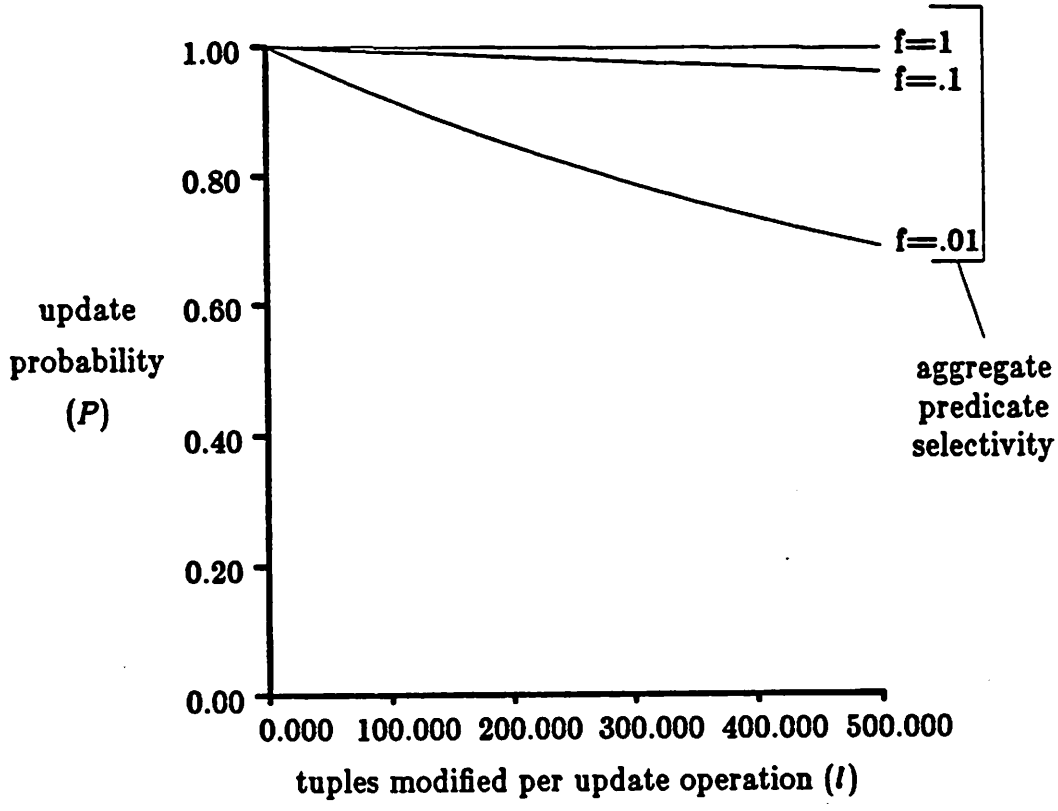
Figure 4.12 shows how the costs of aggregate materialization and standard aggregate processing depend on update probability and the size of update transactions. The curves in the figure indicate where materialization and standard processing using a clustered index scan have equal cost. Each curve represents a different value of the selectivity of the qualification of the aggregate



Model 3 (single-relation aggregate): Average cost of an aggregate query versus P for deferred and immediate view maintenance, and standard processing using a clustered index scan.

Figure 4.11. Model 3: Query Cost

(f). Query modification using a clustered index scan performs best above each curve, and immediate maintenance performs best below. It is interesting to note that maintaining materialized aggregates is most attractive when f is largest. Aggregates over large amounts of data are thus the best candidates for materialization. Since the number of tuples updated per transaction (l) will usually be small, it is often worthwhile to maintain materialized aggregates even if their conditions match a small number of tuples (i.e., f is small). Cost savings can be obtained by materializing aggregates in significantly more cases than for other views.



Model 3 (single-relation aggregate): Equivalent cost curves for immediate view maintenance, and standard aggregate processing using a clustered index scan. Above curves standard processing is best; immediate maintenance wins below.

Figure 4.12. Model 3: Algorithm Comparison

4.4. Discussion

The performance analysis presented has shown that the choice of the most efficient view materialization algorithm is highly application-dependent. The results are most sensitive to the following parameters:

1. the total fraction of operations that are updates (P).
2. the selectivity factor of the view predicate (f).
3. the fraction of the view retrieved by each query (f_v).
4. the number of tuples written by each update (l).
5. the cost of maintaining the sets of inserted and deleted tuples (either in main memory, or in disk-based hypothetical relations).

Situations where P is high, f is high, or f_v is small, tend to favor not materializing the view at all. Rather, it is best to perform query modification, and retrieve the result from the base relations using a good access plan selected by the query optimizer. An important example of this type of situation is for large views (e.g. the ED view on EMP and DEPT) and queries that always retrieve a single record. When this example was modeled using f=1, l=1, and f_v=1/(number of tuples in the view), it was found that query modification nearly always outperforms materializing the view in advance.

If f_v is large, and P is not extremely high, then it becomes desirable to maintain views in materialized form. Higher values of P, and l favor deferred view maintenance over the immediate scheme because large sets of tuples will be accumulated A and D sets of the hypothetical relation before each query. The deferred view maintenance strategy will thus perform a few large updates to the view rather than many small ones. This reduces the total number of I/Os required to maintain the view due to the nature of the Yao function [Yao77]. Conversely, if P is low, immediate view maintenance has a slight advantage over deferred maintenance.

A phenomenon observed throughout this study was that deferred and immediate view maintenance had nearly equal cost, especially when the update probability (P) was low. Because the two algorithms have costs that are so close, it is clear that by far the most important issue regarding view materialization strategies is whether to use some differential view maintenance algorithm or use query modification. The actual view maintenance algorithm chosen is of secondary importance.

There are a few reasons why deferred and immediate view maintenance were so close in performance. One reason is that for small values of P, view maintenance overhead is always small relative to the cost of queries, regardless of the view maintenance algorithm used. Deferred and immediate view maintenance were close for higher values of P as well because the advantages and disadvantages disadvantages of the two methods nearly canceled each other. The main advantage of deferred view maintenance is that fewer disk writes to the stored copy of a view must be performed than in immediate view maintenance. The reason for this is that *triangle inequality* holds for the Yao function, which is a main determinant of the number of writes to the view. More precisely,

$$y(n,m,a+b) \leq y(n,m,a)+y(n,m,b).$$

for all a, $b>0$. On the other hand, the advantage of immediate view maintenance is that less overhead is usually required to maintain the A and D sets, since they usually will not have to be written to disk (they should fit in the buffer pool except for transactions that update a large fraction of the database). In deferred view maintenance, the A and D sets must be written to disk, since they may live for more than one transaction. Reducing or increasing the overhead of maintaining A and D in either algorithm could give that algorithm a slight overall performance advantage.

Even though the expected cost per query in deferred and immediate view maintenance is nearly equal, deferred view maintenance may be preferred for other reasons. The first is that deferred maintenance seems seems more fair than immediate maintenance because the deferred method makes the transactions that actually use a view (the queries) pay most of the cost of maintaining it. Update transactions pay only the overhead to maintain the differential file data structure. Also, if there is idle CPU and disk time available, it is likely to be useful to put it to work refreshing views asynchronously. This can be done in deferred view maintenance, but not in the immediate scheme. Making use of idle CPU and disk time would improve the response time of view queries in some situations since the views would not have to be refreshed prior to a query,

yet update transactions would still pay little overhead to maintain the view. The evaluation of the usefulness of this optimization is an interesting topic for future study.

Deferred view maintenance may also be preferred in some situations due to the nature of the database architecture. The ADMS± system [RoK86] is based on an architecture where a single mainframe computer keeps an up-to-date copy of the database, and a collection of workstations is connected to the mainframe via a network. All updates submitted from workstations are sent to the mainframe for processing. Users at workstations may define a collection of database views that can be kept materialized using a deferred view maintenance algorithm. A view query submitted from a workstation is processed by first sending a message to the mainframe to see whether the base relations on which the view depends have been updated. If they have, the changes are sent back to the workstation, which updates the view, and then evaluates the query locally. If the user at the workstation is willing to tolerate answers that may be out of date, queries can be processed locally without sending any messages to the mainframe. If immediate view maintenance were used instead of deferred maintenance in ADMS±, the mainframe would have to broadcast changes continually to the workstations, incurring a high overhead for communication. Another reason that deferred maintenance is advantageous in ADMS± is that the system makes use of a write ahead log recovery scheme for transaction processing, and the net changes to base relations that occur after some time T can be easily extracted from the log. This means that ADMS± does not have to use a hypothetical relation data structure to maintain relations, which is a significant cost savings. Hence, in an architecture like ADMS±, deferred view maintenance is clearly preferred over immediate view maintenance.

This analysis has shown that the performance benefits of differential view update algorithms relative to query modification are greater for two-way join views (Model 2) than for simple restrictions (Model 1). This performance pattern is due to the natural clustering of view tuples on a single disk page that occurs when the view is materialized in advance. The performance benefits

of view maintenance algorithms should be even greater for views joining three or more relations.

View maintenance algorithms would prove particularly useful in situations where

1. the update probability is low, and
2. views are complex.

These conditions are met in many statistical and scientific database applications. Also, some databases become almost read-only as they age. For example, consider the situation faced by an aircraft manufacturer when assembling a new plane. A complete description of the plane (e.g. where each part is installed, who made each part etc.) must be maintained for purposes of future maintenance, legal documentation etc. This description is updated frequently while the plane is being built. However, when the construction job is finished, the database will seldom (if ever) be updated again. Materializing views on this database would probably not be appropriate while the plane was being built. However, it could be worthwhile after the plane was finished.

One could speculate that the most significant applications of differential view update may not be related to processing queries against views, since this study has shown that query modification is still quite effective. Rather, view materialization might have a greater impact in applications where a complete copy of the answer to a query is always needed. For example, materialization could support conditions for complex triggers and alerters, as described in [BuC79]. As another example, it could be used as a basis for a "window on a database" facility, where the result of a query would be displayed and updated in real time.

Finally, the performance of different view materialization schemes depends significantly on the database and view structure, and the distribution of queries and updates. Thus, an interesting topic for future research would be to devise an adaptive method to choose the appropriate view materialization algorithm. Future implementation and empirical testing of view maintenance algorithms is also needed to help gain a fuller understanding of the tradeoffs involved.

CHAPTER 5

PERFORMANCE OF PROCEDURE MATERIALIZATION METHODS

The same types of algorithms that can be used for view processing can also be applied to database procedures. As with views, different methods have different costs. This chapter presents a performance analysis comparing different algorithms for processing queries that retrieve the value of a database procedure. The chapter is organized as follows. Section 5.1 reviews the different algorithms that can be used for procedure maintenance. Section 5.2 describes the two procedure models (model 1 and model 2) that will be analyzed. Section 5.3 analyzes the cost of procedure maintenance using model 1. Section 5.4 presents the performance results obtained for model 1. Section 5.5 analyzes the cost of maintaining model 2 procedures. Section 5.6 gives the performance results for model 2. Finally, section 5.7 summarizes and presents conclusions.

5.1. Procedure Maintenance Algorithms

As described in chapter 3, the following algorithms can be used for processing database procedure queries:

Always Recompute:
Compute the value of the procedure from the base relations on each access (this strategy is equivalent to a special case of query modification in which the entire view is retrieved).

Cache and Invalidate:
When the procedure is accessed, if a valid result for it is cached, use it. Otherwise, recompute the value and refresh the cache. If an update command occurs that would change the value of the procedure result, the currently cached result is marked invalid. (This method is also known as simply "caching").

Update Cache:
Maintain a materialized answer to each query in the procedure definition by using a differential view maintenance algorithm. Process procedure queries by returning the stored value.

As discussed in chapter 3, view maintenance algorithms can be divided into two classes, shared and non-shared, depending on whether shared subexpressions elimination is used. In this chapter, both a shared and non-shared algorithm are analyzed. A non-shared, static version of AVM is compared with Rete view maintenance, which is shared and static. Any mention of AVM in this chapter refers to the non-shared, static version of the algorithm.

The performance model for procedures presented in this chapter is different than the one given in Chapter 4 for views because there are differences between view and procedure use. When a view is queried, usually only a small fraction of it is retrieved. In contrast, the entire value of a procedure is retrieved when it is accessed. The number of procedures in a database is likely to be much larger than the number of views. Views also usually contain a large amount of data, while procedure values are typically small. The new performance model for procedures is discussed below.

5.2. Procedure Models Analyzed

Two different models for the structure of procedures will be analyzed. In both models 1 and 2, it is assumed that each stored procedure consists of a single **retrieve** query. In model 1, procedures may be of two types. The first type (P_1) is a simple selection of one relation, R_1. The second type (P_2) is a join query. Procedures of type P_1 have the following structure:

P_1:

retrieve (R_1.all)
where $C_f(R_1)$

Type P_2 procedures have the form:

P_2, Model 1 (2-way join):

retrieve (R_1.fields, R_2.fields)
where R_1.a = R_2.b
and $C_f(R_1)$
and $C_{f_2}(R_2)$

The difference between model 1 and model 2 is that in model 2, type P_2 procedures are three-way joins instead of two-way joins. Type P_2 procedures have this structure in model 2:

P_2, Model 2 (3-way join):

retrieve (R_1.fields, R_2.fields, R_3.fields)
where R_1.a = R_2.b
and R_2.c = R_3.d
and $C_f(R_1)$
and $C_{f_2}(R_3)$

The width of tuples in both P_1 and P_2 procedures is S bytes. The selectivity of the clauses of the form $C_X(R_i)$ is X (e.g. the selectivity of $C_f(R_1)$ is f). For type P_2 procedures the expected number of tuples the procedure will contain is determined as follows. Let f^* be the product of the selectivities of the simple restriction terms C_f and C_{f_2} ($f^* = f\ f_2$). It is assumed that the expected number of tuples in a procedure of type P_2 is

$$f^* \max(|R_1|, |R_2|, |R_3|)$$
$$= f^* \max(N, f_{R_2}N, f_{R_3}N)$$
$$= f^* N$$

The database contains N_1 procedures of type P_1, and N_2 of type P_2. Using a shared view maintenance algorithm there is a possibility of sharing subexpressions in this model. Procedures of type P_1 can form a shared subexpression for procedures of type P_2 if the selection term $C_f(R_1)$ is the same. The models contain a parameter SF which is the *sharing factor*. It is assumed that a fraction SF of the type P_2 procedures are able to use a type P_1 procedure as a shared subexpression. If SF is 0, then no sharing takes place, and if SF is 1, every type P_2 procedure has a shared subexpression.

In the models, k update operations and q procedure accesses occur. Each update modifies l tuples of R_1 in place. Relations R_2 and R_3 are not modified. Each procedure access reads the entire contents of a *single* stored procedure, which is selected at random from the total collection of N_1+N_2 procedures.

Using Cache and Invalidate, when an update causes a stored procedure value to become invalid, this fact must be recorded. The most obvious way to do this is to read the first page of the object, set a flag on it that says the object is invalid, and write it back. Reading and writing the page requires an amount of time equal to $2C_2$ (60 ms) per invalidation. An alternative is to use a data structure kept in high-speed memory with an entry for each procedure indicating whether or not it is valid. One way to make this data structure recoverable is to use a reliable battery power supply for the portion of memory containing it. Another is to log the identifiers of invalidated procedures in a conventional write-ahead recovery log [Gra78]. If the data structure is checkpointed periodically, it can be recovered by playing the latest part of the log against the last checkpoint after a crash. Using either of these methods, the cost per invalidation is much less than $2C_2$ (using battery-backed-up memory, it is essentially zero compared to the cost of reading and writing a page). To measure the significance of the cost of an invalidation, a parameter for it called C_{inval} is included in the models.

A summary of the parameters used in the procedure cost model is shown below. Parameters that are unchanged from the performance model for views presented in Chapter 4 are not listed.

parameter	*meaning*
N_1	number of P_1-type procedures
N_2	number of P_2-type procedures
SF	sharing factor (fraction of P_2 procedures that have a P_1 procedure as a shared subexpression)
f_{R_3}	size of R_3 as a fraction of N
f_2	selectivity factor of predicate term C_{f_2}
C_{inval}	cost to record invalidation of a cached procedure value

The default values of the parameters for the procedure cost analysis are

N	100,000	f	.001
S	100	f_2	.1
B	4,000	f_{R_3}	.1
k	100	f_{R_2}	.1
l	25	C_1	1
q	100	C_2	30
d	20	C_{inval}	0
SF	.5		

The parameters will have the values shown unless stated otherwise. The default for f (.001) is smaller than for views because procedures typically contain a small number of tuples. Using this value of f, type P_1 procedures contain fN=100 tuples. Type P_2 procedures contain f^2N=10 tuples for the default parameters.

The relations involved have the following access methods:

relation	*access method*
R_1	B-tree primary index on field used by selection predicate $C_f(R_1)$
R_2	hashed primary index on attribute a
R_3	hashed primary index on attribute c

5.3. Cost Analysis for Model 1 Procedures

5.3.1. Model 1: Cost of Always Recompute Strategy

The expected cost to compute a procedure value is

the fraction of procedures that are of type P_1, times
the cost to compute a procedure of type P_1 ($C_{queryP1}$)

+

the fraction of procedures that are of type P_2, times
the cost to compute a procedure of type P_2 ($C_{queryP2}$).

$C_{queryP1}$ is the cost to search a B-tree index and read fN tuples from R_1. The height of the B-tree index on R_1 is H_i, as defined in chapter 4. Each of the fN tuples read must be tested against the procedure predicate at a cost of C_1 each. The number of pages read from disk at cost C_2 each is is $\lceil f \cdot b \rceil$. The complete expression for $C_{queryP1}$ is

$$C_{queryP1} = C_1 fN + C_2 \lceil f \cdot b \rceil + C_2 H_i$$

$C_{queryP2}$ is the cost to do a two-way join to retrieve the tuples of a procedure of type P_2. It is assumed that the value of this procedure is found using a B-tree index scan on R_1 and joining qualifying R_1 tuples with R_2 using the hash index on R_2. The number of pages of R_2 that must be read to do the join is

$$Y_1 = y(f_{R_2}N, f_{R_2}b, fN)$$

The total cost is

$$C_{queryP2} = C_1 fN + C_2 \lceil f \cdot b \rceil + C_2 H_i + C_1 fN + C_2 Y_1$$

The expected cost to find the value of a single procedure is

$$C_{ProcessQuery} = \left\lceil \frac{N_1}{N_1+N_2} \right\rceil C_{queryP1} + \left\lceil \frac{N_2}{N_1+N_2} \right\rceil C_{queryP2}$$

The cost of a procedure access when the procedure must be computed from scratch each time is simply

$$TOT_{Recompute1} = C_{ProcessQuery}$$

5.3.2. Model 1: Cost of Cache and Invalidate

The expected cost of accessing the result of a stored procedure using Cache and Invalidate has three components:

1. the probability that a stored procedure value is invalid (IP) times the cost to compute the value and store it (T_1)
2. the probability that the stored value is valid (1−IP) times the cost to read the stored value (T_2)
3. the cost of marking the procedure invalid if necessary (T_3)

These components give the following formula for the expected cost per read of a stored procedure value when using caching:

$$TOT_{CacheInval1} = IP\ T_1 + (1-IP)\ T_2 + T_3$$

The expected cost to compute the procedure value is $C_{ProcessQuery}$. After the values of the procedures are found, the result must be written to update the cache. Type P_1 procedures have $\lceil f \cdot b \rceil$ pages, and type P_2 procedures have $\left\lceil f^* \cdot b \right\rceil$ pages. Thus, the average size of a stored procedure value is

$$ProcSize = \left\lceil \frac{N_1}{N_1+N_2} \right\rceil \lceil f \cdot b \rceil + \left\lceil \frac{N_2}{N_1+N_2} \right\rceil \left\lceil f^* \cdot b \right\rceil$$

The cost to write the procedure value, $C_{WriteCache}$, is the cost to read the pages currently in the cache, change their value, and write them back, which is

$$C_{WriteCache} = 2C_2 ProcSize$$

The complete value for T_1 is the following:

$$T_1 = C_{ProcessQuery} + C_{WriteCache}$$

T_2 is simply the cost to read the cached procedure value, i.e.

$$T_2 = C_2 \text{ProcSize}$$

The cost per update transaction of marking stored procedures invalid (T_3) is determined as follows. For a single stored procedure, the probability that any update transaction will invalidate it (P_{inval}) is one minus the probability that the procedure is not invalidated. Thus, the value of P_{inval} is

$$P_{\text{inval}} = 1-(1-f)^{2l}$$

The cost to mark a procedure value invalid is C_{inval}. Since there are N_1+N_2 total procedures, the expected cost to mark objects invalid after an update is

$$(N_1+N_2)P_{\text{inval}}C_{\text{inval}}$$

Averaging to find the total cost of invalidation per query, the complete expression for T_3 is

$$T_3 = \frac{k}{q}(N_1+N_2)P_{\text{inval}}C_{\text{inval}}$$

Finally, the probability IP that the cache will be invalidated between reads of the procedure value must be found. To account for locality of reference, it is assumed that a fraction Z of all procedures receives a fraction $1-Z$ of all references. The remaining procedures receive a fraction Z of the references. For example, if Z=0.2 then 20% of the procedures are accessed 80% of the time. The value of IP is equal to

> The probability that an access is to a heavily-accessed object $(1-Z)$
> times the probability that a heavily accessed object is invalid (Z_1)
>
> +
>
> the probability that an access is to a seldom-accessed object (Z)
> times the probability that a seldom accessed object is invalid (Z_2).

It is assumed that each update transaction has an equal probability of invalidating any procedure. Each access reads a single stored procedure. The expected number of update transactions (X) between accesses to a single heavily-accessed procedure is equal to

(1) the total number of procedure accesses between queries to an individual frequently-accessed procedure

times

(2) the number of updates per query.

To find (1) recall that the probability that a query is to a frequently accessed object is $1-Z$. If n is the total number of objects ($n=N_1+N_2$) then there are Zn total frequently-accessed objects. Thus, the probability P_F that any query is to a *particular* frequently accessed object is

$$P_F = (1-Z)\frac{1}{Zn}$$

The value of (1) is $1/P_F$. The value of (2) is k/q. The complete formula for X is

$$X = \frac{1}{P_F}\frac{k}{q} = n\frac{Z}{1-Z}\frac{k}{q}$$

Each update transaction modifies l tuples, for a total of $2l$ new and old tuple values. Each of these tuple values has a probability f of breaking a t-lock and invalidating a procedure. The complete formula for Z_1 is

$$Z_1 = 1-(1-f)^{X\cdot 2l}$$

The expression for Z_2 is similar, except that X is replaced by Y, where Y is the expected number of update transactions between queries that read a seldom-accessed procedure. The formula for Y, which can be found using an analysis similar to the one for Z, is

$$Y = n\frac{1-Z}{Z}\frac{k}{q}$$

The expression for Z_2, and the final formula for IP are shown below.

$$Z_2 = 1-(1-f)^{Y\cdot 2l}$$

$$\text{IP} = (1-Z)Z_1+Z\ Z_2$$

5.3.3. Model 1: Cost of Update Cache (Non-Shared)

The following factors contribute to the average cost of retrieving the value of a procedure maintained using AVM:

- the cost to screen updated tuples when t-locks are broken to see if they cause a procedure value to change,
- the cost to compute the sets of tuples to be inserted into and deleted from the procedure value,
- the cost to read and write the procedure value to refresh its contents,
- the overhead to maintain the sets of modified base relation tuples (A_{net} and D_{net}) in an auxiliary data structure during each update, and
- the cost to read the result of the stored procedure when it is accessed.

For screening new tuples there is an expected cost of N_1C_1fl for the N_1 procedures of type P_1 and N_2C_1fl for the N_2 procedures of type P_2.

To compute the changes to procedures of type P_1, there is no extra cost. For type P_2 procedures, a cost is incurred to join qualifying R_1 tuples with R_2. The join requires joining $2fl$ tuples from R_1 to R_2 using the hash index on the join field of R_2. R_2 has $f_{R_2}N$ tuples and $f_{R_2}b$ blocks. Thus, for a single type P_2 procedure, the following number of page reads are required:

$$Y_2 = y(f_{R_2}N, f_{R_2}b, 2fl)$$

The cost to refresh the stored copies of procedures is found in the following way. Procedure values of type P_1 contain fN tuples, and fb blocks. Each update command modifies l tuples (equivalently, l tuples are deleted and l are inserted). Thus, the expected number of pages that must be read and written from a type P_1 procedure after each update command is

$$Y_3 = y(fN, fb, 2fl)$$

The total selectivity of the condition of a type P_2 procedure is f^* so there are f^*N tuples and f^*b blocks in a procedure of type P_2. Thus, refreshing a procedure of type P_2 after a transaction that modifies l tuples requires the following expected number of block reads and writes:

$$Y_4 = y(f^*N, f^*b, 2f^*l)$$

There is also overhead to maintain the sets of new and old tuples (A_{net} and D_{net}) during each transaction. It is assumed that there is one A_{net} and D_{net} set for each procedure that has a lock broken by the update transaction. These sets are maintained in data structures created on the fly. The total size of all the A_{net} and D_{net} sets is equal to the total number of locks broken, which is $2fl(N_1+N_2)$. There is an overhead of C_3 per tuple to maintain these sets during a transaction.

The expected size in pages of a stored procedure value is ProcSize, so the average cost to read a stored procedure value is

$$C_{read} = C_2 ProcSize$$

The components of the cost of a procedure access using AVM to implement the Update Cache strategy are summarized below.

cost component	*name*	*value*
screen R_1 tuples for type P_1 procedures	$C_{screenP1}$	N_1C_1fl
screen R_1 tuples for type P_2 procedures	$C_{screenP2}$	N_2C_1fl
refresh procedures of type P_1	$C_{refreshP1}$	$N_1C_22Y_3$
refresh procedures of type P_2	$C_{refreshP2}$	$N_2C_22Y_4$
maintain A_1,D_1 sets	$C_{overhead}$	$C_32fl(N_1+N_2)$
join R_1 tuples to R_2	C_{join}	$N_2C_2Y_2$
average cost to read a procedure	C_{read}	C_2ProcSize

The cost C_{read} is paid once each time a procedure value is read. The other cost components are paid once each update operation. These components must be multiplied by k/q to find the cost per access. Hence, the average cost of a procedure access using AVM in model 1 procedures is as follows:

$$TOT_{non\text{-}shared1} = C_{read} + \frac{k}{q}(C_{screenP1}+C_{screenP2}+C_{refreshP1}+C_{refreshP2}+C_{overhead}+C_{join})$$

5.3.4. Model 1: Cost of Update Cache (Shared)

The shared view maintenance algorithm analyzed here is Rete view maintenance. The Rete network used to maintain individual procedures of type P_1 and P_2 is shown in Figure 5.1. The costs for screening tuples against the predicate term $C_f(R_1)$ of procedures of type P_1 and to refresh stored copies of those procedures is the same as for AVM. Because a fraction SF of type P_2 procedures have a shared subexpression, screening costs must only be paid for the remaining fraction 1−SF. The total cost of screening tuples against the predicate term $C_f(R_1)$ of type P_2 procedures is

$$C_{\text{ScreenP2-Rete}} = N_2(1-\text{SF})C_1 f 2l$$

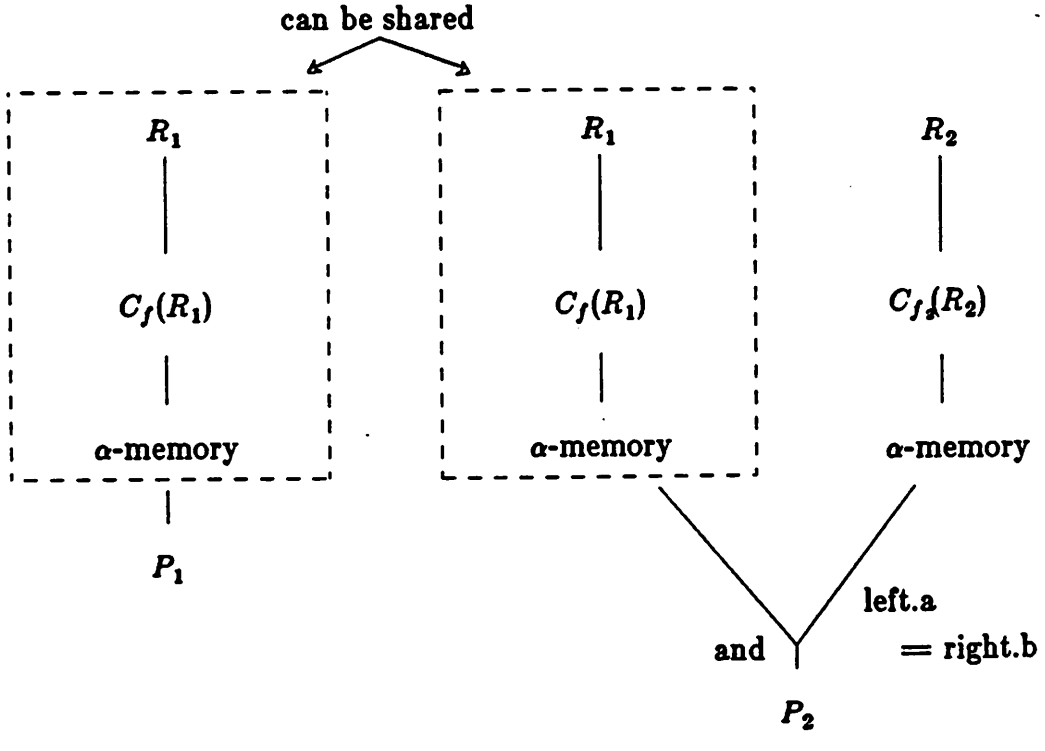


Figure 5.1. Rete networks for type P_1 and P_2 procedures in model 1

For the fraction 1−SF of type P_2 procedures that do not have a shared subexpression, the left α-memory node must be refreshed. The cost of refreshing the α-memory nodes for these procedures is

$$C_{refresh\text{-}\alpha} = N_2(1-SF)2C_2Y_3$$

For each of the tuples inserted into or deleted from the left α-memory, the right memory must be checked for joining tuples. The cost to check for joining tuples is the cost to make $2fl$ probes into the right memory, which contains $f^{**}N$ tuples, where the value of f^{**} is

$$f^{**}=f_2f_{R_2}$$

The expected number of pages that must be read from one right α-memory is

$$Y_5 = y(f^{**}N, f^{**}b, 2fl)$$

The total cost of these reads for all N_2 procedures of type P_2 is

$$C_{join\text{-}\alpha} = N_2C_2Y_5$$

The average cost of reading a procedure value when it is accessed is C_{read}. The components of the cost of accessing a procedure that is maintained using RVM are summarized in the table below.

cost component	*name*	*value*
screen R_1 tuples for P_1	$C_{screenP1}$	(unchanged)
screen R_1 tuples for P_2	$C_{ScreenP2\text{-}Rete}$	$N_1(1-SF)C_1f2l$
refresh procedures of type P_1	$C_{refreshP1}$	(unchanged)
refresh left α-memory for procedures of type P_2	$C_{refresh\text{-}\alpha}$	$N_2(1-SF)2C_2Y_3$
refresh procedures of type P_2	$C_{refreshP2}$	(unchanged)
read right α-memory	$C_{join\text{-}\alpha}$	$N_2C_2Y_5$
read procedures P_1, P_2	C_{read}	(unchanged)

C_{read} is paid once per query. The other costs shown in the table are paid once per update. The average cost per query of maintaining procedures after updates is found by multiplying these figures by the number of updates per query (k/q). The average total cost per query when main-

taining procedures using RVM is

$$TOT_{shared1} = C_{read} + \frac{k}{q}(C_{screenP1} + C_{ScreenP2\text{-}Rete} + C_{refreshP1} + C_{refresh\text{-}\alpha} + C_{refreshP2} + C_{join\text{-}\alpha})$$

5.4. Performance Results for Model 1 Procedures

In this section, the results of the performance analysis for model 1 procedures are presented and discussed. Several figures show the cost of a procedure access for various parameter values using Always Recompute, Cache and Invalidate, and both the shared and non-shared versions of Update Cache. Other figures plot the area where each algorithm performs best for the update probability P versus the object size f.

Figure 5.2 shows query cost versus update probability, assuming that the Cache and Invalidate strategy marks procedures invalid using the straightforward method that requires two disk I/Os. This situation is modeled by setting C_{inval}=60ms. Figure 5.3 plots the same curves for C_{inval}=0. Figures 5.2 and 5.3 clearly show that the total cost per query using Cache and Invalidate is highly sensitive to the value of C_{inval}. Thus, if Cache and Invalidate is implemented, it is important to keep C_{inval} small. C_{inval} can be limited using one of the techniques previously described (e.g. a data structure in battery-backed-up memory). In both figures, the cost of Cache and Invalidate and both versions of Update Cache are equal when the update probability P is zero because there is never any overhead to update or recompute procedure values. In Figure 5.3, there is a significant difference in the cost of Cache and Invalidate and Update Cache for $0<P<0.7$. This difference occurs for the following reasons.

1. For f=0.001 it is less expensive to incrementally update an object when only a few tuples change than to invalidate and recompute it.

2. Update Cache suffers from *false invalidations*, which are invalidations that are not necessary because the object does not really change.

For type P_2 procedures, the probability that an object has really been made invalid given that a

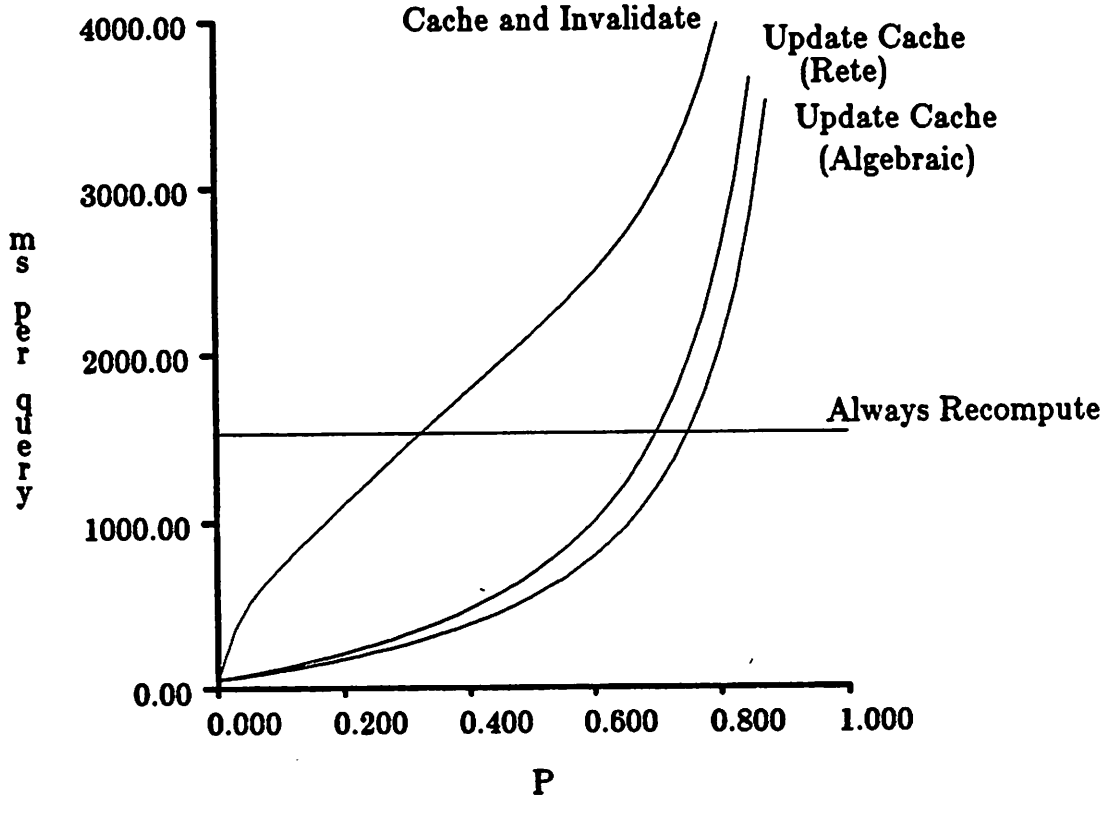


Figure 5.2. Query cost versus update probability for high cache invalidation cost (60 ms)

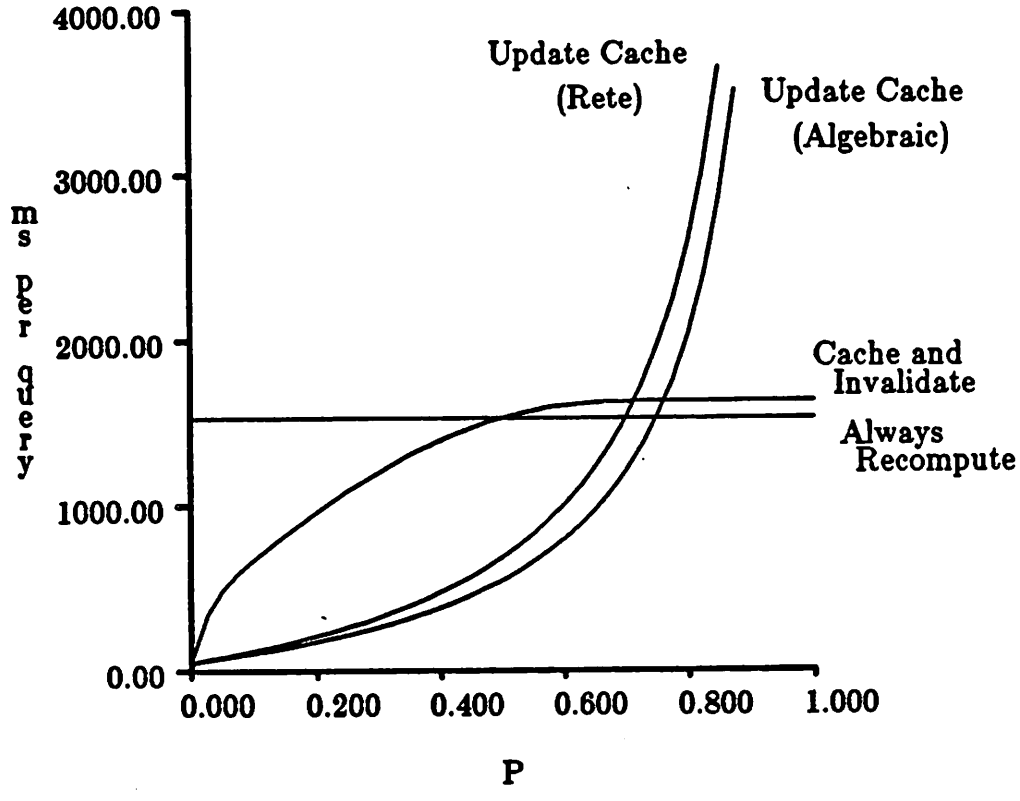


Figure 5.3. Query cost versus update probability for low cache invalidation cost (0 ms)

new tuple matches the predicate $C_f(R_1)$ is f_2 (the selectivity of the other selection term). Hence, the probability that an invalidation is false is $1-f_2$. Since the default value of f is 0.1, the probability of false invalidation is significant. For values of $P>0.6$ in Figure 5.3, the cost of Cache and Invalidate levels off at a plateau slightly above the cost of Always Recompute because stored procedure values are virtually never valid. The slight difference between the two curves represents the effort wasted by Cache and Invalidate to write back procedure values after they are computed. The cost of both Update Cache strategies rises dramaticly for large values of P because stored procedure results must be updated repeatedly between queries.

The cost per query using larger objects (f=0.01) is plotted in Figure 5.4. For this value of f, type P_1 procedures contain 1,000 records and type P_2 procedures contain 100 records. When the update probability is low, it is significantly more efficient to incrementally update a large object than to mark it invalid and require it to be recomputed. Incremental maintenance is superior in this case because only a small amount of work is required to bring an object to the correct state when only a few tuples in it change. Invalidation requires the next query to completely recompute the object, which is expensive for large objects. The cost per query for small objects (f=0.0001) is shown in Figure 5.5. For this value of f, type P_1 and P_2 procedures contain 10

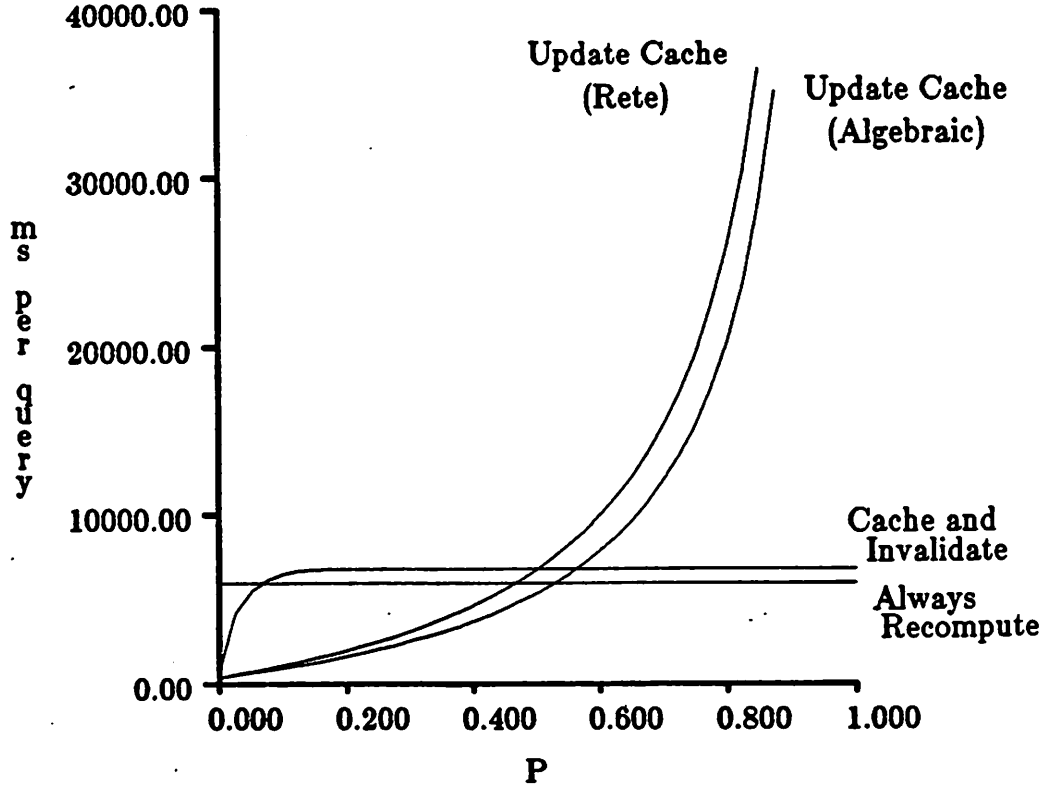


Figure 5.4. Query cost versus update probability for large objects (f=0.01)

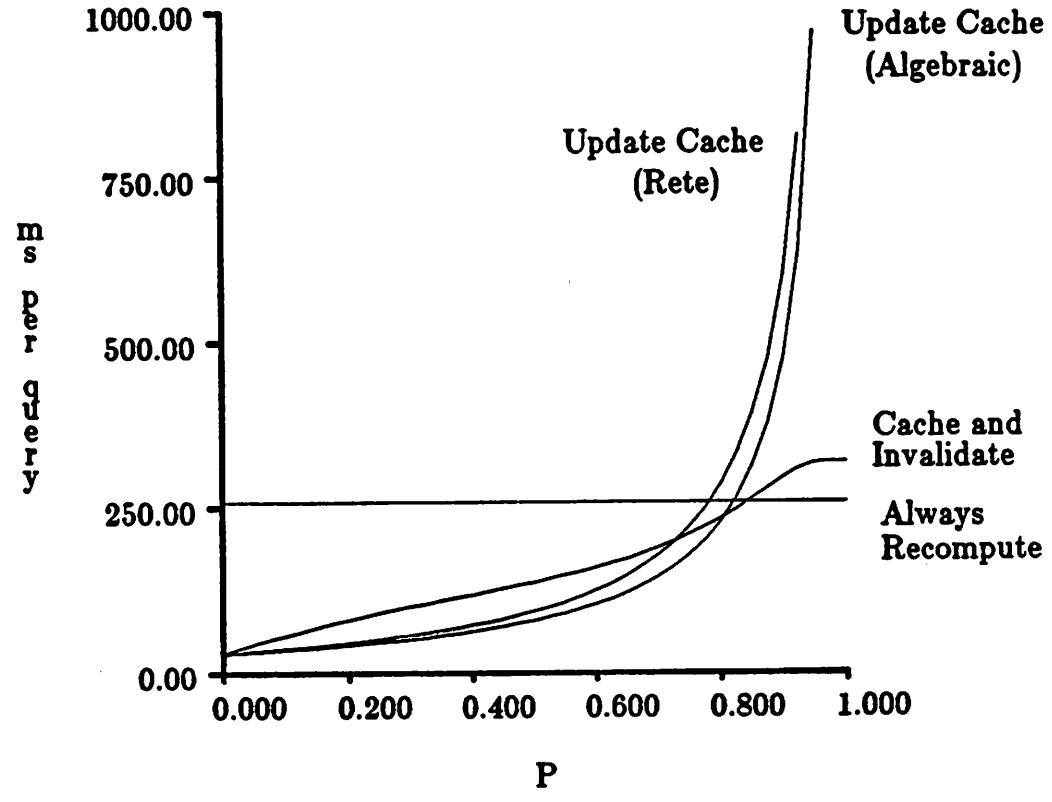


Figure 5.5. Query cost versus update probability for small objects (f=0.0001)

tuples and 1 tuple, respectively. Figure 5.5 shows that when procedures are small, Cache and Invalidate is very competitive with the Update Cache strategies. Furthermore, Cache and Invalidate does not suffer from the severe performance degradation that affects Update Cache when the update probability becomes large. The case where objects are as small as possible (one tuple) is examined in Figure 5.6. In this figure, N_1=100, N_2=0 and f=1/N, meaning that all procedures are selections of one tuple from a single relation. Cache and Invalidate is essentially equivalent to Update Cache under these conditions, except that the performance of Cache and Invalidate does not degrade severely for large P.

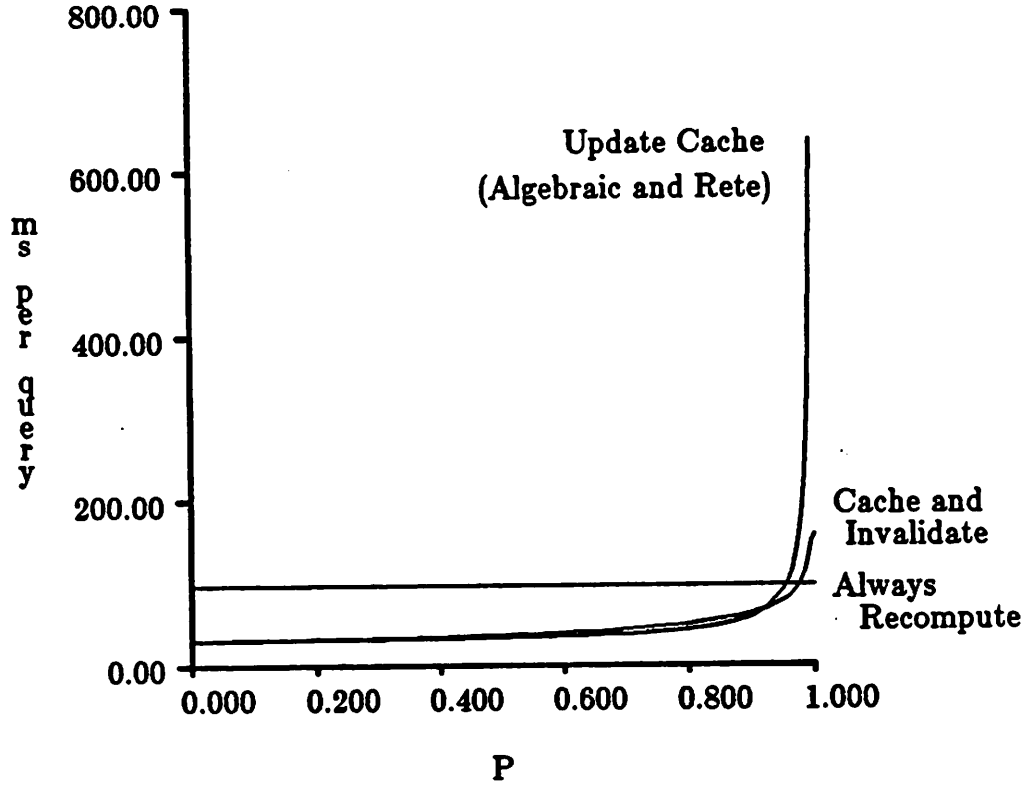


Figure 5.6. Query cost versus update probability for single-tuple objects (f=1/N)

Figure 5.7 shows the cost per query assuming that the locality of reference is high (Z=0.05). Again, Cache and Invalidate is very competitive with Update Cache for low P, and superior for large P. The affect of high locality of reference is similar to the affect of small objects.

The affect of a large number of objects is modeled in Figure 5.8 by setting N_1=N_2=1000. The cost of Cache and Invalidate and Update Cache is the same for zero update probability, but cost increases more rapidly as P increases it does in Figure 5.3. Varying the total number of objects changes the slope of the curves for the Update Cache strategies, and changes the value of P where the cost of Cache and Invalidate reaches its plateau. Figure 5.9 compares the two

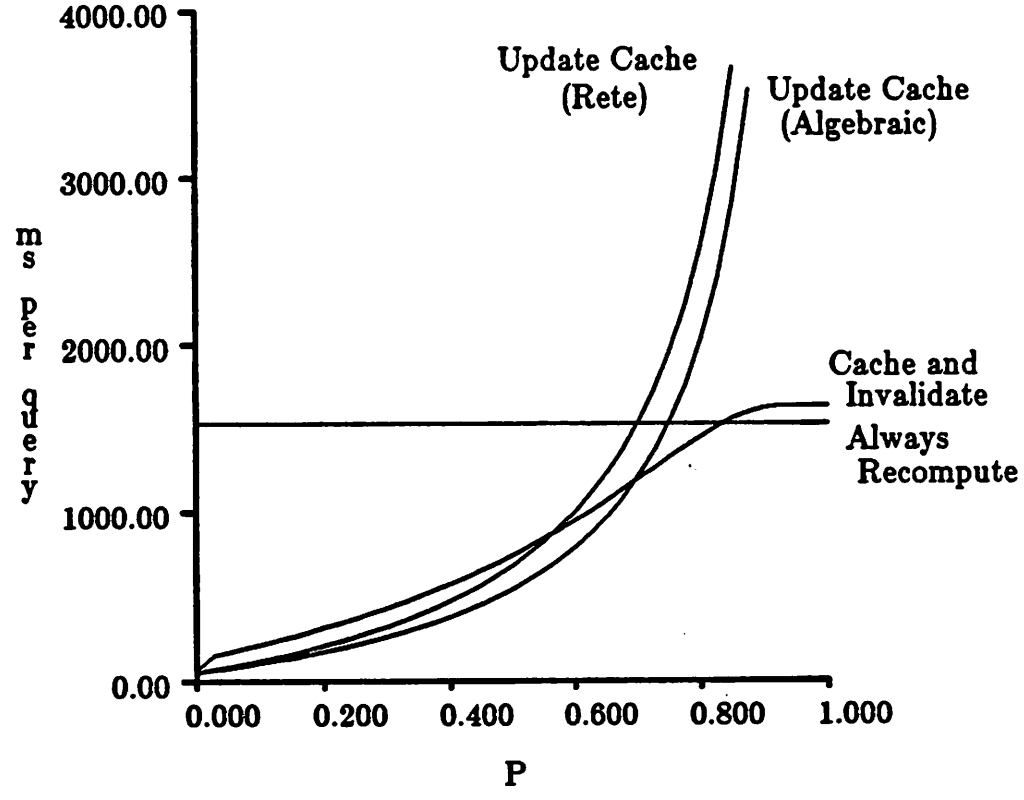


Figure 5.7. Query cost versus update probability for high locality (Z=0.05)

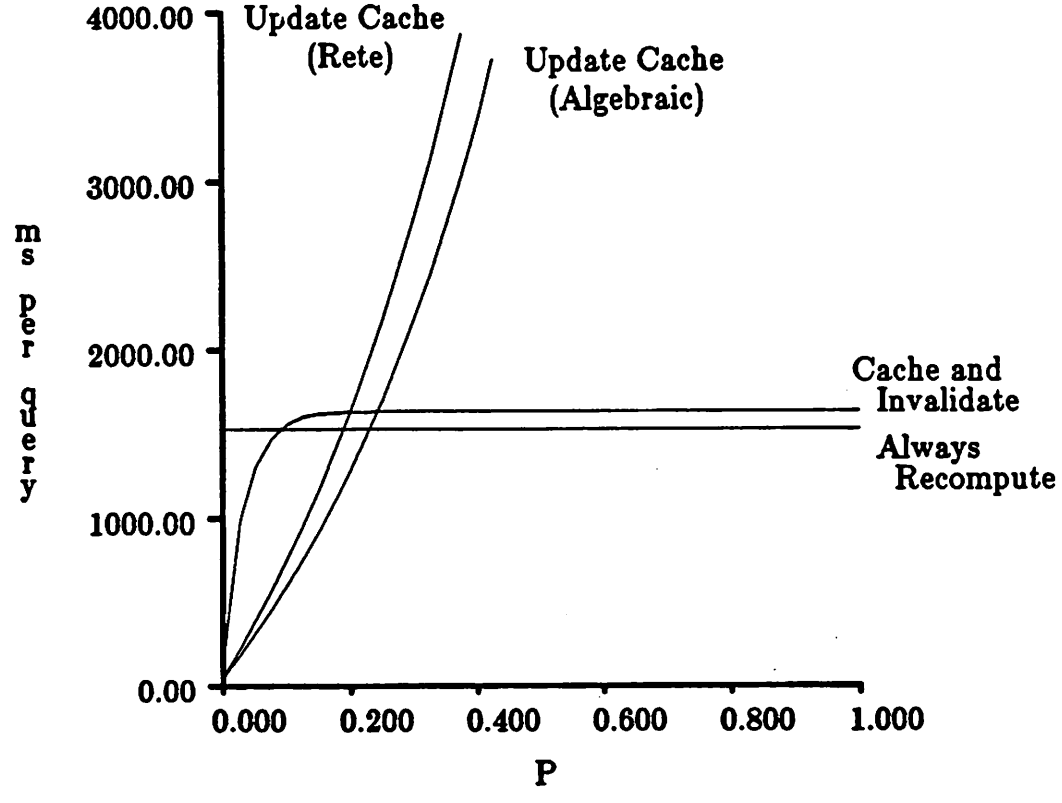


Figure 5.8. Query cost versus P for large number of objects (N_1=N_2=1000)

different Update Cache algorithms (AVM and RVM) focusing on the effect of the level of sharing (SF). In model 1, the cost of RVM becomes comparable to AVM only when almost every type P_2 procedure has a shared subexpression for its selection term on R_1. The reason RVM performs poorly compared to AVM for small sharing factors is that RVM must pay overhead to refresh copies of left α-memory nodes. When procedures contain only two-way joins (as in model 1) only a high level of sharing can make RVM competitive with AVM. Different results are obtained for the three-way join case analyzed later for model 2.

Figure 5.10 shows the regions where each algorithm performs best for different object sizes and update probabilities. The area where Cache and Invalidate wins in Figure 5.10 is

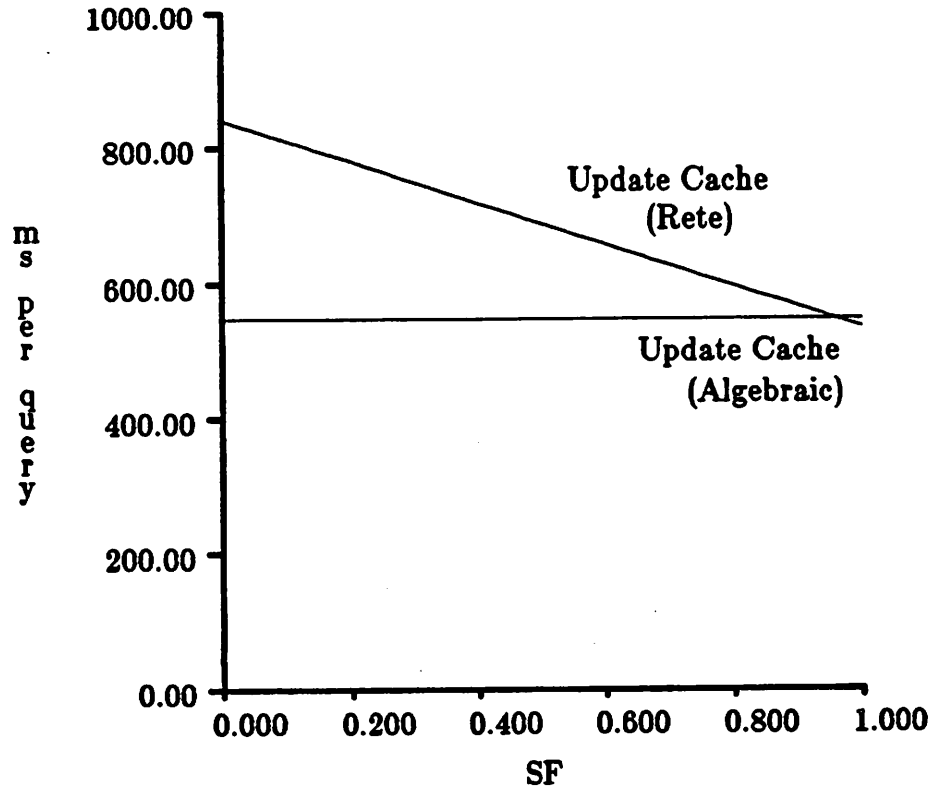


Figure 5.9. Query cost versus sharing factor (SF)

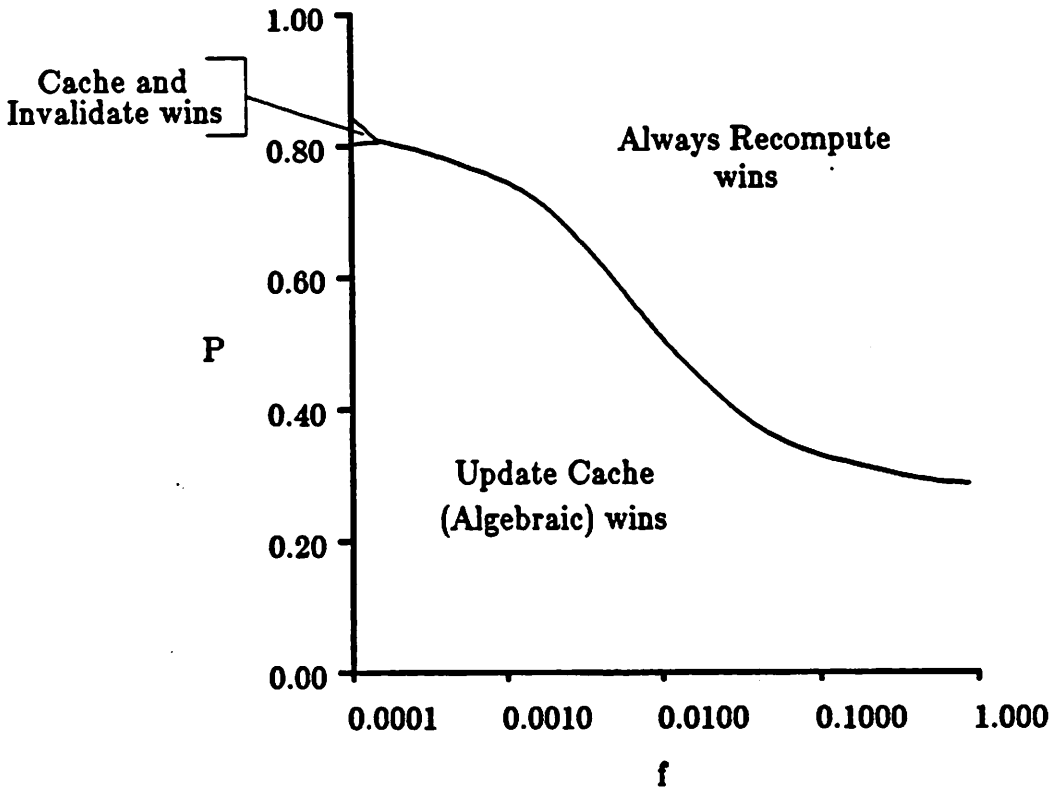


Figure 5.10. Areas where each method wins for object size versus update probability

insignificant, except that it shows that its cost is close to the cost of Update Cache in the vicinity. As expected, the methods with a per-update overhead do not do as well as Always Recompute when the update probability P is large. An interesting phenomenon observed is that Update Cache wins for a smaller range of values for P when objects are large than when they are small. This phenomenon occurs because it is highly likely that any update will affect a large object, so such an object must be maintained often. However, when objects are small, updates are likely not to affect them at all, so little overhead is incurred.

In Figure 5.11, the locality of reference is higher than in the previous figure (Z=0.05). Cache and Invalidate benefits from the increased locality but Update Cache does not. Cache and

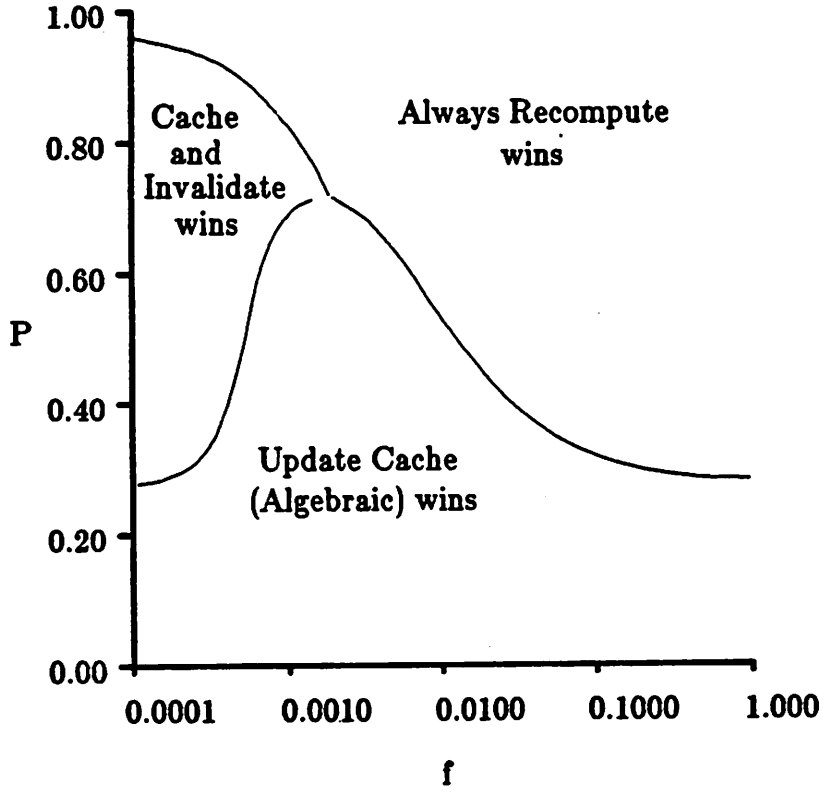


Figure 5.11. Areas where each method wins assuming high locality (Z=0.05)

Invalidate performs best when objects are small (f <0.002). The reason this occurs is that incrementally updating small objects costs nearly as much as recomputing them and writing back the results.

To demonstrate how close Update Cache and Cache and Invalidate are, Figure 5.12 shows the area where Cache and Invalidate is within a factor of two of Update Cache or better for the default parameter settings. When the update probability P is high, Cache and Invalidate is close to or superior to Update Cache because the cost of Update Cache rises rapidly as P grows. Cache and Invalidate is also close to Update Cache for small objects when the update probability is low. Figure 5.13 shows the same information with f_2=1, which reduces the probability of false

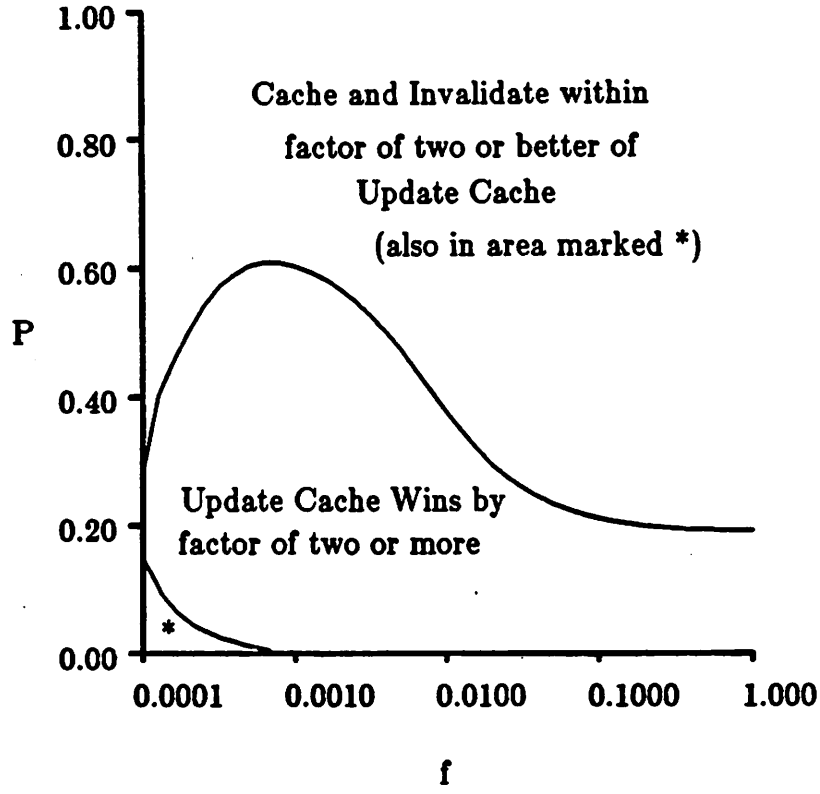


Figure 5.12. Measure of closeness between Cache and Invalidate and Update Cache

invalidation to zero. Cache and Invalidate performs even better for small objects in this situation.

5.5. Cost Analysis for Model 2 Procedures

The cost of maintaining model 2 procedures is analyzed in this section. The difference between models 1 and 2 is that type P_2 procedures required a three-way join in model 2 rather than a two-way join. Below, the cost formulas for model 2 are presented. Most of the formulas remain unchanged, so only the differences from model 1 are shown.

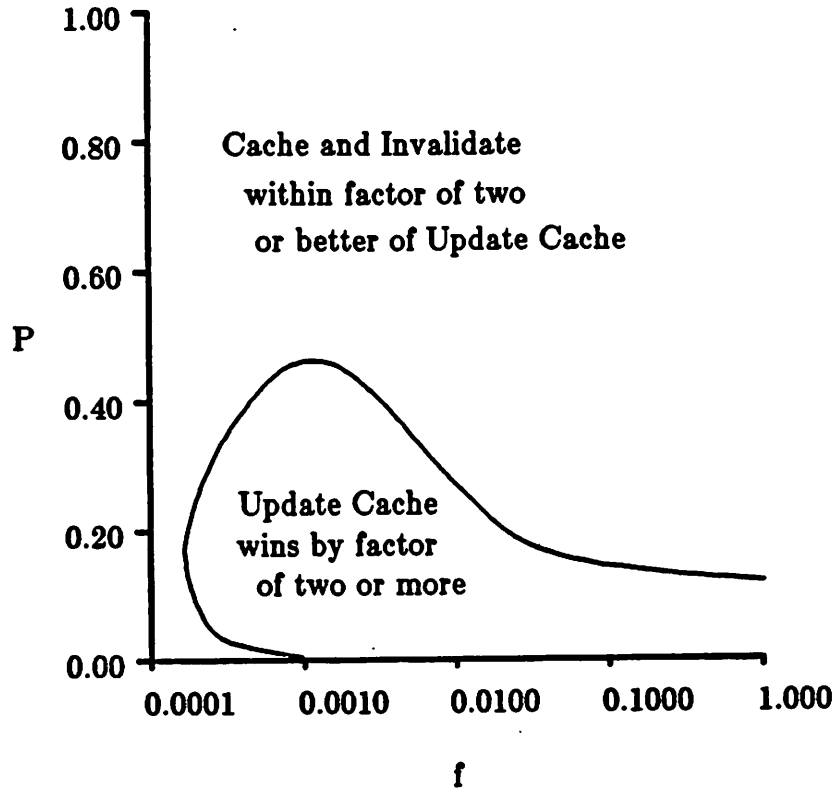


Figure 5.13. Measure of closeness (f_2=1)

5.5.1. Model 2: Cost of Always Recompute

The cost of Always Recompute is different in model 2 than model 1 because a three-way join is required to construct the value of a procedure of type P_2 instead of a two-way join. The cost to compute this three-way join is $C_{queryP2}'$. The value of a type-P_2 procedure is found by

(1) using a B-tree index scan on R_1 to find tuples matching $C_f(R_1)$,
(2) joining qualifying R_1 tuples with R_2 using the hash index on R_2, and
(3) joining the resulting tuples to R_3 using the hash index on R_3.

The cost of (1) plus (2) is the same as C_{query1}. Part (3) requires reading the following number of pages from R_3:

$$Y_6 = y(f_{R_3}N, f_{R_3}b, fN)$$

An additional fN predicate tests are required. The complete expression for $C_{queryP2}'$ is

$$C_{queryP2}' = C_{query1} + C_2 Y_6 + C_1 fN$$

The average cost of computing a procedure value from scratch in model 2 is

$$TOT_{Recompute2} = \left\lceil \frac{N_1}{N_1+N_2} \right\rceil C_{queryP1} + \left\lceil \frac{N_2}{N_1+N_2} \right\rceil C_{queryP2}'$$

5.5.2. Model 2: Cost of Cache and Invalidate

The cost formula for caching in model 2 ($TOT_{CacheInval2}$) is found simply by replacing $C_{queryP2}$ by $C_{queryP2}'$.

5.5.3. Model 2: Cost of Update Cache (Non-Shared)

In model 2 the tuples resulting from the join of R_1 and R_2 must be joined to R_3 when the non-shared algorithm (AVM) is used. The join of tuples from R_1 to R_2 requires reading Y_2 pages from R_2. The fN tuples resulting from this join are then joined to R_3. R_3 has $f_{R_3}N$ tuples and $f_{R_3}b$ blocks, so this last join requires the following number of page reads:

$$Y_7 = y(f_{R_3}N, f_{R_3}b, 2fl)$$

The total join cost (C_{join}') is

$$C_{join}' = N_2 C_2 (Y_2 + Y_7)$$

The total cost per query for AVM in model 2 is found by substituting C_{join}' for C_{join} in the formula from model 1, yielding the formula

$$TOT_{non\text{-}shared2} = C_{read} + \frac{k}{q}(C_{screenP1} + C_{screenP2} + C_{refreshP1} + C_{refreshP2} + C_{overhead} + C_{join}')$$

5.5.4. Model 2: Cost of Update Cache (Shared)

The cost components $C_{screenP1}$, $C_{ScreenP2\text{-}Rete}$, $C_{refreshP1}$ and $C_{refresh\text{-}\alpha}$ are unchanged from the analysis for model 1. In model 2, a β-memory rather than an α-memory forms the right input to the and node above a type P_2 procedure, as shown in Figure 5.14. The part of the figure in the dashed box can be a shared subexpression. A fraction SF of the type P_2 procedures share that portion of the network with a procedure of type P_1. Tuples that reach the left input of the and node must be joined to the β-memory node. The β-memory contains $f_2^{**}N$ tuples and $f_2^{**}b$ blocks, where f_2^{**} has the following value:

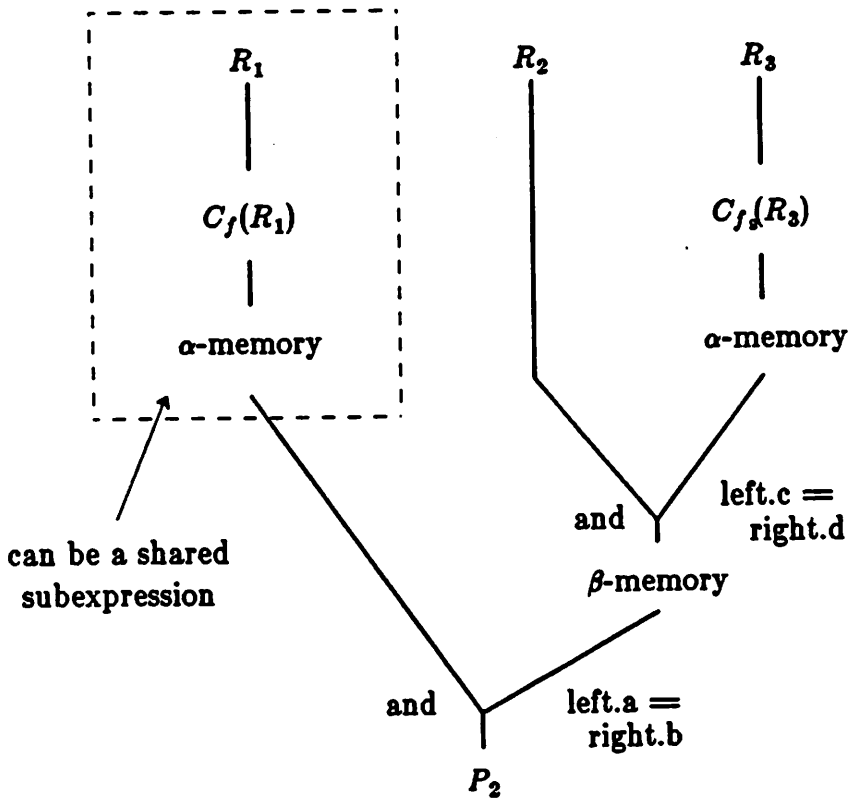


Figure 5.14. Model 2: Rete Network for P_2 Procedures

$$f_2^{**} = f_2 f_{R_3}$$

The following number of pages must be read from the β-memory node to perform the join:

$$Y_8 = y(f_2^{**}N, f_2^{**}b, 2fl)$$

The expected cost to join tuples from the left input to the β-memory after each update is

$$C_{\text{join-}\beta} = N_2 C_2 Y_8$$

$C_{\text{refreshP2}}$ is the same as for model 1 because type P_2 procedures are the same size as in model 1, and the expected number of tuples in a type P_2 procedure that change after an update transaction is still the same. The average cost to read a procedure in model 2 is also unchanged from model 1. Thus, the only difference in cost from model 1 is that $C_{\text{join-}\alpha}$ is replaced by $C_{\text{join-}\beta}$. The total cost formula for maintaining procedures using RVM in model 2 is

$$\text{TOT}_{\text{shared2}} = C_{\text{read}} + \frac{k}{q}(C_{\text{screenP1}} + C_{\text{ScreenP2-Rete}} + C_{\text{refreshP1}} + C_{\text{refresh-}\alpha} + C_{\text{refreshP2}} + C_{\text{join-}\beta})$$

5.6. Performance Results for Model 2 Procedures

The performance results for Model 1 and Model 2 are similar, as can be seen by comparing Figure 5.15 with Figure 5.3. The main difference is that the shared view maintenance algorithm (RVM) performs significantly better in model 2 than in model 1 compared to the non-shared algorithm (AVM). Figure 5.16 shows the performance of the two algorithms versus the sharing factor SF. For a sharing factor of approximately 0.47, the two algorithms are equivalent in cost. For higher sharing factors, RVM is superior to AVM. RVM has an advantage in this situation because when tuples in R_1 change, they must be joined only to the right β-memory, but AVM must join the tuples to R_2 and then join the resulting tuples to R_3. Using RVM, as the sharing factor increases, the cost of maintaining the left α-memory becomes less than the advantage provided by the precomputed subexpression in the β-memory. Figure 5.17 shows the areas where each algorithm performs best for update probability versus object size in Model 2. Figure 5.17 is similar to Figure 5.10 for Model 1, except that the best version of Update Cache is RVM instead

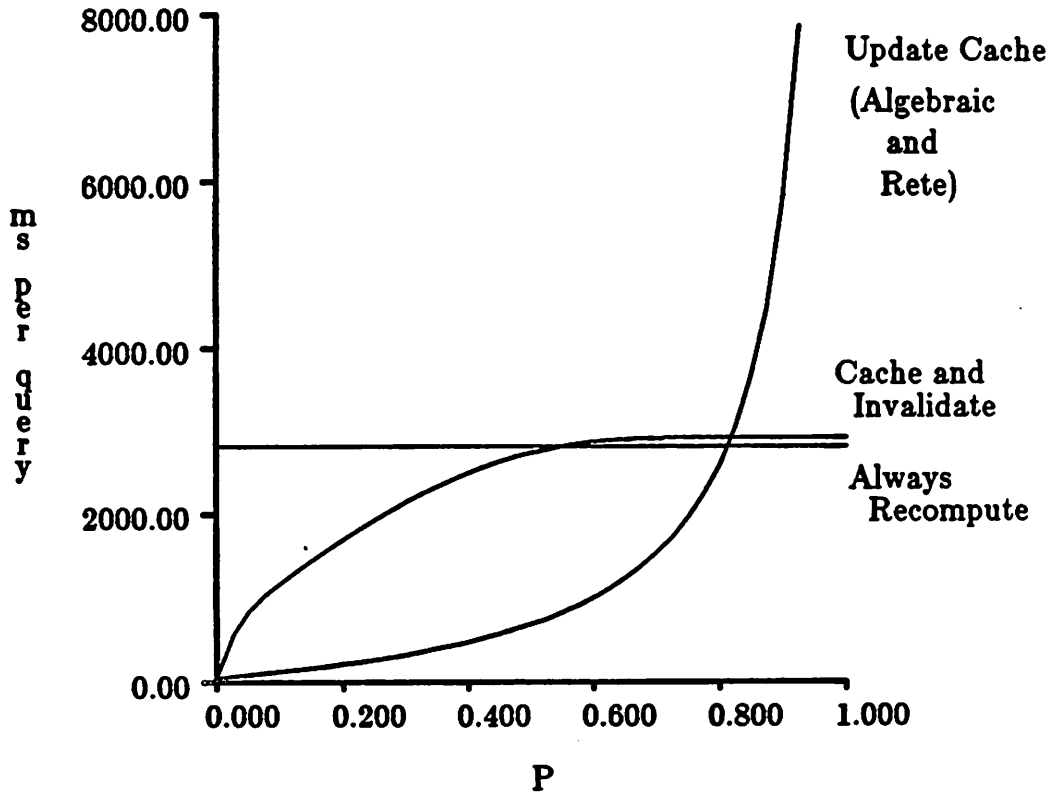


Figure 5.15. Model 2: Query cost for default parameters

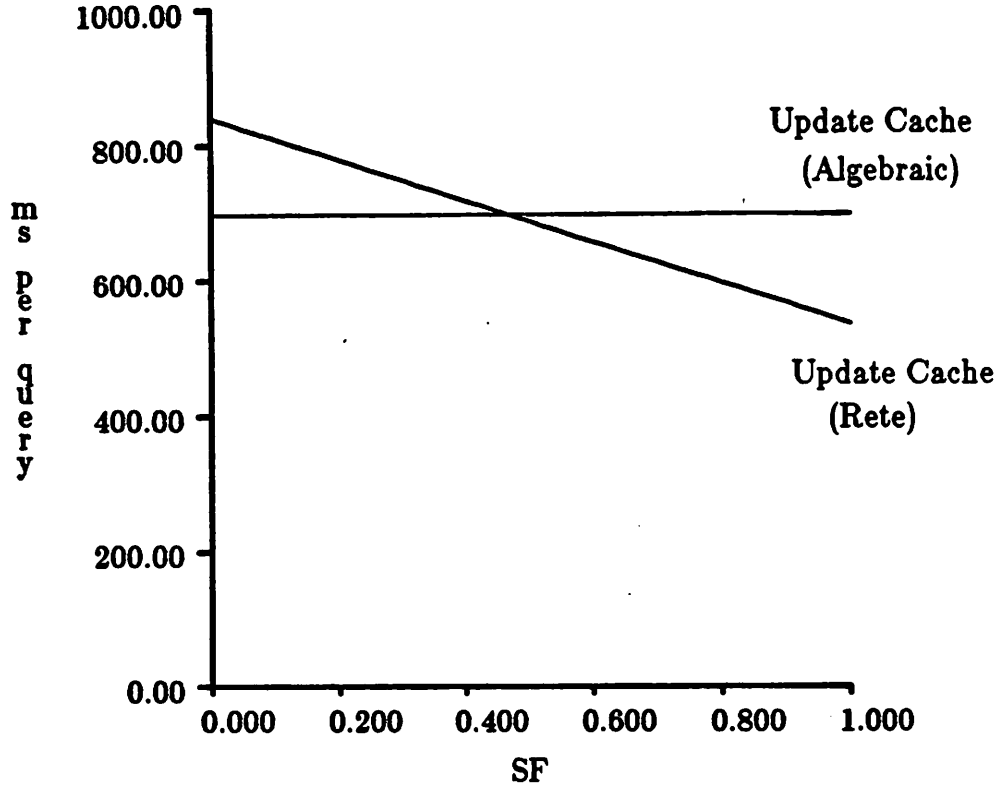


Figure 5.16. Model 2: Query cost of Update Cache alternatives versus sharing factor

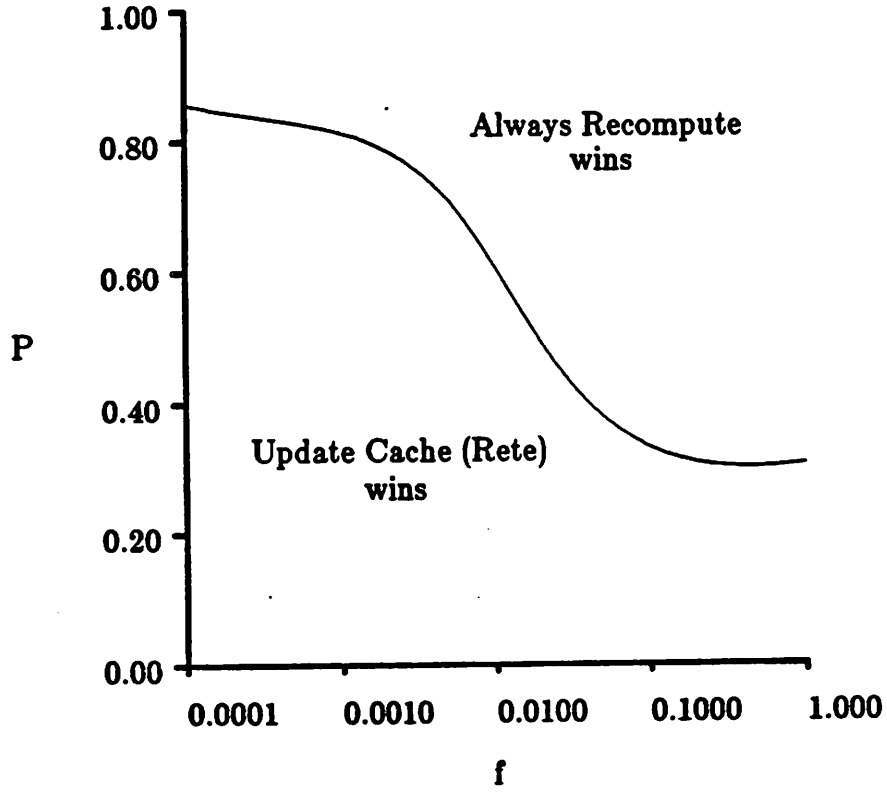


Figure 5.17. Model 2: Winners for update probability versus object size

of AVM.

5.7. Summary and Conclusions

This study has brought out several points regarding the effectiveness of Always Recompute, Cache and Invalidate, and Update Cache for processing database procedures. It is critical to use some method to limit the cost of marking a procedure invalid in Cache and Invalidate. Otherwise, its performance is significantly worse than that of Update Cache. If a low-cost invalidation method is used and procedure results are small, Cache and Invalidate is as efficient (or only slightly worse than) Update Cache. A problem with Update Cache is that its performance

degrades severely at high update probabilities. Cache and Invalidate does not suffer from this problem if the invalidation cost is small. Its performance is only slightly worse than that of Always Recompute for high update probability. This phenomenon makes Cache and Invalidate a much safer algorithm than Update Cache if there is a possibility that update frequency will be high. Both Cache and Invalidate and Update Cache bring substantial savings if the update probability is small. For example, using f=0.0001 (as shown in Figure 5.5), with P=0.1, Cache and Invalidate and Update Cache outperform Always Recompute by factors of approximately 5 and 7, respectively. Update Cache is significantly better than Cache and Invalidate for large objects when update probability is low. This occurs because it is inexpensive to incrementally update a large object when it changes relative to the cost of recomputing it entirely. Another interesting observation made in this study is that Update Cache sometimes outperforms Cache and Invalidate for both small and large objects when update probability is low. This occurs because Cache and Invalidate can suffer from false invalidations.

There are major differences in performance between Always Recompute, Cache and Invalidate, and Update Cache which depend primarily on update probability and object size. For the different versions of Update Cache, including a shared algorithm (RVM) and a non-shared algorithm (AVM), relative performance is insensitive to update probability and object size. The important parameters when comparing AVM and RVM are

(1) the likelihood of finding shared subexpressions (sharing factor),
(2) the number of joins in a procedure query, and
(3) the relative frequency of updates to different relations.

Increasing the sharing factor makes RVM perform better, but does not affect the performance of AVM. In the analysis of this chapter, when procedures contain only two-way joins (as in model 1) AVM is never significantly better than RVM. This will be true in general for two-way joins because the cost saved by RVM through sharing subexpressions is canceled by the overhead of maintaining α-memory nodes. If procedures contain joins of three or more relations (as in model

2) RVM can perform better than AVM. This is possible because there will be precomputed subexpressions containing joins of two or more relations. These subexpressions can be used to limit the total number of joins that RVM must perform compared to AVM. For example, in model 2, RVM only has to compute a two-way join, but AVM must do a three-way join.

The relative frequency of updates to different relations is an important factor that was not analyzed in this chapter. Static optimization methods will use statistics on relative update frequency when designing an optimal plan for maintaining procedures (e.g. an optimized Rete network). Hence, the plan produced will be efficient for the given update pattern. Because of this, it is expected that the benefits of static optimization observed in the analysis performed in this chapter will be observed in actual application. However, further study of staticly optimized procedure (or view) maintenance algorithms is needed before this can be concluded with certainty.

The shared version of AVM described in chapter 3 was not analyzed in this chapter. Shared, static AVM will probably outperform both non-shared, static AVM and RVM when the sharing factor is high. This is likely because shared, static AVM benefits from shared subexpressions without paying overhead to maintain the Rete network memory nodes used in RVM. A potential drawback of the staticly optimized algorithms is their fixed execution plan (e.g. the Rete network), which may cause them to become more costly than dynamicly optimized algorithms if the structure of the database or the update frequency changes significantly. Experience is needed to know whether the drawbacks of the fixed execution plan used in staticly optimized algorithms will overwhelm the advantages gained by avoiding run-time compilation overhead, and by combining shared subexpressions.

An important issue with the Cache and Invalidate and Update Cache strategies is how to decide whether or not to maintain a cached copy of given object. Sellis has considered this issue for Cache and Invalidate [Sel86b, Sel87]. The question is even more important for Update Cache because the potential cost of a wrong decision (e.g. maintaining an object when the update proba-

bility is too high) is much larger than for Cache and Invalidate. How to make this decision when using Update Cache is an interesting problem for future study.

One would expect the results of database procedures to be small in most applications. This expectation combined with the observations made in this study suggest the following strategy for implementing database procedures. Always Recompute should be implemented first because it is simplest. If sufficient resources are available to implement a second method, Cache and Invalidate should be chosen. It will give good performance benefits for small objects, and it does not degrade significantly if the system makes a mistake (e.g. by caching an object that is seldom accessed). The Update Cache strategy can be added later if the programming effort can be justified. This will make it possible to efficiently maintain stored procedure values that are large. The same code written to implement Update Cache can be used to support materialized views and complex trigger condition testing as well.

CHAPTER 6

AN ENHANCED DATABASE RULE LANGUAGE

The initial proposal for POSTGRES database management system allows powerful rules to be defined by tagging POSTQUEL commands with the keyword **always**. There are, however, some difficulties with the **always** rule proposal. These difficulties can be broken down into two separate dimensions. First, **always** rules suffer from shortcomings in the POSTQUEL language itself, since such rules are simply POSTQUEL commands tagged with a special modifier. Second, the semantics of **always** rules specify that a command tagged as such appears to run forever (as a practical matter, it runs until it no longer changes the database). It is not clear that these semantics alone are adequate for building rule-based applications. This chapter analyzes both these issues. Based on this analysis, enhancements to POSTQUEL are proposed to increase its power, and some new rule semantics different than **always** are proposed.

6.1. Weaknesses in the Query Language

6.1.1. Negated Conditions

A feature commonly needed in rule-based systems is the ability to specify a *negated* rule condition. Negated conditions are true if *there does not exist* any record or collection of records in the database matching some pattern. Measurements of several large expert systems written in OPS5 show that approximately 30% of all production rules contain one or more negated condition elements [GuF83]. Negated conditions are sometimes used to define rules that draw some conclusion if no evidence to the contrary exists. For example, in an expert system for automobile troubleshooting, the following rule might be specified:

if the starter won't crank **and**
  it is *not* known that the battery is charged
then
  hypothesize that the battery is dead

In the original QUEL query language, there is no direct way to specify a negated condition [HSW75]. However, the "any" aggregate can be used to check if there are no values matching some condition as follows:

any (*target_list* [**from** *from_list*] [**where** *qual*]) = 0

(The **count** aggregate can be used in the same fashion.) Use of an **any** or **count** aggregate is an inconvenient and unintuitive method for specifying negated conditions. Thus, a special aggregate **no** which returns a boolean value is proposed here. A new aggregate syntax has been proposed for POSTQUEL, the POSTGRES query language [RoS87,StR86] (see Appendix 2 for a description of the new aggregate syntax). The new syntax allows grouping in aggregate functions to be specified explicitly, rather than implicitly as in the original QUEL [Eps79,HSW75,SWK76]. Based on the POSTQUEL aggregate notation, the syntax of the **no** function is as follows:

no '{' *target_list* [**from** *from_list*] [**where** *qual*] '}'

The meaning of this function is the same as the expression above using the **any** aggregate. Note that **no** does not have to be built-in to the POSTGRES system since it can be defined using the facility for creating user-defined aggregates.

By using **no** like an aggregate function, a negated clause can be linked to the outer part of a query. For example, consider a situation where there are two relations:

STUDENT (id, name, ...)
ENROLLED (id, name, ...)

STUDENT contains information about students who were enrolled last year, and ENROLLED contains information about currently enrolled students. The field "id" of both relations is a unique identifier for a student. Suppose one wishes to delete all STUDENT records for students

who are not currently enrolled. This deletion can be expressed as follows using **no**:

```
delete STUDENT where
no{ENROLLED.id where STUDENT.id = ENROLLED.id}
```

The above is much simpler than the following equivalent command expressed in standard QUEL:

```
delete STUDENT where
any(STUDENT.id by STUDENT.id
    where STUDENT.id = ENROLLED.id) = 0
```

It is also simpler than the equivalent command expressed in the POSTQUEL aggregate syntax using the **any** aggregate:

```
delete STUDENT where
any{ENROLLED.id where STUDENT.id = ENROLLED.id} = 0
```

It is clearly easier to express negation using **no** than using an **any** or **count** aggregate. However, an even simpler syntax is possible for specifying some negated conditions. It is proposed here that the operators **in** and **not in** also be added to POSTQUEL (both these operators are included in the SQL query language [CAE76]). The syntax of the **in** and **not in** operators for POSTQUEL is

scalar_exp **op** *set_exp*

where **op** is either **in** or **not in**, *scalar_exp* returns a single value and *set_exp* returns a set of values. A *scalar_exp* has the same syntax as defined for expressions in POSTQUEL. A *set_exp* has the form

'{' *scalar_exp* [**from** *from_list*] [**where** *qual*] '}'

(the argument to a POSTQUEL aggregate is also a *set_exp*). The syntax of the *set_exp* is nearly identical to the *relation-constructor* mechanism of the RIGEL database programming language [RoS79].

The **in** and **not in** operators have *nested iteration* semantics, as in SQL*. In general, a

* POSTQUEL aggregates also have nested iteration semantics.

query containing an **in** or **not in** operator has the following structure, where **op** is either **in** or **not in**:

retrieve (*target_list*)
where Q_1 **and** exp_1 **op** {exp_2 **where** Q_2}

The portion of the query

retrieve (*target_list*)
where Q_1

is referred to as the *outer block* and

{exp_2 **where** Q_2}

as the *inner block*.

As an example, using the **not in** operator, the **delete** statement above can be specified as follows:

delete STUDENT **where**
STUDENT.id **not in** {ENROLLED.id}

This example demonstrates that the **not in** operator simplifies the specification of negated conditions even further than the **no** function in some situations.

The **no** function and the **in** and **not in** operators are not strictly necessary. Both **no** and **not in** can be simulated using aggregates as shown in previous examples. The **in** operator can be simulated using an equi-join, as described in [GaW87,Kim82] (details of how these transformations are performed will be discussed in a later section). The **no, in** and **not in** facilities have been added to the query language because they allow a large class of queries, updates and rules to be specified in a more convenient and understandable way.

6.1.2. Executing a List of Statements

When specifying rules in an expert systems application, it is important that a list of statements can be executed when a rule fires. If all rules can execute only one statement in their action, programming becomes difficult. For example, a technique commonly used in expert systems is to reason using *certainty factors* (CF's) like those found in MYCIN [ShB75,Sho76]. A certainty factor is a real number in the range [0,1] which indicates the measure of belief in a fact. The belief status of a fact with a CF of 0 is completely unknown, while a fact with a CF of 1 is believed with certainty. In a forward chaining rules system like OPS5 or ART, certainty factor combination is typically done using a rule with the following structure:

if
  there is a fact F_1 supporting conclusion A with certainty X **and**
  there is a fact F_2 supporting conclusion A with certainty Y
then do
  create a new fact F_3 supporting conclusion A with certainty $X+Y-X\cdot Y$
  delete F_1
  delete F_2
end

This rule has multiple commands in its action. It would be cumbersome to program it in a language that allowed the action of a rule to contain only one statement.

All rules specifiable in the rule sublanguage of POSTQUEL are single POSTQUEL commands tagged with a modifier [StR86,SHP87] In POSTQUEL, using the execute command it is possible to specify a list of statements to be run. Hence, it is possible to create a rule who's action is a list of statements as follows:

execute (dummy = " ... list of statements ... ")

Unfortunately, the execute command is awkward for specifying a block of commands to run because a dummy field name and extra quotation marks are required. Hence, the following variation of the execute command is proposed (the existing execute syntax is still legal):

execute (*stmt* { ; *stmt* })

For example, the following command purges all information about the Toy department from the employee/department database:

```
execute ( delete EMP where EMP.dept="Toy" ;
          delete DEPT where DEPT.dname="Toy" )
```

The primary advantages of the new variation of the **execute** command are that it provides a clean way to specify a collection of commands to be executed, and it facilitates specification of rules that have more than one command in their action (use of **execute** to specify rules will be discussed in section 6.3). This **execute** syntax makes it explicit that a group of statements are to be executed together. The database system is therefore free to optimize them all at once, forming an efficient joint execution plan [Sel86b]. Also, the system can attempt to execute the statements in parallel, which may yield significant performance improvements on multi-processors (e.g. SPUR [Hil86]), or even on a uni-processor if increased disk utilization can be achieved.

6.1.3. A Conditional Abort Command

It is sometimes necessary for database transactions to abort themselves due to exceptional conditions. For example, an abort might be required if the user presses the BREAK key, or if the database application determines that an update violates an integrity constraint. Rules designed to enforce integrity constraints may also need to abort the current transaction if the constraints are violated. To facilitate transaction aborts in both user applications and rules, a command **abort** is proposed here. The syntax of **abort** is as follows:

abort [(*target_list*)] [**from** *from_list*] [**where** *qual*]

The target list, **from** clause, and **where** clause are all optional. If the target list is not specified, a dummy target list of the form

(DummyAttribute = "constant")

is implicitly created. The **abort** command aborts the current transaction if there are any tuples retrieved by a query with the *target_list, from_list,* and *qual* specified. For example, consider the following relation containing one record for each employee of a city government:

PUBSERVANTS(name, ...)

Suppose there is a law that no individual can hold more than one city job. An application could run the following **abort** command to enforce this integrity constraint (tid is a unique tuple identifier field):

```
abort
from p1, p2 in PUBSERVANTS
where p1.name = p2.name
and p1.tid != p2.tid
```

Because **abort** is a full-fledged query language command, it can be used in conjunction with the rule syntax to be described later. This will allow construction of rules to enforce integrity constraints.

6.2. Processing Commands Containing Proposed Features

Normal query processing can be extended in a straightforward way to handle the **no**, **in** and **not in** functions, the new variation of the execute command, and the **abort** command. The function

no { *target_list* [**from** *from_list*] [**where** *qual*] }

can be defined in POSTGRES as a user-defined aggregate. It would be implemented in a way very similar to the **any** aggregate, except that it would return a boolean value FALSE when **any** would return the integer value 1, and a boolean value TRUE when **any** would return the integer 0.

To allow processing of the **in** operator, query modification can be applied as described in [Kim82]. A query of the form

retrieve (*target_list*)
where Q_1 **and** (exp_1 **in** (exp_2 **where** Q_2))

can be translated into the following normal query:

retrieve (*target_list*)
where Q_1 **and** Q_2 **and** $exp_1 = exp_2$

The resulting command is an ordinary **retrieve** which can be submitted directly to the query processor. The query optimizer can often find a more efficient execution plan for the conventional form than for the nested form. For a complete description of the algorithms for translating nested queries into a canonical representation without nesting, the reader is referred to the paper by Kim [Kim82] and subsequent papers that describe some errors in Kim's work and their solution [GaW87,Kie84].

Implementation of the **not in** operator is straightforward because it is a special case of the **no** function. The following two commands are equivalent:

C1.

retrieve (*target_list*)
where Q_1 **and** exp_1 **not in** (exp_2 **where** Q_2)

C2.

retrieve (*target_list*)
where Q_1 **and no(** exp_1 **where** $exp_1 = exp_2$ **and** Q_2)

In C1, Q_1 consists of all the parts of the qualification that are not part of either the left or right operands of the **not in** operator. To process a query of type C1, it can be transformed into one of type C2, which can be reduced to one using an **any** aggregate as discussed previously.

Consider an **execute** command with the following form:

execute (*stmt* { ; *stmt* })

This command can be processed easily by executing the statements one after the other. As mentioned previously, performance can be improved in some cases by applying multi-statement query

optimization techniques and also by attempting to parallelize execution of the statements.

An abort command has the following structure:

abort [(*target_list*)] [**from** *from_list*] [**where** *qual*]

This command can be implemented easily using the following steps:

1. form the query

   retrieve (*target_list*) [**from** *from_list*] [**where** *qual*]

2. Begin execution of this query. If a tuple is returned, halt execution and perform any actions needed to abort the current command (e.g. undo the effects of the transaction if necessary).

3. Otherwise, if no tuples are returned, continue execution of the current transaction.

6.3. Alternate Rule Semantics

Although the usefulness of **always** semantics for rules is clear, there are other possible meanings for rules. A list of desirable rule semantics with significantly different properties includes:

rule type	*rule semantics*
always	appear to always have just been run
new	execute once whenever one or more new tuples match qualification
old	execute once whenever one or more tuples that used to match qualification no longer match

These rule semantics have strengths that are quite different, so it is instructive to look at the usefulness of each for different applications. Because they appear to run forever, **always** rules are a natural mechanism for maintaining certain types of complex integrity assertions. For example, consider an assertion that all employees must make the salary specified for their job. Supposed the correct salaries are specified using a relation

SALTABLE(job, salary)

Using this relation, the desired integrity assertion can be enforced using the following **always** rule:

always replace EMP (salary = SALTABLE.salary)
where EMP.job = SALTABLE.job

An area where **always** rules do not provide the desired semantics is in certain types of expert systems applications. For example, consider the following collection of rules that might be taken from an expert system for automotive trouble-shooting:

if
the car won't start **and** there is a gasoline smell
then
hypothesize that the carburetor is flooded

if
there is a leaking fuel line **and**
there is a hypothesis that the carburetor is flooded
then
delete the hypothesis that the carburetor is flooded

Using **always** semantics, these two rules cause an infinite loop, since when the second rule deletes the hypothesis that the carburetor is flooded the first rule wakes up and re-asserts the hypothesis. In this situation, **new** semantics are needed since they will cause the rules to behave as desired, i.e. execution of the second rule will not cause the first rule to wake up again. Rules with **new** semantics are similar to production rules found in expert system shells, including OPS5, KEE, ART, and many others [FiK85,For81,Gev87,Sho87]. The semantics of production rules have proven to be quite useful for constructing rule-based applications in those systems.

Both **new** and **old** rules are useful for specifying *transition constraints*. These include simple constraints, such as "an employee's new salary may not be more than 10% of their old salary," as well as more complex constraints like *referential integrity* [Dat81b]. Examples of use of **new** and **old** rules to enforce these kinds of integrity constraints will be shown in section 6.3.2.

There are some forms of constraints for which the **old** semantics are essential. For example, a different type of referential integrity rule might specify that no DEPT record can be deleted if there are still employees working in that department. A rule like this must be able to wake up when data is deleted from the database, which is not possible using **new** and **always.**

The three different types of rule semantics discussed so far, **always, new,** and **old,** are each useful in their own right. Together, they allow a broad class of useful rules to be specified. Furthermore, **always, new,** and **old** rules are significantly different from each other – there is no straightforward way to simulate any one with the others. Hence, we argue that all three types of rules should be provided by a database rules system.

One might make the case that there are other types of rules besides **always, new,** and **old** that are useful. A list of other possible rule semantics is shown below:

rule type	*rule semantics*
once	execute once when new tuple matches condition and delete self
repeat N	same as **new**, but deletes self after N wake-ups
at *list-of-times*	execute once daily at each time shown on list-of-times
sleep t	execute once every t time units

This list could probably be extended indefinitely. Since it would be virtually impossible to provide every desired type of rule semantics, the position taken here is that the best approach is to provide only a small, general set of rule semantics (i.e. **always, new** and **old**). These can be used to simulate the others. For example, **once** can be simulated easily with **new** by having a rule delete itself with the last statement in its action (an example of this will be given in section 6.3.2). The **repeat** semantics can be simulated with **new** by having a rule count how many times it has run, and delete itself after N executions. Similarly, if there is a relation TIME that is updated periodically by the system (say once per minute), **new** can be used to simulate **at** and **sleep** by referring to TIME in rule conditions.

6.3.1. Language Features to Support Alternate Rule Semantics

An attractive feature of **always** rules, is that they can be specified using a simple extension of the database query language – ordinary POSTQUEL commands can be turned into rules just by tagging them with a special modifier. This is an elegant property of the rule language that is desirable to maintain. Thus, rules with **new** and **old** semantics will be specified in the same way, by tagging POSTQUEL commands with the keywords **new** and **old** respectively.

The general syntax of rules in the proposed version of POSTQUEL is shown in figure 6.1*. When a statement with this syntax is submitted to the DBMS, the appropriate rule is registered with the system. The *rule_target_list* may be either an ordinary target list, or the special symbol '*', which acts as a place holder in the case that no target list is specified. If '*' is used instead of an explicit target list, the '*' is implicitly replaced with a target list defined as follows:

---

rule → *tag command*

tag → **always** | *meta_tag*

meta_tag → **for** [**new** | **old**] *rule_target_list*
[**from** *from_list*]
[**where** *qual*]

rule_target_list → '*' | (*target_list*)

command → *retrieve*
| *replace*
| *delete*
| *append*
| *do*
...

Figure 6.1. General rule syntax

---

*A BNF grammar is mixed with the [] and {} notation to describe the syntax here. Non-terminals are shown in italics and terminals are shown in bold.

1. If there are no tuple variables at the outermost nesting level in the **where** clause, '*' is replaced by a dummy target list with the following definition:

(DummyAttribute = "constant")

2. If there are tuple variables $t_1, t_2, \cdots t_k$ at the outermost nesting level in the **where** clause, then '*' is replaced by the following:

(t_1.all, t_2.all, ... , t_k.all)

Rules with **new** and **old** semantics can refer to two special sets of tuples:

NEW	those tuples most recently matching (*rule_target_list*) **where** *qual*
OLD	those tuples that just left (*rule_target_list*) **where** *qual*

NEW and OLD cannot be referenced in **always** rules. The NEW and OLD sets will be formed through the use of an immediate view maintenance algorithm; details of this are discussed in section 6.4.

The precise meaning of **new** and **old** rules is specified below. A **new** or **old** rule R has a **for** clause associated with it. This **for** clause has a target list and a qualification, which implicitly defines a database view V. Let V_0 be the initial contents of V before the start of an update transaction T. Suppose T executes and attempts to commit. Let the new contents of V (reflecting the updates made by T) be V_1. The definition of the OLD and NEW sets at this point is as follows:

$$\text{OLD} = V_0 - V_1$$
$$\text{NEW} = V_1 - V_0$$

If the rule R has **new** semantics and NEW is not empty, then R is eligible to run. If R has **old** semantics and OLD is not empty, then R is eligible to run. Otherwise, R is not awakened. When R runs, it has read-only access to the values stored in NEW and OLD. NEW and OLD cannot be updated.

In general, a transaction in the DBMS extended with rules will consist of the execution of the body of the transaction T, followed by execution of zero or more rules, $R_1, \cdots R_N$. Consider a point in time P that lies after execution of T and before the first rule R_1, or between execution of any adjacent pair of rules, R_i and R_{i+1}. At point P, the values of the NEW and OLD sets reflect all changes to the database made after the start of the transaction T, and prior to P. Suppose P lies just before rule R_i. The value of V at this point is V_i. During execution of rule R_i, the OLD and NEW sets have the following values:

$$\text{OLD} = V_0 - V_i$$
$$\text{NEW} = V_i - V_0$$

There is an exception to the above if the same rule executes more than once. Suppose that one rule R occurs in the sequence $R_1, \cdots R_N$ as both R_i and R_j, where $j>i$. In this case, the value of OLD and NEW seen by the execution of R_j is the following:

$$\text{OLD} = V_i - V_j$$
$$\text{NEW} = V_j - V_i$$

The above discussion has not considered the issue of how to select the order of execution $R_1, \cdots R_N$ for the rules. This question will be addressed in a later section.

6.3.2. Examples Using New Rule Syntax

Rules of the form described above are useful for a variety of purposes. Several examples are given below:

One-Shot Rules

As mentioned previously, there are situations where a rule should execute only once, and then disappear (this is the once rule semantics described previously). One-shot rules can be implemented using new rules in combination with the POSTGRES remove rule command. For example, consider the following rule, which waits for a record for an employee named "Bob" to be

inserted, and then performs its action and removes itself:

```
define rule BobRule is
  for new (EMP.all) where EMP.name = "Bob"
  execute
  (
    ... rule action ...

    remove rule BobRule
  )
```

Logging Update History

The following rule makes a log entry each time a record of a Toy department employee is inserted or modified.

```
for new (EMP.all) where EMP.dept = "Toy"
append to TOYLOG (NEW.all, time=TOD(), user=User())
```

When this rule executes, the temporary relation NEW contains all newly inserted or modified EMP with the value "Toy" in the dept field. Hence, the desired records are appended to TOY-LOG when the rule executes.

Transition Constraints

By making use of the **abort** command, the rule below aborts the current transaction if a Toy department employee gets a raise of more than 10%.

```
for old (EMP.all) where EMP.dept = "Toy"
abort where NEW.salary > 1.1*OLD.salary and NEW.name = OLD.name
```

Non-always semantics

A problem with **always** semantics in some situations is that the rule always wakes up when data it has written is updated by the user. For example, one might wish to specify the rule

```
If Bob's salary changes
then Set Jim's salary to Bob's salary
```

Using an **always** rule, one would attempt to specify this as follows:

```
always replace EMP ( salary = E.salary )
from E in EMP
where EMP.name="Jim" and E.name = "Bob"
```

However, this rule will cause any user update to Jim's salary to be refused [SHP87]. This is not the desired meaning. A rule with the correct semantics can be specified using new as follows:

```
for new (EMP.all) where EMP.name = "Bob"
replace EMP (salary=E.salary)
from E in EMP
where EMP.name = "Jim" and E.name = "Bob"
```

Notice that the above rule will run whenever any field of Bob's EMP record is updated. The rule condition can be modified as shown below so that it only fires when the salary field of Bob's record is updated:

```
for new (EMP.salary) where EMP.name = "Bob"
replace EMP (salary=E.salary)
from E in EMP
where EMP.name = "Jim" and E.name = "Bob"
```

Expert System Support

The new semantics are also suitable for supporting rule-based expert system applications. Consider an expert system designed to assist a broker in trading stocks. This system uses a large shared database of information on corporations. Some rules that might be present in such a system are shown below:

Automated Stock Trader:

```
rule1:
      If at least 4 aluminum companies have value > X
            and
       there is no goal to analyze the metal market
      then hypothesize aluminum strong


rule2:
      If at least 4 steel companies have value > Y
            and
       there is no goal to analyze the metal market
      then hypothesize steel strong
```

```
rule3:
      if hypothesize aluminum strong
            and
        hypothesize steel strong
      then
        delete aluminum hypothesis
        delete steel hypothesis
        create a goal to analyze metal market


rule4:
      if goal is to analyze metal market
      then execute metal market analysis procedure
```

In actual implementation, the expert system might be based on a database with the following schema:

```
company(name, product, value)    -- company database
hypothesis(name)                 -- current hypothesis
goal(name)                       -- current goals
procedures(name, code)           -- collection of stored procedures
```

Using new notation, the rules above can be written as follows:

```
define rule rule1 is
     for new * where
        count(company.tid where company.product = "aluminum"
              and company.value > X) ≥ 4
      append to hypothesis(name="aluminum strong")


define rule rule2 is
      for new * where
         count(company.tid where company.product = "steel"
               and company.value > X) ≥ 4
      append to hypothesis(name="steel strong")


define rule rule3 is
      for new (h1.all,h2.all)
      from h1,h2 in hypothesis
      where h1.name="aluminum strong"
        and h2.name = "steel strong"
      execute (
            delete hypothesis where hypothesis.name="aluminum strong" or
                hypothesis.name="steel strong" ;
            append to goal(name="Analyze Metal Market")
      )
```

```
define rule rule4 is
      for new (goal.name) where goal.name = "Analyze Metal Market"
      execute (procedures.code)
            where procedures.name="MetalsAnalysis"
```

This example would be extremely difficult to specify using only the previously proposed features of the POSTGRES rule language. For example, rule 3 benefits from the use of the execute command since its action contains multiple statements. Furthermore, if always semantics were used, rule 3 would interact with rules 1 and 2 in an undesirable way. The first two rules would wake up and reinsert the hypothesis tuples "steel strong" and "aluminum strong" when rule 3 deleted them unless specific measures were taken to prevent it from happening.

6.4. Testing Rule Conditions

This section discusses methods for testing the conditions of rules with old and new semantics. Testing of both negation-free conditions and conditions with negation is discussed. The issue of conflict resolution is also addressed.

6.4.1. Conditions Without Negation

Any immediate view maintenance algorithm can be used to test the conditions of new and old rules that do not use negation (deferred view maintenance algorithms are not applicable because it is essential to know immediately whether a rule should fire). The new semantics can be implemented by making a rule eligible to run whenever a set of new tuples (A_{net}) enters the view corresponding to the condition of the rule. The special tuple variable NEW ranges over A_{net} during execution of the rule. The old semantics are implemented in a similar fashion, except that a rule becomes eligible to run whenever a set of old tuples (D_{net}) leaves the view defined by the rule condition. The tuple variable OLD ranges over D_{net} when the rule executes.

Because any immediate view maintenance algorithm can be used to test rule conditions, one would like to determine which algorithm performs best in this application. To analyze the cost of condition testing, the assumption can be made that conditions have the same structure as the database procedures analyzed in chapter 5. The only difference in cost between procedure maintenance and rule condition testing is that the query cost (i.e. the cost to read the procedure value) is not included for condition testing. Only the maintenance overhead is included. The query cost is exactly equal for both differential view maintenance algorithms analyzed in chapter 5 (RVM and static, non-shared AVM) so comparing the cost of the two for rule condition testing would yield the same results as for procedures. Hence, the performance results found when comparing the two algorithms in chapter 5 are valid for rule condition testing as well.

A performance problem with testing rule conditions using a view maintenance algorithm is that the view defined by the condition must be maintained on disk, and this can be expensive. It is necessary to maintain a complete copy of the view because it may contain duplicates. A tuple that is "inserted" into the view might thus result only in causing the duplicate count of an existing tuple to be incremented. Such a tuple does not constitute a net change to the view so it is not placed in the NEW set for the rule corresponding to the view. Fortunately, there is a restricted class of rule conditions for which the view does not have to be physically stored, resulting in substantial cost savings. Below, this class of conditions is described, and a method is presented for computing the NEW and OLD sets for conditions that belong to the class. The class includes all relational algebra expressions R that satisfy the following restriction:

Restriction A:

(1) All base relations appearing in R are stored with duplicates removed,

and

(2) R does not include the projection operation, or R projects a unique tuple identifier for each relation appearing in the expression.

For an expression R that satisfies Restriction A, it is possible to determine NEW and OLD

without maintaining a physically materialized copy of the entire view. If a tuple is found to be in NEW after an update transaction, it is guaranteed to be a new tuple for the view because every tuple in the view is unique (similarly for OLD). Uniqueness is assured by Restriction A.

There is a significant performance advantage using expressions that satisfy Restriction A: there is no overhead to write data to a physically materialized copy of the result of the expression. The only cost is to compute A_{net} and D_{net} for each view after each update transaction.

As an example, consider the following POSTQUEL target list and qualification, which represents the condition of a new rule.

```
( EMP.all , DEPT.all )
where EMP.dept = DEPT.dname
and EMP.age > 50
and DEPT.dname = "Fire"
```

Assume that there is B-tree index on EMP.age. When the rule with this condition is installed, t-locks will be set in this index to lock the range of values age > 50. This expression satisfies Restriction A because there is no projection (all attributes of EMP and DEPT appear in the result, and it is assumed that there are no duplicates in EMP or DEPT). Suppose that a transaction appends the following single tuple EMP:

t = <name = "Bob", dept = "Fire", age = 55, salary = 20K, job ="Firefighter">

This tuple will break the t-lock set on EMP.age > 50. The value of A_{net} can be found by executing the following query:

```
retrieve ( t.all , DEPT.all )
where t.dept = DEPT.dname
and t.age > 50
and DEPT.dname = "Fire"
```

There is no need to maintain a materialized view containing all the tuples that match the rule condition.

6.4.2. Conditions With Negation

Rule conditions using negation can also be tested using extended versions of the Rete and algebraic view maintenance algorithms. The **no** function can be implemented directly in RVM using **not** nodes in the Rete network (see the description of the Rete match algorithm in Chapter 1). Using AVM, negated conditions can be implemented by converting the **no** function into the equivalent condition based on the **any** aggregate, and then using the algorithm for maintaining views with aggregates presented in Chapter 3. The **not in** function reduces to **no**, so it can be implemented using the same method.

6.4.3. The Rule Execution Strategy

As discussed by Eswaran [Esw76], to maintain serializability and recoverability of transactions in a DBMS enhanced with triggers, the rules awakened directly or indirectly by a transaction must run as part of that transaction. To achieve this, the concurrency control subsystem must treat reads and writes performed by rule actions as part of the transaction that caused the rule to fire. Similarly, the recovery subsystem must associate all of a rule's reads and writes with the transaction that triggered the rule. If these protocols are observed, concurrency control and recovery methods that are correct for normal transaction processing will work properly in the presence of rules.

A transaction in a DBMS enhanced with triggers has the following two parts, which are run in order:

(1) execution of the body of the transaction

(2) execution of rules

After (1) and during (2) there is a collection of rules that are eligible to run, which is known as the *conflict set*. The DBMS must use some strategy for executing these rules. Borrowing terminology from the OPS5 expert systems shell, the strategy for executing triggers is called the

Recognize-Act Cycle [For81]. This cycle consists fo the following steps:

> (1) *Conflict Resolution*: Select one eligible rule for execution. If no rule has a satisfied condition, cease execution of rules and attempt to commit the current transaction.
>
> (2) *Act*: Execute the action part of the chosen rule.
>
> (3) *Match*: Based on the changes to the database made by execution of the previous rule, determine which other rules are eligible to run, and add them to the conflict set. Go to step (1).

6.4.4. Conflict Resolution

The following conflict resolution scheme is proposed for the rules system described in this chapter (this is a modified version of the LEX strategy used in the OPS5 system [For81]). The set of rules eligible to run can be divided into the following categories:

(a) **always** rules that have been awakened due to a t-lock conflict

(b) **new** rules that have tuples in their NEW set

(c) **old** rules that have tuples in their OLD set

All these rules are treated uniformly. The *recency* of a rule is the time at which its condition was most recently satisfied, where time is measured as follows. All update commands executed as part of a transaction (including commands that are part of rule actions) are assigned a *sequence number*. These numbers are assigned to commands in strictly ascending order. The recency of an **always** rule is defined to be the highest sequence number of a command in the transaction that has broken a *t*-lock for the rule. The recency of a **new** or **old** rule is the highest sequence number of any command that has caused a tuple to be inserted into the NEW or OLD set, respectively. Rules are ranked according to their recency, with the most recent rule having highest priority.

Conflict resolution chooses one rule from the conflict set for execution using the following algorithm:

1. Discard from the conflict set any rules that have already fired. If a discarded rule is of the new or old variety, delete from the NEW and OLD sets associated with the rule all tuples that were present before execution of the body of the rule. If no rules remain, conflict resolution fails and no rule is returned.

2. If one rule is more recent than all the others, return it.

3. If more than one rule is tied as most recent, choose one of them at random and return it.

Prioritizing rules based on recency results in a depth-first execution of rules, i.e. rules triggered by other rules execute immediately without waiting for already-eligible rules to run. This is important because it has the effect of allowing localized tasks being performed by an expert system application to complete without being interrupted by other rules. Expert systems programmers depend on this depth-first rule execution for control flow in forward-chaining rule based applications like those written in OPS5 and ART [For81, Sho87].

6.4.5. Act

The rule returned by conflict resolution is simply passed to the query processor for execution.

6.4.6. Match

Rule matching is accomplished using one of the condition testing algorithms described in section 6.4.

6.5. Discussion

This chapter has proposed enhancements to the database rule language proposed in [Sto85]. Some of the enhancements were needed due to inadequacies in POSTQUEL itself. These include the **no** and **not In** functions to allow negated rule conditions to be specified easily, and the extension of the **execute** command to allow execution of a block of POSTQUEL statements in a convenient way. Other extensions were needed because the **always** modifier is not adequate to

specify all the types of rules desired for expert system applications. A modifier **for** was proposed which turns POSTQUEL commands into production rules similar to the forward chaining rules found in OPS5 and other expert systems shells. These rules can have either **new** or **old** semantics. Methods for implementing rules using these new features were also discussed. In particular, the algebraic and Rete view maintenance algorithms can be used to test the conditions of **new** and **old** rules. It was argued that the relative performance of RVM and static, non-shared AVM when applied to testing rule conditions will be the same as observed in chapter 5. Finally, a general strategy was proposed for executing **always**, **new**, and **old** rules in a DBMS.

CHAPTER 7

CONCLUSION

The focus of this thesis has been the design and analysis of methods for efficiently supporting rules and derived objects in a relational database system. Section 7.1 summarizes the results presented in chapters 2-6. Section 7.2 provides a comparison of this work with other research. Section 7.3 discusses the implications of this thesis for the field of database management. Section 7.4 considers some limitations of the work. Finally, section 7.5 discusses possibilities for future research on support for rules and derived objects in a DBMS.

7.1. Summary

In chapter 2, a new lock-based rule indexing algorithm called Mark Intersection was proposed, and compared with two other lock-based rule indexing algorithms (Basic Locking and Reduced Basic Locking). The results showed that Mark Intersection is at worst approximately equivalent to Basic Locking in efficiency, and is superior if predicates tend to have more than one term that lies on an indexed attribute. However, Basic Locking performs quite well in most cases, it uses less storage space than Mark Intersection, and it is significantly easier to implement. Hence, Basic Locking is the algorithm of choice unless most predicates have a large number of terms, and many of them lie on indexed attributes. Reduced Basic Locking is a space-saving version of Basic Locking that is applicable only if updates in place are implemented as deletes followed by inserts (otherwise a large amount of wasted I/O must be done to search every index on a relation after a tuple is modified). Mark Intersection and Basic Locking can be used to help support both triggers and inference rules, but Reduced Basic Locking is only viable for triggers.

Chapter 3 presented a collection of techniques for maintaining materialized copies of objects derived from a database. A new incremental view maintenance algorithm called Rete View Maintenance was proposed and proved correct. An interesting feature of the RVM algorithm is that it is staticly optimized, and shared, i.e. a complete execution plan for maintaining views (the Rete network) is compiled in advance, and shared subexpressions are evaluated only once. The AVM algorithm proposed by Blakeley [BLT86] is dynamically optimized since planning how to update materialized views is done after each transaction that modifies the base relations. Also, standard AVM does not take advantage of shared subexpressions (i.e. it is non-shared). The following variations of AVM where also proposed in the chapter:

1. dynamic, shared
2. static, non-shared
3. static, shared

In addition, any of the view maintenance algorithms discussed in chapter 3 can be implemented in either an immediate or deferred manner. Based on the use of any view maintenance algorithm, a class of algorithms was developed for maintaining materialized aggregates and aggregate functions. Finally, methods were proposed for materializing database procedures, as well as views and procedures containing aggregates and aggregate functions.

Chapter 4 analyzed the performance of different methods for answering queries that refer to views, including query modification, deferred view maintenance, and immediate view maintenance. An interesting finding was that deferred and immediate view maintenance performed almost equally, independent of the parameters of the model. One reason for this is that the advantage of the deferred strategy (i.e., processing large sets of tuples to update views instead of small ones and thus doing less total I/O) is approximately offset by a disadvantage (i.e., the overhead of maintaining base relations using a hypothetical relation algorithm). The other more important reason is that if the ratio of updates to queries is low, the cost of processing queries far outweighs the cost to incrementally update views. Hence, the fairly small differences in the cost of immediate

and deferred view maintenance strategies become insignificant.

Since view materialization algorithms are close in cost, the main issue becomes whether to maintain a view in materialized form and query it directly versus performing query modification to process view queries. For simple selection-projection views on a single table with a clustered access path available for processing view queries, materializing a view is almost never worthwhile because query modification reads nearly the minimum possible number of pages to process a query. Maintaining a materialized view becomes more attractive when the view involves joins. The reason materialization becomes attractive for views with joins is that the parts of a tuple in a materialized view reside on a single page, whereas if the view is not materialized, many pages must be read to construct the tuple. In other words, view materialization is a very effective data clustering mechanism.

Even though deferred and immediate view maintenance differ only slightly in cost, there are other reasons why one algorithm may be preferred. For example, deferred may be superior in some distributed database architectures because it limits communication overhead (e.g. ADMS± [RoK86]). Another is that if there is substantial free I/O and CPU time available, it can be put to use refreshing views if immediate maintenance is not used (one can view this strategy as intermediate between immediate and deferred).

Chapter 5 explored some performance aspects of algorithms for processing queries that retrieve the value of database procedures. The algorithms considered were

1. Always Recompute (construct procedure value from base relations)
2. Cache and Invalidate (read the stored procedure result if it is valid; otherwise compute the result and write it back)
3. Update Cache (use a view maintenance algorithm to maintain and incrementally update the stored procedure value)

Two different versions of Update Cache were considered: one based on a non-shared view materialization algorithm (AVM), and another based on a shared one (RVM). There are dramatic

differences in performance between Always Recompute, Cache and Invalidate, and Update Cache. The differences in performance between the shared and non-shared versions of Update Cache are less pronounced.

One finding was that the cost of Cache and Invalidate is highly sensitive to the cost of invalidating a cached object. If it is necessary to read and write a page from an object just to invalidate it, Cache and Invalidate performs poorly. Hence, it is important to used some technique to reduce the cost of invalidating cached objects (e.g. a table in battery-backed primary storage listing the validity status of each object). Assuming that some low-cost invalidation technique is used, Cache and Invalidate performs approximately as well as Update Cache if update probability is low, objects are small, and there is some locality of reference among cached objects.

Cache and Invalidate always becomes superior to Update Cache as the update probability approaches 1. The reason Cache and Invalidate outperforms Update Cache for high update probability is that repeatedly invalidating a cached object costs almost nothing, but updating an object many times per query is expensive. Hence, Cache and Invalidate is a much safer algorithm to use than Update Cache if there is a possibility that the ratio of updates to queries might be high.

Another finding was that Cache and Invalidate is not a good algorithm to use if objects are large (e.g., more than one disk page in size). Update Cache is superior for large objects because the bigger an object is, the more likely it is that an update transaction will invalidate it. When a large object is invalidated, only a small part of it has usually changed. Thus, Update Cache can maintain a large object very efficiently after it has been made invalid, while Cache and Invalidate must pay a high cost to recompute the object.

In chapter 6 some enhancements to the POSTQUEL query language [StR86] were proposed to increase its power for specifying queries and rules. A function **no** was developed to allow negated conditions to be specified. Addition of operators **in** and **not in** was also proposed as a

way to make certain queries and rules easier to specify. It was observed that the rule language formed by enhancing QUEL with the **always** modifier is not adequate to express some rules commonly needed in expert systems applications. One problem is that it is inconvenient to specify rules that have more than one command has their action. To solve this problem, a modified version of the command **execute** was proposed to allow a block of QUEL statements to be executed. Furthermore, the built-in iteration implied by **always** semantics makes it difficult to specify some rules. To overcome this problem, the addition of another modifier **new** was proposed. The semantics of **new** rules are similar to production rules in expert systems shells such as OPS5 [For81]. Finally, a modifier **old** was proposed to allow expression of rules that fire when data no longer matches a condition.

7.2. Comparison with Other Research

The analysis in chapter 2 focussed on the differences among lock-based rule indexing techniques. A previous paper on rule indexing in database systems [SSH86] compared one lock-based technique (basic locking) with a method based on the R-tree (predicate indexing). The paper concluded that if the R-tree structure is so large that it must be disk-based, then basic locking is superior to predicate indexing unless the number of rules covering each tuple is large. The relative performance of R-tree-based rule indexing and lock-based rule indexing is affected significantly by the total number of rules. However, the number of rules does not affect the relative performance of different lock-based techniques. Thus, if mark intersection and reduced basic locking were compared directly to predicate indexing, results similar to those found in [SSH86] would be expected.

In chapter 3, the discussion on view maintenance identifies the issue of static versus dynamic optimization, which is not discussed in previous work on the subject [BLT86]. Using a dynamically optimized view maintenance algorithm, an execution plan for incrementally updating views

must be found after each update transaction that affects those views. Thus, the issue of static versus dynamic optimization is important because planning overhead may be high using dynamic optimization, especially if many views are being maintained.

The aggregate materialization methods described in chapter 3 allow maintenance of aggregates or aggregate functions over the result of any query composed of selects, projects and joins. In this sense, they can be considered completely general. A previous paper on maintaining general aggregates describes techniques that are not applicable to the relational data model [KoP81]. Other work on maintenance of aggregate information has focussed on special cases. A simple example of this is the practice of maintaining aggregate information such as the sum of values in a column, the total number of tuples in a relation etc. Another special-case aggregate maintenance method is the ordered index known as the *A-tree* [Rub86,Rub87]. The A-tree allows fast computation of aggregates over ranges of an ordered collection of records. For example, consider the relation

EVENT(id, duration)

where an ordering of events is implied, and there is an A-tree on the "duration" attribute. This structure makes it possible to efficiently answer questions like the following:

1. Assuming that the first event starts at time 0, when does the event with id $= X$ start?

2. How much time elapsed between the start of event X and the end of event Y?

It would not be feasible to use the aggregate maintenance algorithms of chapter 3 to maintain materialized answers to every possible query of this form that might be asked. The techniques of chapter 3 are applicable to a different problem: maintaining the answer to a single aggregate query. Of course, if it were important to be able to efficiently answer range queries over a view V, and both materialized views and A-trees were available, V could be materialized, and an A-tree index could be constructed on top of V.

The performance issues regarding different view materialization algorithms had not been studied in detail prior to the work presented in chapter 4. Previous work related to maintenance of derived data objects (e.g., views [BLT86,BuC79,RoK86] and database snapshots [AdL80,LHM86]) has focussed mainly on algorithms for performing the task.

The intent of the work presented in chapter 5 is to give a better intuitive picture of the tradeoffs between the different strategies for processing queries against database procedures (Always Recompute, Cache and Invalidate and Update Cache). The material presented there is the first detailed performance study comparing the three alternatives. Other work on database procedures discusses the implementation of the Always Recompute strategy in a version of INGRES [SAH84,SAH85], and suggests that Cache and Invalidate would perform well in some situations [StR86]. Work by Sellis focuses on the optimization issues involved in choosing whether to process procedure queries using Always Recompute or Cache and Invalidate [Sel86b,Sel87]. Sellis developed a method to decide whether an object should be cached depending on the cost criterion involved. He does not present any performance figures comparing the cost of Cache and Invalidate and Always Recompute for different parameter values. Also, his work does not consider the possibility of using a view maintenance algorithm to support an Update Cache strategy.

The work presented in chapter 6 proposes new features to be added to the query language POSTQUEL to allow easy specification of negated rule conditions. The ability to specify a negated condition in a convenient syntax is provided by the **not in** operator in the SQL language [CAE76]. In addition, chapter 6 describes an extension to POSTQUEL that allows complex triggers to be specified. These triggers are similar to the production rules found in expert systems shells (e.g. OPS5 [For81]). Triggers with production rule semantics differ from those with **always** semantics proposed in [Sto85]. These two styles of triggers complement each other since they have different strengths and weaknesses. Previous work on complex triggers in database systems considered only implementation issues [BuC79]. It did not propose an extension to a query

language to allow specification of rules. Other query languages with facilities to support triggers allow simple triggers only, not complex ones [CAE76,Esw76,How86].

7.3. Implications of This Work

A primary implication of this thesis is that it appears worthwhile to implement a view maintenance algorithm as part of a general-purpose relational DBMS. A great deal of mileage can be gotten from a single implementation of the view maintenance algorithm. The same code used to support materialized views can be used as a basis for materialized aggregates and database procedures. Furthermore, the code can be used to test complex trigger conditions efficiently.

Facilities to allow maintenance of derived objects will improve performance in query-intensive applications. The performance improvement can be great if common queries retrieve aggregate values or retrieve data from views or procedures that contain joins. There appears to be a large class of applications that utilize complex databases which are not frequently updated. Many statistical, scientific, technical and engineering database applications fit this model. These applications need the benefits provided by general-purpose relational systems (e.g. data independence, views, integrity control, protection etc.), but the performance of current database systems is in many cases not adequate [RKC87]. A database system equipped with the ability to efficiently maintain materialized objects would thus be a useful tool to support these applications.

It is quite significant that efficient implementation of complex triggers is now feasible through the combined use of rule indexing and view maintenance techniques. Although the notion of complex triggers has existed for several years [BuC79], such triggers have not been implemented in a general-purpose relational database system. A key reason for this is that it was not clear how to efficiently implement complex triggers in a DBMS up to this point. Finally, the results of chapter 6 show that it is possible to create a database rule language to support complex triggers with just a few extensions to a query language.

7.4. Limitations of Results

The limitations of this the results of this thesis are primarily due to simplifying assumptions that were required in the performance studies of chapters 2, 4 and 5. Some assumptions were necessary to make analytical performance evaluation tractable. For example, it is assumed in chapter 2 that all rule predicates have the same selectivity, and that ranges they cover are uniformly distributed over the data. Similar uniformity assumptions are made in chapters 4 and 5. Ideal assumptions are also made regarding placement of indexes, so that all join predicates can be processed using nested loop join with an index on the inner relation. Although simplifying assumptions were made, the results presented still serve as a useful basis of comparison for the algorithms studied. It should be possible to make well-informed design decisions based on the results. It will be appropriate to review the results later after some of the algorithms analyzed have been implemented in real systems and actual usage patterns are established. Implementation of lock-based rule indexing techniques, rules, and database procedures is being undertaken for the POSTGRES system [StR86]. The implementation will provide a useful testbed for further analysis of some of the performance issues studied in this thesis.

7.5. Directions for Future Research

The work presented in this thesis suggests several topics that should be explored further. First, the performance analysis done in this thesis could be extended by taking into consideration the total amount of main memory available for the buffer pool. This would give a better picture of the cost of the rule indexing techniques and the algorithms for maintaining derived objects that were analyzed in chapters 2, 4 and 5. Since rule indexing and object maintenance techniques appear to be potentially quite useful, a prototype implementation of them should be built into a DBMS. It will then be valuable to do empirical tests that compare the performance of object maintenance techniques with conventional methods.

If facilities for maintaining materialized objects are included in a relational database system, many new optimization problems arise. In general, there are two possible options:

1. Maintain a copy of the object and process queries against it by reading all or part of the stored copy.
2. Compute all or part of the object on demand.

Option 1 in effect provides new alternatives for the physical database designer. These new possibilities complicate the physical database design process. Now, physical database design involves the following tasks:

1. selection of primary and secondary indexes for base relations
2. selection of derived objects (e.g. views and procedures) to maintain in materialized form
3. selection of primary and secondary indexes for materialized views

An interesting area for future research is to develop both manual and automatic techniques for performing tasks 2 and 3. Sellis has addressed this this optimization issue for database procedures [Sel87]. This work can serve as a starting point for future research on how to decide whether or not to materialize a view. Previous work on physical database design can provide a foundation for future research on methods for deciding how to index a materialized view [HaC76, WoK80].

Once the decision is made to materialize a view and appropriate indexes are defined on the attributes of the view, the system must decide whether to process a query against the view by reading the materialized view or using a normal query plan that accesses the base relations. Processing the query using the materialized view is not always the best choice (e.g. the base relations might be clustered on an access path useful for processing the query, while the materialized view is clustered on another access path). Fortunately, extending a conventional query optimizer [Sel79] to make this decision is straightforward. The optimizer can find the best plan for processing the query normally, and the best one to process the query using the stored view. The plan that should be used is the one with least expected cost.

Another optimization issue arises due to the work presented in chapter 3 on staticly optimized view maintenance algorithms (RVM and SAVM). Using a staticly optimized view maintenance algorithm, when given a collection of views, an efficient structure must be found for maintaining those views. In the case of RVM this structure is a Rete network, and in the case of SAVM the structure is a shared execution plan. Current expert system shells that use a Rete network for testing rule conditions [For81,For82,Sho87] use heuristics to construct the network. This appears satisfactory in an environment where the data base and rule base are small (e.g. at most a few thousand facts and rules) and will thus fit entirely into main memory. In a database environment where the number of facts and rules may be far greater, the optimization issue is much more important. Hence, an interesting area for future research is to find algorithms for constructing optimized Rete nets and shared execution plans for RVM and SAVM, respectively.

Besides the issues of optimization, there are transaction processing questions that must be addressed if support for materialized objects is added to a DBMS. Future research is required on concurrency control and recovery in database systems enhanced with rules and derived objects.

It appears that all the pieces are finally available for building a highly general yet efficient database rules system. A high-priority area research area is the construction of a working rules system in a DBMS (this is being undertaken as part of the POSTGRES project [StR86,SHH87]). It is important to prototype some large database rule applications. The knowledge gained through this endeavor will be invaluable to researchers attempting refine the rules system. It will also provide a sound basis for making decisions about the design of future database rule processing facilities.

APPENDIX 1

Implementation of T-locks in B-Tree Indexes

All lock-based rule indexing algorithms require the ability to set t-locks on ranges of data in a B-tree index. For predicate terms of the form *attribute* = *constant*, a t-lock is set in the index by inserting a dummy record for the value *constant*. The dummy record has fields to hold *constant*, a tag indicating that the lock is for an equality condition, and the indentifier of the rule that set the lock.

The problem of locking ranges of values is slightly more complex. Predicate terms specifying ranges can be of the following form:

$$constant_1 \text{ lower\_op } attribute \text{ upper\_op } constant_2$$

The symbols lower_op and upper_op are either $<$ or $\leq$, and either or both of the constants can be positive or negative infinity. Allowing the constants to be infinity handles the special case of open-ended ranges such as *attribute* $<$ *constant*. Recall that the structure of a B-tree index page is the following:

$$ptr_0, key_1, ptr_1, \cdots, key_m, ptr_m$$

Everything in a subtree pointed to by ptr_i has a key value that is $\geq key_i$ and $< key_{i+1}$. Intervals are locked in the following way. The structure of an interval t-lock set by a rule with identifier RuleID is

$$t = [\text{RuleID, RuleType, } constant_1, \text{ lower\_op, } constant_2, \text{ upper\_op}]$$

Locks are set in the tree using the following recursive algorithm, which is initially called on the root page of the tree:

```
LockRange(Page, t-lock)
{
if Page is a leaf or the lock range covers all values on Page then
  put a copy of t-lock on the page and return
else
  for each ptr_i on Page such that the interval [key_i, key_{i+1}) overlaps the lock range do
    LockRange(↑ptr_i, t-lock)
  end
end if
}
```

When a new value k is inserted into a B-tree containing interval locks the insertion algorithm must determine the set of locks that conflict with the new value. This is done by executing the following section of code for each page visited on the trip from the root to the leaf (ConflictSet is initially empty):

```
if page is an internal node then
    ConflictSet := ConflictSet ⋃ { locks on page }
else /* this is a leaf page */
    ConflictSet := ConflictSet
            ⋃ { T | T is a t-lock with a range that covers k }
end if
```

After the insertion, ConflictSet contains the set of locks that conflict with k.

The question arises as to what action to take when a B-tree page must be split or merged. In the case of merges, the set of locks on the two pages must simply be unioned together to form the set of locks for the new page. For splits, if the page being split is a leaf node, each new page contains all the locks from the old page that overlap some value on the new page. If the page being split is an internal node, the lock set from the original page is copied to both of the new pages.

APPENDIX 2

This appendix describes the new aggregate syntax proposed for the POSTQUEL language [StR86] and discusses how POSTQUEL aggregates are processed. A comparison of the old QUEL aggregate notation [Eps79,HSW75] with the new POSTQUEL syntax is then given.

New Aggregate Syntax for POSTQUEL

The goal of the POSTQUEL aggregate notation is to make all linkage between the inner and outer part of the query explicit, thus simplifying specification of complex aggregate queries. The general form of a POSTQUEL aggregate is as follows:

aggregate_name '{' *exp* [**from** *from_list*] [**where** *qual*] '}'

Using this syntax, all tuple variables appearing in the query are global unless they are re-defined in the **from** clause of an aggregate. Scalar aggregates can be specified in much the same way as in QUEL. For example, the following POSTQUEL query retrieves the average salary of employees working in the Sales department:

retrieve (a = avg{EMP.salary **where** EMP.dept = "Sales"})

In general, a query containing an aggregate function consists of an *outer block* and one or more nested *inner blocks* (inner blocks may in turn contain other nested blocks). The semantics of a nested query of this form are that it appears to be processed using the following algorithm known as *nested iteration:*

> For each tuple t that meets the qualification of the outer block, substitute t into the inner block(s) and evaluate them. Form a modified query by replacing the inner block(s) with the values they return. Evaluate the condition of this query, and if it is true, add t to the result.

Processing POSTQUEL Aggregates

As an example, a query to retrieve all department records for departments with an average employee salary of greater than 15,000 dollars can be specified as follows in POSTQUEL:

Q_1:

```
retrieve (DEPT.all)
where avg {EMP.salary where EMP.dept=DEPT.dname} > 15000
```

Using nested iteration, the outer block of Q_1 retrieves every DEPT record, and the inner block is evaluated once for each one. For example, suppose that there is a DEPT tuple with dname = "Toy". When processing Q_1 using nested iteration, the following subquery is formed:

```
retrieve (dname = "Toy", ...)
where avg {EMP.salary where EMP.dept="Toy"} > 15000
```

Then the avg aggregate is evaluated. Suppose that the result is that the average Toy department salary is 13,000 dollars. Then the following subquery would be formed:

```
retrieve(dname = "Toy", ...)
where 13000 > 15000
```

Since 13,000 is not greater than 15,000, no tuple is retrieved, so the Toy department record is not part of the result.

Although the nested iteration algorithm is useful for defining the semantics of aggregation queries, it is often not the most efficient way to process aggregates. For example, suppose that there are 20 records for each unique dname value in the DEPT relation (e.g. 20 Toy department records, 20 Sales department recores, etc.). In this case, using nested iteration, the inner block would be evaluated 20 times for each dname – a tremendous waste of effort!

The SQL query language [CAE76] also provides aggregate queries with nested iteration semantics. Previous research has shown how to translate nested aggregate queries written in SQL into a form that can be processed much more efficiently [GaW87,Kie84,Kim82]. These techniques

are directly applicable to processing nested aggregation queries in POSTQUEL. For brevity, these techniques are not reviewed here; the reader is referred to the original sources for a complete discussion. The principle behind the techniques is illustrated below using an example. Consider the query Q_1 given previously. Q_1 can be transformed into the following query which makes use of an explicit by clause rather than nested iteration to express grouping:

Q_1':

```
retrieve (DEPT.all)
where avg(EMP.salary by DEPT.dname
          where DEPT.dname=EMP.dept) > 15000
```

This syntax is not legal for user queries in POSTQUEL, however the system may construct queries with this structure for purposes of optimization *. Q_1' can be processed using the following steps, as described in [Eps79]:

1. Initialize a relation TEMP to hold the result of the aggregate function as follows:

```
retrieve unique into TEMP(count=0, avg=0, dname=DEPT.dname)
```

2. Run the following query (without removing duplicates), and for each tuple retrieved, update the count and avg fields of the appropriate tuple in TEMP:

```
retrieve (EMP.salary, DEPT.dname)
where EMP.dept = DEPT.dname
```

3. Modify Q_1' to form the following query:

Q_1''

```
retrieve(DEPT.all)
where TEMP.avg > 15000
and TEMP.dname = DEPT.dname
```

4. Execute Q_1'' and return the result.

* Since aggregate queries based on nested iteration can be translated into a format using an explicit by clause, the techniques for maintaining the results of aggregation queries proposed in chapter 3 can be used with nested iteration queries as well.

The procedure outlined above can be significantly more efficient than nested iteration. For example, in the case where there are 20 DEPT records for each unique dname value, each EMP record would be accessed 20 times using nested iteration, but only once using the above algorithm.

Comparison of QUEL and POSTQUEL Aggregate Notation

The QUEL aggregate function syntax has proven difficult for users to master. A major problem with the QUEL notation is that the result relation of an aggregate function is implicitly linked to the outer part of the query. In order to successfully specify specify a non-trivial query containing an aggregate function, the user must understand the fairly complex algorithm used to perform the linkage. As an example, consider the following schema for a database containing information about authors and the books they have written or co-written:

```
author(aid, name, ...)  /* unique authors */
book(bid, title, ...)   /* unique books */
ab(aid, bid)            /* relationship showing authorship of a book */
```

Consider a query to find all pairs of authors that have co-authored more than three books together. This query can be specified as follows using QUEL:

```
range of a, a2 is author
range of b is book
range of ab1, ab2 is ab
retrieve(a.all,a2.all)
where count(b.bid by a.aid, a2.aid
           where a.aid=ab1.aid and ab1.bid = b.bid
           and a2.aid = ab2.aid and ab2.bid = b.bid
           and a.aid != a2.aid) >= 3
```

The tuple variables a and a2 in the target list of this command are *not* the same as a and a2 in the aggregate function (this is an example of the source of confusion using QUEL aggregates). The above query is processed in INGRES as follows. The aggregate function in the query is computed to form the following temporary relation showing pairs of authors, and the number of books they have co-authored.

```
TEMP(count, aid1, aid2)
```

The original query is then modified to form the following:

```
retrieve(a.all,a2.all)
where TEMP.count >= 3
and TEMP.aid1 = a.aid
and TEMP.aid2 = a2.aid
```

The result of this modified query is returned to the user.

The same query can be specified as follows using POSTQUEL:

```
retrieve(a.all,a2.all)
from a, a2 in author, b in book, ab1, ab2 in ab
where count {b.bid where a.aid = ab1.aid and ab1.bid = b.bid
            and a2.aid = ab2.aid and ab2.bid = b.bid
            and a.aid != a2.aid} >= 3
```

Here, a and a2 refer to the same tuple in both the target list and the aggregate. The meaning of this query is more clear from the text of the query itself than it is in the QUEL example because linkage between the aggregate and the outer part of the query is specified explicitly using a and a2.

References

[AdL80] Adiba, M. E. and Lindsay, B. G., "Database Snapshots", *Proceedings of the International Conference on Very Large Data Bases*, October 1980, 86-91.

[AgD83] Agrawal, R. and DeWitt, D. J., "Updating Hypothetical Data Bases", *Information Processing Letters 16* (April 1983), 145-146, North Holland .

[ASU79] Aho, A. V., Sagiv, Y. and Ullman, J. D., "Efficient Optimization of a Class of Relational Expressions", *ACM Transactions on Database Systems 4*, 4 (1979), 435-454.

[ABC76] Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W. and Watson, V., "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems 1*, 2 (June 1976), 97-137.

[BBD82] Bates, D., Boral, H. and DeWitt, D. J., "A Framework for Research in Database Management for Statistical Analysis", *Proceedings of the 1982 ACM-SIGMOD Conference on Management of Data*, June 1982.

[BLT86] Blakeley, J. A., Larson, P. and Tompa, F. W., "Efficiently Updating Materialized Views", *Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data*, Washington DC, May 1986, 61-71.

[Blo70] Bloom, B. H., "Space/Time Trade-offs in Hash Coding with Allowable Errors", *CACM 13*, 7 (July 1970).

[BoW77] Bobrow, D. and Winograd, T., "An Overview of KRL, a Knowledge Representation Language", *Cognitive Science 1*, 1 (Jan.-Mar. 1977).

[Bra86] Bratko, PROLOG Programming for Artificial Intelligence, 1986.

[BuC79] Buneman, O. P. and Clemons, E. K., "Efficiently Monitoring Relational Databases", *ACM Transactions on Database Systems 4*, 3 (September 1979), 368-382.

[But86] Butler, M., "An Approach to Persistent Lisp Objects", *Proceedings of the Thirtieth Computer Society International Conference*, San Francisco, CA, March 1986.

[Car75] Cardenas, A. F., "Analysis and Performance of Inverted Data Base Structures", *CACM 18*, 5 (May 1975), 253-263.

[CAE76] Chamberlin, D. D., Astrahan, M. M., Eswaran, K. P., Griffiths, P. P., Lorie, R. A., Mehl, J. W., Reisner, P. and Wade, B. W., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", *IBM Journal of Research and Development 20*, 6 (1976).

[Cha82] Chang, S. K., "Database Alerters for Knowledge Management", *Proceedings of the Workshop on Self-Describing Data Structures*, Univ. of Maryland, College Park, October 1982.

[Cod70] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", *CACM 13*, 6 (June 1970), 377-387.

[Dat81a] Date, C. J., "Referential Integrity", *Proceedings of the 7th VLDB Conference*, Cannes France, September 1981.

[Dat81b] Date, C. J., *An Introduction to Database Systems*, Addison Wesley, 1981.

[DeF86] Deering, M. and Faletti, J., "Database Support for Storage of AI Reasoning Knowledge", in *Expert Database Systems/Proceedings From the First International Workshop*, L. Kerschberg (editor), 1986, Benjamin/Cummings.

[End72] Enderton, H. B., A Mathamatical Introduction to Logic, 1972.

[Eps79] Epstein, R., "Techniques for Processing of Aggregates in Relational Database Systems", UCB/ERL M79/8, University of California, February 1979.

[EGL76] Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database System", *CACM 19*, 11 (November, 1976).

[Esw76] Eswaran, K. P., "Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System", IBM Research Report RJ1820(26414), IBM Research Laboratory, San Jose, CA, August 1976.

[FiK85] Fikes, R. and Kehler, T., "The Role of Frame-Based Representation and Reasoning", *CACM 28*, 9 (September 1985).

[For81] Forgy, C. L., "OPS5 User's Manual", CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA 15213, July 1981.

[For82] Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", *Artificial Intelligence 19* (1982), 17-37, North Holland.

[For84] Forgy, C. L., "OPS83 Report", CMU-CS-84-133, Carnegie-Mellon University, Pittsburgh, PA 15213, May 1984.

[FWA86] Fox, M. S., Wright, J. M. and Adam, D., "Experiences with SRL: An Analysis of a Frame-based Knowledge Representation", in *Expert Database Systems/Proceedings From the First International Workshop*, L. Kerschberg (editor), 1986, Benjamin/Cummings.

[GaM78] H. Gallaire and J. Minker, eds., *Logic and Data Bases*, Plenum Press, New York, NY , 1978.

[GMN81] H. Gallaire, J. Minker and J. M. Nicolas, eds., *Advances in Data Base Theory* , Plenum Press, New York, NY , 1981.

[GaW87] Ganski, R. A. and Wong, H. K. T., "Optimization of Nested SQL Queries Revisited", *Proceedings of the 1987 ACM-SIGMOD Conference on Mangagement of Data*, San Francisco CA, May 1987.

[Gev87] Gevarter, W. B., "The Nature and Evaluation of Commercial Expert System Building Tools", *IEEE Computer*, May 1987.

[Gra78] Gray, J. N., "Notes on Data Base Operating Systems", IBM Research Report RJ2254, IBM Research Laboratory, San Jose, CA, August 1978.

[GuF83] Gupta, A. and Forgy, C. L., "Measurements on Production Systems", CMU-CS-83-167, December 1983.

[Gut84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, June 1984.

[HaC76] Hammer, M. and Chan, I., "Index Selection in a Self-Adaptive Data Base System", *Proceedings of the 1976 ACM-SIGMOD Conference on Management of Data*, Washington DC, June 1976.

[Han84] Hanson, E. N., "User-Defined Aggregates in the Relational Database System INGRES", Masters Report, University of Califoria, Berkeley CA, December 1984.

[Han87] Hanson, E. N., "A Performance Analysis of View Materialization Strategies", *Proceedings of the 1987 ACM-SIGMOD Conference on Management of Data*, San Francisco CA, May 1987.

[HSW75] Held, G., Stonebraker, M. and Wong, E., "INGRES – A Relational Database System", *Proc. of the National Computer Conference*, 1975.

[Hil86] Hill, M., "et al., "Design Decisions in SPUR", *IEEE Computer*, November 1986.

[HoT86] Horwitz, S. and Teitelbaum, T., "Generating Editing Environments Based on Relations and Attributes", *ACM Transactions on Programming Languages and Systems 8* , 4 (October 1986), 577-608.

[How86] Howe, L., "Sybase Data Integrity For On-Line Applications", Sybase, Inc. 2910 Seventh Street, Berkeley California 94710, 1986.

[Ioa85] Ioannidis, Y., "A Time Bound on the Materialization of Some Recursively Defined Views", *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, August 1985.

[Ioa86] Ioannidis, Y., "Enhancing INGRES with Deductive Power", in *Expert Database Systems/Proceedings From the First International Workshop*, L. Kerschberg (editor), 1986, Benjamin/Cummings.

[JCV84] Jarke, M., Clifford, J. and Vassiliou, Y., "An Optimizing Prolog Front-End to a Relational Query System", *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, June 1984.

[Jar86] Jarke, M., "External Semantic Query Simplification: A Graph-Theoretic Approach and its Implementation in PROLOG", in *Expert Database Systems/Proceedings From the First International Workshop*, L. Kerschberg (editor), 1986, Benjamin/Cummings.

[Kie84] Kiessling, W., "SQL-Like and QUEL-Like Correleation Queries With Aggregates Revisited", UCB/ERL M84/75, Univ. California, Berkeley, September 1984.

[Kim82] Kim, W., "On Optimizing an SQL-like Nested Query", *ACM Transactions on Database Systems 4*, 4 (September 1982), 443-469.

[KoP81] Koenig, S. and Paige, R., "A Transformational Framework for the Automatic Control of Derived Data", *Proceedings of the 7th International conference on Very Large*

Data Bases, France, 1981, 306-318.

[LHM86] Lindsay, B. G., Haas, L., Mohan, C., Pirahesh, H. and Wilms, P., "A Snapshot Differential Refresh Algorithm", *Proceedings of the 1986 ACM-SIGMOD International Conference on Management of Data*, June 1986, 53-60.

[Min75] Minsky, M., "A Framework for Representing Knowledge", in *The Psychology of Computer Vision*, P. Winston (editor), New York, NY, 1975, McGraw Hill.

[Mis84] Mishkin, N., "Managing Permanent Objects", PhD Thesis, Department of Computer Science, Yale University, New Haven, Connecticut, 1984.

[Pai80] Paige, R., "An Efficient Implementation of Automatic Finite Differencing", Department of Computer Science, Rutgers University, August 1980.

[RoG77] Roberts, I. and Goldstein, R., "NUDGE, A Knowledge-Based Scheduling Program", *Proceedings of the 5th International Joint Conference on Artificial Intelligence* , Cambridge, MA, August 1977, 257-263.

[RoK86] Roussopoulos, N. and Kang, H., "Principles and Techniques in the Design of ADMS±", *Computer*, December 1986.

[RoS79] Rowe, L. A. and Shoens, K. A., "Data Abstraction, Views and Updates in RIGEL", *Proceedings of the 1979 ACM-SIGMOD International Conference on Management of Data*, Boston Massachusetts, June 1979.

[RoS87] Rowe, L. A. and Stonebraker, M. R., "The POSTGRES Data Model", *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton England, August 1987.

[Rub86] Rubenstein, W. B., "A-Trees: An Indexing Abstraction for Ordered Aggregates", U.C. Berkeley Memo No. UCB/ERL/M86/77, 12 September 1986.

[RKC87] Rubenstein, W. B., Kubicar, M. S. and Cattell, R. G. G., "Benchmarking Simple Database Operations", *Proceedings of the 1987 ACM-SIGMOD Confernence on Management of Data*, San Francisco, May 1987.

[Rub87] Rubenstein, W. B., "Data Management of Musical Information", PhD Dissertation, Dept. of Computer Science, U.C. Berkeley, Berkeley CA, June 1987.

[ScW86] Sciore, E. and Warren, D. S., "Towards an Integrated Database-PROLOG System", in *Expert Database Systems/Proceedings From the First International Workshop*, L. Kerschberg (editor), 1986, Benjamin/Cummings.

[Sel79] Selinger, P., "et al., "Access Path Selection in a Relational Database Management System", *Proceedings of the 1979 ACM-SIGMOD International Conference on Management of Data*, Boston, MA, June 1979.

[Sel86a] Sellis, T., "Global Query Optimization", *Proceedings of the 1986 ACM-SIGMOD International Conference on Management of Data 15*, 2 (June 1986), 191-205.

[Sel86b] Sellis, T., "Optimization of Extended Relational Database Systems", PhD Thesis, University of California, Dept of EECS, Berkeley CA, 1986.

[Sel87] Sellis, T. K., "Efficiently Supporting Procedures in Relational Database Systems", *Proceedings of the 1987 ACM-SIGMOD Conference on Mangagement of Data*, San Francisco CA, May 1987.

[SeL76] Severance, D. and Lohman, G., "Differential Files: Their Application to the Maintenance of Large Databases", *ACM Transactions on Database Systems 1*, 3 (September 1976), 256-267.

[ShI84] Shmueli, O. and Itai, A., "Maintenance of Views", *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, June 1984.

[ShB75] Shortliffe, E. H. and Buchanan, B. G., "A Model of Inexact Reasoning in Medicine", *Mathematical Biosciences 23* (1975), 251-379.

[Sho76] Shortliffe, E. H., *Computer-Based Medical Consultations: MYCIN*, American Elsevier, New York, 1976.

[Sho87] Shoup, A., personal communication, Inference Corporation, San Francisco, CA, 1987.

[Sto75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification", *Proceedings of the 1975 ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, June 1975.

[SWK76] Stonebraker, M., Wong, E., Kreps, P. and Held, G., "The Design and Implementation of INGRES", *ACM Transactions on Database Systems 1*, 3 (September 1976), 189-222.

[SAH84] Stonebraker, M., Anderson, E., Hanson, E. and Rubenstein, B., "QUEL as a Data Type", *Proceedings of the 1984 ACM-SIGMOD International Conference on Management of Data*, Boston, MA, June 1984.

[Sto85] Stonebraker, M., "Triggers and Inference in Data Base Systems", *Proceedings of the Islamorada Expert Database Conference*, February 1985.

[SAH85] Stonebraker, M., Anton, J. and Hanson, E., "Extending a Data Base System with Procedures", *(to appear in ACM Transactions on Database Systems, September 1987)*, Berkeley, CA, July 1985.

[SSH86] Stonebraker, M., Sellis, T. and Hanson, E., "An Analysis of Rule Indexing Implementations in Data Base Systems", *Proceedings of the First Annual Conference on Expert Database Systems*, Charleston SC, April 1986.

[StR86] Stonebraker, M. and Rowe, L., "The Design of POSTGRES", Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data, 1986.

[SHH87] Stonebraker, M., Hanson, E. and Hong, C., "The Design of the POSTGRES Rules System", *Proc. 1987 IEEE Data Engineering Conference*, Los Angeles California, February 1987.

[SHP87] Stonebraker, M., Hanson, E. and Potamianos, S., "A Rule Manager for Relational Database Systems", *IEEE Transactions on Software Engineering*, 20 May 1987.

[Ull85] Ullman, J., "Implementation of Logical Query Languages for Data Bases", *Proceedings of the 1985 ACM-SIGMOD International Conference on Management of Data*, Austin, TX, May 1985.

[WoK80] Wong, E. and Katz, R. H., "Logical Design and Schema Conversion for Relational and DBTG Databases", in *Entity-Relationship Approach to Systems Analysis and Design*, North Holland Publishing Co., 1980, Amsterdam.

[WoS83] Woodfill, J. and Stonebraker, M., "An Implementation of Hypothetical Relations", *Proceedings of the Ninth Very Large Data Base Conference*, Florence, Italy, December 1983.

[Yao77] Yao, S. B., "Approximating Block Accesses in Database Organizations", *CACM 20*, 4 (April 1977).

[Yao78] Yao, A. C., "On Random 2-3 Trees", *Acta Informatica 9*, 2 (1978).

[Zan85] Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects", *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, August 1985.

[Zan86] Zaniolo, C., "Safety and Compilation of Non-Recursive Horn Clauses", *Proceedings of the First International Conference on Expert Database Systems*, Charleston, SC,

April 1986.