PROCESSING QUERIES AGAINST DATABASE PROCEDURES: A PERFORMANCE ANALYSIS

by

Eric N. Hanson

Memorandum No. UCB/ERL M87/68

1 September 1987

PROCESSING QUERIES AGAINST DATABASE PROCEDURES: A PERFORMANCE ANALYSIS

by

Eric N. Hanson

Memorandum No. UCB/ERL M87/68

1 September 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Processing Queries Against Database Procedures: A Performance Analysis

Eric N. Hanson

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Abstract

A database procedure is a collection of queries stored in the database. Several methods are possible to process a query that retrieves the value returned by a database procedure. The conventional algorithm is to execute the queries in the procedure whenever it is accessed. A second strategy requires caching the previous value returned by the database procedure. To process a query, if the cached value has been invalidated by an update, the value is recomputed and stored back into the cache. Otherwise, the stored procedure result is still valid, so it is returned. A third strategy is possible, in which a differential view maintenance algorithm is employed to maintain at all times an up-to-date copy of the value returned by the procedure. This paper compares the performance of these three alternatives. The results show that the choice of the preferred algorithm depends heavily on the database environment, particularly the frequency of updates and the size of objects retrieved by database procedures.

1. Introduction

Extensions to relational database systems have been proposed to allow database commands as well as data to be stored in the database [SAH84,SAH85]. A collection of query language statements stored in a field of a record is known as a *database procedure*. As described in [SAH85], database procedures can provide support for several desirable features, including (1) stored queries, (2) objects with unpredictable composition, (3) complex objects with shared subobjects (e.g. a form with trim, labels and icons), (4) referential integrity [Dat81], and (5) aggregation and generalization [SmS77].

Several different algorithms are possible for processing queries that retrieve the value computed by a database procedure. The conventional method, which will be called *Always Recompute*, is to compute the result of a database procedure from the base relations whenever the procedure is accessed. This strategy has been implemented in a version of INGRES enhanced with database procedures [SAH85]. Another scheme which is based on saving previous values returned by the database procedure will be called *Cache and Invalidate*. In Cache and Invalidate, when the procedure is accessed, if a valid result

Sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089.

for it is in the cache, it is used. Otherwise, the procedure result is recomputed and stored to refresh the cache. If an update command occurs that changes the value of the procedure result, the currently cached result is marked invalid. This method has been proposed in [SAH85,StR86] and is also known as *caching*. Database procedures are simply collections of database queries, and queries have the same structure as database views. Thus, differential view maintenance algorithms [BLT86,RoK86] can be applied to maintain at all times an up-to-date copy of the value retrieved by each query in a database procedure. This represents a third algorithm for processing procedure queries, known as *Update Cache*. In Update Cache, queries that retrieve the result of a database procedure are processed by simply reading the value maintained in the cache.

The three strategies discussed above have different cost characteristics. Each may perform best, depending on the environment. For example, the average cost of a query that reads a procedure value will depend on the relative frequency of queries and updates, the size of objects, and other parameters. The rest of this paper presents a performance analysis comparing the costs of the Always Recompute, Cache and Invalidate, and Update Cache algorithms for processing queries against database procedures*. The paper is organized as follows. Section 2 describes the three algorithms in more detail (in particular, it presents two versions of the Update Cache strategy based on different view maintenance algorithms). Section 3 describes the two procedure models (model 1 and model 2) that will be analyzed. Section 4 analyzes the cost of procedure maintenance assuming model 1. Section 5 presents the performance results obtained for model 1. Section 6 analyzes the cost of maintaining model 2 procedures. Section 7 gives the performance results for model 2. Finally, section 8 summarizes and presents conclusions.

2. Details of Query-Processing Algorithms for Procedures

Below we present a more detailed description of the Alway Recompute, Cache and Invalidate, and Update Cache algorithms. Always Recompute is a straightforward extension of normal query processing. It is assumed in this paper that when using Always Recompute, an optimized execution plan for the query(s) in the procedure is compiled in advance and stored with the procedure. This plan (or collection of plans) is executed when the result of the procedure is retrieved. There is no compilation overhead at run time.

Using Cache and Invalidate, a precompiled execution plan is stored with the procedure just as in Always Recompute, so there is no run-time compilation overhead. As described previously, Cache and Invalidate recomputes procedure results only if a valid result is not available in the cache. A reliable mechanism is required to invalidate cache entries that have been made invalid by a database update. This is done using a technique called *rule indexing* [SSH86]. Using rule indexing, when the value of a database procedure is retrieved, special persistent locks called *invalidate locks* or *i*-locks are set on all data read during query processing, including any index intervals inspected. Each *i*-lock contains the identifier of the database procedure for which it was set. If an update later sets a write lock that conflicts with an *i*-lock, the cached procedure value for which that *i*-lock was set is marked invalid.

---

*A related paper compared the cost of query modification vs. use of a view maintenance algorithm for processing queries against views [Han87a]

Two different versions of the Update Cache strategy are analyzed in this paper. The first is based on the view maintenance algorithm proposed in [BLT86]. This algorithm is based on manipulations in relational algebra, and hence will be called *algebraic view maintenance* (AVM). For example, consider a view $V(A,B)$ defined using a relational algebra expression on relations A and B. Suppose a transaction updates A by appending a set of tuples a and deleting a set of tuples d. The new value of V after the transaction can be represented as follows:

$$V(A \cup a - d, B) = V(A,B) \cup V(a,B) - V(d,B)$$

The expression $V(A,B)$ does not have to be computed because it is equal to the stored copy of the view. Only $V(a,B)$ and $V(d,B)$ need to be found. This is usually much less expensive than completely recomputing V.

The second Update Cache strategy is based on a view maintenance algorithm called *Rete view maintenance* (RVM) proposed [Han87b]. The RVM algorithm is based on the *Rete network* [For82], a type of discrimination network used in production-rule system interpreters including OPS5 [For81], OPS83 [For84] and ART [Gev87,Sho87]. Using a Rete network, after an update transaction, a collection of *tokens* are created to represent the changes to the database made by the transaction. Inserted and deleted tuples are represented by a tokens with tags "+" and "−", respectively (modifications are treated as deletes followed by inserts). These tokens are inserted into the Rete network at a special node called the **root**, and allowed to propagate through the data structure. In general, a Rete network for maintaining views defined using relational algebra can be built using nodes whose types and functions are described below:

• **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

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. Any 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. All tokens that arrive at an α-memory node are passed on to all successors of the node.

• **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. If a token arrives at the input of an **and** node, the memory node that forms the opposite input is searched to see if there are any tuples that join with the token. A new token is formed for each [token,tuple] pair that meets the join qualification associated with the **and** node. The tokens formed have the same tag ("+" or "−") as the original token input to the **and** node. These tokens are passed on to the **and** node's successor.

• **β-memory** nodes: These nodes hold the output of **and** nodes. Otherwise, they are similar to α-memory nodes.

The Rete network can be used for view maintenance for the following reason: α and β-memory nodes are equivalent to views. The contents of a memory node m in the network is equal to the current value of the view whose qualification is represented by the network nodes that are ancestors of m [Han87b].

Consider as an example the following schema and pair of views:

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

/* all programmers who work on the first floor */

define view PROGS1 (EMP.all, DEPT.all)
where EMP.dept = DEPT.dname
and EMP.job = "Programmer"
and DEPT.floor = 1

/* all clerks who work on the first floor */

define view CLERKS1 (EMP.all, DEPT.all)
where EMP.dept = DEPT.dname
and EMP.job = "Clerk"
and DEPT.floor = 1
```

A Rete network for maintaining materialized copies of these two views is shown in figure 1. The two β-memory nodes at the bottom of the diagram contain the views PROGS1 and CLERKS1, respectively. As an example of how the Rete network is used to maintain views, suppose that the following tuple t is added to the relation EMP:

t = <name="Susan", age=28, dept="Accounting", salary=30K, job="Programmer">

This insertion will cause a token $T = [+,t]$ to be deposited at the root of the Rete network. Suppose that first T is passed to the t-const node with condition "relation = DEPT." Since T is from the relation EMP, it will not meet this condition, and will be discarded. T will then be passed to the node marked "relation = EMP," and it will meet that qualification and be propagated onward. It will fail the qualification "job = Clerk," but will meet the qualification "job = Programmer." Hence, it will be inserted into the α-memory node below the node labeled "job = Programmer," and passed to the succeeding **and** node. The opposite α-memory will then be checked to see if there is a joining tuple. Assuming that there is a tuple in that memory node with the value

<dname = "Accounting", floor = 1>

a new token T' with the following value will be formed:

T' =

[+, <name="Susan", age=28, dept="Accounting", salary=30K, job="Programmer">;
<dname = "Accounting", floor = 1>]

T' will then be added to the β-memory corresponding to the view PROGS1.

The Rete network shown contains a shared subexpression for the predicate term "DEPT.floor = 1." Rete networks take advantage of this type of sharing whenever possible. Because of the possibility of sharing subexpressions in the Rete network, RVM is called a *shared* view maintenance algorithm. In contrast, no shared subexpression

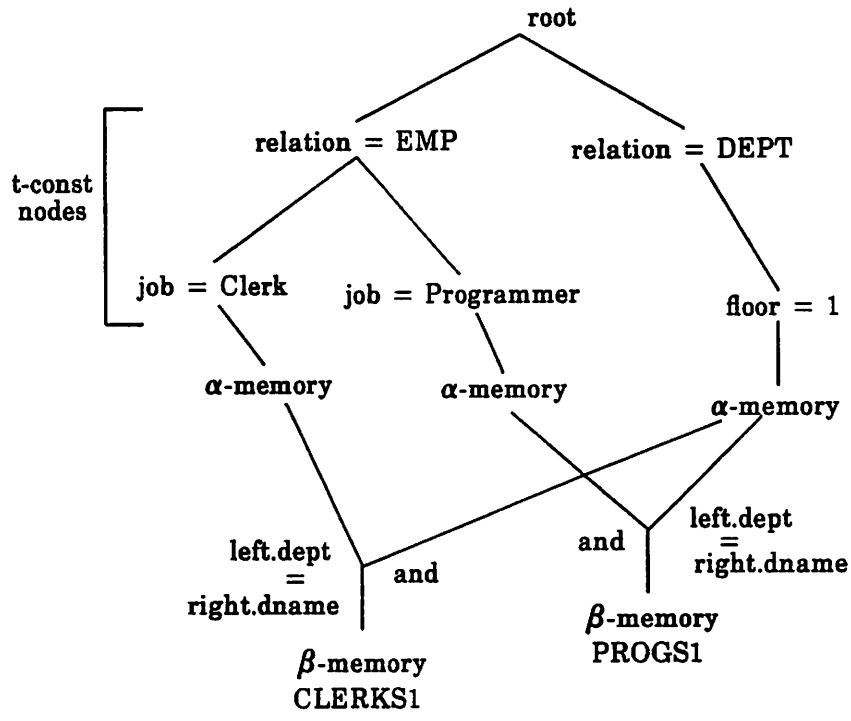

Figure 1. Example Rete network

elimination techniques are used in the version of AVM analyzed in this paper, which is a *non-shared* algorithm.

In both view maintenance algorithms considered here, execution plans for maintaining views (i.e. Rete networks and query plans for evaluating relational algebra expressions) are compiled in advance. These algorithms are called *staticly optimized* because all optimization overhead is paid only once when the execution plan is built; no optimization cost is incurred at run time [Han87b]. A *dynamicly optimized,* version of AVM exists which finds execution plans for evaluating expressions at run time [BLT86]. The advantage of static optimization is the low planning overhead. However, the disadvantage is that the execution plan for maintaining views may not always be optimal.

We now turn to the performance evaluation of the view maintenance algorithms described above. The performance model for database procedures is described in the next section.

3. Database Procedure Models Analyzed

Two different models for the structure of procedures will be evaluated. 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. This 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 use conventional write-ahead log recovery and log the identifiers of invalidated procedures [Gra78]. If the data structure is checkpointed periodicly, 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, and their defalt values, is shown in figure 2. Unless stated otherwise, the parameters will have the values shown. Using the default value of f, type P_1 procedures contain $fN = 100$ tuples. Type P_2 procedures contain $f^*N = 10$ 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

4. Cost Analysis for Model 1 Procedures

4.1. 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, which is defined as follows:

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

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

parameter	*definition*
N	number of tuples in relation R_1
S	bytes per tuple
B	bytes per block
b	total blocks ($b = NS/B$)
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 procedure queried
u	number of tuples updated between queries ($u = kl/q$)
P	probability that a given operation is an update ($P = k/(k+q)$)
f	selectivity factor of predicate term C_f
f_2	selectivity factor of predicate term C_{f_2}
f_{R_2}	size of R_2 as a fraction of N
f_{R_3}	size of R_3 as a fraction of N
C_1	CPU cost in ms to screen a record against a predicate
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 AVM
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)
C_{inval}	cost to record the invalidation of a cached procedure value

N	100,000	f	.001
S	100	f_2	.1
B	4,000	f_{R_2}	.1
k	100	f_{R_3}	.1
l	25	C_1	1
q	100	C_2	30
d	20	C_3	1
SF	.5	C_{inval}	0

Figure 2. Procedure query cost parameters and default values

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

$C_{queryP2}$ 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, which is found using the following formula based on the Yao function $y(n,m,k)$ described in Appendix A (the Yao function gives the expected number of pages touched when k records are accessed in a file containing n records on m pages).

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

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

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

4.2. 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)

This gives 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

$$C_{WriteCache} = 2C_2ProcSize$$

This gives the following value for T_1:

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

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

$$T_2 = C_2ProcSize$$

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

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

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

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

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

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

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.

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

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

4.3. 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. This 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	$C_{refreshP1}$	$N_1C_2 2Y_3$
refresh procedures of type P_2	$C_{refreshP2}$	$N_2C_2 2Y_4$
maintain A_1,D_1 sets	$C_{overhead}$	$C_3 2fl(N_1+N_2)$
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:

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

4.4. 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 3. 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

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

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 to do this 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

$$f^{**} = f_2 f_{R_2}$$

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

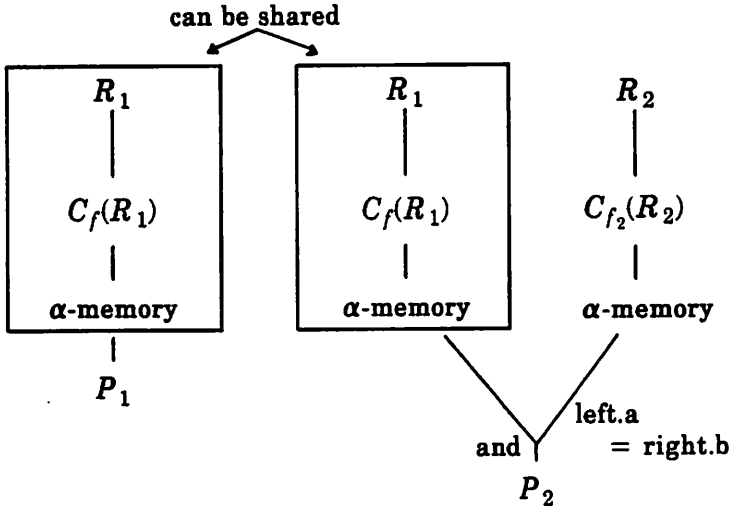

Figure 3. Rete networks for type P_1 and P_2 procedures in model 1

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

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

$$C_{join\text{-}\alpha} = N_2 C_2 Y_5$$

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	$C_{ScreenP2\text{-}Rete}$	$N_1(1-SF)C_1 f2l$
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 maintaining 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. 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 vs. the object size f (All subsequent figures appear at the end of the paper in Appendix B.)

Figure 4 shows query cost vs. 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 plots the same curves for C_{inval}=0. Figures 4 and 5 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. This could be done 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, there is a significant difference in the cost of Cache and Invalidate and Update Cache for $0<P<0.7$. This 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 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, 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 6. 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. This occurs 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 7. For this value of f, type P_1 and P_2 procedures contain 10 tuples and 1 tuple, respectively. Figure 7 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 8. 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 9 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 10 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. 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 11 compares the two 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 12 shows the regions where each algorithm performs best for different object sizes and update probabilities. The area where Cache and Invalidate wins in Figure 12 is 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 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 13, 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 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 14 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 15 shows the same information with $f_2=1$, which reduces the probability of false invalidation to zero. Cache and Invalidate performs even better for small objects in this situation.

6. 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.

6.1. 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

$$TOT_{Recompute2} = \left\lceil \frac{N_1}{N_1+N_2} \right\rceil C_{queryP1} + \left\lceil \frac{N_2}{N_1+N_2} \right\rceil C_{queryP2}'$$

6.2. 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 $C_{queryP2}'$.

6.3. 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

$$TOT_{non\text{-}shared2} = C_{read} + \frac{k}{q}(C_{screenP1} + C_{screenP2} + C_{refreshP1} + C_{refreshP2} + C_{overhead} + C_{join}')$$

6.4. 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 16. 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:

$$f_2^{**} = f_2 f_{R_3}$$

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

$$C_{join\text{-}\beta} = N_2 C_2 Y_8$$

$C_{refreshP2}$ 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 $C_{join\text{-}\alpha}$ is replaced by $C_{join\text{-}\beta}$. The total cost formula for maintaining procedures using RVM in model 2 is

$$TOT_{shared2} = C_{read} + \frac{k}{q}(C_{screenP1} + C_{ScreenP2\text{-}Rete} + C_{refreshP1} + C_{refresh\text{-}\alpha} + C_{refreshP2} + C_{join\text{-}\beta})$$

7. Performance Results for Model 2 Procedures

The performance results for Model 1 and Model 2 are similar, as can be seen by comparing Figure 17 with Figure 5. 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 18 shows the performance of the two algorithms vs. 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 19 shows the areas where each algorithm performs best for update probability vs. object size in Model 2. Figure 19 is similar to Figure 12 for Model 1, except that the best version of Update Cache is RVM instead of AVM.

8. 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 7), 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 paper, 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 paper. 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 paper 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.

As mentioned previously, a potential drawback of the staticly optimized procedure or view maintenance 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 [Sel86,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 probability 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 stategy can be added later if the programming effort can be justified. This will make it possible to efficiently maintain large stored procedure values. The view maintenance code written to implement Update Cache can also be used to provide a materialized view facility.

References

[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.

[Car75] Cardenas, A. F., "Analysis and Performance of Inverted Data Base Structures", *CACM 18*, 5 (May 1975), 253-263.

[Dat81] Date, C. J., "Referential Integrity", *Proceedings of the 7th VLDB Conference*, Cannes France, September 1981.

[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.

[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.

[Han87a] 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.

[Han87b] Hanson, E. N., "Efficient Support for Rules and Derived Objects in Relations Database Systems", PhD Thesis, University of California, Dept of EECS, Berkeley CA, 1987.

[RoK86] Roussopoulos, N. and Kang, H., "Principles and Techniques in the Design of ADMS±", *Computer*, December 1986.

[Sel86] 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.

[Sho87] Shoup, A., "", personal communication, Inference Corporation, San Francisco, CA, 1987.

[SmS77] Smith, J. and Smith, D., "Database Abstractions: Aggregation and Generalization", *ACM Transactions on Database Systems 2*, 2 (June 1977),

105-133.

[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.

[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.

[Yao77] Yao, S. B., "Approximating Block Accesses in Database Organizations", *CACM 20*, 4 (April 1977).

Appendix A

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

Appendix B

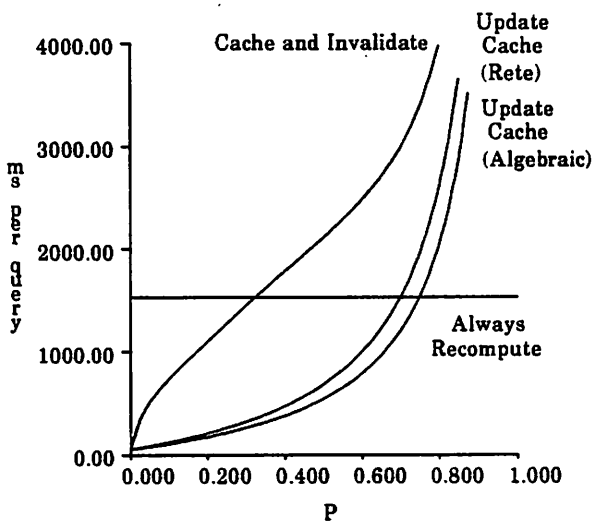

Figure 5. Query cost vs. update probability for high cache invalidation cost (60 ms)

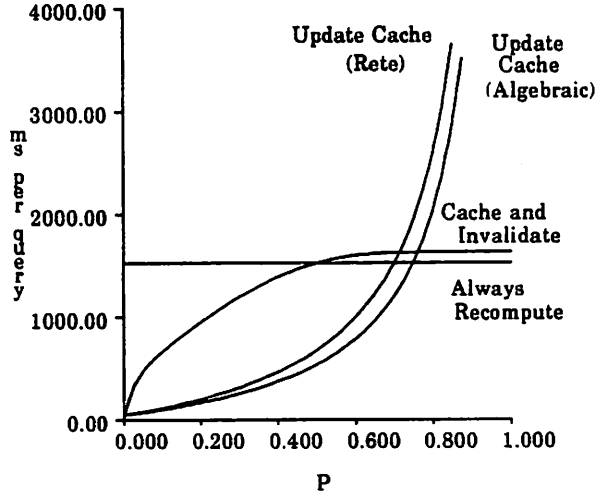

Figure 6. Query cost vs. update probability for low cache invalidation cost (0 ms)

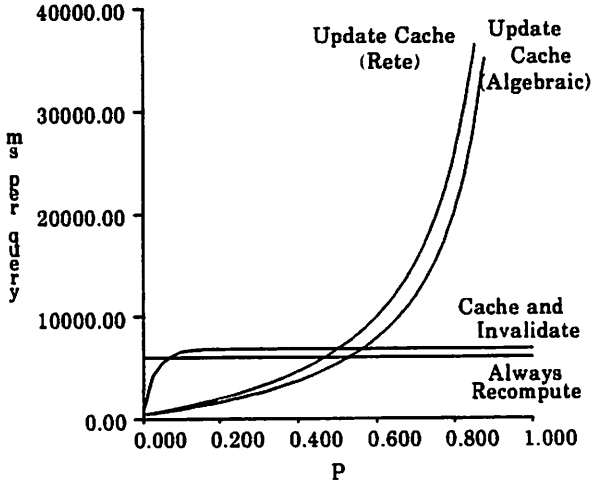

Figure 7. Query cost vs. update probability for large objects ($f = 0.01$)

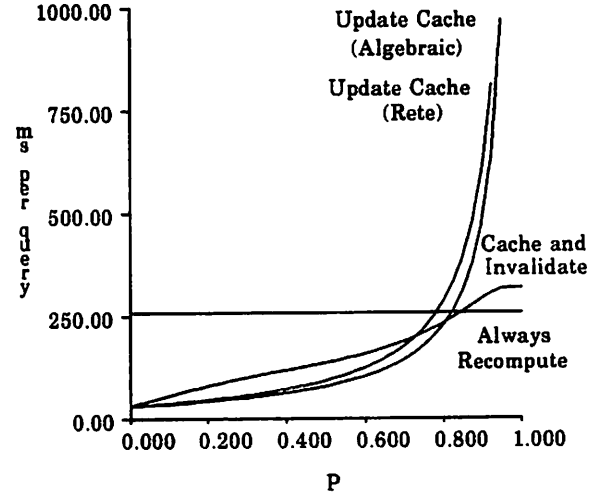

Figure 8. Query cost vs. update probability for small objects ($f = 0.0001$)

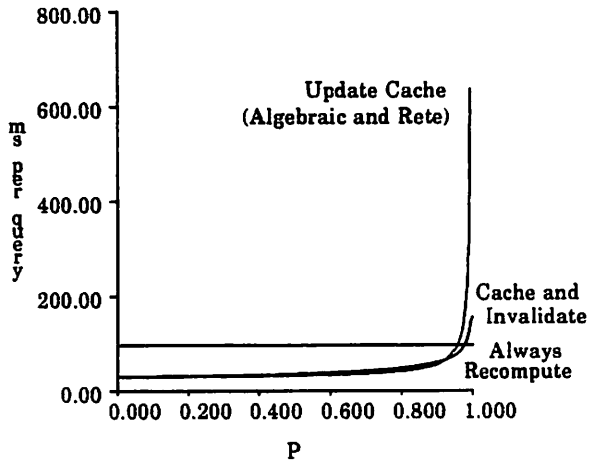

Figure 9. Query cost vs. update probability for single-tuple objects ($f = 1/N$)

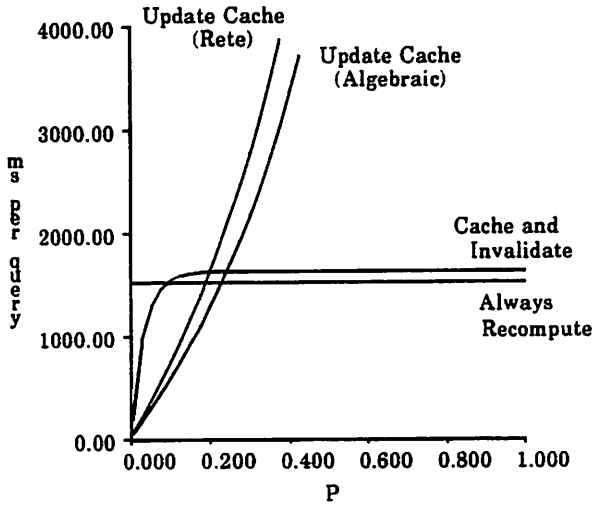

Figure 11. Query cost vs. P for large number of objects ($N_1 = N_2 = 1000$)

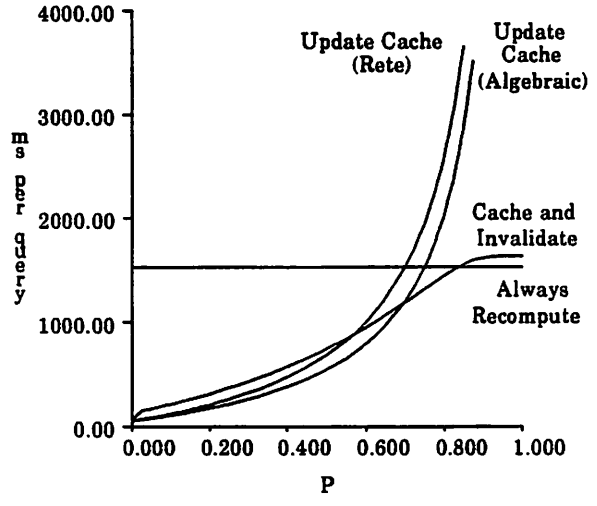

Figure 10. Query cost vs. update probability for high locality ($Z = 0.05$)

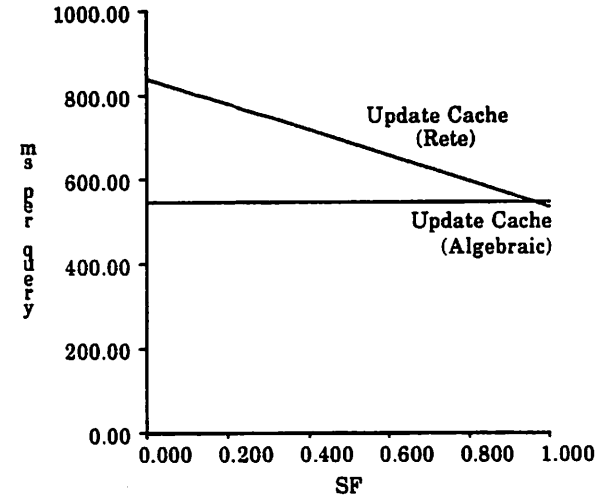

Figure 12. Query cost vs. sharing factor (SF)

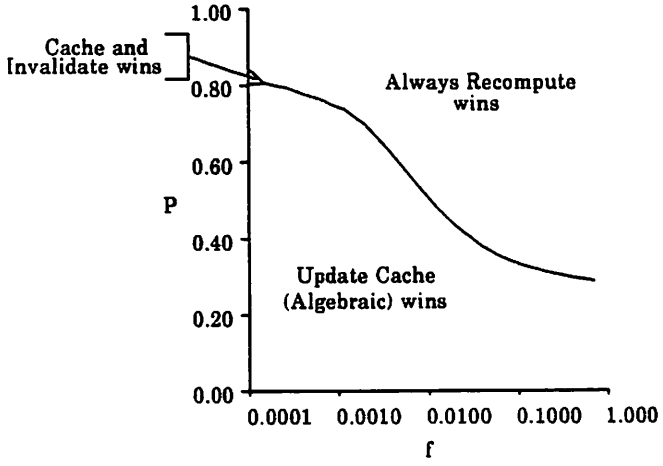

Figure 13. Areas where each method wins for object size vs. update probability

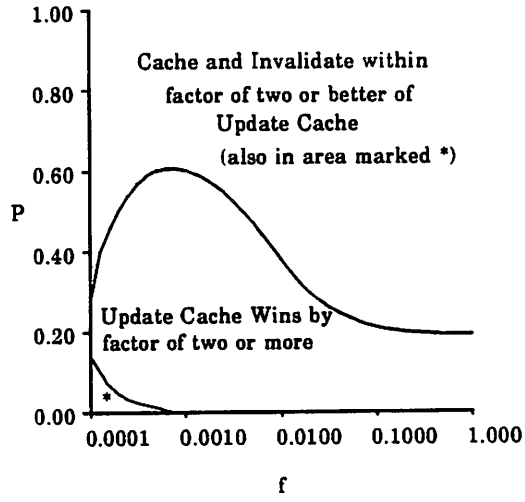

Figure 15. Measure of closeness between Cache and Invalidate and Update Cache

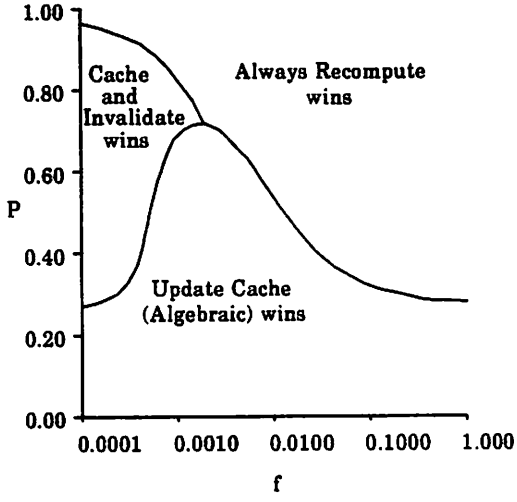

Figure 14. Areas where each method wins assuming high locality ($Z = 0.05$)

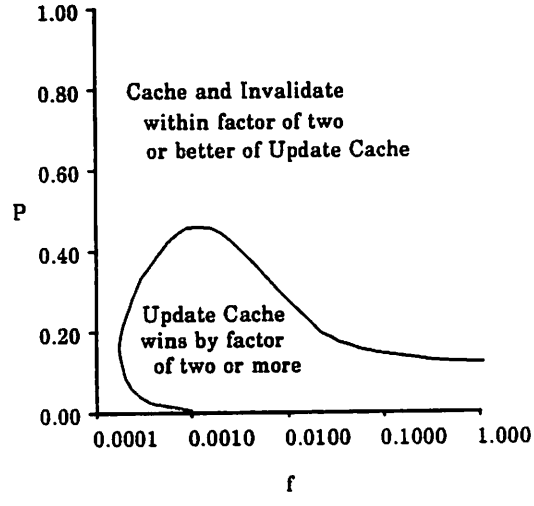

Figure 16. Measure of closeness ($f_2 = 1$)

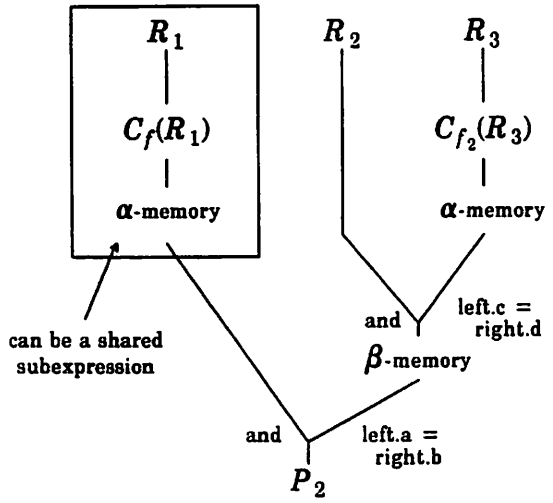

Figure 17. Model 2: Rete Network for P_2 Procedures

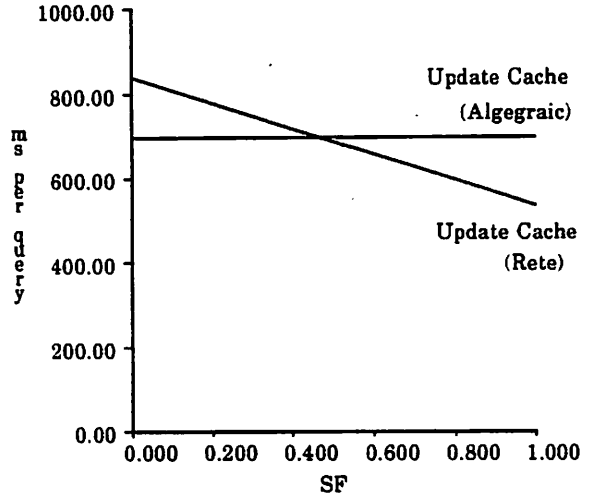

Figure 19. Model 2: Query cost of Update Cache alternatives vs. sharing factor

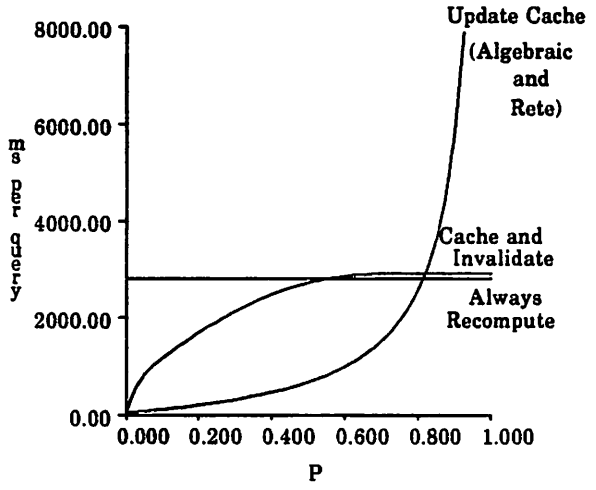

Figure 18. Model 2: Query cost for default parameters

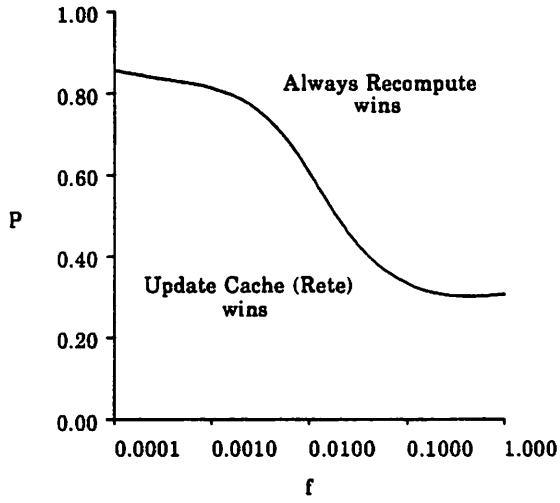

Figure 20. Model 2: Winners for update probability vs. object size