A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures †

Anant Jhingran
EECS Department
University of California, Berkeley

Abstract

POSTGRES allows fields of a relation to have procedural (executable) objects. POSTQUEL is the query language supporting access to these fields, and in this paper we consider the optimizing process for such queries. The simplest algorithm for optimization assumes that the procedural objects are executed in full, whenever needed. As a refinement to this basic process, we propose an algorithm wherein cost savings are achieved by modifying the procedural queries before executing them. In another direction of refinement, we consider the caching of the materialized results. Two caching strategies—caching in tuples, and separate caching—are considered. The fifth algorithm is flattening, where a POSTQUEL query is modified into an equivalent flat query, and then optimized through a traditional optimizer. We study the relative performances of these algorithms under varying conditions and parameters. Our results show that caching wins when updates do not occur with a high frequency, and that separate caching is, in general, better than caching in tuples. We further show that when the composition of the objects in the procedural field is predictable and parameterizable, flattening is a good option.

1. INTRODUCTION

Query optimization in relational database systems has been a traditional research problem. A number of algorithms for optimizing queries have been proposed [SELI79, WONG76]. They are based on a variety of paradigms [JARK84], and work well for the traditional relational model.

† This research was sponsored by the Defense Advanced Research Project Agency (DoD), Arpa Order No. 4781, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

However, a number of recent proposals which enhance Codd's [CODD70] model require the modification of the existing algorithms to optimize the new set of queries that were not possible before. In this paper we study the query optimization problem in one such extended relational model, namely POSTGRES. We present a number of algorithms for optimization of queries in such an environment, and do a performance study of each.

The rest of the paper is organized as follows. In Section 2 we present the extensions in POSTGRES relevant to our study. We also discuss the previous work on optimization of queries on procedural objects. The optimizing paradigm and the details of the algorithms under consideration are then discussed in Section 3. In Section 4 we present the framework in which we compare the various algorithms. Section 5 presents the results of our study. Finally, this paper ends with the conclusions on the viability of each algorithm.

2. POSTGRES PROCEDURES

The relational model has been found deficient in many areas of database applications (e.g., knowledge management [ZANI85], and engineering applications [STON83]). As a result, there have been several proposals to enhance the relational model. These extensions either address specific deficiencies (e.g., ADT INGRES [STON83]), or a broad spectrum of inadequacies (e.g., Starburst, EXODUS, GENESIS, DASDBS and POSTGRES [IEEE87]).

POSTGRES [STON86], a new relational database system currently being developed at Berkeley, extends the relational model in several ways. The one relevant to this study is the addition of procedural data types. Thus, in addition to the standard data types permitted by all systems (integer, real, character etc.) and abstract data types permitted by some systems [STON83], fields of the relations in POSTGRES can contain procedural objects.

Procedural objects are executable programming constructs. In this paper we restrict ourselves to those procedures that are queries on the underlying database. The fields containing such queries are called POSTQUEL fields. The presence of procedural objects lends a

high degree of flexibility to the design of a database schema and allows many complex problems (e.g., storage of query plans, representation of hierarchical information, and sharing of subobjects by complex objects) to be naturally addressed [STON87].

Consider the following relations in a database schema:

DEPT (number = c10, name = c10, mgr = c10)
EMP (name = c10, hobbies = POSTQUEL,
dept = POSTQUEL)
SOFTBALL (name = c10, day = c10, position = c10)
FOOTBALL (name = c10, day = c10, position = c10)
MUSIC (name = c10, instrument = c10)

The field day in SOFTBALL and FOOTBALL refers to the day the person plays that game. Each employee has zero or more hobbies. The field EMP.hobbies consequently contains up to three POSTQUEL queries, one for each hobby. Each employee belongs to exactly one department. For example, the relation EMP may look like:

name	hobbies	dept
John	retrieve (SOFTBALL.day, SOFTBALL.position) where SOFTBALL.name = "John" retrieve (FOOTBALL.day, FOOTBALL.position) where FOOTBALL.name = "John"	retrieve (DEPT.all) where DEPT.number = 9
Mary	retrieve (SOFTBALL.day, SOFTBALL.position) where SOFTBALL.name = "Mary" retrieve (MUSIC.instrument) where MUSIC.name = "Mary"	retrieve (DEPT.all) where DEPT.number = 8

Table 1 gives a set of queries which would be used to illustrate the optimizing algorithms.

Query#	Query
1	retrieve (EMP.hobbies.day) where EMP.name = "John"
2	retrieve (EMP.name) where EMP.dept.name = "TOY"

Table 1: Example POSTQUEL queries

Query 1 retrieves the days John plays something, and query 2 retrieves the names of the employees who work in the "TOY" department.

The POSTQUEL fields can contain arbitrary POSTQUEL queries. To distinguish between the POSTQUEL queries present in the procedural fields (such as those in EMP.hobbies) and the queries used to access these fields (such as those in Table 1), we refer to the former as objects and the latter as queries.

POSTQUEL extends QUEL in many ways [ROWE87]. The extension which deals with the procedural objects is the *multiple dot* notation (like GEM [ZANI83]). The execution of the queries in the procedural fields in a multiple dot notation is implicit. For example, in Query 1, the target field EMP.hobbies.day can only be determined after the queries in EMP.hobbies are executed.

The *depth* of a field is one less than the number of dots in its multiple dot representation. Thus EMP.name is at depth zero, and EMP.hobbies.day is at depth one. Clauses containing fields with depth greater than zero are called *extended* clauses. The rest are called *ordinary* clauses.

The two POSTQUEL fields in EMP are fundamentally different in terms of the nature of the objects they contain. EMP.hobbies contains objects of unpredictable composition. For example, as the database evolves, employees may take up new hobbies, and give up old ones. As a result, the set of queries that occupy EMP.hobbies may change dynamically, and can only be determined after each tuple in EMP is fetched and examined. In contrast, the composition of objects in EMP.dept is fixed—each tuple of EMP contains exactly *one* object of the form:

retrieve (DEPT.all) where
DEPT.number = $dept-number

where $dept-number may differ across the tuples.

In the case of EMP.dept, it makes sense to store only the parameter $dept-number instead of the entire POSTQUEL query. The relation EMP would thus contain:

name	hobbies	dept
John	retrieve (SOFTBALL.day, SOFTBALL.position) where SOFTBALL.name = "John" retrieve (FOOTBALL.day, FOOTBALL.position) where FOOTBALL.name = "John"	9
Mary	retrieve (SOFTBALL.day, SOFTBALL.position) where SOFTBALL.name = "Mary" retrieve (MUSIC.instrument) where MUSIC.name = "Mary"	8

Now consider Query 2. It can be converted into the following "flattened" query:

retrieve (EMP.name) where
EMP.dept = DEPT.number and
DEPT.name = "TOY"

This sort of query modification, which removes the extended clauses in a POSTQUEL query, is generally possible if the structure of the objects in a POSTQUEL field is the same across all the tuples. Such query modification is referred to as *flattening*. It is similar in

concept to the flattening of SQL queries introduced in [KIM82].

2.1. Previous Work

Optimization of queries on procedural objects has been studied from a perspective different from ours in [SELL87] and [HANS88]. [SELL87] is an overview of the preliminary ideas on optimizing POSTQUEL queries. It discusses an optimizing strategy for POSTQUEL queries based on the decomposition and tuple substitution strategy in [WONG76]. This optimizer postpones the evaluation of the extended clauses to the very end of the query plan. Furthermore, it is based on a greedy approach. Consequently, the plans that it produces may be suboptimal. In contrast, our optimization strategies are based on an exhaustive search approach as used in System R, with the extended clauses integrated into such a framework. We have shown the viability and advantages of such an approach in [JHIN87].

A significant part of [SELL87] is devoted to discussions on the caching strategies for materialized objects. Both finite and infinite cache space are considered. However, the discussion is exploratory in nature and fails to reach specific conclusions.

Hanson [HANS88] studies the relative performance of three algorithms for dealing with procedural fields—Always Recompute, Cache and Invalidate, and Update Cache. The last two algorithms differ in the ways in which the cached set of objects is kept current. An elaborate parameterized model is presented, which is then used to compare the three algorithms. The study assumes the availability of infinite cache space and concludes that caching strategies win if the probability of update to the cached objects is low.

Our study has points in common with Hanson's, but is more extensive in many aspects. We study two Cache and Invalidate strategies (as opposed to just one in [HANS88]), and make the realistic assumption of bounded cache space. Furthermore, we discuss the actual query optimization problem. In [HANS88] the objective function for the optimization algorithms is the expected cost of accessing *one* procedural object. A similar assumption is made in [SELL87] when the caching alternatives are discussed. We, however, have the objective function as the cost of an entire POSTQUEL query, which may involve processing many objects. In particular, we identify two different classes of POSTQUEL queries which result in very different algorithm performance. We also discuss query modification strategies which reduce query costs substantially under some circumstances.

3. OPTIMIZATION ALGORITHMS

We have developed five basic algorithms for optimizing POSTQUEL queries. These algorithms treat ordinary clauses similarly; they differ only in their handling of the extended clauses. Each algorithm assumes a different query processing strategy. Accordingly, the discussion of the optimization algorithms is also a discussion of the corresponding query processing strategies.

3.1. The Basic Strategy

The System R query optimizer looks through most of the viable query plans and estimates the cost of each. It then selects the plan with the least estimated cost [SELI79]. To do so, it needs the estimates of the *selectivities* of the clauses in the query. The cost of a plan is a function of the selectivities and the costs of relational accesses and joins.

The functions used for selectivities and costs must be modified in order to be applicable to extended clauses [JHIN87]. To discover if a tuple satisfies an extended clause may involve execution of one or more procedural objects. Determining the cost and selectivity of this process requires some statistical information about the queries in the procedural fields. The execution of procedural objects is also termed as *materialization.*

Ordinary clauses can be used to reduce the cost of relational accesses if suitable indexes are available [SELI79]. In other words, tuples not satisfying ordinary clause(s) may be filtered out by using a suitable access path. The same is not true for extended clauses, which can generally be tested for only after the tuples have been fetched. For example, while an index on EMP.name would help in Query 1 (where one may fetch only the tuples satisfying the clause), the clause in Query 2 cannot act as a filter for accessing tuples of EMP. For optimization purposes, extended clauses can be treated like ordinary clauses, provided the modified cost and selectivity functions are used, and the above limitation (of when extended clauses can be tested for) is kept in mind. All our algorithms (except FLAT) use this approach, but differ in their cost functions for evaluating extended clauses. They use an exhaustive search strategy, and evaluate the extended clauses from left to right. For example, in Query 2 (having the extended clause EMP.dept.name = "TOY"), a plan would involve fetching a tuple of EMP and materializing the object in the dept field of that tuple to determine the matching tuple of DEPT. In case the extended clause is deeper, this process would continue further.

The first algorithm, *Complete Materialization with No Caching* (CM) is the simplest one. CM assumes that the cost of executing an object has to be paid in full every time materialization is needed. There is no concept of storing these materialized results for future use. For example, in Query 2, the cost of the plan in CM includes

the cost of fetching the tuples of EMP, the cost of executing the procedural object in each of these tuples, and the cost of checking for each of these materializations if the result has the name field as "TOY."

3.2. Restricted Materialization (REST)

Materialization returns all the subobjects of a procedural object. Sometimes we are not interested in the entire relation returned by the materialization of a POSTQUEL object; a subset of the tuples might suffice. Under these circumstances, it is possible that cost savings may be achieved by modifying the POSTQUEL object(s) before executing them so that they only return the tuples of interest. In Query 2, the plan in REST pays the cost of fetching the tuples of EMP, and for each tuple e in EMP, the cost of the following *restricted* materialization:

retrieve (DEPT.all) where
DEPT.number = "emp-dept"
and DEPT.name = "TOY"

where "emp-dept" is the actual value of the parameter $dept-number in e. Note that under such a plan, there is no need to check if the tuple(s) returned by the (restricted) materialization have their name field(s) as "TOY."

It is thus possible that the extended clause in a query can be used to modify some or all the objects that need to be materialized. REST checks if such an object modification is semantically valid. If this is the case, the plan includes restricted materialization of these objects. Otherwise, it is similar to CM in all aspects.

3.3. Caching Strategies

The two algorithms mentioned above keep no history. Consider the case where the employees "Mary" and "John" belong to the same department, and therefore contain identical objects in their corresponding dept field. Moreover, assume that the tuple for "John" is accessed before that of "Mary" in answering Query 2. The POSTQUEL object in John's tuple will be executed first. If the result of this query execution could be stored, then the execution of the object in Mary's tuple could be avoided. It is thus clear that caching of materialized objects might help in reducing the cost of executing a POSTQUEL query.

There is another important benefit of caching. Consider a sequence of queries, $Q_1, \ldots, Q_n$, which are not submitted as a batch (and hence global query optimization algorithms such as in [SELL88] do not work). The processing of any query would involve materializing some objects. It is possible that the execution of a query Q_j can utilize one or more of the objects materialized by the queries $\{Q_i : i < j\}$. Of course, updates will invalidate materialized objects, but where they are infrequent, caching is likely to be beneficial [SELL87, STON87].

Caching strategies can be broadly classified into result caching and plan caching. In this study, only the former is considered since our model does not take into account the cost of generating a plan. Even result caching can be accomplished in various ways. Here we discuss two result caching strategies, which lead to two different optimizing algorithms. Both these algorithms materialize an object only if its current version does not exist in cache. In all other aspects, they are similar to CM.

3.3.1. Complete Materialization—*Cache Separately* (CS)

In CS, the materialized objects are cached in a separate **cache** relation on the disk. Each object in the database has a *unique_id* which is a function of the *query_block* (the structure of the object), and *list_of_parameters* (the set of parameters that uniquely identify a particular object within the objects that have the same *query_block*). The *unique_id* is the input to a hash function that determines the *slot* in the **cache** relation where the object should be cached.

Whenever an object needs to be evaluated, CS determines its *unique_id* and then hashes into **cache.** If a current version of the materialized object is found, it is retrieved and the cost of executing the object is avoided. If such a version does not exist, the object is materialized and stored in **cache** if space permits. Note that under these assumptions, one page access is required to check if a result is cached. The number of page accesses to retrieve a cached relation, of course, depends on its size.

3.3.2. Complete Materialization—*Cache in Tuples* (CT)

In this approach, the materialized objects are stored in the tuples themselves. When the results are small and there is some free space in each page, the materialized result can be cached in the same page as the tuple containing the object. As a result, if a small object is cached, it can often be retrieved without paying any extra cost of I/O. For large objects, we make the assumption that the first page of the cached object is stored clustered with the tuple. Under these assumptions, it follows that the number of page accesses required to retrieve a cached object in CT is one less than the number of accesses required in CS. We refer to the extra cost in CS as the cost of *cache lookup.* Note that in CS, this cost has to be paid even if the object is not cached.

On the other hand, if the fraction of all objects that are cached is *cached_fraction*, then CT may need to cache many more objects (and hence require much more space) to achieve the same *cached_fraction* as CS. This happens because objects may be repeated across tuples. For example, consider the objects in EMP.dept. If there

are 100,000 employees, and 500 departments, then to achieve a *cached_fraction* equal to one, CS would need to cache 500 objects, whereas CT would need 100,000.

3.4. Flattening (FLAT)

Flattening as a means of evaluating a POSTQUEL query has been discussed in Section 2. A flattened version of a POSTQUEL query can be passed through a traditional optimizer and a plan generated. This plan can be no worse than the plan of CM and REST. The other algorithms evaluate an extended clause in a top down approach (i.e., from left to right). This order of relational accesses is just one of the many options available in FLAT query. If the other options are cheaper, then FLAT would do better.

Consider Query 2, and its flattened version:

```
retrieve (EMP.name) where
EMP.dept = DEPT.number and
DEPT.name = "TOY"
```

The other algorithms pick a tuple of the EMP relation, and for each tuple, fetch the "matching" tuples of DEPT. This is equivalent to a nested loop join in a FLAT strategy with EMP as the outer relation, and DEPT as the inner relation. The cost of an inner fetch in FLAT corresponds exactly to the cost of a (restricted) materialization in REST. FLAT is likely to win if a merge join (i.e., join after sorting EMP and DEPT on the fields EMP.dept and DEPT.number respectively), or a bottom up evaluation (i.e., using DEPT as the outer and EMP as the inner relation) is cheaper.

There are two factors that mitigate the seeming superiority of FLAT. The first is that there is no hope of caching materialized objects. Thus, while FLAT would certainly be better than REST or CM, it may be worse than CS or CT. The second is a more practical reason. If the number of objects in a tuple is large, and/or their composition is unpredictable, then flattening is unviable. For example, consider Query 1. Since the set and structure of queries in EMP.hobbies may change dynamically, it is not possible to store the parameters of these objects in suitable field(s).

Techniques for flattening a POSTQUEL query parallel various view modification algorithms [STON75], and their discussion is beyond the scope of this paper.

4. MODEL

In order to compare the various algorithms, we have constructed optimizers of limited functionalities which generate the cheapest plan and its expected cost. This expected cost is the yardstick used to evaluate the corresponding processing strategy. To simplify our evaluation task, we have made certain assumptions and parameterized some conditions. These parameters characterize the POSTQUEL query and other system and database characteristics. In this section we discuss our model in detail.

4.1. POSTQUEL queries

We restrict our study to the POSTQUEL queries of two types:

Type	Structure
1	retrieve ($REL.Procfield.Ordfield_1$) where $REL.Ordfield_2$ **operator** value
2	retrieve ($REL.Ordfield_2$) where $REL.Procfield.Ordfield_1$ **operator** value

$Ordfield_1$ is an ordinary field in the target list of the POSTQUEL queries in the POSTQUEL field *REL.Procfield*. $Ordfield_2$ is an ordinary attribute of *REL*. In queries of Type 1, all the objects in the POSTQUEL field of the selected tuple of *REL* need to be materialized. In contrast, in queries of Type 2, the materialization process can stop as soon as a match is found. The relative performance of the algorithms is highly dependent on the type of the query involved. More complicated queries (deeper and/or more extended clauses) are beyond the scope of this paper, and further, their behavior is often similar to the simpler ones of our choice.

Apart from the Type, there is one more parameter associated with the POSTQUEL queries. For queries of Type 1, *Selbot* is the selectivity of the clause in the qualification. In Type 2 queries, *Selbot* is the selectivity of the clause that is used to modify a POSTQUEL object in REST. The relative performance of the query modification algorithms (both REST and FLAT) is dependent on this parameter.

4.2. Update Model

Updates to the underlying database invalidate some or all of the objects materialized by the POSTQUEL queries. This section discusses the model for studying the performance of the caching algorithms in the presence of updates.

4.2.1. Object Invalidation

When an object is materialized and cached for the first time, I-locks (Invalidation locks) are set on the tuples and index intervals read during the execution of that object. Updates to the tuples and index intervals involve invalidation of all those objects which have I-locks on them. Updates do not remove the I-locks set on the tuples, and hence only the first materialization and caching of an object requires the setting of I-locks. However, all updates do involve paying the extra cost of invalidation over the case where no caching is done.

With the passage of time, the number of objects that have been materialized and cached at least once increases. We make the simplifying assumption that *all the objects have been cached at least once* before we begin comparing the performance (in other words, we ignore the transient behavior). Thus the cost of setting I-locks does not enter our calculations.

4.2.2. Query Sequence

Consider a random sequence of queries, where each query is an update with probability $Pr(UPDATE)$ and a retrieve with probability $1 - Pr(UPDATE)$. All retrieve queries are POSTQUEL queries made solely of either Type 1 or of Type 2 queries (depending on the experiment under consideration). All updates are POSTQUEL **replace** commands (without any extended clauses, though) which update a fraction of the tuples in the relation(s) touched during the materialization of POSTQUEL objects. This fraction is fixed at 0.01 for the remainder of the study. The tuples that are updated are selected at random.

The fundamental nature of the caching algorithms is best brought out by their average case responses; and not the responses to some particular query sequence. Therefore, for a fixed set of parameters, we have used random query sequences of length hundred, and then averaged the behavior over a hundred such sequences. The average responses are fairly stable at this point.

4.3. Database Structure

The relations in the database contain the fields required to make the objects and the queries syntactically correct. The indexes on the various fields can be of the type Primary (PRIM), Clustered Secondary (C-SEC), Non Clustered Secondary (NC-SEC) or non-existent (NEXIST). In the absence of any index of these types, a relation can be accessed through a SEGMENT scan [SELI79]. The assumptions of the cost of relational access through these indexes (as well as for joins) are the same as in [SELI79].

The POSTQUEL field contains *objects_per_tuple* POSTQUEL objects in a tuple of *REL*. These objects are simple selections and projections on a (set of) relation(s) such that the POSTQUEL queries of Type 1 and 2 are syntactically valid. Each object is a single relation query containing exactly one qualification clause—a selection. Thus its structure is:

retrieve ($ObjRel.Ordfield_1$) where
$ObjRel.Ordfield_3$ **operator** value

The number of tuples returned by an object is fixed at ten for the remainder of this study. (Note that within the same column, *ObjRel* may not be the same in all the objects. For example, EMP.hobbies contains objects with *ObjRel* as SOFTBALL, FOOTBALL and MUSIC.) A top down query plan accesses the tuples of *REL*, and for each such tuple, determines the matching tuples of *ObjRel* by materializing (if necessary) the object(s) in *REL.Procfield*. A bottom up plan accesses *ObjRel* before *REL*. Such a bottom up plan is possible only in a flattened query.

The cost of an object depends on (among other things) the index on the attribute in its qualification. This index is called *Object_Index*, and its type determines the cost of the objects:

[1] *Easy*: These objects have *Object_Index* as C-SEC, and typically cost 2-3 page accesses for their materialization.

[2] *Hard*: These objects have *Object_Index* as NEXIST, and cost a complete SEGMENT scan for materialization.

A *PROC_MIX* fraction of the objects in each POSTQUEL field are hard, and $1 - PROC_MIX$ are easy. Thus a *PROC_MIX* near zero represents a majority of easy objects, and a *PROC_MIX* near one, a majority of hard objects.

Use_Factor is the expected number of times each object is repeated in a column. Thus if there are *P distinct* objects (i.e., no two having the same *query_block* and *list_of_parameters*) in *REL.Procfield*, then

$$Use_Factor = |REL| \times objects_per_tuple \; / \; P$$

In the absence of updates and with limited *Size_Cache*, the *cached_fractions* in CS and CT are in this ratio.

There is one more parameter of interest—*Flat_Index*. This is the index on the fields that store the parameters for FLAT. The bottom up plan in FLAT is aided by the presence of indexes on these fields. For example, a *Flat_Index* = C-SEC means that there exists a clustered secondary index on EMP.dept when it stores the parameter $dept-number. In a bottom up query plan for the flattened version of Query 2, a tuple e in EMP that matches a tuple d in DEPT satisfies the following condition: e.dept = "particular-dept-number." Here "particular-dept-number" is the value in the field d.number. A nested loop/merge join is facilitated by the presence of *Flat_Index*.

4.4. Parameters of Study

Table 2 shows the parameters of the study, along with their default values. On the basis of some fixed parameters (not shown above), the size of the database relation is about 50 MBytes.

5. PERFORMANCE RESULTS

In this Section, the results of the performance analysis of the optimizing algorithms is presented. The

Query Parameters	
Name	Default
Type	1 or 2
Selbot	0.1
Database Parameters	
Name	Default
objects_per_tuple	1
PROC_MIX	0.0001 or 0.4
Use_Factor	2
Object_Index	C-SEC or NEXIST
Flat_Index	NC-SEC
Other Parameters	
Name	Default
Size_Cache	10 MBytes
Pr(UPDATE)	0.2

Table 2: Parameters of Study

cost of an algorithm is the estimated cost of the plan it generates. The lower this cost, the better the algorithm is. We first discuss the cost characteristics of the algorithms as functions of some of the important parameters— *PROC_MIX*, *Pr(UPDATE)*, *Size_Cache* and *Use_Factor*. In the accompanying graphs, all costs have been normalized such that $Cost(CM) = 1$ at the smallest x coordinate. We next study the behavior of these algorithms as functions of pairs of these parameters. Finally, the effects of the other parameters not included in the list above are discussed.

5.1. Cost of the Objects

Figure 1(a) plots the normalized costs as a function of *PROC_MIX* for queries of Type 1, while Figure 1(b) does the same for queries of Type 2. Note that *PROC_MIX* is an indicator of the expected cost of materialization of the procedural objects.

5.1.1. Type 1 Queries

For these types of queries, a top down plan involves restricting *REL* using the qualification, and then executing the objects in the selected tuples to determine the target field values. No modification of the objects (as desired in REST) is possible, and hence REST performs identically to CM. CT and CS are better than CM for all choices of *PROC_MIX*. Thus caching is a clear winner under these circumstances.

At low *PROC_MIX*, the extra cost of cache lookup is a significant fraction of the total cost. As a result, CT performs better than CS, in spite of having a lower *cached_fraction*. At higher *PROC_MIX*, the lookup cost is negligible, and the higher *cached_fraction* for CS makes it win.

As mentioned before, the cost of an inner fetch in a top down, nested loop join in FLAT is the same as the cost of a materialization in REST (and hence, the same as CM). Thus if *PROC_MIX* is low, then this plan is the cheapest for FLAT. Since this plan is identical to CM's, FLAT follows the curve of CM. When *PROC_MIX* becomes sufficiently large, nested loop join is no longer the cheapest, and flat switches to merge scan, while maintaining a top down access of relations. For higher values of *PROC_MIX*, it abandons the top down approach altogether, and does a bottom up query evaluation. Under these circumstances, the cost of FLAT becomes independent of the cost of the objects.

5.1.2. Type 2 Queries

In a flattened version of a query of Type 2, there is a restriction on *ObjRel*. This, together with the presence of a default secondary index on *REL.Procfield* (*Flat_Index* = NC-SEC), makes the bottom up plan the best for a FLAT query. Since this plan is unaffected by *PROC_MIX*, the curve for FLAT is a horizontal line, which is substantially lower than the other curves. Thus FLAT is definitely superior for queries of Type 2, especially for high *PROC_MIX*. CS and CT show a behavior similar to Type 1 queries.

For queries of Type 2, REST is always cheaper than CM. This is primarily due to the fact that the cost of materializing modified objects is never more than that of the corresponding objects. The extra clause (a selection on $ObjRel.Ordfield_1$) helps in reducing the cost of materialization if an index exists on $ObjRel.Ordfield_1$, and if the access of *ObjRel* through such an index is cheaper than the other indexes. The default parameters provide for an NC-SEC on $ObjRel.Ordfield_1$. At low *PROC_MIX*, a scan of *ObjRel* through this index is not the cheapest, but beyond a certain *PROC_MIX*, this is the best plan. From the figure we note that for *PROC_MIX* > 0.1, materialization of a modified object is cheaper than the corresponding object, and is independent of *PROC_MIX*. For very high *PROC_MIX*, REST is even better than the caching algorithms.

5.2. Updates

Figure 2 plots the curves for Type 1 query as a function of *Pr(UPDATE)*. It can be seen that as *Pr(UPDATE)* increases, the two caching algorithms deteriorate. With an increase in the frequency of updates, there is an increase in the number of objects being invalidated. This has a two-fold effect. First, invalidation costs increase. Second, a retrieve query sees fewer cached objects on the average; and hence has to do more materialization, and pay a higher processing cost.

An update in a query sequence invalidates a higher number of cached objects if they are being cached in

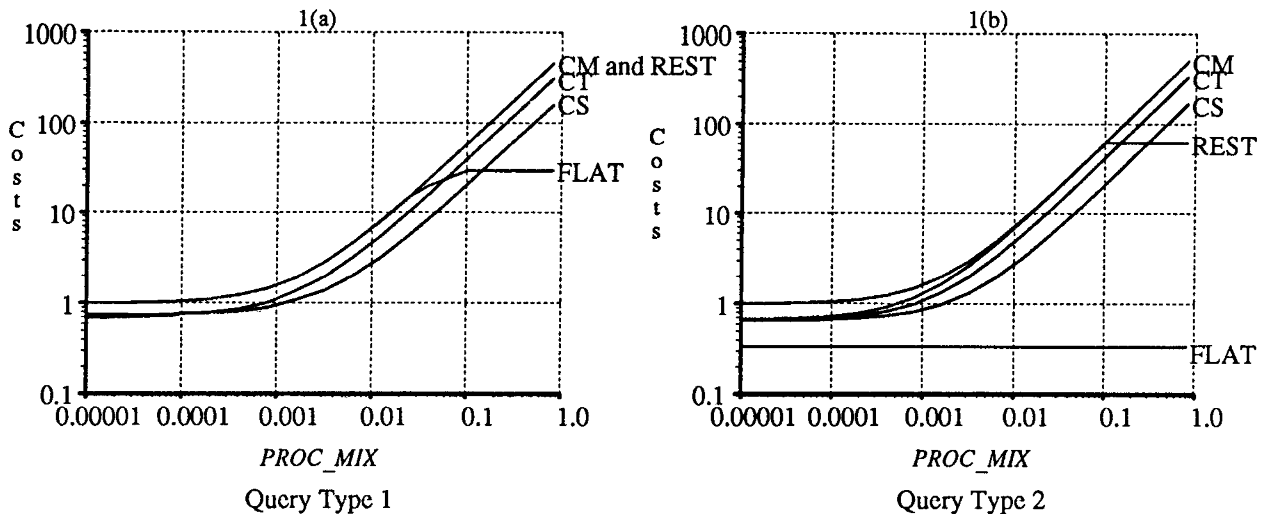


Figure 1: Normalized Costs as a function of *PROC_MIX*

tuples compared to when they are cached separately (in an approximate ratio of *Use_Factor*). As a consequence, updates penalize CT more than CS.

5.3. Size of the Cache

According to the parameters, CS requires a cache space of ≈ 12.5 MBytes to achieve a *cached_fraction* of one. At sizes more than this, only CT benefits. Figure 3 plots the cost characteristics of the two caching algorithms and CM as a function of *Size_Cache*. The curves for REST and FLAT are omitted for the sake of clarity. CT is better than CS for either very small *Size_Cache* (where the performance penalties of extra lookup are more than the extra caching benefits of CS); or for a very large *Size_Cache* (≈ 22 Mbytes), where the *cached_fraction* in CT is close to one.

5.4. Level of Sharing

An increase in *Use_Factor* has a two-fold effect on the cost of CS. First, for a given *Size_Cache*, the *cached_fraction* increases. Second, an update causes a lower number of invalidations. Thus it is obvious that as *Use_Factor* increases, CS would be more and more appealing. Figure 4 plots $\frac{Cost(CT)}{Cost(CS)}$ as a function of the *Use_Factor* for the four possible choices of *PROC_MIX* and query type. The significant point of note is the earlier flattening in low *PROC_MIX* queries. Thus for inexpensive objects, CT gives a comparable performance to CS, for all values of *Use_Factor*.

We have seen various reasons why CT, in general, performs worse than CS. In Figure 5 we attempt to capture these reasons for a high *PROC_MIX* query of Type 1. Note how the ratio falls as *Size_Cache* is raised to 25 MBytes (which is sufficient to cache all objects in CT). Even with this *Size_Cache*, the curve of Cost(CT)/Cost(CS) is above one. The reason for this is the extra penalties of updates in CT. When *Pr(UPDATE)* is made zero (and *Size_Cache* is still 25 MBytes), the ratio of costs drops to below one. This curve represents the ideal conditions for a caching algorithm—enough cache space, and no updates. Under these circumstances, CT is definitely superior.

From now on, we restrict ourselves to $Size_Cache \leq 10 MBytes$. This is in keeping with our assumption of a bounded cache space. The larger sizes which we encountered so far were used only to bring out the fundamental differences in the algorithms.

5.5. Regions of Optimal Performance

We now turn to the behavior of the algorithms as functions of pairs of the above parameters by plotting the regions where each algorithm performs the best.

In Figures 6(a) and 6(b), the regions as functions of *Pr(UPDATE)* and *Use_Factor* are shown for queries of Type 1. It is clear that for a sufficiently high *Pr(UPDATE)*, the caching algorithms would prove to be non-competitive. Referring to Figure 6(a), consider a horizontal line drawn through $Use_Factor = 2$. CT is the best algorithm until $Pr(UPDATE) \approx 0.4$. Then CS becomes the best. As *Pr(UPDATE)* increases further, even CS fails to be better than the other algorithms. Note that for $Use_Factor < 1.5$, CS never wins.

When the objects are expensive (Figure 6(b)), CT is never preferred. CS is the best for high *Use_Factor*

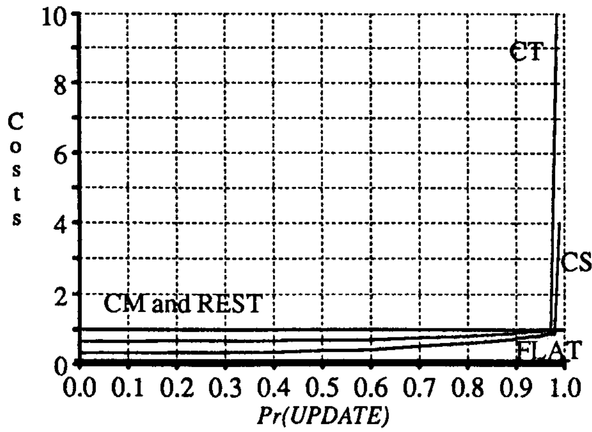


Figure 2: Costs vs. *Pr(UPDATE)*

(Type 1 Queries, *PROC_MIX*=0.4)

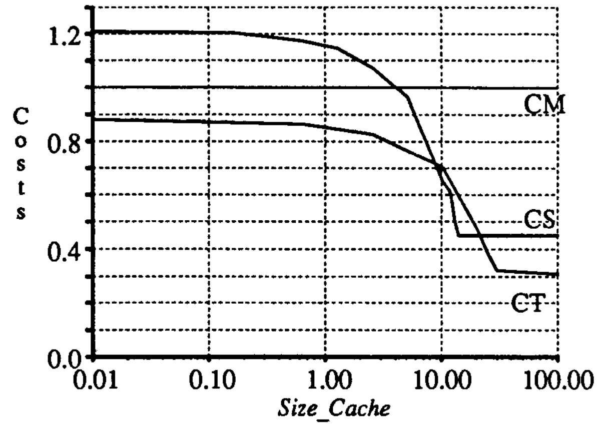


Figure 3: Costs vs. *Size_Cache*

(Type 1 Queries, *PROC_MIX*=0.0001)

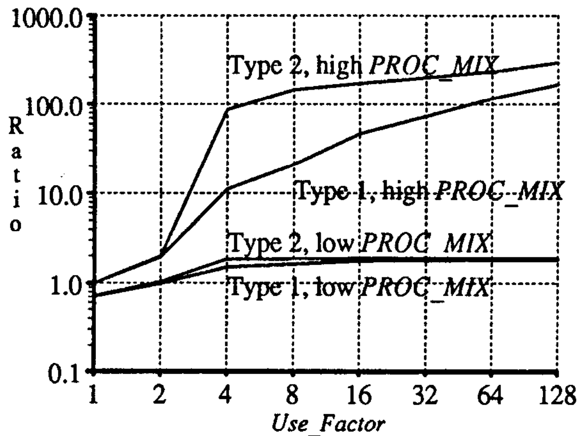


Figure 4: Cost(CT)/Cost(CS) vs *Use_Factor*

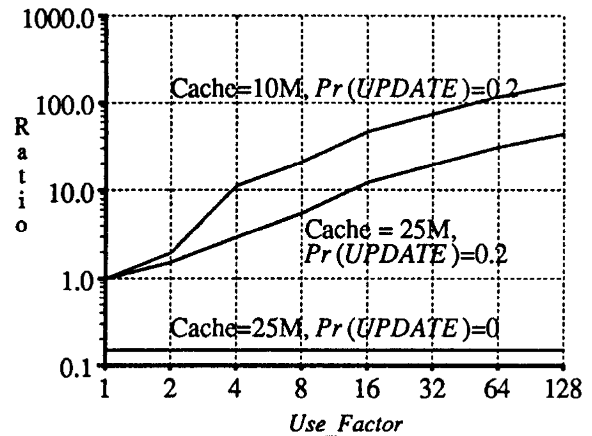
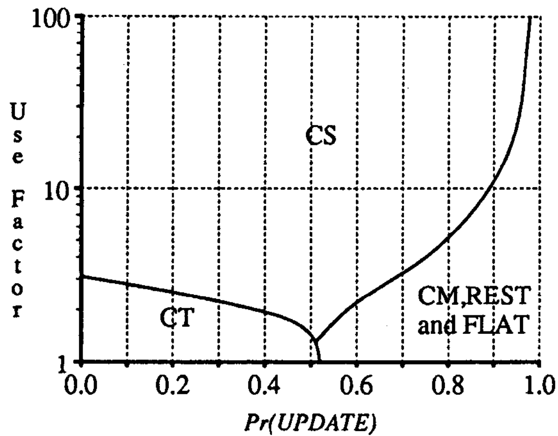
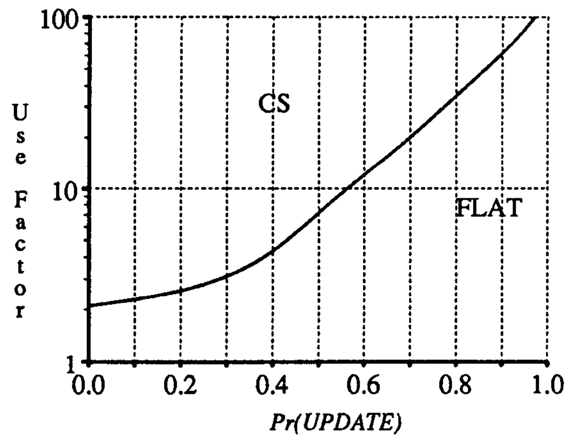


Figure 5: Dependence of Cost(CT)/Cost(CS) on *Size_Cache* and *Pr(UPDATE)*



6(a): *PROC_MIX* = 0.0001



6(b): *PROC_MIX* = 0.4

Figure 6: Regions of best performance as functions of *Use_Factor* and *Pr(UPDATE)*

and/or low $Pr(UPDATE)$. From both the figures, it is clear that if the *Use_Factor* > 100, CS is extremely competitive unless $Pr(UPDATE) \approx 1$.

Figure 7 plots the regions as a function of the two most important parameters for the caching algorithms— $Pr(UPDATE)$ and *Size_Cache*. It can be seen that for low $Pr(UPDATE)$, CT is better than CS for low cache sizes. Taking a vertical slice, (say at $Size_Cache = 10\,MBytes$), for $Pr(UPDATE) < 0.4$, CT is the best, for $0.4 < Pr(UPDATE) < 0.6$, CS is the best, and for $Pr(UPDATE) > 0.6$, the non-caching algorithms perform the best. This result is similar to what was obtained in Figure 6(a) along the line $Use_factor = 2$.

The next figure (Figure 8) captures the behavior as function of *PROC_MIX* and $Pr(UPDATE)$. In the upper left corner, note how an increase in *PROC_MIX* makes CS more and more competitive. This is because as *PROC_MIX* increases, so does the cost of materialization. Consequently, the benefits of caching go up. This continues until the bottom up algorithm for FLAT beats any caching approach (also see Figure 1a).

5.6. Other Parameters

In this subsection we briefly discuss the other parameters of our study.

5.6.1. Number of Objects per Tuple

FLAT has been shown to be distinctly superior in case of queries of Type 2 and under some circumstances for queries of Type 1. This is partly a result of the default choice of $objects_per_tuple = 1$. We now discuss the implications of $objects_per_tuple > 1$ on FLAT. Consider the following schema:

PAIRS (seed = i4, partners = POSTQUEL)
MEN (seed = i4, name = c10, country = c10)
WOMEN (seed = i4, name = c10, country = c10)

which describes the players taking part in a tennis tournament. PAIRS contains the data about the mixed double tournament, and MEN and WOMEN about the singles tournament for men and women respectively. PAIRS.partners contains two queries of the form:

retrieve (MEN.all) where
MEN.name = $param1

retrieve (WOMEN.all) where
WOMEN.name = $param2

The POSTQUEL query

retrieve (PAIRS.seed) where
PAIRS.partner.seed < 5

returns the seeds of the mixed double teams where either partner has a seed better than 5 in his/her respective tournament. Assume that we store the parameters of these objects (instead of the full queries) in the fields PAIRS.male and PAIRS.female. We have seen before that FLAT performs similar to REST if it chooses a top down approach. This holds even if $objects_per_tuple > 1$, as is in this case.

In contrast, in a bottom up plan (i.e., accessing MEN and WOMEN before PAIRS), FLAT needs to

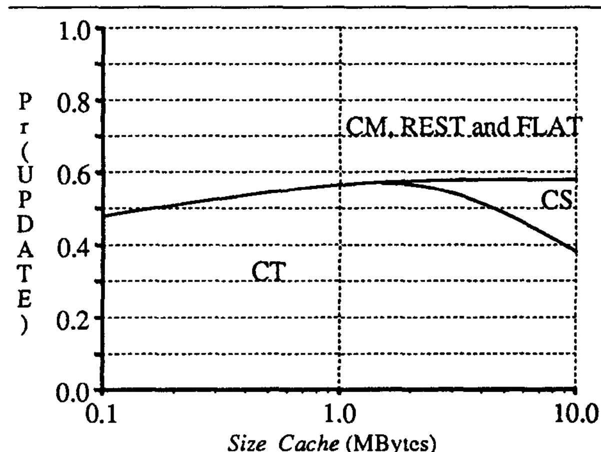


Figure 7: Regions of best performance as functions of *Size_Cache* and *Pr(UPDATE)* (Type 1 Queries, *PROC_MIX*=0.0001)

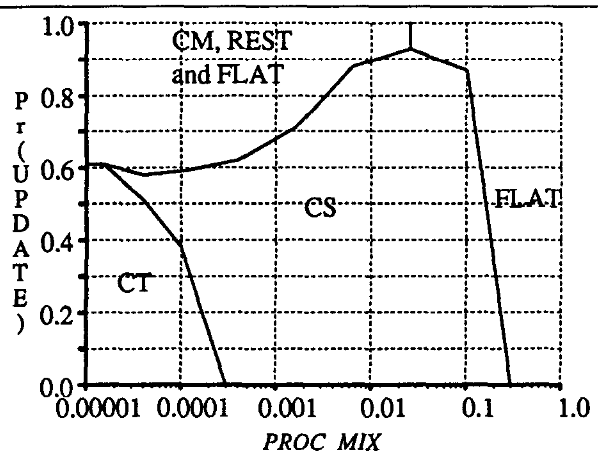


Figure 8: Regions of best performance as functions of *PROC_MIX* and *Pr(UPDATE)* (Type 1 Queries)

execute (an equivalent of) the following two queries:

retrieve (PAIRS.seed) where
PAIRS.male = MEN.name and
MEN.seed < 5

retrieve (PAIRS.seed) where
PAIRS.female = WOMEN.name and
WOMEN.seed < 5

Assuming the availability of indexes on PAIRS.male and PAIRS.female, the best plan for each subquery is bottom up. If the total cost of these two subqueries is more than a top down plan, the latter is chosen. Otherwise, FLAT chooses the option of executing these two subqueries.

In general, for low *objects_per_tuple*, a sequence of subqueries performing an equivalent task would be cheaper. As *objects_per_tuple* increases (and so does the number of subqueries), the bottom up approach is no longer the best, and then FLAT switches to top down and performs similar to REST. This is confirmed in Figure 9.

5.6.2. Flat Index

Figure 10 plots the curve for FLAT and CM as a function of *Selbot* for different choices of *Flat_Index*. In queries of Type 2, FLAT has a bottom up plan if the clause on *ObjRel* is highly selective. As this clause becomes less selective, the cost of the bottom up plan increases, and after a point FLAT switches to top down. As we have seen before, a *Flat_Index* helps lower the cost of a bottom up plan. Consequently, FLAT maintains a competitive edge till a high value of *Selbot* if *Flat_Index* is C-SEC. On the other hand, the absence of this index makes FLAT switch to a top down plan at low values of *Selbot*.

As *objects_per_tuple* increase, a *Flat_Index* is needed on each field that stores a parameter, if FLAT is to perform better than other algorithms.

6. CONCLUSIONS

We have shown that the caching algorithms are competitive in situations of low to moderate update probability. In this, our conclusions are similar to [HANS88]. Moreover, it has been demonstrated that separate caching is better than caching in tuples under most circumstances. This is especially true when the cache size is limited and *Use_Factor* is high because separate caching is able to achieve a higher *cached_fraction*. Furthermore, updates penalize CT more than they do CS. There are two factors that may mitigate this superiority of CS. First, if the objects are cheap, then CS would suffer because of the extra lookup costs. The second factor is the implementation problems of CS. We have assumed the availability of "hashing" into a cache relation. As the objects become more complex, so would the hashing strategy. Since our model does not take this into account, its effects have not entered the picture.

In cases where the number of objects in a tuple is near one and their composition is predictable and easily parameterizable, it has been further shown that flattening

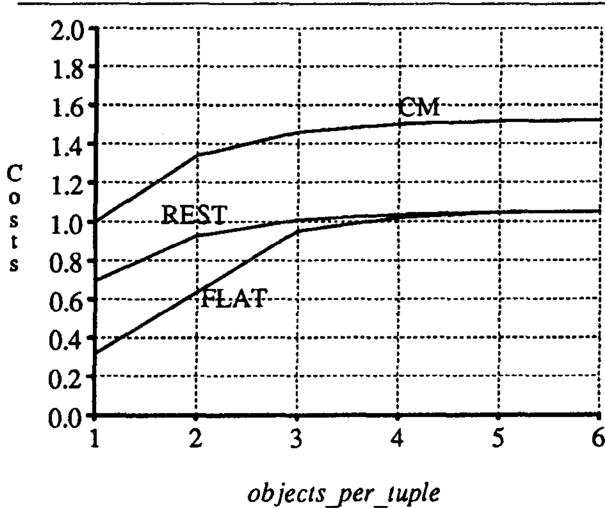


Figure 9: Effect of *objects_per_tuple*
(Query Type 2, *PROC_MIX* =0.0001)

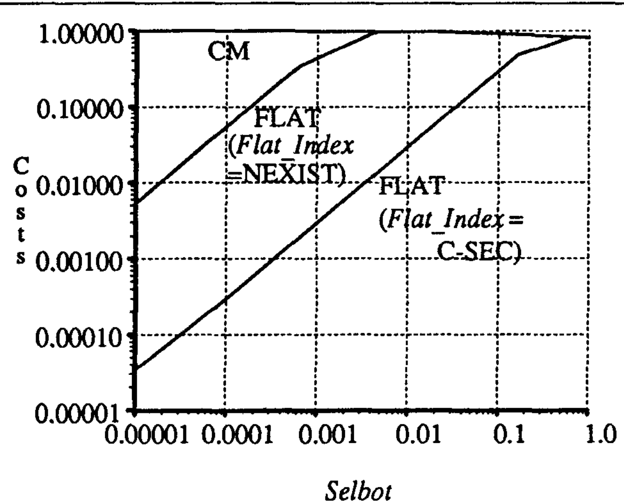


Figure 10: Effect of *Flat_Index* on FLAT
(Query Type 2, *PROC_MIX* =0.0001)

is a good option. This is especially true if the query is best solved by a bottom up approach. As the number of objects per tuple increases, FLAT loses its competitive edge. To emphasize again, flattening is not possible when the composition of the objects is unpredictable.

It is clear that CM is the preferred alternative in presence of frequent updates, and where flattening is not viable. REST is never worse than CM, but its marginal utility is often negligible. Moreover, if the cost of generating the plans (which has not entered our picture) is also a criterion, then REST would perform worse than it does in our studies.

It may seem that caching and restricted materialization are orthogonal (and thus the two may be applied together in a strategy). However, it can be shown that caching benefits restricted materialization only within a query (Section 3.3) with inter query benefits being highly unlikely. We consider the latter as much more important, and have therefore not examined this possibility.

A real query optimizer will, in general, be based on one or more of the above strategies. The actual choice(s) of the strategies will depend strongly on the factors discussed in this study. It is necessary to determine these parameters before such a choice can be made.

Though we have discussed the optimizing algorithms in a specific environment, the discussion on the various strategies should extend to any system supporting procedural objects.

Acknowledgements

The author wishes to thank his advisor, Professor Michael Stonebraker, for his advice on the work and on the preliminary drafts of this paper. Thanks are also due to the anonymous referees for their valuable suggestions.

REFERENCES

[CODD70] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," Comm. of ACM, June 1970.

[HANS88] Hanson, E., "Processing Queries against Database Procedures: A Performance Analysis," Proc. ACM-SIGMOD Conference on Management of Data, June 1988.

[IEEE87] Bulletin of the Computer Society of the IEEE TC on Database Engineering, Special Issue on Extensible Database Systems, Carey, M. ed., June 1987.

[JARK84] Jarke, M., and Koch, J., "Query Optimization in Database Systems," ACM Computing Surveys, June 1984.

[JHIN87] Jhingran, A., "A Compile Time Optimizer for Database Systems Supporting Procedures," Master's Report, University of California at Berkeley, May 1987.

[KIM82] Kim, W., "On Optimizing an SQL-like Nested Query," ACM Trans. on Database Systems, 7.3, Sept. 1982.

[ROWE87] Rowe, L., and Stonebraker, M., "The POSTGRES Data Model," Proc. 13th VLDB Conf., Brighton, 1987.

[SELI79] Selinger, P. et al., "Access Path Selection in a Relational Database Management System," Proc. ACM-SIGMOD Conference on Management of Data, 1979.

[SELL87] Sellis, T., "Efficiently Supporting Procedures in Relational Database Systems," Proc. ACM-SIGMOD Conference on Management of Data, May 1987.

[SELL88] Sellis, T., "Multiple-Query Optimization," ACM Trans. on Database Systems, 13.1, March 1988.

[STON75] Stonebraker, M., "Implementation of Views and Integrity Control by Query Modification," Proc. ACM-SIGMOD Conference on Management of Data, 1975.

[STON83] Stonebraker, M., et al., "Application of Abstract Data Types and Abstract Indices to CAD Databases," Proc. ACM-SIGMOD Conference on Engineering Design Applications, 1983.

[STON86] Stonebraker, M., and Rowe, L., "Design of POSTGRES," Proc. ACM-SIGMOD Conference on Management of Data, 1986.

[STON87] Stonebraker, M., et al., "Extending a Database System with Procedures," ACM Trans. on Database Systems, Sept. 1987

[WONG76] Wong, E., and Youssefi, K., "Decomposition—A strategy for query processing," ACM Trans. on Database Systems, Sept. 1976

[ZANI83] Zaniolo, C., "The Database Language GEM," Proc. ACM-SIGMOD Conference on Management of Data, 1983.

[ZANI85] Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects," Proc. International Conference on VLDB, 1985.