UMIACS–TR–89–63
CS–TR–2275

July, 1989

Incremental Implementation Model for Relational Databases with Transaction Time †

Christian S. Jensen‡
Department of Computer Science
University of Maryland

Leo Mark and Nick Roussopoulos
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

The database literature contains numerous contributions to the understanding of time in relational database systems. In the past, the focus has been on data model issues and only recently has efficient implementation been addressed. We present an implementation model for the standard relational data model extended with transaction time. The implementation model exploits techniques of views materialization, incremental computation, and deferred update. It is more flexible that previously presented partitioned storage models. A new and interesting class of detailed queries on change behavior of the database is supported.

† Supported by NSF grant number IRI–8719458 and AFOSR grant number AFOSR–89–0303.

‡ Additional support provided by Aalborg University, Denmark.

1 Introduction

There seems to be general agreement in the database community that efficient and user-friendly time support is needed.

Three orthogonal concepts of time, transaction time, logical time, and user-defined time have been described, see [SA85]. Transaction time is time in the input subsystem of the database system and is therefore application independent. In contrast, logical time is time in the part of reality modeled in the database, and user-defined time is an uninterpreted time domain. The orthogonality of these domains allows us to consider them one at a time. A database supporting only transaction time is termed a *static rollback database*: Once entered, data are logically never deleted, and it is possible to rollback or *time-slice* the database in order to see a previous state. For surveys and further references to work on time extended relational data models, see [SA85] and [BADW82]; for a comparison of some of the best documented temporal data models, see [Sno87].

As a sharp contrast to the extensive amount of work on data models, there has only been done little work on the efficient implementation of temporal databases [SA88]. The main efforts have been in the investigation of partitioned storage strategies, where the fundamental assumption is that fast retrieval is needed for current data while a gracefully degrading performance is allowed for retrievals of older data; see [SA88] [SL89] for presentations and references. Our storage strategy is far more flexible than partitioned storage strategies. While partitioned strategies only support recent data efficiently the model of this paper is able to support efficient access to arbitrary time-slices of arbitrary relations. Another approach [SK86], suggests grid files as a means of implementation. First, they assume an ordering on time and surrogate domains; we find an ordering on surrogates unnatural. Second, no indices on other attribute domains are allowed which again is unsatisfactory to us. For an overview of and further references to efforts relevant to storage of temporal databases, see [McK88].

Our main design objective is to make a small and yet powerfull extension of the standard relational model. Consequently, we choose to time stamp tuples as opposed to attribute values, thus remaining within 1. NF. Also, relations that have a time dimension and thus are three dimensional are required to be time-sliced into normal, "flat" relations before they can be manipulated in the relational algebra. This assures that the standard operators work without modification in the extended model, and it eases implementation. In our efforts to make the extension both semantically and syntactically transparent to users using only the standard features of the model, we introduce for each base relation a backlog relation that records the change history of the corresponding base relation. Furthermore appropriate default syntax has been provided for. The backlogs, when used directly in queries, allow for the retrieval of more detailed information about the change history of

their relations than do any other models at present known to the authors. Updates to relations are entered as time stamped change requests into the backlogs. Propagation to the corresponding relations can use update strategies ranging from eager to lazy. When a query is computed, an access path consisting of intermediate views and a final view is generated. The access paths can be stored as either index caches or materialized data. An index cache for a view is a collection of pointers pointing to the tuples of the views or base relations the view is derived from. The model allows for dynamic control of redundancy. Finally, the computation, and recomputation, of views is done incrementally or decrementally from already computed and stored views.

The view concept is central in the implementation model. During the past ten years much work has centered on the exploitation of views for various purposes. The idea of materializing access paths is presented in [Rou82b] and [Rou82a] where the standard relational data model is the framework. The ADMS± system [RK86] utilizes lazy evaluation and incremental update based on access paths stored as index caches [Rou89]. The implementation model of the present paper can be seen as a natural extension of this work. It it based on a time extended data model. We permit choices between update strategies ranging from lazy to eager; we allow for choosing between storing and not storing access paths, and when storing them we allow choices on how they are stored. Work on how to efficiently maintain views stored as materialized data can be found in [TB88] [BLT86] [BCL86]. This work focus on how to detect irrelevant and (conditionally) autonomously computable updates. Other related work can be found in [Han] [Cam81] [DB78] [AL80] [OS89].

The contents of the remaining sections of the paper are:

Section 2, **Data Structures of the Implementation Model**, describes the different kinds of relations that make up the data structures of the implementation model. First the *backlog* is introduced. Then *base relations* are presented. Storage of fixed and time dependent time-slices of base relations is discussed, and the presentation is generalized to *views*. The last concept, the *differential file*, is presented and the section is concluded with a summary.

Section 3, **Query Evaluation and Redundancy Control**, consists of two parts. In the first, the function of a query evaluation subsystem is described. The decisions to be made by the system in order to efficiently evaluate queries are briefly outlined. In the second part, a number of simplifying assumptions are applied to the general case, and it is described how the system incrementally or decrementally evaluates queries in the simplified setting.

Section 4, **Sample Queries**, shows how sample queries are evaluated in the model presented in the previous two sections. Special focus is put on the helpfulness of backlogs in answering queries on change history of relations.

The last section contains the **Conclusion and Future Research**.

2 Data Structures of the Implementation Model

In this section we present the basic concepts of the storage model. We present the different kinds of relations in the model, i.e. *backlog, base relation, view, backlog view, differential file.* We characterize these relation types according to traditional understandings of base relations and views, and according to persistence and time dependence.

2.1 Backlogs

A *backlog*, B_R, for a relation, R, is a relation that contains the complete history of change requests to relation R.

The schema of relation R and its corresponding backlog is shown in figure 1.

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 1: Schema for the relation R and its backlog, B_{Emp}.

The tuples of backlogs are termed *change requests* because a backlog contains change requests to its corresponding base relation. 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 transaction time stamps, *TTIME*. The value of a time stamp is the time when the transaction - resulting in the request to be stamped - is ready to commit. Thus we assume the existence of a system clock correctly reflecting real time.

The database management system (DBMS) automatically generates a backlog for each base relation (i.e. user-defined and schema relations). A backlog, B_R, of change requests is illustrated in figure 2.

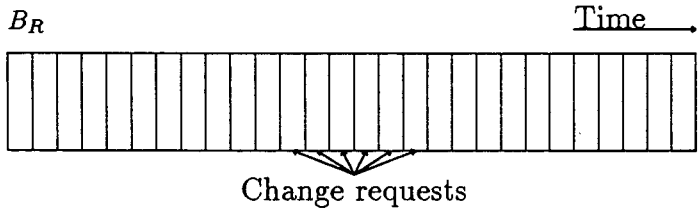

Figure 2: A backlog, B_R, consisting of change requests. For illustrative purposes we assume that change requests are ordered by increasing time stamp values from left to right.

Also the DBMS maintains the backlog relations. Figure 3 shows the effect on backlogs of 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.

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, k)
modify R(k, new value)	insert B_R(id, *Mod*, time, k, new value)

Figure 3: System controlled insertions into a backlog.

EXAMPLE: We introduce a sample database to illustrate the concept of backlog and other concepts to be presented in the sequel. The database consists of one user-defined relation, *Emp*, with three attributes. Figure 4 shows the schema of *Emp* and its backlog, B_{Emp}. Note that relation *Emp* is used as a domain for the "Tuple" attribute. This is only a conyenient shorthand, not a nested attribute (cf. figure 1).

□

Relation name	*Emp*	
Attribute name	*Key*	*Domain name*
Id	nil	*SURROGATE*
Name	K_1	*STRING(20)*
Salary	nil	*INT*

Relation name	B_{Emp}
Attribute name	*Domain name*
Id	*SURROGATE*
Op	{*Ins, Del, Mod*}
Time	*TTIME*
Tuple	*Emp*

Figure 4: Schema for the user-defined relation, *Emp*, and its backlog, B_{Emp}.

2.2 Base Relations

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:

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

When the database is initialized, it has no history and 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. Note, that this feature helps provide transparency to the naive user. We also introduce the special variable NOW which assumes the time when the query is executed.

Time-slices of base relations can be either stored or recomputed when needed. There are two ways of *storing* base relations:

- index cache
- materialized data

Firstly, a base relation can be stored as an *index cache* (or just *cache* for short). A cache is a pointer array, where the entries contain pointers to the associated backlog. The name of this

structure is due to the observation that it inherits both characteristics from caches and indexes. Like a cache it avoids search - all it has to do is to fetch the data pointed to. Like an index, it stores pointers to actual data records, and is buffered into main memory and is read in its entirety [Rou89]. Secondly, a relation can be stored as actual materialized data. The coexistence between caches and materialized data is shown in figure 5.

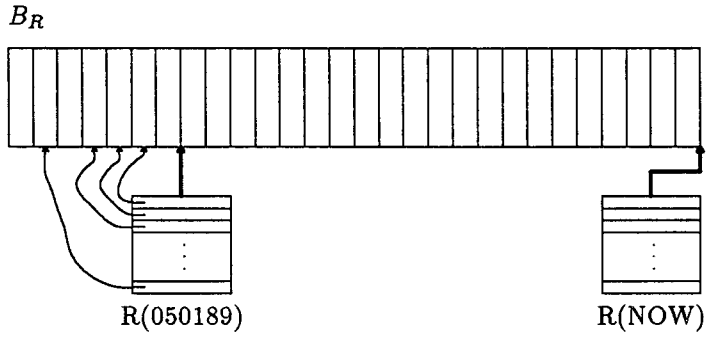

Figure 5: Several differently stored time-slices of a base relation can exist at the same time.

The choice between materializing data or creating a cache reflects a trade-off between replication of data and speed, just as does the choice between creating a cache or nothing at all.

Time-slices of base relations can be

- *fixed*
- *time dependent*

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 never get outdated, time dependent time-slices do. Thus, stored time dependent relations must be updatable. Update strategies ranging from eager to lazy can be adopted. In an eager approach change requests inserted into backlogs are immediately reflected into the corresponding time-slices. In a lazy approach change requests inserted into backlogs at update-time are not reflected in time-slices until at retrieval-time.

EXAMPLE: Figure 6 shows the extension of *Emp* at two points in time. The extension of the corresponding backlog is shown in figure 7.

□

Emp(NOW - 20 days)		
Id	*Name*	*Salary*
"surrogate"	Mark	90 000
"surrogate"	Brown	32 000
"surrogate"	Jensen	10 000

Emp		
Id	*Name*	*Salary*
"surrogate"	Smith	30 000
"surrogate"	Brown	32 000
"surrogate"	Jensen	11 000

Figure 6: Time-slices of a relation, where NOW = 4:00 pm, May 1. 1989.

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

Figure 7: The backlog corresponding to the time-slices in the previous figure. Note that the last three columns together constitute tuples of R and were referenced as a single attribute "Tuple" in the description of the schema of the backlog.

2.3 Views on Base Relations and Backlogs

In the previous subsection we only considered base relations. In this subsection we generalize to views. Views can be created from other views, base relations or backlogs. What was said about base relations in the previous subsection is true for views as well:

- Views can be stored as index caches or as materialized data.
- Views are either time dependent or fixed.

Some explanation and qualification is, however, needed. Cached views have references to the relations and views they are derived from and consist of pointers to these. Views can only be cached if the relations they are derived from are stored. Time dependent views stored as materialized data

only need references to the relations and views they are derived from to facilitate update. Views of this kind also require the relations and views they are derived from to be stored. Finally, fixed views stored as data need no references to the relations and views they are derived from since they never get outdated. Consequently they can be stored independently of whether the relations and views they are derived from are stored or not.

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. Figure 8 illustrates a view.

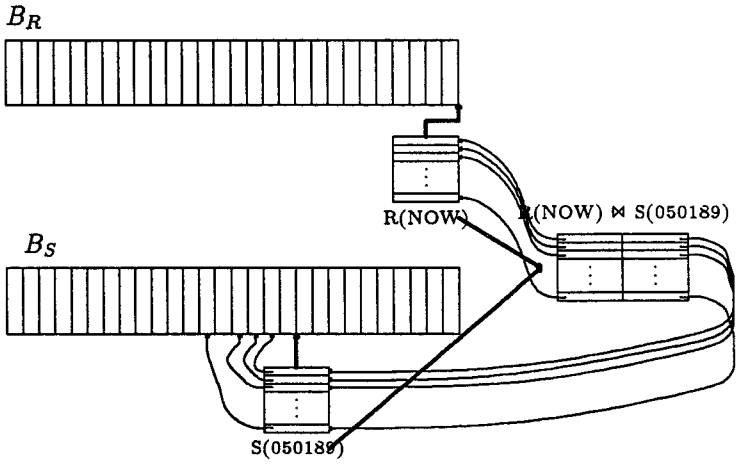

Figure 8: An example of a cached view derived from a materialized and a cached relation. The derived relation is time dependent because the materialized view is time dependent.

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*. For an example, see figure 9.

EXAMPLE: We can use B_{Emp} to retrieve all the employees that, during the last month, had variations in their salaries. This is easily done with the following query:

$$\sigma_{NOW-30\ days \leq Time \leq NOW \wedge Op = Mod} B_{Emp}$$

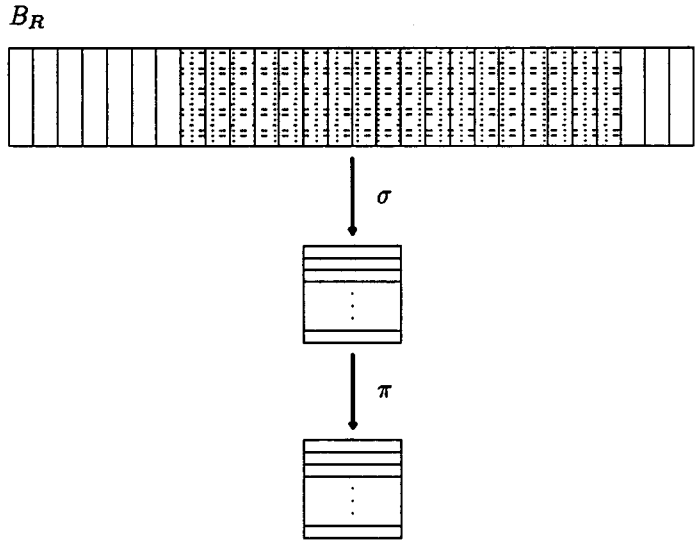

Figure 9: Views can be derived directly from backlogs.

The exact meaning of the query will be made clear in section 4. The result of the query is given in figure 10.

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

Figure 10: A backlog query and its result . NOW = May 11. 1989.

□

2.4 Differential files

Stored, time dependent relations (base relations, views, and backlog views) in general get outdated as time passes and the database is updated. Updates result in insertions of change-requests into the backlogs in the database. Such change-requests have to be reflected in the relations when they are retrieved. If a base relation is to be retrieved, then the relevant change requests since the last retrieval are a set of tuples in the associated backlog; this set of tuples is referred to as a *differential*

file. The differential file of a view is derived from the differential files of the relations and views, the view is derived from and it contains the change requests relevant for updateing the view. While differential files of relations directly derived from backlogs are physically stored, differential files of all other relations (views) are purely conceptual constructs. See figure 11 and 12.

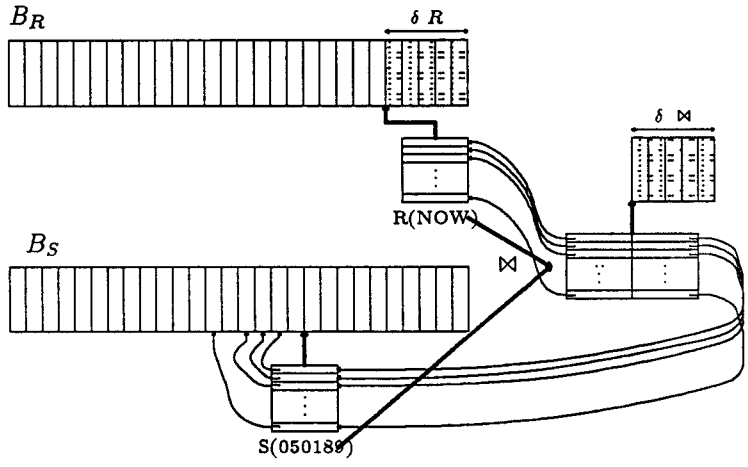

Figure 11: The differential file of a view is derived from the differential files of the relations it is derived from.

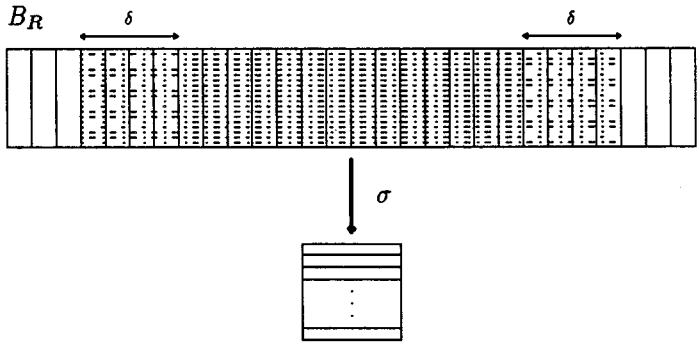

Figure 12: A "sliding window" backlog view.

EXAMPLE: Suppose that we compute the time-slice $R(NOW - 6)$ at time $t = 10$. The time-slice (assuming it is a view cache) will consist of pointers to the change-requests that indicate the valid tuples at time $t = 4$, i.e. it will be pointers to change requests issued before $t = 4$. If we look at this time-slice at $t = 15$, then the change requests after $t = 4$ and before $t = 9$ will constitute the differential file of $R(NOW - 6)$, because now we need to display the valid tuples at $t = 9$. While

fixed views do not get outdated, time dependent views in general get outdated. □

2.5 Summary of Data Structures

It is common practice to distinguish between *views* and *base relations*. In this section we have presented several kinds of relations. To get a better understanding of these, let us look at the generally accepted characterizations of views and base relations. Two dimensions are used to distinguish these. The first is the question of physical storage. Traditionally, base relations are stored while views are computed or virtual. The second is the question of logical derivability. Base relations are not derivable from other relations, while views are. See figure 13 where generally accepted characterizations of base relation and view are summarized [Dat86] [Ull82] [Cod79].

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. The *Emp* relation is an example. 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 info from the database as if it is. A view definition is part of a *subschema*.

Figure 13: The usual definitions of relation types.

We have given the relation concepts new meaning. Every base relation has a backlog. All data of such relations are stored in the relations 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 14. In the sequel, we will use the definitions presented there.

We can summarize the concepts presented in this section as illustrated in figure 15.

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

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. Thus the base relation *Emp* is derivable from B_{Emp} and is not necessarily physically existent.
view	The data of a view still is - directly or indirectly - constructable 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 14: Redefinitions of relation types.

$$\left\{\begin{array}{c}\text{view}\\ \text{base relation}\\ \text{backlog view}\end{array}\right\} \times \left\{\begin{array}{c}\text{time dependent}\\ \text{fixed}\end{array}\right\} \times \left\{\begin{array}{c}\text{view cache}\\ \text{materialized data}\end{array}\right\}$$

Figure 15: Different types of persistent views.

otherwise till they were deleted. Backlog views inherit this notion of lifespan.

The second dimension in figure 15, ***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. 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.

The third dimension of figure 15 is ***persistence***. When the system chooses to store a view, it can be done in two ways. First, a view can be stored as a pointer array, where an entry contain pointers to the relations (view or base) the view is derived from. The schema entry for a view contains

information on how to materialize the view from the data pointed to (recursively). This structure is called an *index cache* (or just *cache* for short) because it inherits characteristics from both caches and indexes. Second, a view can be stored as actual materialized data.

3 Query Evaluation and Redundancy Control

Evaluation of queries and management of storage are closely tied together in the implementation model. First, we present the problem of query evaluation and control of redundancy in the general setting presented in this paper. Second, we make simplifying assumptions to reduce complexity, and describe an algorithm for incremental/decremental computation.

3.1 A General Framework

We are now in a position to outline the function of the *query evaluation subsystem.* We will only present the choices that the query evaluation subsystem must make in order to evaluate queries. How to actually make these choices is left for future research.

In the process of computing a view intermediate views are computed. All intermediate views are treated the same way as is the resulting view.

Let us assume that a query is issued against our database and let us take a closer look at the decisions we have to make in the process of evaluating it. These are outlined in figure 16.

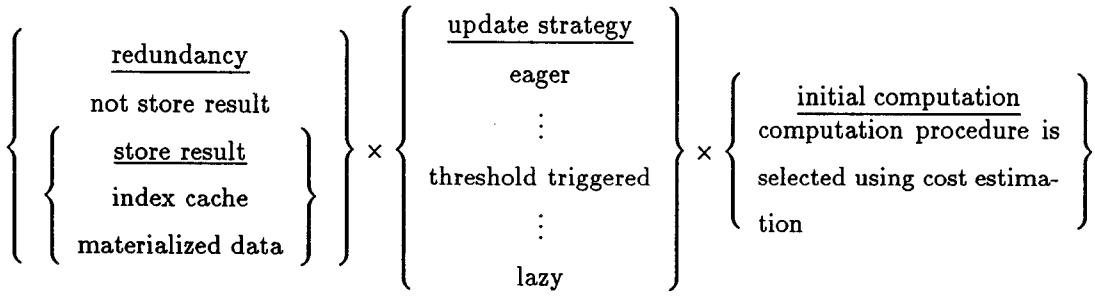

Figure 16: Decisions made by the query evaluation subsystem.

It is decided whether the result (similarly for the subresults) should be stored or not, and, in the case of storing, whether a view cache or materialized data should be chosen. Some restricitons apply: In subsection 2.3 we mentioned that for any time dependent result to be stored, the subresults it is derived from must be stored, too. Also, subresults of a view cached result (fixed or time dependent) must be stored. The decision is made by consulting usage and profile statistics for the database.

These should include update and retrieval frequencies for relations, relation sizes and cardinalities, attribute sizes and cardinalities, domain cardinalities, etc. For references on how to derive and apply such statistics, see [Rou82a] [Rou89] [SAC*79] [Man88]. The idea is to estimate whether the results to be computed might come in handy when future queries are issued against the database. The factors that influence the mutual feasibility of these three alternatives are subject to current research. The fundamental trade-off is one between space/redundancy versus overall computation and retrieval speed.

In case we choose to store the result, we must now decided how it should be updated. We choose between strategies ranging from lazy to eager evaluation. Intermediate strategies such as threshold triggered update based on global system work-load are considered. Note, that these questions only are of interest for time dependent views. Also note, that some restrictions apply: Generally, a relation derived from another relation should not be more eagerly updated than that relation. These are choices of when and how much to process. A lazy strategy might compromise the responsiveness of the system for specific queries but allows for minimal overall processing time, too.

It is decided how to most efficiently compute the query. In general a strategy where already computed - and stored - partial results are either decremented or incremented is selected on the basis of estimated costs. Costs are again computed from usage and profile statistics.

Finally previous decisions on what to store and how to maintain stored views are reconsidered and possibly changed. This is necessary to *dynamically* control the level of redundancy and the trade-off between system overhead and response time.

The sequence in which to carry out the above decisions is not obvious. Even though the activities are inter-dependent there still are possibilities for some concurrency.

There is definitely a need to investigate a general framework as the one presented above, because no single choice is optimal in all situations. Due to the overwhelming complexity of the general framework we suggest that this is done by investigating less general but still promising settings one at a time.

3.2 Lazy Evaluation and Cache Indices

To reduce the complexity of the problem, we make a number of simplifying assumptions on the query evaluation scheme discussed above. We choose to consider only lazy evaluation of time dependent base relations (and views). We disregard base relations, views, and backlog views that are fixed. Base relations and views are stored as index caches. The views corresponding to subresults of queries are stored the same way as are the final results. Our choices are outlined in figure 17. They coincide

with the way the ADMS± system is designed.

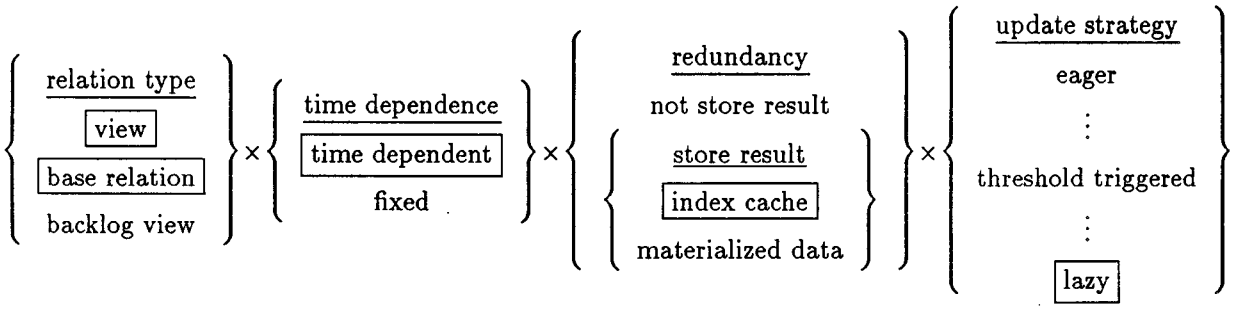

Figure 17: The boxed choices are considered.

The task of computing a query is now reduced to computing and storing time-sliced base relations and views efficiently. This implies the usage of incremental techniques.

A time-slice can be either incrementally or decrementally computed from either older or more recent stored time-slices. The empty relation is the oldest possible time-slice, and the current state of the relation is the newest possible (but it is not necessarily stored) time-slice.

Top Level Freezing

The algorithm *freeze* presented below produces a time-slice $R(t)$ of R at time t from the backlog, B_R, of R. The time-slice is produced from the neighbor on the left or right, either of which is the most promising. See figure 18. Note that the existence of differential files of neighbor time-slices is irrelevant; only the references to the backlog telling when the time-slices were up-to-date are required.

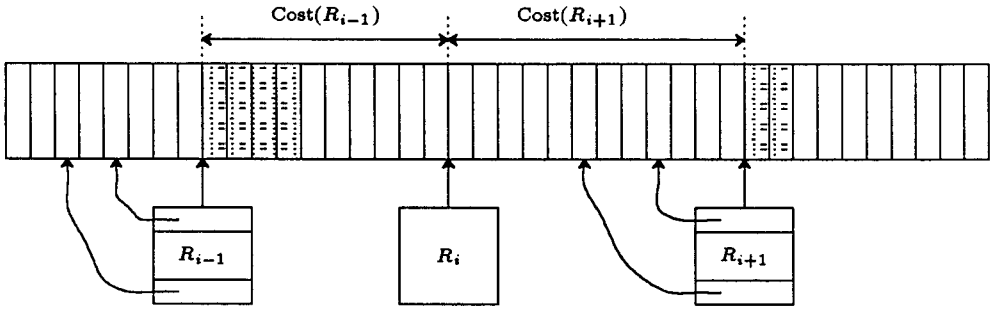

Figure 18: Costs are estimated, compared and a strategy is chosen.

freeze(R,t_i)

Find $R_{t_{i-1}}$, where $t_{i-1} = \max\{t \leq t_i \wedge \exists$ a time-slice at time $t\}$

Find, if it exists, $R_{t_{i+1}}$, where $t_{i+1} = \min\{t \geq t_i \wedge \exists$ a time-slice at time $t\}$

if two time-slices are found

then estimate costs of using each time-slice as the outset

if cost$(R_{t_{i-1}}) \leq$ cost$(R_{t_{i+1}})$

then *increment*(R,t_{i-1},t_i)

else *decrement*(R,t_{i+1},t_i)

else *increment*(R,t_{i-1},t_i)

The two subroutines *increment* and *decrement* used by *freeze* are described in the following subsections.

Incremental Freezing

Figure 19 illustrates incremental freezing.

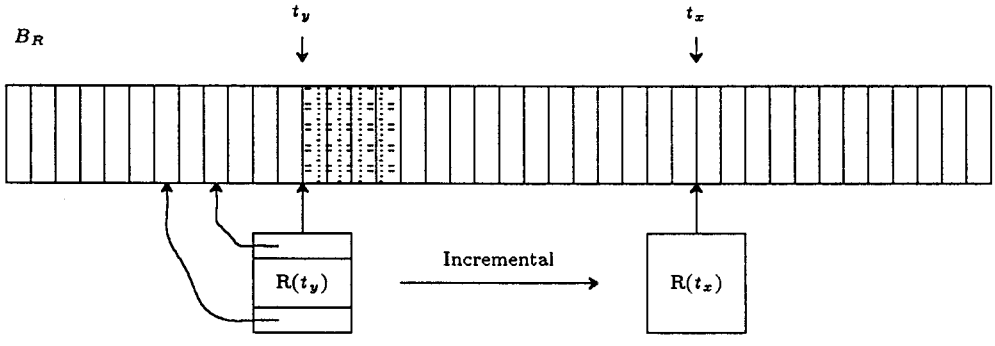

Figure 19: Incrementing an existing time-slice.

increment(R,t_y,t_x)

Res $\leftarrow R(t_y)$

$t \leftarrow t_y$

while change requests with time stamps between t and t_x

do

pick oldest change request, at t', bigger than t

update $t \leftarrow t$'

case request type

DELETE

remove from Res the pointer pointing to delete-requested tuple

INSERT

insert into Res pointer to change-request tuple

MODIFY

insert into Res pointer to change-request tuple

return($R(t_x) \leftarrow$ Res)

Decremental Freezing

Figure 20 shows decremental computation of a view.

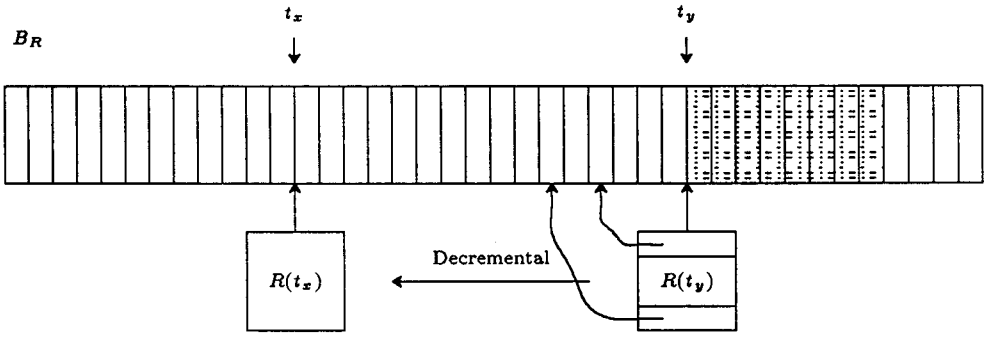

Figure 20: Undoing change requests.

decrement(R,t_y,t_x)

Res $\leftarrow R(t_y)$

$t \leftarrow t_y$

while change requests with time stamps between t and t_x

do

pick newest unmarked change request, at t', less than t

update $t \leftarrow t$'

case request type

DELETE

insert into Res the pointer pointing to delete-requested tuple

INSERT

remove from Res the pointer pointing to insert-requested tuple

MODIFY

remove from Res the pointer pointing to modify-requested tuple

return($R(t_x) \leftarrow$ Res)

Computation of Views

Above we only covered base relations. Here we give an example of how a view can be computed. Assume we have base relations R, S and T, and $V(t)$ is defined as follows

$$V(t) \stackrel{\text{def}}{=} \sigma_{F_1} R(t) \bowtie (\sigma_{F_2} S(t) \bowtie \pi_A T(t))$$

To compute $V(NOW)$, ignoring for clarity all the permutations from standard query optimization, we break the defining expression into subexpressions looking for stored results that can be used in incremental computations:

1. If $V(t)$, for some t, has been computed already then at least one - possibly outdated - version of V exists, and becomes the chosen. If several views containing $V(t)$, again for arbitrary values of t, exist then the one with the lowest estimated cost is chosen. The chosen view is then used in an incremental computation of $V(NOW)$.

2. If $V(t)$ has never been computed, time-slices containing $\sigma_{F_1} R(t)$ are located; one is chosen, and $\sigma_{F_1} R(NOW)$ is computed using the algorithms above.

3. In case $V(t)$ had not already been computed, we now continue with the next subexpression in the definition of $V(t)$. If a view containing

   $$\sigma_{F_2} S(t) \bowtie \pi_A T(t)$$

   has already been computed, the most promising such one is selected and used in the incremental computation of the subview. Otherwise, the two subexpressions are computed similarly to the computation involving R.

4. Finally, when subexpressions are computed they are combined in further computations and the final result is achieved.

Many details of the computations of views are left unspecified in the description above. Specifically, easily computable and good cost estimation formulas for choosing between candidate views are subject to current research.

4 Sample queries

The time extension and the additional data structures require an extended operation language. We use the standard relational algebra as a basis for such an extension. Since base relations must be time-sliced before they can be arguments of algebra operators and since system generated relations are already "flat" in the same sense, the operators of the standard relational model work in our setting without modifications. Our transaction time extension is very small and it is also transparent. This simplicity is a major advantage of our design. In subsection 4.1 we present and shortly discuss the operators of our data model. In subsection 4.2 we illustrate the utility of parts of the operation language. Also, we describe how to evaluate queries in terms of the storage model and algorithm in the previous section.

4.1 Operators and Notation

In figure 21 we present the basic operators of our operation language. Note the special variable "NOW" that has as value the time of the transaction it is a part of when used in a query. It is useful when specifying conditions on transaction time attributes. We will not discuss the operators of figure 21 in further detail.

From a conceptual point of view user-defined relations are historical, i.e. they have transaction time attributes. These attributes are crucial for time-slicing to be meaningful. The combination of the facts that we only manipulate time-sliced relations and that we want to comply with the *transparency principle* has resulted in the choice of hiding transaction time attributes in user-defined and schema relations. To display a transaction time attribute it must be *explicitly selected.* In system generated relations time stamp attributes are displayed. A simple *projection* is required to remove such an attribute.

In connection with the time domain of our data model we add the ability to specify the *time unit.* Since we have chosen second to be the lowest unit, second, minute, hour, day, week, month and year are possible values. We use a *unit* function with these values. A query with specified unit could look like this.

$$\sigma_{t_x \leq Time \leq t_y \wedge unit(Time)=day} B_S$$

The default unit is minutes.

4.2 Sample Queries

We present a sequence of queries gradually getting more and more complicated. For each it is indicated how incremental evaluation is applied. At first queries on traditional base relations and

views are presented. Then it is discussed how queries on backlogs are helpfull in answering queries on change history.

Retrieve all employees as of close of business May 1. 1989.

$$Emp(0501891600)$$

This is an example of a fixed time-slice of a base relation. See figure 6. If no other queries have been issued on the relation *Emp* this view is incrementally computed from $Emp(t_{init})$. The result stored in the database is a set of pointers to *Ins* and *Mod* tuples in B_{Emp} and a reference to B_{Emp} that indicates the time of validity of the time-slice[1].

Retrieve all current employees.

$$Emp(NOW) \text{ or, alternatively: } Emp$$

This is a time dependent time-slice of a base relation. When this view is first computed the previous fixed time-slice is utilized in an incremental computation. Later retrievals will utilize the immediate predecessor in an incremental computation. The differential file of this view is all the change requests to *Emp* that have arrived after it was last retrieved/computed.

Retrieve the employees that were in the company 20 days ago (as of now).

$$Emp(NOW - 20days)$$

This query is very similar to the previous one. Here, however, the differential file is the change requests with time stamp values between $NOW - 20$ *days* when the query is evaluated and the current value of $NOW - 20$ *days*. A computation of this query will utilize that of the previously computed and stored views from above that has the lowest estimated cost.

Let us *define* a view as follows:

$$Rich\_Emp(t) = \sigma_{Salary \geq 50000} Emp(t)$$

A definition does not result in any computation, all that happens is that the query expression itself is stored in the database system. The first step in evaluating any query is to time-slice the constituent relations. So, to retrieve data from the view, an expression that evaluates to a value in the domain *TTIME* must be supplied and substituted for t. Then the selection is computed.

Retrieve all very rich employees as of close of business May 1. 1989

$$\sigma_{Salary \geq 80000} Rich\_Emp(0501891600)$$

[1] The time of validity is the half-opened interval that contain 050189160000 and is bound by the two closest time stamps in B_{Emp}.

This is a fixed time-slice that involves several levels of computation. First, *Rich_Emp* is substituted for its definition (as in Ingres style query modification). Then, second, the time-slice is computed. Third, the selection(s) is performed. Depending on whether the two selections are collapsed into one (the second) the computation of the query results in three or two separate index caches: The initial time-slice, (the first selection,) the second selection.

Retrieve all very rich employees

$$\sigma_{Salary \geq 80000} Rich\_Emp(NOW)$$

This query differs from the previous one in that it is time dependent. To compute the very rich employees the *Rich_Emp* in general must be brought up to date first.

Let us now turn our attention towards queries directly involving backlog relations. Initially, however, let us look at how the usefulness of backlog relations supplement that of usual relations.

Suppose we want **the changes to** *Emp* **between** t_x **and** t_y. Although not the only possibility, we might write something like this:

$$Emp(t_y) - Emp(t_x)$$

The result of this query is the tuples in *Emp* at t_y not in *Emp* at t_x. This does not tell us what took place between t_x and t_y. We will not retrieve any deletions that might hve taken place in the time interval, for example. If we really want to know what took place between t_x and t_y we would be better off using the backlog of *Emp*. We have to make clear what we precisely want. Let us look at some possibilities.

$$\sigma_{t_x \leq Time \leq t_y} B_{Emp}$$

This query retrieves all possible information about what happened to *Emp*. Insertions, deletions and modifications are distinguished and the times when the requests were placed are available.

$$\pi_{Tuple} \sigma_{t_x \leq Time \leq t_y} B_{Emp}$$

This query eliminates the special backlog attributes from the result. Thus several changes back and forth between identical *Emp* tuples will be eliminated and it will not be possible to distinguish between operation types anymore.

$$\pi_{Tuple \wedge Time} \sigma_{t_x \leq Time \leq t_y} B_{Emp}$$

This result of this query differs from the above in that time stamps are retained, potentially allowing more tuples to be retrieved. Still, it is impossible to distinguish a modification from a deletion which in many cases may be unfortunate.

$$\pi_{Tuple \wedge Time} \sigma_{t_x \leq Time \leq t_y \wedge Op=Ins} B_{Emp}$$

Here we get the time stamped tuples that were inserted in the interval. The result is a list containing the employees hired in the interval.

$$\pi_{Tuple \wedge Time} \sigma_{t_x \leq Time \leq t_y \wedge Op=Del} B_{Emp}$$

Here we retrieve the employees tht left the company.

$$\pi_{Tuple} \sigma_{t_x \leq Time \leq t_y \wedge (Op=Ins \vee Op=Mod)} B_{Emp}$$

Finally, we have retrieved all employees that "changed" salary, either because they were hired or because their previous salaries were updated.

The possibilities listed above are by no means exhaustive but somewhat representative for vast number of easily formulated queries possible on backlogs. Let us now take a look at the evaluation of other kinds of queries.

To get **the first employee to leave the company after April 30, 1989**, we use the min operator.

$$min(\sigma_{043089 \leq Time \wedge Op=Del} B_{Emp})$$

Since change requests are assumed to be ordered according to time stamps the system need not first retrieve all "*Del*" change requests inserted after April 30. and then find the one with the smallest time stamp value. Instead the qualifying tuple can be retrieved directly.

Find all the employees at t_x that changed salary between t_a and t_b. This and the following query involves both a time-sliced base relation and a backlog.

$$Emp(t_x) \rhd \pi_{Tuple\ } \sigma_{t_a \leq Time \leq t_b \wedge Op=Mod} B_{Emp}$$

The compliment is given by

$$Emp(t_x) - \pi_{Tuple\ } \sigma_{t_a \leq Time \leq t_b \wedge Op=Mod} B_{Emp}$$

Note that the results of these queries are fixed: Once computed they never get outdated.

The following query results in **a list of** $(Name, Time)$ **of employees with salary change on December 27, 1988, but with a time granularity of one hour.**

$$\pi_{Name,unit(Time)=hour}\ \sigma_{27128800 \leq Time \leq 28128800 \wedge\ Op='Mod'} B_{Emp}$$

The *unit* operator rounds-off all time values to the closest value in the new unit of measure. Thus, if an employee changes salary more than once within the same hour (e.g. as a result of a typing error and a following correction) this will not be visible in the result.

Suppose we want **the employees that changed salary about the average number of times during the last 2 years.** This can be done with the queries:

$$Q_1(Name, Count) = \pi_{Name,count(Time)} \sigma_{Op='Mod' \wedge NOW-2\ yrs. \leq Time \leq NOW} B_{Emp}$$

$$Q_2 = avg(\pi_{Count} Q_1)$$

$$Q_3 = \pi_{Name} \sigma_{.8*Q_2 \leq Count \leq 1.2*Q_2} Q_1$$

In Q_1 we count, for each employee, the number of times the salary was changed within the given 2 year period. The attributes of the view are named *Name* and *Count*. This is an example of a sliding time window on a backlog screening out and aggregating relevant tuples. See figure 12. Both incremental and decremental techniques are used to keep Q_1 up to date. In Q_2 we then find the average number of changes. In Q_3 we finally compare the number of changes of an employees salary to the average number of changes and keep employees that are close.

We retrieve **the employees with abnormal change pattern** by

$$\pi_{Name}\ \sigma_{Op='Mod' \wedge NOW-2\ yrs. \leq Time \leq NOW} B_{Emp}\ -\ Q_3$$

If we only want the employees with very few salary changes during the last 2 years, we can issue this query, using Q_1 and Q_2 above:

$$\pi_{Name} \sigma_{0 \leq Count \leq .2*Q_2} Q_1$$

In summary, we have shown how to conveniently retrieve detailled information about change history of relations. We have used incremental computation of queries formulated in the operation language of our extended data model. In particular we have demonstrated the convenience of the backlog relation.

5 Conclusion and Future Research

We have presented a relational data model extended with transaction time and an incremental implementation model for it.

The data model only require one new operator, the time-slice operator. All the standard relational model operators still work with their standard semantics. The extension is transparent to users not using the extended capabilities of the model. The model allow for easy retrieval of detailled information about change history.

The implementation model exploits techniques for eager/lazy update and incremental/decremental computation in the context of persistant views stored as either index caches or materialized data. The model is a natural generalization of the work of Roussopoulos.

Several topics touched upon in this paper are subject for current research. Most prominently, we are further investigating the general framework discussed in subsection 3.1. Sets of rules and cost estimation formulas for the decisions of the query evaluation subsystem are being developed. These include rules for when and how views should be stored; rules for how often a stored, time dependent result should be updated; rules for which existing results should be used in incremental computations.

Topics for future research include:

- Extending the data model with complex objects.
- Incorporating a general version handling mechanism into the data model.
- Extending the data model and the implementation model to support logical time.
- Extending the query language to support queries on change behaviour. This includes investigation of facilities for detection of common patterns of behavior and deviations from these.

Standard Operators of the Operation 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. This notation is slightly more general, because time intervals can be easily expressed. In the present setting where only points in time are meaningful, 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 relation.
σ_F	Selection	The standard selection operator. The condition F can contain an arbitrary sub-query.
$\times$	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.
"aggregating functions"		We allow for a full range of aggregating functions: max, min, mean, count, avg, sum, product, unit.
:=	Assignment	This operator has the usual assignment semantics.

Figure 21: Notation, names and descriptions of basic operators.

References

[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.

[BCL86] Jose A. Blakeley, Neil Coburn, and Per-Åke Larson. *Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates.* CS-86-17, University of Waterloo. Computer Science Department, May 1986.

[BLT86] Jose A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD '86*, pages 61–71, May 1986.

[Cam81] Stephanie Cammarata. Deferring updates in a relational data base system. In *Proceedings of the Seventh International Conference on Very Large Databases*, pages 286–292, 1981.

[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.* Volume first of *The Systems Programming Series*, Addison Wesley Publishing Company, fourth edition, 1986.

[DB78] Umeshwar Dayal and Philip A. Bernstein. On the updatability of relational views. In *Proceedings of the Fourth International Conference on Very Large Data Bases*, pages 368–377, 1978.

[Han] Eric Hanson. *A Performance Analysis of View Materialization Strategies.* , Department of Electrical Engineering and Computer Sciences - University of California, Berkeley, CA 94720, .

[Man88] Michael V. Mannino. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, September 1988.

[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, Departmen of Computer Science, CB 3175, Sitterson Hall, Chapel Hill, NC 27599-3175, October 1988. Ph.D. Dissertation.

[OS89] Anthony B. O'Hare and Amit P. Sheth. The interpreted-compiled range of ai/db systems. *Sigmod Record*, 18(1):32–42, March 1989.

[RK86] Nick Roussopoulos and Hyunchul Kang. Principles and techniques in the design of *adms±*. *Computer*, ():19–25, December 1986.

[Rou82a] Nick Roussopoulos. The logical access path schema of a database. *IEEE Transactions on Software Engineering*, 8(6):, November 1982.

[Rou82b] Nick Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems*, 7(2):, June 1982.

[Rou89] Nick Roussopoulos. *The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis.* , Department of Computer Science, University of Maryland, College Park, MD 20742, March 1989.

[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.

[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.

[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.* Volume of *Computer Software Engineering Series*, Computer Science Press, second edition, 1982.