# Processing Queries Against Database Procedures: A Performance Analysis

Eric N. Hanson

*Air Force Wright Aeronautical Laboratories*
*Wright-Patterson AFB, OH 45433*

## Abstract

A database procedure is a collection of queries stored in the database. Several methods are possible for processing queries that retrieve the value returned by a database procedure. The conventional algorithm is to execute the queries in a procedure whenever it is accessed. A second strategy requires caching the previous value returned by the database procedure. If the cached value is valid at the time of a query, the value is returned immediately. If the cached value has been invalidated by an update, the value is recomputed, stored back into the cache, and then returned. A third strategy uses a differential view maintenance algorithm to maintain 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 which algorithm is preferred 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,SAH87]. A collection of query language statements stored in a field of a record is known as a *database procedure*. For example, consider the following relation schema:

faculty(name, dept, salary, hobbies)

The "name", "dept" and "salary" fields have ordinary data types. However, "hobbies" is of type procedure. An example tuple in "faculty" is shown below:

| | |
|---|---|
| name: | Susan |
| dept: | Art |
| salary: | 20K |
| hobbies: | **retrieve** (music.all)<br>**where** music.name = "Susan"<br><br>**retrieve** (tennis.all)<br>**where** tennis.name = "Susan" |

The queries stored in the "hobbies" field retrieve the hobbies of Susan from the "music" and "tennis" relations.

Database procedures can provide support for several desirable features [SAH87], 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].

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. The original version of this paper was done while the author was at the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.



ACM 0-89791-268-3/88/0006/0295 $1.50

Several different algorithms are possible for processing queries that retrieve the value computed by a database procedure. The conventional method, 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 [SAH84,SAH87]. Another scheme which is based on saving previous values returned by the database procedure is called *Cache and Invalidate*. In Cache and Invalidate, when the procedure is accessed, if a valid result for it is in the cache, the stored result is used. Otherwise, the result is recomputed, stored to refresh the cache, and returned. If a base relation update changes the data on which the procedure result depends, the currently cached result is marked invalid. This method has been proposed in [StR86,SAH87] 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,Han87b] can be applied to maintain 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 discusses the analysis of the cost of the three different query processing strategies for database procedures. Section 5 presents the performance results for model 1. Section 6 gives the performance results for model 2. Finally, section 7 summarizes and presents conclusions.

## 2. Details of Query-Processing Algorithms for Procedures

Below we present a more detailed description of the Always 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

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

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.

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:

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

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 in [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 [Sho87,Gev87]. Using a Rete network to maintain views 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.

• **$\alpha$-memory** nodes: These nodes serve to hold the output of **T-const** nodes. Any token input to an **$\alpha$-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 $\alpha$-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.

• **$\beta$-memory** nodes: These nodes hold the output of **and** nodes. Otherwise, they are similar to $\alpha$-memory nodes.

The Rete network can be used for view maintenance because *$\alpha$ and $\beta$-memory nodes are equivalent to views.* Consider a memory node $m$ in a Rete network. The portion of the Rete network above $m$ represents the qualification of a view which we will call $V_m$. For any transaction-consistent state of the database, node $m$ contains the current value of $V_m$ [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 $\beta$-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 (angle brackets are used to delimit tuples):

$t$ = <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," where 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 $\alpha$-memory node below the node labeled "job = Programmer," and passed to the succeeding **and** node. The opposite $\alpha$-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 ($T'$ is a combination of two other tuples, as indicated by the enclosing set of angle brackets):

$T'$ =

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

$T'$ will then be added to the $\beta$-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 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

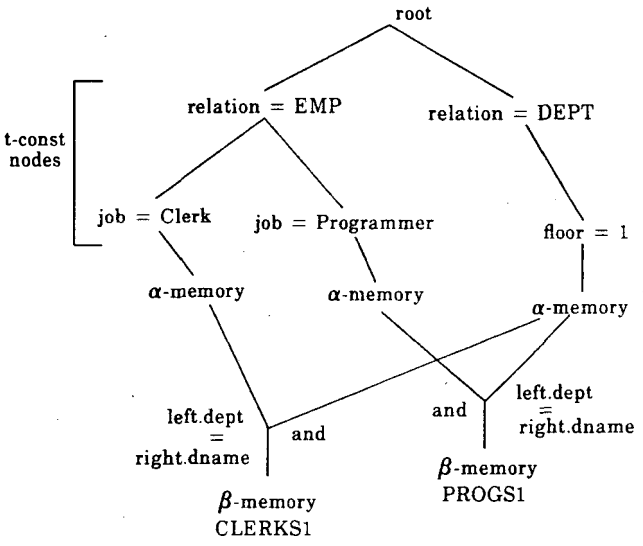


**Figure 1.** Example Rete network

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.

A significant advantage of RVM is that a single execution plan suffices for maintaining a view, no matter which base relations used in the view definition are modified. In AVM, many different plans may be required to update a view, depending on which base relations are modified. It may not always be possible to compute all these plans in advance because of the memory required to store them all. Perhaps only plans that are expected to be used frequently could be stored. It is assumed in this paper that all plans needed for AVM are computed in advance and stored. This assumption is reasonable for the simple update model assumed here. However, it may not be a valid assumption in more complex update environments.

Clearly, the code necessary to implement either Update Cache strategy will be complex. However, similar code has been successfully implemented before in forward-chaining rules systems for artificial intelligence programming [For82]. It may be worth the effort to implement the code since it can be used for several purposes, including

1. view maintenance
2. procedure maintenance
3. testing database trigger conditions
   (triggers are forward chaining rules)

The use of view maintenance algorithms to test trigger conditions is discussed in [Han87b].

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

## 3. Database Procedure Models Analyzed

Two different models for the structure of procedures are evaluated. In both models 1 and 2, it is assumed that each stored procedure consists of a single **retrieve** query. These queries retrieve data from one or more of the following relations, which have the storage structures indicated below (the notation sel-$x(R)$ indicates a selection predicate on relation $R$ with selectivity $x$):

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

In model 1, procedures may be of two types. The first procedure type type ($P_1$) is a simple selection of one relation, $R_1$. The second procedure type ($P_2$) is a two-way join query. In model 2, there are also two types of procedures, $P_1$ and $P_3$. As in model 1, $P_1$ procedures are simple selections, however, $P_3$ procedures are 3-way joins. Procedures of type $P_1$, $P_2$ and $P_3$ have the following structure:

$P_1$ (simple selection):

**retrieve** ($R_1$.all)
**where** sel-$f(R_1)$

$P_2$ (2-way join):

**retrieve** ($R_1$.fields, $R_2$.fields)
**where** $R_1$.a = $R_2$.b
**and** sel-$f(R_1)$
**and** sel-$f_2(R_2)$

$P_3$ (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** sel-$f(R_1)$
**and** sel-$f_2(R_3)$

The following table summarizes the procedure types contained in models 1 and 2:

| | model 1 | model 2 |
|---|---|---|
| $P_1$ | yes | yes |
| $P_2$ | yes | no |
| $P_3$ | no | yes |

The width of tuples in $P_1$, $P_2$ and $P_3$ procedures is $S$ bytes. The database contains $N_1$ procedures of type $P_1$, and $N_2$ of type $P_2$ (model 1) or $P_3$ (model 2). Using a shared view maintenance algorithm (RVM) there is a possibility of sharing subexpressions in both models. Procedures of type $P_1$ can form a shared subexpression for procedures of type $P_2$ or $P_3$ if the selection term sel-$f(R_1)$ is the same. The models contain a parameter SF which is the *sharing factor*. It is assumed that a fraction SF of the type $P_2$ or $P_3$ 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 $P_2$ or $P_3$ 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 after a crash by playing the latest part of the log against the last checkpoint. 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 invalidating an object, a parameter for it called $C_{\text{inval}}$ is included in the models.

A summary of the parameters used in the procedure cost model is shown in figure 2. Their default values are shown in figure 3.

## 4. Cost Analysis

A detailed analytical performance model has been developed to estimate the average cost of accessing a database procedure. This model is based on the procedure structure models 1 and 2 and the parameters described in the previous section. Due to space limitations, it is not possible to include the derivation of the performance model here. Readers interested in the derivation are referred to the original technical report for a full discussion [Han87c].

| *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 sel-$f$ |
| $f_2$ | selectivity factor of predicate term sel-$f_2$ |
| $f_{R_2}$ | 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) |
| $C_{\text{inval}}$ | cost to record the invalidation of a cached procedure value |
| $Z$ | locality of reference factor |

**Figure 2.** Procedure query cost parameters

| | | | |
|---|---|---|---|
| $N$ | 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 | $C_{\text{inval}}$ | 0 |
| $Z$ | .2 | | |

**Figure 3.** Default parameter values

The cost analysis derives estimates for the average cost of accessing a procedure in both model 1 and model 2 for the following procedure materialization strategies:

Always Recompute
Cache and Invalidate
Update Cache (Non-Shared)
Update Cache (Shared)

The non-shared view maintenance algorithm analyzed is algebraic view maintenance. The shared view maintenance algorithm analyzed is Rete view maintenance. As an example, the Rete network used to maintain individual procedures of type $P_1$ and $P_2$ in model 1 is shown in figure 4. The potential exists for sharing $P_1$ procedures as subexpressions of $P_2$ procedures. Similarly, in model 2, $P_1$ procedures may be shared subexpressions of $P_3$ procedures.

## 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 predicate selectivity $f$.

Figure 5 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 $C_{\text{inval}}$=60ms. Figure 6 plots the same curves for $C_{\text{inval}}$=0. Figures 5 and 6 clearly show that the total cost per query using Cache and Invalidate is highly sensitive to the value of $C_{\text{inval}}$. Thus, if Cache and Invalidate is implemented, it is important to keep $C_{\text{inval}}$ 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 6, 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.

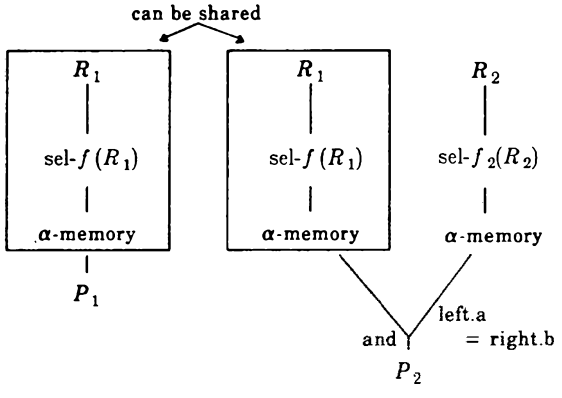


Figure 4. Rete networks for type $P_1$ and $P_2$ procedures in model 1

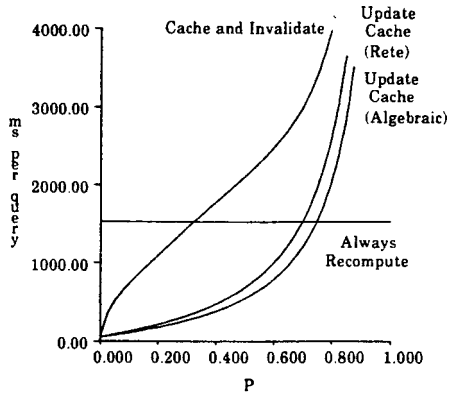


**Figure 5.** Query cost vs. update probability for high cache invalidation cost (60 ms)

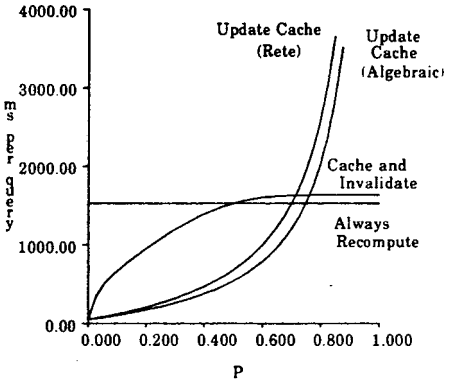


**Figure 6.** Query cost vs. update probability for low cache invalidation cost (0 ms)

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 sel-$f$ ($R_1$) 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 6, 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 7. 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 8. For this value of $f$, type $P_1$ and $P_2$ procedures contain 10 tuples and 1 tuple, respectively. Figure 8 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 9. 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 $P$.

An attempt was made in the analysis to model situations where some objects are accessed more frequently than others. 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. Figure 10 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 11 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 than it does in figure 6. 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 12 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 $\alpha$-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 13 shows the regions where each algorithm performs best for different object sizes and update probabilities. The area where Cache and Invalidate wins in figure 13 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.

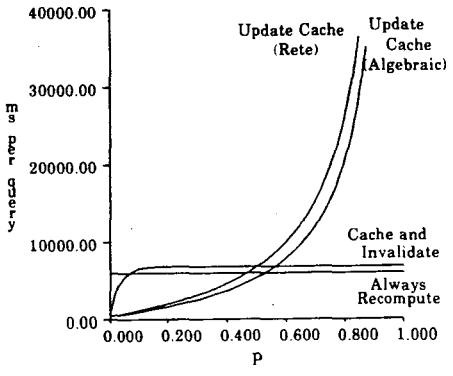


**Figure 7.** Query cost vs. update probability for large objects ($f$ =0.01)

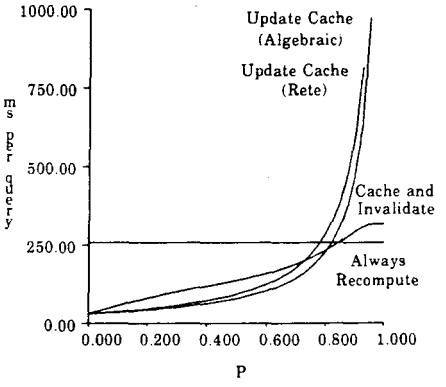


**Figure 8.** Query cost vs. update probability for small objects ($f$ =0.0001)

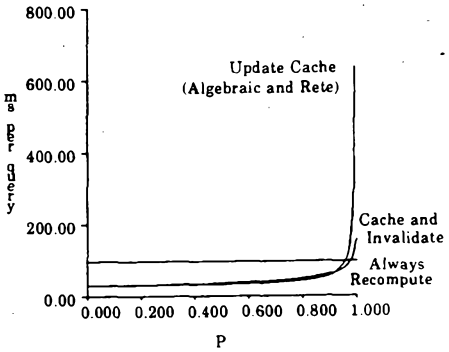


**Figure 9.** Query cost vs. update probability for single-tuple objects ($f$ =1/$N$)

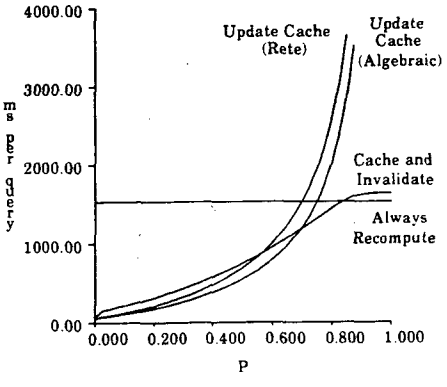


**Figure 10.** Query cost vs. update probability for high locality ($Z$=0.05)

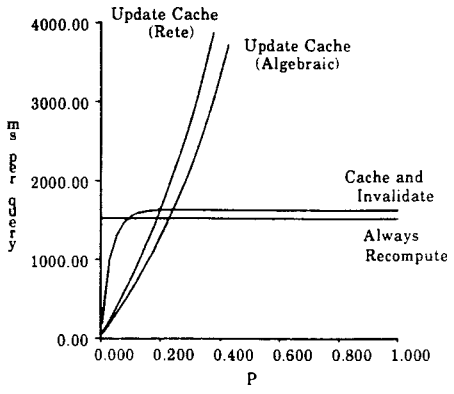


**Figure 11.** Query cost vs. $P$ for large number of objects ($N_1$=$N_2$=1000)

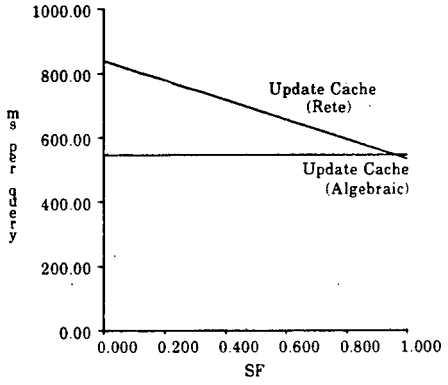


**Figure 12.** Query cost vs. sharing factor (SF)

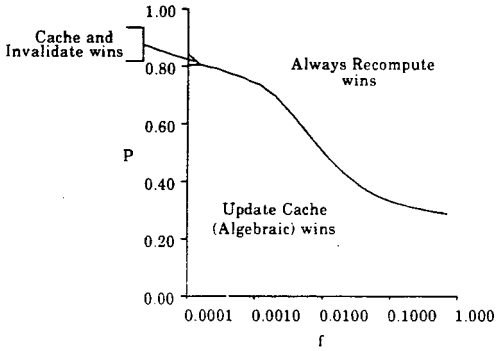


**Figure 13.** Areas where each method wins for object size vs. update probability

In figure 14, 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 15 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 16 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.

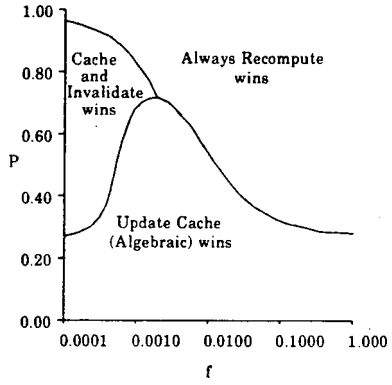


**Figure 14.** Areas where each method wins assuming high locality ($Z$=0.05)

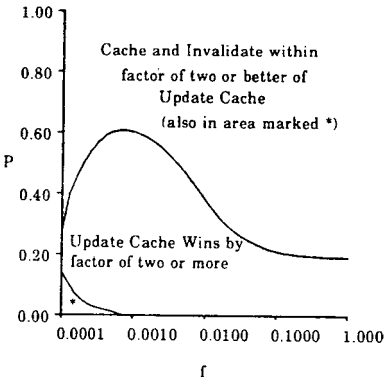


**Figure 15.** Measure of closeness between Cache and Invalidate and Update Cache

## 6. Performance Results for Model 2 Procedures

In model 2, a different Rete network structure is used in the Update Cache strategy based on RVM, as shown in figure 17. The performance results for Model 1 and Model 2 are similar, as can be seen by comparing figure 18 with figure 6. 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 19 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 $\beta$-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 $\alpha$-memory becomes less than the advantage provided by the precomputed subexpression in the $\beta$-memory. Figure 20 shows the areas where each algorithm performs best for update probability vs. object size in Model 2. Figure 20 is similar to figure 13 for Model 1, except that the best version of Update Cache is RVM instead of AVM.

## 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

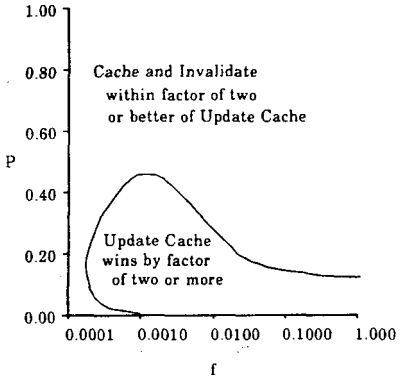


**Figure 16.** Measure of closeness ($f_2$=1)

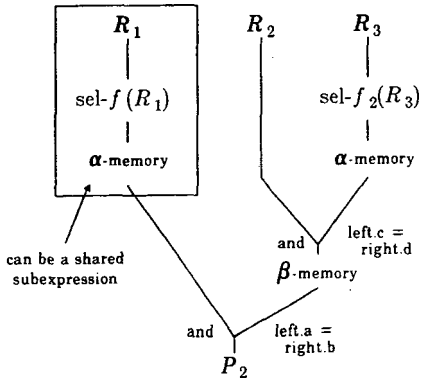


**Figure 17.** Model 2: Rete Network for $P_3$ Procedures

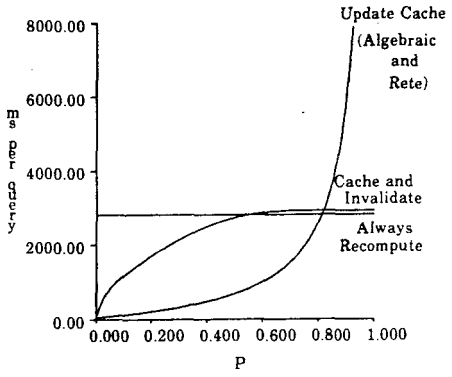


**Figure 18.** Model 2: Query cost for default parameters

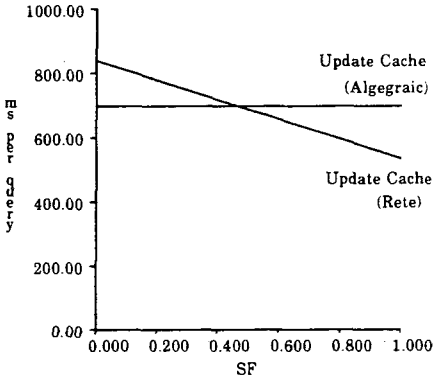


**Figure 19.** Model 2: Query cost of Update Cache alternatives vs. sharing factor

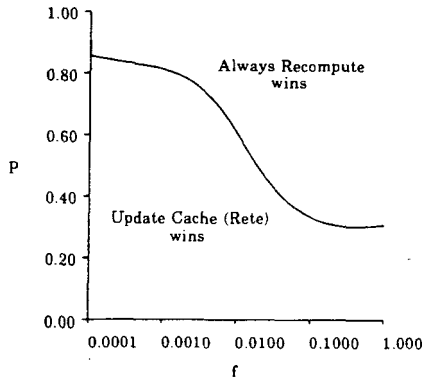


**Figure 20.** Model 2: Winners for update probability vs. object size

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 with $P$=0.1 in model 1 (as shown in figure 8), 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 $\alpha$-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 strategy 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, and to test the conditions of database triggers.

**References**


[BLT86]
Blakeley, J. A., Larson, P. and Tompa, F. W. Efficiently Updating Materialized Views. In *Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data*, Washington, DC, May 1986.

[Dat81]
Date, C. J. Referential Integrity. In *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. *Computer* (May 1987).

[Gra78]
Gray, J. N. Notes on Database 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. In *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 Relational Database Systems.* PhD Thesis, University of California, Dept of EECS, Berkeley CA, 1987.

[Han87c]
Hanson, E. N. Processing Queries Against Database Procedures: A Performance Analysis. University of California Memo. No. UCB/ERL M87/68, Berkeley, CA 94720, 1 Sept 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. Efficiently Supporting Procedures in Relational Database Systems. In *Proceedings of the 1987 ACM-SIGMOD Conference on Management 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. In *Proceedings of the 1984 ACM-SIGMOD Conference on Management of Data*, Boston, Mass., June 1984.

[SAH87]
Stonebraker, M., Anton, A. and Hanson, E. Extending a Database System with Procedures. *ACM Transactions on Database Systems, 2*, 3, September 1987, 350-376.

[StR86]
Stonebraker, M. and Rowe, L. The Design of POSTGRES. In *Proceedings of the 1986 ACM-SIGMOD Conference on Management of Data*, Washington, DC, May 1986.