UMIACS–TR–90–25 February 1990
CS–TR–2413

Using Caching, Cache Indexing, and Differential Techniques to Efficiently Support Transaction Time*

Christian S. Jensen†, Leo Mark, Nick Roussopoulos, and Timos Sellis
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742, USA

ABSTRACT

We present a framework for query processing in the relational model extended with transaction time. The framework integrates standard query optimization and computation techniques with new differential computation techniques. Differential computation incrementally or decrementally computes a query from the cached and indexed results of previous computations. The use of differential computation techniques is essential in order to provide efficient processing of queries that access very large temporal relations. Alternative query plans are integrated into a state transition network, where the state space includes backlogs of base relations, cached results from previous computations, a cache index, and intermediate results; the transitions include standard relational algebra operators, operators for constructing differential files, operators for differential computation, and combined operators. A rule set is presented to prune away parts of state transition networks that are not promising, and dynamic programming techniques are used to identify the optimal plans from remaining state transition networks. An extended logical access path serves as a "structuring" index on the cached results and contains in addition vital statistics for the query optimization process, including statistics about base relations, backlogs, about previously computed and cached, previously computed, or just previously estimated queries.

Keywords: *Temporal databases, transaction time, efficient query processing, incremental and decremental computation*

---

*The work was supported by NSF IRI-8719458, and AFOSR-89-0303. Correspondence via e-mail to jensen@brillig.umd.edu

†Supported by Aalborg University, Denmark.

Using Caching, Cache Indexing, and Differential Techniques to Efficiently Support Transaction Time

Christian S. Jensen* Leo Mark Nick Roussopoulos Timos Sellis†

Department of Computer Science, University of Maryland, College Park, MD 20742, USA

February 1989

Abstract

We present a framework for query processing in the relational model extended with transaction time. The framework integrates standard query optimization and computation techniques with new differential computation techniques. Differential computation incrementally or decrementally computes a query from the cached and indexed results of previous computations. The use of differential computation techniques is essential in order to provide efficient processing of queries that access very large temporal relations. Alternative query plans are integrated into a state transition network, where the state space includes backlogs of base relations, cached results from previous computations, a cache index, and intermediate results; the transitions include standard relational algebra operators, operators for constructing differential files, operators for differential computation, and combined operators. A rule set is presented to prune away parts of state transition networks that are not promising, and dynamic programming techniques are used to identify the optimal plans from remaining state transition networks. An extended logical access path serves as a "structuring" index on the cached results and contains in addition vital statistics for the query optimization process, including statistics about base relations, backlogs, about previously computed and cached, previously computed, or just previously estimated queries.

Keywords: *Temporal databases, transaction time, efficient query processing, incremental and decremental computation*

1 Introduction

The relational model presented by E. F. Codd twenty years ago [Cod70, Cod79] has gained immense popularity and is today regarded as a defacto standard for business applications. A main reason for the success is the generality of the model; it makes very few assumptions about specific application areas. This, however, has its drawbacks because the model does not provide detailed and customized support for some application areas. Extensions that make the relational model more suitable for the application areas have been an area of interest in the database research community ever since the relational model was presented.

*Supported by Aalborg University, Denmark.

†The work was supported by NSF IRI-8719458, and AFOSR-89-0303. Correspondence via e-mail to jensen@brillig.umd.edu

This paper presents an implementation model, IM/T (Implementation Model/Time), for an extension of the relational model supporting transaction time, DM/T (Data Model/Time) [JMR89, JM89]. Data is never deleted once entered into a database in this model; it is possible to see the database as of any time during the past, and it is possible to analyze the change history. Many applications will benefit from efficient transaction time support. In the literature, engineering, econometrics, banking, inventory control, medical records, and airline reservations have been mentioned as candidates (E.g. [MS89]).

Traditional implementation models can not efficiently cope with huge, ever growing quantities of historical data. The predominant approach taken to solve this problem has been partitioned storage, where data of individual relations are partitioned, and a storage hierarchy is maintained which favors efficient support of queries solely accessing recent data (E.g. [LDE*84, SL89]). While still allowing for partitioned storage, the data organization of IM/T allows efficient access changes to frequently accessed states, recent or old, of individual relations thus providing efficient support of any state.

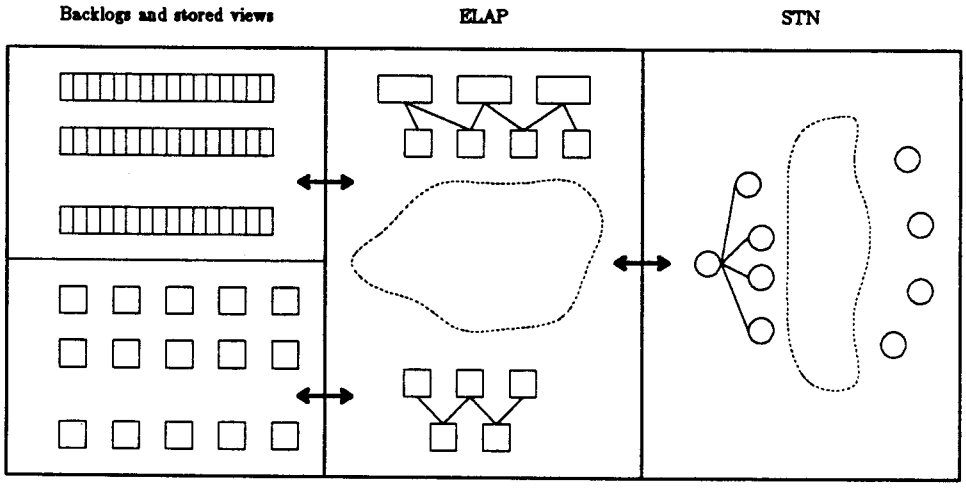

Figure 1: IM/T has three stores, one for base data, one for derived data, and one for the ELAP containing statistics and representing the structure of base and derived data. During query optimization plans using the stored data are enumerated in STNs.

IM/T exploits caching of query results. Caching is the idea of storing on secondary memory results of previous computations and subsequently using them to avoid redoing expensive computations [Rou82b, Sel88a]. Caching trades replication of data for speed of retrieval. It is potentially a very powerful technique, but a number of issues must be dealt with intelligently in order to gain the full benefits. Let us mention the most important ones.

First, there is the question of how to cache results. In IM/T, query results can be stored as actual data or as pointers to base data, possibly via several levels of indirection. Pointer cache storage gives a fixed, small tuple size and makes results very compact thus allowing for efficient use of main memory [Rou89]. For transaction time databases, however, in extreme cases, one base data page must be read for each pointer. Data cache storage solves this potential problem, because it allows for control of locality of reference. Additionally, it allows for reduction of references to slower storage areas.

Second, the utility of caching can be improved by means of cache indexing. IM/T extends the logical access

path [Rou82b] to an Extended Logical Access Path (ELAP), which allows for efficient identification of all potentially useful results during query processing. It is a persistent query graph with nodes for all cached, computed or just estimated results.

Third, to gain the full benefit of caching, results should be used in conjunction with differential computation techniques [Rou89]. The application of such techniques prolongs the usefulness of cached results because slightly outdated results need not be discarded and recomputed, but can instead be efficiently incremented or decremented to answer a query. IM/T generalizes incremental computation to differential computation using both incremental and decremental techniques, and it unifies differential computation and traditional recomputation.

Fourth, only potentially beneficial results should be cached, and if the cache is full, appropriate replacement strategies must be used. Cache management is part of IM/T, but it is not a topic of this paper.

Fifth, the fact that cached results get outdated must be addressed. Any possible update strategy ranging from eager (i.e. when relevant base data are entered), over threshold triggered, to lazy (when the result is requested) update is possible [RK86, Han87]. The details of cache update are not part of this paper.

In IM/T there is a *choice* of caching results. The motivation is that neither caching of all results (and differential computation) or no caching at all (and recomputation) is superior to the other in every given situation. Caching is attractive in environments characterized by many queries, few updates, very large underlying base relations, and comparably small results.

In a temporal setting the maintenance of stored results is likely to be more feasible than in snapshot settings. The reasons are that relations are large because previous states are retained and essential additional semantics for the process of selective caching is available. For example, fixed views are primary candidates for caching because they never get outdated, and the future outdatedness of time dependent views issued against past states can be estimated at the time of computation.

Query plan generation in IM/T uses the concept of state transition network (STN) [LW86], and query plan selection uses dynamic programming; see figure 1. We present a set of rules for pruning the STNs generated, the idea is to avoid generating inferior paths, and to save both space and time during cost estimation. During query plan generation and selection we use results from the cache, and we use both recomputation and differential computation versions of the operators of the query language of DM/T as possible transitions in STN's. Apart from defining the operators, we discuss how to efficiently implement the differential versions. In addition, combined operators are introduced to minimize the need for storage of intermediate results during query computation.

Efficient query processing is a central theme in database research, and consequently the work of this paper is related to a number of previous efforts.

The transaction time extension of this paper was designed to be transparent to the naive user of the standard relational model. To our knowledge, none of the other temporal extensions of the relational model (see [SS88, SA85, BADW82, Sno87, MS89, Bub77] for surveys and further references) shares this characteristic.

IM/T allows for partitioned storage and supports both reverse and forward chaining. Related efforts can be found in [DLW84, LDE*84, Ahn86, SA88, SL89, KS89]. Grid files have been suggested as a means of implementation of

temporal data [SK86], but they seem inappropriate since surrogates, for which no natural ordering exists, would be one dimension and time the other. In addition, indexing of other attributes is not allowed, which again is unsatisfactory. The subject of [RS87a] is multi dimensional file partition for static files with time as one of multiple dimensions.

The research reported in [GS89, SG89, GSS89] concentrate on different kinds of temporal joins (time-union, time-intersection, and event-joins) and temporal selectivity estimation. While interesting, we do not address these issues.

The focus of the work presented in [McK88] is the data model for a temporal database, and it is closely related to our work. It formally defines incremental algebra operators, resembling those of our state transition space. In addition, it surveys applications of incremental techniques in the relational model, and discusses ways to combine previous efforts into an implementation supporting both transaction time and valid (logical) time. Our work concentrates on implementation and on transaction time, only. We present a detailed design of an implementation model and concentrate on query optimization and processing.

IM/T exploits caching of views and the literature contains many contributions to the understanding of its many aspects. Aspects of materialized views relevant to distributed processing are presented in [SF89b, SF89a]. The performance of three techniques, lazy incremental computation, eager incremental computation, and recomputation has been compared in [Han87]. It was demonstrated that none of the techniques were superior to the others in all cases. In [Sel87, Sel88a, Jhi88] caching of query results is addressed to support query language procedures (programs, rules) stored in relational fields efficiently. Techniques aimed at reducing the cost of maintaining materialized views have most recently been reported in [BCL86, BLC89, TB88]. The ideas are to detect updates to base data that do not affect a view, and to detect when a view can be correctly updated using only the data already present in the view. IM/T generalizes and unifies traditional recomputation and incremental computation such that a single query can be computed partly using both recomputation, incremental computation, and decremental computation. Traditional systems, e.g. Ingres [WY76] and System R [SAC+79] use recomputation. In [KD79, KD84] it is described how to extend the RAQUEL II database management system to support dynamic derived relations using eager incremental update. In ADMS(±), a database management system implementing the standard relational model, incremental computation of views stored as pointer structures is used [Rou82a, Rou89, Rou87]. Our work has some resemblance to Postgres, where previous history is also retained. The temporal support, however, never was the focus, and time stamps and backlog queries are not supported as in IM/T. Postgres exploits caching, but since indexing, differential cache maintenance, and query execution are missing, the full potential of caching is not achieved [RS87b].

For previous work on query optimization, and further references, see [SC75, SAC+79, JK84, SS85].

State transition networks have to our knowledge never been applied in a temporal setting and in settings involving caching. In [LW86], STNs are used as a framework for query optimization in distributed environment, and in [HW89] STNs have been applied to multiple query optimization.

The structure of the remaining part of the paper is as follows. Section 2 presents **Transaction Time in the Relational Model, DM/T** and serves as a specification of the functionality to be supported by IM/T. The

transaction time concept, the data structures, and the query language of DM/T are presented. The remaining sections are devoted to IM/T and the efficient processing of DM/T queries. Section 3, **Structures of the Implementation Model, IM/T**, describes the three stores of IM/T, the store for base data, the cache, and the ELAP. Section 4 describes **Query Plan Generation and Selection**. STNs are used for enumerating alternative query plans, and dynamic programming is used to collect costs of entire plans from costs of single transitions. The concrete state and transition spaces, incorporating the use of cached results, differential computation, standard query computation techniques, and support for combined operators, are introduced. It is discussed how to use the ELAP to find promising results from the cache, to be considered when STNs are generated. In section 5, **Implementation of Operators of STNs**, we first present the cases to cover when implementing the operators. Then the three types of operators are considered: Recomputation operators, operators that construct differential files, and differential operators. Section 6, **Pruning the Search Space**, presents rules for reducing the sizes of the generated STNs. Section 7, **Conclusion and Future Research**, is the final section.

2 Transaction Time in the Relational Model, DM/T

In this section we briefly introduce the transaction time extension of the basic relational model [Cod70, Cod79], which was first introduced in [JMR89]. The purpose is to identify the kinds of queries that should be supported by IM/T.

The properties of the the time concept offered by DM/T are outlined in figure 2 and are discussed below.

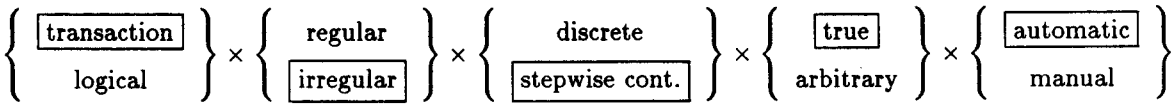

Figure 2: Characterization of the time concept offered by DM/T.

Two orthogonal time dimensions have been studied in temporal databases [SA85]. Logical time models time in the part of reality modeled by a database. Transaction time models time in the part of the reality that surrounds the database, the input subsystem. While logical time is application dependent, transaction time depends only on the database management system, and is inherently application independent.

First, DM/T supports transaction time as opposed to logical time. Second, a domain is regular if the distances between consecutive values of the active domain are identical. Otherwise the domain is irregular. DM/T supports an irregular time domain. Third, a time domain can be discrete or stepwise continuous. Facts with discrete time-stamps are only valid at the exact times of their time stamps. In contrast, in a stepwise continuous domain facts have an interval of validity. The DM/T time domain has this property (also termed *stability*), because the values of a relation remain the same until the relation is changed by a new transaction. Fourth, DM/T supports true time as opposed to arbitrary time. True time reflects the actual time of the input subsystem while an arbitrary time domain only needs to have a metric and a total order defined on it; the natural numbers is a possible arbitrary time

domain. Fifth, DM/T has automatic time-stamping, which is the natural choice for transaction time. Manual, user supplied time-stamp values are natural for logical time. We have chosen tuple stamping as opposed to attribute value stamping. The major motivation has been to provide a 1.NF model which is a simple and yet powerful extension of the basic relational model.

In order to record detailed temporal data and still be able to use the operators of the basic relational model we have introduced the concept of a backlog relation. A *backlog*, B_R, for a relation, R, is a relation that contains the complete history of change requests to relation R [RK86]. Backlog B_R contains three attributes in addition to those of R. Attribute *Id* is defined over a domain of logical, system generated unique identifiers, i.e. surrogates. The values of *Id* represent the individual tuples, termed change requests. The attribute *Op* is defined over the enumerated domain of operation types, and values of *Op* indicate whether an insertion (*Ins*), a deletion (*Del*) or a modification (*Mod*) is requested[1]. Finally, the attribute *Time* is defined over the domain of transaction time stamps, *TTIME*, as previously discussed. DM/T automatically generates and maintains a backlog for each base relation (i.e. user defined relations and schema relations). Figure 3 shows the effect on backlogs resulting from operation requests on their corresponding relations. As a consequence of the introduction of time stamps, a base relation is now a function of time. To retrieve

Requested operation on R:	Effect on B_R:
insert R(tuple)	insert B_R(id, *Ins*, time, tuple)
delete R(key)	insert B_R(id, *Del*, time, tuple(key))
modify R(key, new value)	insert B_R(id, *Mod*, time, tuple(key,new value))

Figure 3: System controlled insertions into a backlog. The function "tuple" returns the tuple identified by its argument.

a base relation it must first be time sliced. Let R be any base relation, then the following are examples of *time slices* of R:

$$
\begin{aligned}
R(t_{init}) &\stackrel{\text{def}}{=} R_{init} \\
R(t_x) &\stackrel{\text{def}}{=} R \text{ "at time } t_x\text{"}, \; t_x \geq t_{init} \\
R &\stackrel{\text{def}}{=} R(NOW)
\end{aligned}
$$

When the database is initialized, it has no history and every relation is empty. If R is parameterized with an expression that evaluates to a time value, then the result is the state of R as it was at that point in time. It has no meaning to use a time before the database was initialized and after the present time. If R is used without any parameters this indicates the current R. Time sliced relations have an *implicit* time stamp attribute, not shown unless explicitly projected. Note, that these features help provide transparency to the naive user. We also introduce the special variable *NOW* which assumes the time when the query is executed.

[1] On a lower level modifications are modeled by a deletion followed by an insertion, each with the same time stamp.

If the expression, E, of a time sliced relation, $R(E)$, contains the variable NOW, then R is *time dependent.* Otherwise, it is *fixed.* While fixed time slices of relations never get outdated, time dependent time slices do and they are consequently updated by the DBMS before retrievals.

A view is time dependent if at least one of the relations and views it is derived from is time dependent. Otherwise it is fixed. Traditional views are ultimately derived directly and solely from time sliced base relations. If a view ultimately is derived directly, i.e. not via a time sliced base relation, from at least one backlog, then we term it a backlog view. Backlog views are time sliced as are base relations and views. Backlog view time slices involving NOW are time dependent, and, as above, so are backlog views derived from views involving NOW. We define:

$$B_R(t_x) \stackrel{\text{def}}{=} \sigma_{Time \leq t_x} B_R$$

$$B_R \stackrel{\text{def}}{=} B_R(NOW)$$

By introducing the time slice operator it is possible to use the standard relational algebra as the query language. The query language of DM/T was presented in [JMR89] and in [JM89] it was extended to support analysis of change history. In this paper we only consider time-slice, selection, projection, and equi-join. We adopt a set of precedence rules to simplify the appearance of query expressions. Time-slice has highest precedence, and is followed by projection and selection with the same precedence, which, in turn, are followed by binary operators all with the same precedence. Parentheses are used to control precedence in the standard way, and evaluation is from left to right.

3 Structures of the Implementation Model, IM/T

In the previous section we described the data model, DM/T. The subject of this and the remaining sections is the implementation model, IM/T, which supports the data structures and operators of DM/T. We present the three different stores of IM/T: the store containing backlogs and indices; the cache containing views; the ELAP which contains information about queries, and is an index to the cache.

3.1 Storage of Backlogs

Backlogs assume the role of base relations and are always stored. They are stored like traditional base relations with the possibilities of traditional indexing. Throughout this paper we will assume that tuples of a backlog are clustered according to the values of their time stamp attribute. Also, mainly for simplicity, we assume that backlog tuples actually contain all the data of their attributes; compression techniques [Bas85] can be applied to backlogs, since the amount of data stored in backlogs can grow arbitrarily. To cope with the bulk of historical data, we allow for partitioning the backlog store [DLW84, LDE*84, Ahn86, SA88, SL89, KS89, Chr87]. In this paper we will not present this facility.

Finally, realizing that even WORM storage is limited and that some historical data might not be needed by any user we offer advanced facilities for pruning historical data. This is the topic of a separate, forthcoming paper.

3.2 The Pointer and Data Cache of IM/T

The cache of IM/T is a collection of query results stored as either pointers or data. Physically, a part of secondary memory is allocated for the cache. Each entry of the cache is of the form (`rid`, `result`), where `rid` uniquely identifies an entry and `result` is of the format

$$\texttt{result} \leftarrow \texttt{array of ptr} \mid \texttt{array of (ptr} \times \texttt{ptr)} \mid \texttt{relation}$$

Tuples of the same entry are stored consecutively and are sorted on `tid`'s (pointers) or surrogate attribute values (data). There can exist indices on the tuples of results.

The ELAP, discussed in the next subsection, is a structuring index on the cache and is used to identify cache entries to be used in query processing. In the ELAP, a cache entry is represented by its `rid`, and therefore an index on `rid`'s of results is desirable. Clustering of results according to the structure of the ELAP is an interesting topic not addressed in this paper.

Differential files computed as intermediate results during query processing are not stored in the cache. It can, however, be beneficial to store statistics about such files. The statistics can help estimate the cost of processing future differential files and help choose between different ways of processing a differential file. The design of data structures and algorithms that maintain the statistics and the use of the statistics during query optimization is a subject of current research.

The cache contains the current states of all base relations, and they are updated eagerly. The motivation is that the extension of the data model DM/T is transparent to the naive user and we further want IM/T to retrieve current data and check standard integrity constraints efficiently.

3.3 The Extended Logical Access Path of IM/T

The ELAP is a directed acyclic graph (DAG) [Rou82b]. Each node is associated with a set of equivalent query expressions, a list of statistics about each query expression, and an optional reference to a cached result. The edges are labeled by operators, and an edge (or a pair, possibly ordered) from node N_a (and N'_a) to node N_b indicates that the operator constructs an expression associated with N_b from an expression associated with N_a (and N'_a).

The ELAP integrates graphs of query expressions that have been computed or have been subject to estimation of statistics into a unifying structure by merging nodes representing common (sub-)expressions. It is important to observe that, while the expressions of a node all produce the same result, they may have different processing costs. The ELAP is a generalized AND/OR DAG where, at a single node, there is a choice ("OR") of one of several sets of "AND" edges [MB85, Ric83], where "AND" edges correspond to binary operators. To illustrate, consider the following equivalent query expressions:

$$\pi_{Emp(t_1).Name,Emp(t_2).Salary}\sigma_{Emp(t_1).Salary \geq 30}(Emp(t_1) \bowtie_{Emp(t_1).Name=Emp(t_2).Name} Emp(t_2))$$

$$\pi_{Emp(t_1).Name,Emp(t_2).Salary}(\sigma_{Salary \geq 30} Emp(t_1) \bowtie_{Emp(t_1).Name=Emp(t_2).Name} Emp(t_2))$$

$$\pi_{Emp(t_1).Name,Emp(t_2).Salary}(\pi_{Name}\sigma_{Salary \geq 30} Emp(t_1) \bowtie_{Emp(t_1).Name=Emp(t_2).Name} \pi_{Name,Salary} Emp(t_2))$$

They all return the names and salaries at time t_2 of employees that were employed at both time t_1 and t_2 and that earned more than 30K at t_1. Yet, they are different expressions with different processing characteristics. The ELAP for these expressions is shown in figure 4.

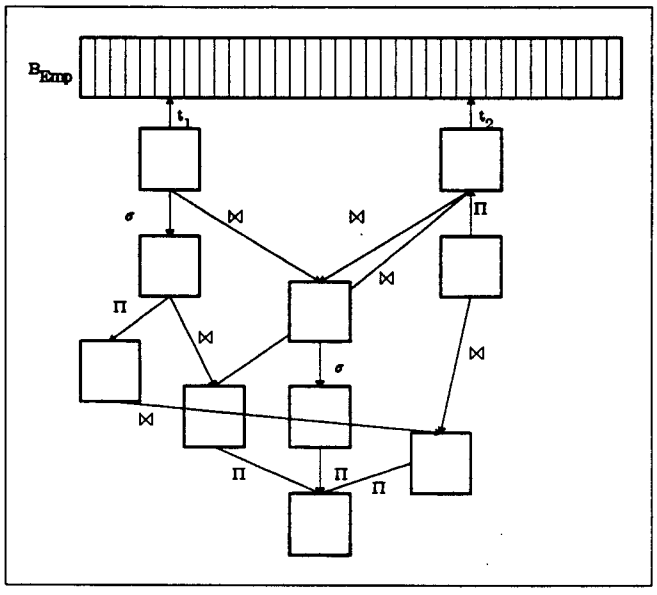

Figure 4: The view corresponding to a node can be computed from several query language expressions.

It follows that a cached result of a node could have been computed in several ways, and that it subsequently can be computed in several ways. A node tells from which expression a cached result was most recently computed. There is at most one cache entry per node.

Nodes can belong to one of six categories depending on the computational status of the labeling query expressions. The result of a query expression can be cached as data or pointers; the result of the query expressions can have been cached previously as data or pointers; it is possible that no result of the query expressions has ever been cached, but results might have been computed or just estimated; finally, a node can denote a backlog. Edges of the ELAP are labeled by operators of the query language, here restricted to selection, projection, and join. In addition, we allow for combined operators to avoid storage of intermediate results.

Different types of statistics can be kept in the six types of nodes. Individual statistics should only be maintained if the cost of doing so is less than the benefits achieved from having them available during query optimization. Practical experiments are needed to determine when this is the case. Possible statistics include: (1) *cardinality of stored result*, (2) *result stored as pointer or data*, (3) *tuple size*, (4) *which expression is cached*, (5) *up-to-date status*, (6) *how often used*, (7) *usage*, (8) *computation cost*, (9) *when deleted*, (10) *why deleted*, and (11) *available indices*.

4 Query Plan Generation and Selection

To efficiently compute a query, the system generates a state transition network (STN) where the initial state contains the uncomputed query, the backlog relations it is defined in terms of, the cache, and the ELAP. A state transition occurs when the cost of a partial computation toward the total computation of the query is estimated. The new state is identical to the predecessor state except it is assumed that the cost estimated computation has been performed. A final state is reached when the costs of all computations have been estimated. By following all paths from the initial to a final state and accumulating costs for each path, the total costs of computing the query in different ways, are obtained, and we can choose the query plan with the lowest cost. The purpose of this section is to formalize and elaborate on the generation of query plans as just described.

4.1 State Transition Network

A STN for a query, Q, is a labeled DAG, and can be defined as

$$STN(Q) = (\mathcal{S}, \mathcal{P}, P, \Gamma, x_0, \mathcal{X}_f) \ , \tag{1}$$

where $\mathcal{S}$ is a set of states (nodes); each node contains what remains to be calculated of query Q along with the data structures that can be used to compute the query[2] (i.e. intermediate results, the ELAP, the cache, backlogs). $\mathcal{P}$ is a set of operators, which describe the query processing and label the edges of the DAG. P is a mapping: $\mathcal{S} \rightarrow 2^{\mathcal{P}}$, which maps the state space into the power set space of operations, and describes the set of operations applicable at a given state. Γ is the set of transitions, $\Gamma \subseteq \mathcal{S} \times P(\mathcal{S}) \times \mathcal{S}$; thus, an edge is a triplet, (x_1, p, x_2), containing a start state, a label, and an end state. The last two elements of (1), $x_0 \in \mathcal{S}$, and $\mathcal{X}_f \subseteq \mathcal{S}$ are the initial and the final states, respectively. The initial state contains the uncomputed query, and a final state contains the computed query, and possibly various intermediate results.

A plan for a query, Q, and a state, x, tells which sequence of operators to apply to the partially computed query Q at state x in order to arrive at the final state. If $x \neq x_0$ then the plan is partial. If we let $p_1 \circ x$ denote the application of operator p_1 at state x, then a plan can be expressed as

$$p_1, p_2, p_3, \ldots, p_n \ , \text{where } p_n \circ \ldots \circ p_3 \circ p_2 \circ p_1 \circ x \in \mathcal{X}_f$$

We associate a cost C with each plan in the obvious way. First, we define $cost : (\mathcal{S}, P(\mathcal{S}), \mathcal{S}) \rightarrow [0; \infty[$ to be the cost of applying an operator to a state to get a new state (i.e. the cost of an edge in our DAG). Then the cost of a plan is

$$C(x, p_1, p_2, p_3, \ldots, p_n) = cost(x, p_1, s_2) + cost(s_2, p_2, s_3) + cost(s_3, p_3, s_4) + \ldots + cost(s_n, p_n, x_f) \ ,$$

where $x_f \in \mathcal{X}_f$; figure 5 shows this plan as a part of a larger network.

The minimal cost of a query Q is defined as the minimum over all possible plans for Q and x:

$$C_Q(x) = \min\{C(x, p_1, p_2, p_3, \ldots, p_n) \mid p_n \circ \ldots \circ p_3 \circ p_2 \circ p_1 \circ x \in \mathcal{X}_f\}$$

A plan $p_1, p_2, p_3, \ldots, p_n$ for which $C(x, p_1, p_2, p_3, \ldots, p_n) = C_Q(x)$ is *optimal*.

[2]Note that no computations are actually carried out. We are merely estimating assumed computations.

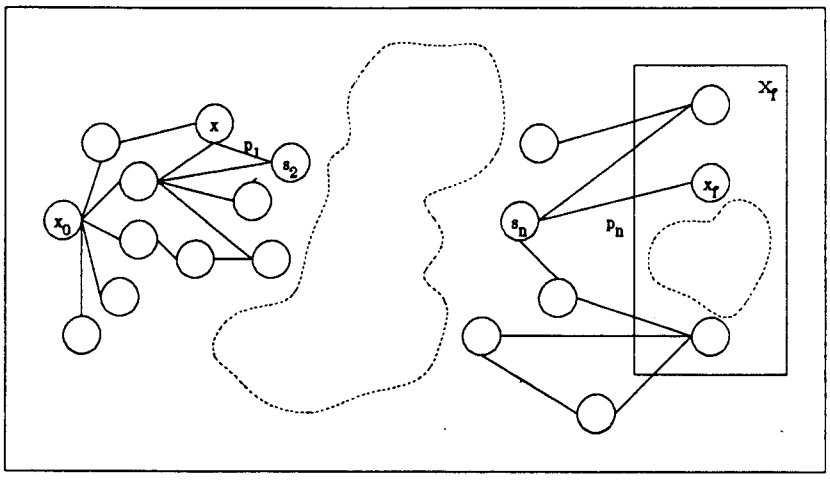

Figure 5: An outline of a STN.

4.2 Plan Selection

Assuming we have costs for all single state transitions, the cheapest query plan in the network can be found by applying dynamic programming techniques. The function $C_Q(x)$ of the previous subsection can be expressed as:

$$C_Q(x) = \min_{p \in P(x)} \{cost(x, p, x') + C_Q(x')\}$$

Dynamic programming is applicable because the cost of a single transition in a STN depends only on local information and not e.g. on the nature of previous transitions that lead to the state of the current transition. This has been termed the *separation assumption* [LW86].

When we use dynamic programming, the task of finding a good query plan is conceptually divided into two phases: generation of the STN of the query to be computed; estimation and selection of the optimal path in the STN. In practice, the whole STN need not be computed before phase two is initiated; parts needed during phase two must, however, be made available when needed, and upon completion all of the STN will be needed. For this reason, dynamic programming requires a relatively large amount of storage space [Sed88, RND]. Among the heuristic techniques the A^* algorithm (e.g. [Ric83]) is an alternative, but until an easily computable and precise heuristic function has been found, dynamic programming seems more promising.

To improve performance we introduce pruning rules (section 6), that specify the function P. They allow us to eliminate paths that are generally not competitive, and therefore limit the search space with little chance of eliminating advantageous plans.

4.3 State and Transition Spaces

We now present the specific design of the type of STN to be used in IM/T. We describe what constitutes a state and which transitions are possible on the states.

State Spaces of IM/T

IM/T generates a separate STN for each query it optimizes, and each STN has its own state space. A state space is a set of states, each consisting of a set of objects. All the types of objects in a state space are stored on secondary memory, and can be read[3], used, and as a result new objects can be created.

The query of a STN is ultimately defined in terms of a set of **backlogs**. These are part of all the states for that STN. Together with the backlogs **cached results** constitute the outsets for query computation, and the contents of the cache is part of all states. The cache is not changed during plan enumeration and selection, but can be updated when the selected plan is processed. Similarly, the **ELAP** is part of each state of any STN. The final component of states is **intermediate results**. An intermediate result is any query that can be expressed in terms of backlogs, cached results, and existing intermediate results. Generally a state will contain a set of intermediate results to be used in further computations in order to achieve the evaluation of the query of the STN at hand. Such results can later be stored in the cache if they are part of the plan chosen for actual execution. In this case the ELAP is updated to reflect the new state of the cache. Even if the state of the cache is not changed, the ELAP can be updated with statistics of computed, or estimated temporary results.

Two states with mutually equivalent objects are identical states.

Transition Space of IM/T

Below, we define the transition space. In section 5, we will discuss implementation of the operators of the transition space.

The relational operators projection (π), selection (σ), and equi-join ($\bowtie$) are included.

Operators that increment or decrement a computed result obtainable by performing a selection or a projection on a relational algebra expression, or a join of two relational algebra expressions, for which differential files exist, are included: $DIF(\sigma_F R, \delta_R)$, $DIF(\pi_A R, \delta_R)$, $DIF(R \bowtie S, R, \delta_R, S, \delta_S)$. The differential file of relation R, δ_R, consists of a set of insertions, δ_R^+, and a set of deletions, δ_R^-, such that R updated with the differential, δ_R, is given by $(R - \delta_R^-) \cup \delta_R^+$. There are no references from δ_R^- to δ_R^+, i.e. no deletions of insertions. For example, if we already have $\sigma_F R(t_x)$ and $\delta_{R,(t_x,t_a)}$ we can get $\sigma_F R(t_a)$ by computing $DIF(\sigma_F R(t_x), \delta_{R,(t_x,t_a)})$. Generally, if $R' = (R - \delta_R^-) \cup \delta_R^+$, then

$$
\begin{aligned}
DIF(\sigma_F R, \delta_R) &= (\sigma_F R)' \\
DIF(\pi_A R, \delta_R) &= (\pi_A R)' \\
DIF(R \bowtie S, R, \delta_R, S, \delta_S) &= (R \bowtie S)'
\end{aligned}
$$

Operators that construct a differential file of a computed result obtainable, as above, by selection, projection, or join of intermediate results are included: $DELTA(\sigma_F, \delta_R)$, $DELTA(\pi_A, \delta_R)$, $DELTA(\bowtie, R, \delta_R, S, \delta_S)$. Thus, if δ_R

[3] Objects still exist after they have been read.

exists, we can get $\delta_{\sigma_F R}$ of $\sigma_F R$ by computing $DELTA(\sigma_F, \delta_R)$. We have:

$$\begin{aligned} DELTA(\sigma_F, \delta_R) &= \delta_{\sigma_F R} \\ DELTA(\pi_A, \delta_R) &= \delta_{\pi_A R} \\ DELTA(\bowtie, R, \delta_R, S, \delta_S) &= \delta_{R \bowtie S} \end{aligned}$$

Combined operators are included. To increment, say, the result of $\pi_A \sigma_F R(t_x)$ we can get first $\delta_{R(t_x)}$, then, second, $\delta_{\sigma_F R(t_x)}$, and third we can apply the incremental operator for projections to finally obtain the new, incremented result. This implies storage of $\delta_{R(t_x)}$. A combined operator, $DIF(\pi_A \sigma_F R, \delta_R)$ would eliminate storage of an intermediate result by processing π_A and σ_F in a single pass. More specifically we allow for combining a selection or a projection with another operator ($\sigma, \pi, \bowtie$, or combined) into a combined operator.

Identity Transformations

A user query can be processed in many ways to produce the desired result. Identity transformations for relational expressions are utilized to generate at any state all applicable operations that will contribute towards the complete processing of the query. In addition, identity transformations are used to decide whether two states of a STN are identical or not. We add the following three transformation rules to the ones presented in the literature [SC75, Ull82, JK84]:

substituting selection and differential selection

$$DIF(\sigma_F R, \delta_R) \equiv \sigma_F DIF(R, \delta_R)$$

substituting projection and differential projection

$$DIF(\pi_A R, \delta_R) \equiv \pi_A DIF(R, \delta_R)$$

substituting join and differential join

$$DIF(R \bowtie S, R, \delta_R, S, \delta_S) \equiv DIF(R, \delta_R) \bowtie DIF(S, \delta_S)$$

4.4 Getting Views from the Cache

We have included a cache for views in IM/T, and we have defined an ELAP as a "structuring index" on the cache. The role of the ELAP is to allow for efficient identification of cached results that can be used to compute a query at hand.

Let DB be a database instance, i.e. an instance of the backlog store, and Q^c the defining expression of a cached result, then $Q^c(DB)$ is the cached result of Q^c on DB.

The result $Q^c(DB)$ is only useful for the computation of a (sub-)query, Q_s, if the data of $Q_s(DB)$ are all contained in $Q^c(DB)$, and can be extracted from $Q^c(DB)$ using an expression, E, of the query language (confer [LY85]). If this is the case for any database instance, we say that Q^c *covers* Q_s, $Q_s \sqsubseteq Q^c$. Coverage is an intensional property. Formally,

$$Q_s \sqsubseteq Q^c \stackrel{\text{def}}{\Leftrightarrow} \forall DB \; \exists E : \; \tilde{Q}_s(DB) = E(\tilde{Q}^c(DB)),$$

where $\tilde{Q}$ denotes Q where temporal information (time slice) is ignored. Thus, $\sigma_{x \geq 15} R(t_1) \sqsubseteq \sigma_{x \geq 10} R(t_2)$, even if $t_1 \neq t_2$, because $\forall DB : \sigma_{x \geq 15} R = \sigma_{x \geq 15} \sigma_{x \geq 10} R$.

The covering queries we are most interested in are the ones that are most cheaply modified to the requested query, i.e. the minimal covering queries. Certainly, if $Q_1 \sqsubseteq Q_2 \sqsubseteq Q_3$ then, considering only coverage, we would prefer to use Q_2 instead of Q_3 to compute Q_1.

Orthogonal to the issue of coverage there is the issue of *temporal closeness*, which we have disregarded so far. There is both an intensional and an extensional aspect. We address the intensional aspect first. When we have retrieved a result from the cache it might not reflect the state we are interested in. If we let $Q_s = \sigma_{x \geq 10} R(t_1)$ and let $Q_1^c = \sigma_{x \geq 10} R(t_a)$, then the two queries are identical under coverage, but if $t_1 \neq t_a$ the operator DIF (probably) still needs to be applied to Q_1^c and an appropriate differential file to make it correctly reflect the desired state.

Assume the existence of $Q_2^c = \sigma_{x \geq 10} R(t_b)$. If the temporal expressions t_a and t_b are both fixed, then we would choose Q_1^c if t_a is closer to t_1 than is t_b. Otherwise we would choose Q_2^c. The concept of closeness is defined in terms of the cost of the differential computation that has to be carried out in order to reach the desired state, and it depends on the size of the portions of the associated backlog that has to be processed. The distance between time stamps is an intensional property which can be used for comparing closeness. However, if $t_a \leq t_1 \leq t_b$ or $t_b \leq t_1 \leq t_a$, the distance between time stamps is not a reliable means of comparison.

The extensional aspect of closeness is important because cache entries generally get outdated. In the context of time dependent views it is not sufficient only to look at the intensions of queries as we did above where we compared t_1, t_a, and t_b. For example, if $Q_s = \sigma_{x \geq 10} R(t_1)$, and the cache contains $Q_1^c = \sigma_{x \geq 10} R(t_1)$ and $Q_2^c = \sigma_{x \geq 10} R(t_2)$, where $t_1 \neq t_2$ then Q_2^c still can be more useful than Q_1^c. This is so because t_1 could be time dependent and Q_1^c could be very outdated. Outdatedness of a cached query result is defined as the closeness between the defining query expression at the time it was computed and the current defining query expression. Because of the variable NOW query expressions in general change over time.

For each cached result the ELAP stores the value of the variable NOW at the time when the result was computed so that the states of cached results can be inferred without actually accessing them. Also the ELAP holds statistics that can help estimate the outdatedness of results (i.e. estimate the number of change requests between two points in time and the cost of processing them appropriately).

5 Implementation of Operators of STNs

In this section we discuss the operators of STNs in more detail. Initially, we outline the different cases to consider. Based on these we discuss alternatives for implementation of the operators.

5.1 Overview of Operators

In order to completely account for the implementation of the operators of the STNs of IM/T, a large number of cases must be considered. Disregarding for the moment combined operators, the cases are outlined in figure 6. The figure

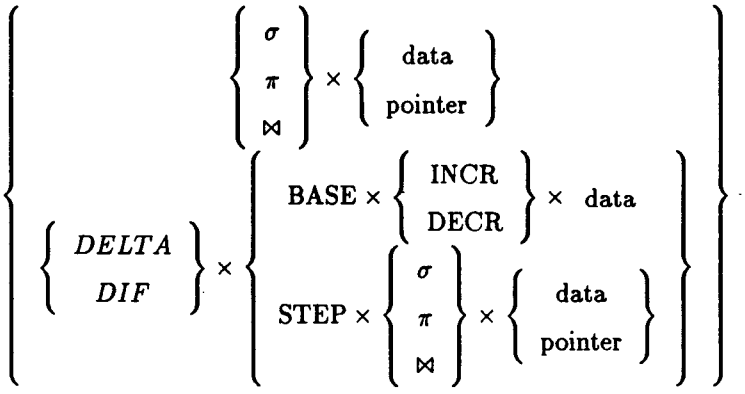

Figure 6: Implementation of operators of STNs.

has 22 entries, each corresponding to a separate case. In IM/T stored results, possibly cached, can be stored as either actual data or pointers that point to the data. The entries "data" and "pointer" indicate the type of arguments. As can be seen all operators must work on both kinds of arguments. The only exceptions are that the base cases for *DIF* and *DELTA* only work on data, since it does not make sense to store base data of backlogs as pointers. The figure does not include information about the type of result returned by the operators. If the arguments are pointers, then the results are pointers as well, and if the arguments are data the results can be both data and pointers, the only restriction being that differential files are assumed to be data. This adds an additional 8 cases.

We find, as the first six entries, the ordinary operators σ, π, and $\bowtie$[4]. These operators have their standard semantics and can be implemented as suggested in the literature (e.g. [Sha86] [SAC+79]).

The remaining sixteen cases concern the two new operators, *DIF* and *DELTA*. The operator *DELTA* derives differential files. The base cases are the incremental and decremental processing of sequences of change requests to get differential files. The step cases are the computations of differential files of relations from the differential files of relations from which they are derived by either projection, selection, or join.

The operator *DIF* differentially updates a stored result to correctly reflect a desired state. In the two base cases a time sliced base relation is either incrementally or decrementally updated with change requests from the backlog of the relation. The three step cases for pointer and data arguments differ on how the outset is related to the differential file(s) to be used. It is possible to use the differential file of a relation from which the outset is derived by a projection (including the identity projection) or a selection, and the differential files of relations from which the outset is derived by a join can be used.

Finally, selections and projections can be done on the fly, meaning that a selection and a projection can be

[4] In the following $\bowtie$ denotes equi-join.

performed interleaved with another operator (selection, projection, or join) in a single pass without storage of intermediate results.

We will, when not explicitly stated otherwise, assume that operators take data arguments and produce data results.

5.2 Selection, Projection, and Join

The traditional relational algebra operators, selection, projection, and join can be applied to any relation, including differential files (δ_R^+, δ_R^-). The expression F of the selection operator, $\sigma_F R$, can contain a conjunction of selection criteria of the form **Att_Name op Att_Name** or **Att_Name op Value**, where op is one of $=, <, >, \geq, \leq, \neq, \not<, \not>, \not\geq$, or $\not\leq$, and **Att_Name** is an attribute identifier of the relation R. The most advantageous implementation of selection depends on numerous factors, and has already been addressed in many settings. Consequently we will not address it here.

The projection expression, A, of $\pi_A R$ is any subset of attributes of R. When we do differential computations we would like to be able to distribute projections over difference. In order to make this legal we must at all times make sure that unique identification of tuples is possible. We choose to do this by always retaining the primary key of relations and in addition remembering whether it was removed by projection or not.

The equi-join operator $R \bowtie_F S$ can be used on any two relations. The condition F is a list of elements of the form **Att_Name_1 = Att_Name_2** or **Att_Name_1 = Att_Name_2**, where **Att_Name_1** is an attribute of relation R (S) and **Att_Name_2** is an attribute of relation S (R). Several ways have been suggested for doing binary joins, e.g. Hash-Join, Nested-Loop-Join, Sort-Merge-Join. For a thorough treatment, see [Sha86].

Finally, selection and projection can be combined with any operator (possibly combined) to form a combined operator.

5.3 Computing Differential Files

The operator $DELTA$ computes differential files, and can be applied to a number of different arguments. Here we discuss each case.

First, however, a differential file for a relation R is denoted δ_R, and $\delta_R = (\delta_R^+, \delta_R^-)$. A differential is relative to an outset, and it is used to update it to a desired state. If R' is the result of updating R with δ_R, we have: $R' = (R - \delta_R^-) \cup \delta_R^+$.

The Base Cases

The base case is the process of generating a differential file, $\delta_{R(t_x)}$, directly from a backlog, B_R, i.e. $DELTA(R(t_a), (t_a, t_x))$, where t_x is the requested state of R. This corresponds to the case $DELTA(\pi_A, R)$ on page 13, with an identity projection. The base case differs from all other applications of $DELTA$ because the argument is a list of change requests while the arguments of other applications are relations.

If $t_a < t_x$ the requested state of R is a future state relative to its current state, and we are in the "incremental" case. If $t_a > t_x$ we are in the "decremental" case.

The construction procedure for $\delta^+_{R(t_a)}$ and $\delta^-_{R(t_a)}$ starts with the initialization of these to empty relations. The schema of $\delta^+_{R(t_a)}$ is that of R, and $\delta^-_{R(t_a)}$ only contains the primary key attribute of R[5]. Then we process change requests from the outset in the direction of t_x until the next change request to be processed has a time stamp that is not in the half open interval from t_a to, and including, t_x.

Each request is projected to remove superfluous attribute values. Let us assume we are in the incremental case. Insertion requests go into $\delta^+_{R(t_a)}$, which optionally can be kept sorted on key values, or/and an (hash) index on key values can be maintained. A deletion request refers either to a tuple in the outset or to a tuple in $\delta^+_{R(t_a)}$[6]. First $\delta^+_{R(t_a)}$ is searched for a tuple matching the deletion request, and if a match is found, then the request is disregarded, and the matching tuple of the current $\delta^+_{R(t_a)}$ is deleted, since the net effect is that no change takes place. Otherwise the projected deletion request goes into $\delta^-_{R(t_a)}$. Note that no action was taken above when we encountered an insertion request of a previously encountered deletion request. Such insertion requests and corresponding deletion requests must be carried out because they update implicit time stamp attributes of base relations; such attributes are hidden, but can be seen by explicit projections. So far, we have ignored these implicit attributes, and will continue to do so. Tuples of $\delta^+_{R(t_a)}$ and $\delta^-_{R(t_a)}$ are written to secondary memory one page at a time. Note that there are no references from $\delta^-_{R(t_a)}$ to $\delta^+_{R(t_a)}$, making the sequence of operation in the formula above valid in the sense that the outcome is, in fact, $R(t_x)$. Also note that there can be references from $\delta^+_{R(t_a)}$ to $\delta^-_{R(t_a)}$, making the sequence of operation in the formula the only valid one.

When there are no more change requests, the optional index on $\delta^+_{R(t_a)}$ is deleted and both differentials are stored sorted on key values.

In the decremental case the only change is that deletion requests assume the role of insertion requests and insertion requests that of deletion requests.

The Step Cases

What is left now is the cases where a differential file of a result is constructed from the differential file of another result. In $DELTA(\sigma_F, \delta_R)$ the operator constructs the differential file of $\sigma_F R$ from the differential file of R, δ_R, where R denotes any query expression. This is just a selection:

$$DELTA(\sigma_F, \delta_R) = \sigma_F \delta_R = (\sigma_F \delta^-_R, \sigma_F \delta^+_R)$$

Similarly, in $DELTA(\pi_A, \delta_R)$, we make a projection:

$$DELTA(\pi_A, \delta_R) = \pi_A \delta_R = (\pi_A \delta^-_R, \pi_A \delta^+_R)$$

[5] In algebra expressions we assume, for simplicity, that the schema is that of R.

[6] Note that the eagerly maintained current states of base relations allow for checking that deletions and insertions actually make sense, i.e. that deletions actually delete something existing and, conversely, that insertions actually insert something not already existing. These are system enforced integrity constraints.

Remember, that key information is retained to overcome the problem of indistinguishable tuples when distributing a projection over a difference. Figure 7 is a schematical representation of selection and projection for $DELTA$.

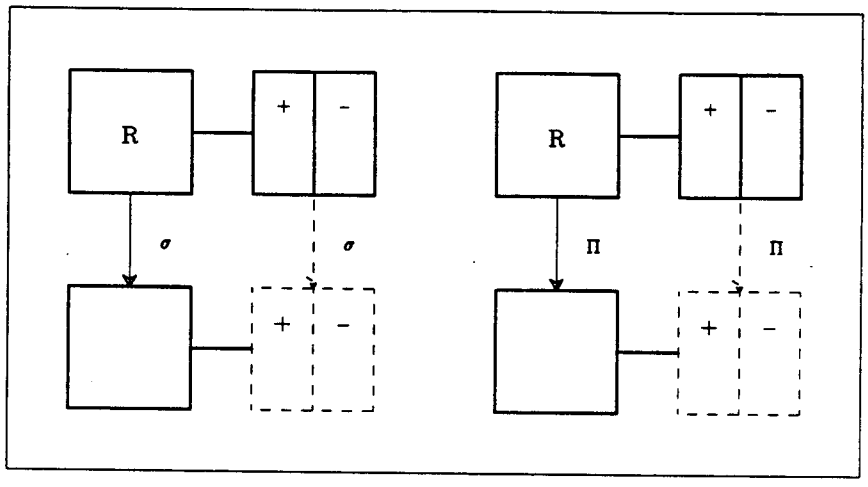

Figure 7: Schematic representation of $DELTA$ for projection and selection.

The last case is the join: $DELTA(\bowtie, R, \delta_R, S, \delta_S)$. To construct the differential file of $R \bowtie S$, we need both R, S, δ_R, and δ_S. If we let R', S', and $(R \bowtie S)'$ denote the updated versions of R, S, and $R \bowtie S$ respectively, then we want to compute $\delta_{R \bowtie S}$ such that equality 2 below is obeyed, and there are no deletions of insertions. For this purpose we in addition use the equalities 3 and 4:

$$R' \bowtie S' = (R \bowtie S)' \tag{2}$$

$$(R \bowtie S)' = [(R \bowtie S) - \delta^-_{R \bowtie S}] \cup \delta^+_{R \bowtie S} \tag{3}$$

$$R' \bowtie S' = [(R - \delta^-_R) \cup \delta^+_R] \bowtie [(S - \delta^-_S) \cup \delta^+_S] \tag{4}$$

We derive $\delta_{R \bowtie S}$ by transforming the right hand side of equality 4 into an equivalent expression of the form $[(R \bowtie S) - \boxed{x}] \cup \boxed{y}$. Then, by equality 2 and 3, $(\delta^-_{R \bowtie S}, \delta^+_{R \bowtie S}) = (\boxed{x}, \boxed{y})$.

To do the transformation we need two transformation rules:

$$(R \cup S) \bowtie T = (R \bowtie T) \cup (S \bowtie T) \quad \text{and} \quad (R - S) \bowtie T = (R \bowtie T) - (S \bowtie T)$$

To derive the first, observe that $(R \cup S) \times T = (R \times T) \cup (S \times T)$. Since, in addition, $R \bowtie S = \sigma_F(R \times S)$, where F is the equi-join condition, then

$$\begin{aligned} (R \cup S) \bowtie T &= \sigma_F[(R \cup S) \times T] = \sigma_F[(R \times T) \cup (S \times T)] \\ &= \sigma_F(R \times T) \cup \sigma_F(S \times T) = (R \bowtie T) \cup (S \bowtie T) \end{aligned}$$

The second is proven as follows. First, assume that $x \in (R - S) \bowtie T$; then we prove that $x \in (R \bowtie T) - (S \bowtie T)$. The element x is of the form $x_1 x_2$, where $x_1 \in (R - S)$ and $x_2 \in T$. Further, $x_1 \in R$ and $x_1 \notin S$. Hence $x_1 x_2 \in R \bowtie T$ and $x_1 x_2 \notin S \bowtie T$.

Second, we assume the converse and prove that $x \in (R-S) \bowtie T$. Here $x \in R \bowtie T$ and $x \notin S \bowtie T$. Consequently, $x_1 \in R$ and $x_2 \in T$, and also $x_1 \notin S$. But then $x_1 \in R - S$.

We now have

$$
\begin{aligned}
&[(R-\delta_R^-) \cup \delta_R^+] \bowtie [(S-\delta_S^-) \cup \delta_S^+] \\
&\quad = \{(R-\delta_R^-) \bowtie [(S-\delta_S^-) \cup \delta_S^+]\} \cup \{\delta_R^+ \bowtie [(S-\delta_S^-) \cup \delta_S^+]\} \\
&\quad = \{[(R-\delta_R^-) \bowtie (S-\delta_S^-)] \cup [(R-\delta_R^-) \bowtie \delta_S^+]\} \cup \{[\delta_R^+ \bowtie (S-\delta_S^-)] \cup (\delta_R^+ \bowtie \delta_S^+)\} \\
&\quad = \{\{[R \bowtie (S-\delta_S^-)] - [\delta_R^- \bowtie (S-\delta_S^-)]\} \cup [(R \bowtie \delta_S^+) - (\delta_R^- \bowtie \delta_S^+)]\} \cup \\
&\qquad \{[(\delta_R^+ \bowtie S) - (\delta_R^+ \bowtie \delta_S^-)] \cup (\delta_R^+ \bowtie \delta_S^+)\} \\
&\quad = \{\{[(R \bowtie S) - (R \bowtie \delta_S^-)] - [(\delta_R^- \bowtie S) - (\delta_R^- \bowtie \delta_S^-)]\} \cup [(R \bowtie \delta_S^+) - (\delta_R^- \bowtie \delta_S^+)]\} \cup \\
&\qquad \{[(\delta_R^+ \bowtie S) - (\delta_R^+ \bowtie \delta_S^-)] \cup (\delta_R^+ \bowtie \delta_S^+)\} \\
&\quad = (R \bowtie S) - (R \bowtie \delta_S^-) - [(\delta_R^- \bowtie S) - (\delta_R^- \bowtie \delta_S^-)] \cup [(R \bowtie \delta_S^+) - (\delta_R^- \bowtie \delta_S^+)] \cup \\
&\qquad [(\delta_R^+ \bowtie S) - (\delta_R^+ \bowtie \delta_S^-)] \cup (\delta_R^+ \bowtie \delta_S^+)
\end{aligned}
$$

The two last right hand sides contain different equivalent expressions for $DELTA(\bowtie, R, \delta_R, S, \delta_S)$. For example, using the last, we have

$$
\begin{aligned}
DELTA(\bowtie, R, \delta_R, S, \delta_S) &= \delta_{R \bowtie S} = (\delta^-_{R \bowtie S}, \delta^+_{R \bowtie S}) \\
&= ([R \bowtie \delta_S^-, (\delta_R^- \bowtie S) - (\delta_R^- \bowtie \delta_S^-)], \\
&\qquad [(R \bowtie \delta_S^+) - (\delta_R^- \bowtie \delta_S^+), (\delta_R^+ \bowtie S) - (\delta_R^+ \bowtie \delta_S^-), \delta_R^+ \bowtie \delta_S^+])
\end{aligned}
$$

The components of $\delta^-_{R \bowtie S}$ are: the deletions to $R \bowtie S$ due to deletions from S and the deletions to $R \bowtie S$ due to deletions from R, but with overlapping deletions (i.e. $\delta_R^- \bowtie \delta_S^-$) removed.

The components of $\delta^+_{R \bowtie S}$ are: (1) insertions to the outset due to tuples from R matching insertions to S, but not including tuples due to matches between insertions to S and deletions to R; (2) a component symmetric, in R and S, to (1); (3) insertions to the outset due to matches between insertions in R and insertions in S. Figure 8 shows all the constituent joins of $\delta^-_{R \bowtie S}$ and $\delta^+_{R \bowtie S}$ by means of dotted lines connecting two relations.

As can be appreciated the differential of a join is a complex query, and it can be computed in many ways [BCL86]. Techniques from multiple query optimization can be exploited [CM86, Kim84, Sel86, Mat84, Sel88b]. For example, keeping all six argument relations sorted, joins can be done interleaved, and pagewise (pipe-line join).

It is possible to use $DELTA$ with arguments as in $DELTA(\pi_A \sigma_F R, \delta_R)$, where a combined projection and selection has to be carried out. This is done by means of the combined operators of the previous subsection. Also "combined" generation of differential files directly from change requests and selections/projections is possible.

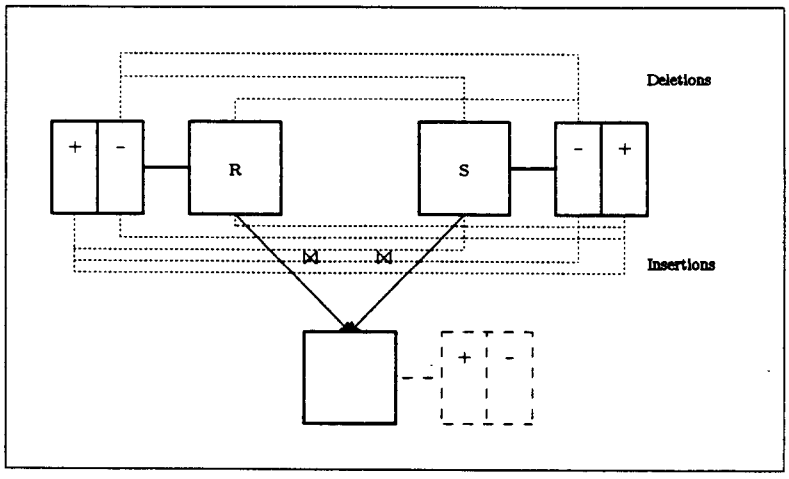

Figure 8: Computation of differentials of joins.

5.4 Incrementing/Decrementing Relations

Now we discuss how to incrementally update a relation, and again we distinguish between the base cases and the step cases.

Time Slicing Base Relations - The Base Cases

The simplest case of differential computation is time slicing of base relations, $DIF(R(t_x),(t_x,t_y))$, see figure 9. Both incremental and decremental computation are always possible (with $t_y = t_{init}$ and $t_y = NOW$, respectively).

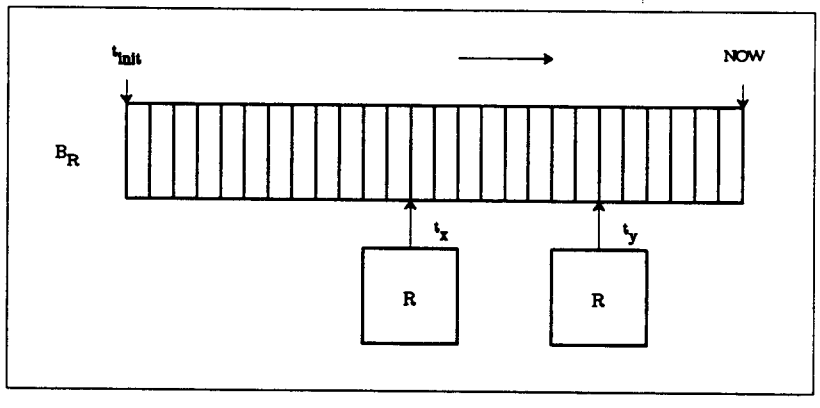

Figure 9: Time slicing a base relation.

We will discuss two basic strategies for time slice with distinct characteristics that make them useful in different contexts. The first is the simplest. Change requests are processed one at a time from the outset towards the requested state until the stamp of the next change request to be considered exceeds the time of the desired state. The result of an insertion request is that the tuple of the request is entered into the current outset, and the result of a deletion

request is that the tuple identified by the request is removed from the current state.

The second makes use of two temporary relations, δ^+ and δ^-, and it resembles the base case procedure for computation of differentials. As there, insertions are entered into δ^+ which optionally can be kept sorted and/or indexed on the key of the relation. Deletion requests can be deletions of tuples of the outset or deletions of insertions, and they make sense. Thus, the tuple to delete is in either δ^+ or the current outset; δ^+ is searched for the tuple to be deleted, and if it is found, the request is disregarded and the tuple is removed from δ^+; if not, key information is stored in δ^-. Tuples of δ^+ and δ^- are written one page at a time. When there are no more change requests to consider, δ^- is sorted, and δ^+ is sorted if it was not sorted already. Both δ-files are then simultaneous "merged" with the outset: first a page of deletions is read, then the first relevant page of the outset and the first relevant page of the insertions are read. Deletions are performed on the outset first, then relevant insertions are performed. Whenever a page is totally read the next page of the relation is read. In the case of the outset processed pages are written, and only pages that are relevant for the deletions are read (irrelevant pages can be considered processed and written already). When there are neither deletions nor insertions left, the processing terminates. Following this procedure pages of the three relations are only read once, and irrelevant pages of the outset need not be read at all.

The choice between the first and a variation of the second strategy is based on the characteristics of the arguments, i.e. the size of the outset used, and the differential file. IM/T contains a component that, given the name of a backlog and a start and an end time, returns estimates: the number of insertions, the total number of change requests, and the number of deletions of insertions. The input to the component is produced during from non-eager processing of change requests. If the first strategy is used, counts of insertions and deletions are used; if the second strategy is used, again counts of insertions and deletions are available, but so is also the final number of insertions. How these inputs are most efficiently used to generate the output is a topic of current research.

The first strategy is advantageous if the total number of change requests to be processed is low. The choice of keeping δ^+ sorted or not depends on the number of insertions into δ^+ compared to the number of deletions to be processed against δ^+. If sorting is adopted, insertion has an overhead, and if not, then search for deletions must be done by sequential scan.

The Step Cases

There are three cases for data arguments (figure 10).

Selection, $DIF(\sigma_F R, \delta_R)$, can be computed as follows:

$$DIF(\sigma_F R, \delta_R) = (\sigma_F R - \sigma_F \delta_R^-) \cup \sigma_F \delta_R^+$$

This is correct, since $\sigma_F \delta_R = \delta_{\sigma_F R}$ (see page 12).

When we consider projection there are two subcases, the identity projection, $DIF(R, \delta_R)$, and non-trivial projections, $DIF(\pi_A R, \delta_R)$. The computations are:

$$DIF(R, \delta_R) = (R - \delta_R^-) \cup \delta_R^+ \text{ and } DIF(\pi_A R, \delta_R) = (\pi_A R - \pi_A \delta_R^-) \cup \pi_A \delta_R^+$$

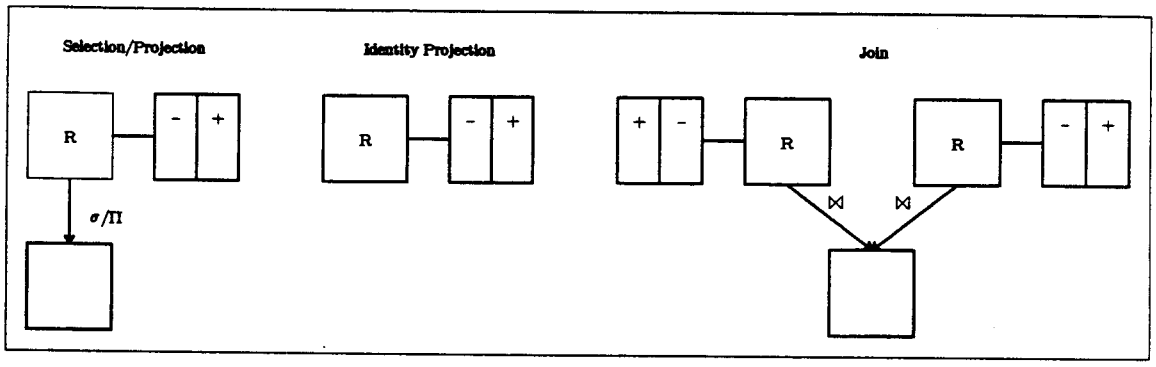

Figure 10: Differential computation: selection, projection, and join. For each case the arguments and their relations are shown. Note that the (non-bold) "R" of "Selection/Projection" is not an argument.

The final case is the incremental join, $DIF(R \bowtie S, R, \delta_R, S, \delta_S)$. From the subsection about the operator DIF, we have:

$$\begin{aligned} & DIF(R \bowtie S, R, \delta_R, S, \delta_S) \\ &= (R \bowtie S) - \underbrace{(R \bowtie \delta_S^-)}_{1} - \underbrace{[(\delta_R^- \bowtie S) - (\delta_R^- \bowtie \delta_S^-)]}_{2} \cup [(R \bowtie \delta_S^+) - (\delta_R^- \bowtie \delta_S^+)] \cup \\ & \quad [(\delta_R^+ \bowtie S) - (\delta_R^+ \bowtie \delta_S^-)] \cup (\delta_R^+ \bowtie \delta_S^+) \end{aligned}$$

Let us consider processing of the deletions to the outset. The two components can be explained as follows: (1) $R \bowtie \delta_S^-$ are all the deletions from the outset due to deletions to S; (2) $(\delta_R^- \bowtie S) - (\delta_R^- \bowtie \delta_S^-)$ are all the deletions to the outset due to deletions to R with duplicate deletions due to overlaps between δ_R^- and δ_S^- and already included in (1) removed. The overlaps can be ignored without affecting the correctness of the final result, and the deletions represented by the two remaining terms can be performed using only $R \bowtie S$, δ_R^-, and δ_S^-. A tuple of the outset is of the form $x_R x_S$, where x_R is a tuple compatible with R and x_S is a tuple compatible with S. Tuples of $R \bowtie S$ where x_S match a tuple in δ_S^- are simply deleted; similarly tuples where x_R match a tuple in δ_R^- are deleted. No joins need be performed.

Now, let us turn to the insertions. It is instructive to reformulate the expression for $(R \bowtie S)'$:

$$\begin{aligned} (R \bowtie S)' &= (R - \delta_R^-) \bowtie (S - \delta_S^-) \cup [(R - \delta_R^-) \bowtie \delta_S^+] \cup \{\delta_R^+ \bowtie [(S - \delta_S^-) \cup \delta_S^+]\} \\ &= R \bowtie S - \delta_{R \bowtie S}^- \cup [(R - \delta_R^-) \cup \delta_R^+ - \delta_R^+] \bowtie \delta_S^+] \cup \{\delta_R^+ \bowtie [(S - \delta_S^-) \cup \delta_S^+]\} \\ &= R \bowtie S - \delta_{R \bowtie S}^- \cup \{[(R - \delta_R^-) \cup \delta_R^+] \bowtie \delta_S^+\} - [\delta_R^+ \bowtie \delta_S^+] \cup \{\delta_R^+ \bowtie [(S - \delta_S^-) \cup \delta_S^+]\} \\ &= R \bowtie S - \delta_{R \bowtie S}^- \cup \underbrace{\{[(R - \delta_R^-) \cup \delta_R^+] \bowtie \delta_S^+\}}_{1} \cup \underbrace{[\delta_R^+ \bowtie (S - \delta_S^-)]}_{2} \end{aligned}$$

The insertion, $\delta_{R \bowtie S}^+$, now is defined by two joins. The first (1) has δ_S^+ as one argument, and the second (2) has δ_R^+ as one argument. This explains the superiority of differential computation when differentials are small and relations large, because in such cases an expensive join of two large relations, R' and S', is avoided and two joins of a small

relation with a large relation is done instead[7].

See [Rou89] and [Sta89] where algorithms, costs, and efficient implementation of incremental join for pointer views in ADMS are discussed in detail.

6 Pruning the Search Space

We already have presented a complete framework for query optimization. Here we introduce the concept of pruning a STN. Pruning is a means of further optimization of plan selection. The motivation is to reduce the sizes of the STNs generated without leaving out promising query plans. Reduced STNs mean reduced costs of estimating costs of single transitions and a smaller argument of the dynamic programming algorithm which therefore executes more efficiently. The purpose of introducing the mapping P in the definition of a STN was exactly to be able to include pruning into the framework. The rules of this section restrict the number of possible transitions at a state.

The rules below illustrate the kind of rules that can be integrated into IM/T. Rules from standard query optimization [Ull82] can be applied, too.

Rule 1 *Only apply a differential to its outset if exactly the selections/projections performed on the outset have been performed on the differential, too.* Obeying this rules will ensure we do selections/projections on only the outset or the differential, and never on the updated outset. This is reasonable since at least the differential can be assumed to be much smaller than the updated outset.

rule 2 *Apply operators as early as possible.* If the arguments in state x_b of an operation p transforming x_b into x_c are present in an predecessor state, x_a, of x_b then p should be applied to x_a instead of to x_b.

rule 3 *Only compute a differential of an outset, it the outset already exists.* Both sequences are possible, but a STN should only include one of them, and a differential is not useful if the outset is not available.

rule 4 *Application of maximal combined operators is preferable to the sequential application of the constituent operators of the combined operators.*

rule 5 *Only use the smallest cached result out of covering results equally outdated with respect to the desired state.* This and the following rule attempt to only consider the most promising cached results during generation af a STN.

rule 6 *Only use the least outdated cached result out of covering results of equal size.*

7 Conclusion and Future Research

Extending the relational model to automatically record transaction time is not a new idea, but implementing the extended model by storing the complete history of change in relation backlogs is. Such an implementation will support not only queries on previous database states, but queries on the nature of change itself.

[7] Differential and re-computation both involve additional processing apart from joins, but since join is the most expensive operation we ignore this.

We expect queries on the nature of change to play a key role in future information systems. With ever increasing amounts of constantly changing information it will be impossible for an individual user to digest all the information that pertains to a given situation and stay abreast with all the changes of it. We will see applications where the user is not interested in the current state of the database and the changes made to it, as long as they are both normal. On the other hand, if the current state of the database or the change made to it is abnormal, then the user is interested and must be notified. The price paid for the added functionality is a substantial increase of space consumption and a decrease of query processing efficiency.

The topic of this paper has been efficient support of transaction time in the relational model. The concrete results include:

- a transparent extension of the relational model (DM/T), where the transparency is supported by the underlying implementation (IM/T)
- a general query optimization and processing framework which utilizes partitioned backlog storage, selective pointer and data view caching, eager/lazy view update, cache indexing, and state transition networks with dynamic programming.
- integration of recomputation and differential computation of queries
- a symmetrical, general notion of differential computation integrating incremental and decremental computation
- formulas for differential computation
- a generalization of the notion of query subsumption to utilize differential computation
- augmentation of standard query optimization with rules for optimization of differential query processing

We are currently investigating a vacuuming subsystem to control the increased space requirements. This subsystem will enforce data vacuuming rules specified in terms of the temporal query language. Other research topics include caching of differential files, caching policies, statistics for query optimization, optimal algorithms for operators of STNs, and support for general versioning.

In the references below, the following abbreviations are used: **UNC:** Dept. of Comp. Sci., Univ. of North Carolina. Chapel Hill, NC 27599-3175. **LBL:** Inf. and Comp. Sci. Div., Lawrence Berkeley Laboratory, 1 Cyclotron Road, Berkeley, California 94720. **UM:** Dept. of Comp. Sci., Univ. of Maryland, College Park, MD 20742.

References

[Ahn86] Ilsoo Ahn. Performance Modeling and Access Methods for Temporal Database Management Systems. TR86-018, *UNC*, August 1986.

[BADW82] A. Bolour, T. L. Anderson, L. J. Dekeyser, and H. K. T. Wong. The Role of Time in Information Processing: A Survey. *ACM SIGMOD Record*, 12(3):27-50, April 1982.

[Bas85] M. A. Bassiouni. Data Compression in Scientific and Statistical Databases. *IEEE TSE* SE-11(10):1047-1058, October 1985.

[BCL86] Jose A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. TR, CS-86-17, Comp. Sci. Dept., Univ. of Waterloo, May 1986.

[BLC89] Jose A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM TODS*, 14(3):369-400, September 1989.

[Bub77] Janis A. Bubenko, jr. The Temporal Dimension in Information Modeling. in G. M. Nijssen (ed.) Architecture and Models in Data Base Management Systems, North Holland, 1977.

[Chr87] Stavros Christodoulakis. Analysis of Retrieval Performance for Records and Objects Using Optical Disk Technology. *ACM TODS*, 12(2):137-169, June 1987.

[CM86] U. S. Chakravarthy and J. Minker. Multiple Query Processing in Deductive Databases using Query Graphs. *VLDB '86*, pages 384-391.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM* 13(6):377-387, June 1970.

[Cod79] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM TODS*, 4(4):397-434, December 1979.

[DLW84] P. Dadam, V. Lum, and H. D. Werner. Integration of Time Versions into a Relational Database System. *VLDB '84*, pages 509-522.

[GS89] Himawan Gunadhi and Arie Segev. A Framework for Query Optimization in Temporal Databases. TR, LBL-26417, *LBL*, 1989.

[GSS89] Himawan Gunadhi, Arie Segev, and George J. Shantikumar. Selectivity Estimation in Temporal Databases. TR, LBL-27435, *LBL*, 1989.

[Han87] Eric N. Hanson. A Performance Analysis of View Materialization Strategies. *ACM SIGMOD '87*, pages 440-453.

[HW89] Wei Hong and Eugene Wong. Multiple Query Optimization Through State Transition and Decomposition. Memorandum, UCB/ERL M89/25, *Elec. Research Lab., Coll. of Eng., Univ. of California, Berkeley*, March 1989.

[Jhi88] Anant Jhingran. A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures. *VLDB '88*, pages 88-99.

[JK84] Matthias Jarke and Jurgen Koch. Query Optimization in Database Systems. *Comp. Surveys*, 16(2):111-152, June 1984.

[JM89] Christian S. Jensen and Leo Mark. Queries on Change in an Extended Relational Model. CS-TR-2299, UMIACS-TR-98-80, *UM*, August 1989. Subm. for publ.

[JMR89] Christian S. Jensen, Leo Mark, and Nick Roussopoulos. Incremental Implementation Model for Relational Databases with Transaction Time. CS-TR-2275, UMIACS-TR-8963, *UM* June 1989. Subm. for publ.

[KD79] Kathryn C. Kinsley and James R. Driscoll. Dynamic Derived Relations Within the RAQUEL II DBMS. *ACM Annual Conference '79*, pages 69-80.

[KD84] Kathryn C. Kinsley and James R. Driscoll. A Generalized Method for Maintaining Views. *ACM Annual Conference '84*, pages 587-593.

[Kim84] Won Kim. Global Optimization of Relational Queries: A First Step. In Won Kim, David S. Reiner, and Don S. Batory (eds.), Query Processing in Database Systems, Pages 206-216, Springer 1984.

[KS89] Curtis Kolovson and Michael Stonebraker. Indexing Techniques for Historical Databases. *Data Eng. '89*, pages 127-137.

[LDE*84] V. Lum, R. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch. H. Werner, and J. Woodfill. Designing DBMS Support for the Temporal Dimension. *ACM SIGMOD '84*, pages 115-130.

[LW86] Stephane Lafortune and Eugene Wong. A State Transition Model for Distributed Query Processing. *ACM TODS*, 11(3):294-322, September 1986.

[LY85] H. Z.Yang and P.-Å. Larson. Computing Queries from Derived Relations. *VLDB '85*, pages 259-269.

[Mat84] Matthias Jarke. Common Subexpression Isolation in Multiple Query Optimization. In Won Kim, David S. Reiner, and Don S. Batory (eds.), Query Processing in Database Systems, Pages 191–205, Springer 1984.

[MB85] A. Mahanti and A. Bagchi. AND/OR Graph Heuristic Search Methods. *JACM*, 32(1):28-51, January 1985.

[McK88] Leslie Edwin McKenzie. An Algebraic Language for Query and Update of Temporal Databases. TR88-050, *UNC*, October 1988.

[MS89] Edwin McKenzie and Richard Snodgrass. An Evaluation of Algebras Incorporating Time. TR-89-22. *Dept. of Comp. Sci., Univ. of Arizona*, September 1989.

[Ric83] Elaine Rich. Artificial Intelligence. *McGraw-Hill*, 1983.

[RK86] Nick Roussopoulos and Hyunchul Kang. Principles and Techniques in the Design of *ADMS±*. *Computer*, 19(12):19-25, December 1986.

[RND] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. Combinatorial Algorithms: Theory and Practice. *Prentice-Hall.*

[Rou82a] Nick Roussopoulos. View Indexing in Relational Databases. *ACM TODS*, 7(2):258-290, June 1982.

[Rou82b] Nick Roussopoulos. The Logical Access Path Schema of a Database. *IEEE TSE*, 8(6):563-573, November 1982.

[Rou87] Nick Roussopoulos. Overview of ADMS: A High Performance Database Management System. *Fall Joint Comp. Conf. '87*, pages 452-460.

[Rou89] Nick Roussopoulos. The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis. UMIACS-TR-89-15, CS-TR-2193 *UM*, February 1989.

[RS87a] Doron Rotem and Arie Segev. Physical Organization of Temporal Data. *Data Eng. '87*, pages 547-553.

[RS87b] Lawrence A. Rowe and Michael R. Stonebraker (eds.). The Postgres Papers. Memorandum, UCB/ERL M86/85, *Elec. Res. Lab., Coll. of Eng., Univ. of California at Berkeley.*, June 1987.

[SA85] Richard Snodgrass and Ilsoo Ahn. A Taxonomy of Time in Databases. *ACM SIGMOD '85*, pages 236-246.

[SA88] Richard Snodgrass and Ilsoo Ahn. Partitioned Storage for Temporal Databases. *Inf. Syst.*, 13(4):369-391, 1988.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. *ACM SIGMOD '79"*, pages 82-93.

[SC75] John Miles Smith and Philip Yen-Tang Chang. Optimizing the Performance of a Relational Algebra Interface. *CACM* 18(10):569-579, October 1975.

[Sed88] Robert Sedgewick. Algorithms. *Addison-Wesley*, 2nd. ed., 1988.

[Sel86] Timos Sellis. Global Query Optimization. *SIGMOD '86*, pages 191-205.

[Sel87] Timos K. Sellis. Efficiently Supporting Procedures in Relational Database Systems. *ACM SIGMOD '87*, pages 278-291.

[Sel88a] Timos K. Sellis. Intelligent Caching and Indexing Techniques for Relational Database System. *Inf. Syst.*, 13(2):175-185, 1988.

[Sel88b] Timos K. Sellis. "Multiple-Query Optimization. *ACM TODS* 13(1): 23-52, March 1988.

[SF89a] Arie Segev and Weiping Fang. Concurrency-based Updates to Distributed Materialized Views. TR-LBL-27359, *LBL*, 1989.

[SF89b] Arie Segev and Weiping Fang. Optimal Update Policies for Distributed Materialized Views. TR-LBL-26104 *LBL*, 1989.

[SG89] Arie Segev and Himawan Gunadhi. Event-join Optimization in Temporal Relational Databases. TR-LBL-26600, em LBL, 1989.

[Sha86] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM TODS*, 11(3):239-264, September 1986.

[SK86] Arie Shoshani and Kyoji Kawagoe. Temporal Data Management. *VLDB '86*, pages 79-88.

[SL89] B. J. Salzberg and D. Lomet. Access Methods for Multiversion Data. *ACM SIGMOD '89*, pages 315-324.

[Sno87] Richard Snodgrass. The Temporal Query Language TQuel. *ACM TODS*, 12(2):247-298, June 1987.

[SR88] Jaideep Srivastava and Doron Rotem. Analytical Modeling of Materialized View Maintenance. TR, *LBL* February 1988.

[SS85] Timos K. Sellis and Leonard Shapiro. Optimization of Extended Database Query Languages. *ACM SIGMOD '85*, pages 424-236.

[SS88] Robert B. Stam and Richard Snodgrass. A Bibliography on Temporal Databases. *Data Eng.*, 7(4):53-61, December 1988.

[Sta89] Antonios G. Stamenas. High Performance Incremental Relational Databases. UMIACS-TR-89-49, CS-TR-2245, *UM*, May 1989.

[TB88] Frank Wm. Tompa and Jose A. Blakeley. Maintaining Materialized Views Without Accessing Base Data. *Inf. Syst.*, 13(4):393-406, 1988.

[Ull82] Jeffrey D. Ullman. Principles of Database Systems. *Computer Science Press*, 2nd. ed., 1982.

[WY76] E. Wong and K. Youseffi. Decomposition - A Strategy for Query Processing. *ACM TODS* 1(3):223-241, September 1976.