TR-87-33

Efficiently Supporting Procedures in Relational Database Systems

by

Timos K. Sellis

EFFICIENTLY SUPPORTING PROCEDURES IN RELATIONAL DATABASE SYSTEMS

by
Timos K. Sellis

Department of Computer Science
and Systems Research Center
University of Maryland
College Park, MD 20742

To be presented at the 1987 ACM International Conference on the Management of Data,

San Francisco, CA, May 1987.

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 83–0254 while the author was at the University of California, Berkeley, and in part by the National Science Foundation under Grant CDR–85–00108.

EFFICIENTLY SUPPORTING PROCEDURES IN RELATIONAL DATABASE SYSTEMS

by
Timos K. Sellis

Department of Computer Science
and Systems Research Center
University of Maryland
College Park, MD 20742

Abstract

Recent developments in the design of database systems include proposals of several extensions to the basic model of relational database systems. Such a system is presented in this paper. The system is based on QUEL+ which extends the query language QUEL by introducing among others database procedures as full fledged objects. We briefly describe a variation of the original INGRES decomposition algorithm and then some ideas that can improve the performance of a system that supports procedures are discussed. First, we examine the idea of storing results of previously processed procedures in secondary storage (*caching*). Using a cache, the cost of processing a query can be reduced by preventing multiple evaluations of the same procedure. Problems associated with cache organizations, such as replacement policies and validation schemes are examined. Another means for reducing the execution cost of queries is indexing. Indexing results of procedures can be done through conventional schemes. However, at any given time, it is highly probable that not all procedures stored in a relation have been evaluated which makes known indexing techniques useless, since the latter assume that all values are known. As a solution to that problem, a new indexing scheme, Partial Indexing, is proposed and analyzed.

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 83-0254 while the author was at the University of California, Berkeley, and in part by the National Science Foundation under Grant CDR-85-00108.

1. Introduction

Recent developments in the design of database systems include proposals of several extensions to the basic model of relational database systems. Systems based on the Object Oriented Programming paradigm [DERR86,COPE84], systems that provide support for storing both knowledge and data [ZANI85,ULLM85,JARK84] (also called Expert Database Systems [KERS84,KERS86]) and finally systems that have certain extensibility capabilities [CARE86,BATO86,MOHA86,STON86b], 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 [GUTT84,KERS84,BATO85,KERS86].

Clearly, all proposals will need some kind of extended relational query language to support a high level user interface. Examples of such languages are GEM [ZANI83], POSTQUEL [STON86b] and DATALOG [MORR86]. Because of their extended capabilities, such languages need special support for the efficient execution of queries. For example, languages supporting recursive queries need specialized algorithms to process recursion [BANC86,IOAN86a]. Query processing algorithms need to be modified in light of all such extensions. The purpose of this paper is to discuss some query processing issues that arise in extended relational database systems. Although we concentrate on a specific query language which is an extension to QUEL [STON76], the discussion should apply to other languages as well.

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 [GUTT84,IOAN86b,VALD86], general recursive queries [BANC86,IOAN86a], deduction [GRAN81,ULLM85], database procedures [SELL85,SELL86c], etc. Physical and conceptual modeling, concurrency control and crash recovery are some of other well known DBMS problems [ULLM82]. 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. The goal of this paper is to examine ways of improving the performance by providing more sophisticated optimization tactics. More specifically, 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 [STON86b] 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 presents the language QUEL+ and motivates its use with a couple of examples. Then, in section 3 we briefly examine the problem of query processing by presenting a variation to the INGRES decomposition algorithm [WONG76] along with some possible improvements. Sections 4 and 5 present the main contributions of this paper. We focus on schemes that improve the performance of the system, like caching and indexing. Finally, we conclude in section 6 by summarizing the discussion of this paper and pointing out interesting future research problems.

2. The Query Language QUEL+

QUEL+ [STON85] is an extension to QUEL, the query language designed for INGRES [STON76]. There are two major extensions made to QUEL:

a) repetitive execution of commands, and

b) storing query language commands in relation fields

The first extension allows the user to implement iteration using the query language itself instead of escaping to a general purpose programming language. In EQUEL/C [ALLM76] for example, the programmer can embed INGRES commands in C [KERN78] programs and therefore can implement iteration through the iterative constructs of C. The second feature follows the paradigm of LISP [WILE84] and allows the uniform treatment of data and control information, or procedures in [STON85], where the latter is implemented using database commands. Stonebraker *et al* give in [STON85] a detailed discussion of the language. We review here the second extension since it will serve as the basis of our presentation.

It was first proposed in [STON84] that QUEL commands be stored in relation fields in the same way data is stored in relations. For simplicity, these fields are thought as variable length strings. In INGRES, relation fields can be accessed individually through the dot (.) operator. For example, given a relation `EMP (name,salary,mgr)`, with the

obvious meanings for the three fields, EMP.mgr accesses the manager names recorded in EMP. Extending these semantics, it will be assumed that accessing a relation field containing QUEL commands (*QUEL field*) implies the *execution* of the commands that are stored in the field.

Processing QUEL fields amounts to evaluating the commands that are stored in these fields. The problem of efficiently evaluating the contents of QUEL fields has been studied in [SELL85] and [SELL86b] respectively. Here, our focus is the problem of processing QUEL+ queries. Before we proceed to discuss that problem however, we present an example of QUEL+.

Consider, a relation **EMP (name,salary,mgr,hobbies)** where *name, salary* and *mgr* are conventional fields while *hobbies* is a field of type QUEL. 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)
MUSIC    (name,instrument,performance)
```

Assume also the following instance of the EMP relation

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"
Lam	80	Moore	retrieve (MUSIC.all) where MUSIC.name = "Lam"
..	....	....	.....

The QUEL syntax is extended using the *multiple dot* notation borrowed from Zaniolo's GEM language [ZANI83,ZANI84]. For example, one can retrieve the performance of Jones in all his hobbies as follows:

```
retrieve (EMP.hobbies.performance)
    where EMP.name = "Jones"
```

The number of dots that can be used depends on the relation nesting level. With the use of the multiple dot notation, QUEL+ allows the user to actually "navigate" through relations using QUEL fields as links between the accessed tuples.

Clearly, the result of evaluating ("*materializing*") a QUEL field is a set of relations, or in general a set of tuples. These sets are themselves database objects (relations). QUEL+ provides relation level operators allowing a user to usee set operations as well (such as set equality, set inequality, union, intersection, etc). For example, one may wish to get all pairs of employees that play in the same positions and with the same performance in their hobbies. The above query can then be formulated as

```
range of EMP,EMP1 is EMP
retrieve (EMP.name,EMP1.name)
    where EMP.name ≠ EMP1.name
    and   EMP.hobbies == EMP1.hobbies
```

where == is the set equality operator. Issues involved in the implementation of such operators are discussed in more detail in [SELL86b].

After reviewing the structure and semantics of QUEL+, we now examine the problem of query processing.

3. Processing QUEL+

This section presents a query processing algorithm that INGRES can use to evaluate QUEL+ queries. First, it discusses how the original decomposition algorithm of Wong and Youssefi [WONG76] was extended to handle queries in relation fields and the extended relation level operators. Then, some possible improvements are suggested and explained through examples. A complete discussion of the algorithm can be found in [SELL86c].

3.1. Extended Decomposition

Figure 1 shows a diagram of the extended decomposition algorithm as suggested in [STON85]. The modifications done to the original Wong–Youssefi algorithm can be summarized as follows

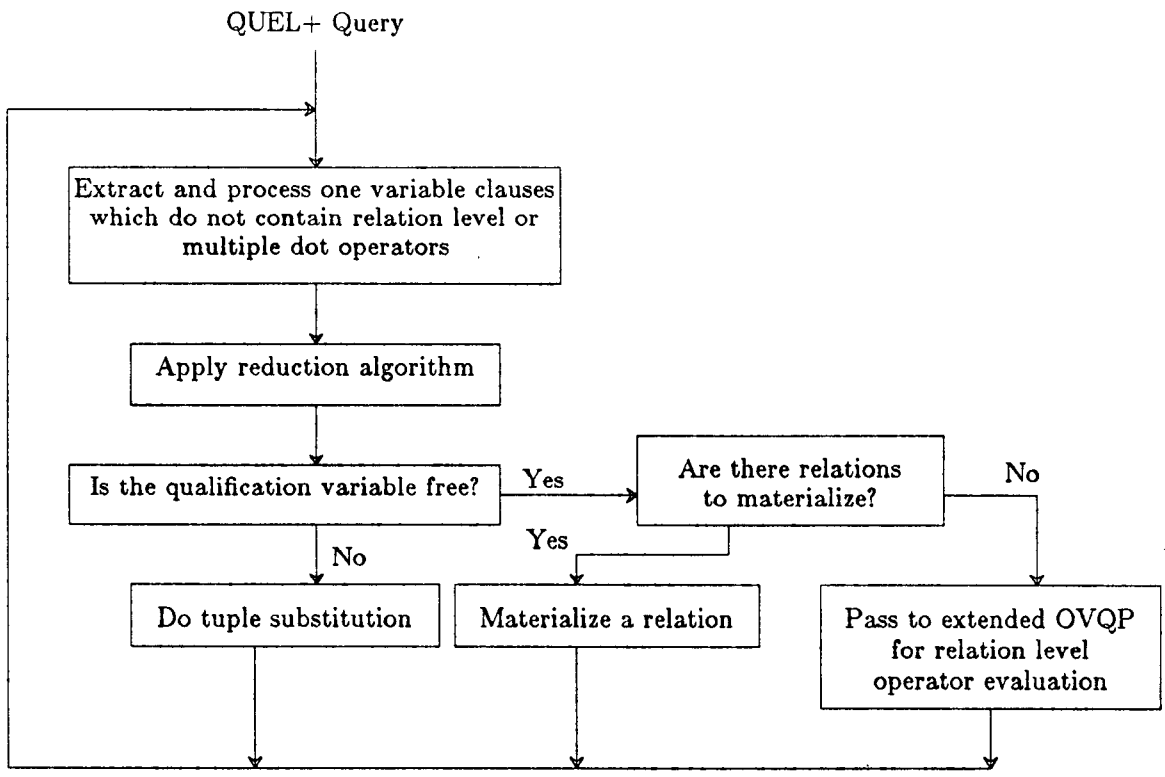

Figure 1: Extended Decomposition Strategy
(OVQP : *One Variable Query Processor*)

a) All one-variable clauses except those that include a multiple dot reference or a relation level operator are processed first. The reason is that clauses involving extended operators cannot be processed efficiently. For example, none of the following two clauses

```
EMP.hobbies.position = "catcher"
```

or

```
EMP.hobbies == some_constant_relation
```

should be processed first because that would imply the materialization of the *hobbies* entries of *all* employees, which is very expensive. An exception to that is the case where an index exists on `EMP.hobbies.position`. This case is discussed in more detail in section 5.

b) An extra step is required to check if all QUEL field entries have been materialized. Materialization is done by passing the queries found in the QUEL field to a second INGRES process which in turn returns the result relation(s). The decomposition algorithm continues processing one–variable clauses and materializing QUEL fields until no more such fields are left unevaluated.

c) In [WONG76] the criterion for selecting a relation to iterate over in the case of tuple substitution, is the size of the relations. The presence of QUEL fields makes this criterion ineffective. Not only the number of tuples but the cost for materializing the corresponding QUEL fields should be considered. The reason is that during tuple substitution, each tuple variable will be replaced with specific field values read from the relation. In case of QUEL fields these values are the materialization results. Therefore the criterion for selecting a relation to iterate over will generally be a function of the size of the relation and the characteristics of the materialized objects. One of these characteristics which is of major importance is the ability of the system to keep materialized objects in secondary storage, i.e. caching. This aspect is treated in more detail in section 4.

The extended decomposition algorithm delays materializing a QUEL field until there is nothing else that the conventional query processor can do. Even tuple substitution must be done first, the reason being that checking a condition that involves multiple dot references implies a loop over all tuples in the relation. During that loop QUEL fields are materialized and checked through lower level fields. Generally, the absence of any information about the contents of relations in QUEL fields makes optimization very hard, if not impossible. In the next section we discuss one possible improvement through saving the results of materializing QUEL fields (*caching*); in this case, the contents of QUEL fields are known and conventional cardinality estimation methods [SELI79] can be used to estimate the cost of the various processing strategies. However, before moving to caching we suggest some other possible improvements that apply directly on the algorithm itself.

3.2. Improvements to Extended Decomposition

In this subsection some possible improvements to the algorithm presented above are examined. First, we give some rules that can be applied in general; then, some other special case transformations that can be used are outlined.

The first general rule as, suggested above, is to process one–variable clauses and do reduction as the initial Wong–Youssefi algorithm proposes [WONG76]. This will certainly be the best thing to do independent of the number of relations or QUEL field materializations that will follow. The problem arises when tuple substitution is necessary. We motivate our proposal using an example.

Assume that in the EMP relation the *hobbies* field produces a relation, which itself has a field *performance* that also produces a relation as a result and the field we are interested in is the *location* field of that last relation. We also assume the existence of another relation DEPT (name,mgr,location). The query is

```
retrieve (EMP.name,DEPT.name)
    where EMP.hobbies.performance.location = DEPT.location
    and   EMP.mgr = DEPT.mgr
```

The question that arises here is over which relation to iterate doing tuple substitution. The main idea behind tuple substitution is to introduce single variable selection clauses as early as possible. Using such clauses relation sizes are reduced and, consequently, the number of materializations that will be needed is also lower. For example, in the above query tuple substitution should be done over DEPT independently of the sizes of the two relations. The reason is that tuple substitution will create some one–variable clauses which can then be used to restrict the number of EMP tuples that need be considered for materialization of their fields. In general, an algorithm that selects a relation to iterate over, attempts to minimize the total number of tuple substitutions required, assuming the most expensive processing lies in QUEL field materializations. Such an algorithm is formally discussed in [SELL86c].

We now describe a different technique that can be used to improve the performance of the query processor in some special cases. The basic idea is that when an entry from a QUEL field is materialized, the query that has to be processed next is known. More

specifically, the structure of the query is known and through that the optimizer can identify access structures that may be desirable in order to speed up processing. For example, in the query

```
retrieve (EMP.name,DEPT.name)
    where EMP.hobbies.performance.average = 10
    and   EMP.mgr = DEPT.mgr
    and   EMP.hobbies.leader = DEPT.mgr
```

the query processor will choose to tuple substitute over DEPT, and after detaching one-variable clauses and substituting EMP with TEMP, the new query will be

```
retrieve (TEMP.name,constant-1)
    where TEMP.hobbies.performance.average = 10
    and   TEMP.hobbies.leader = constant-2
```

At this point the query processor will start materializing entries from the *hobbies* field of TEMP. Let TEMP1 be the result of materializing a specific entry of *hobbies*; then the type of queries that will have to be processed for each TEMP tuple will be

```
retrieve (constant-2,constant-1)
    where TEMP1.performance.average = 10
    and   TEMP1.leader = constant-2
```

From that last query one can observe that depending on the size of TEMP1 it may be beneficial to build a secondary index on *leader* so that the second qualification clause can be processed efficiently. This structure will be built in the process of producing TEMP1 (*on the fly*) and no extra time need be spent at the time the query will be evaluated. Dynamic creation of indexes or imposing other structures on relations (like sorting) has also been used in conventional query processing [YOUS78,KOOI82]. However, a difference is that in the QUEL+ environment no significant additional cost need be spent on creating the index. At the same time a result of a materialization is produced and stored in a temporary relation, some adequate organization is chosen or a secondary indexing structure is built.

In the same spirit we describe another optimization technique that can be used to reduce the cost of processing a query. Clearly, one wants to materialize QUEL fields and produce results that will be used subsequently in the course of processing a given query. However, in some cases, not all queries stored in QUEL fields will give relevant information. For example, consider the relation EMP (name,salary,mgr,hobbies) of the

previous section, and the query

```
retrieve (EMP.name)
    where EMP.hobbies.instrument = "violin"
```

When the various entries in the *hobbies* field are materialized, only those queries that involve in their result a field *instrument* should be evaluated. In our example, the queries that retrieve data from the SOFTBALL and SOCCER relations should not be evaluated. Moreover, even if the query in *hobbies* retrieves many fields from the MUSIC or any other relation that includes a field *instrument*, the contents of the materialized relations should be restricted to contain only the information that is absolutely necessary, in this case the *instrument* field. We should also notice here that the same idea exists in conventional query processing as well in the form of projections made to restrict sizes of intermediate results [WONG76,SELI79].

The above technique tries to reduce the amount of space required for storing materialized objects. However, there are some cases where *no* space at all need be allocated for materialization. This is the case where a QUEL field contains a single **retrieve** or **define view** command. In this special (but very common case) there is no need to even produce the result of the command. Conventional query modification [STON75] can be used. For example, consider the following query

```
retrieve (EMP.hobbies.position)  where  EMP.hobbies.average ≤ 5
```

where the *hobbies* field of the EMP relation contains one of the following QUEL expressions

```
retrieve (SOCCER.all) where SOCCER.name = constant
```

or

```
retrieve (SOFTBALL.all) where SOFTBALL.name = constant
```

i.e. all employees have at most one hobby. Then the given query can be transformed to

```
retrieve (REL.position)
    where REL.average ≤ 5
    and   REL.name = constant
```

where REL is either SOCCER or SOFTBALL. This transformation not only prevents the query processor from materializing relations, but it also allows the optimizer to have more information on the structure of the query, and therefore to process it with a better

access plan. It is also possible to generalize this technique to handle multiple statements [SELL85b].

This concludes our presentation of the extended decomposition algorithm for processing QUEL+ queries. In addition to the basic algorithm, we presented some less general tactics that can be used to improve the performance of the query processor. In the two sections that follow two other issues that are of significant importance to query processing are discussed, namely *caching* and *indexing* of the results of QUEL fields.

4. Caching Materialized QUEL Fields

As it was seen in the previous section, materializing an entry of type QUEL amounts to executing, possibly several, QUEL queries. Hence, it will be generally very slow to perform this operation every time a QUEL field is accessed. This section examines ways to make QUEL+ processing more efficient through the use of a cache.

4.1. What is Caching?

We mentioned at several points in the previous sections that one way to avoid evaluating the same QUEL field entries multiple times, is *caching*. By caching we mean computing the values of QUEL fields and storing them 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 QUEL field entries since it occurs before even the content of the specific field is accessed. However, our focus here is on the latter case which is more natural. The basic idea is to keep in secondary storage 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 [MATT70]. Notice also, that the cache can be used not only for materialized QUEL fields 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 [FINK82,LARS85,SELL86a].

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

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 object?*

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

```
(Qid,Query_expression,Result)
```

are stored. `Qid` is some unique identifier, `Query_expression` is some canonical representation for queries, e.g. query graphs [WONG76], and `Result` is the relation resulting after executing the query or set of queries that were found in some QUEL field and described by the second field (`Query_expression`). The following four subsections give answers to each of the above mentioned questions (a) through (d).

4.2. Which Query Results to Cache?

Depending on the information known about the queries, the system can decide whether a result is worth caching it or not. For a given materialization result R, this decision will be generally 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, Table 1 shows the list of parameters to the caching problem **C** is the number of disk pages allocated for the cache. r_i and u_i are the probabilities of referencing and updating respectively a result R_i. M_i is the cost of materializing the QUEL field that gives the result R_i while S_i and U_i are the costs of writting to and reading from the cache R_i respectively. Finally, it will be assumed that invalidating an 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 $C=\infty$ 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. Since each object is exam-

Caching Problem Parameters	
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 writting 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

Table 1: Caching Problem Parameters

ined individually, it will be $u_i + r_i = 1$. The criterion is based on comparing the cost of processing R_i 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 C_i respectively. In the case where no caching is used, the result must be produced at each reference by materializing the corresponding QUEL field. 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 C_i we will differentiate between the following four cases for the types of two subsequent requests:

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 object must be re-materialized 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

$$C_i = r_i\ u_i\ IN + r_i\ r_i\ U_i + u_i\ u_i\ IN + u_i\ r_i\ (M_i + S_i)$$

or, since $r_i + u_i = 1$,

$$C_i = u_i\ IN + r_i\ [r_i\ U_i + u_i\ (M_i + S_i)]$$

Comparing now C_i and NC_i we can identify the cases where it is worth caching result R_i. That happens when $NC_i > C_i$. Using the formulas extracted above, we can see that this is true if

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

Checking the above condition will determine if the result of a given QUEL field 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. Hence, in this case **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 ground relation, it cannot any more be assumed 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 do with space limitations. This restriction indicates that the total space occupied by cached results cannot be more than **C**. Given all these parameters we formulate now the problem of caching in the case of Bounded Space.

Let $A: \mathbb{N} \rightarrow \{0,1\}$ 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

$$BC_i = \begin{cases} C_i & \text{if } A(i)=1 \\ NC_i & \text{if } A(i)=0 \end{cases}$$

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

$$\sum_{i=1}^{N} BC_i \qquad \text{is minimal} \tag{C1}$$

subject to the constraint

$$\sum_{i=1}^{N} A(i)|R_i| \leq \mathbf{C} \tag{C2}$$

This problem of optimal allocation has been shown to be NP–complete (see [CHAN77] for a similar problem). However, almost identical constraints have to be satisfied in the *view indexing* problem that Roussopoulos examined in the context of improving the performance of view based queries [ROUS82a,ROUS82b]. In [ROUS82a], he defines a state model to formulate the above allocation problem and then gives an A* algorithm [RICH83] that finds a near–optimal allocation. We will not go here into the details of that algorithm; the reader is referred to [ROUS82a] for a rigorous and detailed presentation of the technique.

The output of the A* algorithm identifies which results are worth keeping in the cache. Hence, this approach is meaningful only in the case where all QUEL fields are

materialized in advance and a decision is made on which of them should be cached. Clearly, that policy may not be always 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 though, such a solution is undesirable. The next subsection provides some insight for better approaches.

In summary, the above two cases shared the fact 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. We will not present here a special algorithm for the case where no statistics are available. The following subsection discusses that issue in the context of the replacement policies that can be used for the cache.

4.3. 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 $<R_1,R_2,\cdots,R_n>$ 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 updating R_i
M_i	Cost of producing R_i (materialization)
$\|R_i\|$	Size of R_i

and that the cost of writting and reading an object from the cache is equal to the size of that object. Let **S** and **R** be the set of all possible states and results to cache, respectively. Then, a *replacement policy* P, is a function $P: \mathbf{S}\times\mathbf{R}\rightarrow \mathbf{S}$ 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 whole (past and future) pattern of references to decide on which pages should be cached (see algorithm OPT in [MATT70]). This algorithm is not practical though, unless one can predict with high probability the future behaviour of the system. The closest approximation is the LRU (Least Recently Used) algorithm which selects to discard the object with maximum time since last reference. In the area of database management systems, the same policy can be used in the design of buffer managers. DeWitt and Chou give in a recent article [CHOU85] 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 this 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 ground relations are given. 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 ground 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 it is known that either nested loops or merge scan will be used, we can predict the way QUEL fields are accessed, and therefore have a rough estimate for the needed probabilities.

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. However, in the general case LRU will not work. 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

(1) $rank(M_i,u_i,t_i,|R_i|) = M_i$

Objects with low M_i values should be discarded to free space for objects with high materialization costs.

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

Pure LRU algorithm based on the time since last reference.

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

Very frequently updated results are not worth caching.

(4) $rank(M_i,u_i,t_i,|R_i|) = |R_i|$

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

Trying to generalize *rank* by combining all four functions we suggest the following function for *rank*

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

This formula is the simplest one that can be devised and incorporates in an easy way the effects of the various parameters. The specific format was chosen to agree with the formulas derived during the analysis of section 4.2. 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 behaviour. How to derive the weights w_1, w_2 and w_3 is an interesting open problem and should be attacked through extensive experimentation.

4.4. Checking the Validity of Cached Objects

Cached results of materialized QUEL field 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 subsection.

b) One can use the updates performed to the underlying relations and *propagate* them to all cached entries affected by these updates. Update propagation algorithms are described in various articles and in different contexts [BUNE79,ADIB80,KUNG84,BLAK86].

In our environment however, the second approach suffers from two very serious drawbacks. First, it is the case that between two references to a specific cached result many updates to underlying relations may be performed. Clearly, for each of these updates significant effort will be spent doing propagation of the updates. The second drawback is due to the fact that updates may be propagated to bring up to date entries that *may never* be used in the future. Clearly, 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 [ROUS86], or simply re–evaluate the query. That is an optimization question and depends on the specific characteristics of the query and the updates.

Finally, the problem of detecting which cached results are affected by a given set of updates must be addressed. [STON86a] presents a detailed discussion of the problem and the proposed solutions. Two approaches to solve the problem are described in detail. One is based on physical locking of data involved in queries with cached results (*Basic Locking*) while the other one is based on checking predicate expressions affected by the updates (*Predicate Indexing*). These schemes share the same properties with physical and predicate locking respectively [GRAY78,ESWA76] as used in concurrency control. The

interested reader is refered to [STON86a].

Performance analysis results in [STON86a], show that it is not possible to choose one implementation to support efficiently any cache based environment. Depending on the probability of updating ground relations and the number of cached entries that overlap (in the sense that their read sets share some tuples from ground 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 future research.

This last subsection concludes our presentation on caching results of QUEL fields. A working version of extended INGRES has a very simplified cache which performed very well in the experiments of [STON85]. POSTGRES [STON86b] will be supported by a more sophisticated caching scheme which will use LRU for replacement and Basic Locking for checking the validity of the entries.

5. Indexing Results of QUEL Fields

Imagine a query that is frequently asked and has the following form

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

One would most probably like to build an index on `EMP.hobbies.average` in the same way indexes are built on simple attributes. However, there is a difficulty in using conventional indexing schemes to index results of QUEL fields. This would require the materialization of *all* entries in the QUEL field and, moreover, materialization must be done when a new tuple with a QUEL field 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.average` must be 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 in a relation, but the execution of (possibly) many queries as well, the ones stored in QUEL fields. In particular, in the case of queries involving clauses with multi–dot expressions, response time may increase drastically. Second, by precomputing QUEL field entries the system materializes

all objects and therefore spends a lot of time (and possibly space in the cache) in processing field entries that may be never referenced in the future.

Another proposal that overcomes the above problems is presented here. The main idea is to have the index reflect only values that have been seen in the past and not all possible ones. Through this scheme, it is expected to achieve better performance in cases where the same set of queries is frequently asked. We are also willing to pay some penalty to update the index in the case where the set of queries changes. Given a field, the structure to be described, contains information on all values of that field that appear solely in results of *materialized* entries. These results do not have to exist in the cache; they can exist in the index even if the object that included them has been flushed out of the cache. In these cases, the index simply shows that some QUEL fields, even if not currently materialized, can produce the specific values stored. Moreover, some extra information is associated with the index; information that 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 relation `EMP (name,salary,mgr,hobbies)` of section 2 has an index defined on `EMP.hobbies.average`. Figure 2 indicates the tuples that are currently in `EMP` 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 `EMP` 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 stored 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.average < 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.average` 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 as shown in Figure 3. Notice that the

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

Figure 2

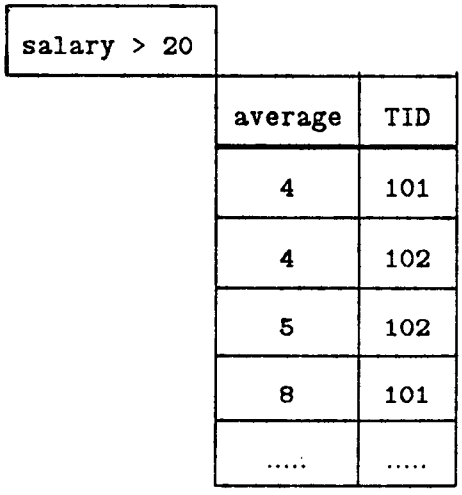

Figure 3

above index differs in two ways from conventional indexes. First, there may be more than one *average* values for the *same TID* value. This cannot be true in conventional

relations because all fields carry a single value (First Normal Form [ULLM82]). Second, there is a predicate associated with the index (`salary > 20`). This predicate uses *only* non–QUEL fields and is a simple way to identify 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.average` and references employees with salaries more than x, with $x>20$, can use the index to avoid a full scan of `EMP`. However, for $x\leq 20$ 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.

In [SELL86c] we present in detail the operations on a partial index (e.g searching or updating the index). We will not go into the details here due to space limitations. Instead, 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 ground 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 ground 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 predicate 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.

6. Summary

This paper first presented the language QUEL+ and its capabilities. Then, an extended decomposition algorithm based on the INGRES query processing algorithm was proposed. The extensions made were mainly due to the fact that one new operation was introduced, namely the materialization of QUEL fields. We showed how the fact that materialization is a very expensive operation is taken under account. Also, some special

case strategies were discussed that aim to reducing the sizes of materialized results.

Caching was then proposed as a way to avoid evaluating the queries found in QUEL fields 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 the QUEL+ environment 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.

Lastly, a new indexing technique, Partial Indexing, was proposed to provide efficient access to results of QUEL field materializations. 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 view the attempt to implement and experiment with the ideas presented in this paper. As mentioned in section 4, POSTGRES sill support QUEL+ and caching will be used to improve performance. Simulations are under development for an analysis of the various caching scheme alternatives. Finally, we are currently investigating the efficient support of partial indices not only for a QUEL+ environment but for a conventional environment as well.

Acknowledgements: I would like to thank my advisor Prof. Michael Stonebraker for giving me the opportunity to work in the area of database procedures. Also, I would like to acknowledge the Systems Research Center of the University of Maryland for its partial support through the National Science Foundation Grant OIR–85–00108.

7. References

[ADIB80] Adiba, M.E. and Lindsay, B.G., *"Database Snapshots"*, Proceedings of the 6th International Conference on Very Large Data Bases, Montreal, October 1980.

[ALLM76] Allman, E. et al, *"EQUEL Reference Manual"*, University of California, Technical Report UCB/ERL, Berkeley, CA, 1976.

[BANC86] Bancillhon, F., and Ramakrishnan, R., *"An Amateur's Introduction to Recursive Query Processing"*, Proceedings of the 1986 ACM–SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[BATO85] Batory, D.S. and Kim, W, *"Modeling Concepts for VLSI CAD Objects"*, ACM Transactions on Database Systems, (**10**) 3, September 1985.

[BATO86] Batory, D.S., et al, *"GENESIS: A Reconfigurable Database Management System"*, University of Texas at Austin, Technical Report TR–86–07, Austin, TX, March 1986.

[BLAK86] Blakeley, J.A. , Larson, P. and Tompa, F.W., *"Efficiently Updating Materialized Views"*, Proceedings of the 1986 ACM–SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[BUNE79] Buneman, O.P. and Clemons, E.K., *"Efficiently Monitoring Relational Databases"*, ACM Transactions on Database Systems, (**4**) 3, September 1979.

[CARE86] Carey, M., et al, *"Object and File Management in the EXODUS Extensible Database System"*, University of Wisconsin at Madison, Technical Report, Madison, WI, March 1986.

[CHAN77] Chandy, K.M., *"Models of Distributed Systems"*, Proceedings of the 3rd International Conference on Very Large Data Bases, Tokyo, October 1977.

[CHOU85] Chou, H. and DeWitt, D.J., *"An Evaluation of Buffer Management Strategies for Relational Database Systems"*, Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, August 1985.

[COPE84] Copeland, G. and Maier, D., *"Making Smalltalk a Database System"*, Proceedings of the 1984 ACM–SIGMOD International Conference on the Management of Data, Boston, MA, June 1984.

[DERR86] Derrett, N.P. *et al*, *"An Object–Oriented Approach to Data Management"*, Proceedings of the 1986 IEEE Spring Compcon Conference,

[ESWA76] Eswaran, K.P. et al, *"The Notions of Consistency and Predicate Locks in a Database System"*, Communications of the ACM, (**19**) 11, 1976.

[FINK82] Finkelstein, S., *"Common Expression Analysis in Database Applications"*, Proceedings of the 1982 ACM–SIGMOD International Conference on the Management of Data, Orlando, FL, June 1982.

[GRAN81] Grant, J. and Minker, J., *"Optimization in Deductive and Conventional Relational Database Systems"*, in "Advances in Data Base Theory" , vol. 1, H. Gallaire, J. Minker and J.–M. Nicolas, Eds., Plenum Press, New York,

1981.

[GRAY78] Gray, J.N., *"Notes on Data Base Operating Systems"*, IBM Research, Technical Report RJ–2254, San Jose, CA, August 1978.

[GUTT84] Guttman, A., *"New Features for Relational Database Systems to Support CAD Applications"*, PhD Thesis, University of California, Berkeley, June 1984.

[IOAN86a] Ioannidis, Y., *"Processing Recursion in Deductive Database Systems"*, PhD Thesis, University of California, Berkeley, July 1986.

[IOAN86b] *"On the Computation of the Transitive Closure of Relational Operators"*, Proceedings of the 12th International Conference on Very Large Data Bases, Kyoto, Japan, August 1986.

[JARK84] Jarke, M., Clifford, J. and Vassiliou, Y., *"An Optimizing PROLOG Front–end to a Relational Query System"*, Proceedings of the 1984 ACM–SIGMOD International Conference on the Management of Data, Boston, MA, June 1984.

[KERN78] Kernighan, B. and Ritchie, D., **"The C Programming Language"** , Prentice–Hall, Englewood Cliffs, NJ, 1978.

[KERS84] Kershberg, L., Editor, *Proceedings of the First International Workshop on Expert Database Systems*, Kiawah Isl., SC, October 1984.

[KERS86] Kershberg, L., Editor, *Proceedings of the First International Conference on Expert Database Systems*, Charleston, SC, April, 1986.

[KOOI82] Kooi, R. and Frankfurth, D., *"Query Optimization in INGRES"*, Database Engineering, **(5)** 3, September 1982.

[KUNG84] Kung, R. et al, *"Heuristic Search in Data Base Systems"*, in [KERS84].

[LARS85] Larson, P. and Yang, H., *"Computing Queries from Derived Relations"*, Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, August 1985.

[MATT70] Mattson, R.L. et al, *"Evaluation Techniques for Storage Hierarchies"*, IBM Systems Journal, **(9)** 2, 1970.

[MOHA86] Mohan, C., *"STARBURST: An Extensible Relational DBMS"*, Panel Discussion on *Extensible Database Systems*, Proceedings of the 1986 ACM–SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[MORR86] Morris, K., et al, *"Design Overview of the NAIL! System"*, Stanford University, Technical Report STAN–CS–86–1108, Stanford, CA, May 1986.

[RICH83] Rich, E., **"Artificial Intelligence"** , McGraw–Hill, 1983.

[ROSE80] Rosenkrantz, D.J. and Hunt, H.B., *"Processing Conjunctive Predicates and Queries"*, Proceedings of the 6th International Conference on Very Large Data Bases, Montreal, October 1980.

[ROUS82a] Roussopoulos, N., *"View Indexing in Relational Databases"*, ACM Transactions on Database Systems, (7) 2, June 1982.

[ROUS82b] Roussopoulos, N., *"The Logical Access Path Schema of a Database"*, IEEE Transactions on Software Engineering, (8) 6, November 1982.

[ROUS86] Roussopoulos, N. and Kang, H., *"Preliminary Design of ADMS±: A Workstation–Mainframe Integrated Architecture for Database Management Systems"*, University of Maryland, Technical Report, College Park, MD, February 1986.

[SELI79] Selinger, P. et al, *"Access Path Selection in a Relational Data Base System"*, Proceedings of the 1979 ACM–SIGMOD International Conference on the Management of Data, Boston, MA, June 1979.

[SELL85] Sellis, T. and Shapiro, L., *"Optimization of Extended Database Languages"*, Proceedings of the 1985 ACM–SIGMOD International Conference on the Management of Data, Austin, TX, May 1985.

[SELL86a] Sellis, T., *"Global Query Optimization"*, Proceedings of the 1986 ACM–SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[SELL86b] Sellis, T., *"Optimization of Extended Relational Database Systems"*, PhD Thesis, University of California, Berkeley, July 1986.

[SELL86c] Sellis, T., *"Query Optimization in Extended Relational Database Systems"*, sumbitted for publication in the ACM Transactions on Database Systems, September 1986.

[STON75] Stonebraker, M., *"Implementation of Integrity Constraints and Views by Query Modification"*, Proceedings of the 1975 ACM–SIGMOD International Conference on the Management of Data, San Jose, CA, June 1975.

[STON76] Stonebraker, M. et al, *"The Design and Implementation of INGRES"*, ACM Transactions on Database Systems, (1) 3, September 1976.

[STON84] Stonebraker, M. et al, *"Quel as a Data Type"*, Proceedings of the 1984 ACM–SIGMOD International Conference on the Management of Data, Boston, MA, June 1984.

[STON85] Stonebraker, M. et al, *"Extending a Data Base System with Procedures"*, University of California, Technical Report UCB/ERL/M85/59, Berkeley, CA, July 1985.

[STON86a] Stonebraker, M., Sellis, T. and Hanson, E., *"Rule Indexing Implementations in Database Systems"*, in [KERS86].

[STON86b] Stonebraker, M. and Rowe, L., *"The Design of POSTGRES"*, Proceedings of the 1986 ACM–SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[ULLM82] Ullman, J., **"Principles of Database Systems"** , Computer Science Press, 1982.

[ULLM85] Ullman, J., *"Implementation of Logical Query Languages for Data Bases"*, Proceedings of the 1985 ACM–SIGMOD International Conference on the Management of Data, Austin, TX, May 1985.

[VALD86] Valduriez, P. and Boral, H., *"Evaluation of Recursive Queries Using Join Indices"*, in [KERS86].

[WILE84] Wilensky, R., **"The LISP PRIMER"** , W. Norton, Co, New York, 1984.

[WONG76] Wong, E. and Youssefi K., *"Decomposition: A Strategy for Query Processing"*, ACM Transactions on Database Systems, (1) 3, September 1976.

[YOUS78] Youssefi, K., *"Query Processing for a Relational Database System"*, PhD Thesis, University of California, Berkeley, 1978.

[ZANI83] Zaniolo, C., *"The Database Language GEM"*, Proceedings of the 1983 ACM–SIGMOD International Conference on the Management of Data, San Jose, CA, May 1983.

[ZANI84] Zaniolo, C., *"PROLOG : A Database Query Language for all Seasons"*, in [KERS84].

[ZANI85] Zaniolo, C., *"The Representation and Deductive Retrieval of Complex Objects"*, Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, August 1985.