Incremental update propagation in materialized views*

Renato Barrera
Douglas L. Hudson
Andrew U. Frank
Department of Surveying Engineering
University of Maine
Orono, ME 04469, USA

October 20, 1989

Abstract

Materialized views are supported in order to decrease system response-time. To maintain efficiency in views, it is important to support update methods that will first consider the changes in the base relations, and if those modifications are relevant, will propagate them to the views in an incremental and localized manner. One way to accomplish this is to maintain *minimal view-states*, i.e., maximally reduced intermediate results that can be updated without reference to their component objects.

This paper is concerned with the construction of minimal view-states by a *transformational* approach, i.e., one that applies a succession of operators to the basic constituents of a view. This work initially sets forth two conditions on intermediate results necessary and sufficient to qualify them as a view-states. It afterwards presents algorithms for the synthesis of view-states; these algorithms start with the base objects as view-states and then apply sequences of transformations that reduce the states and yet keep invariant the aforementioned conditions. A byproduct of these algorithms is the formulation of methods for the detection of relevant updates and for the implementation of incremental update policies.

In this paper the relational model is taken as an illustration, and using this model, the results of the transformational approach are briefly compared to those resulting from an approach based upon *initial algebras*; this latter approach guarantees a maximum reduction of view-states. It was found that to achieve efficient reduction of view-states by means of the transformational approach, we must resort to more powerful languages (i.e., relational languages with aggregate functions).

Work is being done for extending this results to object-oriented models with aggregation and thus to handling a wider class of queries.

1 Introduction

DBMS provide mechanisms, called *views*, that enable the user to work with derived objects, i.e., objects that are a composite of one of more base objects and, possibly, of other views. A view may be either temporarily derived to assist in the resolution of a particular query (a *logical* or *virtual* view) or computed and stored in permanent memory as a *materialized* view in anticipation of a particular class of queries. In the later case, when changes in the base objects occur, the DBMS must keep the consistency of the views either on-the-fly or by using a *lazy* strategy.

*This research was partially funded by grants from DEC (ESDS) (Principal Investigator: Andrew U. Frank). The support from NSF for the NCGIA under grant number SES 88-10917 is gratefully acknowledged.

The focus of this paper is on materialized views and all references to views will implicitly be to this type unless otherwise stated.

For the purposes of user inquiry a view is indistinguishable from a base object. Views are, however, supported for many good reasons:

i) to reduce the level of detail for the user;

ii) to provide some amount of data security at lower DB levels;

iii) to isolate the user from certain changes in the database;

iv) to assist in the transportability of application programs;

v) to augment query processing speed and reduce data transmission time.

Two aspects of updating materialized views have received considerable attention in the literature: how a view can be efficiently updated when a change is executed upon one or more of its base objects (propagation of updates) [ROSE89] [BLAK89] [BLAK86] [BUNE79] and how a base relation can be modified if a change is attempted upon a view (distribution of updates) [DAYA82]. In a further restriction, this paper discusses only the propagation of updates.

There is an obvious trade-off between query processing speed and storage costs with materialized view maintenance; equally important, however, is a concern with query response time versus view modification time. If the entire view must be reconstructed every time some of its components are updated, then its utility may be seriously reduced. Incremental update strategies may provide a solution to the view update problem.

As stated in [ROSE89] views are implemented by observation operators on an object and can admit to an *incremental* representation with the following characteristics:

i) Upon a change, both basic objects and views can be associated with an incremental result that conveys the differences between the *before* and *after* states of the database.

ii) All data validation is done at the base object level.

iii) A change to a view considers the increments of its composing objects as the only external information available.

iv) No *race* situations occur, i.e., if a view must accommodate several changes in its composing objects, the implementation must be such that the order of arrival of the changes is one that gives the desired final result. This can be done by methods akin to those used for firing rules while maintaining integrity constraints [CASA88] or in production systems [COHE89].

A *view-state* is a set of objects associated to a view; the objects in question can be taken from any of three sources: i) the view itself ii) the base objects iii) objects intermediate between the base objects and the view. That aforementioned set of objects, to qualify as a view-state, must fulfill two conditions:

i) The final view can be derived from the view-state.

ii) Updates to the view-state can be effected without reference to any base object that is not a member of the view-state.

The set of base objects constitute, of course, a trivial view-state; however, it is convenient to search for a *minimal* view-state onto which an homomorphism from any other view-state exists.

In the ideal case, updates are *autonomously* computable, i.e., the minimal view-state coincides with the final view. Reference [BLAK89] has given necessary and sufficient conditions for autonomous computability for a project-select-join view in relational algebra.

Even when autonomy can not be satisfied, it might still be more efficient to store the minimal view-states rather than to perform incremental updates of the materialized view based upon the base objects.

This paper discusses methods for reducing view-states, and is organized as follows: the next section introduces the concepts of view-states and minimalism, using the framework of the relational algebra. Section 3 discusses an algorithm for reducing view-states, and the next section reviews an example using the intial algebra approach. Finally, section 5 presents some comments and future research lines.

2 View-States and Minimalism

This section will introduce, using the terminology of relational algebra, the framework needed by the algorithms for the reduction of view-states that will be presented in section 3. In particular, the concepts of view-state, of homomorphism between view-states, and of *split* operators are introduced. A skeleton for a view-state reduction algorithm is explained. These concepts may be extended to any object-oriented formalism that is provided with suitable associativity, commutativity, etc., axioms.

2.1 Version operators, Consistency and Factorability

Definition A *database* $\Re$ is the set of relational schemata $\{R_1, \cdots, R_m\}$ used in a particular application. Each R_i is a schema for either a base relation or view. The standard notation for algebraic relational operators will be used [ULLM80] i.e., $\bowtie_{predicate}(.,.), \sigma_{predicate}(.), \Pi_{attr\_list}(.), \times, -, \cup, \cap$ will stand respectively for the Join, Selection, Projection, Cartesian product, Difference, Union and Intersection operators.

Throughout this work, we will use two base relations in our examples. The schemata for these two relations are *Emp(E_SSN, E_name, Salary)* and *Child(E_SSN, Child_Name, Age)*.

Definition A *view* is an expression $Vw(A_1, \cdots, A_n)$ in relational algebra, where A_i's are either base relations or (nested) subviews. The recursive substitution of subviews by their corresponding expressions must end in base relations. $\underline{A}$ will be used as shorthand for $\{A_1, \cdots, A_n\}$.

Versions We will consider a model similar to the one used in [COHE89], in which during a transaction, base relations A_i are provided with *before* and *after* versions, denoted by $\sim A_i$ and $A_i \sim$. This has several advantages: it simplifies the satisfaction of *no-race* conditions of updates, it facilitates the undoing of transactions, and it allows the inclusion of some temporal constructs used in active databases, such as the **Previous** and **Start** constructs found in [COHE89].

As an example, the following expressions are views on $\Re$:

Example 1:

$Names = \Pi_{E\_name}(Emp)$ Names of Employees
$Rich\_Emp = \sigma_{Salary>32K}(Emp)$ Affluent Employees
$Rich\_Tots = \Pi_{Child\_Name}(\sigma_{Age<6}(Child) \bowtie \sigma_{Salary>32k}(Emp))$ Affluent Children

□

The parse trees of those three views are shown in Fig. 1.

Version Operations

Two operators $DFF(\,)$, $CMP(\,)$ are needed to perform the forward and backward transitions between the versions $(\sim A_i, A_i\sim)$ of base relation A_i:

Operator $DFF(\,)$ reports the difference between two successive versions of A_i. Its properties are:

$$DFF(\sim A_i, A_i\sim) = \Delta\sim A_i \qquad \text{(forward difference)}$$
$$DFF(A_i\sim, \sim A_i) = \Delta A_i\sim \qquad \text{(backward difference)}$$

$\Delta\sim A_i$ and $\Delta A_i\sim$ express the forward and backward changes between the versions. We shall sometimes, for brevity, refer to $\Delta\sim A_i$ as ΔA_i.

Example 2: Modifying a relation.

Suppose that relation $\sim Child$ is updated to $Child\sim$. The results of the Δ are:

$$\Delta Child = DFF(\sim Child, Child\sim) = \quad \{$$
insert <4312, Ralph, 3>
delete <4913, Xavier, 21>
insert <4057, Ruth, 2>
$\cdots\}$

and they represent the relevant changes between those two versions. To reduce view-states, only insertions and deletions to the basic relations will be considered; all other modifications will be represented as sequences of those two operations. □.

The composition operator, $CMP(\,)$ combines a version with its changes. Its properties are:

$$CMP(\sim A_i, \Delta\sim A_i) = A_i\sim \qquad \text{(forward change)}$$
$$CMP(A_i\sim, \Delta A_i\sim) = \sim A_i \qquad \text{(backward change)}$$

The composition operator executes the appropriate set union or set minus operations for the respective tuple insert or deletes found in ΔA_i.

The composition and difference operators obey the following axioms:

$$DFF(\sim A_i, \sim A_i) = DFF(A_i\sim, A_i\sim) = \emptyset$$

$$CMP(\sim A_i, DFF(\sim A_i, A_i\sim) = A_i\sim$$

$$CMP(A_i\sim, \mathrm{DFF}(A_i\sim, \sim A_i) = \sim A_i$$

Definition: Let $Vw(\underline{A})$ be a view on $\underline{A}$, and $\{St_1(A_1,\cdots,A_n),\cdots,St_m(A_1,\cdots,A_n)\}$ be an indexed set of relational algebraic expressions on $\underline{A}$, abbreviated to $\underline{ST}(\underline{A})$. A particular $\underline{ST}(\underline{A})$ is a *view-state* of the view $Vw(\underline{A})$ if the conditions of *consistency* and of *factorability*, set forth below, are met.

Consistency Condition - There exists an expression $OBS(\,)$ (*observer expression*) in relational algebra that obeys the equation:

$$Vw(\underline{A}) = OBS(\underline{ST}(\underline{A})) \qquad (1)$$

□.

Example 3: View-states for the views of example 1.

Fig 2. shows three different view-states and $OBS(\,)$ expressions for view $Names$, while Fig. 3 does the same for view $Rich\_tots$. In Fig. 2 we can see that, when $St(Emp) = \Pi_{Emp\_Name}(Emp)$,

the observer expression is the identity, and when $St(Emp) = \Pi_{Emp\_Name,Salary}(Emp)$ the observer is $\Pi_{Emp\_Name}(St(Emp))$.

Factorability Condition- There exist expressions $\underline{FCT} = \{Fct_1, \cdots, Fct_m\}$ (*factoring expressions*) such that:

$$\underline{ST}(\sim A) = \underline{FCT}(\underline{ST}(A\sim), \Delta\sim A) \tag{2}$$

$$\underline{ST}(A\sim) = \underline{FCT}(\underline{ST}(\sim A), \Delta A\sim) \tag{3}$$

□.

Equation (2) specifies that the *before* state-view can be obtained as a function of the *after* state-view and the forward changes only, while eqn. (3) specifies a symmetrical condition for the *before* state-view.

The factoring expression must be thus able to compute the changes in the view-state as a composition of two factors, one involving the existing view-state and another one related to the changes in the base relations only. The proof (or disproof) of this fact is nontrivial for higher-level view-states. Successful factoring depends upon the algebraic properties of association, commutation, and distribution among the relational operators involved in the view-state expression.

All composition and difference operators for relations can be expressed in the form $A_i \cup \delta A_i$ or $A_i - \delta A_i$ respectively. Thus, we will use those operators to exemplify factoring.

Example 4: View-state for a cartesian product

Let us consider two base relations A_1, A_2 and the expression $St(A_1, A_2) = A_1 \times A_2$. The Cartesian product is symmetric, so we only have to prove that $A_2 \cup \delta A_2$, $A_2 - \delta A_2$ can be factored. Using

$$\begin{aligned} St(A_1, A_2 \cup \delta A_2) &= A_1 \times (A_2 \cup \delta A_2) \\ &= (A_1 \times A_2) \cup (A_1 \times \delta A_2) \\ &= St(\underline{A}) \cup (\Pi_{dom A_1}(St(\underline{A})) \times \delta A_2) \qquad (4) \\ St(A_1, A_2 - \delta A_2) &= A_1 \times (A_2 - \delta A_2) \\ &= (A_1 \times A_2) - (A_1 \times \delta A_2) \\ &= St(\underline{A}) - (\Pi_{dom A_1}(St(\underline{A})) \times \delta A_2) \qquad (5) \end{aligned}$$

We can see that the right hand sides of eqns.(4, 5) has the form prescribed by equations (2, 3) and thus the state $St(A_1, A_2)$ can be factored.

Example 5: View-state for a projection operator.

Let us consider a A_1 and the expression $St(A_1) = \Pi_{\{attr_1\}}(A_1)$, where $\{attr_1\}$ is a subset of $R(A_1)$. If $\{attr_1\}$ contains a key of A_1, then

$$\begin{aligned} St(A_1 \cup \delta A_1) &= \Pi(A_1 \cup \delta A_1) \\ &= \Pi(A_1) \cup \Pi(\delta A_1) \\ &= St(A_1) \cup \Pi(\delta A_1) \\ St(A_1 - \delta A_1) &= \Pi(A_1 - \delta A_1) \\ &= \Pi(A_1) - \Pi(\delta A_1) \\ &= St(A_1) - \Pi(\delta A_1) \end{aligned}$$

□.

Example 6: View-state for the Union operator.

Let us consider the expression $St(A_1, A_2) = A_1 \cup A_2$. Because the operator is symmetrical, we may be concerned with the changes to A_2 only, for the changes to A_1 will be handled in a similar way. The inclusion of new tuples to A_2 can be factored if we remember that $A_1 \cup (A_2 \cup \delta A_2) = (A_1 \cup A_2) \cup \delta A_2$. Deletions, however, cannot be factored since:

$$\begin{aligned} A_1 \cup (A_2 - \delta A_2) &= (A_1 \cup A_2) - (\delta A_2 - A_1) \\ &= St(A_1, A_2) - (\delta A_2 - A_1) \end{aligned}$$

and A_1 can not be obtained from $A_1 \cup A_2$ □.

In a similar fashion, it can be proved that $St(A_1, A_2) = (A_1 \bowtie A_2)$ can not be factorable, since $A_1 \bowtie (A_2 - \delta A_2)$ is equal to $St(\underline{A}) - (A_1 \bowtie \delta A_2)$ and it is in general not possible to reconstruct A_1 from $(A_1 \bowtie A_2)$.

The following table gives the factorability properties of the five basic operators of relational algebra.

Table 1.

STATE $St()$	Fact. Eqn. for $(A_2 \cup \delta A_2)$	Fact. Eqn. for $(A_2 - \delta A_2)$
$A_1 \cup A_2$	Non factorable	
$A_1 - A_2$	Non factorable	
$A_1 \times A_2$	$St() \cup (\Pi(St()) \times \delta A_2)$	$St() \cup (\Pi(St()) \times \delta A_2)$
$\sigma(A_2)$	$St() \cup \sigma(\delta A_2)$	$St() - \sigma(\delta A_2)$
$\Pi_{\{X\}}(A_2)$[1]	$St() \cup \Pi_{\{X\}}(\delta A_2)$	$St() - \Pi_{\{X\}}(\delta A_2)$

[1] where {X} contains a key to A_2

2.2 Reduction of view-states

We will now proceed to define homomorphism between view-states, and to state conditions for reducing view-states and for having minimal view-states.

Definition Let $\underline{ST} = \{St_1, \cdots, St_j\}$ and $\underline{ST'} = \{St'_1, \cdots, St'_k\}$ be two view-states defined on the same view and base relations. Let $OBS(\underline{ST})$, $OBS'(\underline{ST'})$, $\underline{FCT}(\underline{ST})$, $\underline{FCT'}(\underline{ST'})$ be their respective observer and factoring expressions.

There exists an *homomorphism* from $\underline{ST}$ onto $\underline{ST'}$ if there exists an indexed set of expressions $\underline{TR} = \{Tr_1, \cdots, Tr_k\}$ (transformation expressions) fulfilling the two followings equations:

$$OBS(\underline{ST}) = OBS'(\overbrace{\underline{TR}(\underline{ST})}^{ST'}) \tag{6}$$

$$\underline{FCT'}(\overbrace{\underline{TR}(\underline{ST})}^{ST'}, \delta\underline{A}) = \underline{TR}(\underline{FCT}(\underline{ST}), \delta\underline{A})) \tag{7}$$

□.

Equation (6) specifies that the resulting observed value is the same in either case, while Equation (7) means that the factor equations for $\underline{ST'}$ and $\underline{TR}(\underline{ST})$ can be interchanged.

Definition $\underline{ST'}$ *reduces* $\underline{ST}$ if there is an homomorphism from $\underline{ST}$ onto $\underline{ST'}$, but there is no homomorphism from $\underline{ST'}$ onto $\underline{ST}$; the expression $\underline{TR}()$ that performe $\underline{ST'} = \underline{TR}(\underline{ST})$ is called the *reducer*. □.

Loosely speaking $\underline{ST'}$ reduces $\underline{ST}$ if both are view-states from the same view, and the former can be evaluated from the later in an information-losing manner.

Definition A state $\underline{ST}_{min}$ is said to be *minimal* if cannot be further reduced. □.

There can exist several view-states isomorphic to a minimal one; once a minimal view-state has been obtained, techniques similar to those of query optimization procedures ought to be applied to select the most efficient among them.

View-states are states in the sense of dynamical systems [ZADE63] [CANA75]: they obey observer and factoring (or *evolution*) equations, and their minimization procedures (to be discussed in the following sections) are based upon applying classes of equivalence of inputs.

We will now introduce a new definition, crucial for the development of view-state reduction algorithms.

Definition Let $\underline{ST'} = \underline{TR}(\underline{ST})$ be a reduction of view-state $\underline{ST}$. Then $OBS(\underline{ST}) = OBS'(\underline{TR}(\underline{ST}))$ and the pair $\underline{TR}$, OBS' is said to be a *split* of OBS □.

The term *split* is used because a relational expression is split from the observer expression, and is reassociated with the new view-state. Generally a valid split may occur if an axiom of distribution holds between the particular relational operator involved in the split and the set union and set minus operations which are used in the $CMP()$ operator.

The use of splits for view-state reduction is illustrated in the following algorithm:

Let $\underline{ST} = \underline{A}$, Let $OBS = Vw$
WHILE (exists an untried $\underline{TR}$ that splits OBS into $OBS'(\underline{TR})$) DO
    IF $\underline{TR}$ $(CMP(\underline{ST(A)}, \Delta\underline{A})$ is isomorphic to $\{\underline{TR}(\underline{ST(A)}, \Delta\underline{A}\}$ THEN
        BEGIN
        $\underline{ST} = \underline{TR}(\underline{S})$;
        $OBS = OBS'$
        END;
$\underline{ST}_{Min} = \underline{ST}$
END{ *Algorithm* }

The following two sections will deal with the implementation of this algorithm using relational algebra and the initial algebra approaches.

3 Reducing view-states in relational algebra expressions

A much simpler version of the algorithm sketched at the end of last session can be applied when the view is equivalent to a Select-Project-Join query and each relation appears at most once in expression. In that case, the query can be parsed as a Projection operator, followed by a Selection, and terminated by a Cartesian product of all involved relations, that is, in the form:

$$\Pi_{(F)}(\sigma_G(A_1 \times \cdots \times A_n))$$

where $\{F\}$ is a set of attributes of relations in $\Re$, and G is a boolean expression written as a conjunctions of terms of the type $x\ op\ constant$ or $x\ op\ y$, where $op \in \{=, >, \geq, <, \leq\}$, and x, y are attributes of $\Re$. Let G_k be that fragment of G that involves terms of the type $x\ op\ constant$, where x is an attribute of A_l. Let $\{F_k\}$ be a set of attributes composed of the union of a key of A_k, the attributes of A_k that appear as join conidtions in G and those that appear in $\{F\}$.

Then, it can be proved that $\Pi_{\{F_1\}}(\sigma_{G_1}(A_1)), \cdots, \Pi_{\{F_n\}}(\sigma_{G_n}(A_n))$ is a view-state for the Project-Select-Join view, and that it can not be further reduced using operators of relational algebra. Fig 4) shows how that view-state is obtained.

The implementation of the reduction algorithm of Section 2 presents severe difficulties in its general case: i) when enumerating the possible transformations $\underline{TR}$ used to generate split candidates and ii) when it intends to prove or disprove factorability for the new candidate split.

To avoid these difficulties we propose a *greedy* algorithm for the reduction of view-states. The results of this algorithm, even if applied in the simplified Project-Select-Join case, are not guaranteed as minimal.

This algorithm has been designed following these premises:

Transformational approach - It modifies the parsing tree to a standard form and, beginning with the base relations, proceeds to search in a systematic bottom-up fashion for *splits* that reduce view-states.

Disjointness - View-states correspond to subtree-patterns in a modified parse tree and a base relation can participate in one of such patterns only. In particular, a given relation can appear in a single subtree-pattern.

Non-backtracking - The (modified) parse tree of the view is covered by a forest of subtrees categorized according to their patterns; categories are marked as either *alive* or *tested*. Tested subtrees correspond to reduced states, while the subtrees belonging to live categories are retained as prospective building-blocks for new view-states.

Split Characterization - The search for splits will be done considering either unary or binary operators. A unary operator *Uop(.)* will be split as an operator pair $\widehat{Uop}(Uop\_s(.))$, where $\widehat{Uop}()$ (the split operator) will be part of the view expression and *Uop_s()* (the reducer) will go to the view-state. A binary operator *Bop(., .)* will be trivially split as an operator pair *Id(Bop_s(., .))*. *Bop_s(., .)* will go to the view-state. Members of a category that cannot generate a split become tested and will not be considered further in the process.

The *greedy* algorithm proposed here has the following steps:

Step 1: **Preprocess the view.** Execute algebraic transformations on the relational operators required by the view. These transformations are similar to those undertaken by query optimization procedures to maximally reduce the operands of each relational operator involved [SMIT75], such as migrating projections and selections towards the base relations. Consecutive selects and projects may be combined, but since there is a pattern matching phase of the algorithm, some predetermined ordering scheme would be employed, e.g., always select before project, multiple parameters of any operator should be sorted by attribute name, etc.

Step 2: **Initialize the list of view-states.** The starting view-state $\underline{ST_0}(A_1, \cdots, A_n)$ is the trivial one, namely $\underline{A}$. This will correspond to leaves of the relational parse tree representing the particular database view Vw. The starting observation expression, OBS_0 is the same as Vw. Mark all leaves as **live** subtrees.

Step 3: **Pattern matching.** Categorize the live subtrees according to their patterns: two subtrees belong to the same category if and only if they are equivalent. Construct two lists, one for the unary ancestors of live subtrees and other for the binary ancestors of them. The lists have the format:

$$Unary(\{Uop_1, cat_{Child\_1}\}, \cdots, \{Uop_s, cat_{Child\_s}\})$$
$$Binary(\{Bop_1, cat_{L\_child\_1}, cat_{R\_child\_1}\}, \cdots, \{Bop_r, cat_{L\_Child\_r}, cat_{R\_Child\_r}\})$$

Step 4: **Discard categories.** Perform the following simplifications:

i) Apply as many times as required on the members $\{Bop_l, cat_i, cat_j\}$ of the binary list:
   a) If cat_i is tested, mark cat_j as tested and vice versa.
   b) If another member exists and has either the form $\{Bop_m, cat_i, cat_k\}$ or $\{Bop_m, cat_k, cat_j\}$ and $cat_j \neq cat_k$, mark categories (cat_i, cat_j, cat_k) as tested.
   c) If Bop_l is asymmetrical, $cat_i \neq cat_j$,, and there exists a member $\{Bop_l, cat_j, cat_i\}$ with $cat_i \neq cat_j$, mark categories (cat_i, cat_j) as tested.

ii) Drop from the unary and binary lists all those items containing a tested subtree.

The previous three simplifications stem from two of the premises on which the greedy algorithm is based: that of disjointedness of the state components and the one that characterizes splits.

Step 5: **Determination of candidates for splits** Two different procedures are used for generating splits, depending on whether a particular subtree is referred to in the Unary operator or the Binary operator list. The two cases are:

Unary operators Extract the sublist of operators $\{ Uop_1^j, \cdots, Uop_k^j \}$ for each category j in the unary list. Obtain a split operator $\widehat{op}_i^j$ common to all operators in the sublist. The problem of finding a common split operator is reciprocal to the one treated in [LARS85]; the issue there was the computation of queries from derived relations, while the goal here is the obtention of a minimal view from which a set of queries can be computed. For example, the split of two projections (Π_B, Π_C) is $\Pi_{B \cup C}$, and the split of two selections (σ_B, σ_C) is $\sigma_{B \cup C}$. As mentioned before, if the split is eventually accepted, $\widehat{op}_i^j$ will pass to the parse tree, and the state will be substituted by $(op\_s^j(operands\ of\ category\ j))$.

Binary operators There will be at most one binary operator $Bop()$ for each category. The split will be the identity operator and the reducer will be $Bop()$ itself.

Step 6: **Testing for factoring** Test each of the candidate splits for factoring. As a first step, mark as tested all splits that have obtained an unfactorable operator as a reducer (e.g., a union, a join, a projection that does not involve a key, etc.). Once that is done, verify the fulfillment of eqns.(2, 3) for each of the remaining ones. It must be noted that factorability is not certain even if the reducer operator admits factoring; for example, both a Cartesian product and a selection admit factoring, but a join, composed of those two operators, does not.

The members of all rejected splits are marked "tested" and erased from the $Unary$ and $Binary$ lists. Those lists are further modified so that the subtrees corresponding the accepted splits replace the instances of the split's components; the observer operator $OBS()$ is modified accordingly.

Step 7: Iterate from step 4 while there is a subtree alive; otherwise the algorithm terminates. Construct $\underline{FCT()}$ from the factoring expressions of step 6 using the tested categories as the components of the view-state $\underline{ST()}$; use the last version of $OBS()$ as an observer.

The algorithm just described is overly simplistic: it can be improved in several ways:

1. by relaxing its non-backtracking condition and allowing branching of decision and the parallel exploration of several alternatives.

2. by eliminating its disjointness condition permitting a base relation to be covered by several view-states. This implies using horizontal and vertical partitions and has all the inherent complications of cover problems.

3. by accepting more general splits for binary operators, e.g., $Bop(.,.) = \widehat{Uop}(Bop\_s(.,,))$, $Bop(.,.) = \widehat{Bop}(Uop\_s_1(.), Uop\_s_2(.))$, etc.

4. by eliminating its strict bottom-up approach and using more general methods for pattern matching.

Some examples of the behavior of this version of the greedy algorithm will now be presented.

Example 7: View-State for View $Names$ (cf. Example 1).
Let us apply the greedy algorithm. Steps 1-2 of the procedure render relation $Names$ as the only expression in the state and set the observer operator as $OBS() = \Pi_{E\_name}()$. Proceeding with the algorithm, no pattern matching is needed in step 3, and in step 4 nothing is discarded. Step 5 generates a split by decomposing $\Pi_{E\_name}()$ to $\Pi_{E\_name}(\Pi_{\{E\_name,E\_SSN\}}())$; the view-state is then $\Pi_{\{E\_name,E\_SSN\}}(Emp)$, while the observer remains unchanged. In step 6 the view candidate is proven factorable. A second iteration through steps 4-6 declares the state-view as tested and the algorithm terminates□.

Fig. 5a shows the initial view-state; 5b the final view-state; 5c how the factoring is done for the set union operator, and fig 5d, how an incremental implementation of the view can be attained.
This particular view cannot be reduced any more if splits are restricted to be generated by expressions in the relational algebra; however, a further reduction can be performed if we resort to languages with more expressive power, such as the one of relational algebra with aggregate functions [KLUG82], [CHAN82]. That would allow us to specify $\Pi_{\{E\_name,Count(E\_SSN)\}}()$ as a view-state (Employee names plus the count of tuples that have that name). This matter will be examined again in Section 4.

Example 8: View-state for $Rich\_Tots$ (cf. Example 1).
In this example the initial states are $S_1 = Child$, $S_2 = Emp$. Applying the greedy algorithm, nothing new happens until step 5, where splits are found that propose a state composed of two expressions (S_1, S_2) given by $(\sigma_{age<6}(Child), \sigma_{Salary>32K}(Emp))$. Both new expressions pass the factorability test of step 6, and on the second round (steps 4-6 again), both are declared tested; the observer is now $OBS(S_1, S_2 = \Pi_{Child\_name}(S_1 \bowtie S_2)$. Fig. 3a and 3b show the initial and the final choices of values; Fig 6a, the implementation of the factoring expression for the set union operator; and fig 6b, how an incremental view can be implemented □.

The factoring equation gives a straightforward method for including the changes in a view; the incremental form of that equation can afterwards be very easily adapted to render an incremental version of the view.
It should be re-emphasized that the reduction of view-states is related to the removal of redundant information only. This, by itself, does not suffice for achieving an efficient implementation; optimization techniques ought to be used complementarily in the search of efficient equivalent implementations.

4 The initial algebra approach

The previous two sections dealt with the problem of obtaining a reduced view-state using an approach based on the algebraic modification of operators to render new equivalent expressions.

An alternative approach, that of initial algebra, [GOGU78] characterizes the view as a catalog of inputs and outputs; the view itself is defined by three components:

i) A set S of the *sorts* (or basic types) that are used in it.

ii) a *signature* Σ with the names, domains and ranges of its operators

iii) a *presentation* ϵ consisting of the set of axioms that characterize the properties of the operators of Σ.

In the following paragraphs Example 7 (which involved a view of a projection operator) is redone using the initial algebra approach.

Example 9 Our specification of the projection operator involves three basic data types, *Relation*, *tuple* and *Boolean*, and is provided with two constructor operators:

CMP(R, t, b), a constructor for relations. If the boolean parameter b is $TRUE$, it adds tuple t to relation R; otherwise, the tuple is removed.
Π_*tuple*( ) performs the projection of a tuple.

and two observer operators:

Π (that projects a relation) and
Ident(...) i.e., $TRUE$ if two tuples are equal.

The sorts S, signature Σ and presentation ϵ of the algebra are:

$$S = \{Relation,\ tuple,\ Boolean\}$$

$$\Sigma = \begin{cases} \Pi : Relation \rightarrow Relation & \text{Projection of Relations} \\ \Pi\_tuple:\ tuple \rightarrow tuple & \text{Projection of tuples} \\ Ident:\ tuple \times tuple \rightarrow Boolean. & \text{Comparison of tuples} \\ CMP:\ Relation \times tuple \times Boolean \rightarrow Relation & \text{Insertion or removal of a tuple} \\ Null : \lambda \rightarrow Relation & \text{Empty Relation} \end{cases}$$

$$\epsilon = \begin{cases} CMP(\ Null\ ,\ tuple_1, FALSE) = Null \\ CMP(CMP(A, tuple_1, TRUE), tuple_1, TRUE) = CMP(A, tuple_1, TRUE) \\ CMP(CMP(A, tuple_1, TRUE), tuple_2, TRUE) = CMP(CMP(A, tuple_2, TRUE), tuple_1, TRUE) \\ CMP(CMP(A, tuple_1, TRUE), tuple_1, FALSE) = A \\ CMP(CMP(A, tuple_1, FALSE), tuple_2, FALSE) = CMP(CMP(A, tuple_2, FALSE), tuple_1, FALSE) \\ \text{IF } \neg\ Ident(t_1, t_2) \text{ THEN} \\ \quad CMP(CMP(A, tuple_1, FALSE), tuple_2, TRUE) = CMP(CMP(A, tuple_2, TRUE), tuple_1, FALSE) \\ \Pi(CMP(A, tuple_1, TRUE)) = (CMP(\Pi(A), \Pi\_tuple_1, TRUE)) \end{cases}$$

Let us consider a sequence $\{CMP(A, tuple_1, b_1), \cdots CMP(A, tuple_n, b_n)\}$ of insertions and deletions applied on an initially empty relation. From the point of view of the projection operator, and by successive application of the equations of the presentation, that sequence can be proved equivalent to another one constructed by the following steps:

Step i) Remove cancelling insert-delete operations, i.e., if a pair i,j exists obeying $i < j$ and $Ident(tuple_i, tuple_j)$ and $b_i = TRUE$ and $b_j = FALSE$, remove $CMP(A, tuple_i, b_i)$, $CMP(A, tuple_j, b_j)$ from the list.

Step ii) Remove all remaining deletes.

Step iii) Remove redundant inserts i.e., if a pair i,j exists obeying $i < j$ and $Ident(tuple_i, tuple_j)$ and $b_i = TRUE$ and $b_j = TRUE$ remove $CMP(A, tuple_i, b_i)$ from the list.

Step iv) Form groups with the remaining inputs. Two inputs i, j belong to the same group if $Ident(\Pi\_tuple(tuple_i), \Pi\_tuple(tuple_j))$ is $TRUE$.

Any $tuple_k$ in one of the aforementioned groups can be substituted, from the point of view of the projection operator, by any $tuple_l$ for which $Ident(\Pi\_tuple(tuple_k, tuple_l)$ is true; no further reductions can be applied. Therefore, instead of storing all the elements of a group, it suffices to store a single element plus a count of the elements in that group.

The first three steps above are equivalent to filtering all deletes and all redundant inserts; consequently, if only valid updates and deletions to the original relations are propagated to the projection, a reduced state is given by the projection of the relation with each tuple of the projection carrying the count of their corresponding tuples in the basic relation.

Ref. [GOGU78] proves that any possible view-state is homomorphic to the previously defined state, and that all maximum reduced states are isomorphic to it, i.e., they contain exactly the same information. □

We can now compare the two approaches to the obtention of view-states:

The initial algebra approach reduces the view-state to a minimum. Although this approach furnishes us with ways to reduce a sequence of inputs to a minimal sequence that is isomorphic to any view-state, it does not readily provide a mapping from that minimal sequence to a suitable implementation of the state, i.e., one that utilizes the base objects of the view and the operators defined on those objects.

The transformational approach of sections 2, 3, on the other hand, provides that suitable implementation. It may fail, however, to reduce the view-state to a minimum.

5 Conclusions

Support for materialized views is an important database capability. Incremental updating is vital to efficient view maintenance, especially in active databases, since the complete re-evaluation of a view may be unnecessary and its cost unacceptable.

This idea is especially critical in complex information systems where a context for inquiry must be established before meaningful results may be obtained. For instance in a GIS/Mapping program, it is only against an established map background view that questions on other map features make sense [FRAN87]. For instance, a request to display all the public boat access ramps in a given area would be virtually useless without some context such as contour lines or roads and hydrological features to assist the user in gaining a sense of scale or position. If the context view must change rapidly, such as should occur if the system permitted real-time pan and zoom capabilities, then efficient update strategies would be essential.

For similar reasons, efficient view maintenance is important in other applications as well, such as for interactive editing based upon attribute grammars [HORW85], and for the maintenance of *alerters* [BUNE79] to report when some database state has been reached.

We have shown, however, that complete incremental updating becomes difficult or impossible when view construction operators lose information. In these cases, an update may require substantial or complete view re-evaluation. To ameliorate the effects of these problems, we have suggested the maintenance of minimal view-states which are the set of the highest-level intermediate view constructions (not necessarily disjoint) which still admit incremental updating. The convenience of applying this concept must be empirically established, since it is dependent upon the activity levels of the particular database.

We have also shown that minimal view-states and the form of the incremental updates can be mechanically derived using relational algebra, and that this concept may be extended to any object-oriented formalism having the appropriate distribution and association axioms.

Current research is being conducted to extend these ideas to views which include aggregation operations and to methods of empirical analysis to assist in determining in which situations the minimal view-state concept would be most practically employed.

References

[BLAK86] J. A. Blakeley, P. Å. Larson, F. W. Tompa, "Efficiently updating Materialized Views" Proc. of ACM SIGMOD Conference, pp 61-71, May 1986.

[BLAK89] J. A. Blakeley, N. Coburn, P. Å. Larson "Updated Derived Relations: Detecting Irrelevant and Autonomously Computable Updates". ACM TODS, vol 14 # 3, Sept 1989, pp 369-400.

[BUNE79] P. O. Buneman, and E. K. Clemons "Efficiently Monitoring Relational Databases" ACM Transactions on Database Systems, Vol. 4, # 3, pp 368-382 (Sept 1979).

[CANA75] R. Canales, R. Barrera *Fundamentals of Control Theory* (Spanish) Limusa-Wiley, México, 1975.

[CASA88] M.A. Casanova, L. Tucherman, A. L. Furtado "Enforcing inclusion dependencies and referential integrity" Proc 14th VLDB confr. L.A. CA. 1988 pp 38-49.

[COHE89] D. Cohen "Compiling complex database transition triggers" Proc. 1989 SIGMOD, pp 225-234.

[CHAN82] A. K. Chandra, D. Harel "Structure and complexity of relational queries" J. Computer and System Sciences, Vol 25, # 1, pp 99-128.

[DAYA82] U. Dayal, P.A. Bernstein "On the correct translation of update operations on relational views" ACM Trans. on Database Systems, Vol 7, # 3, pp 381-416 (Sept. 1982).

[GOGU78] J. Goguen, J. Thatcher, E. Wagner " An Initial algebra approach to the specification, correctness and implementation of abstract data types" In *Current trends in programming methodology IV*, R. Yeh Ed., Prentice Hall, 1978 pp 80-149.

[FRAN87] A. U. Frank, D. L. Hudson, V. B. Robinson "Artificial Intelligence Tools for GIS". Proc. IGIS Conf. Crystal City, VA., Nov. 1987.

[HORW85] S. Horwitz, T. Teitlebaum " Relations and Attributes: A Symbiotic Basis for Editing Environments " ACM-SIGPLAN Symposium on Language Issues in Programming Environments. SIGPLAN Notices, Vol. 20, # 7, July 1985.

[KLUG82] A. Klug "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions". Journal of the ACM, Vol 29, # 3, pp 699-717, Jul. 1982.

[LARS85] P. Å. Larson, H. Z. Yang "Computing queries from derived relations" Proc VLDB 85, pp 259-269.

[ROSE89] A. Rosenthal, U.S. Chakravarthy, B. Blaustein, J. A. Blakeley "Situation Monitoring for Active Databases" Proc 1989 VLDB Conference.

[SMIT75] J. M. Smith, P. Y. Chang "Optimizing the performance of a relational algebra interface" Comm. ACM, Vol 18(10), pp 568-579

[ULLM80] J. D. Ullman *Principles of Database Systems* 1st Ed. Computer Science Press, 1980.

[ZADE63] L. A. Zadeh, C. A. Desoer *Linear System Theory: The Space State Approach* Mc Graw Hill, 1963.

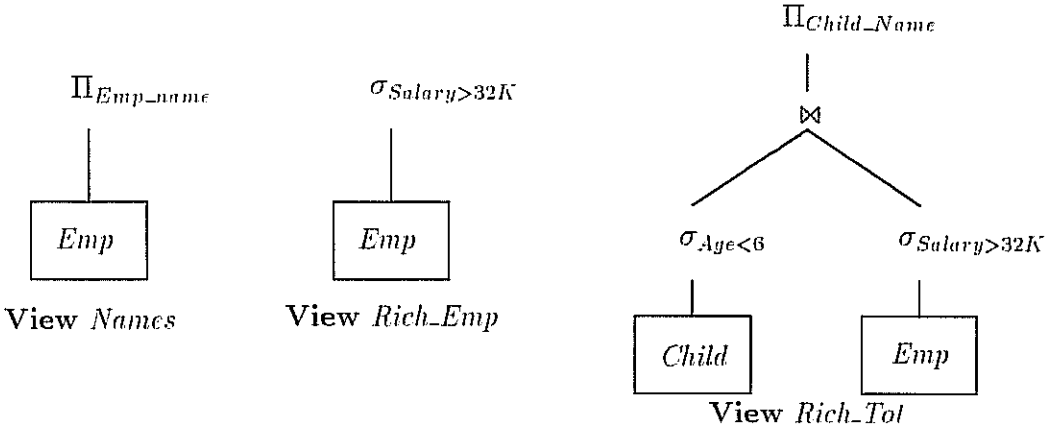


Figure 1: Views used

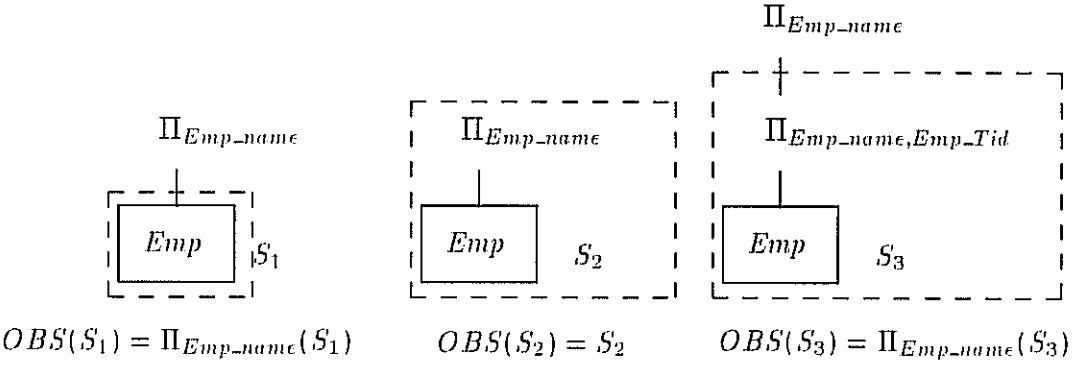
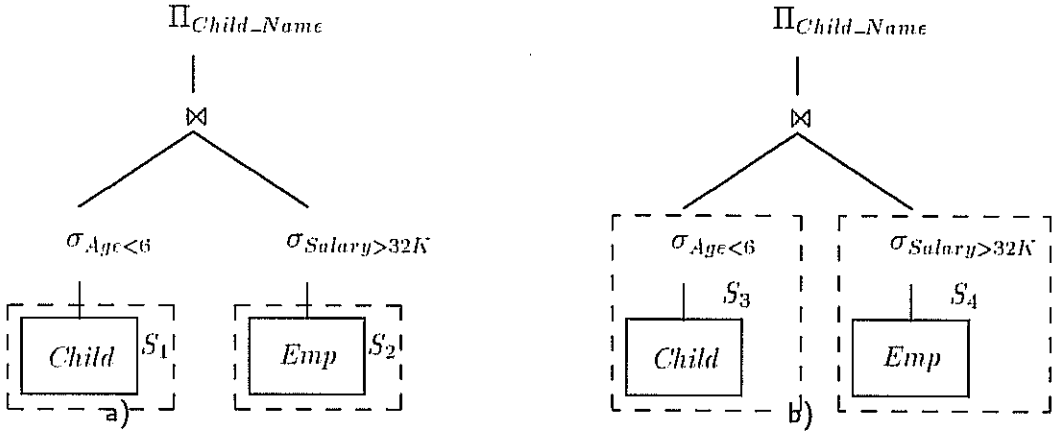


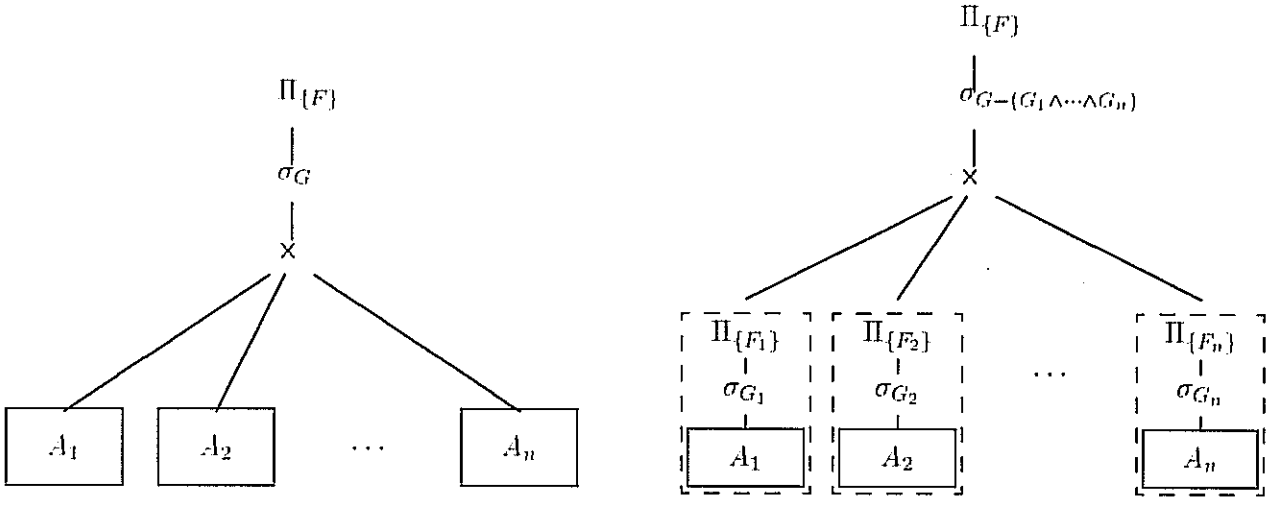
Figure 2: Tentative view-states for Names



$OBS(S_1, S_2) = \Pi((\sigma_{Age<6}(S_1) \bowtie \sigma_{Sal>32K}(S_2))$

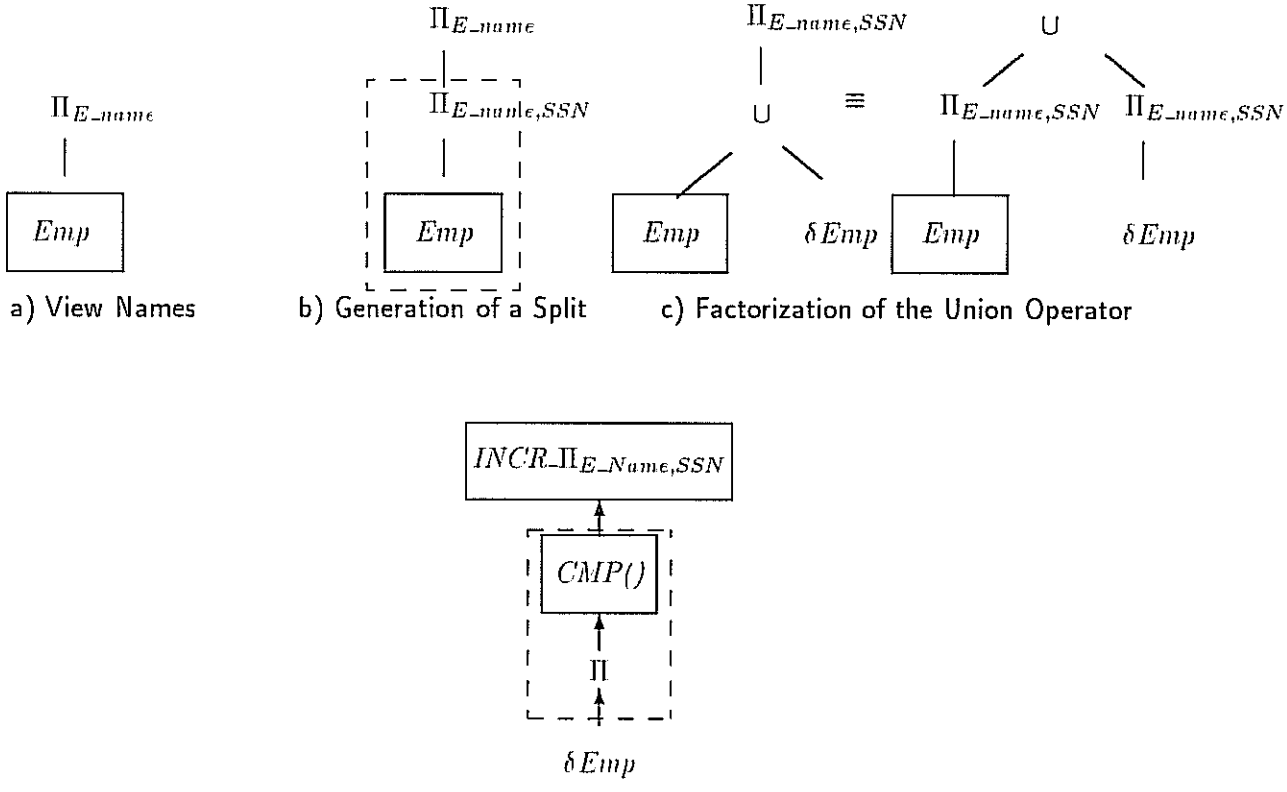
$OBS(S_3, S_4) = \Pi(S_3 \bowtie S_4)$

Figure 3: Tentative view-states for Rich_Tots



a) A PSJ view. b) Its reduced view-state representation

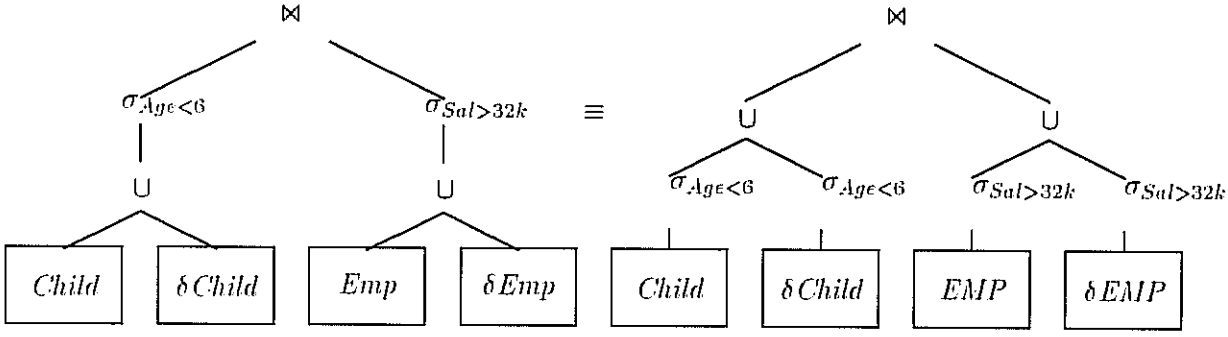
Figure 4: Minimal view-states for a PSJ view.



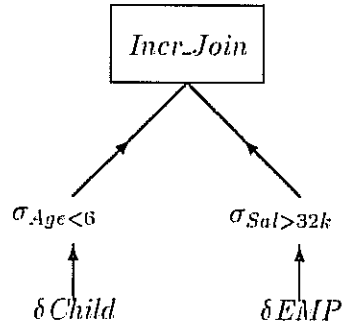
a) View Names b) Generation of a Split c) Factorization of the Union Operator

d) Incremental implementation of view

Figure 5: View_state for view Names



a) Factorization of the Union Operator



b) Incremental implementation of view

Figure 6: View_state for view Rich_Tots