Updating Materialized Database Views

José A. Blakeley

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1

ABSTRACT

Query processing in relational database management systems can be sped up by keeping frequently accessed users' views materialized. However, only if the materialized views are kept up to date with the base relations does this avoid the need to access the base relations in response to queries.

This thesis explores several important subproblems towards the solution of the view maintenance problem, using the assumption that a view is described by selection, projection, and join. The main contributions of this work are:

- the characterization of updates to base relations that cannot affect a materialized view;

  The conditions given for the detection of updates of this type, called *irrelevant updates*, are necessary and sufficient and are independent of the database state.

- the characterization of updates to base relations that can be reflected in a materialized view using only the information provided by the update expression, the view definition, and the contents of the materialized view;

  Updates of this type are called *autonomously computable updates*. Two cases of autonomously computable updates are studied: *unconditional*, or scheme-based, and *conditional*, or instance-based. We present necessary and sufficient conditions for unconditionally autonomously computable insertions, deletions, and modifications, as well as necessary and sufficient conditions for conditionally autonomously computable insertions and deletions.

- an approach to differential update of materialized views.

  The method, based on query modification, uses the state of the base relations and the net changes applied to the base relations since the latest update of the view. Differential re-evaluation leads to a special multiple query optimization problem. This multiple query optimization problem is formally stated and two solutions, one based on query decomposition and the other based on space search methods, are explored.

Finally the thesis explores the support of snapshots by treating them as materialized views that are updated periodically.

Acknowledgements

There are several people who have made possible the successful completion of this work to whom I am indebted. First amongst these is my wife Lucy whom at the expense of putting her own carrer as a public accountant on hold, decided to undertake this adventure in Canada with me. She has been the source of love, encouragement, and support on a daily basis throughout my graduate studies.

I have been extremely fortunate in having Professors Frank Wm. Tompa and Per-Åke Larson as my thesis supervisors. I have benefited a great deal by learning from their technical skills through the many interactions I have had with them. Their encouragement, friendship, support, and prompt feedback at all stages in the development of this work are deeply appreciated.

To the other members of my thesis committee, Professors Stan Burris, Stavros Christodoulakis, Ian Munro, and Abraham Silverschatz for their helpful criticism.

To Neil Coburn for the collaboration we had. In particular, the results in Section 4.2 on unconditionally autonomously computable updates were developed jointly with him. I also want to thank Neil for the useful comments he made to a draft of the thesis.

To my parents José Alfredo and Josefina for their constant love and support. To Mr. Fidencio Ruiz Guajardo and Mrs. Lucinda E. González de Ruiz for their moral support.

To all my friends during my graduate years at Waterloo for the useful discussions at several stages during our studies.

The financial support from the Consejo Nacional de Ciencia y Tecnología de México (CONACYT), the Department of Computer Science at the University of Waterloo, the Natural Sciences and Engineering Research Council of Canada, and Cognos, Inc., Ottawa are gratefully acknowledged.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Database management systems have become a widely used tool for the access and maintenance of large collections of shared data. Systems based on the relational model are rapidly gaining acceptance. The relational model was introduced by Codd [CODD70] and is described in detail in several books [DATE86,MAIER83a,KORTH86,ULLMA82].

In the relational model, the data and the relationships among data are represented by a collection of tables (base relations) each of which has a number of columns (attributes) with unique names. A user interacts with a database management system by submitting requests for data selection, called *queries*, and for data manipulation, called *updates*. Both the queries and the updates are formulated using an appropriate *data manipulation language*.

In a relational database system, a database may be composed of *derived relations* in addition to base relations [MYLOP75,TSICH77]. A derived relation — or *view* — is defined by a relational expression (query) over the base relations. A derived relation may be *virtual*, which corresponds to the traditional concept of a view, or *materialized*, meaning that the relation resulting from evaluating the expression over the current database instance is actually stored. As base relations are modified by update operations, the materialized derived relations may also have to be changed. A materialized derived relation can always be brought up to date by re-evaluating the relational expression defining it, provided that the necessary base relations are available. However, complete re-evaluation of the expression is often wasteful, and the cost involved may be unacceptable. Throughout the thesis, all derived relations are assumed to be materialized, and the terms derived relation and view are used synonymously.

Consider a database scheme $\mathbf{D} = \langle D, S \rangle$ consisting of a set of base relation schemes $D = \{R_1, R_2, \ldots, R_m\}$ and a set of view definitions $S = \{E_1, E_2, \ldots, E_n\}$, where each $E_i \in S$ is a relational algebra expression over some subset of D. Suppose that an update operation $\mathcal{U}$ is posed against the database d on D specifying an update of base relation r_u on $R_u \in D$. To keep the views consistent with the base relations, those views whose definition involve R_u may have to be updated as well. The general *maintenance problem for views* consists of: (1) determining which views may be affected by the update $\mathcal{U}$, and (2) performing the necessary updates to the affected views efficiently.

As a first step towards the solution of this problem, we consider the following important subproblems. Given an update operation $\mathcal{U}$ and a potentially affected view E_i:

- Determine the conditions under which the update $\mathcal{U}$ has no effect on the view E_i, regardless of the database instance. In this case, the update $\mathcal{U}$ is said to be *irrelevant* to E_i.
- If the update $\mathcal{U}$ is not irrelevant to E_i, then determine the conditions under which E_i can be correctly updated using only $\mathcal{U}$ and the instance of E_i, for *every* instance of the database. That is, no additional data from the base relations D is required. In this case, the effect of $\mathcal{U}$ on E_i is said to be *unconditionally autonomously computable.*
- If the effect of the update $\mathcal{U}$ on E_i is not unconditionally autonomously computable, then determine the conditions under which E_i can be correctly updated using only $\mathcal{U}$ and the *current* instance of E_i. In this case, the effect of $\mathcal{U}$ on E_i is said to be *conditionally autonomously computable.*
- If the effect of the update $\mathcal{U}$ on E_i is not conditionally autonomously computable, then determine efficient algorithms to carry out the update of the view.

The maintenance problem for views is important for several reasons.

Structuring the internal level of the database: Larson and Yang proposed a new approach to structuring the database in a relational system at the internal level [LARSO85]. In current relational systems there is a one-to-one correspondence between conceptual relations and stored relations, that is, each conceptual relation exists as a separate stored relation (file). This is a simple and straightforward representation, but its drawback is that the processing of a query often requires data to be collected from several stored relations. Instead of directly storing each conceptual relation, they proposed structuring the stored database as a set of materialized views. The choice of stored relations should be guided by the actual or anticipated query load so that frequently occurring queries can be processed rapidly. To speed up query processing, some data may be redundantly stored in several views.

The structure of the stored database should be completely transparent at the conceptual level. This requires a system capable of automatically transforming any user update against a conceptual relation, into equivalent updates against all stored relations affected. The same type of transformation is necessary to process user queries. That is, any query posed against the conceptual relations must be transformed into an equivalent query against the stored relations. The query transformation problem has been addressed by Larson and Yang [LARSO85,LARSO86]. Preliminary results on the update transformation problem are given by Blakeley, Coburn, and Larson in [BLAK86b].

Enhancing query processing: Materialized views provide an important device to enhance the response time of frequently posed queries in conventional relational systems. When a user knows that he will be using a portion of the database very frequently, he may choose to keep his own view of the data materialized for faster retrieval and reduction of processing and communication costs. From a user's point of view, a materialized view may appear to be always up to date with the base relations. In this case we say that the view is *consistently up-to-date*. There are two ways in which this can be achieved. The first is to update the materialized view immediately after any of the base relations that participate in its definition is updated. The second is to delay the update of the materialized view until the next time the user wants to look at its contents. This type of update is on demand and implies that between any two consecutive accesses of the view no updates are applied to it. Consistently up-to-date views are referred to as *evolving views* by Tsichritzis and Lochovsky [TSICH77].

On the other hand, a materialized view is not necessarily updated immediately when some base relation mentioned in its definition is updated. This type of view may be brought up to date only periodically or upon request from the user. In this case we say that the view is *periodically updated.* Views in the latter category are usually called *snapshots* [MYLOP75,ADIBA80,BLAK86a,LINDS86]. Notice that the support of consistently up to date views on demand as well as the support of snapshots are technically the same problem. The results presented in this thesis apply to the support of either consistently or periodically updated materialized views.

Distributed database environments: The detection of irrelevant or autonomously computable updates also has applications in distributed databases. Suppose that a view is stored at some site and that an update request, possibly affecting the view, is submitted at the same site. If the effect of an update is autonomously computable, then the view can be correctly updated locally without requiring data from remote sites. If the request is submitted at a remote site, then we need to send only the update request itself to the site of the view. If an update is irrelevant to a view, then no action to update that view is required.

All other cases will require sending some amount of data to the site housing the view to bring it up to date. The straightforward way of updating a view is to re-evaluate the expression defining the view at the site where the base relations are stored and then send the result to the remote site. At the remote site, the process of updating the view would consist of deleting the current contents of the view and inserting the new state of the view. However, the communication costs required by such solution may be very high because every time we want to update a view we need to send its whole new state to the remote site. In this thesis we are interested in finding more efficient ways of bringing the view up to date. In particular, we propose a differential method to compute the changes necessary to bring the view up to date as a result of the latest updates to the base relations, thus reducing the amount of data sent between sites.

In addition to the motivations presented above, the maintenance problem for views also has applications in the area of integrity enforcement and trigger support. Static integrity constraints can be enforced through mechanisms that depend on defining a view. If we can show that an update operation has no effect on the view associated with an alerter or integrity constraint, then the update cannot possibly trigger the alerter or result in a database instance violating the integrity constraint. The results presented in this thesis have direct applications in this area as well.

It must be stressed that the problem analyzed in this thesis is completely different from the problem of *updates through views*. In that problem, a user is allowed to pose updates directly to a view, and the difficulty is in determining how to translate updates expressed against a view into updates to the base relations. In the model proposed in this thesis, the user can only update base relations; direct updates to views are not considered. Therefore, rather than analyzing the traditional problem of deriving appropriate update translations, this thesis is concerned with finding efficient ways of keeping materialized views up to date with the base relations. The reader interested in the problem of updates through views may refer to work by Keller [KELLE86] or Medeiros and Tompa [MEDEI86].

1.1 Previous work

The idea of storing derived relations to help improve the response time of the database is not new [SCHMI75]. Schkolnick and Sorenson [SCHOL81] proposed the materialization of join relations as a way to improve the performance of frequently posed queries. The process of transforming a (fourth) normal form database scheme to one containing join relations is called *denormalization*. They suggested the idea of denormalizing base relations at the storage level to reduce the need for joining base relations, yet giving the user the impression that the normalized base relations are actually stored. One of the problems in a denormalized database is to be able to transform queries posed against the normalized base relations to queries involving the joined relations. Schkolnick and Sorenson addressed the query transformation problem for the special case in which the denormalized relations are join relations. Larson and Yang [LARSO85] have addressed the query transformation problem for the more general case in which the stored database consists of relations defined by relational expressions involving selections, projections, and joins. The problem of keeping the materialized derived relations up to date with the base relations is not addressed in either of these two papers.

A *view index*, also called *links* and *selectors* [SCHMI75,TSICH77], is a special case of a materialized view which instead of containing full tuples, contains pointers to the tuples in the base relations that contribute to the tuples in the view. A view index is basically an indirect version of a materialized view.

Schmid and Bernstein [SCHMI75] studied alternative data structures for supporting links and selectors. However, they did not present details on how to keep these indices up to date in the presence of updates to the base relations.

Roussopoulos [ROUSS82] presents an algorithm for selecting a subset of views, which when indexed, minimizes the total cost of answering a given set of queries as well as the cost of maintaining the view indices during updates to base relations, subject to some maximum amount of storage available for indexing. The cost of updating the view index after an update to a base relation is estimated to be the cost of computing the whole view index from scratch by re-evaluating the expression defining the view. The results in this thesis can also be applied to update view indices more efficiently.

Work directly related to the maintenance of materialized views has been reported by Koenig and Paige [KOENI81] and by Shmueli and Itai [SHMUE84]. Koenig and Paige investigated the support of derived data (views) in the context of a functional binary-association data model. This data model puts together ideas borrowed from binary-association models, functional models, and the entity-relationship model, within a programming language suitable for data definition and manipulation. In their model, views can be explicitly stored and then maintained. For each possible change to the operands of the view, there exists a procedure associated with this change that incrementally updates the view. This procedure is called the *derivative* of the view definition with respect to the change. Their approach relies on the availability of such derivatives for various view definition/change statement combinations. This thesis, and particularly the examination of differential re-evaluation of views, can be interpreted as forming a theoretical basis for deriving such derivatives automatically.

Shmueli and Itai's approach consists of continuously maintaining an acyclic database, together with information that may be useful for future insertions and deletions. Their definition of views is limited to the projection of a set of attributes over the natural join of all the relations in the database scheme. Thus selection conditions are omitted. Furthermore, Shmueli and Itai consider the maintenance of only one materialized view.

The need for an efficient mechanism to update materialized views has been expressed before by several authors. Gardarin et al. [GARDA84] considered *concrete views* (i.e., materialized views) as a candidate approach for the support of real time queries. However, they discarded the approach because of the lack of an efficient algorithm to keep the concrete views up to date.

Horwitz and Teitelbaum [HORWI85] proposed a model for the generation of language-based environments which uses a relational database along with attribute grammars, and they suggest algorithms for incrementally updating views, motivated by the efficiency requirements of interactive editing. Buneman and Clemons [BUNEM79] propose virtual views for the support of *alerters*, which monitor a database and report to some user or application whether a state of the database, described by the view definition, has been reached.

In other work, Lindsay et al. [LINDS86] present a differential algorithm for maintaining snapshots defined by relational algebra expressions involving only the operators select and project. More details of their work will be given in Chapter 7.

In summary, the need for the support of materialized views has been reported by several authors in the literature. However, the problem of maintaining such views has as yet received little attention. We address the main issues in the support of updates to materialized views as well as laying the groundwork for future research in the area.

1.2 Thesis outline

In Chapter 2 we provide the reader with the basic notation and concepts used throughout the thesis. Chapter 3 deals with the first aspect of updating materialized views, namely, the detection of irrelevant updates. By being able to detect this type of update, the system may obtain substantial cost savings in the maintenance of materialized views. The chapter presents necessary and sufficient conditions for the detection of irrelevant updates as well as an algorithm to perform such a test.

When an update applied to a base relation is not irrelevant to a view, its effect on the view may be autonomously computable. In Chapter 4 we present necessary and sufficient conditions for the detection of autonomously computable updates as well as showing how to carry out the update based on the update expression and the contents of the view. We consider two cases called unconditionally autonomously computable updates and conditionally autonomously computable updates. Unconditionally autonomously computable updates are those where the update can be carried out for every possible database instance. Conditionally autonomously computable updates are those that can be carried out for the current database instance but not for all possible database instances. By determining whether the effect of an update to a base relation on a materialized view is autonomously computable, great reductions in communication and in processing costs may be obtained. Communication costs are reduced in environments where the materialized view is stored at a site which is different from the site housing the base relations. In such environments, the only information required to be sent to the site containing the materialized view is the update expression itself. Processing costs are reduced in environments where the internal scheme of the system contains derived relations and the user sees the data as represented by the conceptual scheme. In such an environment the user poses updates to relations as represented in the conceptual scheme. To update the derived relations appropriately, the system must be able to translate the update posed to the base relation to an update on the derived relation. This in general may require the reconstruction of some base relations from the derived relations, the application of the update to the base relation, and finally the recomputation of the derived relation. If we know that the update is autonomously computable, then the first two stages are saved, and the system can update the derived relation immediately based on its contents and on the update expression.

If an update to a base relation is neither irrelevant nor autonomously computable, then we could proceed to re-evaluate the expression defining the view from scratch in order to bring it up to date. However, a complete re-evaluation is not always the best solution. In Chapter 5 we present a differential re-evaluation approach to updating materialized views. The idea is to manipulate the expression defining the view and to decompose it into a set of similar expressions operating on small relations in such a way that the individual re-evaluation of each of those expressions is substantially cheaper than complete re-evaluation. The approach provides a procedure to compute the set of changes that have to be applied to the view to bring it up to date. This problem, in turn, introduces an interesting multiple query optimization problem. In Chapter 6 we give some suggestions for the efficient execution of the set of expressions obtained by the differential re-evaluation approach.

The algorithms discussed in Chapters 3 through 6 readily apply to the support of consistently updated materialized views. In supporting periodically updated materialized views, other aspects have to be considered, the main one being, how to keep track of the changes applied to base relations between two consecutive requests to update a materialized view. Chapter 7 presents a thorough discussion of the different alternatives.

Finally, Chapter 8 contains a summary of the main contributions of this thesis along with a discussion of some important problems for future research.

Chapter 2

Notation and Basic Assumptions

In this chapter we present some of the notation and assumptions used throughout the thesis. Additional notation is introduced in the appropriate places when needed. Many of the results in subsequent chapters rely on having an algorithm for testing the satisfiability of Boolean expressions. Such an algorithm, for a certain subclass of Boolean expressions, is also presented in this chapter.

A *database scheme* $\mathbf{D} = \langle D, S \rangle$ consists of a set of (*base*) *relation schemes* $D = \{R_1, R_2, \ldots, R_m\}$, and a set of *view definitions* $S = \{E_1, E_2, \ldots, E_n\}$, where each $E_i \in S$ is a relational algebra expression over some subset of D. A *database instance* d, consists of a set of relation instances $r_1, r_2, \ldots, r_m$, one for each $R_i \in D$. We require no constraints (e.g., keys or functional dependencies) to be imposed on the relation instances allowed. A *view materialization* $v(E_i, d)$ is a relation instance resulting from the evaluation of the relational algebra expression E_i against the instance d. In this thesis, we consider only relational algebra expressions formed from the combination of projections, selections, and joins, called *PSJ-expressions.*

It is well known that every *PSJ*-expression can be transformed into an equivalent expression in a standard form consisting of a Cartesian product, followed by a selection, followed by a projection. It is easy to see this by considering the operator tree corresponding to a *PSJ*-expression. The standard form is obtained by first pushing all projections to the root of the tree and thereafter all selection and join conditions. From this it follows that any *PSJ*-expression can be written in the form $E = \pi_{\mathbf{A}}(\sigma_C(R_{i_1} \times R_{i_2} \times \cdots \times R_{i_k}))$, where $R_{i_1}, R_{i_2}, \ldots, R_{i_k}$ are relation schemes, C is a selection condition, and $\mathbf{A} = \{A_1, A_2, \ldots, A_k\}$ are the attributes of the projection. We can therefore represent any *PSJ*-expression by a triple $E = (\mathbf{A}, \mathbf{R}, C)$, where $\mathbf{A} = \{A_1, A_2, \ldots, A_k\}$ is called the *attribute set*, $\mathbf{R} = \{R_{i_1}, R_{i_2}, \ldots, R_{i_k}\}$ is the *relation set* or *base*, and C is a *selection condition* comprising the conditions of all the select and join operations of the relational algebra expression defining E. The attributes in $\mathbf{A}$ will often be referred to as the *visible* attributes of the view. We also use the notation:

$\alpha(C)$ the set of all attributes appearing in condition C

$\alpha(R)$ the set of all attributes of relation R

$\alpha(\mathbf{R})$ the set of all attributes mentioned in the set of relation schemes $\mathbf{R}$, that is, $\bigcup_{R_i \in \mathbf{R}} \alpha(R_i)$.

The following example illustrates the above notation.

Example 2.1 Consider three relation schemes $R_1(H, I)$, $R_2(J, K, L)$, $R_3(M, N)$, and a view defined by the expression

$$E = \pi_{HLN}(\sigma_{H>20}(R_1) \bowtie_{I=J} \pi_{JL}(\sigma_{J=30}(R_2)) \bowtie_{L>M} \sigma_{N<10}(R_3)).$$

This relational algebra expression can be converted into the equivalent expression

$$E' = \pi_{HLN}(\sigma_{(H>20)(I=J)(J=30)(L>M)(N<10)}(R_1 \times R_2 \times R_3)),$$

which in turn can be represented using the triple notation as

$$E' = (\{H, L, N\}, \{R_1, R_2, R_3\}, (H > 20)(I = J)(J = 30)(L > M) \\ (N < 10)),$$

In this example, $\alpha(C) = \{H, I, J, L, M, N\}$, $\alpha(R_1) = \{H, I\}$, $\alpha(R_2) = \{J, K, L\}$, $\alpha(R_3) = \{M, N\}$, and $\alpha(\mathbf{R}) = \{H, I, J, K, L, M, N\}$. □

We assume that each relation in $\mathbf{R}$ is referenced only once in the associated *PSJ*-expression, that is, the expression contains no self-joins. The reason for this assumption is that we need to identify the relation to which an attribute used in the selection condition of a view definition belongs. This assumption can easily be removed by extending the relational algebra expressions allowed in a view definition with a *renaming* operator and hence to support *RPSJ*-expressions (rename-project-select-join expressions). The renaming operator maps each attribute name used in multiple references of the same relation into a distinct attribute name. A *RPSJ*-expression can thus be represented by a four-tuple $E = (\mathbf{N}, \mathbf{A}, \mathbf{R}, C)$, where $\mathbf{A}$, $\mathbf{R}$, and C have the same meaning as before, and $\mathbf{N} = \{S_1 \leftarrow R_1, \ldots, S_n \leftarrow R_m\}$, $n \geq m$, is a mapping that allows distinct names S_i to be used throughout the remainder of the expression and specifies which relation R_j is to be accessed at run time.

Example 2.2 Consider the relation scheme *employee*(*name*, *salary*, *manager*), and a view containing "all employees who are managers." The view can be defined using the following *RPSJ*-expression:

$$E = (\{S_1 \leftarrow \textit{employee}, S_2 \leftarrow \textit{employee}\}, \{S_1.\textit{name}\}, \{S_1, S_2\}, (\{S_1.\textit{name} = S_2.\textit{manager}\}).$$

□

For simplicity we assume from now on that a relation is referenced only once in a relational expression defining a view, that is, we restrict ourselves to views defined by *PSJ*-expressions rather than *RPSJ*-expressions.

The update operations considered are insertions, deletions, and modifications. In the same way as in relational languages such as SQL [CHAMB76] and QUEL [STONE76], each update operation affects only one base relation. The following notation will be used to describe the update operations:

INSERT (R_u, T): Insert into relation r_u the set of tuples T, where each tuple $t \in T$ is defined over R_u.

DELETE (R_u, C_D): Delete from relation r_u all tuples satisfying condition C_D, where C_D is a selection condition over $\alpha(R_u)$. Notice that by allowing this form of delete operation we automatically cover the form where a set of tuples to be deleted is explicitly presented. That is, the operation DELETE $(R_u, \{t\})$ transforms into the operation DELETE $(R_u, \bigwedge_{A \in R_u}(A = t[A]))$.

MODIFY $(R_u, C_M, \mathbf{F}_M)$: Modify all tuples in r_u that satisfy the condition C_M, where C_M is a selection condition over $\alpha(R_u)$. $\mathbf{F}_M$ is a set of expressions, each expression specifying how an attribute value of r_u is to be modified.

Note that both the attributes modified and the attributes from which the new values are computed are from relation R_u. The set of expressions $\mathbf{F}_M$ of a MODIFY operation is assumed to contain an update expression for each attribute in R_u. We restrict the update expressions in $\mathbf{F}_M$ to unconditional functions that can be computed "tuple-wise." Unconditional means that the expression does not include any further conditions (i.e., all conditions are in C_M). Tuple-wise means that, for any tuple in r_u selected for modification, the value of the expression can be computed from the values of the attributes of that tuple alone. The type of expressions we have in mind are simple, for example, $H := H + 5$ or $I := 5$. Further details are given in Chapter 4.

Several update operations as discussed above may be grouped together within a *transaction*. A transaction is an indivisible sequence of update operations. Indivisible means that either all update operations are successfully performed or none are performed. Further, updates within a transaction may update several base relations. The use of transactions is explored more explicitly in Chapters 5 and 7.

All attribute names in the relations are taken to be unique. We also assume that all attributes have discrete and finite domains. Any such domain can be mapped onto an interval of integers, and therefore we will treat all attributes as being defined over some interval of integers. For Boolean expressions, the logical connectives will be denoted by "$\vee$" for OR, *juxtaposition* or "$\wedge$" for AND, "$\neg$" for NOT, "$\Rightarrow$" for implication, and "$\Leftrightarrow$" for equivalence. To indicate that all variables of a condition C, are universally quantified, we write $\forall$ (C) and similarly for existential quantification. If we need to identify explicitly which variables are quantified, we write $\forall$ X (C), where X is a set of variables.

An *evaluation* of a condition is obtained by replacing all the variable names (attribute names) by values from the corresponding domains. The result is either *true* or *false*. A *partial evaluation* (or *substitution*) of a condition is obtained by replacing some of its variables by values from the corresponding domains. Let C be a condition and t a tuple over some set of attributes. The partial evaluation of C with respect to t involves the replacement of variables in C by values for the corresponding attributes in t and is denoted by $C[t[Y]]$, where Y represents the set of attributes replaced. When Y consists of all attributes that C and t have in common, we denote the partial substitution of C with respect to t by simply $C[t]$. The result of a partial substitution is a new condition with fewer variables.

Detecting whether an update operation is irrelevant or autonomously computable involves testing whether or not certain Boolean expressions are valid (i.e., tautologies), or equivalently, whether or not their complements are unsatisfiable.

Definition 2.1 Let $C(x_1, x_2, \ldots, x_n)$ be a Boolean expression over variables $x_1, x_2, \ldots, x_n$. C is *valid* if $\forall$ $x_1, x_2, \ldots, x_n$ $C(x_1, x_2, \ldots, x_n)$ is *true*, and C is *unsatisfiable* if $\nexists$ $x_1, x_2, \ldots, x_n$ $C(x_1, x_2, \ldots, x_n)$ is *true*, where each variable x_i ranges over its associated domain. □

A Boolean expression is valid if it always evaluates to *true*, unsatisfiable if it never evaluates to *true*, and satisfiable if it evaluates to *true* for some values of its variables. Proving the validity of a Boolean expression is equivalent to disproving the satisfiability of its complement. Proving the satisfiability of Boolean expressions is, in general, *NP*-complete [COOK71]. However, for a restricted subclass of Boolean expressions, polynomial algorithms exist. Rosenkrantz and Hunt [ROSEN80] developed such an algorithm for conjunctive Boolean expressions. Each expression β must be of the form $\beta = \beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_k$,where each β_i is an atomic condition of the form $(x\ \theta\ y + c)$ or $(x\ \theta\ c)$, where $\theta \in \{=, <, \leq, >, \geq\}$, x and y are variables, and c is a (positive or negative) integer. Each variable is assumed to range over the integers. The improved efficiency arises from not allowing θ to be the operator $\neq$.

Deciding whether a conjunctive expression in the subclass described above is satisfiable can be done in time $O(n^3)$, where n is the number of distinct variables in β. The sketch of the algorithm is as follows: (1) the conjunctive expression is normalized, that is, it is transformed into an equivalent one where only the operators $\leq$ or $\geq$ are used in the atomic formulae; (2) a directed weighted graph is constructed to represent the normalized expression; and (3) if the directed graph contains a cycle for which the sum of its weights is negative then the expression is unsatisfiable, otherwise it is satisfiable. To find whether a directed weighted graph contains a negative cycle Floyd's algorithm [FLOYD62] can be used, which finds all the shortest paths between any two nodes in a directed weighted graph. A complete example illustrating the use of this algorithm in the context of detecting irrelevant updates is given at the end of Chapter 3.

We can also efficiently decide the satisfiability of Boolean expressions of the form

$$C = C_1 \vee C_2 \vee \cdots \vee C_m$$

where each $C_i = \beta_{i_1} \wedge \beta_{i_2} \wedge \cdots \wedge \beta_{k_i}$, $i = 1, \ldots, m$, is a conjunctive expression in the subclass described above. The expression C is satisfiable if and only if at least one of the conjunctive expressions C_i is satisfiable. Similarly, C is unsatisfiable if and only if each of the conjunctive expressions C_i is unsatisfiable. We can apply Rosenkrantz and Hunt's algorithm to each of the conjunctive expressions C_i; this takes time $O(mn^3)$ in the worst case, where n is the maximum number of different variables mentioned in a single disjunct in C.

In this thesis, we are interested in the case when each variable ranges over a finite *interval* of integers. The definitions of valid, satisfiable, and unsatisfiable are thus modified to restrict the ranges of variables to predetermined domains. Henceforth all universally and existentially quantified variables are implicitly range-restricted. For this case, Larson and Yang [LARSO85] developed an algorithm whose running time is $O(n^2)$. However, it does not handle expressions of the form $(x \; \theta \; y+c)$ where $c \neq 0$. We have developed a modified version of the algorithm by Rosenkrantz and Hunt for the case when each variable ranges over a finite interval of integers. The full details of the modified algorithm are given in [BLAK86b] and are not reported here.

An expression not in conjunctive form can be handled by first converting it into disjunctive normal form and then testing each disjunct separately. Several of the theorems in Chapters 3 and 4 will require testing the validity of expressions of the form $C_1 \Rightarrow C_2$. The implication can be eliminated by converting to the form $(\neg C_1) \vee C_2$. Similarly, expressions of the form $C_1 \Leftrightarrow C_2$ can be converted to $(C_1 \wedge C_2) \vee (\neg C_1 \wedge \neg C_2)$.

As an aside we note that the *NP*-completeness of the satisfiability problem is caused by the fact that converting an expression into disjunctive normal form may, in the worst case, lead to exponential growth in the length of the expression.

Chapter 3

Irrelevant Updates

In certain cases, an update operation applied to a relation has no effect on the state of a view. When this occurs independently of the database state, we call the update operation *irrelevant* to the view. It is important to provide an efficient mechanism for detecting irrelevant updates so that re-evaluation of the relational expression defining a view can be avoided or, at least, the number of tuples considered in the re-evaluation can be reduced.

This chapter presents necessary and sufficient conditions for the detection of irrelevant updates. The conditions are given for insert, delete, and modify operations as introduced in the previous chapter. First we define what it means for an update to be irrelevant.

Definition 3.1 Let d be an instance on the set of relation schemes D, and let d' be the resulting instance after applying the update operation $\mathcal{U}$ to d. Let $E = (\mathbf{A}, \mathbf{R}, C)$ be a view definition. The update operation $\mathcal{U}$ is *irrelevant* to $v(E, d)$ if $v(E, d') = v(E, d)$ for all instances d. $\square$

If the update operation $\mathcal{U}$ does not modify any of the relations over which the view is defined then, obviously, $\mathcal{U}$ cannot have any effect on the view. In this case $\mathcal{U}$ is said to be *trivially irrelevant* to the view.

3.1 Irrelevant insertions

An insert operation is irrelevant to a view if none of the new tuples will be visible in the view. Theorem 3.1 covers the form of insert operation introduced in Chapter 2.

Theorem 3.1 *The operation INSERT(R_u, T) is irrelevant to the view defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, if and only if $C[t]$ is unsatisfiable for every tuple $t \in T$.*

Proof: (Sufficiency) Consider an arbitrary tuple $t \in T$. If $C[t]$ is unsatisfiable, then $C[t]$ will evaluate to *false* regardless of the assignment of values to the variables remaining in $C[t]$. Therefore, there cannot exist any tuple defined over the Cartesian product of the relations in $\mathbf{R} - \{R_u\}$ that would combine with t to satisfy C and hence cause an insertion into $v(E, d)$.

(Necessity) Consider a tuple $t \in T$, and assume that $C[t]$ is satisfiable. $C[t]$ being satisfiable means that there exists a tuple s defined over the base $\mathbf{R}$, denoted by $s(\mathbf{R})$, such that $s[A] = t[A]$ for every attribute $A \in R_u$, $s[A] = \mu_A$ for every attribute $A \notin \alpha(R_u) \cup \alpha(C)$, where μ_A is the lowest value in in the domain of attribute A, and the rest of the values $s[A]$, $A \in \alpha(C) - R_u$ are assigned in such a way that $C[s] = true$. The fact that $C[t]$ is satisfiable guarantees the existence of values for attributes in $\alpha(C) - R_u$.

We can then construct a database instance d using s, such that the insertion of t into r_u will cause a new tuple to be inserted into the view $v(E, d)$.

To construct d, we build a relation instance r_i for each relation scheme $R_i \in \mathbf{R} - \{R_u\}$. Each relation r_i contains a single tuple t_i, where $t_i[R_i] = s[R_i]$. The database instance d consists of the relation $r_u = \emptyset$ and relations $r_i = \{t_i\}$ for each $R_i \in \mathbf{R} - \{R_u\}$. Clearly, $v(E, d) = \emptyset$. However, if we obtain d' from d by inserting tuple t into r_u, then $v(E, d')$ will contain one tuple. Therefore, the INSERT operation is not irrelevant to the view. □

Example 3.1 Consider two relation schemes $R_1(H, I, J)$, $R_2(K, L)$, and the following view and insert operation:

$$E = (\{H, K, L\}, \{R_1, R_2\}, (H > 23)(J = K)(L < 10))$$

$$\text{INSERT } (R_1, \{(15, 20, 35), (26, 20, 45)\}).$$

To predict the effect that the insertion is going to have on the view, we need to analyze each of the tuples being inserted. For the tuple $t_1 = (15, 20, 35)$, $C[t_1] \equiv (15 > 23)(35 = J)(L < 10) = false$. Since $C[t_1]$ is unsatisfiable, then inserting tuple t_1 into r_1 will not affect the view regardless of the contents of the database. For the tuple $t_2 = (26, 20, 45)$, $C[t_2] \equiv (26 > 23)(45 = J)(L < 10)$ is satisfiable. Therefore inserting tuple t_2 into r_1 is *not* irrelevant because it *might* affect the view, depending on the current contents of the database. □

3.2 Irrelevant deletions

A delete operation is irrelevant to a view if none of the tuples in the view will be deleted. Theorem 3.2 covers the form of delete operation introduced in Chapter 2

Theorem 3.2 *The operation DELETE (R_u, C_D) is irrelevant to the view defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, if and only if the condition $C_D \wedge C$ is unsatisfiable.*

Proof: (Sufficiency) If $C_D \wedge C$ is unsatisfiable, then no tuple t defined on $\mathbf{R}$ can have values such that $C_D[t]$ and $C[t]$ are simultaneously *true*. Assume that t contains values such that $C_D[t]$ is *true*, meaning that the delete operation causes the deletion of the tuple $t[R_u]$ from r_u. Since t cannot at the same time satisfy C, then t could not have contributed to a tuple in the view. Thus the deletion of $t[R_u]$ from r_u will not cause any data to be deleted from the view defined by E. Therefore, the delete operation is irrelevant.

(Necessity) Assume that $C_D \wedge C$ is satisfiable. Let $\alpha(C) \cup \alpha(C_D) = \{x_1, x_2, \ldots, x_l\}$. Because $C_D \wedge C$ is satisfiable, there exists a value combination $x = (x_1^0, x_2^0, \ldots, x_l^0)$ such that $C[x] \wedge C_D[x]$ is *true*. We can then construct an instance of each relation in $\mathbf{R}$ such that deleting one tuple from r_u, $R_u \in \mathbf{R}$, will indeed change the view. Each instance r_j, $R_j \in \mathbf{R}$, contains one tuple t_j as follows:

- if R_j contains attribute x_k, $1 \leq k \leq l$, then $t_j[x_k] = x_k^0$.
- if R_j contains an attribute y, $y \notin \{x_1, x_2, \ldots, x_l\}$, then $t_j[y] = \mu_y$, where the value μ_y is any value in the domain of y, say the lowest value in the domain.

Initially the database instance d contains the relation instances $r_i = \{t_i\}$, $R_i \in \mathbf{R}$. Hence, $v(E, d)$ will contain one tuple. Applying the delete operation to d then gives an instance d' where the tuple t_u from relation r_u has been deleted. Clearly, $v(E, d') = \emptyset$. This proves that the deletion is not irrelevant. □

Example 3.2 Consider two relation schemes $R_1(H, I, J)$, $R_2(K, L)$, and the following view definition and delete operation:

$$E = (\{H, L\}, \{R_1, R_2\}, (I > J)(J = K)(K > 10))$$

$$\text{DELETE } (R_1, (I < 5)).$$

To show that the deletion is irrelevant to the view we must prove that the following condition holds:

$$\forall\, I, J, K\ \neg[(I > J)(J = K)(K > 10)(I < 5)].$$

Clearly, the condition can never be satisfied and therefore the delete operation is irrelevant to the view. □

Corollary 3.3 *The operation DELETE $(R_u, \{t\})$ is irrelevant to the view defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, if and only if $C[t]$ is unsatisfiable.*

Proof: The operation DELETE $(R_u, \{t\})$ can be expressed in terms of the delete operation stated by Theorem 3.2 as DELETE $(R_u, \{R_u\}, \bigwedge_{A \in R_u}(A = t[A]))$.

Theorem 3.2 establishes that in order for a delete operation to be irrelevant, the condition $C_D \wedge C$ must be unsatisfiable. For the operation DELETE $(R_u, \{t\})$, this is equivalent to testing

$$\mathcal{C} \bigwedge_{A \in R_u} (A = t[A])$$

which is equivalent to $\mathcal{C}[t]$. The corollary then immediately follows from Theorem 3.2. □

3.3 Irrelevant Modifications

The detection of irrelevant modifications is somewhat more complicated than insertions or deletions. The form of modify operation considered in this section is as introduced in Chapter 2.

Consider a tuple that is to be modified. It will not affect the view if one of the following conditions applies:

- it does not qualify for the view, neither before nor after the modification;
- it does qualify for the view both before and after the modification, but all the attributes visible in the view remain unchanged.

Theorem 3.4 introduced in this section covers the two cases mentioned above, but before we state the theorem, we need some additional notation.

Consider a modify operation MODIFY $(R_u, \mathcal{C}_M, \mathbf{F}_M)$ and a view defined by $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$. Let $\alpha(R_u) = \{A_1, A_2, \ldots, A_l\}$. For simplicity we will associate an update expression with every attribute in R_u, that is, $\mathbf{F}_M = \{f_{A_1}, f_{A_2}, \ldots, f_{A_l}\}$ where each update expression is of the form $f_{A_i} \equiv (A_i := \langle\text{arithmetic expression}\rangle)$. If an attribute A_i is not to be modified, we associate with it a *trivial update expression* of the form $f_{A_i} \equiv (A_i := A_i)$. If the attribute is assigned a fixed value c, then the corresponding update expression is $f_{A_i} \equiv (A_i := c)$. The notation $\rho(f_{A_i})$ will be used to denote the right hand side of the update expression f_{A_i}, that is, the expression after the assignment operator. The notation $\alpha(\rho(f_{A_i}))$ denotes the variables mentioned in $\rho(f_{A_i})$. For example, if $f_{A_i} \equiv (A_i := A_j + c)$ then $\rho(f_{A_i}) = A_j + c$ and $\alpha(\rho(f_{A_i})) = \{A_j\}$.

By substituting every occurrence of an attribute A_i in $\mathcal{C}$ by $\rho(f_{A_i})$ a new condition is obtained. We will use the notation $\mathcal{C}(\mathbf{F}_M)$ to denote the condition obtained by performing this substitution for every variable $A_i \in \alpha(R_u) \cap \alpha(\mathcal{C})$.

An update expression $\rho(f_{A_i})$ may produce a value outside the domain of A_i. We make the assumption that such a modification will not be performed, that is, the entire tuple will remain unchanged. Each attribute A_i of R_u must satisfy a condition of the form $(A_i \leq U_{A_i})(A_i \geq L_{A_i})$ where L_{A_i} and U_{A_i} are the lower and upper bound, respectively, of its domain. Consequently, the updated value of A_i must satisfy the condition $(\rho(f_{A_i}) \leq U_{A_i})(\rho(f_{A_i}) \geq L_{A_i})$ and this must hold for every $A_i \in \alpha(R_u)$. The conjunction of all these conditions will be denoted by $\mathcal{C}_B(\mathbf{F}_M)$, that is,

$$C_B(\mathbf{F}_M) \equiv \bigwedge_{A_i \in \alpha(R_u)} (\rho(f_{A_i}) \leq U_{A_i})(\rho(f_{A_i}) \geq L_{A_i})$$

The following example illustrates the notation introduced above.

Example 3.3 Consider two relation schemes $R_1(H, I, J)$ and $R_2(K, L)$, and the following modify operation:

$$\text{MODIFY}(R_1, (H > 5) \wedge (I \geq J), \{H := H + 20, I := 15, J := J\}).$$

For this modify operation we have:

$$\begin{array}{lll} f_H \equiv (H := H + 20) & \rho(f_H) \equiv H + 20 & \alpha(\rho(f_H)) = \{H\} \\ f_I \equiv (I := 15) & \rho(f_I) \equiv 15 & \alpha(\rho(f_I)) = \emptyset \\ f_J \equiv (J := J) & \rho(f_J) \equiv J & \alpha(\rho(f_J)) = \{J\} \end{array}$$

$$C_M \equiv (H > 5) \wedge (I \geq J).$$

If the condition of a view definition is $C \equiv (H > 30) \wedge (I = J)$, then

$$C(\mathbf{F}_M) \equiv (H + 20 > 30) \wedge (15 = J).$$

Assuming that the domains of the variables H, I, and J are given by the ranges $[0, 50]$, $[10, 100]$, and $[10, 100]$, respectively, we obtain:

$$C_B(\mathbf{F}_M) \equiv (H + 20 \geq 0) \wedge (H + 20 \leq 50) \wedge (15 \geq 10) \wedge (15 \leq 100) \wedge \\ (J \geq 10) \wedge (J \leq 100).$$

□

Theorem 3.4 *The operation* MODIFY$(R_u, C_M, \mathbf{F}_M)$ *is irrelevant to the view defined by* $E = (\mathbf{A}, \mathbf{R}, C), R_u \in \mathbf{R}$, *if and only if*

$$\forall\, [C_M \wedge C_B(\mathbf{F}_M) \;\Rightarrow\; (\neg C \wedge \neg C(\mathbf{F}_M)) \vee (C \wedge C(\mathbf{F}_M) \bigwedge_{A_i \in I} (A_i = \rho(f_{A_i}))] \quad (3.1)$$

where $I = \mathbf{A} \cap \alpha(R_u)$.

Proof: (Sufficiency) Consider a tuple t from the base $\mathbf{R}$ such that t satisfies C_M and $C_B(\mathbf{F}_M)$. Because condition (3.1) holds for every tuple, it must also hold for t. Hence, either the first or the second disjunct of the consequent must evaluate to *true*. (They cannot both be *true* simultaneously.)

Let us denote by t' the corresponding modified tuple. If the first disjunct is *true*, both $C[t]$ and $C[t']$ must be *false*. This means that neither the original tuple t, nor the modified tuple t', will contribute to the view. Hence changing t to t' will not affect the view.

If the second disjunct is *true*, both $C[t]$ and $C[t']$ must be *true*. In other words, the tuple t contributed to the view and after being modified to t', it still remains in the view. The last conjunct must also be satisfied, which ensures that all attributes of R_u visible in the view have the same values in t and t'. Hence the view will not be affected.

(Necessity) Assume that condition (3.1) does not hold. That means that there exists at least one assignment of values to the attributes, i.e., a tuple t, such that the antecedent is *true* but the consequent is *false*. Denote the corresponding modified tuple by t'. There are three cases to consider.

Case 1: $C[t] = \textit{false}$ and $C[t'] = \textit{true}$. In the same way as in the proof of Theorem 3.1, we can then construct a database instance d from t, where each relation in $\mathbf{R}$ contains a single tuple and such that the resulting view is empty. For this database instance, the modification operation will produce a new instance d' where the only change is to the tuple in relation r_u. The Cartesian product of the relations in $\mathbf{R}$ then contains exactly one tuple, which agrees with t on all attributes except on the attributes changed by the update. Hence, the view $v(E, d')$ will contain one tuple since $C[t'] = \textit{true}$. This proves that the modify operation is not irrelevant to the view.

Case 2: $C[t] = \textit{true}$ and $C[t'] = \textit{false}$. Can be proven in the same way as Case 1, with the difference that the view contains originally one tuple and the modification results in a deletion of that tuple from the view.

Case 3: $C[t] = \textit{true}$, $C[t'] = \textit{true}$ but $\bigwedge_{A_i \in I}(A_i = \rho(f_{A_i}))$ is *false*, that is, $t[A_i] \neq t'[A_i]$ for some $A_i \in \mathbf{A} \cap \alpha(R_u)$. In the same way as above, we can construct an instance where each relation in $\mathbf{R}$ contains only a single tuple, and where the view also contains a single tuple, both before and after the modification. However, in this case the value of attribute A_i will change as a result of performing the MODIFY operation. Since $A_i \in \mathbf{A}$, this change will be visible in the view. This proves that the update is not irrelevant to the view. □

The following example illustrates the theorem.

Example 3.4 Suppose the database consists of the two relations r_1 and r_2 on schemes $R_1(H, I)$ and $R_2(J, K)$ where H, I, J and K all have the domain [0, 30]. Let the view and modify operation be defined as:

$$E = (\{I, J\}, \{R_1, R_2\}, (H > 10)(I = K))$$
$$\text{MODIFY } (R_1, \{R_1\}, (H > 20), \{(H := H + 5), (I := I)\}).$$

Thus the condition given in Theorem 3.4 becomes

$$\begin{aligned}\forall\ \ & H,I,J,K\,[(H>20)(H+5\geq 0)(H+5\leq 30)\\ & \Rightarrow\ (\neg((H>10)(I=K)))\wedge(\neg((H+5>10)(I=K)))\\ & \quad\vee(H>10)(I=K)(H+5>10)(I=K)(I=I)]\end{aligned}$$

which can be simplified to

$$\begin{aligned}\forall\ \ & H,I,K\,[(H>20)(H\leq 25)\\ & \Rightarrow\ (\neg((H>10)(I=K)))\wedge(\neg((H>5)(I=K)))\\ & \quad\vee(H>10)(I=K)].\end{aligned}$$

By inspection we see that if $I = K$, then the second term of the consequent will be satisfied whenever the antecedent is satisfied. If $I \neq K$, the first term of the consequent is always satisfied. Hence, the implication is valid and we conclude that the update is irrelevant to the view. □

3.4 Algorithm for detecting irrelevant tuples

We stated in the beginning of this chapter that the detection of irrelevant updates is important because re-evaluation of the relational expression defining a view may be avoided completely, or at least, the number of tuples considered in the re-evaluation may be reduced.

In Chapter 5, we will present an approach to differentially updating a view, the approach being based on the fact that any update operation applied to a relation can be reduced to a set of tuples being deleted, a set of tuples being inserted, or a combination of both. In that context, detecting which tuples in a set of tuples affecting a relation are irrelevant is important.

This section presents an algorithm based on the algorithm by Rosenkrantz and Hunt [ROSEN80] for determining which tuples in a set of tuples are irrelevant to a view. The type of Boolean expressions allowed by the algorithm are those defined in Chapter 2 on page 11. Before describing the algorithm we need another definition.

Definition 3.2 Consider a tuple $t \in r$ on scheme R, and a conjunctive Boolean expression C. Let $Y = \alpha(C)$, and $Y_1 = Y \cap R$ (the attributes of Y in R). We distinguish between two types of atomic formulae in $C[t]$, called *variant* and *invariant* formulae, respectively.

- Variant formulae are those directly affected by the substitution of $t[A]$ in C for $A \in Y_1$. This type of formula may have the form $(x\ \theta\ t[A])$, or $(t[A]\ \theta\ d)$; where x is a variable and $t[A], d$ are constants. Furthermore, formulae of the form $(x\ \theta\ t[A])$ are called *variant non-evaluable* formulae, and formulae of the form $(t[A]\ \theta\ d)$ are called *variant evaluable* formulae. Variant evaluable formulae are either *true* or *false*.

- Invariant formulae are those that remain invariant with respect to the substitution of t for Y_1 in $\mathcal{C}$. This type of formula may have the form $(x\ \theta\ c)$, or $(x\ \theta\ y + c)$; where x, y are variables, and c is a constant.

Notice that the classification of atomic formulae in $\mathcal{C}$ depends on the relation scheme of the set of tuples t substituting for attributes Y_1 in $\mathcal{C}$. □

Algorithm 3.1

Input:

i) a conjunctive Boolean expression $\mathcal{C} = \beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_n$, where each β_i, $1 \le i \le n$, is an atomic formula of the form $(x\ \theta\ y)$, $(x\ \theta\ y + c)$, or $(x\ \theta\ c)$, where x, y are variables (representing attributes) and c is a constant;

ii) a relation scheme R of the relation to be updated; and

iii) a set of tuples $T_{in} = \{t_1, t_2, \ldots, t_q\}$ on scheme R. T_{in} contains those tuples inserted into or deleted from the relation r.

Output: a set of tuples $T_{out} \subseteq T_{in}$ which are (potentially) relevant to the view.

1. The conjunctive expression $\mathcal{C}$ is normalized (see below).
2. The normalized conjunctive expression $\mathcal{C}_N$ is expressed as $\mathcal{C}_{INV} \wedge \mathcal{C}_{VEVAL} \wedge \mathcal{C}_{VNEVAL}$. $\mathcal{C}_{INV}$ is a conjunctive subexpression containing only invariant formulae. $\mathcal{C}_{VEVAL}$ is a conjunctive subexpression containing only variant evaluable formulae. $\mathcal{C}_{VNEVAL}$ is a conjunctive subexpression containing only variant non-evaluable formulae.
3. Using $\mathcal{C}_{INV}$, build the invariant portion of the directed weighted graph.
4. For each tuple $t \in T_{in}$ do the following:
   4.1 Substitute the values of t for the appropriate variables in $\mathcal{C}_{VEVAL}$. If $\mathcal{C}_{VEVAL}[t] = \mathit{false}$, then ignore tuple t, otherwise continue to step 4.2.
   4.2 Substitute the values of t for the appropriate variables in $\mathcal{C}_{VNEVAL}$. Build the variant portion of the graph and check whether the substituted conjunctive expression represented by the graph is satisfiable. If the expression is satisfiable, then add t to T_{out}, otherwise ignore it. □

An important component of the algorithm is the construction of a directed weighted graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \alpha(\mathcal{C}) \cup \{0\}$ is the set of nodes, and $\mathcal{E}$ is the set of directed weighted edges representing atomic formulae in $\mathcal{C}$. Each member of $\mathcal{E}$ is a triple (n_o, n_d, w), where n_o, $n_d \in \mathcal{V}$ are the *origin* and *destination* nodes respectively, and w is the *weight* of the edge. An atomic formula $(x \le y + c)$ translates to the edge (x, y, c). An atomic formula $(x \ge y + c)$ translates to the edge

$(y, x, -c)$. An atomic formula $(x \leq c)$ translates to the edge $(\text{'0'}, x, c)$. An atomic formula $(x \geq c)$ translates to the edge $(x, \text{'0'}, -c)$.

The normalization procedure mentioned in the algorithm takes a conjunctive expression and transforms it into an equivalent one where each atomic formula has as comparison operator either $\leq$ or $\geq$. Atomic formulae $(x < y+c)$ are transformed into $(x \leq y+c-1)$. Atomic formulae $(x > y+c)$are transformed into $(x \geq y+c+1)$. Atomic formulae $(x = y + c)$ are transformed into $(x \leq y + c) \wedge (x \geq y + c)$.

The satisfiability test consists of checking whether the directed weighted graph contains a negative weight cycle or not. The expression is unsatisfiable if the graph contains a negative cycle. Algorithm 3.1 is a modification of the algorithm by Rosenkrantz and Hunt [ROSEN80] applied to testing the satisfiability of the condition $C[t]$ for each $t \in T_{in}$. Therefore, Algorithm 3.1 runs in time $O(|T_{in}||\mathcal{V}|^3)$. Also, the correctness of Algorithm 3.1 follows directly from the correctness of the algorithm by Rosenkrantz and Hunt.

Certainly, Algorithm 3.1 can also be used to test the satisfiability of Boolean expressions alone, that is, when we do not know what are the values taken by some of the variables mentioned in the condition. In this case, $C_{VEVAL} = nil$, $C_{VNEVAL} = nil$, and $C = C_{INV}$. Algorithm 3.1 is in this case the same as Rosenkrantz and Hunt's. The following example illustrates Algorithm 3.1.

Example 3.5 Consider two relation schemes $R_1(H, I)$ and $R_2(J, K, L)$, and a view defined by the expression

$$E = (\{I, J\}, \{R_1, R_2\}, (H > 10)(L < 50)(I = K)(L > K + 24)).$$

Suppose that the set of tuples $T = \{(10, 20), (12, 24), (14, 32)\}$ are to be either inserted into or deleted from relation r_1 on scheme R_1. The input to Algorithm 3.1 consists of the Boolean expression

$$C \equiv (H > 10)(L < 50)(I = K)(L > K + 24),$$

the relation scheme $R_1(H, I)$, and the set of tuples $T_{in} = T$.

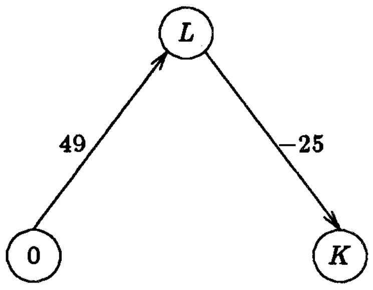
1. The Boolean expression C is transformed into the normalized Boolean expression

$$C_N \equiv (H \geq 11)(L \leq 49)(I \leq K)(I \geq K)(L \geq K + 25).$$

2. Since the tuples in T_{in} are defined on scheme R_1, we obtain:

$$\begin{aligned} C_{INV} &\equiv (L \leq 49)(L \geq K + 25) \\ C_{VEVAL} &\equiv (H \geq 11) \\ C_{VNEVAL} &\equiv (I \leq K)(I \geq K). \end{aligned}$$

3. Using C_{INV}, we build the invariant portion of the directed weighted graph as shown below.

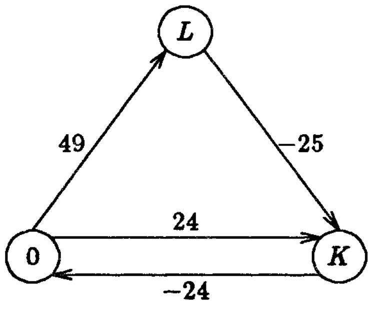


4. This step is performed for each tuple $t \in T_{in}$. For the tuple $t = (10, 20)$ we obtain $C_{VEVAL}[t] \equiv (10 > 11) \equiv \mathit{false}$. Thus, the tuple $(10, 20)$ is irrelevant.

   For the tuple $t = (12, 24)$ we obtain $C_{VEVAL}[t] \equiv (12 > 11) \equiv \mathit{true}$, therefore Step 4.2 of the algorithm is performed. At Step 4.2 we obtain

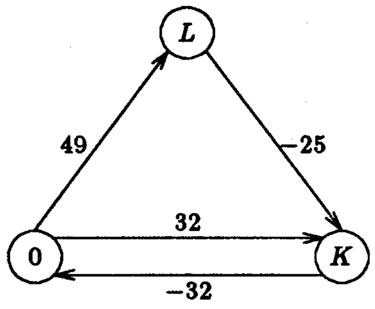
   $$C_{VNEVAL}[t] \equiv (24 \leq K)(24 \geq K).$$

   The complete directed weighted graph is shown below.



   We can see that the graph contains no negative-weight cycle, therefore the expression $C[t]$ is satisfiable which in turn implies that the tuple $(12, 24)$ is ***not*** irrelevant.

   Finally, for the tuple $t = (14, 32)$ we obtain $C_{VEVAL}[t] \equiv (14 > 11) \equiv \mathit{true}$. At Step 4.2 of the algorithm we obtain $C_{VNEVAL}[t] \equiv (32 \leq K)(32 \geq K)$. The complete directed weighted graph is shown below.



The graph contains the negative-weight cycle $0 \rightarrow L \rightarrow K \rightarrow 0$ of weight -8, which clearly indicates that the expression

$$\mathcal{C}[t] \equiv (14 \geq 11)(L < 50)(32 = K)(L > K + 24)$$

can never be satisfied. Therefore, the tuple $(14, 32)$ is irrelevant.

The output of the algorithm is $T_{out} = \{(12, 24)\}$. Thus, using Algorithm 3.1 we have been able to reduce from three to one the tuples that must be consider to update the view defined by E as a result of the insert operation. The next step is to use the set T_{out} as part of the input to an algorithm for differential re-evaluation of views. Such algorithm is presented as Algorithm 5.1 in Chapter 5 of this thesis. □

3.5 Summary

We have presented necessary and sufficient conditions for the detection of irrelevant updates as introduced in Chapter 2. The detection of irrelevant updates is done without any access to the database. The update operations supported represent a nontrivial class of updates available in current relational languages. Finally, a polynomial algorithm for detecting irrelevant tuples was presented.

Chapter 4

Autonomously Computable Updates

If an update operation is not irrelevant to a view, then some data from the base relations may be needed to update the view. An important case to consider, however, is one in which all the data needed is contained in the view itself. In other words, the new state of the view can be computed solely from the view definition, the current state of the view, and the information contained in the update operation. Updates of this type are called *autonomously computable updates.* Within this case, two subcases can be further distinguished, called *unconditional* or *conditional* as explained below.

Definition 4.1 Consider a view definition E and an update operation $\mathcal{U}$, both defined over the relation schemes D. Let d denote an instance of D before applying $\mathcal{U}$ and d' the corresponding instance after applying $\mathcal{U}$.

- The effect of the operation $\mathcal{U}$ on instances of the view defined by E is said to be *unconditionally autonomously computable* if there exists a function $F_{\mathcal{U},E}(v(E,d))$ such that

$$\begin{array}{l} \forall \;\; d \\ \quad v(E,d') = F_{\mathcal{U},E}(v(E,d)). \end{array}$$

- The effect of the operation $\mathcal{U}$ on an instance $v(E,\hat{d})$ of the view defined by E is said to be *conditionally autonomously computable with respect to* $v(E,\hat{d})$ if there exists a function $F_{\mathcal{U},E}(v(E,d))$ such that

$$\begin{array}{l} \forall \;\; d \text{ such that } v(E,d) = v(E,\hat{d}) \\ \quad v(E,d') = F_{\mathcal{U},E}(v(E,d)). \end{array}$$

□

In this chapter all updates considered are assumed to be relevant (i.e., the test for irrelevancy is false). Section 4.1 presents some basic concepts required in the chapter. Section 4.2 presents necessary and sufficient conditions for the detection of unconditionally autonomously computable updates, and Section 4.3 does the same for conditionally autonomously computable updates.

4.1 Basic concepts

The concepts covered by the following definitions were originally introduced by Larson and Yang [LARSO85].

Definition 4.2 Let C be a Boolean expression over the variables $x_1, x_2, \ldots, x_n$. The variables $x_1, \ldots, x_k$, $k \leq n$, are said to be *nonessential* in C if

$$\begin{aligned} \forall \quad & x_1, \ldots, x_k, x_{k+1}, \ldots, x_n, x'_1, \ldots, x'_k \\ & [C(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n) \Leftrightarrow C(x'_1, \ldots, x'_k, x_{k+1}, \ldots, x_n)]. \end{aligned}$$

Otherwise, $x_1, \ldots, x_k$ are *essential* in C. □

A nonessential variable can be eliminated from the condition simply by replacing it with any value from its domain. This will in no way change the value of the condition. For example, the variable H is nonessential in the condition

$$(I > 5)(J = I)((H > 5) \vee (H < 10)),$$

since the predicate $(H > 5) \vee (H < 10)$ will evaluate to *true* for any value assigned to variable H. Similarly H is nonessential in

$$(I > 5)(H > 5)(H \leq 5),$$

since the condition will evaluate to *false* for any value assigned to H.

Definition 4.3 Let C_0 and C_1 be Boolean expressions over the variables $x_1, x_2, \ldots, x_n$. The variables $x_1, x_2, \ldots, x_k$, $k \leq n$, are said to be *computationally nonessential* in C_0 *with respect to* C_1 if

$$\begin{aligned} \forall \quad & x_1, \ldots, x_k, x_{k+1}, \ldots, x_n, x'_1, \ldots, x'_k \\ & [C_1(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n) \wedge C_1(x'_1, \ldots, x'_k, x_{k+1}, \ldots, x_n) \\ & \Rightarrow \quad (C_0(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n) \Leftrightarrow C_0(x'_1, \ldots, x'_k, x_{k+1}, \ldots, x_n))]. \end{aligned}$$

Otherwise, $x_1, x_2, \ldots, x_k$ are *computationally essential* in C_0 with respect to C_1. □

The idea behind this definition is that if a set of variables $x_1, x_2, \ldots, x_k$ are computationally nonessential in C_0 with respect to C_1, then given any tuple defined over the variables $x_1, x_2, \ldots, x_n$ satisfying the condition C_0, where the variables $x_1, x_2, \ldots, x_k$ have been projected out, we can still correctly evaluate whether the tuple satisfies the condition C_1 or not without knowing the exact values for the missing variables $x_1, x_2, \ldots, x_k$. This is done by assigning surrogate values to the variables $x_1, x_2, \ldots, x_k$ as explained by Larson and Yang [LARSO85].

Example 4.1 Let $C_1 \equiv (H > 5)$ and $C_0 \equiv (H > 0)(I = 5)(J > 10)$. It is easy to see that if we are given a tuple (i, j) for which it is known that the full tuple (h, i, j) satisfies C_1, then we can correctly evaluate C_0. If (h, i, j) satisfies C_1 then the value h must be greater than 5, and consequently it also satisfies $(H > 0)$. Hence, we can correctly evaluate C_0 for the tuple (i, j) by assigning to H any surrogate value greater than 5. □

For completeness, we give a brief description from [LARSO85] on how surrogate values are computed. Consider Algorithm 3.1, which given a set of tuples T and a Boolean expression C, tests whether $C[t_i]$, $t_i \in T$, is satisfiable. Algorithm 3.1 has to be modified to return an assignment of values to the variables in $\alpha(C[t_i])$ when the expression is satisfiable, such that the expression evaluates to *true*.

Now, consider the *PSJ*-expression $Q = (\mathbf{A}_q, \mathbf{R}_q, C_q)$ defining a query and the *PSJ*-expression $E = (\mathbf{A}, \mathbf{R}, C)$ defining a view, and suppose that we want to find which tuples in $v(E, d)$ satisfy C_q. We are interested in the case where all variables in the set $\mathbf{N} = (\alpha(C_q) \cup \alpha(C)) - \mathbf{A}$ are computationally nonessential in C_q with respect to C, since every tuple in the view satisfies C. Let $\alpha(C_q) \cup \alpha(C) = \{y_1, y_2, \ldots, y_n\}$, $\mathbf{N} = \{y_1, y_2, \ldots, y_k\}$, $k \leq n$, and assume that we have shown that all variables in $\mathbf{N}$ are computationally nonessential in C_q with respect to C. Surrogate values for $y_1, y_2, \ldots, y_k$ can then be computed by invoking Algorithm 3.1 with input: $v(E, d)$, $\mathbf{A}$, and C.

For each tuple $t_i \in v(E, d)$ the algorithm returns a set of values $y_1^0, y_2^0, \ldots, y_n^0$, where $y_i^0 = t_i[y_i]$, $k+1 \leq i \leq n$. The values $y_1^0, y_2^0, \ldots, y_k^0$ are the required surrogate values needed to evaluate C_q on the tuple t_i.

The fact that $C[t_i]$ is satisfiable guarantees that surrogate values for the variables $y_1, y_2, \ldots, y_k$ always exist.

Algorithm 3.1 already saves time by building the invariant portion of the graph used to test for satisfiability only once.

Definition 4.4 Let C be a Boolean expression over variables $x_1, x_2, \ldots, x_n$, $y_1, y_2, \ldots, y_m$. A variable y_i, $1 \leq i \leq m$, is said to be *uniquely determined* by $x_1, x_2, \ldots, x_n$ and C if

$$\forall \quad x_1, \ldots, x_n, y_1, \ldots, y_m, y'_1, \ldots, y'_m$$
$$[C(x_1, \ldots, x_n, y_1, \ldots, y_m) \wedge C(x_1, \ldots, x_n, y'_1, \ldots, y'_m) \Rightarrow (y_i = y'_i)].$$

□

If a variable y_i (or a subset of the variables $y_1, y_2, \ldots, y_m$) is uniquely determined by a condition $\mathcal{C}$ and the variables $x_1, \ldots, x_n$, then given any tuple $t = (x_1, \ldots, x_n)$, such that the full tuple $(x_1, \ldots, x_n, y_1, \ldots, y_m)$ is known to satisfy $\mathcal{C}$, the missing value of the variable y_i can be correctly reconstructed. How to reconstruct the values of uniquely determined variables was also shown by Larson and Yang [LARSO85] and it is similar to the way surrogate values are derived for computationally nonessential variables as explained above. If the variable y_i is not uniquely determined, then we cannot guarantee that its value is reconstructible for *every* tuple. However, it may still be reconstructible for *some* tuples.

Example 4.2 Let $\mathcal{C} \equiv (I = H)(H > 7)(K = 5)$. It is easy to prove that I and K are uniquely determined by H and the condition $\mathcal{C}$. Suppose that we are given a tuple that satisfies $\mathcal{C}$ but only the value of H is known. Assume that $H = 10$. Then we can immediately determine that the values of I and K must be 10 and 5, respectively. □

Definition 4.5 Let $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$ be a view and let $\mathbf{A}_E$ be the set of all attributes in $\alpha(\mathbf{R})$ that are uniquely determined by the attributes in $\mathbf{A}$ and the condition $\mathcal{C}$. Then $\mathbf{A}^+ = \mathbf{A} \cup \mathbf{A}_E$ is called the *extended attribute set* of E. □

It is proved by Larson and Yang [LARSO85] that $\mathbf{A}^+$ is the maximal set of attributes for which values can be reconstructed for *every* tuple of E.

4.2 Unconditionally autonomously computable updates

It should be stressed that if the update $\mathcal{U}$ on a view defined by E is unconditionally autonomously computable, then the update can be performed for every view instance $v(E, d)$. This characterization is important primarily because of the potential cost savings realized by updating the view using only the information in its current instance.

The theorems presented in this section show that if an update $\mathcal{U}$ is unconditionally autonomously computable on instances of a view defined by E, then there exists a deterministic procedure $P(\mathcal{U}, E, v(E, d))$ which correctly computes $v(E, d')$ using only the information provided by its input parameters $\mathcal{U}$, E, and $v(E, d)$, regardless of the database instance d. In other words, P does not need to access any other relations than the view instance. The actual procedure depends, of course, on the update operation $\mathcal{U}$. The sufficiency parts of the proofs in the theorems outline the required procedure.

The following lemma shows that no such procedure exists when the effect of an update is not unconditionally autonomously computable.

Lemma 4 *Consider a view definition $E = (\mathbf{A}, \mathbf{R}, C)$ with instance $v(E, d)$, and an update $\mathcal{U}$. Let $P(\mathcal{U}, E, v(E, d))$ be a deterministic procedure (function) whose result depends only on the values of its input parameters and which computes the corresponding state of the view after the update. If the effect of the update $\mathcal{U}$ on the view is not unconditionally autonomously computable, then P cannot correctly compute the updated instance $v(E, d')$ for every possible database instance d.*

Proof: If an update is not unconditionally autonomously computable, then there is no function $F_{\mathcal{U},E}(v(E, d))$ which can compute $v(E, d')$ for every database instance d. In other words, there exist at least two database instances d_1 and d_2 such that $v(E, d_1) = v(E, d_2)$ and $v(E, d'_1) \neq v(E, d'_2)$. For these two view instances, the procedure P must produce the same updated view instance because the input parameters are exactly the same. Hence, the result must be wrong for at least one of the database instances d_1 or d_2. $\square$

Therefore, any procedure P that claims to compute the updated state of a view instance based only on the information provided by $\mathcal{U}$, E, and the current view instance will provide an incorrect result when the effect of the update on instances of E is not unconditionally autonomously computable. Under such circumstances, we need a different procedure P' which requires additional data (e.g., tuples from the base relations) to compute the updated view instance correctly.

4.2.1 Insertions

Consider an operation INSERT (R_u, T) where T is a set of tuples to be inserted into r_u. Let a view be defined by $E = (\mathbf{A}, \mathbf{R}, C), R_u \in \mathbf{R}$. The effect of the INSERT operation [1] on the view is unconditionally autonomously computable if

A. for each tuple $t \in T$ we can correctly decide whether t will (regardless of the database instance) satisfy the selection condition C and hence should be inserted into the view, and

B. the values for all attributes visible in the view can be obtained from t only.

Note that if t could cause the insertion of more than one tuple into the view, then the update is not autonomously computable. Suppose that t generates two different tuples to be inserted: t_1 and t_2. Then t_1 and t_2 must differ in at least one attribute visible in the view; otherwise only one tuple would be inserted. Suppose that they differ on $A_i \in \mathbf{A}$. A_i cannot be an attribute of R_u because the exact value of every attribute in R_u is given by t. Hence, the values of A_i in t_1 and t_2 would have to be obtained from other tuples. We cannot always guarantee that the required tuples will be available in the current instance of the view.

[1] If $R_u \notin \mathbf{R}$, then the update cannot have any effect on the view.

Theorem 4.1 *Consider a view defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $\mathbf{R} = \{R_1, \ldots, R_m\}$, and the update INSERT $(R_u, \{t\})$, $R_u \in \mathbf{R}$. The effect of the insert operation on the view E is unconditionally autonomously computable if and only if $\mathbf{R} = \{R_u\}$.*

Proof: (Sufficiency) If $\mathbf{R} = \{R_u\}$, then all attributes required to compute the selection condition C as well as all the visible attributes $\mathbf{A}$ are contained in the new tuple t. Hence, the effect of the insertion is autonomously computable.

(Necessity) If $\mathbf{R}$ includes other base relations schemes in addition to R_u, then the insertion of tuple t into r_u may affect the view defined by E. Whether it does depends on the existence of tuples in relations whose schemes are in $\mathbf{R} - \{R_u\}$. We can easily construct a database instance d where it is necessary to access the database to verify the existence of such tuples, even for the case when $\alpha(C) \subseteq \alpha(R_u)$ and $\mathbf{A} \subseteq \alpha(R_u)$. The database instance $d = \{r_1, r_2, \ldots, r_m\}$ is constructed as follows. Each relation r_i, $1 \leq i \leq m$, $i \notin \{u, j\}$, contains a single tuple t_i, and relations r_u and r_j are empty. Clearly $v(E, d) = \emptyset$. Now suppose that tuple t is inserted into r_u and furthermore, assume that $C[t] = true$. Even though tuple t satisfies the selection condition of the view and it contains all visible attributes, it will not create an insertion into the view because relation r_j contains no tuple. The effect on the view of inserting t then depends on data outside the view. Therefore, the effect of the update cannot be unconditionally autonomously computable. □

Example 4.3 Consider two relation schemes $R_1(H, I)$ and $R_2(J, K)$. Let a view and insert operation be defined as:

$$E_1 = (\{H, I\}, \{R_1\}, (H > 10)(I = 5))$$
$$\text{INSERT}(R_1, \{(6, 10), (12, 5)\}).$$

Clearly, the effect of the insert operation is unconditionally autonomously computable on every instance of the view defined by E_1. On the other hand, if a view is defined by the expression

$$E_2 = (\{H, I\}, \{R_1, R_2\}, (I = J)(H > 10)),$$

then the insert operation cannot be unconditionally autonomously computable (on every instance of E_2) because the insertion of a new tuple into the view will depend on the existence of certain tuples in relation r_2. □

4.2.2 Deletions

To handle deletions autonomously, we must be able to determine, for every tuple in the view, whether or not it satisfies the delete condition. This is covered by the following theorem.

Theorem 4.2 *The effect of the operation DELETE (R_u, C_D) on instances of the view defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, is guaranteed to be autonomously computable if and only if the attributes in $(\alpha(C_D) \cup \alpha(C)) - \mathbf{A}^+$ are computationally nonessential in C_D with respect to C.*

Proof: (Sufficiency) If the attributes in $(\alpha(C_D) \cup \alpha(C)) - \mathbf{A}^+$ are computationally nonessential in C_D with respect to C, then we can correctly evaluate the condition C_D on every tuple in the view $v(E,d)$ by assigning surrogate values to the attributes in $\alpha(C_D) - \mathbf{A}^+$.

(Necessity) Assume that $(\alpha(C_D) \cup \alpha(C)) - \mathbf{A}^+$ contains an attribute x, and assume that x is computationally essential in C_D with respect to C. We can then construct two tuples t_1 and t_2 over the attributes in $\mathbf{A}^+ \cup \alpha(C) \cup \alpha(C_D)$ such that they both satisfy C, t_1 satisfies C_D but t_2 does not, and t_1 and t_2 agree on all attributes except attribute x. The existence of two such tuples follows from the definition of computationally nonessential attributes. In the same way as in the proof of Theorem 3.2, each of t_1 and t_2 can now be extended into an instance of D, where each relation contains a single tuple. Both instances will give the same instance of the view, consisting of a single tuple $t_1[\mathbf{A}]$ (or $t_2[\mathbf{A}]$). In one instance, the tuple should be deleted from the view, in the other one it should not. The decision depends on the value of attribute x which is not visible in the view. Hence the update cannot be unconditionally autonomously computable. □

Example 4.4 Consider two relation schemes $R_1(H,I), R_2(J,K)$. Let the view and the delete operation be defined as:

$$E = (\{J,K\}, \{R_1,R_2\}, (I=J)(H<20))$$
$$\text{DELETE}(R_1, (I=20)(H<30))$$

The attributes in $(\alpha(C_D) \cup \alpha(C)) - \mathbf{A}^+ = \{H,I,J\} - \{I,J,K\} = \{H\}$ must be computationally nonessential in C_D with respect to C in order for the effect of the deletion to be autonomously computable. That is, the following condition must hold:

$$\begin{aligned} \forall\ & H,I,J,K,H'\ [(I=J)(H<20) \wedge (I=J)(H'<20) \\ \Rightarrow\ & ((I=20)(H<30) \Leftrightarrow (I=20)(H'<30))]. \end{aligned}$$

The conditions $(H < 30)$ and $(H' < 30)$ will both be *true* whenever $(H < 20)$ and $(H' < 20)$ are *true*. Any value taken on by the variable I will make the condition $I = 20$ either *true* or *false*, and hence the consequent will always be satisfied. Therefore, the variable H is computationally nonessential in C_D with respect to C. This guarantees that for any tuple in the view we can always correctly evaluate the delete condition by assigning surrogate values to the variable H. Notice that because $I \in \mathbf{A}^+$ is uniquely determined by C and the variables $\mathbf{A}$, we must also find surrogate values for I.

To further clarify the concept of computationally nonessential, consider the following instance of the view E.

$v(E,d)$:

J	K
10	15
20	25

We now have to determine on a tuple by tuple basis which tuples in the view should be deleted. Consider tuple $t_1 = (10, 15)$ and the condition $C \equiv (I = J)(H < 20)$. We substitute for the variables J and K in C the values 10 and 15, respectively, to obtain $C[t_1] \equiv (I = 10)(H < 20)$. Any values for H, I that make $C[t_1] = true$, are valid surrogate values. For I the only value that can be assigned is 10 and for H we can assign, for example, the value 19. We can then evaluate C_D using these surrogate values, and find that $(10 = 20)(19 < 30) = false$. Therefore, tuple t_1 should not be deleted from $v(E, d)$. Similarly, for $t_2 = (20, 25)$ we obtain $C[t_2] \equiv (I = 20)(H < 20)$. Surrogate values for H and I that make $C[t_2] = true$ are $I = 20, H = 19$. We then evaluate C_D using these surrogate values and find that $(20 = 20)(19 < 30) = true$. Therefore, tuple t_2 should be deleted from $v(E, d)$. □

4.2.3 Modifications

Deciding whether modifications can be performed autonomously is more complicated than for insertions or deletions. In general, a modify operation may generate insertions into, deletions from, and modifications of existing tuples in the view as a result of updating a relation. We summarize the conditions imposed by these possibilities in the following four steps and give one theorem for each step.

A. Prove that every tuple selected for modification which does not satisfy C before modification, will not satisfy C after modification. This means that no new tuples will be inserted into the view.

B. Prove that we can correctly select which tuples in the view should be modified. Call this the *modify set.*

C. Prove that we can correctly select which tuples in the modify set will not satisfy C after modification and hence can be deleted from the view.

D. Prove that, for every tuple in the modify set which will not be deleted, we can (autonomously) compute the new values for all attributes in **A**.

Theorem 4.3A *The operation MODIFY* $(R_u, C_M, \mathbf{F}_M)$ *is guaranteed not to create any new tuples which need to be inserted into the view defined by* $E = (\mathbf{A}, \mathbf{R}, C), R_u \in \mathbf{R}$, *if and only if*

$$\forall\, [\neg C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow \neg C(\mathbf{F}_M)].$$

Proof: (Sufficiency) Assume that the condition holds. Consider a tuple t in the Cartesian product of the relations in the base **R**, and assume that t is selected for modification. Let t' denote the corresponding tuple after modifications. Assume that t does not satisfy C and hence will not have created any tuple in the view. Because the above condition holds for every tuple, it must also hold for t and hence

t' cannot satisfy C. Consequently, modifying t to t' does not cause any new tuple to appear in the view.

(Necessity) If the condition does not hold, then without loss of generality we can construct a database instance such that each relation contains only one tuple and the view is empty before modification but contains one tuple after modification. The fact that the condition does not hold indicates that there exists at least one tuple t defined over the variables $\alpha(\mathbf{R})$ such that

$$(\neg C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow C(\mathbf{F}_M))[t] = \mathit{true}.$$

Using t, we can construct the database instance $d_1 = \{r_{1,1}, \ldots, r_{1,m}\}$, where $r_{1,i} = \{t[\alpha(R_i)]\}$, $1 \le i \le m$. Clearly, $v(E, d_1) = \emptyset$ and $v(E, d'_1)$ will contain one tuple. This proves that the condition is necessary. □

Also, from the necessity part of the above proof we can see why when a modify operation causes the insertion of a new tuple into the view, the effect of the operation cannot be unconditionally autonomously computable on every instance of E. Consider the database instance $d_2 = \{r_{2,1}, \ldots, r_{2,m}\}$, where $r_{2,i} = r_{1,i}$, $1 \le i \le m$, $i \ne u$, and $r_{2,u} = \emptyset$. Clearly, $v(E, d_1) = v(E, d_2) = \emptyset$, however, $v(E, d'_1) \ne v(E, d'_2)$. Therefore, there cannot exist a function which based solely on the information provided by $\mathcal{U}$, E, and the instance $v(E, d)$ will produce the correct result $v(E, d')$ for every database instance d.

Before we proceed with the next theorem, we must comment on an interesting implementation aspect. Given a MODIFY operation and a view defined by an expression E, we first test whether or not the update is irrelevant and then (if it is not irrelevant) we proceed to test whether or not the update is autonomously computable.

The test for irrelevant updates is stated by Theorem 3.4 and the first test for autonomously computable modifications is given by Theorem 4.3A. These two tests can be cascaded as explained below.

The condition for irrelevant modifications is given by

$$\forall \quad [C_M \wedge C_B(\mathbf{F}_M) \Rightarrow (\neg C \wedge \neg C(\mathbf{F}_M)) \vee (C \wedge C(\mathbf{F}_M) \textstyle\bigwedge_{B_i \in \mathcal{I}} (B_i = \rho(f_{B_i}))].$$

where $\mathcal{I} = \mathbf{A} \cap \alpha(R_u)$. This is equivalent to testing

$$\begin{aligned} \forall \quad & [(\neg C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow \neg C(\mathbf{F}_M)) \\ & \wedge \ (C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow C(\mathbf{F}_M) \textstyle\bigwedge_{B_i \in \mathcal{I}} (B_i = \rho(f_{B_i})))]. \end{aligned}$$

which is equivalent to testing

$$\begin{aligned} \forall \quad & [\neg C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow \neg C(\mathbf{F}_M)] \wedge \\ \forall \quad & [C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow C(\mathbf{F}_M) \textstyle\bigwedge_{B_i \in \mathcal{I}} (B_i = \rho(f_{B_i}))]. \end{aligned}$$

Notice that the first condition of the outermost "$\wedge$" operator is exactly the condition stated by Theorem 4.3A. Therefore, if the test for irrelevant modifications is done in two stages according to the above condition and if the results of the tests are recorded somewhere for later use, then when it comes to testing for autonomously computable modifications we can save testing the condition of Theorem 4.3A by recalling the result from the test for irrelevant modifications.

Theorem 4.3B *Consider a view $E = (\mathbf{A}, \mathbf{R}, C)$ and a modify condition C_M, and assume that the condition of Theorem 4.3A holds. Then the tuples from the view satisfying the condition $C_M \wedge C_B(\mathbf{F}_M)$ can be correctly selected for every database instance d if and only if the attributes in*

$$[\alpha(C_M) \cup \alpha(C) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$$

are computationally nonessential in $C_M \wedge C_B(\mathbf{F}_M)$ with respect to C. Recall that $\alpha(C_M) \subseteq \alpha(R_u)$.

Proof: (Sufficiency) If the attributes in $[\alpha(C_M) \cup \alpha(C) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$ are computationally nonessential in $C_M \wedge C_B(\mathbf{F}_M)$ with respect to C, we can correctly evaluate the condition $C_M \wedge C_B(\mathbf{F}_M)$ on every tuple in the view $v(E, d)$ by assigning surrogate values to the attributes in $[\alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$.

(Necessity) Assume that $[\alpha(C_M) \cup \alpha(C) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$ contains attribute x, and assume that x is computationally essential in $C_M \wedge C_B(\mathbf{F}_M)$ with respect to C. We can then construct two tuples t_1 and t_2 over the attributes in $\mathbf{A}^+ \cup \alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))$ such that they both satisfy C, t_1 satisfies $C_M \wedge C_B(\mathbf{F}_M)$ but t_2 does not, and t_1 and t_2 agree on all attributes except attribute x. In the same way as in the proof of Theorem 3.2, each of t_1 and t_2 can now be extended into an instance of D, where each relation contains a single tuple. Both instances will give the same instance of the view, consisting of a single tuple $t_1[\mathbf{A}]$ (or $t_2[\mathbf{A}]$). In one instance, the tuple in the view should be modified, in the other one it should not. The decision depends on the value of attribute x which is not visible in the view. Hence, the update cannot be unconditionally autonomously computable. □

Theorem 4.3C *Given that the condition of Theorem 4.3B holds, the tuples in the view defined by E chosen for modification by C_M that will satisfy C after modification can be correctly selected for every database instance d if and only if the attributes in*

$$[\alpha(C(\mathbf{F}_M)) \cup \alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$$

are computationally nonessential in $C(\mathbf{F}_M)$ with respect to the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M)$.

Proof: (Sufficiency) If every attribute $x \in [\alpha(C(\mathbf{F}_M)) \cup \alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$ is computationally nonessential in $C(\mathbf{F}_M)$ with respect to $C \wedge C_M \wedge C_B(\mathbf{F}_M)$, we can correctly evaluate the condition $C(\mathbf{F}_M)$ on every tuple of the view $v(E, d)$ by assigning surrogate values to the attributes in $\alpha(C(\mathbf{F}_M)) - \mathbf{A}^+$.

(Necessity) Assume that $[\alpha(C(\mathbf{F}_M)) \cup \alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$ contains an attribute x and that x is computationally essential in $C(\mathbf{F}_M)$ with respect to the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M)$. In the same way as in the proof of Theorem 3.2, we can then construct two tuples t_1 and t_2 over the attributes in $\mathbf{A}^+ \cup \alpha(C) \cup \alpha(C_M) \cup \alpha(C(\mathbf{F}_M)) \cup \alpha(C_B(\mathbf{F}_M))$. We first build the tuple t_1 and t_2 making sure that $t_1[x] \neq t_2[x]$. The fact that x is computationally essential in $C(\mathbf{F}_M)$ with respect to the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M)$ guarantees that such values exist. Now, since we require both tuples t_1 and t_2 to satisfy the conditions C, C_M and $C_B(\mathbf{F}_M)$, t_1 to satisfy $C(\mathbf{F}_M)$, and t_2 not to satisfy the condition $C(\mathbf{F}_M)$, we choose the values for the set of variables $(\mathbf{A}^+ \cup \alpha(C) \cup \alpha(C_M) \cup \alpha(C(\mathbf{F}_M)) \cup \alpha(C_B(\mathbf{F}_M))) - \{x\}$ on each tuple accordingly. This construction guarantees that tuples t_1 and t_2 will differ at least on attribute x. Let t_1' and t_2' denote the corresponding tuples after modification. Because $x \in (\alpha(C(\mathbf{F}_M)) - \mathbf{A}^+)$ at least one attribute in $\alpha(C)$, say attribute y, must be affected using variable x in some update expression within the modify operation. Because $t_1'[y]$ and $t_2'[y]$ are computed using two different values of x, then $t_1'[y] \neq t_2'[y]$ (this is true because of the type of update expressions we allow as part of the modify operation). Therefore, t_1' and t_2' will differ on at least attribute y. Again, since x is computationally essential in $C(\mathbf{F}_M)$ with respect to the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M)$ then one of the tuples, say t_1' will satisfy C while t_2' will not. We can now extend t_1 and t_2 to obtain two different database instances where each relation contains only one tuple. In both cases the view contains the same tuple and the tuple is selected for modification. In one case (for the instance obtained from t_2) the single tuple in the view should be deleted after the modification, while in the other case it should not. The decision depends on the value of x, which is not visible in the view nor derivable from it. We cannot correctly decide, based on the data in the view, whether the tuple satisfies C after modification, and thus the update cannot be unconditionally autonomously computable. □

The next theorem establishes the conditions under which the modified values for the attributes in $\mathbf{A}$ can be correctly computed. But, before we state the theorem we need a new definition.

Definition 4.6 Let C be a Boolean expression over the sets of variables W, X, Y and let f represent an expression over the sets of variables X, Y, Z. The value of the expression f is said to be ***uniquely determined*** by condition C and the set of variables X if

$$\forall\, W, X, Y, Z, Y', Z'\ [C(W, X, Y) \wedge C(W, X, Y') \Rightarrow (f(X, Y, Z) = f(X, Y', Z'))].$$

□

Example 4.5 Let $C(H, I, J) \equiv (H = 5) \wedge (I = 10 - J)$ and $f(I, J) := (I + J)$. For any values of I and J that satisfy C we are guaranteed that the value of $I + J$, and hence f, is 10. In other words the condition of Definition 4.6 becomes

$$\begin{aligned} \forall\ \ & H, I, I', J, J' \\ & [(H = 5) \wedge (I = 10 - J) \wedge (H = 5) \wedge (I' = 10 - J') \\ & \Rightarrow\ ((I + J) = (I' + J'))]. \end{aligned}$$

As this is a valid implication we conclude that f is uniquely determined by C (in this case the sets X and Z are empty). Note, that we can state this in spite of the fact that we do not know the value of either I or J. □

Theorem 4.3D *Assume that the condition of Theorem 4.3 C holds. For all tuples in the view which are not to be deleted after modification, the new values for the attributes in* $\mathbf{A}$ *can be correctly computed if and only if for each* $B_i \in \mathbf{A} \cap \alpha(R_u)$, *the value of the expression* $\rho(f_{B_i})$ *is uniquely determined by the condition* $C \wedge C_M \wedge C(\mathbf{F}_M) \wedge C_B(\mathbf{F}_M)$ *and the attributes in* $\mathbf{A}^+$.

Proof: (Sufficiency) Assume that for each $B_i \in \mathbf{A} \cap \alpha(R_u)$, the value of the expression $\rho(f_{B_i})$ is uniquely determined by the condition $C \wedge C_M \wedge C(\mathbf{F}_M) \wedge C_B(\mathbf{F}_M)$ and the attributes in $\mathbf{A}^+$. As $\rho(f_{B_i})$ gives the new value for attribute B_i, this means that the given condition and the visible attributes contain sufficient information to determine the updated values of B_i. We have assumed this for each $B_i \in \mathbf{A} \cap \alpha(R_u)$, therefore, the value of every modified attribute in $\mathbf{A}$ is autonomously computable.

(Necessity) Assume that the values for the modified attributes can be correctly computed, that $\mathbf{A} \cap \alpha(R_u)$ contains a single attribute B_j with a non-trivial f_{B_j}, and that $\rho(f_{B_j})$ is not uniquely determined by the condition $C \wedge C_M \wedge C(\mathbf{F}_M) \wedge C_B(\mathbf{F}_M)$ and the attributes in $\mathbf{A}^+$. We can then construct two tuples t_1 and t_2 over the attributes in $\alpha(R_u) \cup \mathbf{A} \cup \alpha(C) \cup \alpha(C_M) \cup \alpha(C(\mathbf{F}_M)) \cup \alpha(C_B(\mathbf{F}_M))$ such that t_1 and t_2 both satisfy $C, C_M, C(\mathbf{F}_M)$, and $C_B(\mathbf{F}_M)$, and the tuples disagree on the values of some of the attributes in $\alpha(\rho(f_{B_j}))$ but agree on the values of all attributes in $\mathbf{A}$ and on the values of all remaining attributes. The different attribute values are such that evaluating $\rho(f_{B_j})$ over t_1 and t_2 gives two different results. In the same way as in the proof of Theorem 3.2, each of t_1 and t_2 can now be extended into an instance of D, where each relation contains a single tuple. Both instances will give the same instance of the view, consisting of a single tuple $t_1[\mathbf{A}]$ (or $t_2[\mathbf{A}]$). In both instances the tuple in the view should be modified. However, the value of the modified attribute, B_j, will be different depending on whether we use t_1 or t_2. Hence, the values of the modified attributes cannot be correctly computed, and therefore, the update cannot be unconditionally autonomously computable. □

Note in Example 4.5 that we cannot, in general, use Algorithm 3.1 to test the condition of Definition 4.6 because the atomic formula in the consequent may involve more than two variables. However, the following corollary establishes a sufficient condition that allows to compute the new values of the attributes in $\mathbf{A}$ for which we can still use Algorithm 3.1.

Recall that $\alpha(\rho(f_{B_i}))$ denotes the set of attributes occurring in the right hand side expression of f_{B_i}. Define the set Z as

$$Z = \cup_{B_i \in \mathbf{A} \cap \alpha(R_u)} \alpha(\rho(f_{B_i}))$$

that is, Z is the set of attributes from which the new values for the attributes in $\mathbf{A}$ are computed.

Corollary 4.3D *The values of the attributes in* $\mathbf{A}$ *can be correctly computed if all variables in* Z *are uniquely determined by the condition* $C \wedge C_M \wedge C(\mathbf{F}_M) \wedge C_B(\mathbf{F}_M)$ *and the attributes in* $\mathbf{A}^+$.

Proof: If the values of the variables in Z are uniquely determined by the condition $C \wedge C_M \wedge C(\mathbf{F}_M) \wedge C_B(\mathbf{F}_M)$ and the attributes in $\mathbf{A}^+$, then all functions f_{B_j}, $B_j \in \mathbf{A} \cap \alpha(R_u)$, are also uniquely determined by the condition $C \wedge C_M \wedge C(\mathbf{F}_M) \wedge C_B(\mathbf{F}_M)$ and the attributes in $\mathbf{A}^+$. Thus, by Theorem 4.3D the values of the attributes in $\mathbf{A}$ can be correctly computed. □

In practice it is probably better to enforce the conditions of Corollary 4.3D than the conditions of Theorem 4.3D.

We give an example which proceeds through the four steps associated with Theorems 4.3A through 4.3D, at each step testing the appropriate condition.

Example 4.6 Suppose a database consists of the two relation schemes $R_1(H, I)$ and $R_2(J, K, L)$ where H, I, J, K and L each have the domain [0, 30]. Let the view and modify operation be defined as:

$$\begin{array}{l} E = (\{I, J\}, \{R_1, R_2\}, (I = K)(L = 20)) \\ \text{MODIFY}(R_2, (K > 5)(K \leq 22), \{(J := L + 3), (K := K), (L := L)\}). \end{array}$$

From the definition of the view we can see that $\mathbf{A} = \{I, J\}$ and $\mathbf{A}^+ = \{I, J, K\}$.

Step A:

$$\begin{array}{l} \forall \;\; I, K, L \;\; [\neg((I = K)(L = 20)) \wedge (K > 5)(K \leq 22) \\ \wedge \;\; (L + 3 \geq 0)(L + 3 \leq 30)(K \geq 0)(K \leq 30)(L \geq 0)(L \leq 30) \\ \qquad \Rightarrow \; \neg((I = K)(L = 20))] \end{array}$$

This shows that the given implication is valid, and therefore, the given modify operation will not introduce new tuples into $v(E, d)$.

Step B: $[\alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+ = \{I, K, L\} - \{I, J, K\} = \{L\}$

$$\begin{array}{l} \forall \;\; I, K, L, L' \;\; [(I = K)(L = 20) \wedge (I = K)(L' = 20) \\ \quad \Rightarrow \;\; ((K > 5)(K \leq 22) \\ \qquad \wedge \;\; (L + 3 \geq 0)(L + 3 \leq 30)(K \geq 0)(K \leq 30)(L \geq 0)(L \leq 30) \\ \qquad \Leftrightarrow \;\; (K > 5)(K \leq 22) \\ \qquad \wedge \;\; (L' + 3 \geq 0)(L' + 3 \leq 30)(K \geq 0)(K \leq 30)(L' \geq 0)(L' \leq 30) \end{array}$$

Clearly, any value for L and L' that make the antecedent *true* will cause the two conditions in the consequent to evaluate to *true*. Therefore, L is computationally nonessential in $C_M C_B(\mathbf{F}_M)$ with respect to C, and thus the tuples in the view that satisfy C_M can be correctly selected.

Step C:

$$\begin{aligned}[\alpha(C(\mathbf{F}_M)) \cup \alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+ &= \{I, K, L\} - \{I, J, K\} \\ &= \{L\}\end{aligned}$$

$$\begin{aligned}\forall\ \ & I, K, L, L'\ [(I = K)(L = 20) \wedge (K > 5)(K \leq 22) \\ & \wedge\ (L + 3 \geq 0)(L + 3 \leq 30)(K \geq 0)(K \leq 30)(L \geq 0)(L \leq 30) \\ & \wedge\ (I = K)(L' = 20) \wedge (K > 5)(K \leq 22) \\ & \wedge\ (L' + 3 \geq 0)(L' + 3 \leq 30)(K \geq 0)(K \leq 30)(L' \geq 0)(L' \leq 30) \\ & \quad \Rightarrow ((I = K)(L = 20) \Leftrightarrow (I = K)(L' = 20))]\end{aligned}$$

From the above condition it is clear that L is computationally nonessential in $C(\mathbf{F}_M)$ with respect to $C \wedge C_M \wedge C_B(\mathbf{F}_M)$. Thus, $C(\mathbf{F}_M)$ can be computed autonomously.

Step D:

$$\begin{aligned}\forall\ \ & I, K, L, L'\ [(I = K)(L = 20) \wedge (K > 5)(K \leq 22) \wedge (I = K)(L = 20) \\ & \wedge\ (L + 3 \geq 0)(L + 3 \leq 30)(K \geq 0)(K \leq 30)(L \geq 0)(L \leq 30) \\ & \wedge\ (I = K)(L' = 20) \wedge (K > 5)(K \leq 22) \wedge (I = K)(L' = 20) \\ & \wedge\ (L' + 3 \geq 0)(L' + 3 \leq 30)(K \geq 0)(K \leq 30)(L' \geq 0)(L' \leq 30) \\ & \quad \Rightarrow ((L + 3) = (L' + 3))]\end{aligned}$$

The above condition verifies that the expression f_J is uniquely determined by the condition $C \wedge C_M \wedge C(\mathbf{F}_M) \wedge C_B(\mathbf{F}_M)$ and the variables $\mathbf{A}^+$. Therefore, the new values of modified tuples in $v(E, d)$ are autonomously computable.

In summary, consider a numeric example for the given database scheme.

Before

r_1:

H	I
1	5
2	15
3	22
4	20

r_2:

J	K	L
19	5	20
10	15	29
16	22	20
18	20	29

$v(E, d)$:

I	J
5	20
22	16

After

r_1:

H	I
1	5
2	15
3	22
4	20

r_2:

J	K	L
19	5	20
32	15	29
23	22	20
32	20	29

$v(E, d')$:

I	J
5	20
22	23

Step A provides assurance that the tuples in the Cartesian product of r_1 and r_2, which do not satisfy C before modification, will not satisfy C after. Step B guarantees that we can determine which tuples in $v(E, d)$ to modify: the second. Step C allows us to determine which modified tuples in $v(E, d)$ will be deleted since they will no longer satisfy condition C; in this case no tuple will be deleted from the view. Step D ensures that we can compute the new values for the modified tuples. □

4.3 Conditionally autonomously computable updates

Even when an update is not unconditionally autonomously computable, for a given database instance d the information provided by the current view $v(E, d)$ and the update operation may still be sufficient to compute the updated state of the view $v(E, d')$. This type of update is called conditionally autonomously computable. The purpose of this section is to explore how far can we push the idea of autonomously computable updates if, during the process of deciding whether an update is autonomously computable, we are given access to the current contents of the view.

In this section we present necessary and sufficient conditions for determining when an update is conditionally autonomously computable. The updates considered are insertions and deletions. Work in progress covering modifications is described in Section 8.2.

4.3.1 Insertions

Consider the update operation INSERT (R_u, T), where T is a set of tuples to be inserted into relation r_u. Let a view be defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, with instance $v(E, d)$. The effect of the INSERT operation on $v(E, d)$ is conditionally autonomously computable if and only if

A. for each tuple $t_u \in T$, we can build the new tuples to be inserted into the view. In this step, the new tuples are assembled using the inserted tuple t_u and the tuples already present in the view $v(E, d)$.

B. we can prove that the new tuples generated in the previous stage represent *all* the tuples that need to be inserted into the view for the current database instance d.

Comparing these with the conditions for unconditionally autonomously computable insertions, notice that now several tuples may be inserted into the view.

Example 4.7 Consider two relation schemes $R_1(H, I)$ and $R_2(J, K, L)$ with relation instances:

r_1:

H	I
1	10
2	20
3	30

r_2:

J	K	L
10	5	25
10	6	26
20	17	31

Let a view be defined by the expression $E = (\{H, I, L\}, \{R_1, R_2\}, (I = J))$ with instance:

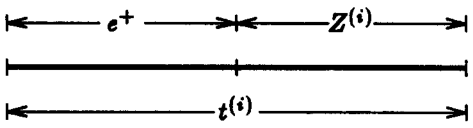
$v(E,d)$:

H	I	L
1	10	25
1	10	26
2	20	31

If relation r_1 is affected by the operation INSERT $(R_1, \{(4,10)\})$, then the tuples $(4,10,25)$ and $(4,10,26)$ will have to be inserted into $v(E,d)$ to bring it up to date. □

We now present the conditions under which it is possible to compute the new tuples that must be inserted into the view as a result of an insertion into a base relation. Theorem 4.4A corresponds to part A, and Theorem 4.4B corresponds to part B. But first, we introduce some additional notation.

Consider the current instance $v(E,d)$ of a view $E = (\mathbf{A}, \mathbf{R}, C)$, $\mathbf{R} = \{R_1, R_2, \ldots, R_m\}$. Throughout this section e denotes a tuple in $v(E,d)$ and e^+ denotes the same tuple augmented so that it is defined over the extended set $\mathbf{A}^+$. Let t_u be a new tuple inserted into relation r_u on $R_u \in \mathbf{R}$. A tuple $e \in v(E,d)$ can be extended into a tuple $t^{(i)}$ that also includes all attributes in $\mathbf{A}^+$ and $Z = \alpha(C) - \mathbf{A}^+$ by concatenating the set of variables $Z^{(i)} = \{z_1^{(i)}, z_2^{(i)}, \ldots, z_q^{(i)}\}$, $q = |Z|$, as shown below.



Let $Z_k = \alpha(R_k) \cap Z$ (the attributes from Z in R_k), $1 \leq k \leq m$, and let the instance $v(E,d)$, extended with $|Z| * |v(E,d)|$ distinct variables, be denoted by v^+. The following definition, adjusted to our notation, is from Maier [MAIER83a].

Definition 4.7 The *project-join mapping* m_E of v^+ is defined as:

$$m_E = \pi_{G_1}(v^+) \times \pi_{G_2}(v^+) \times \ldots \times \pi_{G_m}(v^+),$$

where $G_k = [\alpha(R_k) \cap \mathbf{A}^+] \cup Z_k$ (the attributes from R_k which are either visible or used in C), $1 \leq k \leq m$. □

A tuple $w \in m_E$ defines a tuple in the view $v(E,d)$ if $w[\mathbf{A}] \in v(E,d)$. A tuple $\bar{w} \in m_E$ defines a tuple *not* in the view $v(E,d)$ if $\bar{w}[\mathbf{A}] \notin v(E,d)$. The set of candidate tuples to be inserted into the view as a result of t_u being inserted into r_u is given by

$$T_u = \pi_{G_1}(v^+) \times \ldots \times \pi_{G_{u-1}}(v^+) \times \{t_u\} \times \pi_{G_{u+1}}(v^+) \times \ldots \times \pi_{G_m}(v^+).$$

To accept a tuple $t \in T_u$ into the view, we must prove that $t[\mathbf{A}] \in v(E, d')$, where d' represents the updated database instance d.

Example 4.8 Consider three relation schemes $R_1(H, I)$, $R_2(J, K, L)$, and $R_3(M, N)$, and a view $E = (\{H, I, K, L, N\}, \{R_1, R_2, R_3\}, (I > J)(L = M))$ with instance:

$v(E, d)$:

H	I	K	L	N
2	22	15	30	16
3	26	25	42	19

The extended instance v^+ and the project-join mapping m_E are given by:

v^+ :

H	I	J	K	L	M	N
2	22	$z_1^{(1)}$	15	30	$z_2^{(1)}$	16
3	26	$z_1^{(2)}$	25	42	$z_2^{(2)}$	19

m_E :

H	I	J	K	L	M	N	
2	22	$z_1^{(1)}$	15	30	$z_2^{(1)}$	16	w_1
2	22	$z_1^{(1)}$	15	30	$z_2^{(2)}$	19	$\tilde{w}_1$
2	22	$z_1^{(2)}$	25	42	$z_2^{(1)}$	16	$\tilde{w}_2$
2	22	$z_1^{(2)}$	25	42	$z_2^{(2)}$	19	$\tilde{w}_3$
3	26	$z_1^{(1)}$	15	30	$z_2^{(1)}$	16	$\tilde{w}_4$
3	26	$z_1^{(1)}$	15	30	$z_2^{(2)}$	19	$\tilde{w}_5$
3	26	$z_1^{(2)}$	25	42	$z_2^{(1)}$	16	$\tilde{w}_6$
3	26	$z_1^{(2)}$	25	42	$z_2^{(2)}$	19	w_2

Assuming that the tuple $t_u = (25, 28, 30)$ is inserted into r_2, the set T_u is given by:

$$\begin{aligned} T_u &= \pi_{HI}(v^+) \times \{t_u\} \times \pi_{MN}(v^+) \\ &= \{(2, 22, 25, 28, 30, z_2^{(1)}, 16) \\ &\quad (2, 22, 25, 28, 30, z_2^{(2)}, 19) \\ &\quad (3, 26, 25, 28, 30, z_2^{(1)}, 16) \\ &\quad (3, 26, 25, 28, 30, z_2^{(2)}, 19)\}. \end{aligned}$$

For $t = (2, 22, 25, 28, 30, z_2^{(1)}, 16)$ to influence the view, for example, we must now show that $t[\mathbf{A}] = (2, 22, 28, 30, 16)$ will be in $v(E, d')$. □

Lemma 1 *Consider the current instance $v(E,d)$ of a view $E = (\mathbf{A}, \mathbf{R}, C)$, $\mathbf{R} = \{R_1, R_2, \ldots, R_m\}$, and extend it to include the attributes in $Z = \alpha(C) - \mathbf{A}^+$. Let t be a tuple in T_u. Then $t[\mathbf{A}]$ is guaranteed to be in $v(E,d')$ if and only if*

$$\forall Z^{(1)}, \ldots, Z^{(|v|)} \; (\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow C[t]),$$

where $w_i \in m_E$ is such that $w_i[\mathbf{A}] \in v(E,d)$, $\bar{w}_j \in m_E$ is such that $\bar{w}_j[\mathbf{A}] \notin v(E,d)$, $Z^{(k)} = \{z_1^{(k)}, \ldots, z_m^{(k)}\}$, $1 \leq k \leq |v|$, and $|v| = |v(E,d)|$ represents the number of tuples in $v(E,d)$.

Proof: (Sufficiency) If the condition holds, then no matter which of the values for the sets of variables $Z^{(k)}$, $1 \leq k \leq |v|$ are chosen to make the antecedent true, the condition for inclusion will evaluate to true for t. Thus regardless of the values for unseen attributes in d, the tuple $t[\mathbf{A}]$ will belong into the view $v(E,d')$. If $Z = \emptyset$, then all attributes required to evaluate the condition C are visible and we only need to evaluate $C[t]$. Surrogate values for the missing variables Z can be obtained as explained in Section 4.1.

(Necessity) If the condition does not hold, then there exists a database instance d_1 that generates the view $v(E,d_1) = v(E,d)$ for which the new tuple $t[\mathbf{A}]$ should not be inserted. That is, because the condition does not hold, there exists a variable $z \in Z$ with a value z_1 such that $\bigwedge_i C[w_i](z) \bigwedge_j \neg C[\bar{w}_j](z) \Rightarrow C[t](z)$ is *false*. To construct the database instance d_1 we use the project-join mapping m_E for v^+ corresponding to the given view $v(E,d)$. Surrogate values are assigned to all variables $Z - \{z\}$ in m_E in such a way that $\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j]$ is *true*. We assign the value z_1 to all occurrences of z in the tuples of m_E. The database instance d_1 consists of the relations r_i formed from $\pi_{G_i}(m_E)$, $1 \leq i \leq m$, padded arbitrarily with values for attributes in $\alpha(R_i)$ that are neither visible nor used in C. Clearly, $v(E,d_1) = v(E,d)$. If we now insert the tuple t_u into r_u to obtain the updated instance d'_1, then the tuple $t[\mathbf{A}]$, $t \in T_u$, should not be inserted into the view. □

Lemma 2 *Consider the current instance $v(E,d)$ of a view $E = (\mathbf{A}, \mathbf{R}, C)$, $\mathbf{R} = \{R_1, R_2, \ldots, R_m\}$, extended to include the attributes in $Z = \alpha(C) - \mathbf{A}^+$. Let t be a tuple in T_u. Then $t[\mathbf{A}]$ is guaranteed <u>not</u> to be in $v(E,d')$, if and only if*

$$\forall Z^{(1)}, \ldots, Z^{(|v|)} \; (\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \neg C[t]),$$

where $w_i \in m_E$ is such that $w_i[\mathbf{A}] \in v(E,d)$, $\bar{w}_j \in m_E$ is such that $\bar{w}_j[\mathbf{A}] \notin v(E,d)$, $Z^{(k)} = \{z_1^{(k)}, \ldots, z_m^{(k)}\}$, $1 \leq k \leq |v|$, and $|v| = |v(E,d)|$ represents the number of tuples in $v(E,d)$.

Proof: (Sufficiency) If the condition holds, then again no matter which of the values for the sets of variables $Z^{(k)}$, $1 \leq k \leq |v|$, are present in the database, the tuple $t[\mathbf{A}]$ cannot belong into the view. If $Z = \emptyset$, then all attributes required to evaluate the condition C are visible and we only need to evaluate $\neg C[t]$. Surrogate values for the missing variables Z can be obtained as explained in Section 4.1.

(Necessity) If the condition does not hold, then there exists a database instance d_1 constructed in a similar way as in the proof of Lemma 1 that generates the view $v(E, d_1) = v(E, d)$ for which the new tuple $t[\mathbf{A}]$ should be inserted. □

Theorem 4.4A *If the insertion of the tuple t_u is conditionally autonomously computable, then every tuple $t \in T_u$ that does not satisfy the condition of Lemma 2 satisfies the condition of Lemma 1 and $t[\mathbf{A}]$ will be present in $v(E, d')$.*

Proof: Consider the contrapositive in two parts: (b) If there exists a tuple $t \in T_u$ such that t does not satisfy the condition of Lemma 2 and $t[\mathbf{A}]$ is not present in $v(E, d')$, then the insertion of t_u is not conditionally autonomously computable. The proof of Lemma 2 is a proof of part (b). For part (a), in the same way as in the proofs of Lemmas 1 and 2, we can construct two database instances d_1 and d_2 such that $v(E, d_1) = v(E, d_2) = v(E, d)$, and such that the insertion of t_u into r_u of d_1 (d_2) will cause $t[\mathbf{A}]$ not to be accepted (rejected). Consequently, we cannot decide whether $t[\mathbf{A}]$ should be inserted without further information about the database instance. □

Note that if the conditions of Lemmas 1 or 2 do not hold, then we cannot be sure whether a newly generated tuple $t \in T_u$ constructed using surrogate values for the missing variables in Z, include values that actually exist in the database. Also, if $Z = \emptyset$ (that is, all variables used in the condition C are visible in the view), the test of these conditions reduces to evaluating $C[t]$ for each $t \in T_u$.

Some comments regarding the size of the expressions resulting from the conditions given by Lemma 1 and Lemma 2 are necessary. The size of the antecedent of the conditions of Lemma 1 and Lemma 2, namely $\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j]$, is $|v(E, d)|^m |C|$, where $|v(E, d)|$ denotes the number of tuples in the view, m is the number of different relation schemes in the view definition, and $|C|$ denotes the number of atomic terms in the condition C. However, not all of the Boolean expressions derived from C in the antecedent refer to free variables in the consequent $C[t]$. In Example 4.8, for instance, the variables $z_1^{(i)}$ for attribute J are not free in $C[t]$; satisfying the condition $(I > J)$ in w_i has no effect on $C[t]$. Hence, we want to know how many Boolean expressions derived from C must be included in the antecedent to be able to test the conditions of Lemma 1 and Lemma 2 correctly.

Assuming that the consequent $C[t]$ refers to free variables from k distinct relation schemes, we want to know how many rows $w, \bar{w} \in m_E$ refer to these free variables. Such rows represent exactly the conditions $C[w]$, $\neg C[\bar{w}]$ from the expression $\bigwedge_i C[w_i] \bigwedge_j C[\bar{w}_j]$ that must be tested. To solve this, consider the problem of counting arrangements of size k. Each position of an arrangement may be occupied by one of b different elements. In our case $b = |v(E, d)|$. The total number of arrangements is b^k. We are given the arrangement $b_1, b_2, \ldots, b_k$ and we are asked to find the total number of arrangements that contain the element b_1 in position 1 or the element b_2 in position 2 and so on until position k. This is given by $b^k - (b-1)^k$. The inserted tuple t_u takes the place of the variables in relation scheme R_u. We also have to count all different rows of m_E which are associated with the remaining $m - k$ relation schemes. Thus, the number of Boolean expressions $C[w]$ in the antecedent of the conditions given by Lemma 1 or Lemma 2 is:

$$(|v(E,d)|)^{m-k}\left[(|v(E,d)|)^{k}-(|v(E,d)|-1)^{k}\right].$$

where $(|v(E,d)|)^{m-k}$ is the number of tuples in the project-join mapping of those relations containing none of the free variables in $\mathcal{C}[t]$.

Notice also that the condition of Lemma 1 can be simplified by eliminating the condition $\bigwedge_j \neg\mathcal{C}[\bar{w}_j]$ from the antecedent, still resulting in a sufficient condition for the acceptance of the tuple t. If the condition $\bigwedge_i \mathcal{C}[w_i] \Rightarrow \mathcal{C}[t]$ holds, then the tuple t can be safely accepted. However, if the condition does not hold, then it may still be the case that t could be safely accepted if we use the information provided by the expression $\bigwedge_j \neg\mathcal{C}[\bar{w}_j]$. Similarly, the condition of Lemma 2 can be simplified, to obtain a sufficient condition for the rejection of tuple t.

Theorem 4.4A leads to an algorithm to assemble the new tuples to be inserted into the view as indicated in part A at the beginning of this section. The algorithm is represented by the following steps.

1. Compute the sets T_u and m_E.
2. The set of tuples T_u may be reduced by retaining only one copy of the tuples that agree on all attributes $\mathbf{A}^+ \cup \alpha(R_u)$.
3. For each $t \in T_u$, if the condition of Lemma 1 holds, then keep t in T_u, else if the condition of Lemma 2 holds, then remove t from T_u, else terminate unsuccessfully.
4. The tuples to be inserted into the view are $\pi_{\mathbf{A}}(T_u)$.

Notice that Step 3 could be made more efficient by first discarding all tuples $t \in T_u$ such that $\mathcal{C}[t]$ evaluates to *false*. For an illustration of the algorithm refer to Step A in Example 4.10 on page 48.

The above algorithm indicates how to compute a set of new tuples to be inserted into the view as a result of inserting one tuple into relation r_u. How can we make sure that these are *all* the tuples that must be inserted to bring the view up-to-date?

The rest of this subsection presents a necessary and sufficient condition for testing whether the set of tuples T_u generated by the above algorithm represent all tuples that must be inserted into the view as a result of inserting the tuple t_u into r_u.

To achieve this purpose, we need a notion which will allow us to prove that all values for variables in $\alpha(\mathbf{R} - \{R_u\}) \cap \alpha(\mathcal{C})$ which can potentially interact with the values from the tuple t_u to produce a new insertion into a view $v(E,d)$ are somehow contained in the view itself. This notion is called *coverage* and is stated in the following definition.

Definition 4.8 Consider a view definition $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$, where $\mathbf{R} = \{R_1, R_2, \ldots, R_m\}$, and the update operation INSERT $(R_u, \{t_u\})$, $R_u \in \mathbf{R}$.

Let $v(E,d)$ be the current instance of the view. Let $Y_l = \alpha(R_l) \cap \alpha(C) \cap \mathbf{A}^+$ (i.e., the attributes from relation R_l that participate in C which are visible or uniquely determined by the view), and $Z_l = [\alpha(R_l) \cap \alpha(C)] - \mathbf{A}^+$ (i.e., the attributes from relation R_l that participate in C which are not visible nor uniquely determined by the view), $1 \leq l \leq m$, $l \neq u$. The variables $\bigcup_{l=1, l\neq u}^{l=m}(Y_l \cup Z_l)$ are said to be *covered* in the instance $v(E,d)$ with respect to the insertion of the tuple t_u into r_u if the following condition holds:

$$
\begin{array}{l}
\forall \; Y_1, Z_1, \ldots, Y_{u-1}, Z_{u-1}, Y_{u+1}, Z_{u+1}, \ldots, Y_m, Z_m \\
\quad [C[t_u](Y_1, Z_1, \ldots, Y_{u-1}, Z_{u-1}, Y_{u+1}, Z_{u+1}, \ldots, Y_m, Z_m) \Rightarrow \\
\quad\; \{\forall\, z_1^{(1)}, \ldots, z_m^{(1)}, \ldots, z_1^{(|v|)}, \ldots, z_m^{(|v|)} \\
\quad\quad \bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \bigvee_i C[w_i \setminus R_1](Y_1, Z_1)\} \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \wedge \\
\quad\; \{\forall\, z_1^{(1)}, \ldots, z_m^{(1)}, \ldots, z_1^{(|v|)}, \ldots, z_m^{(|v|)} \\
\quad\quad \bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \bigvee_i C[w_i \setminus R_{u-1}](Y_{u-1}, Z_{u-1})\} \wedge \\
\quad\; \{\forall\, z_1^{(1)}, \ldots, z_m^{(1)}, \ldots, z_1^{(|v|)}, \ldots, z_m^{(|v|)} \\
\quad\quad \bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \bigvee_i C[w_i \setminus R_{u+1}](Y_{u+1}, Z_{u+1})\} \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \wedge \\
\quad\; \{\forall\, z_1^{(1)}, \ldots, z_m^{(1)}, \ldots, z_1^{(|v|)}, \ldots, z_m^{(|v|)} \\
\quad\quad \bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \bigvee_i C[w_i \setminus R_m](Y_m, Z_m)\}],
\end{array}
\tag{4.1}
$$

where $w_i, \bar{w}_j \in m_E$, $C[w_i \backslash R_l]$ denotes the substitution of values from w_i for variables in C except for those in scheme R_l, $z_l^{(\cdot)}$ denotes variables in $[\alpha(R_l) \cap \alpha(C)] - \mathbf{A}^+$, $1 \leq l \leq m$, and $|v| = |v(E,d)|$ denotes the number of tuples in $v(E,d)$. We use R_l instead of $\alpha(R_l)$ in $C[w_i \setminus R_l]$ to simplify notation. □

In other words, the variables $\bigcup_{l=1, l\neq u}^{l=m}(Y_l \cup Z_l)$ are covered in the instance $v(E,d)$ with respect to the insertion of the tuple t_u into r_u if every tuple $t_i \in r_i$, $1 \leq i \leq m$ and $i \neq u$, that can possibly combine with t_u to generate a new insertion t into $v(E,d)$ (i.e., $C[t_1 \times \cdots \times t_{u-1} \times t_u \times t_{u+1} \times \cdots \times t_m] = \mathit{true}$) is already present in the view. That is, $t_i = e[\alpha(R_i)]$, $1 \leq i \leq m$, $i \neq u$, and $e \in v(E,d)$. The following example illustrates the above definition.

Example 4.9 Consider three relation schemes $R_1(H, I)$, $R_2(J, K, L)$, and $R_3(M, N)$ with instances r_1, r_2, and r_3, respectively.

r_1:

H	I
1	20
2	22
3	23
4	25

r_2:

J	K	L
19	14	5
22	15	10
23	50	30

r_3:

M	N
5	1
10	2
30	4

i) First consider the view defined by the expression

$$E_1 = (\{H, I, J, K, L\}, \{R_1, R_2\}, (I < J))$$

with instance $v(E_1, d)$ shown below:

$v(E_1, d)$:

H	I	J	K	L
1	20	22	15	10
1	20	23	50	30
2	22	23	50	30

m_{E_1} :

H	I	J	K	L	
1	20	22	15	10	w_1
1	20	23	50	30	w_2
1	20	23	50	30	w_3
1	20	22	15	10	w_4
1	20	23	50	30	w_5
1	20	23	50	30	w_6
2	22	22	15	10	$\bar{w}_1$
2	22	23	50	30	w_7
2	22	23	50	30	w_8

Suppose that the update operation INSERT $(R_1, \{(4, 21)\})$ is applied to relation r_1. Then we can see that $Y_2 = \{J\}$ and $Z_2 = \emptyset$. To be sure that the set of tuples $T_u = \{(4, 21, 22, 15, 10), (4, 21, 23, 50, 30)\}$ are all the tuples that must be inserted into $v(E, d)$ as a result of the insertion of the tuple $(4, 21)$ into r_1, we need to prove that the variables Y_2 are covered. That is, we must prove the validity of the following condition:

$$\begin{aligned} &\forall \; Y_2 \; (C[t_u](Y_2) \Rightarrow \\ &\quad \{\forall \, z_1^{(1)}, \ldots, z_m^{(1)}, \ldots, z_1^{(|v|)}, \ldots, z_m^{(|v|)} \\ &\quad \textstyle\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \bigvee_i C[w_i \setminus R_2](Y_2)\}] \end{aligned}$$

Since the antecedent $\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] = \mathit{true}$ in this example, the condition simplifies to:

$$\forall \; Y_2 \; (C[t_u](Y_2) \; \Rightarrow \; \textstyle\bigvee_i C[w_i \setminus R_2](Y_2)).$$

This is equivalent to testing the condition:

$$\begin{aligned} \forall \;\; J \; [(21 < J) \; \Rightarrow \; & (20 < J) \vee (20 < J) \vee (20 < J) \vee (20 < J) \\ & \vee (20 < J) \vee (20 < J) \vee (22 < J) \vee (22 < J)]. \end{aligned}$$

Since the implication is valid we can then conclude that the variable J is covered in the instance $v(E_1, d)$ with respect to the given insert operation. That is, there cannot be a tuple in r_2 such that the J value matches the newly inserted tuple but does not match any tuple already present in the view.

ii) Now consider the view $E_2 = (\{H, J, K, L\}, \{R_1, R_2\}, (I < J))$ which is exactly the same view as E_1 except that attribute I is not visible. The project-join mapping for E_2 is given by:

m_{E_2} :

H	I	J	K	L	
1	$z^{(1)}$	22	15	10	w_1
1	$z^{(1)}$	23	50	30	w_2
1	$z^{(1)}$	23	50	30	w_3
1	$z^{(2)}$	22	15	10	w_4
1	$z^{(2)}$	23	50	30	w_5
1	$z^{(2)}$	23	50	30	w_6
2	$z^{(3)}$	22	15	10	$\bar{w}_1$
2	$z^{(3)}$	23	50	30	w_7
2	$z^{(3)}$	23	50	30	w_8

In this case $Y_1 = \emptyset$, $Z_1 = \{I\}$, $Y_2 = \{J\}$, and $Z_2 = \emptyset$. Again, to verify whether the variables in Y_2 are covered we need to test the condition:

$$\begin{aligned} &\forall \;\; Y_2\ (C[t_u](Y_2) \Rightarrow \\ &\quad (\forall\, z^{(1)}, z^{(2)}, z^{(3)}\ \textstyle\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \;\Rightarrow\; \bigvee_i C[w_i \setminus R_2](Y_2)). \end{aligned}$$

This is equivalent to testing the condition:

$$\begin{aligned} &\forall \;\; J\ [(21 < J) \Rightarrow \\ &\quad (\forall\, z^{(1)}, z^{(2)}, z^{(3)}\ (z^{(1)} < 22)(z^{(1)} < 23)(z^{(1)} < 23) \\ &\qquad\qquad\qquad\qquad (z^{(2)} < 22)(z^{(2)} < 23)(z^{(2)} < 23) \\ &\qquad\qquad\qquad\qquad (z^{(3)} \geq 22)(z^{(3)} < 23)(z^{(3)} < 23) \\ &\qquad \Rightarrow\; (z^{(1)} < J) \vee (z^{(2)} < J) \vee (z^{(3)} < J)]. \end{aligned}$$

Once again the implication holds and we can conclude that the variable J is covered in the instance $v(E_2, d)$ with respect to the given insert operation. Therefore, the set of tuples T_u represent all tuples that should be inserted into the view as a result of the given update. Note that for any inserted tuple $t_u = (h, i)$ having a value $i < 21$ the variable J will not be covered in the instance $v(E_2, d)$.

iii) Finally, consider the view

$$E_3 = (\{H, I, J, K, L, M, N\}, \{R_1, R_2, R_3\}, (I = J) \wedge (L = M))$$

with instance:

$v(E_3, d)$:

H	I	J	K	L	M	N
2	22	22	15	10	10	2
3	23	23	50	30	30	4

The corresponding project-join mapping is give by:

m_{E_2}:

H	I	J	K	L	M	N	
2	22	22	15	10	10	2	w_1
2	22	22	15	10	30	4	$\bar{w}_1$
2	22	23	50	30	10	2	$\bar{w}_2$
2	22	23	50	30	30	4	$\bar{w}_3$
3	23	22	15	10	10	2	$\bar{w}_4$
3	23	22	15	10	30	4	$\bar{w}_5$
3	23	23	50	30	10	2	$\bar{w}_6$
3	23	23	50	30	30	4	w_2

Suppose that the operation INSERT $(R_2, \{(22, 60, 30)\})$ is applied to relation r_2. Then we can verify that $Y_1 = \{I\}$, $Z_1 = \emptyset$, $Y_3 = \{M\}$, and $Z_3 = \emptyset$. To be sure that the set $T_u = \{(2, 22, 60, 30, 30, 4)\}$ are all the tuple to be inserted into $v(E_3, d)$ as a result of the given insert operation, we need to prove that the variables in $Y_1 \cup Y_3$ are covered. That is, we need to prove the validity of the condition:

$$\forall \quad Y_1, Y_3\ (C[t_u](Y_1, Y_3) \Rightarrow \\ (\bigvee_i C[w_i \setminus R_1](Y_1) \wedge \bigvee_i C[w_i \setminus R_3](Y_3))).$$

This is equivalent to testing the condition:

$$\forall \quad I, M\ [(I = 22)(30 = M) \Rightarrow \\ ((I = 22)(10 = 10) \vee (I = 23)(30 = 30)) \\ \wedge((22 = 22)(10 = M) \vee (23 = 23)(30 = M))].$$

Since the implication is valid we conclude that the variables I and M are covered in the view $v(E_3, d)$ with respect to the given insert operation. For $t_u = (j, k, l)$, I is not covered if $j \notin \{22, 23\}$ and M is not covered if $j \notin \{10, 30\}$. □

Theorem 4.4B *Consider a view definition $E = (\mathbf{A}, \mathbf{R}, C)$ with instance $v(E, d)$ and the operation INSERT $(R_u, \{t_u\})$. The set of tuples T_u to be inserted into $v(E, d)$ as a result of the insert operation represent all tuples that must be inserted into the view if and only if all variables in $\bigcup_{l=1, l\neq u}^{l=m}(Y_l \cup Z_l)$, $Y_l = \alpha(R_l) \cap \alpha(C) \cap \mathbf{A}^+$ and $Z_l = [\alpha(R_l) \cap \alpha(C)] - \mathbf{A}^+$, are covered in the instance $v(E, d)$ with respect to the insertion.*

Proof: (Sufficiency) If the condition for coverage holds, then it is guaranteed that any combination of values for the variables in $\bigcup_{l=1, l\neq u}^{l=m}(Y_l \cup Z_l)$ that make the condition

$$C[t_u](Y_1, Z_1, \ldots, Y_{u-1}, Z_{u-1}, Y_{u+1}, Z_{u+1}, \ldots, Y_m, Z_m)$$

hold, will be present as part of some subset of tuples currently stored in $v(E, d)$. Each of the conjuncts in the consequent of Condition (4.1) of Definition 4.8, that is,

$$\forall \quad z_1^{(1)}, \ldots, z_m^{(1)}, \ldots, z_1^{(|v|)}, \ldots, z_m^{(|v|)} \\ \bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \bigvee_i C[w_i \setminus R_l](Y_l, Z_l),$$

assures that all combinations of values for the variables Y_l, Z_l which when combined with the inserted tuple t_u will make the antecedent of Condition (4.1) evaluate to true (i.e., creating a new insertion into the view) are already represented in the view. Building the set T_u involves generating all tuples that result from combining t_u with the tuples in $v(E, d)$. Hence T_u will contain all tuples that must be inserted into the view as a result of inserting t_u into r_u.

(Necessity) If the condition for coverage does not hold, then there is at least one variable $z \in \bigcup_{l=1, l \neq u}^{l=m} (Y_l \cup Z_l)$ with value z_1 such that $C[t_u](z_1) = \mathit{true}$ which is not available in any tuple currently stored in the view $v(E, d)$. The value z_1 causes a conjunct of the consequent of Condition (4.1) to be false. Assume that this conjunct is given by

$$\forall \quad z_1^{(1)}, \ldots, z_m^{(1)}, \ldots, z_1^{(|v|)}, \ldots, z_m^{(|v|)} \\ \bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \bigvee_i C[w_i \setminus R_h](z),$$

where $z \in \alpha(R_h)$. We can then construct a database instance $d_1 = \{r_1 \ldots, r_m\}$ such that $v(E, d_1) = v(E, d)$, where database instance d_1 contains the value z_1 in some tuple of relation r_h. To construct the database instance d_1 we use the project-join mapping m_E for v^+ corresponding to the given view $v(E, d)$. Surrogate values are assigned to the variable z in tuples from m_E in such a way that $\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j]$ is true. Relations r_l, $1 \leq l \leq m$, are assigned the set of tuples $\pi_{\alpha(R_l)}(m_E)$. In addition, $r_h = r_h \cup \{t'\}$ where $t'[z] = z_1$ and $t'[x] = w_i[x]$, $x \in \alpha(R_h) - \{z\}$, for some $w_i \in m_E$. Although $t[\mathbf{A}]$ should be inserted into the view, t will not be represented in T_u. □

Corollary 4.4 *The effect of inserting the tuple t_u into r_u on the view is conditionally autonomously computable with respect to the view instance $v(E, d)$ if and only if the conditions of Theorems 4.4A and 4.4B hold.* □

We illustrate the full algorithm resulting from Theorems 4.4A and 4.4B regarding conditionally autonomously computable insertions through an example.

Example 4.10 Consider two relation schemes $R_1(H, I)$, $R_2(J, K, L)$, and their corresponding relation instances:

r_1:

H	I
1	18
2	24
3	28
4	25
5	21

r_2:

J	K	L
17	14	11
21	15	30
26	50	23
3	25	42

Consider a view defined by

$$E = (\{H, I, J, K\}, \{R_1, R_2\}, [(H = 2)(I < J) \vee (H = 5)(I > L) \vee (I = K)])$$

and its corresponding instance:

$v(E, d)$:

H	I	J	K
2	24	26	50
5	21	17	14
4	25	3	25

Assume the following update is applied to the database: INSERT $(R_1, \{(2, 25)\})$.

Step A: First, we build v^+, m_E, and T_u.

v^+:

H	I	J	K	L
2	24	26	50	$z_1^{(1)}$
4	25	3	25	$z_1^{(2)}$
5	21	17	14	$z_1^{(3)}$

m_E:

H	I	J	K	L	
2	24	26	50	$z_1^{(1)}$	w_1
2	24	3	25	$z_1^{(2)}$	$\tilde{w}_1$
2	24	17	14	$z_1^{(3)}$	$\tilde{w}_2$
4	25	26	50	$z_1^{(1)}$	$\tilde{w}_3$
4	25	3	25	$z_1^{(2)}$	w_2
4	25	17	14	$z_1^{(3)}$	$\tilde{w}_4$
5	21	26	50	$z_1^{(1)}$	$\tilde{w}_5$
5	21	3	25	$z_1^{(2)}$	$\tilde{w}_6$
5	21	17	14	$z_1^{(3)}$	w_3

T_u:

H	I	J	K	L
2	25	26	50	$z_1^{(1)}$
2	25	3	25	$z_1^{(2)}$
2	25	17	14	$z_1^{(3)}$

The first tuple $t = (2, 25, 26, 50, z_1^{(1)})$ in T_u can be safely accepted because the condition of Lemma 1, shown below, holds.

$$\begin{array}{lll}
\forall & z_1^{(1)} & \\
 & [((2=2)(24<26) \vee (2=5)(24>z_1^{(1)}) \vee (24=50)) & (\text{from } w_1) \\
\wedge & \neg((4=2)(25<26) \vee (4=5)(25>z_1^{(1)}) \vee (25=50)) & (\text{from } \bar{w}_3) \\
\wedge & \neg((5=2)(21<26) \vee (5=5)(21>z_1^{(1)}) \vee (21=50)) & (\text{from } \bar{w}_5) \\
\Rightarrow & (2=2)(25<26) \vee (2=5)(25>z_1^{(1)}) \vee (25=50)] &
\end{array}$$

For the second tuple $t = (2,25,3,25,z_1^{(2)})$ in T_u we also test the condition of Lemma 1.

$$\begin{array}{lll}
\forall & z_1^{(2)} & \\
 & [\neg((2=2)(24<3) \vee (2=5)(24>z_1^{(2)}) \vee (24=25)) & (\text{from } \bar{w}_1) \\
\wedge & ((4=2)(25<3) \vee (4=5)(25>z_1^{(2)}) \vee (25=25)) & (\text{from } w_2) \\
\wedge & \neg((5=2)(21<3) \vee (5=5)(21>z_1^{(2)}) \vee (21=25)) & (\text{from } \bar{w}_6) \\
\Rightarrow & (2=2)(25<3) \vee (2=5)(25>z_1^{(2)}) \vee (25=25)] &
\end{array}$$

Clearly, the above condition holds, and the tuple $t = (2,25,3,25,z_1^{(2)})$ in T_u can be safely accepted into the view.

Finally, for the third tuple $t = (2,25,17,14,z_1^{(3)})$ in T_u we test the condition of Lemma 2.

$$\begin{array}{lll}
\forall & z_1^{(3)} & \\
 & [\neg((2=2)(24<17) \vee (2=5)(24>z_1^{(3)}) \vee (24=14)) & (\text{from } \bar{w}_2) \\
\wedge & \neg((4=2)(25<17) \vee (4=5)(25>z_1^{(3)}) \vee (25=14)) & (\text{from } \bar{w}_4) \\
\wedge & ((5=2)(21<17) \vee (5=5)(21>z_1^{(3)}) \vee (21=14)) & (\text{from } w_3) \\
\Rightarrow & \neg((2=2)(25<17) \vee (5=2)(25>z_1^{(3)}) \vee (25=14))] &
\end{array}$$

Clearly, the above condition holds, and the tuple $t = (2,25,17,14,z_1^{(3)})$ in T_u can be safely rejected from the view.

Step B: Here we are interested in finding out whether the new tuples assembled in Step A represent all tuples that need to be inserted into the view.

We can verify that in this example $Y_2 = \{J, K\}$ and $Z_2 = \{L\}$. Testing whether the variables $Y_2 \cup Z_2$ are covered requires testing the following condition:

$$\begin{array}{ll}
\forall & Y_2, Z_2\ (C[t_u](Y_2, Z_2) \Rightarrow \\
 & (\forall\, z^{(1)}, z^{(2)}, z^{(3)}\ \bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \bigvee_i C[w_i \setminus R_2](Y_2, Z_2)).
\end{array}$$

This is equivalent to testing the condition:

$$
\begin{aligned}
\forall\ & J,K,L\ ([(2=2)(25<J)\vee(2=5)(25>L)\vee(25=K) \Rightarrow \\
& (\forall\, z^{(1)},z^{(2)},z^{(3)} \\
& \quad ((2=2)(24<26)\vee(2=5)(24>z_1^{(1)})\vee(24=50)) \\
& \quad \wedge((4=2)(25<3)\vee(4=5)(24>z_1^{(2)})\vee(25=25)) \\
& \quad \wedge((5=2)(21<17)\vee(5=5)(21>z_1^{(3)})\vee(21=14)) \\
& \quad \wedge\neg((2=2)(24<3)\vee(2=5)(24>z_1^{(2)})\vee(24=25)) \\
& \quad \wedge\neg((2=2)(24<17)\vee(2=5)(24>z_1^{(3)})\vee(24=14)) \\
& \quad \wedge\neg((4=2)(25<26)\vee(4=5)(25>z_1^{(1)})\vee(25=50)) \\
& \quad \wedge\neg((4=2)(25<17)\vee(4=5)(25>z_1^{(3)})\vee(25=14)) \\
& \quad \wedge\neg((5=2)(21<26)\vee(5=5)(21>z_1^{(1)})\vee(21=50)) \\
& \quad \wedge\neg((5=2)(21<3)\vee(5=5)(21>z_1^{(2)})\vee(21=25)) \\
& \quad \Rightarrow\ (2=2)(24<J)\vee(2=5)(24>L)\vee(24=K) \\
& \quad \wedge\ (4=2)(25<J)\vee(4=5)(25>L)\vee(25=K) \\
& \quad \wedge\ (5=2)(21<J)\vee(5=5)(21>L)\vee(21=K))].
\end{aligned}
$$

The implication is valid, and therefore, attributes J, K, and L are covered in the view with respect to the given insert operation. Hence, the set of tuples T_u generated at Step A as a result of the insertion of the tuple $(2,25)$ into r_1 are all tuples that need to be inserted into $v(E,d)$ to bring it up to date. □

Performing conditionally autonomously computable insertions is in general expensive because the number of terms of the Boolean expressions that have to be tested are typically exponential in the number of tuples of the view. However, not surprisingly, conditionally autonomously computable insertions are easier to perform if the attributes that participate in the condition defining the view are visible in the view (i.e., $\alpha(C) \subseteq \mathbf{A}$).

4.3.2 Deletions

Unconditionally autonomously computable deletions require that all variables in the conditions C_D and C, which are not seen in the view, be computationally nonessential in C_D with respect to C. This condition should hold for every possible combination of values from the domains of the variables in $[\alpha(C_D)\cup\alpha(C)]-\mathbf{A}^+$.

In some situations, even though a variable in $[\alpha(C_D)\cup\alpha(C)]-\mathbf{A}^+$ is computationally essential in C_D with respect to C, it may still be possible to compute the delete operation autonomously.

Example 4.11 Consider again the relation schemes $R_1(H,I)$ and $R_2(J,K,L)$, with corresponding instances

r_1:

H	I
1	16
2	20
3	24

r_2:

J	K	L
17	18	23
19	14	25
24	18	33

Let a view be defined by $E = (\{H, J, K, L\}, \{R_1, R_2\}, (J < I)(I < L))$, and thus its corresponding instance is

$v(E, d)$:

H	J	K	L
2	17	18	23
2	19	14	25
3	19	14	25

Now consider a delete operation DELETE $(R_1, (I < 20) \vee (I > 22))$. Even though I is computationally essential in $C_D \equiv (I < 20) \vee (I > 22)$ with respect to $C \equiv (J < I)(I < L)$, we can still determine from the tuples present in the view as well as the absence of tuple $(3, 17, 18, 23)$ that the third tuple in $v(E, d)$ is precisely the one that needs to be deleted. □

Lemma 3 *Consider the current instance $v(E, d)$ of a view $E = (\mathbf{A}, \mathbf{R}, C)$, $\mathbf{R} = \{R_1, R_2, \ldots, R_m\}$, extended to include the attributes in $Z = \alpha(C) - \mathbf{A}^+$, and the update operation DELETE (R_u, C_D). Then $t \in v^+$ is guaranteed to <u>stay</u> in the view $v(E, d')$ if and only if*

$$\forall\, Z^{(1)}, \ldots, Z^{(|v|)}\ (\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow \neg C_D[t]),$$

where $w_i \in m_E$ is such that $w_i[\mathbf{A}] \in v(E, d)$, $\bar{w}_j \in m_E$ is such that $\bar{w}_j[\mathbf{A}] \notin v(E, d)$, $Z^{(k)} = \{z_1^{(k)}, \ldots, z_m^{(k)}\}$, $1 \leq k \leq |v|$, and $|v| = |v(E, d)|$ represents the number of tuples in $v(E, d)$.

Proof: (Sufficiency) If the condition holds, then no matter which of the values for the sets of variables $Z^{(k)}$, $1 \leq k \leq |v|$ are chosen to make the antecedent true, the tuple $t[\mathbf{A}]$ will belong into the view. If $Z = \emptyset$, then all attributes required to evaluate the condition C_D are visible and we only need to evaluate $C_D[t]$. Surrogate values for the missing variables Z can be obtained as explained in Section 4.1.

(Necessity) If the condition does not hold, then there exists a database instance d_1 that generates the view $v(E, d_1) = v(E, d)$ for which the tuple $t[\mathbf{A}]$ should be deleted. Because the condition does not hold, there exists a variable $z \in Z$ with a value z_1 such that $\bigwedge_i C[w_i](z) \bigwedge_j \neg C[\bar{w}_j](z) \Rightarrow \neg C_D[t](z)$ is *false*. To construct the database instance d_1 we use the project-join mapping m_E for v^+ corresponding to the given view $v(E, d)$. Surrogate values are assigned to all variables $Z - \{z\}$ in m_E in such a way that $\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j]$ is *true*. We assign the value z_1 to all occurrences of z in the tuples of m_E. The database instance d_1 consists of the relations $r_i = \pi_{G_i}(m_E)$, $1 \le i \le m$. Clearly, $v(E, d_1) = v(E, d)$. If we now apply the update to relation r_u to obtain the updated instance d_1', then the tuple $t[\mathbf{A}]$, $t \in v^+$, should be deleted from the view. □

Lemma 4 *Consider the current instance $v(E, d)$ of a view $E = (\mathbf{A}, \mathbf{R}, C)$, $\mathbf{R} = \{R_1, R_2, \ldots, R_m\}$, extended to include the attributes in $Z = \alpha(C) - \mathbf{A}^+$, and the update operation DELETE (R_u, C_D). Then $t \in v^+$ is guaranteed to be deleted from the view $v(E, d')$ if and only if*

$$\forall\, Z^{(1)}, \ldots, Z^{(|v|)}\ (\bigwedge_i C[w_i] \bigwedge_j \neg C[\bar{w}_j] \Rightarrow C_D[t]),$$

where $w_i \in m_E$ is such that $w_i[\mathbf{A}] \in v(E, d)$, $\bar{w}_j \in m_E$ is such that $\bar{w}_j[\mathbf{A}] \notin v(E, d)$, $Z^{(k)} = \{z_1^{(k)}, \ldots, z_m^{(k)}\}$, $1 \le k \le |v|$, and $|v| = |v(E, d)|$ represents the number of tuples in $v(E, d)$.

Proof: (Sufficiency) If the condition holds, then no matter which of the values for the sets of variables $Z^{(k)}$, $1 \le k \le |v|$ are chosen to make the antecedent true, the tuple $t[\mathbf{A}]$ cannot belong into the view. If $Z = \emptyset$, then all attributes required to evaluate the condition C_D are visible and we only need to evaluate $C_D[t]$. Surrogate values for the missing variables Z can be obtained as explained in Section 4.1.

(Necessity) If the condition does not hold, then there exists a database instance d_1 constructed in a similar way as in the proof of Lemma 3 that generates the view $v(E, d_1) = v(E, d)$ for which the new tuple $t[\mathbf{A}]$ should not be deleted. □

Theorem 4.5 *If a delete operation is conditionally autonomously computable, then every tuple $t \in v^+$ that does not satisfy the condition of Lemma 3 satisfies the condition of Lemma 4 and $t[\mathbf{A}]$ will not be present in $v(E, d')$.*

Proof: Consider the contrapositive in two parts: (a) if there exists a tuple $t \in v^+$ that does not satisfy the condition of Lemma 3 and $t[\mathbf{A}] \in v(E, d')$, then the deletion is not conditionally autonomously computable. The proof of Lemma 3 is a proof of part (b). For part (a), in the same way as in the proofs of Lemmas 4 and 5, we can construct two database instances d_1 and d_2 such that $v(E, d_1) = v(E, d_2) = v(E, d)$, and such that the delete operation on r_u of d_1 (d_2) will cause $t[\mathbf{A}]$ not to stay (be deleted). Consequently, we cannot decide whether $t[\mathbf{A}]$ should be deleted without further information about the database instance. □

Example 4.12 Consider Example 4.11 on page 51. Let a view be defined by the expression E with instance $v(E, d)$. The extended instance v^+ and project-join mapping are given by

v^+ :

H	I	J	K	L
2	$z_1^{(1)}$	17	18	23
2	$z_1^{(2)}$	19	14	25
3	$z_1^{(3)}$	19	14	25

m_E :

H	I	J	K	L	
2	$z_1^{(1)}$	17	18	23	w_1
2	$z_1^{(1)}$	19	14	25	w_2
2	$z_1^{(1)}$	19	14	25	w_3
2	$z_1^{(2)}$	17	18	23	w_4
2	$z_1^{(2)}$	19	14	25	w_5
2	$z_1^{(2)}$	19	14	25	w_6
3	$z_1^{(3)}$	17	18	23	$\bar{w}_1$
3	$z_1^{(3)}$	19	14	25	w_7
3	$z_1^{(3)}$	19	14	25	w_8

Now consider a delete operation DELETE $(R_1, (I < 20) \vee (I > 22))$. For the first tuple $t_1 = (2, z_1^{(1)}, 17, 18, 23) \in v^+$ we find that the condition of Lemma 3, given below, holds.

$$\begin{array}{lll} \forall & z_1^{(1)} & \\ & [(z_1^{(1)} > 17)(z_1^{(1)} < 23) & \text{(from } w_1) \\ \wedge & (z_1^{(1)} > 19)(z_1^{(1)} < 25) & \text{(from } w_1, w_3) \\ \Rightarrow & \neg((z_1^{(1)} < 20) \vee (z_1^{(1)} > 22))] & \end{array}$$

holds, thus tuple $t_1[\mathbf{A}]$ should not be deleted from the view.

For the second tuple $t_2 = (2, z_1^{(2)}, 19, 14, 25) \in v^+$ we find that the condition of Lemma 3, given below, also holds.

$$\begin{array}{lll} \forall & z_1^{(2)} & \\ & [(z_1^{(2)} > 17)(z_1^{(2)} < 23) & \text{(from } w_4) \\ \wedge & (z_1^{(2)} > 19)(z_1^{(2)} < 25) & \text{(from } w_5, w_6) \\ \Rightarrow & \neg((z_1^{(2)} < 20) \vee (z_1^{(2)} > 22))] & \end{array}$$

Thus tuple $t_2[\mathbf{A}]$ should not be deleted from the view.

Finally, for the third tuple $t_3 = (3, z_1^{(3)}, 19, 14, 25) \in v^+$ we find that the condition of Lemma 4, given below, holds.

$$\begin{array}{lll} \forall & z_1^{(3)} & \\ & [(z_1^{(3)} > 19)(z_1^{(3)} < 25) & \text{(from } w_7, w_8) \\ \wedge & \neg[(z_1^{(3)} > 17)(z_1^{(3)} < 23)] & \text{(from } \bar{w}_1) \\ \Rightarrow & (z_1^{(3)} < 20) \vee (z_1^{(3)} > 22)] & \end{array}$$

Thus tuple $t_3[\mathbf{A}]$ should be deleted from the view. □

The conditions given by Theorem 4.5 to determine whether a delete operation is conditionally autonomously computable are necessary and sufficient. Testing this condition is likely to be expensive. However, if all the attributes of the condition $\mathcal{C}_D$ are visible in the view, then the test reduces to evaluating $\mathcal{C}_D[e]$ for each $e \in v(E, d)$.

4.4 Summary

This chapter has dealt with the problems of: (a) establishing the conditions to detect when the effect of an update to a base relation is autonomously computable in a view, as well as (b) describing the procedures to carry out the update based on the information provided by the update to the base relation, the expression defining the view, and the contents of the view instance. We have established the difference between unconditionally autonomously computable updates (i.e., those that can be performed regardless of the database instance) and conditionally autonomously computable updates (i.e., those that can be performed for a particular instance). Section 4.2 described necessary and sufficient conditions for unconditionally autonomously computable insertions, deletions, and modifications. Section 4.3 described necessary and sufficient conditions for conditionally autonomously computable insertions and deletions.

Chapter 5

Differential re-evaluation of views

When the effect of an update to a relation is not autonomously computable we need to find other ways of updating the materialized views. As pointed out early in this thesis, a materialized view can always be brought up to date by re-evaluating its defining relational expression against the updated database. We refer to this way of updating a view as *complete re-evaluation.*

In this chapter we present an approach based on "query modification" to re-evaluating a view differentially. By *differential re-evaluation* we mean bringing the materialized view up to date by computing which tuples must be inserted into or deleted from the view based on the actual updated tuples from relations whose schemes are mentioned in the view definition.

The differential re-evaluation mechanism is always performed at the site where the relations are stored. For simplicity, it is assumed that the relations are updated by transactions and that the differential update mechanism is invoked as the last operation within a transaction (i.e., as part of the *commit* of a transaction). It is also assumed that the information available when the differential view update mechanism is invoked consists of: (a) the contents of each relation before the execution of the transaction, (b) the set of tuples actually inserted into or deleted from each relation, (c) the view definition, and (d) the contents of the view that agrees with the contents of the relations before the execution of the transaction.

Notice in particular that (b) only includes the *net* changes to the relations: for example, if a tuple not in the relation is inserted and then deleted within a transaction, it is not represented at all in this set of changes. Thus the set of inserted tuples and the set of deleted tuples have no tuples in common. This assumption is not unreasonable considering that the differential re-evaluation mechanism is invoked as part of a transaction, and the net changes to the base relations are still available, for instance, in main memory or in the log file.

Because of the above, it does not make sense in this chapter to talk about updates of the form:

- DELETE (R_u, C_D)
- MODIFY $(R_u, C_M, \mathbf{F}_M)$

and instead we only need the forms:

- INSERT (R_u, T)
- DELETE (R_u, T)

For convenience, in this chapter and the next we refer to the view definition by the actual relational algebra expression instead of the triple $(\mathbf{A}, \mathbf{R}, C)$ used in previous sections. We also refer to the view instance $v(E, d)$ by simply v.

5.1 Select views

A *select view* is defined by the expression $E = \sigma_{C(Y)}(R)$, where C (the selection condition) is a Boolean expression defined on $Y \subseteq \alpha(R)$. Let i_r and d_r denote the set of tuples inserted into or deleted from relation (instance) r, respectively. The new state of the view, called v', is computed by the expression $v' = (v \cup \sigma_{C(Y)}(i_r)) - \sigma_{C(Y)}(d_r)$. That is, the view can be updated by the sequence of operations

INSERT $(E, \sigma_{C(Y)}(i_r))$
DELETE $(E, \sigma_{C(Y)}(d_r))$.

Assuming $|v| \gg |d_r|$, it is cheaper to update the view by the above sequence of operations than recomputing the expression E from scratch.

Example 5.1 Consider the relation scheme $R(H, I)$ and a view defined by $E = \sigma_{H>10}(R)$. Suppose that after the view v is materialized, the relation r on R is updated by the insertion of the set tuples i_r and the deletion of the set of tuples d_r, where $i_r \cap d_r = \emptyset$. Then the new state of the view, called v', is computed by the expression

$$v' \quad = \quad v \cup \sigma_{H>10}(i_r) - \sigma_{H>10}(d_r).$$

From this it is clear that if the size of the set $\sigma_{H>10}(d_r)$ is comparable to the size of v, then complete re-evaluation of the view will be cheaper. □

5.2 Project views

A *project view* is defined by the expression $E = \pi_X(R)$, where $X \subseteq \alpha(R)$. The project operation causes more difficulty than a select operation for updating views differentially. The difficulty arises when the relation r is updated through a delete operation.

Example 5.2 Consider a relation scheme $R(H, I)$, a project view defined as $E = \pi_I(R)$, and the relation r shown below:

r:

H	I
1	10
2	10
3	20

v:

I
10
20

If the tuple $(3, 20)$ is deleted from relation r, then the view can be updated by deleting the tuple (20) from it. However, if the tuple $(1, 10)$ is deleted from relation r, then the view cannot be updated by deleting the tuple (10) from it. The reason for this difficulty is that the distributive property of projection over difference does not hold (i.e., $\pi_X(r_1 - r_2) \neq \pi_X(r_1) - \pi_X(r_2)$). □

There are two alternatives for solving the problem.

1. Attach an additional attribute to each tuple in the view, a multiplicity *counter*, which records the number of operand tuples that contribute to the tuple in the view. Inserting a tuple already in the view causes the counter for that tuple to be incremented by one. Deleting a tuple from the view causes the counter for that tuple to be decremented by one; if the counter becomes zero, then the tuple in the view can be safely deleted.

2. Include the key of the underlying relation within the set of attributes projected in the view. This alternative allows unique identification of each tuple in the view. Insertions or deletions cause no trouble since the tuples in the view are uniquely identified.

We choose alternative (1) since we do not want to impose restrictions on the views other than the class of relational algebra expressions allowed in their definition. In addition, alternative (2) becomes an special case of alternative (1) in which every tuple in the view has a counter value of one.

We require that relations and views include an additional (invisible) attribute, which we will denote $\mathcal{N}$. For relations, this attribute need not be explicitly stored since its value in every tuple is always one. The select operation is not affected by this assumption. The project operation is redefined as

$$\begin{aligned}\pi_X(r) &= \{t(X') \mid X' = X \cup \{\mathcal{N}\} \text{ and} \\ &\quad \exists u \in r[(u[X] = t[X]) \wedge (t[\mathcal{N}] = \sum_{w \in W} w[\mathcal{N}] \\ &\quad \text{where } W = \{w \mid w \in r \wedge w[X] = t[X]\})]\}.\end{aligned}$$

The union and difference of two relations on scheme R are defined as:

$$\begin{aligned}r_1 \cup r_2 &= \{t(R) \mid (\exists u \in r_1(t[R - \{\mathcal{N}\}] = u[R - \{\mathcal{N}\}]) \vee \\ &\quad \exists v \in r_2(t[R - \{\mathcal{N}\}] = v[R - \{\mathcal{N}\}])) \wedge \\ &\quad t[\mathcal{N}] = u[\mathcal{N}] + v[\mathcal{N}]\}\end{aligned}$$

and

$$\begin{aligned}r_1 - r_2 &= \{t(R) \mid (\exists u \in r_1(t[R - \{\mathcal{N}\}] = u[R - \{\mathcal{N}\}]) \wedge \\ &\quad \nexists v \in r_2(t[R - \{\mathcal{N}\}] = v[R - \{\mathcal{N}\}]) \ \wedge t(R) = u(R)) \vee \\ &\quad (\exists u \in r_1, v \in r_2(t[R - \{\mathcal{N}\}] = u[R - \{\mathcal{N}] = v[R - \{\mathcal{N}\}]) \wedge \\ &\quad t[\mathcal{N}] = max(u[\mathcal{N}] - v[\mathcal{N}], 0))\}\end{aligned}$$

Notice that by redefining these operations, the distributive property of projection over difference now holds (i.e., $\pi_X(r_1 - r_2) = \pi_X(r_1) - \pi_X(r_2)$), and thus the differential update can be accomplished by INSERT $(E, \pi_X(i_r))$ and DELETE $(E, \pi_X(d_r))$.

To complete the definition of operators to include the multiplicity counter the Cartesian product operation is redefined as

$$\begin{aligned}r \times s &= \{t(Y) \mid Y = R \cup S \text{ and } \exists u, v\ [(u \in r) \wedge (v \in s) \wedge \\ &\quad (t(R - \{\mathcal{N}\}) = u(R - \{\mathcal{N}\})) \ \wedge \ (t(S - \{\mathcal{N}\}) = v(S - \{\mathcal{N}\})) \wedge \\ &\quad (t(\mathcal{N}) = u(\mathcal{N}) * v(\mathcal{N}))]\},\end{aligned}$$

where '$*$' denotes scalar multiplication.

5.3 Product views

A *product view* is defined by the expression

$$E = R_1 \times R_2 \times \cdots \times R_p.$$

We consider first changes to the relations exclusively through insert operations, next we consider changes exclusively through delete operations, and finally we consider changes through both insert and delete operations.

Example 5.3 Consider two relation schemes $R(H, I)$ and $S(J, K)$, and a view defined as $E = R \times S$. Suppose that after the view v is materialized, the relation r is updated by the insertion of the set of tuples i_r. Let $r' = r \cup i_r$. The new state of the view, called v', is computed by the expression

$$\begin{aligned} v' &= r' \times s \\ &= (r \cup i_r) \times s \\ &= (r \times s) \cup (i_r \times s). \end{aligned}$$

If we let $i_v = i_r \times s$, then $v' = v \cup i_v$. That is, the view can be updated by inserting only the new set of tuples i_v into relation v. In other words, one only needs to compute the contribution of the new tuples in r to the Cartesian product. Clearly, it is cheaper to compute the view v' by adding i_v to v than to recompute the Cartesian product completely from scratch. □

This idea can be generalized to views defined as the Cartesian product of an arbitrary number of relations by exploiting the distributive property of Cartesian product with respect to union.

Consider a database $d = \{r_1, r_2, \ldots, r_p\}$ and a view defined as $E = R_1 \times R_2 \times \cdots \times R_p$. Let v denote the materialized view, and the relations $r_1, r_2, \ldots, r_p$ be updated by inserting the sets of tuples $i_{r_1}, i_{r_2}, \ldots, i_{r_p}$. The new state of the view v' can be computed as

$$v' = (r_1 \cup i_{r_1}) \times (r_2 \cup i_{r_2}) \times \cdots \times (r_p \cup i_{r_p}).$$

Let us associate a binary variable δ_i with each of the relation schemes $R_i, 1 \leq i \leq p$. The value *zero* for δ_i refers to the tuples of r_i considered during the current materialization of the view v (i.e., the old tuples), and the value *one* for δ_i refers to the set of tuples inserted into r_i since the latest materialization of v (i.e., the new tuples i_r). The expansion of the expression for v', using the distributive property of Cartesian product over union, can be depicted by the truth table of the variables δ_i. For example, if $p = 3$ we have

δ_1	δ_2	δ_3		
0	0	0		$r_1 \times r_2 \times r_3$
0	0	1		$r_1 \times r_2 \times i_{r_3}$
0	1	0	which	$r_1 \times i_{r_2} \times r_3$
0	1	1	repre–	$r_1 \times i_{r_2} \times i_{r_3}$
1	0	0	sents	$i_{r_1} \times r_2 \times r_3$
1	0	1		$i_{r_1} \times r_2 \times i_{r_3}$
1	1	0		$i_{r_1} \times i_{r_2} \times r_3$
1	1	1		$i_{r_1} \times i_{r_2} \times i_{r_3}$

where the union of all expressions in the right hand side of the table is equivalent to v'. The first row of the truth table corresponds to the Cartesian product of the relations considering only old tuples (i.e., the current state of the view v). Typically, a transaction would not insert tuples into all the relations involved in a view definition. In that case, some of the combinations of Cartesian products represented by the rows of the truth table correspond to null relations. Using the table for $p = 3$, suppose that a transaction contains insertions to relations r_1 and r_2 only. One can then discard all the rows of the truth table for which the variable δ_3 has a value of one, namely rows 2, 4, 6, and 8. Row 1 can also be discarded, since it corresponds to the current materialization of the view. Therefore, to bring the view up to date we need to compute only the Cartesian products represented by rows 3, 5, and 7. That is,

$$\begin{aligned} v' &= v \cup (r_1 \times i_{r_2} \times r_3) \\ &\quad \cup (i_{r_1} \times r_2 \times r_3) \\ &\quad \cup (i_{r_1} \times i_{r_2} \times r_3). \end{aligned}$$

The computation of this differential update of the view v is certainly cheaper than recomputing the whole Cartesian product.

So far we have assumed that the relations change only through the insertion of new tuples. The same idea can be applied when the relations change only through the deletion of old tuples.

Example 5.4 Consider again two relation schemes $R(H, I)$ and $S(I, J)$, and the view defined as $E = R \times S$. Suppose that after the view v is materialized, the relation r is updated by the deletion of the set of tuples d_r. Let $r' = r - d_r$. The new state of the view, called v', is computed as

$$\begin{aligned} v' &= r' \times s \\ &= (r - d_r) \times s \\ &= (r \times s) - (d_r \times s). \end{aligned}$$

If we let $d_v = d_r \times s$, then $v' = v - d_v$. That is, the view can be updated by deleting the new set of tuples d_v from the relation v. It is not always cheaper to compute the view v' by deleting from v only the tuples d_v; however, this is true when $|v| \gg |d_v|$. □

The differential update computation for deletions can also be expressed by means of binary tables. Thus, the computation of differential updates depends on the ability to identify which tuples have been inserted and which tuples have been deleted.

Example 5.5 Consider two relation schemes $R(H, I)$ and $S(J, K)$, and a view defined as $E = R \times S$. Let r and s denote instances of the relations named R and S, respectively, and $v = r \times s$. Assume that a transaction T updates relations r and s. For any tuple t in the Cartesian product of the updates, one of the following conditions (or symmetric condition) must hold:

Case 1: $t \in i_r \times i_s$ is a tuple that has to be inserted into v.

Case 2: $t \in i_r \times d_s$ is a tuple that has no effect in the view v, and can therefore be ignored.

Case 3: $t \in i_r \times s$ is a tuple that has to be inserted into v.

Case 4: $t \in d_r \times d_s$ is a tuple that has to be deleted from v.

Case 5: $t \in d_r \times s$ is a tuple that has to be deleted from v.

Case 6: $t \in r \times s$ is a tuple that already exists in the view v.

□

From now on, all tuples are assumed to be tagged in such a way that it is possible to identify inserted, deleted, and old tuples. In general, we can describe the value of the tag field of the tuple resulting from a Cartesian product of two tuples according to the following table.

r_1	r_2	$r_1 \times r_2$
insert	insert	insert
insert	delete	ignore
insert	old	insert
delete	insert	ignore
delete	delete	delete
delete	old	delete
old	insert	insert
old	delete	delete
old	old	old

where the last column of the table shows the value of the tag attribute for the tuple resulting from the Cartesian product of two tuples tagged according to the values under columns r_1 and r_2. Tuples tagged as "ignore" are assumed to be discarded when performing later Cartesian products. In other words, they do not "emerge" from the Cartesian product.

The semantics of the Cartesian product operation has to be redefined to compute the tag value of each tuple resulting from the Cartesian product based on the tag values of the operand tuples. In the presence of projection this will be in addition to the computation of the count value for each tuple resulting from the Cartesian product as explained in the section on project views. Similarly, the tag value of the tuples resulting from a select or project operation is described in the following table.

r	$\sigma_{C(Y)}(r)$	$\pi_X(r)$
insert	insert	insert
delete	delete	delete
old	old	old

In practice, it is not necessary to build a table with 2^p rows. Instead, by knowing which relations have been modified, we can build only those rows of the table representing the necessary subexpressions to be evaluated. Assuming that only k such relations were modified, $1 \leq k \leq p$, building the table can be done in time $O(2^k)$.

Once we know what subexpressions must be computed, we can further reduce the cost of materializing the view by using an algorithm to determine a good order for execution of the joins (i.e., Cartesian products with the associated Boolean predicates). Notice that a new feature of our problem is the possibility of saving computation by re-using partial subexpressions appearing in multiple rows within the table. Efficient solutions to this problem are examined in Chapter 6.

5.4 Project-Select-Join views

A *project-select-join view* (*PSJ*-view) is defined by the expression

$$E = \pi_X(\sigma_{C(Y)}(R_1 \times R_2 \times \cdots \times R_p)),$$

where X is a set of attributes and $C(Y)$ is a Boolean expression. We can again exploit the distributive property of Cartesian product, select, and project over union to provide a differential update algorithm for *PSJ*-views.

Example 5.6 Consider two relation schemes $R(H, I)$ and $S(J, K)$, and a view defined as $E = \pi_A(\sigma_{(H=J)\wedge(J>10)}(R \times S))$. Suppose that after the view v is materialized, the relation r is updated by the insertion of tuples i_r. Let $r' = r \cup i_r$. The new state of the view, called v', is computed by the expression

$$\begin{aligned} v' &= \pi_A(\sigma_{(H=J)\wedge(J>10)}(r' \times s)) \\ &= \pi_A(\sigma_{(H=J)\wedge(J>10)}((r \cup i_r) \times s)) \\ &= \pi_A(\sigma_{(H=J)\wedge(J>10)}(r \times s)) \cup \pi_A(\sigma_{(H=J)\wedge(J>10)}(i_r \times s)) \\ &= v \cup \pi_A(\sigma_{(H=J)\wedge(J>10)}(i_r \times s)). \end{aligned}$$

If we let $i_v = \pi_A(\sigma_{(H=J)\wedge(J>10)}(i_r \times s))$, then $v' = v \cup i_v$. That is, the view can be updated by inserting only the new set of tuples i_v into the relation v. □

We now present the outline of an algorithm to update *PSJ*-views differentially. The algorithm also incorporates a filtering stage which removes irrelevant tuples from each set of updates to the relations.

Algorithm 5.1

Input:

i) the *PSJ*-view definition $E = \pi_X(\sigma_C(R_1 \times R_2 \times \cdots \times R_p))$,

ii) the contents of the relations $r_j, 1 \leq j \leq p$, and

iii) the sets of updates $\hat{r}_j$, $1 \leq j \leq p$, denoting the collection of insertions i_j into and deletions d_j from relation r_j.

Output: a transaction to update the view.

1. Remove irrelevant tuples from each relation $\hat{r}_j$, $1 \leq j \leq p$. This is done by invoking Algorithm 3.1 with input parameters C, R_j, and $\hat{r}_j$. Algorithm 3.1 returns the relation $\hat{r}'_j$ containing only relevant tuples.
2. For each base relation r_j, $1 \leq j \leq p$, compute $r'_j = r_j - d_j$.[1]
3. Build those rows of the truth table with p columns corresponding to the relations being updated. If $\hat{r}'_j = \emptyset$, then discard rows where $\delta_j = 1$.
4. For each row of the table, compute the associated *PSJ*-expression substituting r'_j when the binary variable $\delta_j = 0$, and $\hat{r}'_j$ when $\delta_j = 1$.
5. Perform the union of results obtained for each computation in Step 4. The transaction consists of inserting all tuples tagged as insert, and deleting all tuples tagged as delete. □

Observe that: (I) we can use for E an expression with a minimal number of joins. Such expression can be obtained at view definition time by the tableau method of Aho, Sagiv and Ullman [AHO79] extended to handle inequality conditions [KLUG80]; and (II) Step 4 poses an interesting optimization problem, namely, the efficient execution of a set of *PSJ*-expressions (all the same) whose operands represent different relations and where intermediate results can be re-used among several expressions. Chapter 6 explores this problem in more detail and provides several approaches to its solution.

5.5 Summary

An approach to differentially re-evaluating materialized views defined by arbitrary *PSJ*-expressions has been presented. The method, based on query modification, uses the state of the base relations and the net changes applied to the base relations since the latest update of the view. Differential re-evaluation leads to a special multiple query optimization problem whose solution is explored in the next chapter.

[1]This step has been added to the algorithm from an observation by Hanson [HANSO86].

Chapter 6

A Multiple Query Optimization Problem

6.1 Introduction

In the previous chapter we presented a differential approach to updating materialized views. The approach, based on query modification, requires the computation of several queries, all having the same relational expression but different operands. After a review of related work, we define the multiple query problem formally and present a simplistic cost model. In Section 6.4 we describe several alternatives for solving the problem and also introduce a back-of-the-envelope analysis for the case when the queries are optimized individually and not globally. In Section 6.5 we present a representation of the queries being optimized intended to facilitate the detection of common subexpressions. In Section 6.6 we present a solution based on a new heuristic for query decomposition of multi-query graphs. In Section 6.7 we present a solution to the problem based on space-search methods. The main emphasis in this section is on a new way of generating alternative access plans taking into account the characteristics of this particular multiple query optimisation problem.

6.2 Related work

The majority of the work on query optimisation has focused on improving the performance of queries presented one at a time. The paper by Jarke and Koch [JARKE84] is an excellent survey of this field.

Much work has been done to find efficient ways of computing relational operators and to find efficient methods of selecting access paths. As classical examples, Blasgen and Eswaran [BLASG76] compare algorithms for computing joins by considering only the cost of accessing secondary storage, and Selinger et al. [SELIN79] describe how System R chooses access paths. In the latter, optimal access paths are obtained by an exhaustive search of all possible access paths, where the cost formulae used involve I/O as well as CPU costs. Estimates of the size of intermediate results are obtained by utilizing very approximate selectivities. Ibaraki and Kameda [IBARA84] discuss optimal ways for computing joins represented by a connection graph using the nested loops method. Their results indicate that the optimal nesting order for joins represented by a connection graph having the form of a tree can be found in polynomial time. For more general types of graphs they prove that the problem is *NP*-complete.

An important technique used in query optimization is "query modification." The purpose of query modification is to find expressions that are equivalent to the given query, but whose evaluation is more efficient. Some of the laws of relational algebra used in query modification are described in the books by Ullman [ULLMA82] and by Maier [MAIER83a]. Among the approaches to query optimization that are based on query modification we can include: optimization of algebraic expressions, query decomposition, tableau query optimization, and optimization of conjunctive queries.

Recently, some researchers have started to investigate techniques for improving the performance of a set of queries as a whole, rather than a query at a time [GRANT81,SELL86a,SELL86b,CHAKR86]. All the aspects previously considered for the optimization of single queries are also relevant for the optimization of multiple queries. Among these aspects we can include:

- query representation,
- query modification,
- plan enumeration,
- cost estimation,
- optimal plan selection.

Some other aspects already addressed in the optimization of single queries become even more important in the optimization of multiple queries, for example, the detection and use of common subexpressions [FINKE82,JARKE85]. It is precisely this aspect that makes multiple query optimization worthwhile. If the queries to be optimized do not share any subexpression (i.e., they refer to disjoint sets of relations), then single query optimization is sufficient. Also, the detection of common subexpressions as well as the efficient storage and management of temporary results for multiple queries makes the optimization process closer to the optimization of code for compilers.

6.3 The problem

Suppose we want to maintain the materialized view $v = \pi_X(\sigma_{C(Y)}(r_1 \times r_2 \times \cdots \times r_p))$, and that a transaction $\mathcal{T}$ updates $q \leq p$ relations. The updated view v' is then given by the expression $v' = v \sqcup \Delta v$, where v represents the contents of the view before the execution of the transaction, the operator $\sqcup$ denotes the simultaneous union or difference of tuples, and Δv is given by the expression

$$\begin{array}{rcll} \Delta v & = & \pi_X(\sigma_C(r_1 \times \cdots \times r_{p-q} \times r'_{p-q+1} \times \cdots \times r'_{p-1} \times \hat{r}_p)) & \cup \\ & & \pi_X(\sigma_C(r_1 \times \cdots \times r_{p-q} \times r'_{p-q+1} \times \cdots \times \hat{r}_{p-1} \times r'_p)) & \cup \\ & & \pi_X(\sigma_C(r_1 \times \cdots \times r_{p-q} \times r'_{p-q+1} \times \cdots \times \hat{r}_{p-1} \times \hat{r}_p)) & \cup \\ & & \cdots & \cup \\ & & \pi_X(\sigma_C(r_1 \times \cdots \times r_{p-q} \times \hat{r}_{p-q+1} \times \cdots \times \hat{r}_{p-1} \times \hat{r}_p)). & \end{array}$$

Relations r_i, $1 \leq i \leq p$, represent the relations used to compute v. Relations $r'_j = r_j - d_j$, $p - q + 1 \leq j \leq p$, where $d_j \subseteq \hat{r}_j$ represents the set of deletions made to relation r_j within the transaction. Relations $\hat{r}_i$ represent the ***net*** changes (i.e., insertions and deletions) made to the corresponding relations within the transaction. Relations $\hat{r}_i$ are assumed to be non-empty. Thus, if q relations are updated, then Δv will consist of the union of $2^q - 1$ *PSJ*-expressions. The problem is how to compute Δv efficiently. This is equivalent to trying to minimize the overall cost of the evaluation of the set of queries $\{v_1, v_2, \ldots, v_{2^q-1}\}$, where each v_i is an *PSJ*-expression.

Multiple query optimization is a natural approach to solving this problem. The reasons that make a multiple query optimization approach attractive are: (a) the set of queries to be optimized is known in advance, (b) all queries v_i share the same projection and selection condition, (c) since there is extensive relation overlap among the queries, many subexpressions can be shared.

Before we start dealing with query optimization we need a cost model. Some of the notation used throughout the chapter is given below:

$\lvert r_i \rvert$	size of relation r_i (e.g., number of tuples, number of pages).
p	number of relations involved in a query expression.
q	number of relations updated.
sel_{ij}	selectivity factor of a join involving relations r_i and r_j.

The selectivity sel_{ij} with respect to the join of r_i and r_j is defined to be the expected fraction of tuples from $r_i \times r_j$ satisfying the join condition.

For simplicity, it is assumed throughout this chapter that each query to be optimized is represented by a connection graph consisting of a chain of nodes. A ***chain (of nodes)*** is a connected undirected acyclic graph with maximum degree 2. Each node in the chain represents a relation in an expression. Each edge between two nodes labeled r_i and r_j represents a join condition between these two relations. We also assume that all joins are performed using the ***nested loops method*** [BLASG76,IBARA84].

Given two relations r_i and r_j to be joined, the cost of computing the join $cost(r_i, r_j)$ and the estimated size of the result $size(r_i, r_j)$ are given by:

$$\begin{aligned} cost(r_i, r_j) &= |r_i| * |r_j| \\ size(r_i, r_j) &= |r_i| * |r_j| * sel_{ij}. \end{aligned}$$

Example 6.1 Consider three relation schemes $R_1(H, I, J)$, $R_2(K, L)$, and $R_3(M, N, O)$ with relation instances r_1, r_2 and r_3, and the query

$$E = \sigma_{(J=K)\wedge(L<M)}(r_1 \times r_2 \times r_3).$$

Assume that $|r_1| = 100$, $|r_2| = 60$, $|r_3| = 80$, $sel_{12} = 0.1$, and $sel_{23} = 0.4$. There are two ways of evaluating the expression E depending on two different nesting orders for computing the joins. Let us denote these two different nesting alternatives by $nest_1(E)$ and $nest_2(E)$, respectively. The cost of evaluating the expression E using two different nesting orders is given by:

$$\begin{aligned} cost(nest_1(E)) &= cost(r_1, r_2) + cost(r_1[J = K]r_2, r_3) \\ &= |r_1| * |r_2| + |r_1| * |r_2| * sel_{12} * |r_3| \\ &= (100)(60) + (100)(60)(0.1)(80) \\ &= 54,000 \end{aligned}$$

and

$$\begin{aligned} cost(nest_2(E)) &= cost(r_2, r_3) + cost(r_2[L < M]r_3, r_1) \\ &= |r_2| * |r_3| + |r_2| * |r_3| * sel_{23} * |r_1| \\ &= (60)(80) + (60)(80)(0.4)(100) \\ &= 196,800 \end{aligned}$$

Instead of performing two joins, we can perform the two joins at the same time. The idea is to join each tuple $t \in r_2$ with relations r_1 and r_3. We refer to this way of performing the join as *two-way join*. This strategy is denoted by $nest_3(E)$ and its cost is given by:

$$\begin{aligned} cost(nest_3(E)) &= |r_2| * (|r_1| + |r_3|) \\ &= 60(100 + 80) \\ &= 10,800 \end{aligned}$$

In all cases, the estimated size of the result is 19200. □

Given a relation r_i and a selection operation to be applied over r_i, the cost of computing the selection $cost_{ii}$ and the estimated size of the result $size_{ii}$ are given by:

$$\begin{aligned} cost_{ii} &= |r_i| \\ size_{ii} &= |r_i| * sel_{ii} \end{aligned}$$

We want to emphasize that this is a simplistic cost model which was adopted in order to concentrate on the treatment of query optimization throughout this chapter. The model carries the assumption of uniformity and independence of attribute values. Other cost models using similar assumptions are given in [BLASG76,SELIN79,YAO78,YAO79,GRANT81,SELL86b]. It has been shown by Christodoulakis [CHRIS81] that such assumptions often lead to pessimistic estimations of query costs.

6.4 Alternative solutions

There are two general alternatives available for computing the expression v'.

1. Complete re-evaluation. This alternative is the simplest. Every time one or more relations participating in a view definition is updated, the view expression is re-evaluated from scratch to bring the view up-to-date with the relations.
2. Differential re-evaluation. In this alternative we have two options depending on the amount of sharing of common subexpressions we are willing to exploit.
   - No sharing of common subexpressions.
     - Optimize each expression that participates in Δv independently.
   - Sharing of common subexpressions. The main problem here is to enumerate a reasonable number of strategies leading to a good amount of sharing.
     - Exhaustive search.
     - Query decomposition.
     - Space search methods.

6.4.1 Differential re-evaluation with no sharing of common subexpressions

In this strategy, the set of changes to be applied to the view is computed by optimizing each of the queries that compose Δv, independently. This is the alternative where current query optimizers can be utilized directly. We assume that all selections on single relations are performed first and at the end we are left only with the problem of finding an optimal order for computing a sequence of joins represented by a chain. The algorithm presented is a dynamic programming solution to the problem of finding an optimal nesting order for computing a sequence of p joins. The algorithm considers one- and two-way joins. The solution is inspired by the algorithm given by Aho et al. [AHO74] to find an optimal nesting for computing a sequence of matrix multiplications.

The following is a description of variables used in the algorithm:

- p the number of relations to be joined.
- *cost*$[i, j]$ is a $p \times p$ array to record the minimum cost of computing the sequence of joins $r_i[\theta_i]r_{i+1}[\theta_{i+1}] \cdots [\theta_{j-1}]r_j$, where $[\theta_l]$, $i \leq l \leq j-1$, represents a theta-join between relations r_l and relation r_{l+1}.
- *sel*$[i]$ is a $p-1$ vector containing the selectivity factor of the join between relations r_i and r_{i+1}.
- *est_size*$[i, j]$ is a $p \times p$ array to record the estimated size of the result of performing $r_i[\theta_i]r_{i+1}[\theta_{i+1}] \cdots [\theta_{j-1}]r_j$.
- *size*$[i]$ is a p vector containing the size of relation r_i.
- *type*$[i, j]$ is a $p \times p$ array to record the type of join which makes *cost*$[i, j]$ minimal. Type one is for one-way join and type two is for two-way join.
- *best*$[i, j]$ is a $p \times p$ array to record the value of k for which *cost*$[i, j]$ is minimal as defined below.

$$cost[i,j] = min(cost1[i,j], cost2[i,j]),$$

where

$$cost1[i,j] = \begin{cases} 0 & \text{if } i = j \\ min_{i \leq k < j}(\ cost[i,k] + cost[k+1,j] + \\ \quad est\_size[i,k] * est\_size[k+1,j]\ ) & \text{otherwise.} \end{cases}$$

and

$$cost2[i,j] = \begin{cases} 0 & \text{if } i = j \\ min_{i \leq k < j}(\ cost[i,k] + cost[k+2,j] + \\ (est\_size[i,k] + est\_size[k+2,j]) * size[k+1]\ ) & \text{otherwise.} \end{cases}$$

Algorithm 6.1. Dynamic programming algorithm to find a minimal-cost nesting-order for computing the sequence of joins $r_1[\theta_1]r_2[\theta_2] \cdots [\theta_{p-1}]r_p$.

Input: *size*$[i]$, $1 \leq i \leq p$ and *sel*$[j]$, $1 \leq j \leq p-1$.

Output: the array *best* from which a minimal cost nesting for computing the $p-1$ joins can be obtained. According to the cost model of Subsection 6.3 the estimated cost of the one-way join $r_i[\theta_i]r_{i+1}$ is given by $size[i] * size[i+1]$, and the estimated size of the result is given by $size[i] * size[i+1] * sel[i]$. The cost of a two-way join $r_i[\theta_i]r_{i+1}[\theta_{i+1}]r_{i+2}$ is given by $size[i+1](size[i] + size[i+2])$. The estimated size of the result is the same as in the case of the one-way join.

```
$cost[i,j] = 0,\ 1 \le i \le j \le p$;
$est\_size[i,j] = 0,\ 1 \le i < j \le p$;
$est\_size[i,i] = size[i],\ 1 \le i \le p$;
for $w = 1$ until $p - 1$ do
  for $i = 1$ until $p - w$ do
    $j = i + w$;
    $cost[i,j] = Mincost(cost, i, j, k, t)$;
    $est\_size[i,j] = est\_size[i,k] * est\_size[k+1,j] * sel[k]$;
    $best[i,j] = k$;
    $type[i,j] = t$;
  od;
od;
return($best$);
```

The subroutine *Mincost* takes as input the array *cost* and the indices i and j, returning the minimal $cost[i,j]$ as well as the value of k, $i \le k < j$, for which

$$cost[i,k] + cost[k+1,j] + est\_size[i,k] * est\_size[k+1,j]$$

or

$$cost[i,k] + cost[k+2,j] + (est\_size[i,k] + est\_size[k+2,j]) * size[k+1]$$

is minimal. If the first expression is minimal, then $t = 1$ otherwise $t = 2$.

The algorithm produces an optimal nesting in time $O(p^3)$. Ibaraki and Kameda [IBARA84] present an algorithm that finds an optimal nesting for joins whose connection graph is a tree in time $O(p^2 \log p)$, using a slightly different cost function than the one presented in Subsection 6.3. Also, Ibaraki and Kameda's algorithm does not take into consideration two-way joins.

Example 6.2 Consider again Example 6.1. The computations made by Algorithm 6.1 can be depicted by the following table.

	$j = 1$	$j = 2$	$j = 3$
$i = 1$	*cost*=0 *est_size*=100	*cost*=6,000 *est_size*=600 *best*=1 *type*=1	*cost*=10,800 *est_size*=19,200 *best*=1 *type*=2
$i = 2$		*cost*=0 *est_size*=60	*cost*=4,800 *est_size*=1,920 *best*=2 *type*=1
$i = 3$			*cost*=0 *est_size*=80

The optimal nesting cost is given by $cost[1,3] = 10,800$. Since $type[1,3] = 2$, the optimal nesting requires a two-way join. Since $best[1,3] = 1$, the optimal nesting involves the scan of relation r_2, where each tuple of r_2 is joined simultaneously with relations r_1 and r_3. In general, if $best[1,p] = k$ and $type[1,p] = 2$, then the optimal nesting involves the scan of relation r_{k+1}, where each tuple of relation r_{k+1} is joined simultaneously with the result of $r_1 \times \cdots \times r_k$ and the result of $r_{k+2} \times \cdots \times r_p$. To obtain the optimal nestings for each of $r_1 \times \cdots \times r_k$ and $r_{k+2} \times \cdots \times r_p$, we look up the values of $best[1,k]$, $type[1,k]$, and $best[k+2,p]$, $type[k+2,p]$, respectively. □

6.4.2 Cost comparison of differential re-evaluation with no sharing of common subexpressions and complete re-evaluation

This section contains a simplistic cost comparison of the differential approach to updating materialized views against complete re-evaluation of the query defining the view when transactions involve insertions only.

For simplicity we assume that all relations r_i have size N; all relations $\hat{r}_j$ have size n; that is, $|r_i \cup \hat{r}_i| = N + n$; and all selectivities $sel_{k,k+1} = s$, $1 \leq k \leq p-1$, $0 < s < 1$. We focus only on the cost of performing p joins. The costs of selections on single relations along with the costs of performing the union of the results are not considered. Joins are computed using the nested loops method.

Let C_{comp} and C_{diff} denote the cost of complete re-evaluation and the cost of differential re-evaluation, respectively. C_{diff} corresponds to the sum of minimal costs (i.e., optimal nestings) for each of the queries in the differential re-evaluation. First, we derive an approximation for C_{comp} as follows:

$$\begin{aligned} C_{comp} &= N + sN^2 + \cdots + s^{p-q-1}N^{p-q} + \\ &\quad s^{p-q}N^{p-q}(N+n) + \cdots + s^{p-1}N^{p-q}(N+n)^q + S_{comp} \\ &= \sum_{i=0}^{p-q-1} s^i N^{i+1} + s^{p-q}N^{p-q}\sum_{j=0}^{q-1} s^j (N+n)^{j+1} + S_{comp}, \end{aligned}$$

where $S_{comp} = 2(p-q-1)N\log_z N + 2q(N+n)\log_z(N+n)$ is an approximation to the cost of sorting all relations, except the one scanned in the outermost loop, on the appropriate attributes before the execution of the nested loops method as in [IBARA84]. The cost of sorting each relation using z-way merge sort is $2\lceil M \log_z M \rceil$, where M is the size of the relation [BLASG76].

To derive C_{diff}, notice that the cost of evaluating an arbitrary expression from the set of expressions to be optimized having l relations of the type $\hat{r}$ is given by:

$$\begin{aligned} C &= n + sn^2 + \cdots + s^{l-1}n^l + s^l n^l N + \cdots + s^{p-1}n^l N^{p-l} + S_{diff} \\ &= n(1 + sn + \cdots + s^{l-1}n^{l-1}) + s^l n^l N(1 + sN + \cdots + s^{p-l-1}N^{p-l-1}) \\ &\quad + S_{diff} \end{aligned}$$

$$= \sum_{i=0}^{l-1} s^i n^{i+1} + s^l n^l \sum_{j=0}^{p-l-1} s^j N^{j+1} + S_{diff},$$

where $S_{diff} = 2(l-1)n \log_z n + 2(p-l)N \log_z N$ is again an approximation to the cost of sorting all relations, except the outermost, before the execution of the nested loops method.

Since there are $\binom{q}{l}$ queries of this type in Δv we have:

$$C_l = \binom{q}{l} \left[\sum_{i=0}^{l-1} s^i n^{i+1} + s^l n^l \sum_{j=0}^{p-l-1} s^j N^{j+1} + S_{diff} \right]$$

The cost of differential re-evaluation C_{diff} is then given by:

$$\begin{aligned} C_{diff} &= \sum_{l=1}^{q} C_l \\ &= \sum_{l=1}^{q} \binom{q}{l} \left[\sum_{i=0}^{l-1} s^i n^{i+1} + s^l n^l \sum_{j=0}^{p-l-1} s^j N^{j+1} + S_{diff} \right]. \end{aligned}$$

If $n \ll N$, then taking the most significant terms in C_{comp} above we obtain

$$C_{comp} = s^{p-1} N^p + (s^{p-1} qn + s^{p-2}) N^{p-1} + O(N^{p-2}),$$

and similarly for C_{diff} above we obtain

$$C_{diff} = s^{p-1} qn N^{p-1} + O(N^{p-2}).$$

Thus,

$$C_{comp} = C_{diff} + s^{p-1} N^p + s^{p-2} N^{p-1} + O(N^{p-2}).$$

The costs S_{comp} and S_{diff} are included within the term $O(N^{p-2})$ in each of the expressions for C_{comp} and C_{diff}, respectively.

From the above expressions it is clear that if $n \ll N$, then differential re-evaluation will be cheaper than complete re-evaluation.

We now look at the expressions C_{comp} and C_{diff} when n is comparable to the value of N assuming insertions only. Let $n = \alpha N$, $0 \leq \alpha \leq 1$. Then we can express C_{comp} as

$$\begin{aligned} C_{comp} &= \sum_{i=0}^{p-q-1} s^i N^{i+1} + s^{p-q} N^{p-q} \sum_{j=0}^{q-1} s^j (N + \alpha N)^{j+1} + S_{comp} \\ &= \sum_{i=0}^{p-q-1} s^i N^{i+1} + s^{p-q} N^{p-q} \sum_{j=0}^{q-1} s^j N^{j+1} (1+\alpha)^{j+1} + S_{comp} \\ &\approx s^{p-1} (1+\alpha)^q N^p + s^{p-2} (1+\alpha)^{q-1} N^{p-1} + O(N^{p-2}), \end{aligned}$$

and C_{diff} as

$$\begin{aligned}
C_{diff} &= \sum_{l=1}^{q}\binom{q}{l}\sum_{i=0}^{l-1} s^i\alpha^{i+1}N^{i+1} + \sum_{l=1}^{q}\binom{q}{l} s^l\alpha^l N^l \sum_{j=0}^{p-l-1} s^j N^{j+1} + S_{diff} \\
&\approx \sum_{l=1}^{q}\binom{q}{l}\alpha^l s^l N^l \left[s^{p-l-1}N^{p-l} + s^{p-l-2}N^{p-l-1} + O(N^{p-l-2})\right] \\
&= s^{p-1}((1+\alpha)^q - 1)N^p + s^{p-2}((1+\alpha)^q - 1)N^{p-1} + O(N^{p-2}).
\end{aligned}$$

Thus,

$$\begin{aligned}
C_{comp} &= C_{diff} + s^{p-1}N^p - s^{p-2}N^{p-1}\alpha(1+\alpha)^{q-1} + s^{p-2}N^{p-1} \\
&\quad + O(N^{p-2}).
\end{aligned}$$

When the value of n is comparable to the value of N, then both alternatives have similar costs, but C_{diff} is still smaller than C_{comp}.

The main reason for the difference is that differential re-evaluation avoids computing the expression representing the set of tuples already in the view. It must be stressed that, because all the expressions that compose Δv compute disjoint sets of tuples, the amount of work required by differential re-evaluation will not be larger than the amount of work required by complete re-evaluation when tuples are inserted only.

Figure 6.1 depicts a comparison between C_{diff} and C_{comp} for the case when $n \ll N$. The values of the parameters are: $N = 10,000$, $s = 0.01$, and $p = 10$ and the relations have been updated by inserting $n = 100$ tuples and $n = 1,000$ tuples. Dashed lines and solid lines represent the cost of differential and complete re-evaluation, respectively.

Figure 6.2 depicts a comparison between C_{diff} and C_{comp} for the case when n is comparable to N. The values of the parameters are: $N = 10,000$, $s = 0.01$, and $p = 10$ and the relations have been updated by inserting $n = 8,000$ tuples and $n = 10,000$ tuples. In this case, dashed and solid lines overlap.

It should be clear from the discussion in this section that even with no sharing of common subexpressions, differential re-evaluation is an advantageous alternative against complete re-evaluation provided that the updated tuples can be "separated" easily. For the case in which relations are modified by inserting a large number of tuples, differential re-evaluation is not worse than complete re-evaluation.

Differential re-evaluation can provide even more savings if we allow sharing of common subexpressions among the queries being optimized. But in order to do this we need to adopt some query representation that will allow us to identify and take advantage of such common subexpressions. Section 6.5 presents such representation, Section 6.6 presents an approach for solving our problem using query decomposition, and finally Section 6.7 presents an approach for solving our problem using space search methods.

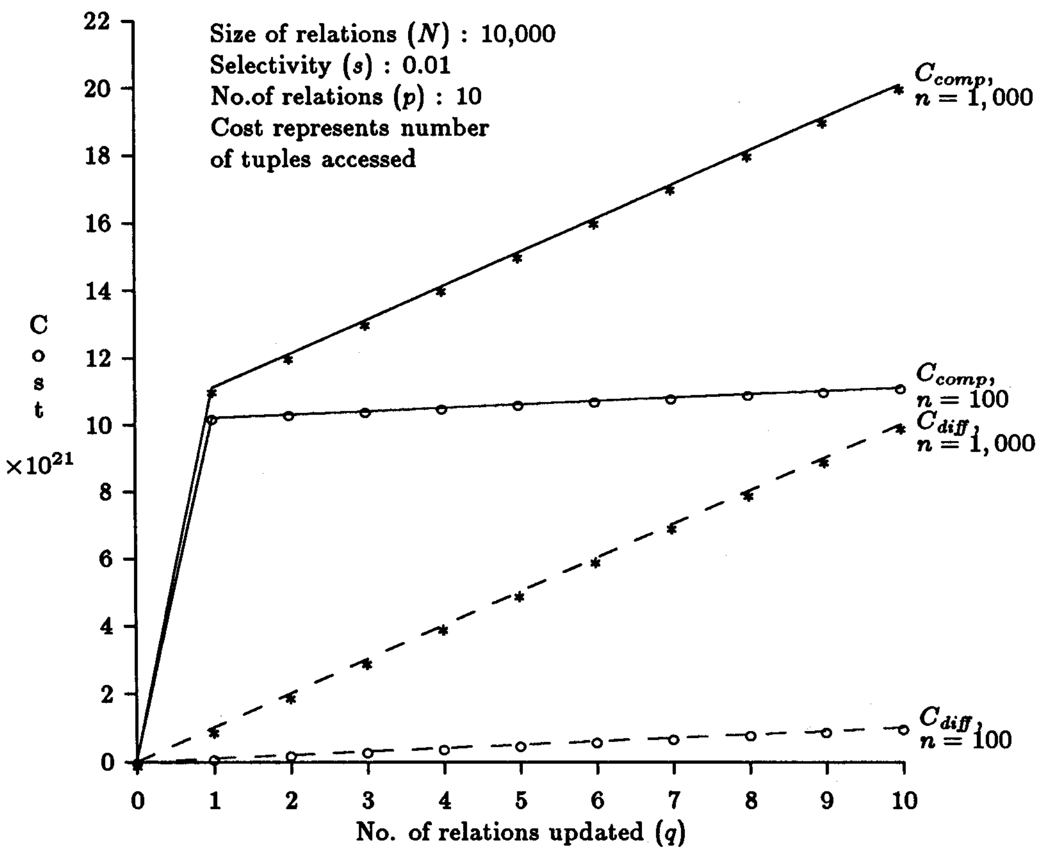


Figure 6.1: Comparison between complete and differential re-evaluation

6.5 Query representation

Researchers in query optimization have used a number of different representations for queries to facilitate their study. Examples of such representations are expressions in terms of tuple and domain relational calculus, parse trees, tableaux [AHO79], qual graphs [FINKE82], connection graphs (e.g., Ch.8 [ULLMA82]), and multi-query graphs [CHAKR86].

Some of the problems detected in the representations previously used are briefly summarized below.

- Finkelstein's approach is based on the use of temporary results which can potentially be used in later queries. Each temporary result represents a selec-

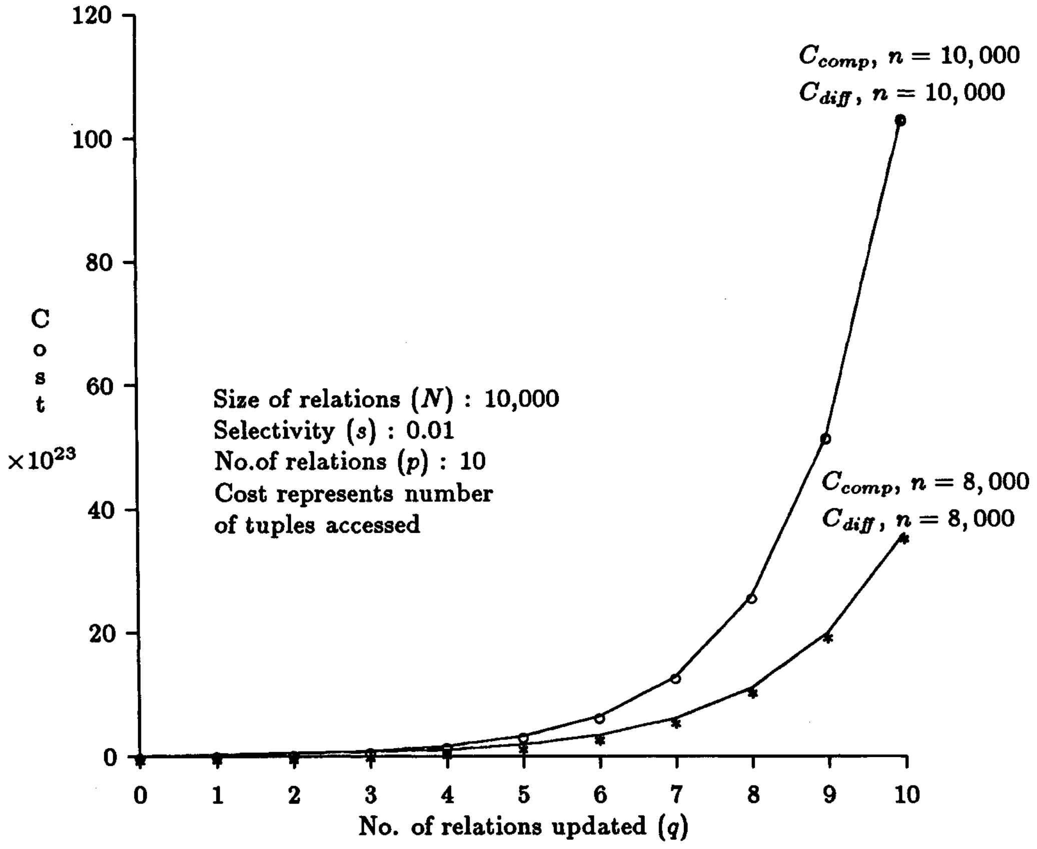


Figure 6.2: Comparison between complete and differential re-evaluation

tion and a projection on a single relation. Thus, his approach does not take advantage of temporary results involving joins.

- Sellis' approach [SELL86a] is based on a slight variation of Finkelstein's qual graphs. Each task involved in the execution of the query (e.g., a selection or a join) is represented by a node in the graph. Edges impose an order on the execution of the tasks represented by the nodes they connect. This representation can be seen as an operator graph, since all information relevant for sharing is encoded in the nodes, thus allowing the sharing of join operations.

- The tableau is a query modification tool which has been used for the optimization of single queries to derive an equivalent expression requiring a minimal number of joins. We can make use of this tool at view definition time to obtain an equivalent view expression with a minimal number of joins.

- Parse trees are not well suited for multiple query optimization because common subexpressions are hard to detect in that representation. Jarke [JARKE85] provides examples.

We choose Chakravarthy and Minker's representation [CHAKR86] called the *multi-query graph*, which is actually an extension of the connection graph of Ullman [ULLMA82].

Algorithm 6.2. Construction of an undirected multi-query graph $G = (V, E)$ for the set of queries $\{v_1, v_2, \ldots, v_{2^q-1}\}$.

Input: the set of queries $\{v_1, v_2, \ldots, v_{2^q-1}\}$, where each v_j is of the form

$$\pi_X(\sigma_{C(Y)}(r_1 \times r_2 \times \cdots \times r_p)).$$

Output: an undirected multi-query graph $G = (V, E)$.

1. Initialize the sets V and E to be the empty set.
2. Decompose the conjunctive condition $C(Y)$ into disjoint predicates $C_k(Y_k)$, $1 \leq k \leq m$. Each predicate $C_k(Y_k)$ contains either the conjunction of atomic formulae that use attributes of the same relation, or a single atomic formula representing a join condition. Also, each predicate $C_k(Y_k) = (r_{i_1}.x\ \theta\ r_{i_2}.y + c)$ is transformed into an equivalent predicate for which $i_1 < i_2$.
3. For each query v_j do steps 3.1 and 3.2.

   3.1. For each relation r_i $(\hat{r}_i)$, $1 \leq i \leq p$, mentioned in the query v_j insert a node r_i $(\hat{r}_i)$ into the set V.

   3.2. For each predicate $C_k(Y_k)$ in $C(Y)$ create an edge e and insert it into the set E. Each edge is a four-tuple (z_1, z_2, z_3, z_4) where, z_1 and z_2 represent the two nodes connected by the edge, z_3 (called the color of the edge) is a number that identifies the query to which the edges belong, and z_4 is a label representing the predicate associated with the nodes connected by the edge. Each edge is constructed as follows:

   - if C_k is of the form $(r_{i_1}.x\ \theta\ r_{i_2}.y + c)$, then $e = (r_{i_1}, r_{i_2}, j, C_k)$.
   - if C_k is of the form $(\hat{r}_{i_1}.x\ \theta\ r_{i_2}.y + c)$, then $e = (\hat{r}_{i_1}, r_{i_2}, j, C_k)$.
   - if C_k is of the form $(r_{i_1}.x\ \theta\ \hat{r}_{i_2}.y + c)$, then $e = (r_{i_1}, \hat{r}_{i_2}, j, C_k)$.
   - if C_k is of the form $(r.x_1\ \theta\ c_1) \wedge \cdots \wedge (r.x_u\ \theta\ c_u)$, then $e = (r, r, j, C_k)$.
   - if C_k is of the form $(\hat{r}.x_1\ \theta\ c_1) \wedge \cdots \wedge (\hat{r}.x_u\ \theta\ c_u)$, then ignore it since the relation $\hat{r}$ contains only relevant tuples. □

At the end of this process, we will have an undirected multi-query graph $G = (V, E)$ with $|V| = p + q$ and $|E| \leq m(2^q - 1)$, where m is the number of disjoint predicates in $C(Y)$.

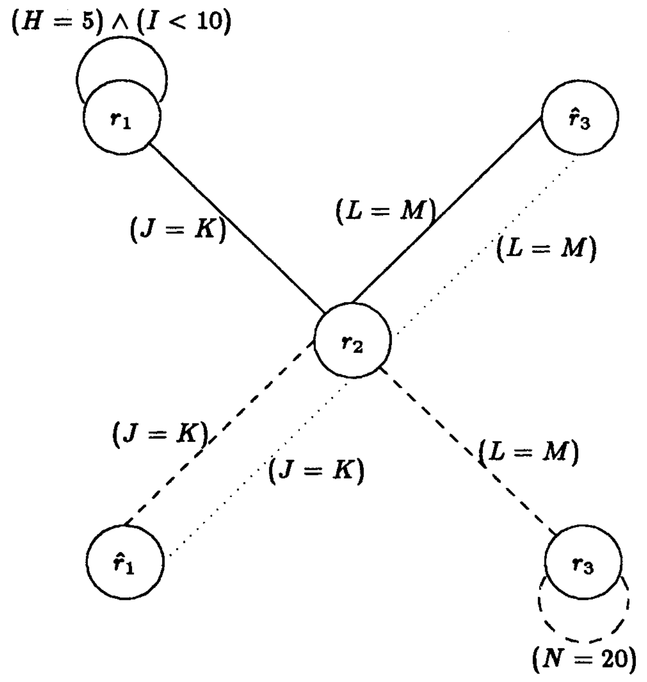


Figure 6.3: Multi-query graph for Δv

Example 6.3 Let $E_1 = \pi_{KN}(\sigma_{(J=K)\wedge(L=M)\wedge(H=5)\wedge(I<10)\wedge(N=20)}(R_1 \times R_2 \times R_3))$ be a view defined on relation schemes $R_1(H, I, J)$, $R_2(K, L)$, $R_3(M, N, O)$, and v be the latest materialization of the view. If relations r_1 and r_3 are updated, then Δv will be given by

$$\begin{aligned}\Delta v \quad = \quad & \pi_{KN}(\sigma_{(J=K)\wedge(L=M)\wedge(H=5)\wedge(I<10)\wedge(N=20)}(r_1 \times r_2 \times \hat{r}_3)) \quad \cup \\ & \pi_{KN}(\sigma_{(J=K)\wedge(L=M)\wedge(H=5)\wedge(I<10)\wedge(N=20)}(\hat{r}_1 \times r_2 \times r_3)) \quad \cup \\ & \pi_{KN}(\sigma_{(J=K)\wedge(L=M)\wedge(H=5)\wedge(I<10)\wedge(N=20)}(\hat{r}_1 \times r_2 \times \hat{r}_3))\end{aligned}$$

The corresponding multi-query graph for Δv is shown in Figure 6.3. The condition $C(Y)$, where $Y = \{H, I, J, K, L, M, N\}$, contains four disjoint predicates: $C_1 \equiv (J = K)$, $C_2 \equiv (L = M)$, $C_3 \equiv (H = 5) \wedge (I < 10)$, and $C_4 \equiv (N = 20)$. Notice that the edges representing predicates over relations $\hat{r}_1$ and $\hat{r}_3$ need not be shown in the graph. □

Having decided on a query representation, the next step is to state the problem more precisely in terms of this representation. Ideally, we are interested on finding the optimum sequence of operations that computes the set of queries

$\{v_1, v_2, \ldots, v_{2^q-1}\}$. If we regard an operation to be performed as an edge that exists in the multi-query graph and the removal of the edge as the execution of the operation it represents, then our objective is to find the sequence of removal of edges that computes the set of queries of interest with the minimum cost.

We could associate a cost with each of the operations represented by the edges in the graph. The cost function could incorporate I/O cost as well as CPU cost.

Finding the order of operations with minimum cost seems to be hard. We could enumerate all possible orders of execution and pick the one of minimal cost. We could also use some heuristic algorithms to avoid the combinatorial growth of exhaustive search. In the next two sections we propose two alternatives which are aimed at finding sequences for computing our multiple queries at a cost which is less than computing the optimal sequence of operations of each query independently. Section 6.6 presents an alternative based on the query decomposition idea by Wong and Youssefi [WONG76] extended to handle multiple queries by Chakravarthy and Minker [CHAKR86]. The algorithm presented in Section 6.6 uses the multi-query graph to help detect common subexpressions among the queries to be optimized; when the multi-query graph evolves to a graph containing only disconnected components with single edges, the decomposition is carried out according to the optimal nesting order for each of the subexpressions represented by the disconnected components.

Section 6.7 presents an alternative form of query optimization based on the space search methods originally proposed by Grant and Minker [GRANT82]. The efforts in this approach have been focused on trying to improve the algorithms that search the solution space. In their paper, Grant and Minker propose a branch and bound type of algorithm and suggest that the search time can be improved using the Algorithm A*. Sellis [SELL86a,SELL86b] explores the idea of using the Algorithm A* to search the solution space and proposes a cost estimate that helps the algorithm to find (on average) an optimal solution faster than Grant and Minker's. However, both papers ignore the problem of actually generating the set of plans to execute each of the queries to be optimized. In Section 6.7 we present an idea on how to generate promising query plans and then give a generalized algorithm that searches the solution space using any desired search space method.

Although the ideas presented in Sections 6.6 and 6.7 are given in the context of the multiple query optimization problem obtained from our differential approach to updating materialized views, they can be used to improve the execution of multiple queries in a more general setting.

6.6 A solution using query decomposition

Query decomposition was originally proposed by Wong and Youssefi [WONG76] as the query processing strategy for the relational language QUEL [STONE76]. In this method, the query to be optimized is represented by a connection graph

[ULLMA82]. The execution of the query can be seen as a series of operations on the connection graph. Each operation has the effect of constructing a new relation used as an intermediate result in the evaluation of the query, as well as simplifying the graph itself. At each step during the decomposition there are two operations that can be applied to the connection graph called *instantiation* and *(tuple) substitution* or *iteration*. These operations on the graph correspond to edge removal and node removal and the goal is to transform the connection graph into a graph with no edges.

- Instantiation is analogous to pushing the selections and projections towards the leaves of the query's parse tree. It actually forces the application of a selection and a projection on a single relation early in the evaluation of the query. The purpose of this operation is to reduce the size of an operand relation to be used later in a Cartesian product.
- Tuple substitution corresponds to performing a Cartesian product. The effect of this operation is to dissect the graph by eliminating a node from the connection graph and replacing it by each of the tuples in the relation represented by that node.

Query decomposition actually produces a family of algorithms to process a query. Each algorithm results from different orders in which instantiation and tuple substitution are applied to the connection graph. (Several heuristics have been proposed to decide at each step whether to instantiate or to substitute. Youssefi and Wong [YOUSS79] present an empirical study of different heuristics and conclude that instantiation is usually better than iteration as a first move in the decomposition.)

Chakravarthy and Minker [CHAKR86] have extended the query decomposition method to the case of a multi-query graph.

When we apply query decomposition to a multi-query graph, we may reach a point in which the graph consists of several disconnected components all having single edges between pairs of nodes. When that occurs, the decomposition can be carried through in the same way as in single query optimization for each of the disconnected components of the graph. Since we assume that all the queries to be optimized are represented by chains, we can make use of the single query optimizer to provide us with an optimal nesting order for the portion of the queries still to be evaluated within each component. In this section we propose a hybrid algorithm which involves this form of mixture of query decomposition and Algorithm 6.1. Before we describe the algorithm we need a few definitions.

Given the undirected multi-query graph $G = (V, E)$. Let $adjacent(r)$ be the set of vertices adjacent to node r, that is, $adjacent(r) = \{r' \mid \exists\, j, \rho\ (r, r', j, \rho) \in E \vee (r', r, j, \rho) \in E, 1 \leq j \leq 2^q - 1\}$. Let $edges(r, r')$ be the set of all edges between nodes r and r', that is, $edges(r, r') = \{e \mid \exists\, j, \rho\ e = (r, r', j, \rho) \in E, 1 \leq j \leq 2^q - 1\}$. Let $\nu(r, r')$ be a function defined as follows:

$$\nu(r, r') = \begin{cases} 0 & \text{if } |edges(r, r')| = 0 \\ |edges(r, r')| - 1 & \text{if } |edges(r, r')| \geq 1 \end{cases}$$

We define a function *savings*(r) to be the savings in cost provided by iterating on relation r as follows:

$$savings(r) = \sum_{\forall r' \in adjacent(r)} cost(r, r') * \nu(r, r'),$$

where again, $cost(r, r') = |r| * |r'|$ is the estimated cost of performing a join between relations r and r'. The estimated size of the result of a join between relations r and r' according to the predicate ρ is given by $est\_size(r, r', \rho) = |r| * |r'| * sel(\rho)$.

Briefly, what the function *savings* is trying to capture is the following. Suppose that there is a pair of nodes connected by three edges representing the same predicate, meaning that there are three queries sharing the same subexpression. If the system computes the expression defined by one of the edges, then the system will not need to compute the same expression for the other two queries again (provided the result is stored for later use), which basically saves twice the cost of computing the expression. Savings are obtained when there are at least two edges representing the same predicate between two nodes.

Algorithm 6.3. A hybrid algorithm between query decomposition and optimal nesting order for joins to solve a multiple query optimization problem.

Input: a set of queries $\{v_1, v_2, \ldots, v_{2^q-1}\}$ and its corresponding multi-query graph $G = (V, E)$.

Output: a program that computes the expressions represented by the multi-query graph.

Repeat choosing the lowest numbered option among the following set of options.

1. Instantiate whenever possible.

2. Iterate on the relation that provides the highest cost saving. That is, iterate on the relation represented by node r such that $savings(r) = max(savings(r_i)) > 0$, $1 \leq i \leq p$.

   If there is a tie between nodes r and r', then iterate on node r if

   $$\sum_{\forall s \in adjacent(r)} \nu(r, s) \geq \sum_{\forall t \in adjacent(r')} \nu(r', t),$$

   otherwise iterate on r'. Notice that with this tie breaker we are favoring the node that will lead to a faster dissection of the multi-query graph.

3. The algorithm reaches this point when $savings(r) = 0$, $\forall r \in V$. However, a node may still be connected to other nodes through single edges of several colors. Iterate on the node connected to the largest number of edges of different colors. If there are ties, then iterate on the node representing the relation with smallest cardinality.

4. When the algorithm reaches this point, the remaining portion of the graph consists of a number of disconnected components where each component contains edges of only one color. That means there are no more common subexpressions to be shared. Therefore, for $j = 1$ to $j = 2^q - 1$: Use Algorithm 6.1 to find an optimal nesting order for computing the joins represented by the remaining edges in query j which are defined by the set $\{e \mid e = (r, r', j, \rho) \in E\}$ and then iterate on the relations for the component j according to the optimal nesting order.

Until the multi-query graph has no edges. □

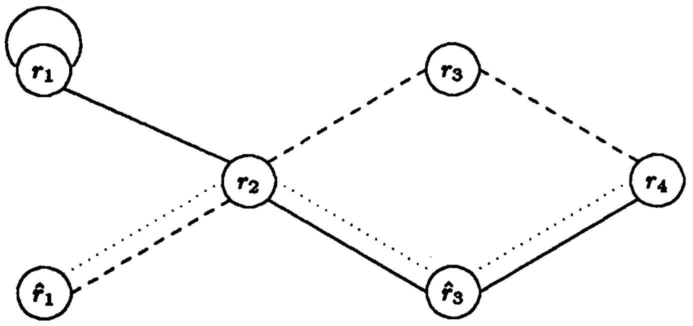
It should be noted that Algorithm 6.3, provides a more precise set of heuristics than other query decomposition algorithms for single queries presented in [MAIER83a,ULLMA82], and for multiple queries in [CHAKR86]. In particular, by making use of the selectivities in the predicates as well as the cardinalities of the relations, we are able to eliminate the nondeterminism present in the other decomposition algorithms mentioned. To confirm what was said in the last paragraph of the previous section, we are using decomposition to take advantage of the common subexpressions among the queries and when there are no more common subexpressions we rely on a more accurate optimization of the individual queries still to be computed.

The following example uses the description of instantiation and iteration for multi-query graphs introduced in [CHAKR86]. They are shown in detail here in order to illustrate each of the steps of Algorithm 6.3.

Example 6.4 Consider four relation schemes $R_1(H, I)$, $R_2(J, K)$, $R_3(L, M)$, and $R_4(N, O)$ and the following set of expressions:

$$\begin{aligned}
v_1 &= \sigma_{(H<10)\wedge(I=J)\wedge(K=L)\wedge(M=N)}(r_1 \times r_2 \times \hat{r}_3 \times r_4) \\
v_2 &= \sigma_{(I=J)\wedge(K=L)\wedge(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4) \\
v_3 &= \sigma_{(I=J)\wedge(K=L)\wedge(M=N)}(\hat{r}_1 \times r_2 \times \hat{r}_3 \times r_4)
\end{aligned}$$

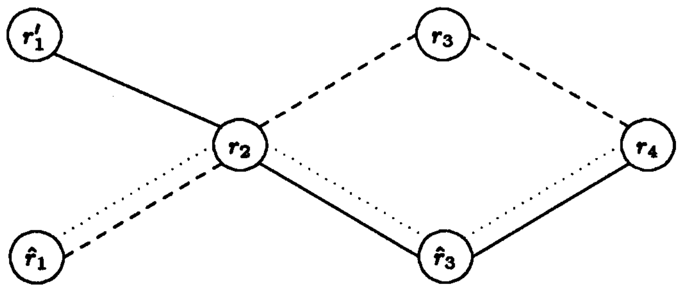
with its corresponding multi-query graph:



where the expression v_1 is represented by solid edges, expression v_2 by dashed edges, and expression v_3 by dotted edges. The expressions v_1, v_2, and v_3 are generated by the differential re-evaluation algorithm for the case in which relations r_1 and r_3 have been updated since the latest materialization of the view. As always, relations $\hat{r}_1$ and $\hat{r}_3$ represent the set of net changes on relations r_1 and r_3, respectively.

Assume that $|r_1| = 1000$ pages, $|r_2| = 80$ pages, $|r_3| = 90$ pages, $|r_4| = 200$ pages, $|\hat{r}_1| = 1$ page, $|\hat{r}_3| = 2$ pages, $sel_{11} = 0.1$, $sel_{12} = 0.1$, $sel_{23} = 0.15$, and $sel_{34} = 0.03$.

In the first iteration of the algorithm, the action taken is to instantiate relation r_1. This action removes the edge $(r_1, r_1, 1, (H < 10))$ producing a new node r'_1. The resulting graph is:



and the statement generated in this step is:

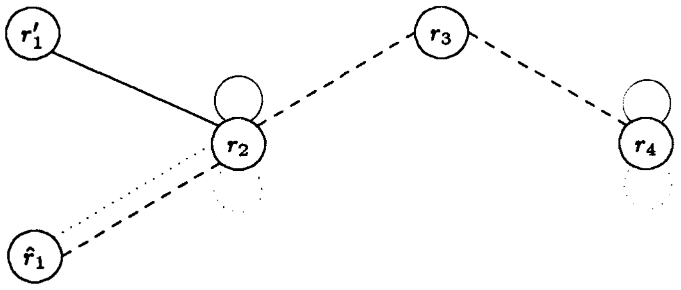
$$
\begin{aligned}
r'_1 &\leftarrow \sigma_{(H<10)}(r_1);\\
v_1 &\leftarrow \sigma_{(I=J)(K=L)(M=N)}(r'_1 \times r_2 \times \hat{r}_3 \times r_4);\\
v_2 &\leftarrow \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4);\\
v_3 &\leftarrow \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times \hat{r}_3 \times r_4);
\end{aligned}
$$

The size of relation r'_1 is estimated to be $sel_{11} * |r_1| = (0.1)(1000) = 100$ pages.

In the second iteration of the algorithm no more instantiations are possible, therefore Step 2 is performed. In this step, the function *savings* is computed for each of the nodes of the graph. The values of this function for each of the relations is:

$$
\begin{aligned}
savings(r'_1) &= 0\\
savings(\hat{r}_1) &= 80\\
savings(r_2) &= 240\\
savings(r_3) &= 0\\
savings(\hat{r}_3) &= 560\\
savings(r_4) &= 400
\end{aligned}
$$

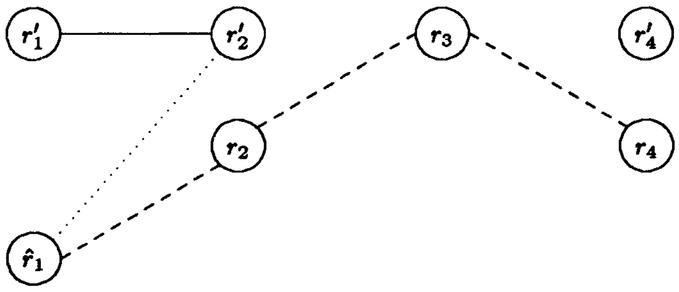
Thus, the next action taken by the algorithm is to iterate over relation $\hat{r}_3$. This action removes the edges $(r_2, \hat{r}_3, 1, (K = L))$, $(r_2, \hat{r}_3, 3, (K = L))$, $(\hat{r}_3, r_4, 1, (M = N))$, and $(\hat{r}_3, r_4, 3, (M = N))$, and the resulting graph is:



The statement generated by this step is:

```
$r_1' \leftarrow \sigma_{(H<10)}(r_1)$;
$v_1 \leftarrow \emptyset$; $v_3 \leftarrow \emptyset$;
for each $t$ in $\hat{r}_3$ do
    $v_1^2 \leftarrow \sigma_{(I=J)(K=t[L])(t[M]=N)}(r_1' \times r_2 \times r_4)$;
    $v_1 \leftarrow v_1 \cup (v_1^2 \times \{t\})$;
    $v_3^2 \leftarrow \sigma_{(I=J)(K=t[L])(t[M]=N)}(\hat{r}_1 \times r_2 \times r_4)$;
    $v_3 \leftarrow v_3 \cup (v_3^2 \times \{t\})$;
od;
$v_2 \leftarrow \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$;
```

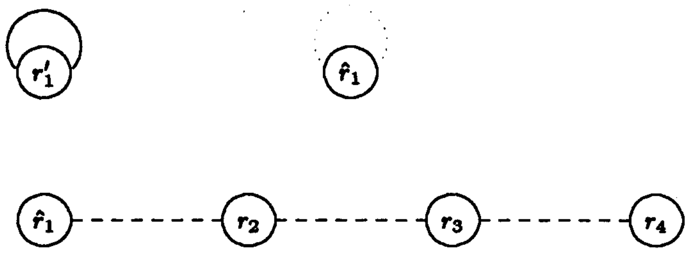
In the third and fourth iterations of the algorithm relations r_2 and r_4 are instantiated. The resulting graph is:



and the resulting program is:

```
$r'_1 \leftarrow \sigma_{(H<10)}(r_1)$;
$v_1 \leftarrow \emptyset$; $v_3 \leftarrow \emptyset$;
**for each** $t$ **in** $\hat{r}_3$ **do**
  $r'_2 \leftarrow \sigma_{(K=t[L])}(r_2)$;
  $r'_4 \leftarrow \sigma_{(t[M]=N)}(r_4)$;
  $v_1^2 \leftarrow \sigma_{(I=J)}(r'_1 \times r'_2 \times r'_4)$;
  $v_1 \leftarrow v_1 \cup (v_1^2 \times \{t\})$;
  $v_3^2 \leftarrow \sigma_{(I=J)}(\hat{r}_1 \times r'_2 \times r'_4)$;
  $v_3 \leftarrow v_3 \cup (v_3^2 \times \{t\})$;
**od**;
$v_2 \leftarrow \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$;
```

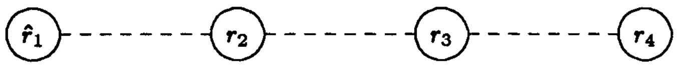
In the fifth iteration of the algorithm, step 3 is performed and the action taken is to iterate on relation r'_2. The resulting graph is:



and the corresponding generated program is:

```
$r'_1 \leftarrow \sigma_{(H<10)}(r_1)$;
$v_1 \leftarrow \emptyset$; $v_3 \leftarrow \emptyset$;
**for each** $t$ **in** $\hat{r}_3$ **do**
  $r'_2 \leftarrow \sigma_{(K=t[L])}(r_2)$;
  $r'_4 \leftarrow \sigma_{(t[M]=N)}(r_4)$;
  $v_1^2 \leftarrow \emptyset$; $v_3^2 \leftarrow \emptyset$;
  **for each** $s$ **in** $r'_2$ **do**
    $v_1^4 \leftarrow \sigma_{(I=s[J])}(r'_1 \times r'_4)$;
    $v_1^2 \leftarrow v_1^2 \cup (v_1^4 \times \{s\})$;
    $v_3^4 \leftarrow \sigma_{(I=s[J])}(\hat{r}_1 \times r'_4)$;
    $v_3^2 \leftarrow v_3^2 \cup (v_3^4 \times \{s\})$;
  **od**;
  $v_1 \leftarrow v_1 \cup (v_1^2 \times \{t\})$;
  $v_3 \leftarrow v_3 \cup (v_3^2 \times \{t\})$;
**od**;
$v_2 \leftarrow \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$;
```

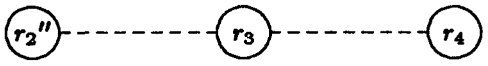
In the sixth iteration of the algorithm relations r_1' and $\hat{r}_1$ are instantiated. The resulting graph is:



and the corresponding program is:

```
$r_1' \leftarrow \sigma_{(H<10)}(r_1)$;
$v_1 \leftarrow \emptyset$; $v_3 \leftarrow \emptyset$;
for each $t$ in $\hat{r}_3$ do
   $r_2' \leftarrow \sigma_{(K=t[L])}(r_2)$;
   $r_4' \leftarrow \sigma_{(t[M]=N)}(r_4)$;
   $v_1^2 \leftarrow \emptyset$; $v_3^2 \leftarrow \emptyset$;
   for each $s$ in $r_2'$ do
      $r_1'' \leftarrow \sigma_{(I=s[J])}(r_1')$;
      $\hat{r}_1' \leftarrow \sigma_{(I=s[J])}(\hat{r}_1)$;
      $v_1^4 \leftarrow (r_1'' \times r_4')$;
      $v_1^2 \leftarrow v_1^2 \cup (v_1^4 \times \{s\})$;
      $v_3^4 \leftarrow (\hat{r}_1' \times r_4')$;
      $v_3^2 \leftarrow v_3^2 \cup (v_3^4 \times \{s\})$;
   od;
   $v_1 \leftarrow v_1 \cup (v_1^2 \times \{t\})$;
   $v_3 \leftarrow v_3 \cup (v_3^2 \times \{t\})$;
od;
$v_2 \leftarrow \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$;
```

At the seventh iteration of the algorithm, we are left with a component consisting of single edges all of the same color. At this point, we can make use of the optimal nesting order for the expression represented by this component to guide the instantiation and iteration processes. Now, since the estimated size of the subexpression $\hat{r}_1 \times r_2$ is given by $|\hat{r}_1| * |r_2| * sel_{12} = (80)(1)(0.1) = 8$ which is not greater than both of $|\hat{r}_1|$ and $|r_2|$ then the appropriate action is to iterate on either of these two relations and then save the result for the rest of the computation. We iterate over relation $\hat{r}_1$ and then at the eighth iteration of the algorithm we instantiate relation r_2. The resulting graph of this step is:



and the corresponding generated code (replacing the last line of the previous code) is:

```
$v_2 \leftarrow \emptyset$;
for each $u$ in $\hat{r}_1$ do
    $r_2'' \leftarrow \sigma_{(u[I]=J)}(r_2)$;
    $v_2^7 \leftarrow \sigma_{(K=L)(M=N)}(r_2'' \times r_3 \times r_4)$;
    $v_2 \leftarrow v_2 \cup (v_2^7 \times \{u\})$;
od;
```

At the ninth iteration of the algorithm, the alternative is again to iterate on either r_2'' or r_3 and proceed exactly as in the previous stage. However, in this case the size of the result $r_2'' \times r_3$ is greater than the size of both r_2'' and r_3, namely $|r_2''| * |r_3| * sel_{23} = (8)(90)(.15) = 108$; therefore, it is better to iterate over r_3 and dissect the graph. Finally, at the tenth iteration of the algorithm we instantiate relations r_2'' and r_4.

The code generated for computing the expression

$$v_2 = \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$$

is:

```
$v_2 \leftarrow \emptyset$;
for each $u$ in $\hat{r}_1$ do
    $r_2'' \leftarrow \sigma_{(u[I]=J)}(r_2)$;
    $v_2^7 \leftarrow \emptyset$;
    for each $w$ in $r_3$ do
        $r_2''' \leftarrow \sigma_{(K=w[L])}(r_2'')$;
        $r_4' \leftarrow \sigma_{(w[M]=N)}(r_4)$;
        $v_2^{11} \leftarrow (r_2''' \times r_4')$;
        $v_2^7 \leftarrow v_2^7 \cup (v_2^{11} \times \{w\})$;
    od;
    $v_2 \leftarrow v_2 \cup (v_2^7 \times \{u\})$;
od;
```

The complete code for the set of queries is therefore:

```
$r_1' \leftarrow \sigma_{(H<10)}(r_1)$;             $cost = 1000$    $|r_1'| = 1000 * 0.1 = 100$
$v_1 \leftarrow \emptyset$; $v_3 \leftarrow \emptyset$;
for each $t$ in $\hat{r}_3$ do
    $r_2' \leftarrow \sigma_{(K=t[L])}(r_2)$;        $cost = 80$      $|r_2'| = 80 * 0.15 = 12$
    $r_4' \leftarrow \sigma_{(t[M]=N)}(r_4)$;        $cost = 200$     $|r_4'| = 200 * 0.03 = 6$
    $v_1^2 \leftarrow \emptyset$; $v_3^2 \leftarrow \emptyset$;
    for each $s$ in $r_2'$ do
        $r_1'' \leftarrow \sigma_{(I=s[J])}(r_1')$;   $cost = 100$     $|r_1''| = 100 * 0.1 = 10$
```

```
      $\hat{r}'_1 \leftarrow \sigma_{(I=s[J])}(\hat{r}_1)$;           $cost = 1$        $|\hat{r}'_1| = 1$
      $v_1^4 \leftarrow (r''_1 \times r'_4)$;                $cost = 60$       $|v_1^4| = 60$
      $v_1^2 \leftarrow v_1^2 \cup (v_1^4 \times \{s\})$;        *ignored*
      $v_3^4 \leftarrow (\hat{r}'_1 \times r'_4)$;                $cost = 6$        $|v_3^4| = 6$
      $v_3^2 \leftarrow v_3^2 \cup (v_3^4 \times \{s\})$;        *ignored*
    od;                                  $cost = (100 + 1 + 60 + 6) * 12 = 2004$
    $v_1 \leftarrow v_1 \cup (v_1^2 \times \{t\})$;
    $v_3 \leftarrow v_3 \cup (v_3^2 \times \{t\})$;
  od;                                    $cost = (80 + 200 + 2004) * 2 = 4568$
  $v_2 \leftarrow \emptyset$;
  for each $u$ in $\hat{r}_1$ do
    $r''_2 \leftarrow \sigma_{(u[I]=J)}(r_2)$;            $cost = 80$       $|r''_2| = 80 * 0.1 = 8$
    $v_2^7 \leftarrow \emptyset$;
    for each $w$ in $r_3$ do
      $r'''_2 \leftarrow \sigma_{(K=w[L])}(r''_2)$;        $cost = 8$        $|r'''_2| = 8 * 0.15 = 1.2 \approx 2$
      $r'_4 \leftarrow \sigma_{(w[M]=N)}(r_4)$;          $cost = 200$      $|r'_4| = 200 * 0.03 = 6$
      $v_2^{11} \leftarrow (r'''_2 \times r'_4)$;             $cost = 12$       $|v_2^{11}| = 12$
      $v_2^7 \leftarrow v_2^7 \cup (v_2^{11} \times \{w\})$;      *ignored*
    od;                                  $cost = (8 + 200 + 12) * 90 = 19800$
    $v_2 \leftarrow v_2 \cup (v_2^7 \times \{u\})$;           *ignored*
  od                                     $cost = (80 + 19800) * 1 = 19880$;
```

The cost of the strategy for computing the set of queries v_1, v_2, and v_3 given by Algorithm 6.3 is given by: $cost = 1000 + 4568 + 19880 = 25448$. In contrast, the cost of performing each of the queries independently is 16760 for v_1, 22400 for v_2 and 576 for v_3 for a total of 39736. Therefore, using decomposition we obtain a 36% saving for the set of queries in this example. □

6.7 A solution using space search methods

The idea of using space search methods for multiple-query optimization was first applied by Grant and Minker [GRANT82] in the context of deductive databases. In particular, they proposed a variant of a branch and bound algorithm in which the initial approximation to the solution as well as the expansion of nodes proceeds in a depth first fashion. Improvements on the cost estimation have been proposed by Sellis [SELL86b].

Using Grant and Minker's notation, we are given a set of queries v_i, $1 \leq i \leq n$, whose evaluation is to be optimized globally; the plans P_{ij}, $1 \leq j \leq p_i$, for evaluating each query v_i; the distinct atomic tasks t_{ij}^k, $1 \leq k \leq q_{ij}$, which comprise each plan P_{ij}; and the actual or estimated cost $cost(t_{ij}^k)$ for each task. Equivalent tasks among plans are assumed to be known. The objective is to find a sequence of plans $\langle P_{1k_1}, P_{2k_2}, \ldots, P_{nk_n} \rangle$, whose cost is minimal.

Good candidates for the plans P_{ij} are the locally optimal plans and the ones that use common subexpressions among the given queries. In our particular query optimization problem, finding all plans that share common subexpressions is an exponential process. However, since all queries to be optimized are represented by the same relational expression, a plan to execute one query can be used to execute any other query provided that the appropriate operand relations are used. Therefore, we propose to use all different locally optimal plans among the $2^q - 1$ queries as the set of alternative plans for each query and then use a search space method such as Grant and Minker's algorithm to find the set of plans whose cost is minimal.

Consider a solution vector $S_k = \langle P_{1k_1}, P_{2k_2}, \ldots, P_{nk_n} \rangle$. The cost of the solution vector S_k is given by

$$cost(S_k) = \sum_{t \in \bigcup_{i=1}^{n} P_{ik_i}} cost(t)$$

The *coalesced cost* on tasks [GRANT82] is given by

$$coalesced\_cost(t) = \frac{cost(t)}{n_q}$$

where n_q is the number of queries in which task t occurs. Similarly, for plans we have

$$coalesced\_cost(P_{ij}) = \sum_{k=1}^{q_{ij}} coalesced\_cost(t_{ij}^k)$$

A description of the multiple query optimization algorithm using search space methods can be given as follows.

Algorithm 6.4

Input:

i) a set of queries v_i, $1 \leq i \leq n$,

ii) the plans P_{ij}, $1 \leq j \leq p_i$, for evaluating each query v_i, and

iii) the tasks t_{ij}^k along with their costs $cost(t_{ij}^k$, $1 \leq k \leq q_{ij}$, comprising each plan P_{ij}.

Output: a sequence of plans $\langle P_{1k_1}, P_{2k_2}, \ldots, P_{nk_n} \rangle$, whose cost is minimal.

1. For each query v_i, $1 \leq i \leq n$, find the optimal execution plan.

2. For each query v_i generate the alternative plans P_{ij} based on the different plans generated in Step 1. At this step identify all equivalent tasks among the different plans, and find the "actual" cost associated with each task.

3. Compute the estimated costs for tasks and plans in the optimal solution vector (e.g., coalesced costs).
4. Run a search space algorithm to find a solution vector of minimal cost (e.g., Grant and Minker's algorithm). □

Let us apply the above algorithm to the queries in Example 6.4. We first have to find the locally optimal access plans for each of the three queries repeatedly by applying Algorithm 6.1. For the query

$$v_1 = \sigma_{(H<10)(I=J)(K=L)(M=N)}(r_1 \times r_2 \times \hat{r}_3 \times r_4)$$

the optimal access plan is given by the sequence of tasks

$$\begin{aligned}
&s_1 \leftarrow \sigma_{(H<10)}(r_1)\\
&s_2 \leftarrow \sigma_{(M=N)}(\hat{r}_3 \times r_4)\\
&s_3 \leftarrow \sigma_{(K=L)}(r_2 \times s_2)\\
&s_4 \leftarrow \sigma_{(I=J)}(s_1 \times s_3)
\end{aligned}$$

For the query

$$v_2 = \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$$

there are two optimal access plans given below. The first plan is:

$$\begin{aligned}
&s_1 \leftarrow \sigma_{(I=J)}(\hat{r}_1 \times r_2)\\
&s_2 \leftarrow \sigma_{(K=L)}(s_1 \times r_3)\\
&s_3 \leftarrow \sigma_{(M=N)}(s_2 \times r_4)
\end{aligned}$$

and the second plan is:

$$\begin{aligned}
&s_1 \leftarrow \sigma_{(I=J)}(\hat{r}_1 \times r_2)\\
&s_2 \leftarrow \sigma_{(M=N)}(r_3 \times r_4)\\
&s_3 \leftarrow \sigma_{(K=L)}(s_1 \times s_2)
\end{aligned}$$

An finally for the query

$$v_3 = \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times \hat{r}_3 \times r_4)$$

the optimal access plan is:

$$\begin{aligned}
&s_1 \leftarrow \sigma_{(I=J)}(\hat{r}_1 \times r_2)\\
&s_2 \leftarrow \sigma_{(M=N)}(\hat{r}_3 \times r_4)\\
&s_3 \leftarrow \sigma_{(K=L)}(s_1 \times s_2)
\end{aligned}$$

Notice that apart from the selection on relation r_1 required in query v_1 the locally optimal access plans provide three different sequences for performing the joins. Since all the queries in our multiple query optimization problem share the same join predicates, we propose to use these local optimal access plans as alternative plans for each of the queries and then use a search space method to obtain the globally optimal set of plans.

The plans to consider are given in Table 6.1 and the actual costs are given in Table 6.2. The equivalent tasks are:

$$
\begin{aligned}
&t^1_{11} \equiv t^1_{12} \equiv t^1_{13} \equiv \sigma_{(H<10)}(r_1),\\
&t^1_{21} \equiv t^2_{23} \equiv \sigma_{(M=N)}(r_3 \times r_4),\\
&t^1_{22} \equiv t^1_{23} \equiv t^1_{32} \equiv t^1_{33} \equiv \sigma_{(I=J)}(\hat{r}_1 \times r_2),\\
&t^2_{11} \equiv t^3_{13} \equiv t^1_{31} \equiv t^2_{33} \equiv \sigma_{(M=N)}(\hat{r}_3 \times r_4),\\
&t^2_{12} \equiv t^2_{13} \equiv \sigma_{(I=J)}(t^1_{12} \times r_2) \equiv \sigma_{(I=J)}(t^1_{13} \times r_2) \equiv, \text{ and}\\
&t^3_{11} \equiv t^2_{31} \equiv \sigma_{(K=L)}(r_2 \times t^1_{31}) \equiv \sigma_{(K=L)}(r_2 \times t^2_{11}).
\end{aligned}
$$

Plans	Tasks			
P_{11}	t^1_{11}	$t^2_{11} \equiv \sigma_{(M=N)}(\hat{r}_3 \times r_4)$	$t^3_{11} \equiv \sigma_{(K=L)}(r_2 \times t^2_{11})$	$t^4_{11} \equiv \sigma_{(I=J)}(t^1_{11} \times t^3_{11})$
P_{12}	t^1_{12}	$t^2_{12} \equiv \sigma_{(I=J)}(t^1_{12} \times r_2)$	$t^3_{12} \equiv \sigma_{(K=L)}(t^2_{12} \times \hat{r}_3)$	$t^4_{12} \equiv \sigma_{(M=N}(t^3_{12} \times r_4)$
P_{13}	t^1_{13}	$t^2_{13} \equiv \sigma_{(I=J)}(t^1_{13} \times r_2)$	$t^3_{13} \equiv \sigma_{(M=N)}(\hat{r}_3 \times r_4)$	$t^4_{13} \equiv \sigma_{(K=L}(t^2_{13} \times t^3_{13})$
P_{21}	t^1_{21}	$t^2_{21} \equiv \sigma_{(K=L)}(r_2 \times t^1_{21})$	$t^3_{21} \equiv \sigma_{(I=J)}(\hat{r}_1 \times t^2_{21})$	
P_{22}	t^1_{22}	$t^2_{22} \equiv \sigma_{(K=L)}(t^1_{22} \times r_3)$	$t^3_{22} \equiv \sigma_{(M=N)}(t^2_{22} \times r_4)$	
P_{23}	t^1_{23}	$t^2_{23} \equiv \sigma_{(M=N)}(r_3 \times r_4)$	$t^3_{23} \equiv \sigma_{(K=L)}(t^1_{23} \times t^2_{23})$	
P_{31}	t^1_{31}	$t^2_{31} \equiv \sigma_{(K=L)}(r_2 \times t^1_{31})$	$t^3_{31} \equiv \sigma_{(I=J)}(\hat{r}_1 \times t^2_{31})$	
P_{32}	t^1_{32}	$t^2_{32} \equiv \sigma_{(K=L)}(t^1_{32} \times \hat{r}_3)$	$t^3_{32} \equiv \sigma_{(M=N)}(t^2_{32} \times r_4)$	
P_{33}	t^1_{33}	$t^2_{33} \equiv \sigma_{(M=N)}(\hat{r}_3 \times r_4)$	$t^3_{33} \equiv \sigma_{(K=L)}(t^1_{33} \times t^2_{33})$	

Table 6.1: Alternative plans for queries v_1, v_2, and v_3.

Plans	Tasks				Total
P_{11}	$t^1_{11} = 1000$	$t^2_{11} = 400$	$t^3_{11} = 960$	$t^4_{11} = 14400$	16760
P_{12}	$t^1_{12} = 1000$	$t^2_{12} = 8000$	$t^3_{12} = 1600$	$t^4_{12} = 48000$	58600
P_{13}	$t^1_{13} = 1000$	$t^2_{13} = 8000$	$t^3_{13} = 400$	$t^4_{13} = 9600$	19000
P_{21}	$t^1_{21} = 18000$	$t^2_{21} = 43200$	$t^3_{21} = 6480$		67680
P_{22}	$t^1_{22} = 80$	$t^2_{22} = 720$	$t^3_{22} = 21600$		22400
P_{23}	$t^1_{23} = 80$	$t^2_{23} = 18000$	$t^3_{23} = 4320$		22400
P_{31}	$t^1_{31} = 400$	$t^2_{31} = 960$	$t^3_{31} = 144$		1504
P_{32}	$t^1_{32} = 80$	$t^2_{32} = 16$	$t^3_{32} = 600$		696
P_{33}	$t^1_{33} = 80$	$t^2_{33} = 400$	$t^3_{33} = 96$		576

Table 6.2: Costs for tasks and plans.

The estimated costs for tasks and plans based on the coalesced costs are given by Table 6.3.

Plans	Tasks				Total
P_{11}	$t_{11}^1 = 1000$	$t_{11}^2 = 200$	$t_{11}^3 = 480$	$t_{11}^4 = 14400$	16080
P_{12}	$t_{12}^1 = 1000$	$t_{12}^2 = 8000$	$t_{12}^3 = 1600$	$t_{12}^4 = 48000$	58600
P_{13}	$t_{13}^1 = 1000$	$t_{13}^2 = 8000$	$t_{13}^3 = 200$	$t_{13}^4 = 9600$	18800
P_{21}	$t_{21}^1 = 18000$	$t_{21}^2 = 43200$	$t_{21}^3 = 6480$		67680
P_{22}	$t_{22}^1 = 40$	$t_{22}^2 = 720$	$t_{22}^3 = 21600$		22360
P_{23}	$t_{23}^1 = 40$	$t_{23}^2 = 18000$	$t_{23}^3 = 4320$		22360
P_{31}	$t_{31}^1 = 200$	$t_{31}^2 = 480$	$t_{31}^3 = 144$		824
P_{32}	$t_{32}^1 = 40$	$t_{32}^2 = 16$	$t_{32}^3 = 600$		656
P_{33}	$t_{33}^1 = 40$	$t_{33}^2 = 200$	$t_{33}^3 = 96$		336

Table 6.3: Coalesced costs for tasks and plans.

Using Grant and Minker's algorithm, the solution vector for the above problem is given by $\langle P_{11}, P_{22}, P_{33}\rangle$ with a cost of 39256 which provides savings of 1.2% over the solution with no sharing of common subexpressions. For this particular problem, the solution using query decomposition is better than the solution using space search methods. A reason for the difference in cost between the two methods is the definition of tasks considered by each method. In query decomposition, a scan of one relation may be shared for the computation of several joins simultaneously, whereas the tasks defined for the space search involve only joins between two relations. In order to make fair comparisons between the two methods, they should consider the same type of tasks in the optimization.

6.8 Summary

In this chapter we have formally posed the problem of optimally finding the set of tuples to update a view using differential re-evaluation. We have presented two alternatives for this form of multiple query optimization. The first alternative is based on query decomposition for multiple queries where we presented a new set of heuristics for multi-query decomposition. The second alternative is based on space search methods based on query decomposition where we proposed to use the locally optimal access plans for the individual queries as the set of alternatives in each of the queries.

Chapter 7

Updating Materialized Views

7.1 Introduction

As explained early in this thesis, we can classify materialized views into two classes according to the frequency with which they are updated. A materialized view can be kept *consistently* updated with respect to its base relations; that is, every time a base relation is updated all its associated materialized views are updated as well. On the other hand, a materialized view can be updated *periodically* or on demand. The latter class of materialized view is usually called a *snapshot* [MYLOP75,TSICH77,ADIBA80], and the operation of bringing the snapshot up to date is referred to as a *snapshot refresh.*

The purpose of this chapter is twofold. First, we show how the results described in Chapters 3 through 6 can be used to support consistently up-to-date materialized views. Second, we analyze the issues involved in the support of snapshot refresh. We do not claim to give a definitive answer to the question of how snapshot refresh should be supported. Rather we want to summarize the main issues involved in the support of such an operation. A more definitive answer requires further analysis of the cost of the various alternatives.

There are three design objectives we want to keep in mind throughout this chapter: (1) we want to support materialized views defined by arbitrary *PSJ*-expressions, (2) we want to allow a single base relation to participate in the definition of multiple materialized views, and (3) we want the materialized view update mechanisms to be easily incorporated into a relational database management system. As in previous chapters, a materialized view is defined by a *PSJ*-expression represented by the triple $E = (\mathbf{A}, \mathbf{R}, C)$ or by the actual expression $\pi_{\mathbf{A}}(\sigma_C(R_1 \times R_2 \times \cdots \times R_p))$.

7.2 Consistently up-to-date materialized views

The material presented in Chapters 3 through 6 describe the essential components required to keep materialized views consistently up to date with respect to their underlying base relations.

It is assumed that relations are updated through transactions, and that the view update operation is performed as the last operation within the transaction. Clearly, the requirement of keeping a set of materialized views consistently up to date with the base relations will increase the update cost. Within a transaction we have available the update expressions applied to the base relations, as well as, the net effect of the update expression on the relations expressed as a set of inserted and/or deleted tuples. Therefore, if we use the complete set of tools developed in Chapters 3 through 6, a materialized view can be brought up to date by the following procedure:

1. For each materialized view E_i potentially affected by this transaction perform steps 2 to 6.
2. For each of the update expressions potentially affecting E_i, test whether the update is irrelevant or not. This step will partition the set of update expressions into two sets, one consisting of irrelevant updates and the other consisting of relevant updates. Irrelevant updates can then be ignored.
3. For each relevant update expression, test whether the effect is autonomously computable on E_i.
4. Perform each autonomously computable update.
5. For each update which is not autonomously computable, obtain the set of net changes on the base relations expressed as a set of tuples to be inserted into or deleted from the relations and then apply the differential re-evaluation algorithm. This will produce the set of (full) tuples that have to be inserted into or deleted from the view. Notice that the differential re-evaluation algorithm takes care of checking for irrelevant tuples as well.
6. Update the materialized view E_i using the set of tuples obtained in the previous step. □

Notice that the procedure described above makes use of tools that are based on the update expression, as well as tools that are based on the actual tuples resulting from the update expression. This may not always be practical. A better alternative may be to use the tools that are based on the update expression at the time the transaction is compiled and to use the tools that are based on the tuples at the time the transaction is executed.

Definition 7.1 Let E be a view definition and $\mathcal{T}$ a transaction consisting of a sequence of update expressions $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_n$, where each $\mathcal{U}_i$ is of the form INSERT

(R, T), DELETE $(R, \mathcal{C}_D)$, or MODIFY $(R, \mathcal{C}_M, \mathbf{F}_M)$. The transaction $\mathcal{T}$ is said to be *irrelevant* to E if every update $\mathcal{U}_i$, $1 \leq i \leq n$, is irrelevant to E.

Definition 7.2 Let $\mathcal{T}$ be a transaction consisting of a sequence of update expressions $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_n$, where each $\mathcal{U}_i$ is an update expression as before, and let $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$ be a view definition. The effect of the transaction $\mathcal{T}$ on $v(E, d)$ is said to be *unconditionally autonomously computable* if (for every database instance d) the effect of each update $\mathcal{U}_i$, $1 \leq i \leq n$, is unconditionally autonomously computable or irrelevant to the view.

Based on the above definitions a transaction could at compile time be classified as irrelevant or as autonomously computable. Irrelevant transactions will certainly require no extra effort to update materialized views defined on base relations which are updated within the transaction. Unconditionally autonomously computable transactions are especially useful in environments where a materialized view is stored at a site different from the site containing the base relations. The system will only have to send the transaction definition to the site where the materialized view is stored, and its update can be carried through locally at that site. If a transaction is not irrelevant or its effect is not autonomously computable, then we could still be able to apply differential re-evaluation to obtain the changes that have to be applied to the materialized view at the time the transaction is executed.

7.3 Snapshots

The main characteristic of a snapshot is that it is not updated immediately when a relation participating in its definition is updated. Snapshots are used to store a view of the database at some point in time, and therefore they may be out of date with respect to the underlying relations in the database. This means that many updates may have been applied to a relation in the database before a snapshot refresh is requested. Conversely, a relation in the database may participate in the definition of several snapshots, each of which may have been refreshed at a different time. This implies that the set of updates applied to a relation, which is relevant to the refresh of one snapshot, may not be the same set of updates relevant to the refresh of some other snapshot.

Work directly related to the support of snapshot refresh has been done in the context of System R* and is reported in a paper by Lindsay et al. [LINDS86]. They propose a differential algorithm for snapshot refresh for a limited class of snapshots. The class is restricted to snapshots defined by relational algebra expressions involving selection and projection. The main purpose of the algorithm is to reduce the amount of data sent to the remote site where the snapshot is stored.

An important issue in the design of algorithms to carry out the snapshot refresh operation consists of whether the "old values" of tuples from the underlying relations are available to the snapshot refresh mechanism or not. If the old values are available, then the snapshot refresh mechanism can be based on the differential re-evaluation approach to updating materialized views as described in Chapters 5 and 6.

Recall that the information required to carry out a differential re-evaluation of a materialized view consists of: (1) the view definition, (2) the state of the base relations that is consistent with the current state of the view, and (3) the set of net changes applied to the base relations since the latest materialization of the view.

A disadvantage of using a differential re-evaluation approach to perform a snapshot refresh is the possibility of incurring a high cost in setting up the information required by such an approach. The high cost comes from having to construct the state of the relations which is consistent with the latest snapshot refresh operation, as well as, having to collect the subsequent updates applied to the relation. Obviously, in order for this approach to be of benefit, the cost of setting up the information required by the differential re-evaluation plus the cost of performing the differential re-evaluation itself should not be higher than the complete re-evaluation of the expression defining the snapshot.

There may be situations where old values are not available to the snapshot refresh mechanism. In this case it is interesting to explore the support of the snapshot refresh operation based only on the latest state of the base relations plus some additional bookkeeping. The next subsections explore ways of supporting snapshot refresh both for the case when old values are available and for the case when they are not.

7.3.1 Snapshot refresh using old and new values

The main issue in this setting concerns the partitioning of each of the base relations that participate in the snapshot definition into two sets; one containing the "old tuples", i.e., the tuples considered in the latest snapshot refresh, and the other containing the "new tuples", i.e., the set of changes.

Several alternatives can be used to achieve the above partitioning:

1. combination of base relations and differential files,
2. combination of base relations and recovery data,
3. combination of base relations and some minimal amount of bookkeeping on the base relation itself, and
4. a database that fully supports the notion of time.

We now turn to discussing each of the above alternatives.

Base relations and differential files

Differential files [SEVER76] represent a potentially good mechanism for supporting snapshot refresh. The idea is to associate a differential file with each relation in the database. The differential file contains all recent changes to the relation. This idea was suggested by Roussopoulos and Kang [ROUSS86] for the support of view indices in a mainframe-workstation architecture. In their design, they propose a "lazy update" of view indices. That means that the indices are not updated every time the base relations are updated but rather every time the view is accessed. This type of update policy for views corresponds to refresh on demand. A view index is a special case of a materialized view and thus the issues discussed in this chapter apply to the maintenance of view indices as well.

A differential file contains a list of tuples inserted into or deleted from a base relation. To distinguish the inserted tuples from the deleted tuples within the differential file we need only one bit per tuple. The main problem with the differential file is how to identify the tuples in the differential file that have already been considered in a previous snapshot refresh. A simple way of solving this problem is by maintaining a pointer into the differential file (a "water level mark") per snapshot, indicating the point up to which changes have been considered with respect to the particular snapshot requesting the refresh [ROUSS86].

When a snapshot refresh request is made, the base relation along with all changes recorded from the beginning of the differential file up to the tuple before the water level mark represent the old state of the base relation. Tuples from the water level mark to the last tuple in the differential file represent the changes made to the base relation since the latest snapshot refresh.

If the size of the differential file is limited, then when the water level mark reaches the end of the file we can either force all snapshots affected to be refreshed immediately, or turn on an "invalid" indicator associated with each snapshot which forces complete re-evaluation next time a refresh request is made. All changes recorded in the differential file have to be made permanent into the base relation before the water level mark wraps around to the beginning of the file.

A variation of the differential file described above may be obtained by allowing subsequent modifications of the same tuple to be done in place within the differential file. This variation may provide a way of controlling the growth of the differential file but on the other hand it requires a full scan of the differential file every time there is snapshot refresh request.

A differential file may contain update expressions rather than the explicit tuples. This idea was suggested by Cammarata [CAMMA81] as a way of logging the updates applied to base relations and defer their application until the time the base relations were accessed. This represents another example of lazy evaluation of updates for base relations. Using this kind of differential files does not seem to be practical for the support of snapshot refresh because of the overhead involved in computing the updates to obtain the actual affected tuples every time a snapshot refresh request is made.

Base relations and recovery data

Here, by recovery data we mean the complete hierarchy of storage where recovery information is stored. According to Haerder and Reuter [HAERD83], three storage levels typically contain recovery data.

1. the *log buffer* is a designated area of main memory containing information about the most recent update activity that has taken place within the transaction currently being executed.
2. the *temporary log file* is a direct access storage file containing information useful for crash recovery. It contains information needed to reconstruct the most recent database buffer. It supports transaction UNDO, global UNDO, and partial REDO.
3. the *archive log file* is a sequential storage file containing information required for the support of global REDO after a media failure. It contains all changes committed to the database after the state reflected in the archive copy of the database obtained by the latest backup.

From the above sources of data we can easily discard alternatives 1 and 3. Alternative 1 is for a single transaction and therefore provides only the information required to support materialized views that are kept consistently up to date with the base relations and so the support of snapshots is not possible. Alternative 3 has the disadvantage of being available in sequential storage only, which can imply slow access. Therefore, the only source of recovery data that seems attractive for the support of snapshot refresh is the temporary log file.

There are two types of log information to support recovery actions as described by Haerder and Reuter, namely, physical logging and logical logging. Physical logging refers to the physical representation (bit pattern) written to the log. It may contain physical pages or bit patterns representing the state transition of a physical page. Logical logging refers to a higher level of log information representing the operation applied to records or tuples in the database, along with their arguments (values). The latter type of data is usually represented as a set of (transaction-id, tuple-id, field, old value, new value) tuples, as described by Gray et al. [GRAY81].

Both physical logging and logical logging provide enough information to the system to determine the changes that have been applied to the relations since the latest refresh. However, because we are using the differential re-evaluation approach to updating materialized views as the basic tool for the support of the snapshot refresh operation, logical logging may provide a more straightforward way of obtaining the changes to the base relations in the form of tuples.

The main problem faced when using recovery data as the source of information is how to efficiently compute the state of the base relation corresponding to the latest refresh of the snapshot, as well as the set of subsequent changes applied to the relation. Not only do we need to access the base relation to achieve this, but also we have to reconstruct the changed tuples from the data in the log. Since this may be time consuming, the idea of using the recovery data appears impractical, unless we impose some data structure on top of the log file which will allow efficient retrieval of the log records that are relevant to a given snapshot. This will probably incur a considerable overhead on the recovery management subsystem. The end result of this idea, however, will be faster location of the most recent changes applied to the base relations which is essentially a differential file for the whole database.

Base relations along with extra bookkeeping

In this alternative the idea is to use the base relations as the primary source of data for the snapshot refresh operation. This means, we want to use neither recovery data nor differential files but we still want to keep track of old values.

As the basis of this approach, a newly inserted tuple is appended to the relation, a deleted tuple is only marked as "deleted" but it is retained within the relation, and the modification of an already existing tuples is done by first deleting the old tuple and then inserting the new one. Retaining deleted tuples within a relation may be a feasible alternative when the base relations are updated only by insertion or deletions of tuples and the number of deletions is "small" compared to the amount of insertions. On the other hand, if the amount of deletions is comparable to the amount of insertions (which is the case when modifications are allowed), then keeping many deleted tuples in the relations may be impractical because of the waste of storage space. In addition, some criterion for determining when to actually eliminate deleted tuples is required.

To use differential re-evaluation for snapshot refresh, we must be able to identify the state of a relation which is consistent with the current snapshot as well as be able to identify the latest set of updates applied to the relation. If we were to support only one snapshot per base relation, then we could do it by attaching an extra bit of data to each tuple identifying whether the tuple was written after the snapshot refresh or not. But because we want to allow a base relation to participate in multiple snapshot definitions, and since snapshots may be refreshed at different times, one bit is not enough.

This problem can be solved by maintaining an additional attribute for each tuple in the database. This attribute contains a *timestamp* of latest write, an idea originally described by Lindsay et al. to support snapshot refresh [LINDS86]. The approach requires the system to maintain a timestamp of latest refresh for each snapshot. Whenever a snapshot refresh request is made, the tuples from the relations having a timestamp greater than the timestamp of the snapshot represent the set of changes to the relation since the latest refresh operation. Tuples from the base relation having a timestamp less than the timestamp of the snapshot, represent the state of the relation that is consistent with the current snapshot.

Under the scenario described, the procedure required to obtain the set of changes to be applied to refresh a snapshot is given as follows:

1. Partition each relation r_i that participates in the definition of the snapshot being refreshed into three sets r_{i_1}, r_{i_2}, and r_{i_3}. Set r_{i_1} contains all tuples inserted into or deleted from r_i whose timestamp is greater than the timestamp of the snapshot. Set r_{i_2} contains all non-deleted tuples whose timestamp is less than the timestamp of the snapshot. Set r_{i_3} contains all deleted tuples whose timestamp is less than the timestamp of the snapshot. Notice that by detecting the set r_{i_3} we are able to avoid processing tuples marked as deleted which must have already been deleted from the snapshot in the latest snapshot refresh.

2. Use the differential re-evaluation algorithm using the set of tuples r_{i_1} as the set of net changes and the set r_{i_2} as the old state of the relation. □

The tuples obtained in the previous stage represent the set of changes that must be applied to the snapshot to refresh it.

In general, the support of the snapshot refresh operation can be easily implemented in a database that supports the notion of time. Recently, several researches have proposed extensions to relational database management systems to support the notion of time [CLIFF83,SNODG84].

In this environment we require the snapshot to contain the time of latest refresh which is implicitly maintained by such historical database. The relations in the database contain the historical evolution of their tuples. When a snapshot refresh request is made, the state of the base relations consistent with the current state of the snapshot can be easily obtained by querying for the state of the relations at the time of the latest refresh. All changes that have occurred in the base relations since the latest refresh can also be obtained by querying the database to obtain the evolution of each relation from the time of the latest refresh operation was performed to the present time. Thus, all information required by the differential re-evaluation mechanism is readily available. However, the cost may be high.

7.3.2 Snapshot refresh using only new values

The scenario we have in mind in this subsection corresponds to the case when the relations in the database are updated and the "old values" corresponding to tuples that have been deleted from or modified in the relations are not available. In such a situation, the primary source of information to carry out the snapshot refresh operation consists of the latest state of the relations as well as the snapshot which is presumably out of date with respect to its underlying relations. This again means that we want to use neither recovery data nor differential files. However, we still want the snapshot refresh operation to avoid complete re-evaluation.

We assume that a newly inserted tuple is appended to the relation, a deleted tuple is deleted from the relation and its storage space is made available to subsequently inserted tuples, and the modification of an already existing tuple is done "in place". When a snapshot refresh request is made, the system only knows the new state of the base relations.

We still need to be able to determine which tuples from the underlying relations have been updated since the latest snapshot refresh. Thus, we require that each tuple from the relations contains a timestamp attribute indicating the time at which the the tuple was last written. We also assume that the system maintains a timestamp of latest refresh for each snapshot.

In this scenario we face two problems, namely, how to handle *deleted* and *modified* tuples. A deleted tuple frees storage space within the updated relation which will be occupied as soon as a new tuple is inserted leaving no trace of the deleted tuple. A modified tuple may in general produce insertions into, deletions from, and modifications to a snapshot. This requires the snapshot refresh mechanism to be able to identify which tuples in the snapshot should be deleted, inserted, or modified as a result of the modification of a tuple in the base relation.

To illustrate the above problems, consider a database d consisting of a single relation r on scheme R and a snapshot s defined by a select expression $E = (\alpha(R), \{R\}, C)$. Assume that: (1) the tuples from r are updated in place and that the old values are not retained or are not available to the snapshot refresh mechanism, and (2) the tuples from r have a special attribute containing a timestamp of latest write which allows the snapshot refresh mechanism to detect which tuples have been updated and which ones have not since the latest refresh of the snapshot. For a relation scheme $R_i \in D$, the attribute containing the timestamp is denoted by TS_i.

Let r' be an arbitrary instance of R such that it contains one tuple t' that has been modified since the latest refresh of s. Let r and t be the old instances of the relation and the tuple, respectively. Let $s = v(E, r)$ and $s' = v(E, r')$. Let P be a procedure that, based solely on E and r', determines a set of tuples T_d to be deleted from s and a set of tuples T_i to be inserted into s. In order for procedure P to correctly refresh the snapshot s, we must have $s' = (s - T_d) \cup T_i$.

Recall that the set of old values are not available to procedure P. Also, since we do not impose any constraints on the definition E, the identification of tuples to be inserted into or deleted from the snapshot should be based on the "full contents" of the affected tuples in the snapshot. Clearly, the set T_i should contain the tuple t' and the set T_d should contain the tuple t. The key problem is that $t = u$, for some tuple $u \in v(E, r)$, and the difficulty is that we cannot tell which tuple u to delete. Unless old values that uniquely identify the set of tuples to be deleted from the snapshot are available, we cannot identify the set T_d correctly. Hence, to be sure that the tuple t is in fact deleted, we must delete every tuple in the snapshot except, possibly, the tuples in $v(E, r')$. However, computing $v(E, r')$ is equivalent to a complete re-evaluation.

The above discussion implies that the system must be able to uniquely identify the correspondence between a modified tuple t' in r' and its associated old tuple $u \in v(E, r)$. Having an *immutable tuple-id attribute* associated with every relation scheme R, as well as with the set of attributes available in the snapshot, permits the unique identification of the set of tuples to be deleted from the snapshot. By immutable tuple-id attributes we mean attributes that uniquely identify the tuples and that cannot be changed by an update operation. If the key of R is visible in E, this may serve as the immutable tuple-id attribute.

In summary, if the snapshot refresh mechanism does not have access to old values, then differential update of materialized views based on the insertion and deletion of full tuples does not work and must be changed. This implies that, some additional information has to be maintained in the base relations as well as in the snapshot to carry out snapshot refresh correctly. In addition to the timestamp of latest refresh associated with each snapshot, denoted by τ, as well as a timestamp attribute TS_i incorporated within every relation scheme R_i in the database scheme D, the additional information that we propose be maintained consists of the following:

- Every tuple from the underlying relations has a unique identifier called the *tuple-id*. For any relation scheme $R_i \in D$, its corresponding *tuple-id attribute* will be denoted by TID_i.
- A snapshot definition must include, as members of the set $\mathbf{A}$, the tuple-id attributes from all underlying relations mentioned in its definition. For example, the snapshot definition $E = (\{H, K\}, \{R_1, R_2\}, (H = I))$ will change to $E = (\{H, K, TID_1, TID_2\}, \{R_1, R_2\}, (H = I))$ (assuming that attributes H and K are not keys). The tuple-id attributes stored in the snapshot are not necessarily accessible to users.

We still have to solve the problem of how to determine the set of tuples that have been deleted from a base relation. There are two ways of solving the problem.

1. The system keeps a trace file associated with each base relation containing (tuple-id, timestamp) pairs corresponding to the identifier of a deleted tuple as well as the timestamp indicating the time at which the deletion took place. This idea, in a way, goes against the main purpose of this subsection, namely, old values are not available to the snapshot refresh mechanism. Nevertheless, this idea suggests a limited amount of old values being kept by the system.
2. The system obtains the set of deleted tuples from the information available in the snapshot and the new state of the base relations. This idea is still within the spirit of the subsection.

The following example illustrates the the second approach.

Example Consider two relation schemes $R_1(H, I, TID_1)$, $R_2(J, K, TID_2)$, a snapshot defined by the expression

$$E = (\{H, K, TID_1, TID_2\}, \{R_1, R_2\}, (I = J) \wedge (H > 10)),$$

and their corresponding instances:

r_1:

H	I	TID_1	TS_1
10	20	1	81
11	30	3	82
12	40	4	96
13	50	7	84

r_2:

J	K	TID_2	TS_2
20	500	8	83
30	550	9	85
30	552	10	90
40	570	11	91

$v(E, d)$:

H	K	TID_1	TID_2
11	550	3	9
11	552	3	10
12	570	4	11

Suppose that relation r_1 is changed by deleting the tuples $(10, 20, 1)$ and $(11, 30, 3)$. Both tuples will disappear leaving no trace in relation r_1 about their existence. Using the information available in the current snapshot $v(E, d)$ and the updated instances of the base relations, the system can still obtain the tuple-ids of the deleted tuples. The set of unique identifiers of deleted tuples T_d is obtained by the expression $T_d = \pi_{TID_1}(v(E, d)) - \pi_{TID_1}(r_1) = \{3, 4\} - \{4, 7\} = \{3\}$. Hence, the appropriate tuples from the snapshot can be deleted by performing the operation DELETE $(E, TID_1 = 3)$. □

Given the above, consider a snapshot $v(E, d)$ obtained by the expression

$$v(E, d) = \pi_{\mathbf{A}}(\sigma_C(r_1 \times r_2 \times \cdots \times r_p)).$$

The snapshot refresh operation will involve the following two procedures.

Procedure 7.1: Required to compute the changes that have to be applied to the snapshot to refresh it.

Input:

i) the timestamp of latest refresh of the snapshot $v(E, d)$ denoted by τ,

ii) the sets of tuple-id values $\pi_{TID_i}(v(E, d))$, $1 \le i \le p$, and

iii) the relations r_i, $1 \le i \le p$.

Output: a set of (*attribute-name, value*) pairs T_d and a set of tuples T_i.

1. For each relation r_i, $1 \le i \le p$, partition the set of tuples in r_i into two sets $r_{i_1} = \sigma_{TS_i < \tau}(r_i)$ and $r_{i_2} = r_i - r_{i_2}$. The set r_{i_2} contains the set of tuples inserted into or modified in relation r_i since time τ.

2. For each relation r_{i_2}, $1 \le i_2 \le p$, extract only the tuples that have been modified. Tuples can be identified as inserted, deleted, or modified by means of a tag. Denote this set by mod_{i_2}. From the sets of modified tuples mod_{i_2}, $1 \le i_2 \le p$, form the set T_1 of (*attribute-name, value*) pairs, as follows:

$$T_1 = \bigcup_{i_2=1}^{p} \{(TID_{i_2}, t[TID_{i_2}]) \mid t \in mod_{i_2}\}.$$

3. For each relation r_i, $1 \leq i \leq p$, form the set T_2 of (*attribute-name*, *value*) pairs, as follows:

$$T_2 = \bigcup_{i=1}^{p} \{(TID_i, x) \mid x \in \pi_{TID_i}(v(E,d)) - \pi_{TID_i}(r_i)\}.$$

4. Form the set $T_d = T_1 \cup T_2$.

5. Apply Algorithm 5.1 with input relations r_i replaced by relations r_{i_1} and input relations $\hat{r}_i$ replaced by relations r_{i_2}, $1 \leq i, i_1, i_2 \leq p$. The set of tuples obtained from this evaluation will represent the new tuples T_i to be inserted into the snapshot. □

Procedure 7.2: Required to update the snapshot.

Input: the sets T_d and T_i and the snapshot s.

Output: the refreshed snapshot s'.

1. For each pair $(TID, value) \in T_d$, perform the operation

   DELETE$(s, (TID = value))$.

2. Perform the operation INSERT (s, T_i). □

7.4 Amount of data required to update a materialized view

In this section we are interested in summarizing what is the minimum amount of data required to update a materialized view. This is an important aspect to be considered when a materialized view is stored at a site which is different from the site containing the base relations because of the obvious implications on the communication costs.

In general such an amount of data ranges from no data at all to the complete new state of the snapshot. Table 7.1 describes the amount of data required to update a materialized view depending on the characterization of updates given in this thesis. Notice that when the effect of an update is relevant but not autonomously computable on the view, the system may decide to send either the net changes to the view or the new state of the view whichever is smaller.

One might still envision intermediate situations in which an update to a base relation requires sending a combination of tuples plus autonomously computable expressions to bring the materialized view up to date. The detection of such situations is a subject of further research and will not be discussed in this thesis.

Type of update	Amount of data
Irrelevant	No data
Autonomously computable (intensional or extensional)	Update expression
Relevant and not autonomously computable	Net changes to the view or New state of the view

Table 7.1: Amount of data required to update a materialized view

7.5 Summary

In this chapter we have summarized the way in which the results from Chapters 3 to 6 can be used in the support of consistently up-to-date views. We have also discussed the main issues involved in the support of snapshot refresh through differential re-evaluation. The most important of which is the availability of old and new values to the snapshot refresh mechanism. Several techniques to carry out the snapshot refresh were discussed. The performance evaluation of the various techniques discussed remains a subject of further research.

Chapter 8

Conclusions and Future Research

8.1 Conclusions

The objective of this thesis has been to study the update problem for materialized views. We summarize the main contributions of the thesis in the following subsections.

8.1.1 Irrelevant updates

In Chapter 3 we presented a characterization of irrelevant updates, that is, updates to base relations that cannot cause any effect on their corresponding materialized views. This characterization is important because when an update to a relation is irrelevant to a view, no re-evaluation is required to bring the materialized view up to date. Even when an update represented by an expression is not irrelevant, it may still be the case that when the update is translated to a set of actual tuples, some of those tuples may be irrelevant to the view. Here, the characterization of tuple-wise irrelevant updates is important because the number of tuples considered in the re-evaluation may potentially be reduced.

The detection of irrelevant updates is also important in the context of supporting a database whose internal scheme is structured as a set of derived relations. Updates to conceptual relations which are irrelevant to a derived relation do not need to be performed.

The detection of irrelevant updates has applications also in the area of integrity enforcement. Consider an integrity constraint defined by a *PSJ*-expression. If the database is consistent with respect to such an integrity constraint, then the materialized view obtained by evaluating the expression represented by the complement of the integrity constraint expression, called the *integrity view*, must be empty. Updates to base relations which maintain the integrity of the database must be irrelevant to the integrity view. A similar approach to integrity enforcement is presented by Bernstein and Blaustein [BERNS81].

Buneman and Clemons [BUNEM79] also mentioned the detection of irrelevant updates, which they called *readily ignorable updates*, as an important component in the support of triggers and alerters. In their paper, they presented a syntactic approach to detecting readily ignorable updates which represent a subset of the irrelevant updates discussed here.

The detection of irrelevant updates is implicit in the work by Maier and Ullman [MAIER83b] on updating fragments of relations. In their study, a *fragment* may be a *physical* or *virtual* relation over a single relation scheme, defined by selection and union operators on physical relations or other virtual relations. A fragment f_1 is related to fragment f_2 through a *transfer predicate* β_{12}. β_{12} is a Boolean expression defining the tuples from f_1 that also belong to f_2. When a set of tuples is (say) inserted into f_1, only those tuples satisfying β_{12} will be transferred to f_2. Tuples not satisfying β_{12} are irrelevant to f_2.

Necessary and sufficient conditions for detecting when an update operation is irrelevant to a view (or integrity constraint) have not previously been available for any nontrivial class of updates and views.

8.1.2 Autonomously computable updates

In Chapter 4 we introduced the notion of autonomously computable updates, that is, updates to relations that can be reflected in a materialized view by using only the knowledge provided by the update operation, the expression defining the view, and the current contents of the view.

This characterization leads to savings in communication costs in situations where a materialized view is stored at a remote site, that is, a site that is different from the one housing the base relations. The communication cost savings come from not requiring any data about the base relations to be sent to the site where the view is stored when bringing it up to date. Instead, the materialized view can be updated autonomously.

More importantly, this characterization can be used in the design of views in such a way that updates to some or all the base relations participating in the view definition are autonomously computable with respect to the view. The way to achieve this is by allowing the appropriate attributes to be part of the set of attributes that are visible in the view. As a direct consequence of this, the characterization of autonomously computable updates is an important component towards the support of databases where the internal scheme is structured as a set of derived relations [LARSO85,LARSO86]. By designing the stored derived relations in such a way that users' updates are autonomously computable, the need of collecting data from other derived relations to bring the affected derived relation up to date is avoided. Preliminary results on the detection of autonomously computable updates were presented in [BLAK86b] for unconditionally autonomously computable updates. In this thesis, we have gone further by establishing the conditions for conditionally autonomously computable updates.

Testing the conditions given in the theorems of Chapters 3 and Section 4.1 is efficient in the sense that it does not require retrieval of any data from the database. According to our definitions, if an update is irrelevant or unconditionally autonomously computable, then it is so for *every* instance of the base relations. The fact that an update is not irrelevant does not mean that it will always affect the view. Determining whether it will, requires checking the current instance. The same applies for autonomously computable updates.

8.1.3 Differential re-evaluation of views

In Chapter 5 we presented an approach to differentially re-evaluating materialized views defined by arbitrary *PSJ*-expressions. The method, based on query modification, uses the state of the base relations and the net changes applied to the base relations since the latest update of the view.

If an update to a relation is neither irrelevant nor autonomously computable with respect to a view, then we propose to avoid whenever possible the need for a complete re-evaluation of the expression defining the view to bring it up to date. We propose instead, to update the view by differential re-evaluation.

The differential re-evaluation approach to updating materialized views is better than complete re-evaluation whenever the relations are updated mainly by insertions rather than by deletions.

To improve the performance of the differential re-evaluation approach, we presented in Chapter 6 the formulation of a new multiple query optimization problem. Two alternative solutions to this problem were presented. The first is based on query decomposition of multiple-query graphs as in [CHAKR86]. We improved on the idea of multiple-query decomposition by presenting a better algorithm for decomposing a multiple query graph. The algorithm incorporates some limited knowledge about selection and join selectivities as well as cardinalities of relations to help make better decisions on the application of the various heuristics throughout the decomposition. Although the algorithm was presented in terms of the particular multiple query optimization problem induced by the differential re-evaluation approach, the idea also has application to the general multiple query optimization problem. The second solution proposed uses space search methods and is based on the ideas of Grant and Minker [GRANT82] and Sellis [SELL86a,SELL86b]. To apply a space search method, one has to be able to generate all the promising plans for executing each of the queries to be optimized. In our case, each of the queries in the set has a large number of alternative plans and hence enumerating all possibilities may not be practical. Since in our problem all the queries to be optimized share the same relational algebra expression, applied to a different set of operands, we propose to use the set of individual locally optimal plans as alternative plans for each of the queries in the multiple query optimization. Previous work in multiple query optimization using space search methods do not address the problem of generating the plans to be considered in the optimization.

Finally, in Chapter 7 we summarized the use of the tools developed in Chapters 3 to 6 to the support of consistently and periodically updated views. We presented in that chapter a thorough discussion of the issues involved in the support of snapshot refresh.

8.2 Future Research

There are several directions for future research that can be taken at this point. This thesis presents a starting point to the update transformation problem for databases whose internal scheme is structured as a set of derived relations. So far, the methods presented in the thesis to update materialized views make the assumption that the base relations are available. Therefore, the development of the theory and methods for updating derived relations which are based on extracting the necessary information from other derived relations rather than from the base relations is an important direction for future research.

In Chapter 4, we have presented a characterization of conditionally autonomously computable updates when the update operations are represented by insertions and deletions. Conditionally autonomously computable updates can also be extended to handle modifications. Clearly, the theorems on conditionally autonomously computable deletions directly apply to the theorems required to evaluate the set of tuples in the view that should be modified, as well as to evaluate the set of modified tuples in the view that should remain in the view after modification. Unconditionally autonomously computable modifications require an update to a base relation not to generate new insertions into the view. Conditionally autonomously computable updates may be able to handle, in the same way as with the insert operation, a limited amount of new insertions into the view as long as they can be computed based on the information provided by the update, the view definition, and the current contents of the view. The conditions under which this is possible remain to be derived. Finally, the theorem that establishes the conditions under which the update functions can be computed remains to be derived as well.

We have not imposed any restrictions on valid instances of base relations, for example, constraints specified by functional dependencies or inclusion dependencies. Any combination of attribute values drawn from their respective domains has been assumed to represent a valid tuple, and any set of valid tuples is a valid instance of a base relation. If relation instances are further restricted, then the conditions given in Chapters 3 and 4 are still sufficient, but they may not be necessary.

The results presented in Chapters 3 and 4 on deletions and modifications assume that the set of tuples to be updated is chosen by a select expression on attributes from the updated base relation. Conditions for irrelevant and autonomously computable updates for the case when the set of tuples to be updated is chosen by a *PSJ*-expression remains an important extension to this work.

The results of this thesis apply to materialized views defined by *PSJ*-expressions. Our results readily apply to the maintenance of views defined by the union of several *PSJ*-expressions. The extension of the concepts of irrelevant and autonomously computable updates as well as differential re-evaluation to the maintenance of views containing *aggregate* data is another direction for future work.

When the effect of an update of a base relation on a materialized view is not autonomously computable, there may still be intermediate cases where the update can be decomposed into a portion that is autonomously computable and a portion that requires differential re-evaluation. The detection of these cases is a direction for future research.

The results on irrelevant and unconditionally autonomously computable updates might be better realized when used as components of a tool for designing materialized views that can be updated efficiently. The input to such tool would consist of a view definition and a set of updates to base relations where the conditions defining the tuples to be updated are parameterized. The tool will in turn find the appropriate boundary values for the parameters in the (update) conditions beyond which the update is irrelevant or autonomously computable. This boundary values will then be kept stored along with the view definition so that at run time, the system will only check the values of a given parameter against the already stored values and determine whether the update is irrelevant or autonomously computable, hence, avoiding the execution of the test for satisfiability at run time. The design of this tool remains a problem for future research.

It should be emphasized that the theorems hold for any class of Boolean expressions. However, actual testing of the conditions requires an algorithm for proving the satisfiability of Boolean expressions. Currently, efficient algorithms exist only for a restricted class of expressions, the main restriction being on the atomic conditions allowed. An important open problem is to find efficient algorithms for more general types of atomic conditions. The core of such an algorithm is a procedure for testing whether a set of inequalities/equalities can all be simultaneously satisfied. The complexity of such a procedure depends on the type of expressions (functions) allowed and the domains of the variables. If linear functions with variables ranging over the real numbers (integers) are allowed, the problem is equivalent to finding a feasible solution to a linear programming (integer programming) problem.

Another important problem for future research is the study of better algorithms for the solution of the multiple query optimization problem defined in Chapter 6. The design of performance models under which the problem can be solved accurately as well as a better quantification of the improvements in cost obtained when using heuristics methods seem to be feasible directions for future work.

In Chapter 7 we presented a discussion of the issues involved in the support of snapshot refresh. A performance study of the various techniques discussed in that chapter is another direction for future work.

Bibliography

[ADIBA80] Michel Adiba and Bruce G. Lindsay. "Database Snapshots." In *Proc. of the 6th. International Conference on Very Large Databases*, pages 86–91, Montreal, (1980).

[AHO74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, (1974).

[AHO79] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. "Efficient Optimization of a Class of Relational Expressions." *ACM Transactions on Database Systems,* Vol. 4, No. 4, pages 435–454, (December 1979).

[BERNS81] Philip A. Bernstein and Barbara Blaustein. "A Simplification Algorithm for Integrity Assertions and Concrete Views." In *Proc. of COMPSAC 81,* pages 90–99, Chicago, (November 1981).

[BLAK86a] José A. Blakeley, Per-Åke Larson, Frank Wm. Tompa. "Efficiently Updating Materialized Views." In *Proc. of the ACM SIGMOD International Conference on Management of Data,,* pages 61–71, Washington, D.C., (May 1986).

[BLAK86b] José A. Blakeley, Neil Coburn, and Per-Åke Larson. "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates." Technical Report CS-86-17, Department of Computer Science, University of Waterloo, (May 1986).

[BLASG76] Michael W. Blasgen and Kapali P. Eswaran. "On the Evaluation of Queries in a Relational Data Base System." IBM Research Report RJ 1745, (April 1976).

[BUNEM79] Peter O. Buneman and Eric K. Clemons. "Efficiently Monitoring Relational Databases." *ACM Transactions on Database Systems,* Vol. 4, No. 3, pages 368–382, (September 1979).

[CAMMA81] Stephanie Cammarata. "Deferring Updates in a Relational Data Base System." In *Proceedings of the 7th International Conference on Very Large Data Bases,* pages 286–292, Cannes, (1981).

[CLIFF83] J. Clifford and D. S. Warren. "Formal Semantics of Time in Databases." *ACM Transactions on Database Systems,* Vol. 8, No. 2, pages 214–254, (June 1983).

[CHAKR86] Chakravarthy, Upen S. and Jack Minker, "Multiple Query Processing in Deductive Databases." In *Proc. of the 12th. International Conference on Very Large Data Bases,* pages 384–391, Kyoto, (August 1986).

[CHAMB76] D.D. Chamberlin et al. "SEQUEL2: A unified approach to data definition, manipulation, and control." In *IBM Journal of Research and Development,* Vol. 11, pages 560–575, (November 1976).

[CHRIS81] Stavros Christodoulakis. "Estimating Selectivities in Data Bases." Technical Report CSRG-136, University of Toronto, (December 1981).

[CODD70] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM,* Vol. 13, No. 6, pages 377–387, (June 1970).

[COOK71] Stephen A. Cook. "The Complexity of Theorem-proving Procedures." In *Proc. 3rd Annual ACM Symposium on Theory of Computing,* pages 151–158, (1971).

[DATE86] C. J. Date. *An Introduction to Database Systems,* Fourth Edition. Addison-Wesley, (1986).

[FINKE82] Sheldon Finkelstein. "Common Expression Analysis in Database Applications." In *Proc. of the ACM SIGMOD International Conference on Management of Data,* pages 235–245, Orlando, FL., (June 1982).

[FLOYD62] Robert W. Floyd. "Algorithm 97: Shortest Path." *Communications of the ACM,* Vol. 5, No. 6, page 345. (June 1962).

[GARDA84] G. Gardarin, E. Simon, and L. Verlaine. "Querying Real Time Relational Data Bases." In *IEEE-ICC International Conference,* pages 757–761, Amsterdam, (May 1984).

[GRANT81] John Grant and Jack Minker. "Optimization in Deductive and Conventional Relational Database Systems." In *Advances in Database Theory,* Vol. 1, pages 195–234, (1981).

[GRANT82] John Grant and Jack Minker. "On Optimizing the Evaluation of a Set of Expressions." *International Journal of Computer and Information Sciences.* Vol. 11, No. 3, pages 179–191, (June 1982).

[GRAY81] Jack Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. L. Traiger. "The Recovery Manager of the System R Database Manager." *ACM Computing Surveys,* Vol. 13, No. 2, pages 223–242, (June 1981).

[HAERD83] Theo Haerder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery." *ACM Computing Surveys,* Vol. 15, No. 4, pages 287–317, (December 1983).

[HANSO86] Eric Hanson. "A Performance Analysis of View Materialization Strategies." Memorandum No. UCB/ERL M86/98, University of California, Berkeley, 12 December 1986.

[HORWI85] Susan Horwitz and Tim Teitelbaum. "Relations and Attributes: A Symbiotic Basis for Editing Environments." In *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments,* pages 93–106, (July 1985).

[IBARA84] Toshihide Ibaraki and Tiko Kameda. "On the Optimal Nesting Order for Computing N-Relational Joins." *ACM Transactions on Database Systems,* Vol. 9, No. 3, pages 482–502, (September 1984).

[JARKE84] Matthias Jarke and Jürgen Koch. "Query Optimization in Database Systems." *ACM Computing Surveys,* Vol. 16, No. 2, pages 111–152, (June 1984).

[JARKE85] Matthias Jarke. "Common Subexpression Isolation in Multiple Query Optimization." In *Query Processing in Database Systems,* W. Kim, D. Reiner, D. Batory (eds.), Springer-Verlag, pages 191–205, (1985).

[KELLE86] Arthur M. Keller. "The Role of Semantics in Translating View Updates." *Computer,* Vol. 19, No. 1, pages 63–73, (January 1986).

[KLUG80] Anthony Klug. "On Inequality Tableaux." *CS Technical Report 403,* University of Wisconsin, Madison, WI, (November 1980).

[KOENI81] Shaye Koenig and Robert Paige. "A Transformational Framework for the Automatic Control of Derived Data." In *Proc. of the 7th. International Conference on Very Large Data Bases,* pages 306–318, Cannes, (1981).

[KORTH86] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill, (1986).

[LARSO85] Per-Åke Larson and H. Z. Yang. "Computing Queries from Derived Relations." In *Proc. of the 11th International Conference on Very Large Data Bases,* pages 259–269, Stockholm, (1985).

[LARSO86] Per-Åke Larson and H. Z. Yang. "Computing Queries from Derived Relations." *Full manuscript submitted for publication,* (1986).

[LINDS86] Bruce Lindsay, Laura Hass, C. Mohan, Hamid Pirahesh, and Paul Wilms. "A Snapshot Differential Refresh Algorithm." In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* pages 53–60, Washington, D.C., (1986).

[MAIER83a] David Maier. *The Theory of Relational Databases.* Computer Science Press, (1983).

[MAIER83b] David Maier and Jeffrey D. Ullman. "Fragments of Relations." In *SIGMOD' 83 Proceedings of Annual Meeting,* Sigmod Record, Vol. 13, No. 4, pages 15–22, San Jose, CA., (1983).

[MEDEI86] Claudia Bauzer Medeiros and Frank Wm. Tompa. "Understanding the Implications of View Update Policies." *Algorithmica,* Vol. 1, No. 1, pages 337–360, (1986).

[MYLOP75] J. Mylopoulos, S. Schuster, and D. Tsichritzis. "A Multi-level Relational System." In *Proc. 1975 National Computer Conference,* AFIPS Press, Arlington, VA., pages 403–408.

[ROSEN80] Daniel J. Rosenkrantz and Harry B. Hunt III. "Processing Conjunctive Predicates and Queries." In *Proc. of the 6th International Conference on Very Large Data Bases,* pages 64–72, Montreal, (1980).

[ROUSS82] Nicholas Roussopoulos. "View Indexing in Relational Databases." *ACM Transactions on Database Systems,* Vol. 17, No. 2, pages 258–290, (June 1982).

[ROUSS86] Nicholas Roussopoulos and Hyunchul Kang. "Preliminary Design of ADMS±: A Workstation-Mainframe Integrated Architecture for Database Management Systems." In *Proceedings of the 12th International Conference on Very Large Data Bases,* pages 355–364, Kyoto, (August 1986).

[SCHMI75] Hans A. Schmid and Philip A. Bernstein. "A Multi-level Architecture for Relational Data Base Systems." In *Proceedings of the International Conference on Very Large Data Bases,* pages 202–226, Framingham, Massachusetts, (September 1975).

[SCHOL81] Mario Schkolnick and Paul Sorenson. *The Effects of Denormalization on Database Performance.* RJ 3082, IBM, (April 1981).

[SELIN79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. "Access Path Selection in a Relational Database Management System." In "Proc. of the ACM SIGMOD 1979 International Conference on Management of Data," pages 23–34, (1979).

[SELL86a] Timos K. Sellis. "Global Query Optimization." In *Proc. of the ACM SIGMOD International Conference on Management of Data,* pages 191–205, Washington, D.C., (May 1986).

[SELL86b] Timos K. Sellis. *Optimization of Extended Relational Database Systems.* Ph.D. Thesis, University of California, Berkeley, (July 1986).

[SEVER76] Dennis G. Severance and Guy M. Lohman. "Differential Files: Their Application to the Maintenance of Large Databases." *ACM Transactions on Database Systems,* Vol. 1, No. 3, pages 256–267, (September 1976).

[SHMUE84] Oded Shmueli and Alon Itai. "Maintenance of Views." *Sigmod Record,* Vol. 14, No. 2, pages 240–255, (1984).

[SNODG84] Richard Snodgrass. "A Temporal Query Language TQUEL." In *Proceedings of the 3rd. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,* pages 204–212, Waterloo, Ont., (April 1984).

[STONE76] M. Stonebraker, E. Wong, P. Kreps, and G.D. Held. "The Design and Implementation of INGRES." *ACM Transactions on Database Systems,* Vol. 1, No. 3, pages 189–222, (September 1976).

[TSICH77] Dionysios C. Tsichritzis and Frederick H. Lochovsky. *Data Base Management Systems.* Academic Press, (1977).

[TSICH82] Dionysios C. Tsichritzis and Frederick H. Lochovsky. *Data Models.* Prentice-Hall, (1982).

[ULLMA82] Jeffrey D. Ullman. *Principles of Database Systems,* Computer Science Press, 2nd. edition, (1982).

[WONG76] Eugene Wong and Karel Youssefi. "Decomposition - A Strategy for Query Processing." *ACM Transactions on Database Systems,* Vol. 1, No. 3, pages 223–241, (September 1976).

[YAO78] S. Bing Yao and David DeJong. "Evaluation of Database Access Paths." In *Proc. of the ACM SIGMOD International Conference on Management of Data,* pages 66–77, (1978).

[YAO79] S. Bing Yao. "Optimization of Query Evaluation Algorithms." *ACM Transactions on Database Systems,* Vol. 4, No. 2, pages 133–155, (June 1979).

[YOUSS79] Karel Youssefi and Eugene Wong. "Query Processing in a Relational Database Management System." In *Proc. of the 5th. International Conference on Very Large Data Bases,* pages 409–417, Rio de Janeiro, (October 1979).