THE UNIVERSITY OF AALBORG

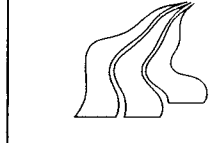
A Framework for Efficient Query Processing Using Caching, Cachae Indexing, and Differential Techniques in the Rational Model Extended with Transaction Time

by

Christian S. Jensen, Leo Mark, Nick Roussopoulos & Timos Sellis

R 89-45 December 1989

INSTITUTE FOR ELECTRONIC SYSTEMS
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
Strandvejen 19 — DK 9000 Aalborg — Denmark
Tel.: +45 98 13 87 88 — TELEX 69 790 aub dk



ISSN 0106-0791

A Framework for Efficient Query Processing Using Caching, Cache Indexing, and Differential Techniques in the Relational Model Extended with Transaction Time

Christian S. Jensen*

Leo Mark Nick Roussopoulos Timos Sellis†

December 1989

Abstract

We present a framework for query processing in the relational model extended with transaction time. The framework integrates standard techniques for query optimization and computation with techniques for incremental and decremental, i.e. differential, computation from cached and indexed results of previous computation in order to provide efficient processing of queries on 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, the cache index, and intermediate results; 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 that are not promising, and dynamic programming techniques are used to identify the optimal plan of the resulting state transition network. 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.

The framework exploits eager, threshold triggered, and eager propagation of update to ensure consistency between base data and cached data. It integrates previously proposed approaches to supporting views, i.e. recomputation, storage of data snapshots, and storage of pointer structures, and it generalizes incremental computation techniques to differential computation techniques.

*University of Aalborg, Institute for Electronic Systems, Department of Mathematics and Computer Science, Strandvejen 19.2, DK-9000 Aalborg, Denmark

†University of Maryland, Department of Computer Science, College Park, Maryland 20742, USA. The work has taken place at University of Maryland and was supported by NSF IRI-8719458, and AFOSR-89-0303. Correspondence via e-mail to jensen@brillig.umd.edu

Categories and Subject Descriptors: H.2.1 [**Database Mangement**]: Logical Design *data models* H.2.2 Physical Design *access methods* H.2.3 Languages *query languages* H.2.4 Systems *query processing* H.3.2 [**Information Storage and Retrieval**]: Information Storage *file organization* H.3.3 Information Search and Retrieval *clustering, query formulation, retrieval models*

General Terms: Design, Performance, Algorithms

Additional Keywords and phrases: relational model, transaction time, query processing, rules, persistent views, view cache, eager, lazy, logical access path, incremental/decremental computation, dynamic programming, state transition networks

1 Introduction

The introduction has three parts. First, we introduce the subject of the paper along with some motivation. Second, we discuss the relation to previous research efforts, and third, we outline the structure of the paper.

1.1 The Main Ideas

The relational model as formulated by E. F. Codd about 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, is a disadvantage as well, because the relational model does not provide detailed, and customized, support for its 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 proposed.

This paper presents an implementation model, IM/T, for an extension of the relational model supporting transaction time, DM/T [JMR89, JM89]. Once entered into a database of a data base management system (DBMS) supporting DM/T, data are never deleted, and it is possible to see the database as of any time during the past. Consequently accounting for the past is supported; decision support tools are provided with access to historical data, and future business policy can be based on detailed analysis of the past. It is possible to relate past states and analyze change history. Many applications will benefit from efficient transaction time support. In the literature, econometrics, banking, inventory control, medical records, and airline reservations have been mentioned as candidates [MS89]. Also, engineering applications will benefit.

As can be imagined, efficient transaction time support is not without complications. Retaining not only the current state, but any past state makes relations very large and ever growing. Traditional implementation models are not sufficient to 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. Partitioned storage is inflexible in that only recent data are accessed efficiently. The motivation for transaction time support, in the first place, was that historical data actually were of interest. While still allowing for partitioned storage, IM/T organizes data as changes to frequently accessed states, recent or old, of individual relations thus providing efficient support of any state.

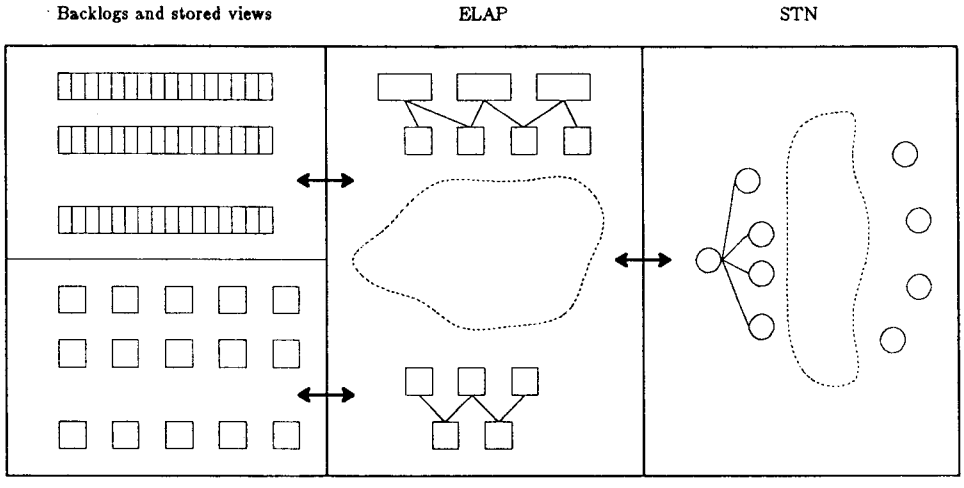


Figure 1: Overview of IM/T. There are three stores, one for backlogs containing base data, one for materialized views, and one for the ELAP containing statistics and representing the structure of base and derived data. During query optimization plans using the stored data are enumerated in STN's.

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 doing expensive computations over and over again. 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 store results. In IM/T query results can be stored as actual data or as pointers, pointing, possibly via several levels of indirection, to base data. Pointer cache storage gives a fixed, small tuple size and makes results very compact thus allowing for efficient use of main memory. In extreme cases, however, to retrieve a result stored as pointers, one base data page must be read for each pointer. Data cache storage solves the potential problem, because it

allows for control of locality of reference, and it in addition allows to eliminate references from the cache to base data on slow storage.

Second, the utility of caching can be improved by means of cache indexing. The index of IM/T, the Extended Logical Access Path (ELAP), allows for efficient identification of all results potentially useful during processing of queries. It can be thought of as a persistent query graph with nodes for all cached results. Apart from a reference to the result, a node contains statistics about the result useful during query processing. In addition, the ELAP contains nodes for results not cached, if statistics are available.

Third, to gain full benefit of caching, results should be used in conjunction with differential computation techniques. The application of such techniques prolongs the lifetime of cached views because slightly outdated views need not be discarded and then recomputed, but can instead be efficiently incremented/decremented to the requested state. IM/T generalizes incremental computation to differential computation (i.e. both incremental and decremental) and unifies traditional recomputation and differential computation.

Fourth, the issue of cache management must be dealt with. 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. A result is outdated if it does not correctly reflect its defining query expression given the current state of the base data. In IM/T cache update is done differentially. Also any strategy ranging from eager (i.e. when relevant base data are entered) over threshold triggered to lazy (when the result of the defining expression is requested) update is possible. Eager update favors retrieval speed at the expense of overall work load and insertion speed. Lazy update, on the other hand, favors insertion and overall work load at the expense of retrieval speed. We can control these trade-offs. The details of cache update are not part of this paper.

In IM/T there is a *choice* of caching computed queries. The motivation is that neither no caching (and recomputation) nor caching of all computed queries (and differential computation) is superior to the other in every given situation. In particular, when the database environment is characterized by relatively many updates, recomputation is superior. This is so because update of existing views becomes costly when there are many updates to be processed. Caching is attractive in environments characterized by (1) many queries and few updates, (2) very large underlying base relations, (3) comparably small views.

In a temporal setting the maintenance of stored views is likely to be more feasible than in snapshot settings. This is so because (1) relations are large because previous states are retained (2) different

kinds of views have different characteristics when it comes to storage feasibility; for example are fixed views primary candidates of caching because they never get outdated (3) the future outdatedness of time-dependent views issued against past states can be estimated at the time of computation. The temporal setting provides for essential additional semantics for the process of selective caching of views.

Now, let us outline how cached results, differential computation, and recomputation are integrated in query processing in IM/T. Query processing consists of (1) transformation into an internal representation, (2) query plan generation, (3) plan selection, and (4) query computation.

We use relational algebra for the internal representation of queries. Query plan generation in IM/T is based on the general concept of State Transition Networks (STN's). An STN is a directed acyclic graph. To optimize a query, IM/T generates a STN where the initial node contains the uncomputed query. A transition takes place when cost estimation of a partial computation towards the total computation takes place. In this way a cost is associated with each transition. A final state is reached when there are no more computations to estimate. By following each path from the initial to the final state, costs of alternative processing strategies can be accumulated, and upon comparison, the most promising can be chosen for actual processing. We use dynamic programming for plan selection. See figure 1.

Candidate results from the cache are used, and both recomputation and differential computation versions of the operators of the query language of DM/T are possible transitions. 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.

In the context of very large relations, accessing data is likely to be costly compared to accessing statistics (meta data) and main memory processing. Therefore, it is desirable to spend considerable efforts during query optimization. Never the less, we put forward a set of rules aimed at pruning the STN's generated, the idea being to avoid generating inferior paths and thus save both cost estimation time and space. The rule set includes standard rules for query optimization, and new rules. Therefore, IM/T builds on and further extend standard optimization techniques.

1.2 Related Work

The work presented in this paper touches upon several issues within databases that have been subject to previous research, including design of temporal data models, storage and retrieval of temporal data, caching techniques, support for persistent views, incremental computation techniques, query

optimization, and state transition networks. In this subsection we outline related research in these areas.

This paper is about optimizing and computing queries of the data model, DM/T, which was originally presented in [JMR89], and its expressive power was investigated in [JM89]. One of the major concerns in the design of DM/T was to provide a small yet powerful extension of the relational model. Consequently, DM/T was designed to be completely transparent to a naive user only expecting a standard relational model. Although increasingly intense efforts have been put into design of temporal extensions of the relational model (see, for instance, [SS88, SA85, BADW82, Sno87, MS89] for surveys and further references) none so far shares this characteristic.

Efficient implementation of transaction time data models is difficult and has not been addressed until recently. As already mentioned, traditional implementation models fall short of meeting the requirements for efficient support of transaction time. Their performance is inversely proportional to the total amount of data. While the total amount grows, the amount of current data is a relatively stable quantity. This observation, and the assumption that current data are needed more frequently and urgently than old data, served as the motivation for partitioned storage, where tuples of relations are partitioned according to the values of their time attribute. To avoid waste of space, the technique of storing previous states as changes to the current has been advocated (reverse chaining). Then old states can be decrementally computed. Aspects of partitioned storage, in different settings, have been addressed by [DLW84, LDE+84, Ahn86, SA88, SL89]. We regard it as one step along the path towards efficient and flexible support. Therefore we include it in IM/T where both reverse and forward chaining are possible.

In [KS89] the problem of maintaining R-tree like indices on historical data migrating from magnetical disk to write-once read-many optical disk is discussed. Indices residing on either magnetical disk only, on optical disk only, or on both types of disks are compared.

Grid files have been suggested as a means of implementation [SK86], but seems inappropriate since surrogates, for which no natural ordering exists, should be one dimension (and time the other). In addition, indexing of other attributes is not allowed, which again is unsatisfactory. The work reported in [RS87a] is similar in that data of a temporal database is perceived as multi dimensional, time being one dimension. The subject of that paper is the problem of multi dimensional file partition for static files.

The research reported in [GS89, SG89, GSS89] is concentrated on different kinds of temporal joins (time-union, time-intersection, and event-joins) and temporal selectivity estimation. We do

not address these issues.

The work most closely related to the present is presented in [McK88] where a data model supporting both transaction time and what is termed valid time is formally defined, and an implementations based on incremental techniques is discussed. Our work share the basic assumptions behind the discussion of implementations, i.e. application of incremental techniques based on persistent views, and the integration of a logical access path into the framework. The focus of [McK88] is, however, different from ours. His main focus is on the data model level. As a part of those efforts incremental relational algebra operators, resembling those of our state transition space, are formally defined. In addition a thorough survey of previous efforts on applying incremental techniques in settings supporting the standard relational model is given, and this is extended into a discussion of ways to combine the benefits of previous efforts into an advantageous implementation in a setting where both transaction time and valid time is supported. Our work contrasts by concentrating on implementation and on transaction time, only. We present a detailed design of an implementation model where the main concern is the description of a framework for query optimization and processing.

IM/T exploits caching of views. There are many aspects of this technique and the literature contains many contributions to its understanding.

The concept of snapshot addressed in [AL80, Adi81] resembles that of a view. A snapshot is defined in terms of a query expression involving base relations, and it is persistent. However, once computed it is separated from its defining relations and perceived as a base relation on its own; it can not be updated, but can be refreshed, i.e. recomputed. Since a snapshot shows a selected part of the database "as of" a past time it can be thought of as a primitive support for transaction time.

In [SR88] an analytical queueing model and a classification of algorithms for maintenance of materialized views is presented. The model enables policies for when to update views to be proven optimal for differing processing environments, considering both response time and processing cost.

Several researchers have applied the idea of materialized views to processing in distributed settings, where materialized views can be seen as a compromize between fully synchronized replicated data and single copies of data. The problems of how and when to update views have been addressed. One of the most recent publications is [SF89b], where the problem of determining optimal policies for updating distributed materialized views is addressed, and an analytical model incorporating a minimum-cost objective function is put forward. Optimization is constrained by specifications of maximal allowable user response times, maximal allowable outdatedness (currency) of materialized views, and a particular view update policy. See this work (and [SF89a]) for expositions to aspects of materialized views relevant to distributed processing.

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 at all times. This supports the decision of including them all in IM/T.

In [Sel87, Sel88a] caching of query results is addressed, the motivation being to support efficiently query language procedures (programs, rules) stored in relational fields. Which results of procedure invocations to cache and cache replacement strategies are relevant topics addressed. See also [Jhi88], where separate caching of results of procedure invocations is compared and found generally superior to caching in tuples.

Techniques aimed at reducing the cost of maintaining materialized views have most recently been reported in [BCL86, BCL89, 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. We do not address these techniques.

For references to work on general caching (i.e. caching of disk blocks of data), see [AHH89] where an analytical cache model is proposed, [EB84], where the conventional buffer is generalized to a database cache and issues related to transaction management are addressed, and [RD89] where frequency based replacement is explored.

IM/T generalizes and unifies traditional recomputation and incremental computation into a setting where a single query can be computed partly using both recomputation, incremental computation, and decremental computation. Traditional systems use recomputation (e.g. Ingres [WY76], System R [SAC+79], and more recent systems). 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($\pm$), a database management system implementing the standard relational model, incremental computation of views stored as pointer structures is used [Rou82, Rou89, Rou87]. The system employs deferred update based on differential files. In the database sytem Cactis, based on the Cactis Data Model, a two-pass graph traversal, incremental update algorithm for derived attributes is used [DKH]. In the first pass of the lazy algorithm, outdated derived attributes of an attribute dependency graph are identified; in the second pass recomputation of only identified attributes is done.

There is some resemblance to Postgres, where previous history also is retained. The temporal support, however, never was the point of focus, and neither time stamps nor backlog queries including distinction of insertions, deletions, and modifications is supported. Postgres exploits caching, but

since indexing and differential cache maintenance and query execution are missing, the full potential of caching is not achieved [RS87b].

IM/T benefits from and builds on previous work on query optimization as reported in e.g. [SC75, SAC+79, JK84, JKS84, SS85, STNO85].

State Transition Networks have to our knowledge never been applied in a temporal setting and in settings involving caching. In [LW86] STN's are used as a framework for query optimization in a distributed environment. There a state is a vector of the same dimension as the number of sites, and each entry contains the relations located at a site. A transition happens when a set of pairs of relations in one state are joined and then possibly projected and selected. In order to apply dynamic programming to the problem of finding optimal paths in STN's, the separation assumption is made. Therefore, it is not possible to take into account the physical ordering of tuples in relations, and the contribution is limited to a strictly logical level. In [HW89] STN's have been applied to multiple query optimization. The particular state and transition spaces of IM/T are new, and so, consequently, are the issues that arrive from dealing with them.

1.3 Structure of the Paper

The next section, **Transaction Time in the Relational Model, DM/T**, section 2, presents the data model of IM/T and serves as a specification of the functionality to be supported. First, the transaction time concept of DM/T is characterized; second, the data structures of the model and third, the query language is presented; fourth, physical and logical aspects of the data model are related to the standard relational model. The remaining part of the paper is devoted to IM/T, i.e. the efficient processing of DM/T queries.

Section 3, **Structures of the Implementation Model, IM/T**, describes the three stores of IM/T. First, storage of base data is considered. Second, the architecture of the cache is presented, and third, the ELAP is described.

Section 4, **Query Plan Generation and Selection**, is the most central part of the paper. First, the general notion of State Transition Network is presented as a means of enumerating alternative query plans. Second, the graph search problem of how to collect costs of entire plans from costs of single transitions is discussed. We choose a dynamic programming approach. Third, the concrete state and transition spaces incorporating the use of cached results and differential computation along with standard query computation techniques and support for combined operators is presented. Then, fourth, an example that shows how the STN of a query, for a specified state of the cache, is

generated. Fifth, it is discussed how the ELAP is used to find promising results from the cache, to be considered when STN's are generated. Finally, some of the trade-offs made during the design of IM/T are discussed.

In section 5, **Implementation of Operators of STN's**, we look at the operators of STN's in more detail, and point to ways of implementation. An overview ot the operators is given, genral assumptions are presented, and the three types of operators, recomputation operators, operators that construct differential files, and differential operators, are discussed in turn.

The topic of section 6, **Pruning the Search Space**, is further query optimization. Rules, to be integrated into the framework of section 4, that specify preferred state transitions, are presented.

Section 7, **Conclusion and Future Research**, is the final section. We review the features of the implementation model point to an extensive set of very interesting current and future areas of research.

2 Transaction Time in the Relational Model, DM/T

In this section we briefly introduce the data model DM/T, a transaction time extension of the basic relational model [Cod70, Cod79], which was first introduced in [JMR89].

The purpose is to define the functionality to be supported by the query evaluation system, i.e. the kind of queries to efficiently support. We only present the data structures and the query language of DM/T and will mainly focus on how it differs from the basic relational model.

2.1 Transaction Time Concept

In this subsection we briefly characterize the the time concept supported by DM/T.

Within the area of temporal databases two orthogonal time dimensions have been studied [SA85]. *Logical time* models time in the 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.

DM/T supports transaction time. Figure 2 characterizes the particular transaction time concept we have chosen.

First, the time domain is characterized as 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. We support an irregular time domain.

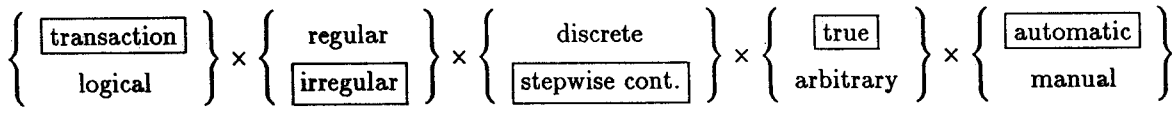


Figure 2: The time concept supported by DM/T.

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. Our time domain has this property, because, until a relation is changed by another transaction, the data as they were after the previous transaction are valid, i.e. they are part of the current state of the database.

Fourth, we support 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.

Fifth, we finally have 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.

2.2 Backlogs, Base Relations and Views

Here we present the data structures of DM/T as seen by the user. These structures are the arguments that can be manipulated by the operators of DM/T.

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].

The schema of relation R and its corresponding backlog is shown in figure 3.

Each tuple in a backlog is a *change request*. As shown, B_R contains three attributes in addition to the attributes of R. *Id* is defined over a domain of logical, system generated unique identifiers, i.e. surrogates. The values of *Id* represent the individual change requests, they can be referenced but not read by users/application programs. 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. Finally, the attribute *Time* is defined over the domain of

Relation name	R
Attribute name	*Domain name*
A_1	D_1
A_2	D_2
...	...
A_n	D_n

Relation name	B_R
Attribute name	*Domain name*
Id	*SURROGATE*
Op	{*Ins, Del, Mod*}
Time	*TTIME*
A_1	D_1
A_2	D_2
...	...
A_n	D_n

Figure 3: Schema for the relation R and its backlog, B_R.

transaction time stamps, *TTIME*, as discussed in the previous subsection.

A DBMS supporting DM/T automatically generates and maintains a backlog for each base relation (i.e. user defined relations and schema relations). Figure 4 shows the effect on backlogs resulting from operation requests on their corresponding relations. When an insertion into R is requested the tuple to be inserted is entered into B_R. When a deletion is requested key information is entered into the backlog and in the case of modification both key information and new values are inserted into the backlog[1].

The Effect of Requested Operations on Backlogs	
Requested operation on R:	Effect on B_R:
insert R(tuple)	insert B_R(id, *Ins*, time, tuple)
delete R(k)	insert B_R(id, *Del*, time, tuple(k))
modify R(k, new value)	insert B_R(id, *Mod*, time, tuple(k,new value))

Figure 4: System controlled insertions into a backlog. The function "tuple" returns the tuple identified by its argument.

As a consequence of the introduction of time stamps, a base relation is now a function of time. To retrieve 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:

$$R(t_{init}) \stackrel{\text{def}}{=} R_{init}$$

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

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

When the database is initialized, it has no history and it is in an initial state, possibly with every relation equal to the empty set. If R is parameterized with an expression that evaluates to a time value, the result is the state of R as it was at that point in time. It has no meaning to use a time from before the database was initialized and after the present time. If R is used without any parameters this indicates that the wanted relation is 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.

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 of relations do and are consequently correctly updated by the DBMS before subsequent retrievals.

A view is time dependent if at least one of the relations and views it is derived from is time dependent. Otherwise it it 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. We define:

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

Backlog view time slices involving NOW are time dependent, and, as above, so are backlog views derived from views involving NOW.

2.3 Query Language of DM/T

By introducing the time slice operator that makes a temporal relation into a normal "flat" relation it has become possible to basically use the standard relational algebra as the query language. Figure 5 outlines the query language. Together with the types of arguments outlined in the previous subsection it describes the functionality supported by DM/T. The solid horizontal lines partition the operators into: *the time slice, the fundamental operators,* and *derived and extended operators.*

We adopt a set of precedence rules to simplify the appearance of query expressions. The precedence of all unary operators are the same, the precedence for all binary operators are the same,

and unary operators have the highest precedence. Parenthesis are used to control precedence in the standard way, and application takes place from left to right.

Operators of the Query Language		
Notation	*Name*	*Description*
$R(t_x)$	time slice	This operator already was introduced and discussed earlier in the paper. In the literature the notation $\tau_{t=t_x}\ R$ is common, but in the present setting the function application notation is the most convenient.
π	Projection	The standard projection operator. In the context of backlogs we will use π_{Tuple} to mean projection on all attributes of the associated user-defined or schema relation.
σ_F	Selection	The standard selection operator. The condition F can contain an arbitrary subquery.
$\times$	Cartesian product	The standard Cartesian product operator.
-	Difference	The standard difference operator.
$\cup$	Union	The standard union operator.
$\cap$	Intersection	The usual definition applies: $R \cap S \stackrel{\text{def}}{=} R-(R-S) = S-(S-R)$
$\bowtie_F$	(Theta) Join	The standard definition: $R \bowtie_F S \stackrel{\text{def}}{=} \sigma_F\, R \times S$. If no condition F is specified natural join is assumed.
$\rhd_F$	Semi join	$R \rhd_F S \stackrel{\text{def}}{=} \pi_{Att(R)}(R \bowtie_F S)$, where *Att(R)* is the attributes of R.
"aggregate functions"		We allow for a full range of aggregate functions: max, min, mean, count, avg, sum, product, unit.

Figure 5: Notation, names and descriptions of standard operators.

EXAMPLE: Let a sample database contain the relation *Emp* having attributes *Name*, *Height*, and *Salary* with the obvious meanings; the domains are irrelevant for the purpose of the example.

The tables following show four simple queries possible. For all queries, NOW = 4:00 pm, May 1. 1989.

B_{Emp}					
Id	*Op*	*Time*	*Name*	*Height*	*Salary*
"surrogate"	Mod	0420891238	Brown	178	42 000
"surrogate"	Ins	0408891034	Brown	178	32 000
"surrogate"	Ins	0402891456	Mark	177	90 000
"surrogate"	Mod	0420891245	Brown	178	32 000
"surrogate"	Ins	0331891131	Jensen	188	10 000
"surrogate"	Del	0415891209	Mark	177	90 000
"surrogate"	Ins	0419890902	Smith	170	30 000
"surrogate"	Mod	0501891555	Jensen	188	11 000

$\sigma_{NOW-30\ days \leq Time \leq NOW \wedge Op = Mod} B_{Emp}$					
Id	*Op*	*Time*	*Name*	*Height*	*Salary*
"surrogate"	Mod	0420891238	Brown	178	42 000
"surrogate"	Mod	0420891245	Brown	178	32 000
"surrogate"	Mod	0501891555	Jensen	188	11 000

Emp(NOW - 20 days)		
Name	*Height*	*Salary*
Mark	177	90 000
Brown	178	32 000
Jensen	188	10 000

Emp		
Name	*Height*	*Salary*
Smith	170	30 000
Brown	178	32 000
Jensen	188	11 000

□

2.4 Logical and Physical Aspects

Let us characterize the different kinds of relations of DM/T according to two dimensions. The first is related to the question of physical storage. Traditionally, base relations are stored while views are computed or virtual. The second is related to the question of logical derivability. Base relations are not derivable from other relations, while views are. See figure 6 where generally accepted characterizations of base relation and view are summarized [Dat86, Ull82, Cod79].

In DM/T the relation concepts have new meaning. Every base relation has a backlog, and all

Traditional relation concepts	
Concept	*Description*
base relation	Actual data are stored in the database. The relation physically exists, in the sense that there exists records in storage that directly represent the relation. A base relation cannot be derived from other relations. A base relation definition is part of the *schema.*
view	A view is characterized as a ***virtual***, ***derived*** or ***computed*** relation. It is not physically stored, but looks to the user retrieving information from the database as if it is. A view definition is part of a *subschema.*

Figure 6: The usual definitions of relation types.

data of base relations are stored in the backlogs in the form of change requests. This makes backlogs act as base relations and base relations act as views, derived from backlogs. The new meanings are described in figure 7. In the sequel, we will use the definitions presented there.

Redefined relation concepts	
Concept	*Description*
backlog	Backlogs are the relations that now function as base relations in the sense that they are stored and that all other relations (ultimately) are derived from these.
base relation	Base relations are not necessarily stored, and they are derived (directly) from backlogs.
view	The data of a view still is - directly or indirectly - constructible from base relations - or backlogs. However, even though a view still is derived, it is not necessarily virtual or computed. Views can be persistent.

Figure 7: Redefinitions of relation types.

Views can be stored in two ways. First, a view can be stored as a pointer structure, consisting of pointers to the relations (view or base) the view is derived from. Second, a view can be stored as actual materialized data [Rou89, JMR89].

We can summarize the concepts presented in this section as illustrated in figure 8.

We distinguish between *backlog views*, traditional *views* and *base relations*. The only difference

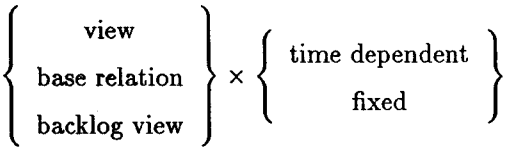


Figure 8: Different types of persistent views.

between views and base relations are that the former are derived indirectly from backlogs while the latter are derived directly. A view is valid only at a single point in time: The time-value specified when it was produced using the query language. Backlogs have an associated lifespan: From the time when the corresponding base relation was created till the current time - if they still exist - or otherwise till they were deleted. Backlog views inherit this notion of lifespan.

The second dimension in figure 8, *time dependence*, distinguishes between *fixed* and *time dependent* views. The valid time of fixed time slices of base relations and views and the lifespan of fixed backlog views never change and therefore they never get outdated. Because it is possible to use the special variable *NOW* in query expressions, both base relations, views and backlog views can, however, be time dependent. A time dependent base relation can be visualized as a view that slides along a backlog as time passes. Similarly a backlog view can be thought of as a filtering window where one or both ends (start time and end time) move along a backlog. Let E be an expression which maps into the domain *TTIME* and R a relation. For each time dependent time slice, $R(E(NOW))$, there is a differential file, $\delta R(E(NOW))$. This differential file is a sequence of change requests in the backlog of the relation, that are not yet reflected in the actual state of the time slice.

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, supporting the data structures and functionality of DM/T.

An implementation model is neither a data model nor an actual implementation, but something in between. A data model is for the user of the system and it describes structuring mechanisms and ways to manipulate structures. Typically, a data model is said to contain data structures, constraints specification mechanisms and a query language. An implementation model is irrelevant to the user and it describes how the functionality of the data model is supported in terms of lower level concepts. Yet it - unlike an implementation - still focuses on principles, ideas, and logical aspects.

In this section we present the three different stores of IM/T, i.e. (1) the store containing backlogs and indices, (2) the cache containing views, and (3) the Extended Logical Access Path (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. Through-out this paper we will assume that the blocks with tuples of a backlog only contain tuples of that backlog, and that, for each backlog, its tuples 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, however, be applied to backlogs. Note, that only an identifier is required for a deletion (insertion) request to serve its purpose during incremental (decremental) update, "property attributes" are not needed. On the other hand backlog queries (see subsection 2.2) can be specified in terms of property attributes and will thus benefit from the replication of these data instead of having to infer them through decompression.

The amount of data to be stored in backlogs can grow arbitrarily. To cope with the bulk of historical data we allow for partitioning this store. In this paper we will not present a specific design for this facility that allows for migrating seldomly used data to slow and cheap mass storage (e.g. write-once, read-many (WORM) optical disk). Instead we refer the interested reader to the literature: [DLW84, LDE+84, Ahn86, SA88, SL89, KS89, Chr87]. As mentioned in the introduction views cached as data can help eliminate references to data on slow storage, a feature that improves the usability of partitioning.

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. For example, it is possible to only keep the current state of selected base relations. We have devoted a separate, forthcoming paper to the presentation of a subsystem that allows for the specification of which historical data to maintain, where issues of query processing in the context of missing historical data are addressed, too.

3.2 The Pointer and Data Cache of IM/T

The cache of IM/T is simply a collection of query results stored as either pointers or data. Physically, a part of secondary memory is allocated for the cache. We denote by C the number of disk pages available 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

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

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.

Indexing of the results is available. The ELAP to be 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 (1) help estimate the cost of processing future differential files, and (2) help choose between different ways of processing a differential file. The design of a data structure that maintains the statistics and its use 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 as follows: We saw that the extensions of the data model DM/T were transparent to the naive user only expecting the relational model. IM/T now takes the transparency further by allowing efficient retrieval of current data and immediate and efficient checking of standard integrity constraints.

3.3 The Extended Logical Access Path of IM/T

The system has three stores: One for backlogs, one for views (the cache), and one for query expressions. This last one is the ELAP. It stores relevant information computed by the query evaluation subsystem about query expressions and possible materializations of query expressions.

The the ELAP is a directed acyclic graph (DAG). With each node there is an associated 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 to node N_b indicates that the operator constructs an expression associated with N_b from an expression associated with N_a.

EXAMPLE: The ELAP consisting only of the expression $R(t_1) \bowtie_F S(t_1)$ will be visualized the following way:

First there will be leaf nodes for the two backlogs B_R and B_S - backlogs are always leaf nodes, and conversely. These will be connected to nodes labeled $R(t_1)$ and $S(t_1)$, respectively. The connecting edges signify the time slice operator. Finally, the node representing $R(t_1) \bowtie_F S(t_1)$ is **connected** to both $R(t_1)$ and $S(t_1)$ with the join operator ($\bowtie_F$). See figure 9.

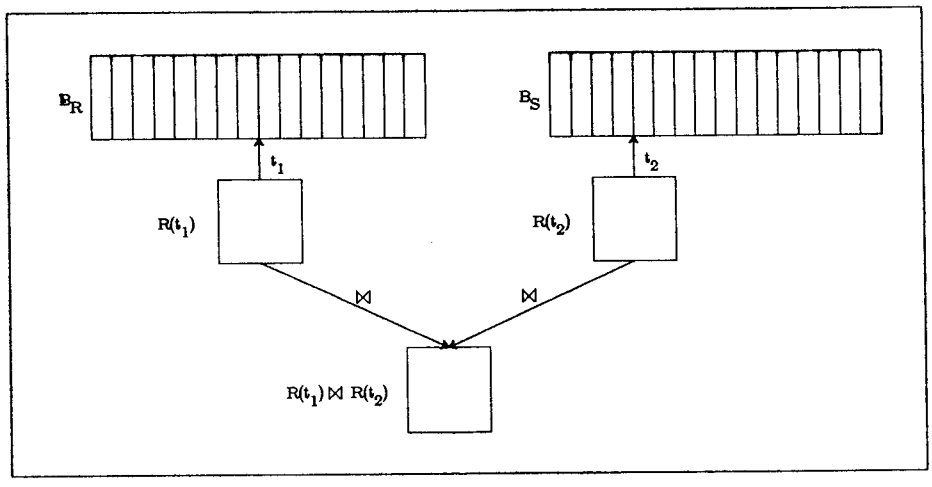


Figure 9: Sample ELAP.

□

The ELAP is not just a query graph, it integrates graphs of query expressions that have been computed or just have been subject to estimation of statistics into a unifying structure. Common (sub-)expressions among the set of query expressions are represented by the same node, or sub-DAG, of the ELAP. Consequently, there can exist several different sub-DAG's with backlog relations as leaves for the same node. The crucial observation is that while the expressions of a node all produce the same result the expressions might be far from equivalent when it comes to cost of processing. The ELAP can be thought of as a generalized AND/OR graph where at a single node there is a choice of one ("OR") of several sets of "AND" edges [MB85, Ric83], where "AND" edges correspond to binary operators.

EXAMPLE:

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 consisting of these expressions is shown in figure 10.

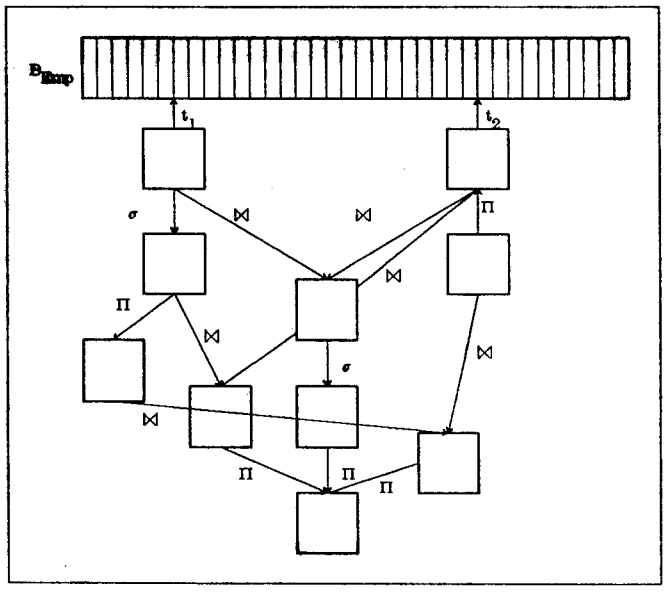


Figure 10: The view corresponding to a node can be computed from several query language expressions.

□

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 the following six categories depending on the computational status of the labeling query expressions:

case 1 the result of the query expressions is a cached pointer structure

case 2 the result of the query expressions is stored as actual data in the cache

case 3 the result of the query expressions previously was a cached pointer structure

case 4 the result of the query expressions previously was stored as actual data in the cache

case 5 no result for the query expressions has ever been stored, but results might have been computed or just estimated

case 6 the query expression denotes a backlog

The transition structure of these types of nodes is illustrated in figure 11.

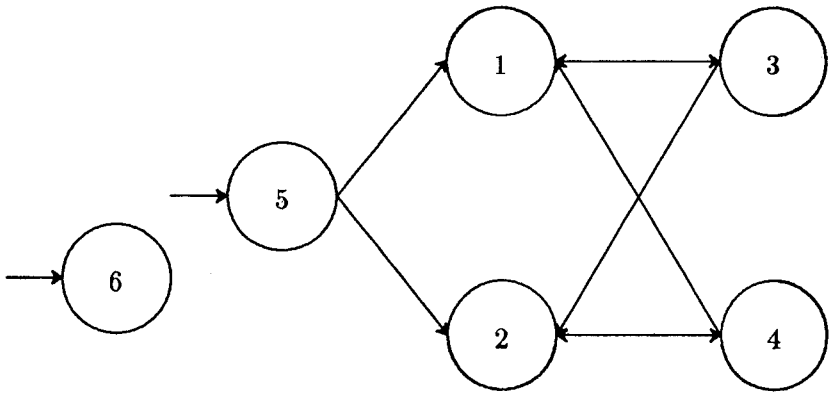


Figure 11: **Transition structure of the six categories of nodes in the ELAP. For example, a node for a previously cached** pointer structure (case 3) can change to a node for cached data **or for cached** data.

Edges of the ELAP are labeled by operators that correspond to those in section 2.3. In addition, we allow for combined operators: A projection and a selection can be done as part of another operator (including a projection/selection). Sample labels are: π_A, $\pi_A\sigma_F$, $\sigma_{F'}(\sigma_F \cdot \bowtie \cdot \pi_A)$. Combined operators are used in IM/T to avoid storage of intermediate results. Unary operators label a single edge, and binary operators label two edges.

Now, let us consider which types of statistics should be kept in the six types of nodes. Below is a listing of the different kinds of statistics; after the list is a cross tabulation of types of statistics with respect to node types.

fixed/time-dependent If the variable "*NOW*" occurs in an underlying time slice expression or selection on a time stamp attribute, the query expressions of the node are time dependent. Otherwise they are fixed.

cardinality This statistics has different interpretations for different node types. When an expression has actually been computed it is the cardinality of the most recently computed result. In **case 5** it is an estimate, and in **case 6** it is the exact number of tuples.

result type Tells whether the cached result of a node is stored as data or pointers.

tuple size For pointers there is a fixed size. The tuple size is the actual size of a data tuple. Implicit attributes are not considered. There is only one tuple size per node.

cached expression The immediate subexpression(s) (i.e. successor node(s)) used in the most recent computation of the nodes result is (are) registered.

up-to-date status For cached results it is indicated how up-to-date the result is, i.e. how faithfully its defining subexpression(s) reflect the current state of the backlogs. This is used to estimate the cost of bringing a stored result up-to-date. It also indicates the validity of statistics based on the stored result.

number of references The number of times a cached result of a node has been used. For **case 5** the value is zero. For backlogs this information can be made more specific as to aid in deciding which parts should remain on fast storage, which parts should be migrated to slow mass storage, and which dependent views should be cached as data.

first reference In order to be able to estimate the frequency of reference it is registered when a cached result of a node was first referenced.

computation cost Each expression of a node has this information. Both the cost of compute the actual data result and the cost of retrieving both data and pointers is included.

usage cost This is the cost of retrieving a result from the cache and bring it up-to-date. It can be computed from information about node type, tuple/pointer size, cardinality, and up-to-date status.

when deleted When the last deletion of a cached result took place.

why deleted Applies to **case 3** and **case 4**. This is useful when deciding if a deleted result should be rematerialized.

indices A list of which indices are available.

node type Which one of the six types of nodes.

	case 1	case 2	case 3	case 4	case 5	case 6
fixed/time-dependent	√	√	√	√	√	
cardinality	√	√	√	√	√	√
tuple size	√	√	√	√	√	√
cached expression	√	√	√	√		
up-to-date status	√	√	√	√		
number of references	√	√	√	√	√	√
first reference	√	√	√	√	√	√
computation cost	√	√	√	√	√	
usage cost	√	√	√	√		
when deleted			√	√		
why deleted			√	√		
indices	√	√				√
node type	√	√	√	√	√	√

The motivation for maintaining statistics as above is that the cost of maintaining them is less than the benefits achieved during query optimization from having them available at hand. Practical experiments are needed to decide for which ones this the case.

Integration of parse trees for query expressions into the ELAP, deletion of parts of the ELAP, and restructuring of it is beyond the scope of this presentation.

We assume that the ELAP can be kept in main memory and ignore the cost of using it during query optimization.

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 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. We formalize the general notion of STN and discuss plan selection using dynamic programming and the A^* algorithm [Ric83]. Then the state and transition spaces of IM/T are defined, an example is given, and the role of the ELAP is described. Finally, trade-offs in the design of IM/T are presented.

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. Thus, an edge is a triplet, $(x_1, p, x_2) \in (\mathcal{S}, P(\mathcal{S}), \mathcal{S})$, 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). The function *cost* is of the form $I/O + \omega CPU$, where I/O is the number of page transfers and CPU is the number of comparisons made by the CPU; the constant ω specifies the relative weight of I/O's and CPU's. 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 12 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:

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

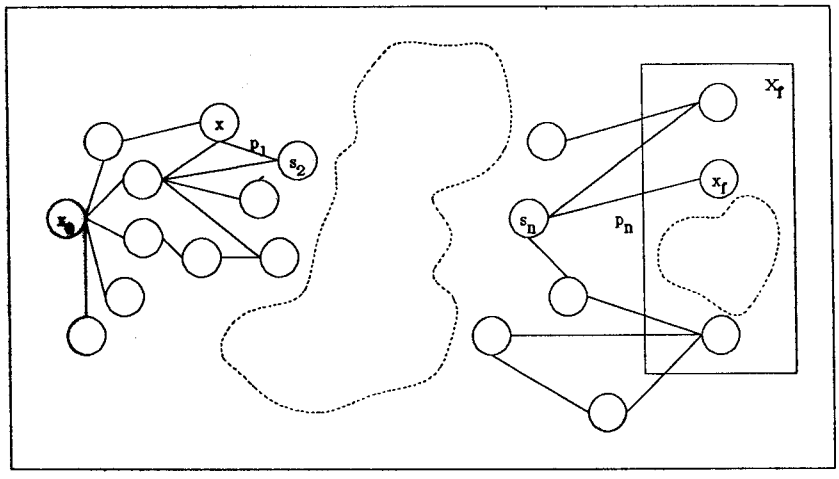


Figure 12: An outline of a STN.

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

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.*

4.2 Plan Selection

Above we have defined the general framework of State Transition Networks for query optimization. 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, (1) generation of the STN of the query to be computed, (2) estimation and selection of the optimal path in the STN. In practice, the whole STN need not be computed before phase (2) is initiated; parts needed during phase (2) 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].

Heuristic techniques are the obvious alternatives to dynamic programming. They have the advantage, that they do not require two phases. Instead they interleave phases by only generating paths as they are explored and afterwards discarding of them. Thus, they require much less storage space. The disadvantage is that they do not necessarily compute the optimal solution.

Let us consider the A^* algorithm [Ric83]. At each step in the process of plan generation, the algorithm will choose the most promising transition among all possible transitions not previously chosen. In doing so, it uses a heuristic function $\hat{f} = g + \hat{h}$ that estimates the true cost (f) of the plan being generated. The term g is the lowest cost found of getting from the initial to the current node, and $\hat{h}$ estimates the true cost (h) of getting from the current node to a final node. Observe that the next transition chosen at a node n only depends on $\hat{h}$. Therefore the applicability of A^* depends heavily on the quality of the estimate $\hat{h}$. In the ideal case where $\hat{h} = h$, no search is done at all; the algorithm immediately converges. The better $\hat{h}$ estimates h the closer it comes to the ideal case, and if it can be guaranteed that $\hat{h}$ never overestimates h, an optimal plan will eventually be generated. In the general case A^* does not return the optimal plan.

Due to the lack of easily computable and high quality candidates for $\hat{h}$ we have chosen a dynamic programming approach. 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], manipulated, and the result can be written to secondary memory as a new object of that state space.

The types of objects include:

1. **Intermediate results** A query result can be any kind of relation. 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 are temporary, but can later be

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

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 just estimated temporary results.

2. **Backlogs** 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.

3. **The ELAP** As mentioned it describes the structure of the cache and in addition stores statistics about queries in general. Each state of each STN contains this object.

4. **The cache** Cached results constitute together with backlogs the outsets for differential computation of queries, and it 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.

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.

1. The relational operators of figure 5 are included. For brevity we will only consider the most important ones, projection (π), selection (σ), and equi-join ($\bowtie$) in detail.

2. Incrementing or decrementing 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 are computed: $DIF(\sigma_F R, \delta_R)$, $DIF(\pi_A R, \delta_R)$, $DIF(R \bowtie S, R, \delta_R, S, \delta_S)$. 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 $R' = (R - \delta_R^-) \cup \delta_R^+$:

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

3. Constructing a differential file of a computed result obtainable, as above, by selection, projection, or join of intermediate results: $DELTA(\sigma_F, \delta_R)$, $DELTA(\pi_A, \delta_R)$, $DELTA(\bowtie, R, \delta_R, S, \delta_S)$. Thus, if δ_R exists, we can get $\delta_{\sigma_F R}$ of $\sigma_F R$ by computing $DELTA(\sigma_F, \delta_R)$. We have:

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

4. Combined operators. 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 and would therefore be more efficient. This is the motivation behind introducing combined operators. More specifically we conceptually 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 [SC75, Ull82, JKS84, JK84] 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. Following is a list of such transformations that are also useful when applying pruning rules to a STN:

commutative law for joins

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

associative law for joins

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

cascade of projections

$$\pi_{F_1}(\pi_{F_2} E) \equiv \pi_{F_1} E \text{ , where } F_1 \subseteq F_2$$

commuting and cascade of selections

$$\sigma_{F_1} \sigma_{F_2} E \equiv \sigma_{F_2} \sigma_{F_1} E \equiv \sigma_{F_1 \wedge F_2} E$$

commuting selections and projections

$$\pi_{F_1}\sigma_{F_2}E \equiv \pi_{F_1}\sigma_{F_2}\pi_{F_1'}E,$$

where $F_1 = A_1, A_2, A_3, \ldots, A_n$, $F_1' = A_1, A_2, A_3, \ldots, A_{n+m}$, and F_2 only involves *some* attributes among $A_i, i = 1, 2, 3, \ldots, n$ in addition to *all* attributes $A_i, i = n+1, n+2, n+3, \ldots, n+m$.

commuting selections and joins

$$\sigma_F(E_1 \bowtie E_2) \equiv \sigma_{F_1}(\sigma_{F_2}E_1 \bowtie \sigma_{F_3}E_2)\ ,$$

where F can be expressed as $F_1 \wedge F_2 \wedge F_3$, so that F_1 involves attributes in both E_1 and E_2, F_2 only involves attributes in E_1, and F_3 involves attributes in E_2.

commuting selections and unions

$$\sigma_F(E_1 \cup E_2) \equiv \sigma_F E_1 \cup \sigma_F E_2$$

commuting selections and differences

$$\sigma_F(E_1 - E_2) \equiv \sigma_F E_1 - \sigma_F E_2$$

commuting projections and unions

$$\pi_F(E_1 \cup E_2) \equiv \pi_F E_1 \cup \pi_F E_2$$

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 incremental join

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

4.4 An Example

Let us illustrate the use of STN's in IM/T with an example. Assume we have a relation, *Emp*, with two attributes:

$$Emp(Id : Int,\ Salary : Int),$$

where *Salary* is the annual pay in thousands of dollars. Assume that the cache contains these results, properly reflected by the ELAP:

Cached Results		
rid	Query representing the result	Data/Pointer (D/P)
(R-1)	$Emp(t_{init})$	-
(R-2)	$\pi_{Salary} Emp(t_\gamma)$	P
(R-3)	$Q_1 = \sigma_{Salary \geq 20} Emp(t_\alpha)$	D
(R-4)	$Q_2 = \sigma_{Salary \geq 40} Q_1$	D
(R-5)	$\sigma_{Salary \geq 55} Q_2$	P
(R-6)	$\sigma_{Salary \geq 25} Emp(t_\beta)$	P

The results (R-4) and (R-5) are defined in terms of the results (R-3) and (R-4), respectively and recursively. The pointers of (R-5) therefore point to (R-4). Let the query to be answered be given as $Q = \sigma_{Salary \geq 30} Emp(t_a)$. We have

$$Q \sqsubseteq \text{(R-1)}, \quad Q \sqsubseteq \text{(R-3)}, \quad Q \sqsubseteq \text{(R-6)}$$

This means that we can use (R-1), (R-3), and (R-6). The tables below show all possible plans after application of the pruning rules following:

rule a *Only combine a differential with its outset if exactly the selections (projections) performed on the outset have been performed on the differential, too.* For example, if we want $\sigma_{Salary \geq 30} Emp(t_a)$ and have $\sigma_{Salary \geq 20} Emp(t_\alpha)$ and $\delta_{\sigma_{Salary \geq 30} Emp(t_a)}$, then $\sigma_{Salary \geq 20} Emp(t_a)$ should not be updated with $\delta_{\sigma_{Salary \geq 30} Emp(t_a)}$ until after $\sigma_{Salary \geq 30}$ has been performed.

rule b *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 c *Only compute a differential of an outset, if the outset already exists.*

STN for $\sigma_{Salary \geq 30} Emp(t_a)$ - *Part I*		
argument state	transition	result state
0	$DIF(\sigma_{Salary \geq 30} Emp(t_{init}), (t_{init}, t_a))$	x_f
0	$DELTA(Emp(t_{init}), (t_{init}, t_a))$	1
0	$DELTA(\sigma_{Salary \geq 30}, Emp(t_{init}, t_a))$	2
1	$\sigma_{Salary \geq 30} \delta_{Emp(t_{init})}$	3
1	$DIF(\sigma_{Salary \geq 30} Emp(t_{init}), \delta_{Emp(t_{init})})$	x_f
1	$DIF(Emp(t_{init}), \delta_{Emp(t_{init})})$	4
2	$DIF(\sigma_{Salary \geq 30} Emp(t_{init}), \delta_{\sigma_{Salary \geq 30} Emp(t_{init})})$	x_f
3	$DIF(\sigma_{Salary \geq 30} Emp(t_{init}), \delta_{\sigma_{Salary \geq 30} Emp(t_{init})})$	x_f
4	$\sigma_{Salary \geq 30} Emp(t_a)$	x_f
0	$DELTA(\sigma_{Salary \geq 20}, Emp(t_\alpha, t_a))$	5
0	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 20} Emp(t_\alpha)$	11
0	$DIF(\sigma_{Salary \geq 20} Emp(t_\alpha), (t_\alpha, t_a))$	14
5	$\sigma_{Salary \geq 30} \delta_{\sigma_{Salary \geq 20} Emp(t_\alpha)}$	7
5	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 20} Emp(t_\alpha)$	12
5	$DIF(\sigma_{Salary \geq 20} Emp(t_\alpha), \delta_{\sigma_{Salary \geq 20} Emp(t_\alpha)})$	8
5	$DIF(\sigma_{Salary \geq 30} Emp(t_\alpha), \delta_{\sigma_{Salary \geq 20} Emp(t_\alpha)})$	x_f
11	$DELTA(\sigma_{Salary \geq 20}, Emp(t_\alpha, t_a))$	12
11	$DELTA(\sigma_{Salary \geq 30}, Emp(t_\alpha, t_a))$	19
11	$DIF(\sigma_{Salary \geq 30} Emp(t_\alpha), (t_\alpha, t_a))$	x_f
8	$\sigma_{Salary \geq 30}\ \sigma_{Salary \geq 20} Emp(t_a)$	x_f
12	$DIF(\sigma_{Salary \geq 30} Emp(t_\alpha), \delta_{\sigma_{Salary \geq 20} Emp(t_\alpha)})$	x_f
19	$DIF(\sigma_{Salary \geq 30} Emp(t_\alpha), \delta_{\sigma_{Salary \geq 30} Emp(t_\alpha)})$	x_f
7	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 20} Emp(t_\alpha)$	13
12	$DELTA(\sigma_{Salary \geq 30}, \delta_{\sigma_{Salary \geq 20} Emp(t_\alpha)})$	13
13	$DIF(\sigma_{Salary \geq 30} Emp(t_\alpha), \delta_{\sigma_{Salary \geq 30} Emp(t_\alpha)})$	x_f
14	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 20} Emp(t_a)$	x_f

STN for $\sigma_{Salary \geq 30} Emp(t_a)$ - *Part II*		
argument state	transition	result state
0	$DELTA(\sigma_{Salary \geq 25}, Emp(t_\beta, t_a))$	6
0	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 25} Emp(t_\beta)$	15
0	$DIF(\sigma_{Salary \geq 25} Emp(t_\beta), (t_\beta, t_a))$	18
6	$\sigma_{Salary \geq 30} \delta_{\sigma_{Salary \geq 25} Emp(t_\beta)}$	9
6	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 25} Emp(t_\beta)$	16
6	$DIF(\sigma_{Salary \geq 25} Emp(t_\beta), \delta_{\sigma_{Salary \geq 25} Emp(t_\beta)})$	10
6	$DIF(\sigma_{Salary \geq 30} Emp(t_\beta), \delta_{\sigma_{Salary \geq 25} Emp(t_\beta)})$	x_f
15	$DELTA(\sigma_{Salary \geq 25}, Emp(t_\beta, t_a))$	16
15	$DELTA(\sigma_{Salary \geq 30}, Emp(t_\beta, t_a))$	20
15	$DIF(\sigma_{Salary \geq 30} Emp(t_\beta), (t_\beta, t_a))$	x_f
10	$\sigma_{Salary \geq 30}\ \sigma_{Salary \geq 25} Emp(t_a)$	x_f
16	$DIF(\sigma_{Salary \geq 30} Emp(t_\beta), \delta_{\sigma_{Salary \geq 25} Emp(t_\beta)})$	x_f
20	$DIF(\sigma_{Salary \geq 30} Emp(t_\beta), \delta_{\sigma_{Salary \geq 30} Emp(t_\beta)})$	x_f
9	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 25} Emp(t_\beta)$	17
16	$DELTA(\sigma_{Salary \geq 30}, \delta_{\sigma_{Salary \geq 25} Emp(t_\beta)})$	17
17	$DIF(\sigma_{Salary \geq 30} Emp(t_\beta), \delta_{\sigma_{Salary \geq 30} Emp(t_\beta)})$	x_f
18	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 25} Emp(t_a)$	x_f

Note that the final state x_f actually is a set of states $f_1, f_2, \ldots, f_k \in \mathcal{X}_f$ that each contain the query to be computed and in addition different sets of intermediate queries. A sequence of change requests with time stamps between t_a and t_b of a backlog are denoted (t_a, t_b), or $B_{Emp}(t_a, t_b)$ to avoid ambiguity. Also, we have only included reasonable transitions in the STN. Transitions that do not help us get closer to the goal or only marginally does so have been left out. For example, the selection $\sigma_{Salary \geq 27} \sigma_{Salary \geq 25} Emp(t_a)$ is not considered relevant because the selection $\sigma_{Salary \geq 30} \sigma_{Salary \geq 25} Emp(t_a)$ can be done instead.

Due to the simplicity of the query to be processed there are not many applications of combined operators. However, the transitions from states 5 and 6 to the set of final states x_f illustrate simultaneous selection on a differential file, selection on an outset and the incremental/decremental update of the (selected) outset with the (selected) differential file.

4.5 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. Let us now in more detail consider its role.

We saw in the example of subsection 4.4 that it is often the case that cached results can be used in many ways to compute a query.

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.

The ELAP is an efficient means of identifying all covering cached results for a given query.

Orthogonally to the issue of coverage there is the issue of *temporal match*, 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 need 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 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).

In summary, we have described how the notion of coverage supported by the ELAP is used to identify cache entries useful in the computation of a query. We have the greatest interest in minimally covering entries. Orthogonally, we are interested in the covering queries that are the temporally closest to the desired queries. The overall best query might be neither the temporally closest query nor the minimally covering query. It is the query from which the desired data can be computed (by both differential update and additional processing) with the lowest cost.

4.6 Trade-offs

In this subsection we have presented a general framework for query optimization. When we designed the framework we had to consider several trade-offs and make reasonable choices. Let us look back and discuss some of the issues.

The over-all goal is to process queries as efficiently as possible. Since processing involves both query plan generation and query computation it is a poor strategy to identify a very efficient query plan if the process of doing so is very expensive. On the other hand it is equally foolish not to consider alternative ways to carry out a query. Thus we have a trade-off between the cost of the selected query plan and the cost of selecting it. A lot of factors influence this trade-off. In our framework, the granularity of atomic operators that cause a state transition is such a factor. With a fine granularity the size and complexity of the STN's increase with more expensive calculations but also potentially more efficient plans as a result. A coarse granularity decreases the STN's, results in less computation and considers fewer alternative plans. Consequently we are likely to get lower quality query plans. Another factor is how faithfully cost functions reflect the actual computations

they estimate - analogically we have a trade-off between the quality of cost estimates and the cost of getting them. Without good cost functions it is not feasible to consider query plans that only differ slightly. E.g. if we only include I/O when computing costs it does not make sense to generate query plans that only differ in CPU cost. It is not only the granularity of operators that affect the trade-off. Selecting carefully which operators to apply at a given state can reduce the size of a STN without seriously affecting the quality of the generated plans. This is the motivation behind the pruning rules of the next section.

In the present version a state transition network models queries on a logical level. It is a topic of future research to investigate the feasibility to extend this to a physical level, where nodes include information on how/whether relations (files) are sorted, indexed etc., and transitions include `sort`, `merge-join`, `nested-loop-join`, ...

A major disadvantage of logical level operators is that, in order to achieve good performance, they should be carried out in an interleaved fashion on a lower level[4]. Thus logical level query plans do not represent faithfully the plans that are carried out and a the selected logical level plan might not be competitive when translated to a physical level. This suggests physical level operators of our STN's. But physical level operators as transitions result in very large STN's. We have chosen to include logical level combined operators to gain the advantages of logical level operators without at the same time inheriting the disadvantages.

5 Implementation of Operators of STN's

In this section we discuss the operators of STN's 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 STN's of IM/T, a large number of cases must be considered. Disregarding for the moment combined operators, the cases are outlined in figure 13.

The figure 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

[4] See [Ull82], where operations to be done in an interleaved fashion are detected after plan selection.

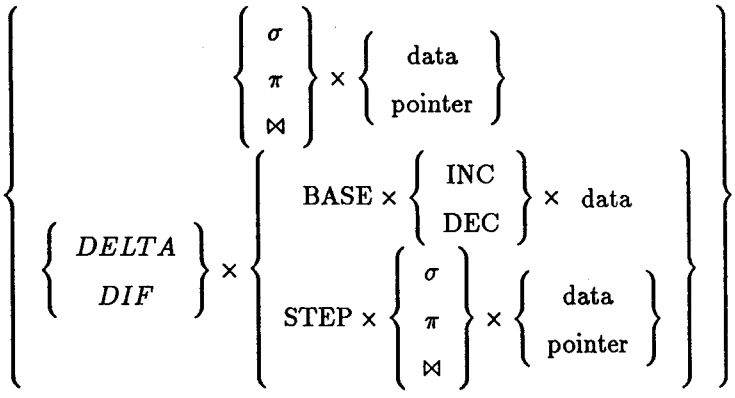


Figure 13: Implementation of operators of STN's.

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 $\bowtie$[5]. 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, we will mention combined operators. Selections and projections can be done on the fly, and therefore a projection can be part of a selection, a selection can be part of a projection, and

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

selections and projections can be parts of a join. This is to be interpreted recursively. For example, if we have a projection operation then a selection can be done on the fly, and a projection can be done as a part of the selection, again on the fly. Combined operators are useful because they avoid writes and subsequent reads of intermediate results.

5.2 General Observations, Assumptions and Notation

One of the strengths of the implementation model is that it generally does not make assumptions about physical details such as particular indexing techniques (B-tree/hash-index/etc.) and buffer management strategies. The orthogonality of such issues allow us to disregard them. Consequently we are also forced to disregard development of precise cost formulas for the operators of STN's since this relies on such assumptions.

Below we make some general observations and assumptions relevant for our discussion of implementations of the operators of STN's. Let the following self-explanatory parameters be given:

Cost Parameters	
notation	explanation
p	page size (space usable for tuples) [bytes]
$\#_R$	cardinality of R
t_size_R	tuple size of relation R [bytes]
p_size	size of a pointer [bytes]

Prior to the application of any operator the argument is read from secondary memory and, upon completion, the result is written back to secondary memory. The cost of read and write af a relation (stored as data) is[6]:

$$\left\lceil \frac{t\_size_R \ \#_R}{p} \right\rceil$$

The cost of retrieval of an unary and a binary pointer relation is:

$$\left\lceil \frac{p\_size \ \#_R}{p} \right\rceil + C^u_{mat} \text{ and }, \left\lceil \frac{2 \ p\_size \ \#_R}{p} \right\rceil + C^b_{mat},$$

where C^u_{mat} and C^b_{mat} are the costs of materializing the pointers. In the case where they point directly to B_R we have:

$$\left\lceil \frac{\#_R \ t\_size_{B_R}}{p} \right\rceil \leq C^u_{mat} \leq \#_R, \text{ and } \left\lceil \frac{2 \ \#_R \ t\_size_{B_R}}{p} \right\rceil \leq C^b_{mat} \leq 2 \ \#_R$$

[6] In order to simplify notation we have assumed that stored relations start on a new page.

The lower boundaries assume that the tuples of R are clustered in B_R. For the binary case we assume that each pointer occurrence is actually materialized. An intelligent approach would be to only materialize each distinct pointer. The upper boundaries assume that only one pointer is materialized for each page read. If the cost of materializing a pointer relation is unacceptable there are two possibilities in IM/T. First, the relation can be deleted from the cache, and second, the relation can be stored as data.

The costs of writing binary and unary pointer views are given as

$$\left\lceil \frac{p\_size \ \#_R}{p} \right\rceil, \text{ and } \left\lceil \frac{2 \ p\_size \ \#_R}{p} \right\rceil$$

Stored relations are clustered on the values of their key attribute in the case of data, and on `tid`'s in the case of pointers. Change requests are clustered on time stamp values.

The only difference between incremental and decremental computation is that the role of deletions and insertions is reversed. Therefore, when we only talk about incremental computation in the sequel, it is without loss of generality.

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

5.3 Selection, Projection, and Join

The traditional relational algebra operators, selection, projection, and join can be applied to any relation, including differential files (δ_R) and their constituent relations (δ_R^+, δ_R^-).

The expression F of the selection operator, $\sigma_F R$, can contain a conjunction of selection criteria:

$$\begin{aligned} F &\leftarrow term \mid term \wedge F \\ term &\leftarrow \texttt{Att\_Name op Att\_Name} \mid \texttt{Att\_Name op Value} \\ \texttt{op} &\leftarrow = \mid < \mid > \mid \geq \mid \leq \mid \neq \mid \not< \mid \not> \mid \not\geq \mid \not\leq \end{aligned}$$

`Att_Name` is an attribute identifier of the relation R.

The most advantageous implementation of selection depends on numerous factors, and has been addressed in many settings. Here we just outline some possible approaches. It can be implemented by sequential scan, by binary search, and using an (clustering) index (e.g. a hash-function, a B-tree, etc.). While sequential scan is always possible it might not prove the most advantageous. Binary search is possible if the tuples of the relation at hand are sorted on the values of the attribute in the selection criteria. Using an index obviously requires the index to exist already or to be build for the purpose. A clustering index is generally superior to a non clustering one. In the cases of selections specified on more than one attribute, multi-attribute techniques (access structures for spatial data)

can be beneficial, or a multi-attribute selection can be carried out as a sequence of single attribute selections.

The projection expression, A, of $\pi_A R$ is any subset of attributes of R. When we mix projection and difference, as we do in differential computations, a problem can occur since projection does not distribute over difference:

EXAMPLE: Assume that we want to compute $\pi_{Salary} Emp(t_a)$ and that we have $Emp(t_b)$ in the cache. A possible way of computing the result would be to compute the differential $\delta_{Emp(t_b)} = (\delta^+_{Emp(t_b)}, \delta^-_{Emp(t_b)})$ from the change requests $Emp(t_b, t_a)$, then do the projection on the outset and on the differential, and finally increment the (projected) outset with the (projected) differential. Expressed in relational algebra this corresponds to

$$\begin{aligned}
\pi_{Salary} Emp(t_a) &= \pi_{Salary}((Emp(t_b) - \delta^-_{Emp(t_b)}) \cup \delta^+_{Emp(t_b)}) \\
&\neq (\pi_{Salary} Emp(t_b) - \pi_{Salary} \delta^-_{Emp(t_b)}) \cup \pi_{Salary} \delta^+_{Emp(t_b)} \\
&= (\pi_{Salary} Emp(t_b) - \delta^-_{\pi_{Salary} Emp(t_b)}) \cup \delta^+_{\pi_{Salary} Emp(t_b)}
\end{aligned}$$

The reason for this is that the projection has made unique identification of tuples impossible. By always retaining the primary key of a relation and remembering whether it was removed by a projection or not we are able to carry out correctly the computation above. □

The equi-join operator $R \bowtie_F S$ can be used on any two relations. The condition F is a list of elements:

$$F \leftarrow \texttt{Att\_Name\_1} = \texttt{Att\_Name\_2} \mid \texttt{Att\_Name\_1} = \texttt{Att\_Name\_2}, F \ ,$$

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].

We allow for combined application of projection, selection, and join. Selection and projection can be combined with any operator (possibly combined) to form a combined operator. This is done to store only as few intermediate results as possible during a computation. For example, the expression $\pi_A \sigma_F \pi_B R(t_x)$ can be computed from $R(t_x)$ without writing and reading intermediate results to and from secondary storage.

5.4 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 29, 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[7]. 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 $\delta_{R(t_a)}^+$[8]. 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 no action was taken. This was so, because there in this case is an effect. Tuples of base relations have implicit time stamp attributes that are hidden, but can be seen by an explicit projection. The effect is that the values of this attribute in R are updated. We have so far ignored this implicit attribute 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.

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

[8] 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.

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:

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

Remember, that key information is retained to overcome the problem of indistinguishable tuples when **distributing** a projection over a difference. Figure 14 is a schematical representation of selection and projection for $DELTA$.

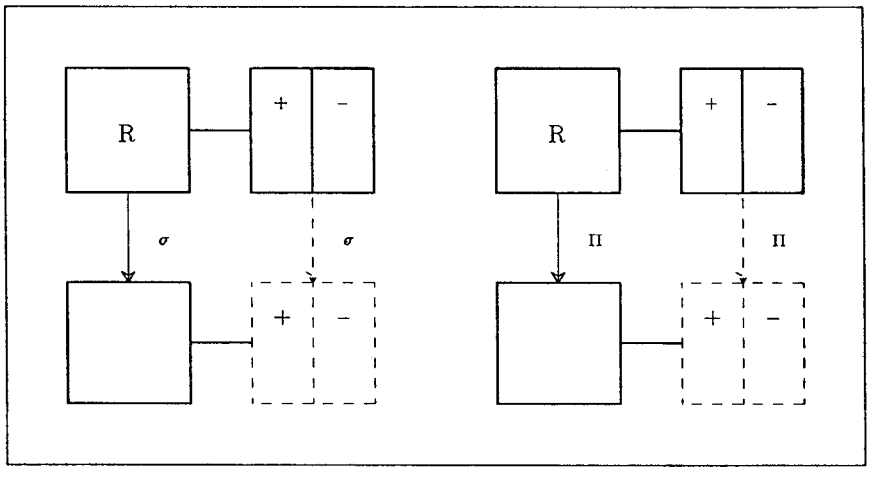


Figure 14: Schematic representation of $DELTA$ for projection and selection.

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:

$$(R \cup S) \bowtie T = (R \bowtie T) \cup (S \bowtie T)$$

$$(R - S) \bowtie T = (R \bowtie T) - (S \bowtie T)$$

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_1x_2, where $x_1 \in (R - S)$ and $x_2 \in T$. Further, $x_1 \in R$ and $x_1 \notin S$. Hence $x_1x_2 \in R \bowtie T$ and $x_1x_2 \notin S \bowtie T$.

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

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

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

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

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

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 15 shows all the constituent joins of $\delta^-_{R \bowtie S}$ and $\delta^+_{R \bowtie S}$ by means of **dotted lines** connecting two relations.

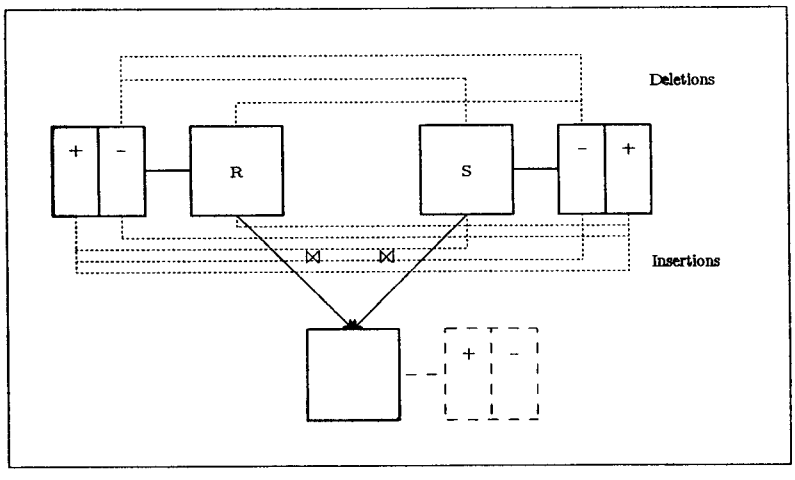


Figure 15: Computation of differentials of joins.

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).

Lastly we will address "combined operators". 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.

5.5 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 16. Both incremental and decremental computation are always possible (with $t_y = t_{init}$ and $t_y = NOW$, respectively).

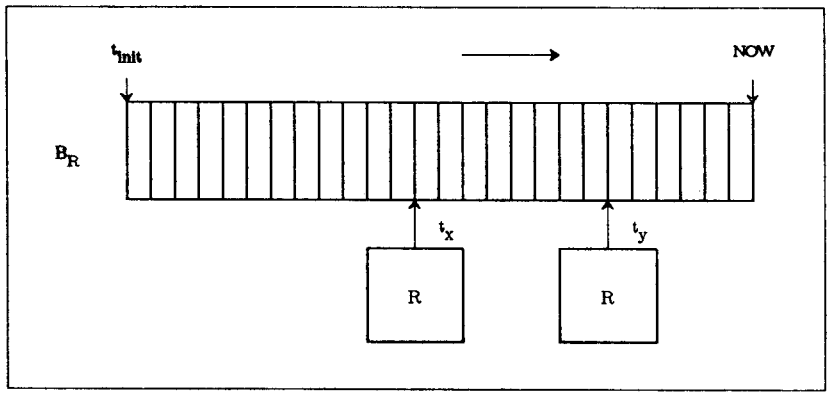


Figure 16: Time slicing a base relation.

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 discarded 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, 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:

$\#_I$	number of insertions
$\#_{DI}$	total number of change requests
$\#_{DoI}$	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.

If we assume that change requests of a backlog are distributed uniformly over time with k_1 requests per time unit, the cost of reading (t_x, t_y) is:

$$\left\lceil \frac{\mid t_y - t_x \mid \; k_1 \; t\_size_{B_R}}{p} \right\rceil$$

The Step Cases

There are three cases for data arguments (figure 17).

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 28).

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:

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

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

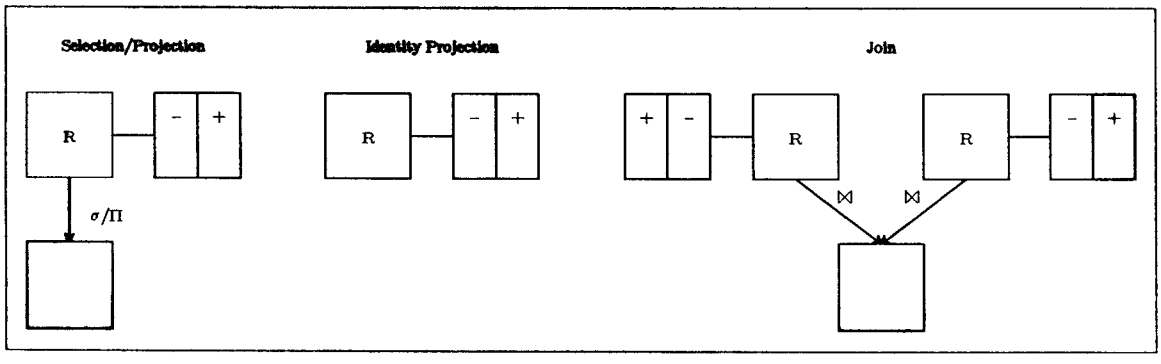


Figure 17: Differential computation: Selection, Projection, and join. For each case the arguments and their relations are shown. Note that the (non-bold) "R" of "Selection/Projection" is not an argument.

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 $(R \bowtie S)'$:

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

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[9].

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

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

motivation is to reduce the sizes of the STN's generated without leaving out promising query plans. Reduced STN's 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 transition at a state.

6.1 Pruning Rules

We present a list of rules that are representative for the type of rules that can be integrated into IM/T. Some have general applicability outside IM/T, and some are specific to IM/T. Also, some make use of the identity transformations of subsection 4.3.

Rule 1 *Reduce arguments by applying selections as early as possible.* This is an example of the general heuristic of trying to minimize the arguments of joins. Instead of doing joins of large relations, selections are done on large relations, if possible.

Rule 2 *Reduce arguments by applying projections as early as possible.* This is similar to the above rule.

Rule 3 *Collect projections and selections from sequences of projections and selections and apply them together.* This rule states that combined application of selections and combined application of projections are preferable to application of "uncollected" projection and selection. Together with the two following rules this one states that combined operators generally are preferable to non-combined operators.

Rule 4 *Carry out selections and projections on arguments to a join together with the join in the case that the unary operators are not followed by other joins*, i.e. combine selection/projection and join.

Rule 5 *Carry out selections and projections on arguments created by joins together with the joins*, i.e. combine join and selection/projection. The next three rules were introduced in subsection 4.4.

Rule 6 *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 7 *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 8 *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 9 *Application of maximal combined operators is preferable to the sequential application of the constituent operators of the combined operators.* This rule adds to rules 3,4, and 5 by including the combined versions of DIF and $DELTA$.

rule 10 *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 11 *Only use the least outdated cached result out of covering results of equal size.*

6.2 Sample Rule Application

When we apply the rules to the example of section 4.4, we get a very simple STN as shown in this table:

STN for $\sigma_{Salary \geq 30 \wedge Rating \in \{G,E\}} Emp(t_a)$ - *Pruning Rules Applied*		
argument state	transition	result state
0	$DIF(\sigma_{Salary \geq 30} Emp(t_{init}), (t_{init}, t_a))$	x_f
0	$DELTA(\sigma_{Salary \geq 20}, Emp(t_\alpha, t_a))$	5
0	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 20} Emp(t_\alpha)$	11
0	$DIF(\sigma_{Salary \geq 20} Emp(t_\alpha), (t_\alpha, t_a))$	14
5	$DIF(\sigma_{Salary \geq 30} Emp(t_\alpha), \delta_{\sigma_{Salary \geq 20} Emp(t_\alpha)})$	x_f
11	$DIF(\sigma_{Salary \geq 30} Emp(t_\alpha), (t_\alpha, t_a))$	x_f
14	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 20} Emp(t_a)$	x_f
0	$DELTA(\sigma_{Salary \geq 25}, Emp(t_\beta, t_a))$	6
0	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 25} Emp(t_\beta)$	15
0	$DIF(\sigma_{Salary \geq 25} Emp(t_\beta), (t_\beta, t_a))$	18
6	$DIF(\sigma_{Salary \geq 30} Emp(t_\beta), \delta_{\sigma_{Salary \geq 25} Emp(t_\beta)})$	x_f
15	$DIF(\sigma_{Salary \geq 30} Emp(t_\beta), (t_\beta, t_a))$	x_f
18	$\sigma_{Salary \geq 30} \sigma_{Salary \geq 25} Emp(t_a)$	x_f

7 Conclusion and Future Research

We introduced an implementation model, IM/T, for the standard relational model extended with transaction time; the new data model, DM/T, stored more detailed base data than did previously proposed transaction time extensions and its extended functionality was transparent to a naive user only expecting the relational model. We pointed out the new and desirable extensions of functionality following from supporting transaction time. Among those were: Analysis of change history, support for corporate decision making based on previous experience, support for accounting applications, etc.

The price to be paid was the introduction of very large and ever growing amounts of historical data. When new data were entered into the database the most recently entered data were retained as historical data, and nothing were deleted. This kind of relations differed from equally large relations of snapshot databases by having only a comparably small current state, i.e. instead of having one large state, a relation of a transaction time database had a sequence of smaller states.

Because of the new anatomy, traditional storage and query processing strategies has fallen short. Instead new strategies have proven to be more advantageous, and the goal has been to provide an efficiency comparable to that of a snapshot database with relations of the same size as single states of relations of a transaction time database. Achievement of this would allow for efficient access to any state of any relation, and it would extend the transparent of the data model extension to the level of efficiency.

The main focus was efficient query processing, and we showed how a host of storage and processing strategies could be integrated into one single, powerful, and very general framework.

Base data were stored as *change requests*, and were subject to *partitioned storage*. In addition, the query language could be used to specify base data not to be retained as a function of their age; this was the task of a *vacuuming* subsystem.

Views could be cached as both *pointers* and *data*, and they were indexed by a structuring index also containing statistics about the cached results.

State transition networks and dynamic programming were used for query plan enumeration and selection, and IM/T integrated differential (incremental/decremental) computation and recomputation of queries; in addition, combined operators were parts of the framework. Rules for pruning STN's were used for speeding up plan selection.

The generality of the model fell along the dimensions, largely independent, below:

- *Arbitrary* compression techniques could be used for for historical data.

- *Independently*, for each relation of base data the historical data to be retained could be specified using the query language.
- Views could be cached *selectively*.
- *Either* data *or* pointer storage of views could be chosen.
- Cached views could be *selectively* updated according to protocols ranging from eager to lazy propagation of update.
- Queries could be computed with *arbitrary* combinations of elements of recomputation and differential computation.
- *Different* sets of pruning rules can be adopted.
- *Any* graph search strategy can be used for plan selection.

In this paper we had to limit the presentation to the general framework. Many interesting extensions and additions are possible, and we very much encourage researchers to join us in our efforts of efficient support of transaction time. Topics of current and future research include:

- Optimal algorithms for computing the operators of STN's.
- A more detailed study of the ELAP. The following issues are considered: (1) node contents (statistics), (2) restructuring, (3) costs of usage.
- Differential files are not cacheable in the current design, but are recreated from sequences of change requests. A study of the reusability of differential files including algorithms for using a differential file as an outset for the differential computation of a requested file and a study of the cost effectiveness of such algorithms is an interesting topic.
- The design of a data structure for maintenance of statistics about previously computed differential files. The objective is to, given intensional properties of the differential file to be computed, be able to efficiently achieve extensional properties of the differential file of relevance for its computation (cost).
- The detailed integration of a partitioned storage strategy utilizing WORM technology into the framework. This includes the study of cached data views as a means of eliminating references to slow storage.

- The design of a vacuuming subsystem. Such a system will be activated regularly to remove temporal data not relevant to the applications supported by the DBMS. Data model extensions that allow for the specification of how much of the history of individual relations is required must be investigated.
- Extensions of DM/T and IM/T to support complex objects, versions, and configurations.
- Extensions of DM/T and IM/T to support logical time.
- Practical experiments with the designed query evaluation subsystem.
- Caching policies. The task of a caching subsystem is to decide which computed results to cache. The cache only has bounded space. When there is space available the problem is to decide whether a result considered for caching can be used advantageously later on. When no space is free the problem becomes one of ranking results already cached or considered for caching.

Acknowledgements

The work documented in this paper took place in the Database Systems Group at the University of Maryland. We wish to thank our fellow researchers for providing an inspiring environment.

References

[Adi81] Michel Adiba. Derived relations: A unifyed mechanism for views, snapshots and distributed data. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 293–305, 1981.

[AHH89] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[Ahn86] Ilsoo Ahn. Performance modeling and access methods for temporal database management systems. Ph.D. Dissertation TR86-018, Department of Computer Science. The University of North Carolina, Chapel Hill, NC 27599-3175, August 1986.

[AL80] Michel E. Adiba and Bruce G. Lindsay. Database snapshots. In *Proceedings of the Sixth International Conference on Very Large Databases*, pages 86–91, 1980.

[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 Transactions on Software Engineering*, 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. Technical Report CS-86-17, University of Waterloo. Computer Science Department, May 1986.

[BCL89] Jose A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[Chr87] Stavros Christodoulakis. Analysis of retrieval performance for records and objects using optical disk technology. *ACM Transactions on Database Systems*, 12(2):137–169, June 1987.

[CM86] U.S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 384–391, August 1986.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[Dat86] C. J. Date. *An Introduction to Database Systems.* The Systems Programming Series. Addison Wesley Publishing Company, fourth edition, 1986.

[DKH] Pam Drew, Roger King, and Scott E. Hudson. Performance of the self-adaptive mechanisms for maintaining derived data in cactis. Technical report, Department of Computer Science, University of Colorado and University of Arozona, Boulder, Colorado 80309 and Tucson, Arizona 85721.

[DLW84] P. Dadam, V. Lum, and H. D. Werner. Integration of time versions into a relational database system. In *Proceedings of the Tenth International Conference on Very Large Databases*, pages 509–522, August 1984.

[EB84] Klaus Elhardt and Rudolf Bayer. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems*, 9(4):503–525, December 1984.

[GS89] Himawan Gunadhi and Arie Segev. A framework for query optimization in temporal databases. Technical report LBL-26417, School of Business Administration and Computer Science Research Department, Lawrence Berkeley Laboratory, University of California, Berkeley, California 94720, 1989.

[GSS89] Himawan Gunadhi, Arie Segev, and J. George Shantikumar. Selectivity estimation in temporal databases. Technical report LBL-27435, School of Business Administration, University of California at Berkeley and Information and Computing Sciences Division, Lawrence Berkeley Laboratory, 1 Cyclotron Road, Berkeley, California 94720, 1989.

[Han87] Eric N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of ACM SIGMOD '87*, pages 440–453, 1987.

[HW89] Wei Hong and Eugene Wong. Multiple query optimization through state transition and decomposition. Memorandum UCB/ERL M89/25, Electronics Research Laboratory, College of Engineering, University of California at Berkeley, Berkeley, California 94720, March 1989.

[Jhi88] Anant Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 88–99, 1988.

[JK84] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *Computing Surveys*, 16(2):111–152, June 1984.

[JKS84] Matthias Jarke, Jurgen Koch, and Joachim W. Schmidt. Introduction to query processing. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, chapter 1, pages 3–28. Springer Verlag, 1984.

[JM89] Christian S. Jensen and Leo Mark. Queries on change in an extended relational model. Technical report CS-TR-2299, UMIACS-TR-89-80, Department of Computer Science. University of Maryland, College Park, MD 20742, August 1989. Submitted for publication.

[JMR89] Christian S. Jensen, Leo Mark, and Nick Roussopoulos. Incremental implementation model for relational databases with transaction time. Technical report CS-TR-2275,

UMIACS-TR-8963, Department of Computer Science. University of Maryland, College Park, MD 20742, June 1989. Submitted for publication.

[KD79] Kathryn C. Kinsley and James R. Driscoll. Dynamic derived relations within the raquel ii dbms. In *Proceedings of the 1979 ACM Annual Conference*, pages 69–80, October 1979.

[KD84] Kathryn C. Kinsley and James R. Driscoll. A generalized method for maintaining views. In *Proceedings of the National Computer Conference*, pages 587–593, 1984.

[Kim84] Won Kim. Global optimization of relational queries: A first step. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, pages 206–216. Springer Verlag, 1984.

[KS89] Curtis Kolovson and Michael Stonebraker. Indexing techniques for historical databases. In *Fifth International Conference on Data Engineering*, pages 127–137, February 1989.

[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. In *Proceedings of the ACM SIGMOD '84*, pages 115–130, June 1984.

[LW86] Stephane Lafortune and Eugene Wong. A state transition model for distributed query processing. *ACM Transactions on Database Systems*, 11(3):294–322, September 1986.

[LY85] Per-Åke Larson and H. Z. Yang. Computing queries from derived relations. In *Proceedings of the 11th Conference on Very Large Data Bases*, pages 259–269, 1985.

[Mat84] Jarke Matthias. Common subexpression isolation in multiple query optimization. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, pages 191–205. Springer Verlag, 1984.

[MB85] A. Mahanti and A. Bagchi. And/or graph heuristic search methods. *Journal of the ACM*, 32(1):28–51, January 1985.

[McK88] Leslie Edwin McKenszie. An algebraic language for query and update of temporal databases. TR 88-050, The University of North Carolina at Chapel Hill, Department of Computer Science, CB 3175, Sitterson Hall, Chapel Hill, NC 27599-3175, October 1988. Ph.D. Dissertation.

[MS89] Edwin McKenzie and Richard Snodgrass. An evaluation of algebras incorporating time. Technical report TR-89-22, Department of Computer Science. University of Arizona, Tucson, Arizona 85721, September 1989.

[RD89] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. Technical report 981, IBM Research Division. T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, 1989.

[Ric83] Elaine Rich. *Artificial Intelligence.* McGraw-Hill Series in Artificial Intelligence. McGraw-Hill Book Company, international student edition, 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, Inc., Englewood Cliffs, New Jersey 07632.

[Rou82] Nick Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

[Rou87] Nick Roussopoulos. Overview of adms: A high performance database management system. In *Proceedings of the 1987 Fall Joint Computer Conference*, pages 452–460, October 1987.

[Rou89] Nick Roussopoulos. The incremental access method of view cache: Concept, algorithms, and cost analysis. Technical report UMIACS-TR-89-15, CS-TR-2193, Department of Computer Science, University of Maryland, College Park, MD 20742, February 1989.

[RS87a] Doron Rotem and Arie Segev. Physical organization of temporal data. In *Third International Conference on Data Engineering*, pages 547–553, February 1987.

[RS87b] Lawrence A. Rowe and Michael R. (eds.) Stonebraker. The postgres papers. Memorandum UCB/ERL M86/85, University of California, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, June 1987.

[SA85] Richard Snodgrass and Ilsoo Ahn. A taxonomy of time in databases. In *Proceedings of the ACM SIGMOD '85*, pages 236–246, 1985.

[SA88] Richard Snodgrass and Ilsoo Ahn. Partitioned storage for temporal databases. *Information Systems*, 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. In *Proceedings of the ACM SIGMOD '79*, pages 82–93, 1979.

[SC75] John Miles Smith and Philip Yen-Tang Chang. Optimizing the performance of a relational algebra interface. *Communications of the ACM*, 18(10):569–579, October 1975.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, second edition, 1988.

[Sel86] Timos Sellis. Global query optimization. In *Proceedings of SIGMOD '86*, pages 191–205, May 1986.

[Sel87] Timos K. Sellis. Efficiently supporting procedures in relational database systems. In *Proceedings of ACM SIGMOD '87*, pages 278–291, 1987.

[Sel88a] Timos K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2):175–185, 1988.

[Sel88b] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[SF89a] Arie Segev and Weiping Fang. Concurrency-based updates to distributed materialized views. Technical report LBL-27359, Computer Science Research Department, Lawrence Berkeley Laboratory, 1 Cyclotron Road, Berkeley, California 94720, 1989.

[SF89b] Arie Segev and Weiping Fang. Optimal update policies for distributed materialized views. Technical report LBL-26104, Computer Science Research Department, Lawrence Berkeley Laboratory, 1 Cyclotron Road, Berkeley, California 94720, 1989.

[SG89] Arie Segev and Himawan Gunadhi. Event-join optimization in temporal relational databases. Technical report LBL-26600, Computer Science Research Department, Lawrence Berkeley Laboratory, 1 Cyclotron Road, Berkeley, California 94720, 1989.

[Sha86] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[SK86] Arie Shoshani and Kyoji Kawagoe. Temporal data management. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 79–88, August 1986.

[SL89] B. J. Salzberg and D. Lomet. Access methods for multiversion data. In *Proceedings of ACM SIGMOD '89*, pages 315–324, June 1989.

[Sno87] Richard Snodgrass. The temporal query language tquel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[SR88] Jaideep Srivastava and Doron Rotem. Analytical modeling of materialized view maintenance. Technical report, Computer Science Research, Lawrence Berkeley Laboratories. University of California, Berkeley, CA 94720, February 1988.

[SS85] Timos K. Sellis and Leonard Shapiro. Optimization of extended database query languages. In *Proceedings of the ACM SIGMOD '85*, pages 424–436, 1985.

[SS88] Robert B. Stam and Richard Snodgrass. A bibliography on temporal databases. *Data Engineering*, 7(4):53–61, December 1988.

[Sta89] Antonios G. Stamenas. High performance incremental relational databases. Technical report UMIACS-TR-89-49, CS-TR-2245, Department of Computer Science, University of Maryland, College Park, MD 20742, May 1989.

[STNO85] Kazuhiro Satoh, Masashi Tsuchida, Fumio Nakamura, and Kazuhiko Oomachi. Local and global query optimization mechanisms for relational databases. In *Proceedings of the Eleventh International Conference on Very Large Data Bases*, pages 405–417, August 1985.

[TB88] Frank Wm. Tompa and Jose A. Blakeley. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, 1988.

[Ull82] Jeffrey D. Ullman. *Principles of Database Systems.* Computer Software Engineering Series. Computer Science Press, second edition, 1982.

[WY76] E. Wong and K. Youseffi. Decomposition - a strategy for query processing. *ACM Transactions on Database Systms*, 1(3):223–241, September 1976.