Inform. Systems Vol. 13, No. 2, pp. 175–185, 1988
Printed in Great Britain.

0306-4379/88 $3.00 + 0.00

INTELLIGENT CACHING AND INDEXING TECHNIQUES FOR RELATIONAL DATABASE SYSTEMS

TIMOS K. SELLIS†
Computer Science Department, University of Maryland, College Park, MD 20742, U.S.A.

(*Received* 22 *April* 1987; *in revised form* 3 *October* 1987)

Abstract—Recent developments in the design of database systems include proposals for several extensions to the basic model of relational database systems. In this paper we present some ideas on the design of intelligent support mechanisms for large databases where procedures in the form of database commands are stored in relation fields. First, we examine the idea of storing results of previously processed procedures in secondary storage (*caching*). Problems associated with cache organizations, such as replacement policies and validation schemes, are examined in depth. Another means for reducing the execution cost of queries is indexing. Conventional indexing techniques assume that all values are known, such schemes cannot be used effectively. As a solution to that problem, a new indexing scheme, Partial Indexing, is proposed and analyzed. Uses of partial indexes in conventional database systems are also described.

1. INTRODUCTION

Recent developments in the design of Database Management Systems (DBMSs) include proposals for several extensions to the basic model of relational systems. Systems based on the Object Oriented Programming paradigm [1, 2], systems that provide support for storing both general knowledge (e.g. in the form of rules) and data [3–5] (also called Expert Database Systems [6, 7]) and finally systems that have certain extensibility capabilities [8–11], have been proposed in an attempt to make DBMSs capable of supporting other than the traditional business applications. Main targets of such systems are Engineering and Artificial Intelligence applications [6, 12, 13].

Clearly, any of these proposals will need some kind of extended relational query language to support a high-level user interface. Examples of such languages are GEM [14], POSTQUEL [11] and DATALOG [15]. Because of their extended capabilities, such languages need special support for the efficient execution of queries. Previous work in the area of processing queries in extended relational DBMSs has focused on optimizing the execution of new types of operations such as transitive closure queries [13, 16, 17], general recursive queries [18, 19], deduction [4, 20], database procedures [21, 22], etc. Physical and conceptual modeling, concurrency control and crash recovery are some other well known DBMS problems [23]. The solutions to many of these problems can still be used in extended relational DBMS environments. However, performance will deteriorate due to the complexity of the new operations. Our goal is to examine ways of improving the performance by providing more sophisticated optimization tactics. In particular, we concentrate on the problem of query processing. Issues that deal with user interfaces, physical and conceptual modeling, consistency in a multiple user environment, and robustness are examined in more detail in [11] in the context of the design of a new DBMS being developed at the University of California, Berkeley, called POSTGRES.

This paper is organized as follows. Section 2 discusses some issues involved in query processing and introduces the basic ideas to be examined, namely caching and indexing. Sections 3 and 4 discuss in detail the use and implementation of caching and indexing techniques respectively. Finally, we conclude in Section 5 by summarizing our contributions and pointing out interesting future research problems.

2. QUERY PROCESSING

Query processing and optimization has traditionally focused on issues such as syntactic query transformations and the use of access structures (indexing). Using such principles, considerable progress has been made towards the design and implementation of sophisticated query processors for record oriented databases [24–26]. Recent developments in the design of extended relational database systems introduce more advanced entities and models such as objects [27] or procedures [28] which significantly increase the functionality of database systems. However such extensions increase the complexity and the cost of query processing as well. For example, the use of procedures adds a new operation, namely the execution of a procedure (i.e. a set of database commands), which is of significant cost.

We motivate the necessity of intelligent caching and indexing techniques through some examples us-

†Also with University of Maryland Systems Research Center and Institute for Advanced Computer Studies.

ing the language QUEL + [28], an extension to QUEL [29] that supports procedures. Originally, Stonebraker *et al.* suggested in [30] the idea of storing database commands in the database as a means for increasing the functionality of the system. Commands are stored in relation fields and can be accessed as any other field using a slightly extended query language. Moreover, since these commands can be executed, a new operation is introduced allowing a user to execute the contents of relation fields. The details of procedure based database systems are outside the scope of this paper and are presented elsewhere [28, 31]. We give here some examples to illustrate the use of procedures.

Example 1: storing programs in a database

In many applications that use data residing in a database there is a need for code written in the data manipulation language of the DBMS, i.e. *database programs*. These programs can be stored in the database and then be executed using the DBMS query language. For example, in [32] it was shown how a problem like heuristic search can be addressed using such an extended database management system. There, a relation

ALGORITHMS(alg_id,alg_type,code)

was used, where alg_id is a unique identifier, alg_type indicates the general class that the given algorithm belongs to (e.g. Dynamic Programming, Branch and Bound, etc.) and code is a field used to store the database procedure that implements the algorithm. Therefore the form of the relation ALGORITHMS will be

alg_id	alg_type	code
10	Dynamic Progr.	code line 1 code line 2
15	Dynamic Progr.	code line 1
20	Branch and Bound	code line 1

The syntax of the DBMS allows the user to select and execute an algorithm based on its alg_id and alg_type. Such a syntax may for example be

```
execute (ALGORITHMS.code)
where ALGORITHMS.alg_id = 15,
```

which will select the Dynamic Programming algorithm with identifier 15 and will process the commands that constitute the body (code).

Example 2: supporting rules

Suppose a relation EMP(name, salary, age), with the obvious meanings for its fields, and another relation CATEG_EMPS with the following contents, are given. This second relation gives a way to categorize employees according to their salaries or salaries and ages. In some sense it is a set of rules that define when an employee is wellpaid, underpaid, etc. A query asking for wellpaid employees would then be

status	emps
wellpaid	retrieve (EMP.name) where EMP.salary > 80 retrieve (EMP.name) where EMP.salary > 60 and EMP.age < 30 retrieve (EMP.name) where EMP.salary > 65 and EMP.age < 40
underpaid	retrieve (EMP.name) where EMP.salary < 20

```
retrieve (CATEG_EMPS.emps.name)
where CATEG_EMPS.status = "wellpaid",
```

where the reference to CATEG_EMPS.emps.name will first evaluate the queries stored in the emps field of CATEG_EMPS and then project the result of this evaluation on the name column. More complicated rules can be expressed using the full capabilities of the query language. In addition, general condition–action rules can be defined, since a procedure in a relation field may include update operations as well. Actions can then be implemented through updates to other relations in the database.

Example 3: supporting complex objects

Complex objects can also be implemented using database procedures. A query expression in a relation field simply describes the way components of other relations (i.e. tuples) are combined to build an instance of a more complex object. As an example, suppose we have a relation POINTS(x, y) describing points in the plane. Another relation

LINES(line_id,description)

can then be defined, where description is a field containing expressions of the form,

```
range of POINT,POINT1 is POINTS
retrieve
  (POINT.x,POINT.y,POINT1.x,POINT1.y),
where Qualification
```

describes how the two points POINT and POINT1 that define a line segment are selected from the POINTS relation. A significant advantage of using procedures for the definition of complex objects is the ability to allow many objects to share the same sub-objects. Hence, a hierarchy of objects can be built and inheritance is free since it can be naturally achieved through retrievals of data from the same relations [28].

It is clear from the above examples that supporting procedures in a DBMS is of significant importance. However, preliminary results in [28] show that there is a serious degradation in performance for non-

standard data retrieval operations. In [31] we have studied some of these problems and suggested techniques that improve the performance of extended database management systems. We will focus here on two of these ideas, namely *caching* and *indexing* of procedure results. Although the discussion to follow is restricted to procedures, our ideas apply to other environments as well (e.g. view materialization, database snapshots, complex objects).

Caching is the idea of storing results of previously processed procedures in secondary storage. Using a cache, the I/O and CPU cost of processing a query can be reduced by avoiding multiple evaluations of the same procedure. Caching problems give rise to other interesting issues such as policies for replacing entries of the cache with newly produced procedure results, algorithms that decide if a given result should be cached, etc. Also, results of procedures may become invalid when relations used in the evaluation of a procedure are updated. Checking the validity of cached entries becomes then an issue. These issues are discussed in detail in Section 3.

Another means for reducing the execution cost of queries is indexing. Indexes are used in DBMSs to provide efficient access to relations. When procedures are evaluated, the fields of the resulting relations can also be indexed. However, at any given time, it is highly probable that not all procedures stored in a relation have been evaluated. Therefore, a conventional indexing scheme cannot be used, for it would assume that all values resulting from the execution of procedures are known. As a solution to that problem, we propose in Section 4 a new indexing scheme called Partial Indexing. A partial index contains information only on results of procedures that have been materialized in the past. As such, it has the advantage of containing information frequently requested and adapts its content based on the observed queries.

3. CACHING RESULTS OF PREVIOUS COMPUTATIONS

As seen in the previous section, processing a database procedure amounts to executing possibly several QUEL queries. Hence, it will generally be very slow to perform this operation every time the result of such a procedure is needed. This section examines ways to making processing more efficient through the use of a cache.

3.1. What is caching?

We mentioned at several points in the previous sections that one way to avoid evaluating the same procedure multiple times, is *caching*. By caching we mean computing the result of a procedure and storing it in some specifically assigned area of secondary storage. This computation can be done either at the time tuples are inserted in relations or the first time they are referenced. We will call the former *precomputation* of procedures, since it occurs before even the result of the procedure is requested. However, our focus here is on the latter case, which is more natural. The basic idea is to keep in secondary storage computed (*materialized*) objects that are frequently used in queries. Under that formulation, the caching problem is conceptually the same with the well known caching problem in operating systems [33]. Notice also that the cache can be used not only for materialized procedures but for generally holding the results of any query issued by the user. These can be saved because either the same query may be given by a user frequently or they can be used to answer other queries [22, 34, 35]. Hence the discussion in this section applies to derived queries as well.

The caching problem introduces several subproblems to be solved. The following list is the set of issues that will be discussed here.

(a) *Which query results to cache?*
(b) *What algorithm should be used for the replacement of cache entries?*
(c) *How to check the validity of a cached procedure result?*
(d) *How to index the entries of the cache?*

We will assume that the general model of the cache is a limited area in secondary storage where entries of the form

(Qid,Signature,Query_expression,Result)

are stored. Qid is some unique identifier, Signature is a string used to partially identify query expressions and is discussed further in Sub-section 3.5, Query_expression is some canonical representation for queries, e.g. query graphs [26], and Result is the relation resulting after executing the query or set of queries that were found in some procedure and described by the third field (Query_expression). The following four sub-sections give answers to each of the above mentioned questions (a)–(d).

3.2. Which query results to cache?

Depending on the information known about the queries, the system can decide whether a result is worth caching or not. For a given materialization result R, this decision will generally be based on the frequency of references to R, the frequency of updating the relations used to build R, and the costs for computing, storing, and using R. Specifically, the following is the list of parameters to the caching problem: (Table 1).

Table 1. Caching problem parameters

Caching problem parameters	
$\mathscr{C}$	Size allocated for the cache
r_i	Probability of referencing result R_i
u_i	Probability of updating R_i
M_i	Cost of producing R_i (materialization)
S_i	Cost of writing R_i in the cache
U_i	Cost of using R_i from the cache
$\|R_i\|$	Size of R_i
IN	Cost of invalidating a cache entry

$\mathscr{C}$ is the number of disk pages allocated for the cache. r_i and u_i are the probabilities of referencing and updating a result R_i respectively. We will assume that these are probabilities over the lifetime of the system. Also, notice that procedure results cannot be updated directly. Thus, updating probabilities are derived from the probabilities of updating base relations used in the definition of procedures. M_i is the cost of materializing the procedure that gives the result R_i, while S_i and U_i are the costs of writing to and reading R_i from the cache respectively. Finally, it will be assumed that invalidating a single object in the cache incurs a cost IN. Given these parameters, we now describe various alternatives for the problem of selecting which results to cache. Depending on the amount of storage allocated for the cache, we differentiate between two cases: Unbounded and Bounded Space.

Unbounded space. In this case $\mathscr{C} = \infty$ and therefore the decision to cache a result R_i is local; that is, it depends only on the values of parameters associated with R_i. Each result is examined on an individual basis and consequently $u_i + r_i = 1$ will hold. This is true since operations on the database will either request for the result of a procedure or will indirectly update that result. The criterion we will use to decide if R_i is worth caching or not is based on comparing the cost of processing R_i throughout the lifetime of the system without using the cache, with the corresponding cost assuming that R_i will be cached. Let the two costs be denoted by NC_i and YC_i respectively. In the case where no caching is used, the result must be produced at each reference by materializing the corresponding procedure. Hence the total cost will be

$$NC_i = r_i M_i.$$

In the case where caching is used, a result is stored in the cache and is invalidated each time an update to the database has some effect on it. In order to compute the cost YC_i we will differentiate between the following four cases for the types of two subsequent requests. The cost formula given corresponds to the cost incurred in accessing the result for the *second* operation only.

(a) *Read–Update:* In this case the result is invalidated because of the update, the contribution to the total cost being

$$r_i u_i IN.$$

(b) *Read–Read:* In this case the result is simply read from the cache with total cost

$$r_i r_i U_i.$$

(c) *Update–Update:* The cost here is due to doing only the invalidation of the cached entry, that is

$$u_i u_i IN.$$

(d) *Update–Read:* This is the case where the procedure must be re-executed and stored in the cache. The total cost will be

$$u_i r_i (M_i + S_i).$$

Hence for the case where the cache is used, the cost of processing will be,

$$YC_i = r_i u_i IN + r_i r_i U_i + u_i u_i IN + u_i r_i (M_i + S_i)$$

or, since $r_i + u_i = 1$,

$$YC_i = u_i IN + r_i [r_i U_i + u_i (M_i + S_i)].$$

Comparing now YC_i and NC_i we can identify the cases where it is worth caching result R_i. That happens when $NC_i > YC_i$. Using the formulas derived above, and assuming that $S_i = 1$ and $IN = 1$ (one page access is needed to update the cache table when a new object is cached or an existing result is invalidated), we can see that this is true if

$$\boxed{M_i > U_i + \left(\frac{1}{r_i^2} - 1\right).}$$

Checking the above condition will determine whether the result of a given procedure materialization should be kept in the cache.

Bounded space. This case is more realistic than the previous, in the sense that some limited space on secondary storage is allocated for caching. Therefore, $\mathscr{C}$ is some finite number of disk blocks. In contrast to the criterion used for Unbounded Space, *all* objects to be cached must be considered. Let N be the number of results to be cached. Each object R_i has reference and update probabilities, r_i and u_i respectively. Since many results can now be affected by the same update to a base relation, one can no longer assume that $r_i + u_i = 1$. We will, however, state the following property that holds in this case,

$$\sum_i (r_i + u_i) = 1.$$

The formulas derived above for the case of using the cache are still valid. There is an additional constraint that must be imposed here, and that has to do with space limitations. This restriction indicates that the total space occupied by cached results cannot be more than $\mathscr{C}$. Given all these parameters, we now formulate the problem of caching in the case of Bounded Space.

Let $A: 1\mathbb{N} \rightarrow \{0, 1\}$ be an *allocation function.* A result R_i will be cached if $A(i) = 1$; if $A(i) = 0$, R_i will be discarded after it is used. Hence in the lifetime of the system, result R_i will contribute a total cost of

$$TC_i = \begin{cases} YC_i & \text{if } A(i) = 1 \\ NC_i & \text{if } A(i) = 0, \end{cases}$$

to the total processing cost. The optimal caching policy will be to cache some of the N objects so that the total cost is minimal and the space required is less than the allowed fragment on secondary storage. In other words, we seek a function A such that

$$\sum_{i=1}^{N} TC_i \quad \text{is minimal,}$$

subject to the constraint

$$\sum_{i=1}^{N} A(i)|R_i| \leqslant \mathscr{C}.$$

This problem of optimal allocation has been shown to be NP-complete [36]. However, Roussopoulos gives for a similar problem [37, 38] (*view indexing*) an A^* algorithm that finds a near-optimal allocation. We will not go into the details of that algorithm here; the reader is referred to [37] for a rigorous and detailed presentation of the technique.

The output of the A^* algorithm identifies which results are worth keeping in the cache. This allocation will be used throughout the lifetime of the system. Hence, this approach is meaningful only in the case where all procedures are materialized in advance and a decision is made on which of them should be cached. However, that allocation may not be the best to use. Periodically the system may re-run the same algorithm and use statistics acquired during the execution of various queries and updates. Even for objects not cached, the system may keep some statistics and recompute the allocation function A so that new results can get a chance to be stored in the cache. Due to the very high cost of the A^* algorithm, however, such a solution may be undesirable. Also, the above two cases assumed that the reference and update probabilities for the various objects were known in advance. In the most general case, the values of the above parameters are not known and the system must be able to dynamically adapt its caching behaviour, so that the contents of the cache always reflect the most frequently used and/or costly results. The following sub-section discusses these two issues in the context of the replacement policies that can be used for the cache.

3.3. Replacement algorithm

The problem of selecting a policy for replacing objects in the cache, is abstractly formulated as follows:

A *state* s of the cache is the set of objects that are stored in it $\langle R_1, R_2, \ldots, R_n \rangle$ along with some statistical information associated with each R_i. We will assume here that this information is

t_i	The time since R_i was last referenced
u_i	Probability of (indirectly) updating R_i
M_i	Cost of producing R_i (materialization)
$\|R_i\|$	Size of R_i,

and that the cost of writing and reading an object from the cache is equal to the size of that object. Let $\mathscr{S}$ and $\mathscr{R}$ be the set of all possible states and results to cache, respectively. Then, a *replacement policy* P, is a function $P: \mathscr{S} \times \mathscr{R} \rightarrow \mathscr{S}$ that, given a state s for the cache and a newly materialized result R_i, decides,

(a) if R_i should be cached, and
(b) in case the answer to (a) is positive but there is not enough free space in the cache to accommodate R_i which other result(s) should be discarded to free the space needed.

In operating systems, an optimal page buffer replacement policy is one that uses the future pattern of references to decide which pages should be cached (see algorithm OPT in [33]). This algorithm is not practical, though, unless one can predict with high probability the future behavior of the system. The closest approximation is the LRU (Least Recently Used) algorithm, which selects to discard the object with maximum time since its last reference. In the area of database management systems, the same policy can be used in the design of buffer managers. Chou and DeWitt [39] and Effelsberg and Haerder in [40] give an analysis of these algorithms in a database environment.

In our caching problem, an object R_i is cached independently of its parameters, as long as space can be allocated to store R_i in the cache. If there is not the case, then some result(s) must be discarded to free the space needed for storing R_i. There are generally two approaches one can take

(a) We can first try to approximate the parameters of Table 1 using the statistics the system has acquired. The sizes $|R_i|$ and the materialization costs M_i are given since the objects have been computed already. The update probability u_i is also easy to derive, assuming that the probabilities of updating base relations are given. For example, the probability of updating the result of a join between two base relations is equal to the sum of the probabilities of updating each of the two relations since each relation is assumed to be updated independently (recall also that $\Sigma_i(r_i + u_i) = 1$). What remains to be provided is the probability of referencing a result as well as the probability of updating the result, in the case where the frequencies with which base relations are updated are not known. For objects already in the cache, these probabilities can be estimated from the reference patterns already observed. For new results, one can predict the reference pattern if the query processing algorithm is known. For example, in the case of processing a join, if the query optimizer suggests that nested loops should be used, tuples are accessed in a pre-determined way (one page in from the outer relation, then all pages are accessed for the inner relation, etc.) and therefore we have a rough estimate for the needed probabilities.

Once these values are known, the A^* algorithm of the previous-section can be run and give a new allocation for the cache. This will provide the system with a good cache allocation for a limited time interval. Clearly, because the A^* algorithm is very expensive to run, one would not like to decide on a new allocation each time a new object is materialized. The solution we propose is to run the A^* algorithm only after some threshold is reached. Such a threshold may be a fixed number of materializations. Another threshold may be the difference between the values for the statistics used to run the allocation algorithm (i.e. reference and update probabilities, sizes of results, etc.) and the actual values observed while the system is running. For instance, if that difference gets above some prespecified percentage of the original estimates, the system may decide to re-run the A^* algorithm.

(b) A different approach is to consider the values of given parameters only and try to approximate the optimal policy with an LRU-like policy. If, for example, we assume that the materialization cost, the size and the probabilities of referencing or updating an object are uniformly distributed over all objects, then LRU will be enough to guarantee a good caching behaviour. The point is that by making the above assumption, the original problem has been reduced to the known page buffering problem in operating systems. However, in the general case LRU will not work since not all the results have the same characteristics, like size, update probability, etc. In that case, we propose the derivation of some experimental formula *rank* $(M_i, u_i, t_i, |R_i|)$ which would rank objects according to the values of their associated parameters, given some weights and scaling factors. The lowest ranked object(s) should be discarded at a point where space is needed. Examples of *rank* are

$$rank(M_i, u_i, t_i, |R_i|) = M_i. \tag{1}$$

The assumption made here is that objects are very expensive to materialize and the rest of the parameters are uniformly distributed. Therefore, objects with low M_i values should be discarded to free space for objects with high M_i values.

$$rank(M_i, u_i, t_i, |R_i|) = \frac{1}{t_i}. \tag{2}$$

In this case objects are expected to be frequently referenced and very rarely updated. Then a pure LRU algorithm based on the times since last reference is a good choice.

$$rank(M_i, u_i, t_i, |R_i|) = \frac{1}{u_i}. \tag{3}$$

If some materialized results are very frequently updated, it may not be worth caching them or, for the purposes of a replacement policy, should be discarded to allow other less frequently updated objects to be cached.

$$rank(M_i, u_i, t_i, |R_i|) = |R_i|. \tag{4}$$

Small objects should be discarded in case larger ones need be cached.

The general format of the *rank* function depends on the application and more specifically on the kind of queries and updates the system supports. In [31] we suggest that one such possible function is

$$rank(M_i, u_i, t_i, |R_i|) = \frac{1}{u_i} \cdot (w_1 M_i + w_2 |R_i|) + w_3 \cdot \frac{1}{t_i} \cdot |R_i|.$$

The specific format was chosen to agree with the formulas derived during the analysis of Sub-section 3.2. The first factor is based on the fact that updates require materialization of objects as well as storing the results in the cache. The second part simply introduces the LRU-like behavior. How to derive the weights w_1, w_2 and w_3 is an interesting open problem and should be attacked through experimentation.

3.4. *Checking the validity of cached objects*

Cached results of materialized procedure entries may become invalid when the relations used to compute these results are modified. Checking the validity of the cached objects amounts to identifying which results are affected from a given update. When such a result R_i is found to be affected, one of two actions can take place:

(a) One can simply invalidate the corresponding entry of the cache. The next query that tries to use the result will find it invalidated and will have to re-evaluate the associated query. This is the scheme assumed in the analysis of the previous sub-section.

(b) One can use the updates performed to the underlying relations and *propagate* them to all cached entries affected by these updates. In this case, some algorithm must be used which, given an update and the query that was used to derive a specific result, will provide a set of update operations that will bring the cached result up to date. Algorithms for similar problems are also described in different contexts in [32, 41–43].

In our environment however, the second approach suffers from two very serious drawbacks. First, between two references to a specific cached result, many updates to underlying relations may be performed. For each of these updates, significant effort will be spent doing propagation of the updates. The second drawback is that updates may be propagated to bring up-to-date entries that *may never* be used in the future. Both problems can be solved by logging all

updates and propagating them at the time a retrieval is performed (batch update) [44]. From the above discussion, it is clear that a good caching scheme will discard these results and replace them with others more frequently used, which makes any effort to propagate updates useless.

We take the approach that entries must be brought up to date *on demand*, that is, the next time the specific entry is requested in a query. Then the system can either *incrementally* propagate the modifications, assuming that we keep the updates in some kind of a log [44], or simply re-evaluate the query. That is an optimization question, and depends on the specific characteristics of the query and the updates. We will not attempt here to discuss in more detail these algorithms.

The rest of this sub-section discusses briefly the problem of detecting which cached results are affected by a given set of updates. [45] presents a detailed discussion of the problem and suggests solutions. The two approaches taken there, *Basic Locking* and *Predicate Indexing*, share the same properties with physical and predicate locking respectively [46] as used in concurrency control. Abstractly, a set of tuples is used to produce the result of some query and our goal is to be able to detect when a given update *conflicts* with this set. Hence, the similarity with the concurrency control problem.

In Basic Locking, all tuples used in processing a given query are marked with a special kind of marker, which contains the identifier Qid of the query. If an index is used for accessing the data tuples, these markers are set on data records *and* on the key interval inspected in the index. Index interval locks are required to deal correctly with insertion of new records (the *phantom problem* in concurrency control). If a new tuple is inserted in one of the relations used to produce the result of a procedure entry, then the collection of markers must be found for the new tuple. To ascertain what collection of cached entries are affected by the insertion of a tuple t, one first collects all the markers on t and then determines which of the corresponding queries are really affected.

In Predicate Indexing, the cache has a specific organization. A data structure is built allowing efficient search of the cache and detection of entries affected by the insertion of a specific tuple in one of the underlying relations. In [45], we suggest that a variation to R-trees [47], R^+-trees [48], are used for that reason. Using Predicate Indexing implies no special treatment of insertions to base relations, but a search of the whole tree is required whenever one asks for the cached entries affected by an update.

Performance analysis results in [45], show that it is not possible to choose one implementation to support efficiently any cache-based environment. Depending on the probability of updating base relations and the number of cached entries that overlap (in the sense that their read sets share some tuples from base relations), the first or the second approach becomes more efficient. Basic Locking seems the most promising because of its ease of implementation, performance in simple environments, and extensibility to join predicates. Analysis of these schemes and investigation of other extensions are a topic of current research.

3.5. Indexing the cache

As a final issue in the caching problem, we touch briefly the problem of indexing the cache. By "indexing" we mean an efficient way to detect if for a given query Q there is a cached result R that is the answer to Q. The problem therefore is to search for Query_expression values in the cache which are *identical* to Q, up to renaming of tuple variables used. In other words, the expressions are identical once we substitute the tuple variables with the names of relations they range over. Checking for identical queries can be done as follows: first, we transform the query to be checked into the canonical form assumed in Sub-section 3.1 and then we test for syntactic matching of the two representations. Recall that a canonical representation must take care of constraint propagation and other well known techniques to make sure that equivalences of expressions can be checked.

However, one does not want to compare all entries of the cache with Q. It is desirable to quickly reject all of the entries that do not relate at all to the given query. This is exactly the reason we have included the field Signature in the cache entries. This field provides high level and easy to check information about the query. The relations involved and the fields that appear in the qualification and the target list of the queries are used to build the signature. If the signatures of a cached entry and the given query match, we can then continue with a more detailed checking, the syntactic comparison of the two canonical representations. To have quick comparison of the signatures themselves, a hash table containing hashed representations of the signatures is used.

4. INDEXING RESULTS OF PROCEDURES

Suppose we are given a relation

EMP(name,salary,mgr,hobbies),

where name, salary, and mgr are conventional fields, while hobbies is a field containing procedure definitions. We use hobbies to retrieve data on the various hobbies of employees. Assume also that the following relations exist in the system

SOFTBALL (name,position,performance)
SOCCER (name,position,goals,performance);

and that EMP contains the tuples

name	salary	mgr	hobbies
Riggs	20	Smith	retrieve (SOFTBALL.position,SOFTBALL.performance) where SOFTBALL.name = "Riggs"
Jones	30	Smith	retrieve (SOFTBALL.position,SOFTBALL.performance) where SOFTBALL.name = "Jones" retrieve (SOCCER.position,SOCCER.performance) where SOCCER.name = "Jones"
Felps	40	Moore	retrieve (SOFTBALL.position,SOFTBALL.performance) where SOFTBALL.name = "Felps"

Imagine now that the following query

```
retrieve (EMP.name) where
  EMP.hobbies.performance < constant
```

is asked very frequently. One would most probably like to build an index on

```
EMP.hobbies.performance
```

in the same way indexes are built on simple attributes. However, there is a difficulty in using conventional indexing schemes to index results of procedures. This would require the materialization of *all* procedures in a relation field and, moreover, materialization must be done when a new tuple is inserted. For example, if a new employee tuple is inserted in the EMP relation, the hobbies field must be processed, the result cached if possible, and the index on EMP.hobbies.performance updated with the new values. This indexing scheme suffers from two serious drawbacks. First, insertion time increases significantly since it is no longer a simple addition of a tuple to a relation, but the execution of (possibly) many queries that constitute the body of the procedure. In particular, in the case of queries involving clauses with multi-dot expressions, response time may increase drastically. Second, by precomputing procedure results, the system blindly materializes *all* objects and therefore spends a lot of time (and space in the cache) in processing procedures that may be never referenced in the future.

We propose here a different approach that overcomes the above problems. The main idea is to have the index reflect only values that have been seen in the past and not all possible ones. This scheme is expected to achieve better performance when the same set of queries is frequently asked. We are also willing to pay some penalty to update the index when that set of queries changes. Such an index on field F contains information on all values of F that appear solely in results of *materialized* entries. These results do not have to exist in the cache; the corresponding values can exist in the index even if the object that included them has since been flushed out of the cache. In these cases, the index simply shows that some procedure results, even though they are not currently cached, *can* produce the specific values stored in the index. Moreover, some extra information is associated with the index; this information characterizes the class of tuples that are indexed. In summary, the indexing scheme proposed is a *partial index* in the sense that it indexes only a part of the relation.

Let us use an example to motivate the discussion on partial indexes that follows. The above relation EMP(name,salary,mgr,hobbies) has an index defined on EMP.hobbies.performance. After some query is processed which results to the materialization of some procedure results, EMP is:

name	salary	mgr	hobbies
Riggs	20	Smith	retrieve (SOFTBALL.position,SOFTBALL.performance) where SOFTBALL.name = "Riggs"
Jones	30	Smith	catcher 4 pitcher 8
Felps	40	Moore	catcher 5 pitcher 4

Assume also that there is a unique tuple identifier *TID* associated with each tuple in the EMP relation, with value 100, 101 and 102 for the first, second and third tuple respectively. These values are stored in the relation but are not visible to the user. The results of the second and third tuples have been materialized and stored in the cache. That is indicated in the above relation by representing them with small relations in the hobbies field of EMP. Suppose the query that has caused that materialization was

```
retrieve (EMP.name)
  where EMP.salary > 20
  and   EMP.hobbies.performance < 6
```

and was processed by scanning EMP and materializing only the hobbies fields of employees with salary more than 20K. The index on

```
EMP.hobbies.performance
```

was of no use because no entries were materialized before the above query was executed. However, after the execution of the query the index was updated to:

salary > 20

performance	TID
4	101
4	102
5	102
8	101

Notice that the above index differs in two ways from conventional indexes. First, there may be more than one performance value for the *same TID* value. This cannot be true in conventional relations because all fields carry a single value (First Normal Form). Second, there is a predicate associated with the index (salary > 20). This predicate uses *only* conventional fields (i.e. no procedure) and is a simple way to partially characterize the kind of tuples indexed by the given index. That predicate is also used to decide if an index is useful in answering a given query. For example, a future query that includes a restriction on EMP.hobbies.performance and references employees with salaries more than xK with $x > 20$, can use the index to avoid a full scan of EMP. However, for $x \leqslant 20$ the relation must be scanned and the entries with salary values under 20 will be materialized. As a side effect, the index table and the corresponding predicate will be updated.

Let us now describe the operation of a partial index. A partial index is a pair $(QUAL,INDX)$, where $QUAL$ is a disjunction of conjunctive one-variable selection clauses and $INDX$ is a conventional index structure. We will say that a qualification $QUAL_1$ *covers* another qualification $QUAL_2$ if the set of tuples satisfying $QUAL_2$ is a subset of the set of tuples satisfying $QUAL_1$, for *any* instance of the database. In any other case we will say that $QUAL_1$ is *not useful* to $QUAL_2$. When an index is requested by a user on a field F which appears in the result of some procedure, a pair $(QUAL,INDX)$ is allocated with initial values $QUAL = false$ and $INDX = \emptyset$. Then depending on the operation performed on the relation, the following actions will take place.

- *Queries that use F in a one-variable clause in the qualification:* Let $QUAL^-$ be the part of the qualification of the query that has no references to procedure results and is composed solely from one-variable clauses on the relation that the index is built on. Then, if the predicate $QUAL$ which is associated with the index covers $QUAL^-$, the query processor may consider using the available index on F for answering the query. If $QUAL$ is not useful to $QUAL^-$, then the query cannot use that index. That index can be used to give only the tuples satisfying $QUAL$ while the rest of the requested tuples must be retrieved from the relation by other means. However, in that case, once the procedures are processed, the values of F are used to update the index and the associated qualification $QUAL$ is changed to $(QUAL \vee QUAL^-)$.
- *Queries that do not use F in any one-variable clause in the qualification but process a procedure that gives F:* In this case, clearly the index is of no use. We take the steps followed in the second case above, that is, the index is updated after the materializations are performed. $QUAL$ remains unchanged.
- *Insertion of a new tuple in the indexed relation:* Given a new tuple to be inserted in the indexed relation, we check if this tuple satisfies $QUAL$ and if so, $QUAL$ is changed to $(QUAL \wedge (\neg QUAL^-))$, where $QUAL^-$ is a qualification that describes the tuple inserted and can be built according to the above discussion. If the tuple does not satisfy $QUAL$ the index remains as is. For example, for a single employee tuple insertion in EMP and for salary 40, $QUAL^-$ will be the *salary* = 40. Using the above algorithm, one avoids inserting the new values in the index. Although this solution is conceptually correct, it is very hard to check whether $QUAL$ covers other predicates if negation is allowed [49]. Simplification of expressions is performed periodically using the algorithm of [49]. Clearly another solution would be to materialize the corresponding procedure and to update the index. However, in this case we may materialize entries that show no indication if they will be used in the future. Recall that this was one of our arguments against pre-materialization of all entries in the beginning of the section.
- *Deletion of a tuple from the indexed relation:* In case of deleting a tuple with tuple identifier *TID*, the entries of the index that contain the same *TID* value are also deleted. Unfortunately, there is no easy solution to updating $QUAL$ in order to reflect the fact that some tuples have been deleted. One way is to change $QUAL$ by introducing negative clauses. However, one value for a field, say salary in our example, may account for more than one occurrence of entries in the index. Therefore, a method based on reference counts should be used in order to make sure that negative clauses are incorporated in $QUAL$ only when no qualifying entries appear in the index.

Updates to base relations may also affect the contents of a partial index. In the case where these updates are affecting results of procedure results, changes may have to occur in the index as well. Using a validation scheme similar to the one used for caching (see Sub-section 3.4) we can check which index entries must be changed after a given update to a base relation.

The above are the only actions required to keep an index up to date. Clearly, the content of the index reflects the dynamics of the system by providing information only on data frequently asked. In that sense, partial indexing is also some kind of *session support* [50], where a user starts up a session and, depending on the queries he/she uses, the system may create secondary structures to speed up common operations. Another comment is that the predicate

QUAL associated with the index may at some point get extremely complicated because of the number of disjuncts it may contain. At such a point the system may use some statistics to estimate the percentage of the tuples that have already been indexed. If that is above a predefined threshold (e.g. 80%), the system may select to materialize and index *all* procedure results. In that case *QUAL* is changed to "*true*". We then arrive at the situation that was discussed in the beginning of the section in which all materialized objects are guaranteed to have an entry in the index table.

Finally, we would like to mention another possible use of partial indexes. Many times users issue all their queries through specific views that they have defined over base relations. Users are not allowed to keep materialized versions of the views in the system because of its high space cost, but they still would like queries to execute fast. Indexes on base relations will be helpful for that. However, these indexes contain more information than what these users need, namely an index *only on the result of the view materialization.* A partial index seems like a clean solution to that problem. The *QUAL* part will be static since it will be the predicate that defines the view, but querying and updating will be performed under the guidelines outlined above. This idea can also be extended to normal relations, since these are special cases of views. Using partial indexes, better performance can be achieved by allowing the index to keep information only on frequently accessed data.

5. SUMMARY

This paper first presented a motivation for extended query languages and suggested intelligent mechanisms to improve the performance of query processing. First, caching was proposed as a way to avoid evaluating the queries found in database procedures more than once. Several issues associated with caching were discussed. Among others, replacement policies, invalidation algorithms and policies that decide which objects to cache were examined in detail. The discussion shows that caching is essential in such environments and various solutions to the above problems can be derived once the cached object characteristics are known. How to compute these characteristics and how to adapt the system caching policies according to these statistics is a very interesting open problem.

Second, a new indexing technique, Partial Indexing, was proposed to provide efficient access to results of procedure results. A partial index is a combination of both a conventional index table and a predicate. Predicates characterize the set of tuples that can be accessed through the corresponding index tables. We also described how the system can check if an index is useful in processing a given query and what are the necessary operations to maintain a partial index when queries and updates are performed.

As interesting future work in that area, we plan to implement and experiment with the ideas presented in this paper. We are also currently investigating the efficient support of partial indices not only for extended relational systems but for a conventional environment as well. This is useful especially in cases where, for performance reasons, the user may request the creation of partial indexes to reflect skewed distributions of data. For example, if one knows that most of its values lie in an interval (10, 20), an index will be really helpful for such an interval. The rest of the values will not play a key role since they will be very rarely referenced in queries. Caching and partial indexes are a way of reflecting the system dynamics. We believe that there is a lot of work that can be done in the area trying to build relational systems that can nicely adapt to changing query distributions.

Acknowledgements—I wish to thank my advisor Professor Michael Stonebraker for giving me the opportunity to work in the area of database procedures. This research was sponsored by the U.S. Air Force Office of Scientific Research under Grant 83–0254 and by the National Science Foundation under Grant DMC–8504633 while the author was with the University of California at Berkeley.

REFERENCES

[1] G. Copeland and D. Maier. Making Smalltalk a database system. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1984).

[2] N. P. Derrett *et al.* An object-oriented approach to data management. *Proc. IEEE Spring Compcon Conf.* (1986).

[3] M. Jarke, J. Clifford and Y. Vassiliou. An optimizing PROLOG front-end to a relational query system. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1984).

[4] J. Ullman. Implementation of logical query languages for data bases. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1985).

[5] C. Zaniolo. The representation and deductive retrieval of complex objects. *Proc. 11th Int. Conf. on Very Large Data Bases* (1985).

[6] L. Kerschberg (Ed.) *Proc. First Int. Workshop on Expert Database Systems* (1984).

[7] L. Kerschberg (Ed.) *Proc. First Int. Conf. on Expert Database Systems* (1986).

[8] D. S. Batory *et al.* GENESIS: a reconfigurable database management system. University of Texas at Austin, Technical Report TR-86-07 (1986).

[9] M. Carey *et al.* Object and file management in the EXODUS extensible database system. University of Wisconsin at Madison, Technical Report (1986).

[10] C. Mohan. STARBURST: an extensible relational DBMS. Panel discussion on extensible database systems. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1986).

[11] M. Stonebraker and L. Rowe. The design of POSTGRES. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1986).

[12] D. S. Batory and W. Kim. Modeling concepts for VLSI CAD objects. *ACM Trans Database Systems* 10 (3) (1985).

[13] A. Guttman. New features for relational database systems to support CAD applications. Ph.D. Thesis, University of California, Berkeley (1984).

[14] C. Zaniolo. The database language GEM. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1983).

[15] K. Morris *et al.* Design overview of the NAIL! system.

Stanford University, Technical Report STAN-CS-86-1108 (1986).
[16] Y. Ioannidis. On the computation of the transitive closure of relational operators. *Proc. 12th Int. Conf. on Very Large Data Bases* (1986).
[17] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In [7].
[18] F. Bancillhon and R. Ramakrishnan. An amateur's introduction to recursive query processing. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1986).
[19] Y. Ioannidis. Processing recursion in deductive database systems. Ph.D. Thesis, University of California, Berkeley (1986).
[20] J. Grant and J. Minker. Optimization in deductive and conventional relational database systems. In *Advances in Data Base Theory*, Vol. 1 (Edited by H. Gallaire, J. Minker and J.-M. Nicolas). Plenum Press, New York (1981).
[21] T. Sellis and L. Shapiro. Optimization of extended database languages. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1985).
[22] T. Sellis. Multiple-query optimization. *ACM Trans. Database Systems*, **13**(1) (1988).
[23] J. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, MD (1982).
[24] R. Kooi and D. Frankfurth. Query optimization in INGRES. *Database Engng* **5**(3) (1982).
[25] P. Selinger *et al.* Access path selection in a relational data base system. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1979).
[26] E. Wong and K. Youssefi. Decomposition: a strategy for query processing. *ACM Trans. Database Systems* **1**(3) (1976).
[27] U. Dayal and K. Dittrich (Eds). *Proc. 1986 Int. Workshop on Object-Oriented Database Systems*. IEEE Computer Society Press, Washington, DC (1986).
[28] M. Stonebraker *et al.* Extending a data base system with procedures. University of California, Berkeley, Technical Report UCB/ERL/M85/59 (1985).
[29] M. Stonebraker *et al.* The design and implementation of INGRES. *ACM Trans. Database Systems* **1**(3) (1976).
[30] M. Stonebraker *et al.* Quel as a data type. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1984).
[31] T. Sellis. Optimization of extended relational database system. Ph.D. Thesis, University of California, Berkeley (1986).
[32] R. Kung *et al.* Heuristic search in data base systems. In [6].
[33] R. L. Mattson *et al.* Evaluation techniques for storage hierarchies. *IBM Systems J.* **9**(2) (1970).
[34] S. Finkelstein. Common expression analysis in database applications. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1982).
[35] P. Larson and H. Yang. Computing queries from derived relations. *Proc. 11th Int. Conf. on Very Large Data Bases* (1985).
[36] K. M. Chandy. Models of distributed systems. *Proc. 3rd Int. Conf. on Very Large Data Bases* (1977).
[37] N. Roussopoulos. View indexing in relational databases. *ACM Trans. Database Systems* **7**(2) (1982).
[38] N. Roussopoulos. The Logical Access Path Schema of a Database. *IEEE Trans. Software Engineering* **8**(6) (1982).
[39] H. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Proc. 11th Conf. on Very Large Data Bases* (1985).
[40] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Trans. Database Systems* **9**(2) (1984).
[41] M. E. Adiba and B. G. Lindsay. Database snapshots. *Proc. 6th Int. Conf. on Very Large Data Bases* (1980).
[42] J. A. Blakeley, P. Larson and F. W. Tompa. Efficiently updating materialized views. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1986).
[43] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *ACM Trans. Database Systems* **4**(3) (1979).
[44] N. Roussopoulos and H. Kang. Preliminary design of ADMS±: a workstation-mainframe integrated architecture for database management systems. *Proc. 12th Int. Conf. on Very Large Data Bases* (1986).
[45] M. Stonebraker, T. Sellis and E. Hanson. Rule indexing implementations in data-base systems. In [7].
[46] J. N. Gray. Notes on data base operating systems. IBM Research, Technical Report RJ-2254 (1978).
[47] A. Guttman. R-Trees: a dynamic index structure for spatial searching. *Proc. ACM-SIGMOD Int. Conf. on the Management of Data* (1984).
[48] T. Sellis, N. Roussopoulos and C. Faloutsos. The R^+-tree: a dynamic index for multi-dimensional objects. *Proc. 13th Int. Conf. on Very Large Data Bases* (1987).
[49] D. J. Rosenkrantz and H. B. Hunt. Processing conjunctive predicates and queries. *Proc. 6th Int. Conf. on Very Large Data Bases* (1980).
[50] S. Kuck. Private communication. University of Illinois, Urbana (1986).