Optimizing Incremental View Maintenance Expressions in Relational Databases

by

Dimitra Vista

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Optimizing Incremental View Maintenance Expressions in Relational Databases

Dimitra Vista

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

1996

In the last few years, there has been significant interest in the design of incremental methods to improve the performance of view maintenance. Despite that, very little analysis or experimentation supports the predominant view that incremental methods are more efficient than their non-incremental counterparts. We argue that the performance of incremental view maintenance depends on system aspects of the database, such as the availability of indices, the sizes of the relations involved, and the sizes of the database updates. We also argue that the database query optimizer is a reasonable component of the database system to decide, at the time of view maintenance, whether a view is to be maintained incrementally or not, because the query optimizer has knowledge of, and access to, all of the parameters that may affect this choice.

We have built the RHODES database query optimizer that supports change propagation and view maintenance. In addition to traditional optimizations, RHODES is also responsible for the generation of the queries to be executed in order to support view maintenance. As there may be many different ways to maintain a view incrementally, the choice of which one to use may affect the performance of incremental view maintenance. Moreover, different maintenance queries are amenable to different optimizations. In this

thesis, we present a repertoire of maintenance-specific optimizations, especially in the presence of key constraints and foreign key references. The underlying data model we use is relational algebra with multiset semantics. Experimental validation of the above claims has been conducted using the TPC-D benchmark database on the DB2 Parallel Edition.

Acknowledgements

Thanks!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 View Maintenance

Traditionally, a *database view* is a query on a database that computes a relation whose value is not stored explicitly in the database, but to the users of the database it appears as if it were. Database views are useful for a number of reasons. They can be used to provide conceptual subsets of the database to different users. They can be used as mechanisms to enforce security by allowing parts of the data to be seen only by users with the appropriate access privileges. They provide a convenient shorthand notation to facilitate query specification. They can be used to replicate data, possibly in geographically remote data sources. Finally, they can be used in query optimization to speed-up query evaluation.

There are two different ways to implement views. The traditional, and still most popular approach, is the *query modification* approach [Sto75]. The definition of each view is stored in the dictionary of the system. Queries referring to the view are answered by substituting the view definition into the body of the queries. Since only the definition is kept, query evaluation of a query involving a view results in re-evaluating the view. The advantage of this method is that it requires practically no extra disk storage or maintenance. However, it might have poor performance if the queries to the views are more frequent than the updates to the database, because frequently accessed views result in repetitive view construction.

The second method to implement views is the *view materialization* approach where the view is explicitly maintained as a stored relation [GM95]. This method requires more storage than the query modification approach but its performance might be significantly better, especially if updates are less frequent than queries referring to the views. A database system should provide the option of materializing views. The choice of which views to materialize should be guided by the actual or anticipated query load so that frequently occurring queries can be evaluated quickly.

The view materialization approach, thus, has the potential of significantly improving the time to access a view. However, it does have some effect on the overall performance of the database system. Next, we describe the major problem associated with view maintenance and its solutions.

1.2 Incremental View Maintenance

One problem with the view materialization approach is that every time a base relation changes, the views that depend on it may need to be re-computed. One approach to this problem is to re-compute all related views. This solution may be acceptable for relatively static databases, but may be prohibitively expensive when updates are frequent. When the views are frequently updated and expensive to compute, the cost of re-computation may not be affordable. The alternative to re-computation is to identify which part of the old materialized view is affected by the database update and to re-compute only the affected part. An algorithm that carries out such a computation is called an *incremental view maintenance* or *incremental query evaluation*[1] algorithm.

The idea of incremental view maintenance can be summarized as follows. Suppose V is the query expression corresponding to a view definition and $V[D]$ is the materialized value of V consistent with database D, i.e., $V[D]$ is the value of V on database D. Suppose that the database changes from D to D^v by some update $\delta(D)$. In order to find the new value of V, we evaluate two query expressions, $\delta^-(V)$ and $\delta^+(V)$, on the database D and the database update $\delta(D)$. These query expressions define the change

[1]Although the two terms mean slightly different things, we choose to use them interchangeably in this thesis, because each view is specified by some query expression.

on $V[D]$, i.e., $\delta^+(V)[D, \delta(D)]$ are the tuples to be inserted into $V[D]$ and $\delta^-(V)[D, \delta(D)]$ are the tuples to be deleted from $V[D]$, in order to make V consistent with the updated database D^v. In other words[2],

$$V[D^v] \equiv (V[D] - \delta^-(V)[D, \delta(D)]) \cup \delta^+(V)[D, \delta(D)]$$

We call $\delta^-(V)$ and $\delta^+(V)$ the *change propagation expressions* of V. Their values under database D, along with the database updates, represent the incremental changes to V when D is updated to the next database state.

There are two main issues related to incremental view maintenance:

1. The choice of which change propagation expressions to use: How do we choose the two query expressions $\delta^-(V)$ and $\delta^+(V)$? Do these expressions depend only on the view definition? Do they depend on the database update? Do they depend on the old database? Do they depend on the old value of the view?

2. The choice of the alternatives to "bring V up-to-date with the database": If our objective is to find the new value of the view V under the updated database, should we use the incremental method and compute $(V[D] - \delta^-(V)[D, \delta(D)]) \cup \delta^+(V)[D, \delta(D)]$, or should we use the re-evaluation method and compute $V[D^v]$ from scratch?

Recently, many approaches have been proposed to specify *how* the change propagation expressions of a given view are formed in terms of the query expression corresponding to the view, the old value of the database, the update, and, perhaps, the old value of the view. Chapter 2 discusses many of them.

As we see in Chapter 2, a significant amount of research addresses the first issue of incremental view maintenance. However, the second issue has not yet received much attention. Very little analysis or experimentation supports the predominant view that incremental methods are more efficient than their non-incremental counterparts. In fact, the consensus seems to be that, for small updates, evaluating the change propagation

[2]The operands of $-$ and $\cup$, here, are multisets, i.e. sets with duplicates. The data model assumed in this thesis is relational algebra with multiset semantics.

expressions of a view is more beneficial than re-evaluating the view. When one relation is completely deleted from the database, it is almost always better to re-evaluate any join involving that relation. On the other hand, when the updates to the database are small, compared to the database itself, the *"principle of inertia"* [GM95], that small changes propagate small changes, seems to favor incremental evaluation of the join. However, we found that the cost of propagating small changes could be about the same as (or more than) the cost of evaluating the view again. Also, when the updates to the database are neither very small nor very big[3], compared to the database, it is not at all clear which of the two alternatives is likely to provide a better solution. Consequently, the choice of whether to perform incremental view maintenance or not cannot be made a priori without first examining all factors affecting this choice.

The performance of incremental view maintenance depends on system aspects of the database, such as availability of indices, sizes of the database relations involved, sizes of the database updates, and so on. There are two justifications for the above statement. The statement is true because incremental view maintenance requires evaluation of the change propagation expressions, which are queries whose performance in general, like that of many queries, depends on the physical design of the database system. However, the statement is also true because the *choice* of which change propagation expressions to use (and there may be many for the same view) depends both on the system aspects of the database and on the database update. In this thesis we claim that we should not commit to incremental view maintenance a priori, but, rather, we should let the database query optimizer decide, at the time of view maintenance, if incremental view maintenance is better than re-evaluation. We also claim that we should let the query optimizer decide which change propagation expressions to use as well as how to execute these change propagation expressions best.

We see the incremental view maintenance problem as an optimization problem. The objective of the optimization is is to minimize the logical I/O necessary to perform incremental view maintenance. The decision involves which maintenance strategy to choose, among incremental maintenance and re-computation, for each materialized view,

[3]Note that *small* and *big* are rather loosely used here.

and each possible update. The knowledge available in this optimization problem is the schema of the database, the definition of views, the update, the cardinalities of the relations in the database and their updates, the distribution of data values, and so on.

Most proposals for incremental view maintenance assume that for each view V, the change propagation expressions $\delta^-(V)$ and $\delta^+(V)$ are generated by a special software component of the DBMS, most likely at view compile time but possibly at view maintenance time. Apart from the fact that the above approach requires a special software component to be developed just for the purpose of generating the change propagation expressions, there are other disadvantages with it. Before presenting them, however, let us see an example that demonstrates some of them.

1.3 Example

Let $V = A \bowtie B \bowtie C$, and suppose that each of the A, B and C relations lose a number of tuples specified by δ_A^-, δ_B^- and δ_C^-, respectively[4]. As we see in the next chapters, there exists an algebraic equation that defines the deletions from a join from the inputs to the join and their deletions. Let δ_{AB}^- be the deletions from the join $A \bowtie B$ (as if this join were materialized); δ_{BC}^- the deletions from the join $B \bowtie C$ (as if this join were materialized); and, δ_{ABC}^- the deletions from V. There are a number of different ways to compute the deletions from V.

1. One alternative is to find the deletions from $A \bowtie B$ and propagate these to V:

$$\begin{aligned} \delta_{ABC}^- &= \delta_{AB}^- \bowtie C \;\cup\; A \bowtie B \bowtie \delta_C^- \;-\; \delta_{AB}^- \bowtie \delta_C^-, \quad \text{where} \\ \delta_{AB}^- &= \delta_A^- \bowtie B \;\cup\; A \bowtie \delta_B^- \;-\; \delta_A^- \bowtie \delta_B^-. \end{aligned}$$

   The justification for the correctness of these equations is as follows[5]. Consider the deletions from the join $A \bowtie B$. Tuples deleted from A that join with tuples in B, generate deletions from $A \bowtie B$. Also, tuples deleted from B that join with tuples in A, generate deletions to $A \bowtie B$. However, if a deleted tuple from A matches

[4]For simplicity of the presentation, we ignore the arguments of the join and we assume that all join orderings are possible.

[5]Note that these equations are *not* correct, if A, B and C also have tuples inserted into them at the time of view maintenance.

a deleted tuple in B, each tuple to be deleted from $A \bowtie B$ is generated twice: once because of $\delta_A^- \bowtie B$ and once because of $A \bowtie \delta_B^-$. Since we support duplicate semantics, each tuple must be deleted once and, therefore, we must subtract $\delta_A^- \bowtie \delta_B^-$.

2. Another alternative is to find the deletions from $B \bowtie C$ and propagate these to V:

$$\begin{aligned} \delta_{ABC}^- &= A \bowtie \delta_{BC}^- \;\cup\; \delta_A^- \bowtie B \bowtie C \;-\; \delta_A^- \bowtie \delta_{BC}^-, \quad \text{where} \\ \delta_{BC}^- &= \delta_B^- \bowtie C \;\cup\; B \bowtie \delta_C^- \;-\; \delta_B^- \bowtie \delta_C^-. \end{aligned}$$

Note that a number of other alternatives are also possible. Our objective with this example is not to list them all. The point that we are trying to make is that there may be more than one alternative equivalent change propagation expressions to compute the deletions from a view V. The *choice* of the alternative may affect both the performance of incremental view maintenance and the optimizations that are possible in the optimizer.

Even if we knew that V is to be maintained incrementally, it is not clear which of the two alternatives listed here offers a better way to maintain V. If a database optimizer is given one of the alternatives to optimize, most likely it will not be able to transform it into the other alternative and may, thus, miss a better execution plan.

Also, although seemingly very alike, the two alternatives are amenable to different optimizations. Suppose, for example, that there is a foreign key reference from A to B[6]. Then, tuples that are deleted from B can only join with tuples deleted from A because, otherwise, the foreign key reference would not be satisfied after the database update. Thus, we can use the following equivalence

$$A \bowtie \delta_B^- = \delta_A^- \bowtie \delta_B^-$$

and, we can rewrite the first alternative as

$$\begin{aligned} \delta_{ABC}^- &= \delta_{AB}^- \bowtie C \;\cup\; A \bowtie B \bowtie \delta_C^- \;-\; \delta_{AB}^- \bowtie \delta_C^-, \quad \text{where} \\ \delta_{AB}^- &= \delta_A^- \bowtie B \end{aligned}$$

while the second alternative cannot be rewritten.

[6]The join argument must be in conjunctive form and it must include a conjunct equating the attribute of A with the foreign key reference with the key attribute of B, for all this to make sense.

Thus, by adopting the fist alternative, we were able to reduce access to the database relations and the total number of joins, and, therefore, increase the likelihood that the performance of the incremental approach be better than re-evaluation.

It is not clear that a database optimizer could easily have incorporated that kind of optimization if it was given the second alternative rather than the first.

1.4 Problems

After the example, we are ready to list some of the disadvantages of using a special component in the DBMS to generate the change propagation expressions for each view.

1. Using the special software component, we commit to incremental view maintenance, even in cases where re-evaluating the view is likely to be more efficient (such as, for example, when deleting entirely a database relation). There is no choice of performing or not performing incremental view maintenance, even though the size of the database update clearly should affect this choice.

2. Even if the optimization of the change propagation expressions occurs when the maintenance is performed, the choices of the query optimizer to generate an efficient execution plan may be restricted. The reason for this is that the generation of change propagation expressions is independent of the database and its updates. The optimizer may not be able to "transform" the change propagation expressions that it is given into equivalent forms that may be more efficient to evaluate (see the example above).

3. Since the generation of change propagation expression occurs independently of the optimization process, the generation phase may produce change propagation expressions for which certain optimizations, perhaps possible in other (equivalent) change propagation expressions, are rather difficult to incorporate by an optimizer (see the example above).

1.5 Our Approach

We have built the RHODES database query optimizer that addresses all of the above problems. In particular:

1. For each view, RHODES determines whether the view is a candidate to be maintained incrementally or not.

2. For each view that should be maintained maintained incrementally, RHODES finds the best change propagation expressions that define the changes to the view.

3. For each change propagation expression (like any other query), RHODES finds a reasonable way to evaluate the expression, by using traditional optimization techniques, such as relation indices, join orderings, sort orders, query transformations, and so on.

4. RHODES incorporates optimizations specific to incremental view maintenance, and optimizations specific to change propagation expressions, especially in the presence of key constraints and foreign key references.

5. For any query that contains a subquery corresponding to a view, RHODES examines the alternative to use the view in the place of the subquery. Using views to improve the performance of ordinary queries has been recognized recently as a potential for query optimization [LMSS95, FRV96, LMSS95], but RHODES is the only optimizer, to the best of our knowledge, that incorporates this optimization.

The query language of RHODES is relational algebra extended to be consistent with SQL semantics. Most research in incremental view maintenance assumes that relations are sets and do not have duplicates [BCL89, BLT86, BC79, CW90, CW91, DT92, Küc91, SI84, UO92, WDSY91]. However, most database systems use *multisets* (sets with duplicates), because many database applications require aggregation and duplicates are necessary for correctly computing aggregate functions. Another reason for the use of duplicate semantics is that duplicate elimination is a rather expensive operation and, for efficiency, is not enforced, unless specifically requested by the user. There is some work related to multisets for the Datalog model [GKM92, GMS93, Ana96, AV95] and the relational model [GL95]. In this thesis, we use, and build on, the change propagation expressions proposed by Griffin and Libkin [GL95] for a multiset algebra.

1.6 Thesis Outline

Here is the outline of the rest of this thesis:

Chapter 2: We present a detailed discussion of previous work on incremental view maintenance and related research. We concentrate on incremental algorithms for view maintenance as well as applications of incremental view maintenance techniques.

Chapter 3: We present our mathematical framework for incremental computation. We present the change propagation expressions for relational algebra with duplicate semantics and prove their correctness.

Chapter 4: We present a repertoire of optimizations specific to change propagation expressions and incremental view maintenance, especially in the presence of key constraints and foreign key references.

Chapter 5: We present an overview of the database query optimizer RHODES and its extensions to support incremental view maintenance. We also discuss the visual browser that accompanies RHODES.

Chapter 6: We present our experimental validation of the claims of this dissertation, some obtained by estimating change propagation queries using RHODES, and others by actually running change propagation queries on the DB2 PE (Parallel Edition) for a number of updates and queries from the TPC-D benchmark.

Chapter 7: We conclude the thesis by presenting the list of its contributions and a discussion of open problems.

Chapter 2

A Survey on View Maintenance: Applications and Techniques

This chapter surveys work related to incremental view maintenance with an emphasis on the development and application of techniques for view implementation and maintenance.

2.1 Strategies for View Maintenance

When updates occur to a database, there are two distinct execution strategies to update all affected materialized views, whether incrementally or not:

- *Immediate update*: All affected views are immediately updated. This strategy creates an overhead for the processing of the updates but minimizes the query response time for queries accessing the view. This is the strategy assumed in this thesis.

- *Deferred update*: All affected views stay outdated until an access to them is made. This strategy avoids the system overhead associated with immediate update propagation, but slows down query evaluation for queries accessing outdated views.

Both immediate and deferred maintenance guarantee that the view is consistent with the underlying database at the time the view is accessed. In contrast, *periodic updates* where all affected views are periodically updated is used to perform updates during periods of low system use or at pre-specified times. Such views are sometimes called *snapshots* and do not guarantee the consistency with the underlying database. Most work in view maintenance assumes the immediate update strategy. Deferred updates have been studied only recently [CG96, CM96, CGL+96, ZH96].

A number of more refined strategies for view maintenance are available in active database management systems [WC96], where so called active rules are used for the maintenance of views. Active systems support view maintenance quite naturally. In fact, some active systems support incremental view maintenance. In contrast, passive database management systems require significant changes in their software to support either materialized views or their incremental maintenance.

Active rules have the general form

$$\texttt{if } E\texttt{, check } C \texttt{ and execute } A$$

where E is an *event* that causes the rule to be *triggered*, C is a *condition* that is checked when the rule is triggered, and A is an *action* that is performed when the condition of the triggered rule is true. The events of active rules are database access events (such as updates or retrievals), transaction events (such as transaction commit), time events (such as midnight), or combinations of the above [Has95, Has96, GJS92]. The conditions of active rules are either predicates (whose value is true or false) or query expressions (whose value is an empty or a non-empty relation) specified in the system's query language [BW93, HBH+95]. Finally, the actions of active rules are sequences of database manipulation commands (such as insertions and deletions). When active rules are used for (incremental) view maintenance the events of the corresponding active rules are the insertions and deletions to the database relations; the conditions are used to compute any updates to the view; and, the actions are statements to update the view.

The notion of *coupling modes* between the triggering event (which usually occurs in a transaction) and the execution of the associated action (which may or may not occur in the same transaction as the triggering event) yields a number of alternative strategies for view maintenance. The *immediate coupling* mode signifies that maintenance is performed, within the same transaction, as soon as the triggering event occurs. The *deferred coupling* mode signifies that the maintenance is performed at the commit point of the transaction with the triggering event. Finally, the *decoupled* mode signifies that the maintenance is done independently of the triggering transaction. Within the decoupled mode, only the *dependent decoupled* mode is relevant, which spawns a different transaction for the maintenance only if the triggering transaction commits. The *independent decoupled* spawns another transaction independently of whether the triggering transaction commits.

2.2 View Implementation

The are a number of different ways to store views:

1. Using Relations: This is the most popular approach and the one used in this thesis. The view relation is stored in a database like any other relation. Index structures may be built to facilitate fast access to the view's data. Accessing the entire view results in scanning the view relation.

2. Using alternative data structures, for example:

   - Using View-Caches: A *view-cache* is an index-like structure that holds pointers to tuples of the database relations (or pointers to tuples in other view-caches) that are used to derive the view data [Rou91, RCK95]. Accessing the view results in reading the index-like structure (which might be small enough to be in main-memory) and then retrieving all related tuples from the underlying database to compute the actual view data. View-caches have been implemented and validated in the ADMS project [Rou91, RCK95].

   - Using Discrimination Networks: A *discrimination network* is a persistent data structure in the form of a tree or a directed acyclic graph. Each node in the network has a persistent relation associated with it. The immediate children of the (artificial) root correspond to the database relations while the leaf nodes correspond to the view relations. Intermediate nodes correspond to intermediate relations (usually selected portions of base relations) which are materialized (replicated). A discrimination network called *Gator* is implemented and used in the ARIEL active database system [HBH$^+$95, Han96].

The view maintenance problem takes a slightly different flavor depending on how views are stored.

2.3 Deltas

Systems that support incremental maintenance need a structure that holds *deltas*: the tuples to be inserted, deleted or modified in one *database transition*. A database transition is a transformation of the database from one state to another through a sequence of data manipulation commands. A transaction for instance may be used to define a database transition. A delta is defined as a data structure that holds (the net effect of) the insertions and deletions to the database during one transition. Deltas are available through system-defined *transition tables* [Wid96, SK96], or *update logs*. The Heraclitus Project [GHJ92, GHJ+93, ZHKF95] elevates deltas into first class citizens of the database management system and, in particular, of its query language. Deltas in Heraclitus are available as system relations.

2.4 Applications

Apart from view maintenance, incremental view maintenance algorithms can be used in a number of other application areas both in databases and elsewhere. This section discusses a few of these applications. The presented list of applications is not intended to be complete, but, rather, indicative of the use of incremental algorithms for both database management systems and application programs that make use of database systems.

Integrity Constraints

Certain types of integrity constraints, including referential integrity and uniqueness of key constraints, can be expressed as views over the database state [CW90, Sto75]. If such a view is non-empty in a particular state, then the constraint is *violated* and the state is *inconsistent*. Symmetrically, a constraint may be violated when its view becomes empty. The former constraints are negative constraints (nothing can be in the view at any time) while the latter are positive constraints (something must be in the view at all times). If an update operation has no effect on the view associated with an integrity constraint, then the update does not result in a database instance violating the constraint. Incremental evaluation can be used to detect violations of integrity [CW90, BC79, BW93, Ple93]. Instead of evaluating the new value of the view every time we check for the integrity of the database, we can use the fact that the view is empty (or non-empty) before an update, and only determine whether the update induces any change to this view. If it does, corrective actions, such as, for example, rolling back and undoing the update operations, are necessary to restore the integrity of the database.

Alerters

Alerters [BC79] are programs which monitor a database and report to some user when a specified condition occurs. An example of an alerter re-orders items for an inventory control system when these items are in stock at a quantity below a pre-specified threshold. Alerters, like integrity constraints, can be associated with views. The triggering events are insertions and/or deletions from the view predicate associated with the alerter. Again, incremental evaluation may be a reasonable alternative to evaluate the view associated with the alerter.

Active Rules

The concept of a trigger is also central to active databases [WC96, GJS92, BA93, BM91, SPAM91] which monitor *happenings of events* for reasons such as authorization checking, general integrity maintenance, alerting, real-time application support, workflow management support and so on. Active rules are a very powerful modeling mechanism and, as discussed above, active rules can be used to specify how view maintenance relates to basic database manipulations. In general, one of the challenges that active rules pose is the efficient evaluation of rule conditions for triggered rules. Conditions are query expressions in the query language of the system and their evaluation is considered to be the "bottle-neck" in the execution of active rules. Note that activation of a rule may trigger other rules which may, in turn, trigger the initial rule again. One can use the fact that the rule did or did not trigger the last time the rule was considered and incrementally determine if it needs to be re-triggered [BW93].

Real-Time Applications

In real-time applications, such as in communication network management [WDSY91, Has96], the database may change independently of query processing. For example, during a network analysis process, certain connectivity data may change asynchronously. New data may arrive after the query or analysis process has already begun, which may invalidate the computed results. An alternative to starting the analysis process again could be to log all new data that has arrived after the process has started and incrementally correct the result based on this information [WDSY91].

Data Warehousing

A *data warehouse* is a repository of replicated or integrated information from a number of possibly heterogeneous and geographically distributed information sources [HGMW+95, ZHKF95, ZGMHW95]. Data warehousing is being recognized as one of the promising new database applications, towards which current research will likely be directed in the next few years [SSU95]. A data warehouse can be thought of as a view over the individual information sources. Special software components in the architecture of a data warehouse, called *mediators* [ZHKF95, Wie92] or *integrators* [HGMW+95], are responsible for updating the warehouse view in response to updates to the individual data sources. An interesting discussion on the architecture and formalization of mediators is presented by Zhou and Hull in [ZH96].

Other

Incremental algorithms have also been studied in a number of other areas. These include: reasoning about changes [Küc91, UO92]; distributed computing [AISN90, Ita91], programming languages [SH91, TR81]; maintenance of graph properties [BKV90, CC82]; and maintenance of other data structures [CH91, Jag90, KP81, LHM+86]. A study of methods for incremental database query computation is provided elsewhere [Vis94].

2.5 Algorithms for Incremental View Maintenance

In this thesis we concentrate solely on the problem of propagating updates from the database to the view. The reverse problem of translating updates submitted to a view into database updates is a much more difficult problem, and is not included in the scope of this thesis. For related work in this area, the reader is referred to [DB82, FSDS79, Kel85, BS81].

This section reviews proposals for incremental query evaluation for both relational and deductive databases [ABW88, BR86, Ull88, GM95].

2.5.1 Non-Recursive Views

Finite Differencing

Koenig and Paige [KP81] support *derived data* in the context of a functional/binary association data model. In their framework, the derived data are base relation attributes or aggregate functions on them. The average salary of employees is such an example. Koening and Paige's approach to the automatic maintenance of derived data is based on the transformational technique of *finite differencing*. Every transaction T is replaced by a semantically equivalent transaction T', which, in addition to what T does, also adjusts the views appropriately. Since T' varies according to the view definitions, it is called the *differential* of the view definition with respect to T. Transaction T' is obtained from transaction T by inserting into T certain lines of code that preserve the view definition. The fundamental unit of such code is the *derivative* and is defined for single derived data and single tuple updates. Hence, the algorithm for computing the differential depends on the availability of derivatives for various derived data/primitive update pairs.

This is the first proposal that addresses incremental view maintenance and many other methods are influenced by it, only in the literature they are referred to as *program transformation* approaches: given a view definition, and perhaps an update, a program is derived[1], whose evaluation maintains the view.

[1] A *program* is a collection of deductive rules or SQL statements.

Counting

Blakeley et al. [BLT86] propose an algorithm for updating views defined with *select-project-join* (SPJ) expressions, an important subset of SQL. An additional attribute, called the *multiplicity counter*, is attached to each tuple to handle deletions correctly. For base relations, it need not be explicitly stored since its value for every tuple is always one. For view tuples, the multiplicity counter records the number of operand tuples that contribute to it. If a tuple is inserted into a relation, its multiplicity counter is incremented by one. If the tuple is deleted, its multiplicity counter is decremented by one. The tuple is deleted only when its counter becomes zero. Basic set-manipulation operations such as *select* and *project* are redefined to consider these counters. Given a set of insertions into and deletions from base relations, the algorithm derives *SPJ* expressions whose evaluation determines the tuples to be inserted into or deleted from the view. A transaction to update the view is also generated.

The counting algorithm of Gupta et al. [GKM92, GMS93] tracks the number of alternative derivations, called *count*, of each tuple in the materialized view, in the same way as the algorithm of Blakeley et al. [BLT86]. Given a program T defining a set of views, the counting algorithm derives a program T_Δ at view compile time. The incremental program T_Δ uses the changes made to base relations and the old values of the base and view relations to produce as output the set of changes that need to be made to the view relations. The count value for each tuple is stored in the materialized view. The changes to base relations are specified by *delta* predicates, where inserted tuples are represented with positive counts and deleted tuples are represented with negative counts. The incremental view maintenance algorithm works for both set and duplicate semantics and for views with safe stratified negation and stratified aggregation[2]. On non-recursive views, counts can be computed at little or no cost above the cost of evaluating the view. The authors recommend the use of this algorithm for non-recursive views only (because for recursive views their method may not terminate). Another property of this method is that it requires all intermediate results computed during the view computation to also

[2]For definitions of stratified negation and aggregation, refer to Ullman [Ull88] and Mumick et al. [MPR90].

be materialized.

Shmueli and Itai [SI84] also use multiplicity counters for the number of different derivations of a tuple but they use specialized data structures to support them. For instance, each tuple in the database contains pointers to all tuples derived from it. Nicolas and Yazdanian [NY83] use counts to reflect some types of derivations (but not all derivations).

Production Rules

Ceri and Widom [CW90, CW91] study views from a larger class of SQL. They define views as general SQL queries with only a few limitations (such as, only one level of nesting in subqueries). The user is required to specify the view along with key information about the base relations. Syntactic analysis on the view definition based on key information determines whether the view may contain duplicates and whether efficient maintenance is possible. If the view does not contain the keys of all relations used to defined it, then it may contain duplicates, and this algorithm does not work. Otherwise, the method of Ceri and Widom automatically derives a set of *production rules* (essentially active rules) for it. This method has been implemented in the Starburst system [HCL+90, Wid96].

Algebraic Methods

Griffin and Libkin provide an algebraic approach to view maintenance [GL95]. They algebraically define the notion of delta propagation and provide two sets of delta propagation expressions: one for deletions and one for insertions. Furthermore, their results are presented for an algebra with multiset semantics. In fact, this is the method adapted in this thesis (see Chapter 3 for more information on this work). The work of Griffin and Libkin was inspired by the earlier algebraic treatment of the problem for the traditional relational algebra by Qian and Wiederhold [QW91], which was later corrected by Griffin, Libkin, and Trickey [GLT]. A similar algebraic approach for views with aggregation has been provided by Quass [Qua96].

2.5.2 Recursive Views

Rederivation Methods

Gupta et al. [GMS93] suggest the use of their counting algorithm for non-recursive views only. For recursive programs, they propose the "*Delete and Rederive*" algorithm. Instead of using counts to handle deletions, this method first deletes from the view an overestimate of the tuples to be deleted and then re-derives those with alternative derivations. Inserted tuples are handled by deriving all new tuples as well as tuples that obtain additional derivations. All these steps are carried out through the execution of automatically generated delta rules. Similar algorithms for stratified Datalog programs are proposed by Küchenhoff [Küc91], and Harrison and Dietrich [HD92].

Maintenance in Languages with Less Expressive Power

Dong and Topor [DT92] study *regular chain* Datalog programs, which are programs with some restricted form of linear recursion. Their algorithm constructs a *non-recursive* program to compute the delta between the view after an update and the view before the update. It first derives a regular expression that corresponds to the view definition, and then, depending on the structure of that regular expression, it generates the appropriate delta rules. Dong and Topor also discuss a modified version of this algorithm for arbitrary Datalog programs but, for arbitrary programs, the generated incremental programs are not necessarily non-recursive.

Their algorithm handles insertions only. Dong, Libkin and Wong [DLW95] showed that transitive closure cannot be maintained in traditional relational languages under deletions of edges. Furthermore, they showed that recursive queries in general cannot be maintained in languages with the expressive power of SQL (excluding, of course, SQL3 which supports recursion).

The problem of maintaining transitive closures has also been studied [CC82, CH91, Jag90, Jak92, BKV90, AISN90].

Reasoning Methods

A method for computing changes in predicates defined in safe stratified Datalog is presented by Urpí and Olivé [UO92]. Their method is based on the notion of events: *external events* are updates to base predicates; *internal events* are updates to derived predicates. A *transition* is a transformation from one database state to the next. There exist equivalences that relate the old state of each predicate with its new state. For example, such an equivalence might be that "a tuple is in the old state, if and only if the tuple is unchanged, or either deleted or modified in the new state". Given these equivalences and the rules of the deductive database, the algorithm derives *transition rules* that relate the old state of a predicate with the new state predicates and events. In addition, *insertion, deletion*, and *modification internal events rules* allow the deduction of the induced insertions, deletions, and modifications that occur in a transition. All these rules are simplified and evaluated using standard SLDNF resolution.

Küchenhoff also develops an algorithm to compute changes induced by updates to deductive databases [Küc91]. Three different classes of potential changes introduced by updates are possible. All of them are described by meta-predicates whose definitions are expressed as rules. The evaluation of these rules is done using the standard evaluation procedure of the deductive system. The first class of changes pertains to the dependency of derived facts from given updates. A specific dependency is relevant to the computation of change, if it corresponds to a successful derivation path before the update but not afterwards (and vice versa). Thus, the second class of potential changes are those to the derivation paths. The full delta is defined as the set difference between the stable model of the state before the update and the state after the update.

Magic Methods

The proposal of Anand and Vista considers deductive databases and programs that contain general recursion, negation and aggregation [Ana96, AV95]. It improves on previous results [GKM92, BLT86, GMS93]) in that it does not require that every derived relation to be stored. Their proposal includes a rewriting stage that guarantees correct evaluation of delta predicates, even when some of the intermediate results are not available in a materialized form. Another improvement of this method is that it does not require a special evaluation procedure for its implementation, but it can, quite naturally, be used with the standard naive and semi-naive evaluation procedures [Ban85, Ull88]. An optimization similar to magic sets [BR87, MPR90] is also incorporated into the algorithm. Mumick and Pirahesh [MP94] discuss the importance of integrating magic sets with traditional optimizations, such as selection pushing.

2.6 Other View-Related Work

1. **Queries Independent of Updates**

   Sometimes the updates to the database leave the views intact. Determining whether a particular view is affected by a given update is a problem that has been studied [BLT86, BCL89, Elk90, LS93]. These proposals (some for relational algebra [BLT86, BCL89] and some for Datalog [Elk90, LS93]) provide tests that the database system must execute to determine the relevance of the update to a view. To be useful, these tests should be less expensive to compute than determining, say through incremental computation, that nothing in the view does, indeed, change.

2. **Self-maintainable Views**

   The idea of self maintainable views can be summarized as follows: for certain views, given an update and the view definition – and perhaps additional information about the view and the database –, one might be able to determine that the view can be updated without accessing the database, by simply manipulating the old value of the view and the update [GM95, BCL89, Huy96]. For example, views that correspond to selections from database relations are self-maintainable, because one can check whether an inserted (or deleted) tuple in the database relation satisfies the selection condition of the view, and therefore whether it needs to be added into (or deleted from) the view.

3. **Adapting Views After Redefinitions**

   This problem refers to the following scenario. Suppose that a view is materialized and the view is redefined by changing its definition slightly. If the second view is also going to be materialized, it might be possible to use the old value of the view and adapt it to conform to the view's new definition appropriately. This problem has been studied by Gupta et al. [GMR95].

4. **Answering Queries Using Views**

   If views are materialized, the query processor might be able to use this set of materialized views, in order to answer other queries [LMSS95, FRV96]. In general, this problem is difficult, but a solution to it might be very useful, especially in applications where the data are not available directly. An example of such an application is the world wide web, where data of some conceptual schema are only available though their views provided at certain web sites [LRO96]. The problem of rewriting a query into an equivalent form that uses the views has been shown to be (at least) NP-complete [LMSS95].

2.7 Relationship to our work

In this thesis we concentrate on the optimization aspect of query expressions for view maintenance. Our work applies to the immediate update propagation strategy discussed in Section 2.1. In our framework, views are implemented as relations and changes to the database relations are also available as relations, as discussed in Sections 2.2 and 2.3. The data model for which we study the incremental view maintenance problem is a multiset algebra with SQL semantics. The change propagation expressions that we study are taken from a paper of Griffin and Libkin [GL95].

Chapter 3

Change Propagation Expressions

In this chapter we lay the foundations of incremental computation. First, we present the algebra for which change propagation and incremental expressions are studied and we show how the data manipulation language SQL maps to this algebra. Then, we present the formal definition of relation updates, database updates, change propagation expressions and incremental expressions. Finally, for each operator in the algebra, we present the change propagation expression that computes the change to the value of the operator from the inputs to the operator and their changes.

3.1 Data Model

The underlying data model for which our results about incremental computation are presented is relational algebra, sufficiently extended to be consistent with the SQL query language [DD93]. The database relations are typically sets or, less often, multisets. A relation is a *multiset* when the relation contains one or more copies of the one or more tuples. Obviously, a set is a multiset. Moreover, the results of operations on multisets are themselves multisets.

Addressing the problem of incremental computation for the algebra and not directly for SQL has a number of advantages: a) it makes the presentation of change propagation and incremental expressions compact; b) it makes the process to derive these expressions easy to understand; c) it simplifies the proofs of correctness; and, d) it makes the framework extendible to allow for the easy addition of new operators.

The operations supported by our model are described next, together with their corresponding SQL construct(s). In what follows, let A and B be two multisets.

- The expression A is the multiset A. It corresponds to SQL's "`SELECT * FROM` A" clause where A is a table.

- The *cartesian product* $A \times B$ has $c_1 \times c_2$ duplicates of tuple $t = (t_1, t_2)$, if t_1 appears c_1 times in A and t_2 appears c_2 times in B. The cartesian product corresponds to SQL's "`FROM`" clause when more than one table reference appears in it. It also corresponds to the "`CROSS JOIN`" clause.

- The *selection* $\sigma_\theta(A)$, where θ is a conditional expression, has c duplicates of tuple t, if t satisfies the condition θ and 0 duplicates if t does not satisfy the condition, for each t that appears c times in A. The selection corresponds to SQL's "`WHERE`" and "`HAVING`" clauses.

- The *projection* $\pi_X(A)$, where X is a list of select items, has as many tuples as A has. From each tuple of A, a tuple appears in $\pi_X(A)$ with only the attributes of X. A *select item* has the form "`scalar-expression [ AS column ]`", where the scalar expression typically (but not necessarily) involves one or more columns of table A. The projection corresponds to SQL's "`SELECT ALL`" and "`SELECT`" clauses, when the items appearing in them do not contain aggregate functions.

- The *duplicate elimination* $e(A)$ has one copy of each tuple $t \in A$. It corresponds to SQL's "`SELECT DISTINCT`".

- The *projection distinct* $\pi^d_X(A)$ is equivalent to $e(\pi_X(A))$ The projection distinct corresponds to SQL's "`SELECT DISTINCT`" clause, when the items appearing in it do not contain aggregate functions.

- The *aggregation* $ag_{[F;G]}(A)$ is an expression where F is a non-empty list of aggregate items and G is a (possibly empty) list of attributes of A. Each *aggregate item* has the form "$f_i(X)$ `AS column`" where f_i is an aggregate function and X is an attribute of A. Common aggregation functions are `COUNT, MAX, MIN` and `SUM`. Informally, the meaning of this operator is defined as follows: We group the tuples of A in such a way that each group contains all tuples with the same values for the attributes in G – thus having as many groups as there are distinct values for the attributes in G. If G is empty, there is only one group. Then, for each resulting group, we extend the attributes of the tuples in the group with as many new attributes as aggregate items appearing in F. The name of a new attribute is described in the corresponding aggregate item. The value of the new attribute is the result of applying the aggregate function of the aggregate item on all the tuples in the group.

More formally,

$$ag_{[F;G]}(A) = ag_{[F_1;G]}(ag_{[F_2;G]}(\ldots ag_{[F_k;G]}(A)\ldots)), \text{ where each } F_i \in F (1 \leq i \leq k)$$

and,

$$ag_{[F_i;G]}(R) = \begin{cases} R \times \{\text{aggregate } f_i(X) \text{ of } F_i \text{ applied to attr. } X \text{ of } R\} & \text{if } G = \emptyset \\ \cup_{t \in \pi^d_G(R)}(ag_{[F_i;\emptyset]}(\sigma_{R.G=t}(R))) & \text{otherwise} \end{cases}$$

- The *difference* $A - B$ has $\max\{c_1 - c_2, 0\}$ duplicates of tuple t, if A has c_1 duplicates of t and B has c_2 duplicates of t. The difference corresponds to SQL's "`EXCEPT ALL`" clause.

- The *difference distinct* $A -^d B$ has a single copy of each tuple t such that $t \in A$ and $t \notin B$. The difference distinct corresponds to SQL's "`EXCEPT`" clause.

- The *union* $A \cup B$ has $c_1 + c_2$ copies of tuple t, if A has c_1 duplicates of t and B has c_2 duplicates of t. The union corresponds to SQL's "`UNION ALL`" clause.

- The *union distinct* $A \cup^d B$ is equivalent to $e(A \cup B)$. The union distinct corresponds to SQL's "`UNION`" clause.

- The *intersection* $A \cap B$ has $min\{c_1, c_2\}$ copies of tuple t, if A has c_1 duplicates of t and B has c_2 duplicates of t. The intersection corresponds to SQL's "`INTERSECT ALL`" clause.

- The *intersection distinct* $A \cap^d B$ is equivalent to $e(A \cap B)$. The intersection distinct corresponds to SQL's "`INTERSECT`" clause.

- Finally, the *join* $A \bowtie_\theta B$ is a shorthand for $\sigma_\theta(A \times B)$, where θ is a conditional expression. The join corresponds to SQL's "A `JOIN` B `ON` θ" clause. If θ is a conjunctive condition involving only equations between attributes of A and attributes of B whose names are identical, the join also corresponds to SQL's "A `JOIN` B `USING` *attributes of* θ". Moreover, if θ contains all attributes of A and B, the join corresponds to the "A `NATURAL JOIN` B" clause.

Thus, the results of $\pi^d, -^d, \cup^d, \cap^d, e$ have no duplicates, while the result of the other operators may have duplicates.

These are all the operators that the RHODES query optimizer supports. Note that there is no operator in the algebra that corresponds to sorting. We do not regard this as a limitation of the language. The presented algebra is a declarative language. An implementation of the algebra, such as the one by RHODES, can introduce sorting, but sorting does not play an important role in the issues discussed in this chapter.

Also, note that we do not provide a formal proof of the equivalence of this algebra with SQL (modulo sorting). Ceri and Gottlob [CG85] described a two-step translation from SQL to a similar algebra. The algebra in their paper does not consider duplicates as we do. The first step of their translation generates from an arbitrary SQL expression an equivalent SQL expression that does not use several of SQL's language constructs, such as nested subqueries with EXISTS, ALL, ANY, IN, and so on. The second part of their translation describes how a grammar can be used to map expressions of this restricted form of SQL into relational algebra. The multiset algebra described here corresponds more directly to the (generalization to multisets of the) restricted form of SQL. It is easy to see that the key language constructs of SQL are preserved in the algebra and to verify that the other SQL constructs can be mapped into the ones that are maintained without difficulty.

3.2 Example

Let us see now an example of how typical SQL statements map into expressions of the presented algebra. The "Top Supplier Query" of the TPC-D Benchmark [TPC95] finds the supplier who contributed the most to the overall revenue for parts shipped in a particular year, say 1995. We assume the following database relations

$$\texttt{LINEITEM}(\texttt{L\_SUPPKEY}, \texttt{L\_PARTKEY}, \texttt{L\_SHIPDATE}, \texttt{L\_DISCOUNT}, \texttt{L\_PRICE}, \ldots)$$

$$\texttt{SUPPLIER}(\texttt{S\_SUPPKEY}, \texttt{S\_NAME}, \ldots)$$

The relation `LINEITEM` records the parts shipped by each supplier, the date of shipment, the discount offered and the total price for the entire quantity of the shipped part before any discount. The relation `SUPPLIER` records information about suppliers. To compute the top supplier(s), we can execute the following SQL statements:

```
CREATE VIEW REVENUE (SUPPLIER_NO, TOTAL_REVENUE) AS
        SELECT L_SUPPKEY, SUM(L_PRICE * (1 - L_DISCOUNT))
        FROM LINEITEM
        WHERE L_SHIPDATE = "1995"
        GROUP_BY L_SUPPKEY;

SELECT S_NAME, TOTAL_REVENUE
FROM SUPPLIER, REVENUE
WHERE S_SUPPKEY = SUPPLIER_NO
        AND TOTAL_REVENUE =
                (SELECT MAX(TOTAL_REVENUE)
                FROM REVENUE);

DROP VIEW REVENUE;
```

In our algebra the above SQL statements are equivalent to the following algebraic expressions:

$\texttt{REVENUE} = \pi_{[\mathrm{X,Y}]}(ag_{[\mathrm{F;G}]}(\sigma_\theta(\texttt{LINEITEM})))$, where

$\theta = (\texttt{L\_SHIPDATE = }\text{“1995”})$

$\texttt{F = SUM(L\_PRICE * (1 - L\_DISCOUNT)) as TOTAL\_REVENUE}$

G = L_SUPPKEY

X = L_SUPPKEY as SUPPLIER_NO and

Y = TOTAL_REVENUE

$$\pi_{[\mathrm{X,Y}]}(\texttt{SUPPLIER} \bowtie_{\theta_2} (\texttt{REVENUE} \bowtie_{\theta_1} \pi^{d}_{[\mathrm{Z}]}(ag_{[\mathrm{F};\emptyset]}(\texttt{REVENUE})))),\text{ where}$$

F = MAX(TOTAL_REVENUE) as R_MAX

Z = R_MAX

θ_1 = (TOTAL_REVENUE = R_MAX)

θ_2 = (S_SUPPKEY = SUPPLIER_NO)

X = S_NAME and

Y = R_MAX

Having presented the database relations and the set of operations for manipulating these relations, we now continue with the formal definition of change propagation and incremental expressions.

3.3 Formal Definitions

As discussed in Section 3.1, a *relation* is a finite multiset of tuples, all having the same (finite) arity. Let $\mathcal{R}$ be the set of all possible values for a relation R.

Definition 3.1. A *change* or *update* $\delta(R)$ of a relation $R \in \mathcal{R}$ is a pair $\delta(R) = (\delta_R^-, \delta_R^+)$ where $\delta_R^- \in \mathcal{R}$ and $\delta_R^+ \in \mathcal{R}$ satisfy the following properties:

$$\begin{aligned} 1.\quad \delta_R^- &\subseteq R, \\ 2.\quad \delta_R^- \cap \delta_R^+ &= \emptyset. \end{aligned}$$

We call δ_R^- the *deletions* from R and δ_R^+ the *insertions* into R.

In condition 1 of Definition 3.1 above, we use $\delta_R^- \subseteq R$ to mean that if tuple t appears c_1 times in δ_R^- and t appears c_2 times in R, then $c_1 \leq c_2$. Condition 1 states that all the (duplicates of) tuples that we delete from a relation are in fact members of that relation. Condition 2 states that no tuple is both inserted and deleted into the relation in the same update.

A *database* is a finite set $\{R_1, R_2, \ldots, R_k\}$ of relations. The relation names $R_1, R_2, \ldots, R_k$ form the *schema* of the database. Let $\mathcal{D}$ be the set of all possible databases of a given schema[1].

Definition 3.2. A *change* or *update* $\delta(D)$ of a database $D = \{R_1, R_2, \ldots, R_k\} \in \mathcal{D}$ is a pair $\delta(D) = (\delta_D^-, \delta_D^+)$ where $\delta_D^- \in \mathcal{D}$ and $\delta_D^+ \in \mathcal{D}$ satisfy the following properties:

$$\begin{array}{lrcl} 1. & \delta_D^- & = & \{\delta_{R_1}^-, \delta_{R_2}^-, \ldots, \delta_{R_k}^-\}, \\ 2. & \delta_D^+ & = & \{\delta_{R_1}^+, \delta_{R_2}^+, \ldots, \delta_{R_k}^+\}, \\ 3. & \multicolumn{3}{l}{(\delta_{R_i}^-, \delta_{R_i}^+) \text{ is an update for relation } R_i, i \leq k.} \end{array}$$

We call δ_D^- the *database deletions* and δ_D^+ the *database insertions*.

Given a relation R with deletions δ_R^- and insertions δ_R^+, the value of the relation after the update is $R^v = [R - \delta_R^-] \cup \delta_R^+ = [R \cup \delta_R^+] - \delta_R^-$. If a tuple appears c times in R, c_1 times in δ_R^- and c_2 times in δ_R^+ (obviously c_1 and c_2 cannot simultaneously be non-zero, and $c_1 \leq c$), then the tuple appears $c - c_1 + c_2$ times in R^v. Similarly, the new value of a database D is $D^v = [D - \delta_D^-] \cup \delta_D^+$, where here $-$ and $\cup$ are taken component-wise.

Definition 3.3. A *query* Q is an expression in the algebra of the data model and can be seen as a function $Q : \mathcal{D} \longrightarrow \mathcal{R}$. We call the single relation $Q(D)$ the *answer* to query Q on database D.

Let $\mathcal{Q}$ be the set of all queries in our data model.

Definition 3.4. A *change propagation query* is a function $CPQ : \mathcal{Q} \times \mathcal{D} \times (\mathcal{D} \times \mathcal{D}) \longrightarrow \mathcal{D} \times \mathcal{D}$ such that, if

1. $Q \in \mathcal{Q}$ is a query,
2. $D \in \mathcal{D}$ is a database, and
3. $\delta(D) \in \mathcal{D} \times \mathcal{D}$ is a database update,

then, $CPQ(Q, D, \delta(D))$ is an update $\delta(Q(D)) \in \mathcal{D} \times \mathcal{D}$ in the answer $Q(D)$ of query Q on database D, such that, if $\delta_{Q(D)}^-$ are the deletions and $\delta_{Q(D)}^+$ the insertions of that update, then

[1] Note that $\mathcal{D}$ may not be finite. All relations have finite attribute sets (arity), but attributes may have infinite domains.

$$[Q(D) \;-\; \delta^-_{Q(D)}] \;\cup\; \delta^+_{Q(D)} \;\;=\;\; Q([D - \delta^-_D] \cup \delta^+_D).$$

Essentially, this definition says that if the change computed by the change propagation query, is incorporated into the value of the query answer that we had *before* the database update, the result is the same as the value of the query answer *after* the database update. We call the expression of the change propagation query, *change propagation*, *delta* or *differential* expression of Q[2]. We call $\delta^-_{Q(D)}$ and $\delta^+_{Q(D)}$ the *incremental updates* to $Q(D)$ with respect to $\delta(D)$.

One corollary of Definition 3.4 is that, for each query Q, database D and database update $\delta(D)$, the value of $CPQ(Q, D, \delta(D))$ is unique, i.e., there is a unique change to every query's answer. A second corollary is that, if the change propagation query computes tuples to be deleted, then these tuples are already in $Q(D)$, and no tuple is computed to be both deleted and inserted in $Q(D)$. This is because we have defined CPQ to return a *change*.

Definition 3.5. An *incremental update* is the update defined by the change propagation query. In particular, if the change propagation query $CPQ(Q, D, \delta(D))$ computes the change $\delta(Q(D)) = (\delta^-_{Q(D)}, \delta^+_{Q(D)})$, we call $\delta^-_{Q(D)}$ the *incremental deletions*, and $\delta^+_{Q(D)}$ the *incremental insertions* of the change.

Definition 3.6. An *incremental query*, $IQ(Q, D, \delta(D))$, of a query $Q \in \mathcal{Q}$, database $D \in \mathcal{D}$ and database update $\delta(D) \in \mathcal{D} \times \mathcal{D}$, is the syntactic expression

$$[Q(D) - CPQ^-] \cup CPQ^+$$

where (CPQ^-, CPQ^+) are the incremental updates to $Q(D)$ with respect to $\delta(D)$.

Therefore, we use the term "change propagation expression" to refer to an expression that specifies the incremental updates to a query's answer while we use the term "incremental expression" to refer to an expression that incorporates incremental changes to a query's old answer.

[2]We use all these names because they all appear in the literature.

After having presented the formal definition of updates to both database relations and relations computed by running queries, we are ready to present the major issues associated with change propagation and incremental maintenance.

The Change Propagation Problem

Given a query Q, a database D and a database update $\delta(D)$, how do we define the change propagation query $CPQ(Q, D, \delta(D))$? What query expression can we use to compute incremental changes? We discuss a solution to the change propagation problem next in this chapter.

The Incremental Maintenance Problem

Given a query Q, a database D and a database update $\delta(D)$, how do we define the incremental query $IQ(Q, D, \delta(D))$? What query expression can we use to compute the new value of the query? Finding the new value of a query answer can be done in a number of ways. The first is to compute the query answer in the new database state, after the database updates have been performed. The second is to find the change to the query answer and incorporate this change into the old value of the query answer, by using an incremental expression. A third is described in Chapter 4, where we discuss the incremental maintenance problem in more detail.

The Optimization Problems

How does computing the incremental changes $CPQ(Q, D, \delta(D))$ compare to computing $Q(D)$ or $Q(D^v)$? Also, how does evaluating the incremental expression of a query compare to evaluating the query again? We address these issues experimentally in Chapter 6.

3.4 The Change Propagation Problem

Let D be a database and Q a query expression in the algebra of Section 3.1. The change propagation query $CPQ(Q, D, \delta(D))$ given a change $\delta(D)$ to the database, is the value of expressions δ_Q^- and δ_Q^+, which are defined in terms of Q and the components of Q as discussed in this section. For simplicity of the presentation we use Q to refer to both a query and the answer $Q(D)$ of the query in a database D.

To derive the change propagation expressions for all operators in the algebra, our methodology is the following. For each query Q, we algebraically manipulate the new value of the query, Q^v, in order to bring it into the form $[Q - m] \cup p$. The multisets m and p potentially define a change for Q. To prove that (m, p) is in fact a change, we prove that:

$\boxed{\text{Property 1:}}$ $m \subseteq Q$, and

$\boxed{\text{Property 2:}}$ $m \cap p = \emptyset$.

The algebraic manipulations in this chapter generate m and p in such a way that Property 1 holds but Property 2 may or may not hold. If Property 2 does not hold, then (m, p) is not a change. In this case, $(m - p, p - m)$ is a change which we consider as the change propagation expression. We can do this, because of the following theorem:

Theorem 3.1.

1. If $m \subseteq Q$, then $[Q - m] \cup p = (Q - (m - p)) \cup (p - m)$
2. If $m \subseteq Q$, then $(m - p) \subseteq Q$
3. For each m, p: $\quad (m - p) \cap (p - m) = \emptyset$

□

Proof. Let c_q, c_m and c_p be the number of duplicates of a tuple t in each of Q, m and p. To prove 1, we use the assumption that $m \subseteq Q$ to get that $c_m \leq c_q$. Tuple t appears $\max\{c_q - c_m, 0\} + c_p = c_q - c_m + c_p$ in the multiset in the left hand side of the equation. Also, tuple t appears $\max\{c_q - \max\{c_m - c_p, 0\}, 0\} + \max\{c_p - c_m, 0\}$ in the multiset in the right hand side. If $c_m \leq c_q$, this expression is $\max\{c_q, 0\} + c_p - c_m = c_q - c_m + c_p$. Otherwise, it is $\max\{c_q - (c_m - c_p)\} + 0 = c_q - c_m + c_p$. Removing the assumption $m \subseteq Q$ makes Property 1 not to hold. Similarly manipulating the number of duplicates, we can also prove the properties 2 and 3 of the theorem.

In the manipulations that follow we use the algebraic properties that appear in Table 3.1. Some of these properties are taken from the paper of Albert [Alb91] and the paper of Grumbach and Milo [GM93b] on multisets. The others can be proven easily.

P_1 :	$A \times (B \cup C) = (A \times B) \cup (A \times C)$
P_2 :	$A \times (B - C) = (A \times B) - (A \times C)$
P_3 :	$(A - B) - C = A - (B \cup C)$
P_4 :	$(A \cup B) - C = (A - C) \cup (B - (C - A))$
P_5 :	$A - (B - C) = (A - B) \cup [[C - (C - B)] - (B - A)]$
P_5 :	$A - (B - C) = (A - B) \cup [C - (B - A)]$, when $C \subseteq B$
P_6 :	$A - B = A - (B - (B - A))$
P_7 :	$(A - B) \cup (A - (A - B)) = A$
P_8 :	$(A - B) \cup C = (A \cup C) - B$, when $B \subseteq A$
P_9 :	$\sigma_\theta(A \cup B) = \sigma_\theta(A) \cup \sigma_\theta(B)$
P_{10} :	$\sigma_\theta(A - B) = \sigma_\theta(A) - \sigma_\theta(B)$
P_{11} :	$\pi_X(A \cup B) = \pi_X(A) \cup \pi_X(B)$
P_{12} :	$\pi_X(A - B) = \pi_X(A) - \pi_X(B)$, when $B \subseteq A$
P_{13} :	$e(A \cup B) = e(A) \cup [e(B) - A]$
P_{14} :	$e(A - B) = e(A) - [B - (A - B)]$
P_{15} :	$e(A) - B = e(A) - e(B)$
P_{16} :	$A - (B - C) = A - B$, when $A \cap C = \emptyset$
P_{17} :	$A - (B - A) = A$, when$A \cap B = \emptyset$
P_{18} :	$A - (A - B) = B - (B - A)$
P_{19} :	$(A \cup B) - C = (A - C) \cup B$, when$B \cap C = \emptyset$
P_{20} :	$A - (B \cup C) = A - B$, when$A \cap C = \emptyset$

Table 3.1: Properties of Multisets

We continue by presenting the change propagation expressions for each operator in the algebra. Most of these expressions have been discussed by Griffin and Libkin [GL95]. We prove the correctness of the change propagation expressions, not only for pedagogical reasons, but also because the proofs describe what over-estimations of changes may be used for view maintenance instead of the actual changes. As we will see in Chapter 4, for incremental view maintenance, it might be desirable to relax $\boxed{\text{Property 2}}$ and only insist that $\boxed{\text{Property 1}}$ be satisfied. Therefore, instead of using the derived (and simplified) $(m - p, p - m)$ to do view maintenance, we can use (m, p) directly, which are not the changes but *over-estimations* of them. Doing this has the potential to improve the performance of incremental view maintenance.

3.4.1 Change Propagation Expressions

Database Relation

If $Q = A$, for any database relation A, then:

$$\begin{aligned} \delta_Q^- &= \delta_A^- \\ \delta_Q^+ &= \delta_A^+ \end{aligned}$$

Cartesian Product

Let $del(A) = A - \delta_A^-$ and $del(B) = B - \delta_B^-$. If $Q = A \times B$, then

$$\begin{aligned} Q^v &= A^v \times B^v \\ &= (del(A) \cup \delta_A^+) \times (del(B) \cup \delta_B^+) \\ &= del(A) \times del(B) \cup \delta_A^+ \times del(B) \cup del(A) \times \delta_B^+ \cup \delta_A^+ \times \delta_B^+ \quad \text{by } P_1 \end{aligned}$$

Let $p = \delta_A^+ \times del(B) \ \cup \ del(A) \times \delta_B^+ \ \cup \ \delta_A^+ \times \delta_B^+$.

Then,

$$\begin{aligned} Q^v &= del(A) \times del(B) \cup p \\ &= (A - \delta_A^-) \times (B - \delta_B^-) \cup p \\ &= ([A \times (B - \delta_B^-)] - [\delta_A^- \times (B - \delta_B^-)]) \cup p \quad \text{by } P_2 \\ &= ([A \times B - A \times \delta_B^-] - [\delta_A^- \times B - \delta_A^- \times \delta_B^-]) \cup p \quad \text{by } P_2 \\ &= (A \times B - [A \times \delta_A^- \cup (\delta_A^- \times B - \delta_A^- \times \delta_B^-)]) \cup p \quad \text{by } P_3 \end{aligned}$$

Let $m = A \times \delta_B^- \cup (\delta_A^- \times B - \delta_A^- \times \delta_B^-) = (A \times \delta_B^- \cup \delta_A^- \times B) - \delta_A^- \times \delta_B^-$.

Then,

$$Q^v = [A \times B - m] \cup p$$

Is (m, p) a change? The answer is no. It is easy to see that $m \subseteq Q$. However, $m \cap p \neq \emptyset$. To see why, let $A = \{1, 1\}, B = \{1\}, \delta_A^- = \{1\}, \delta_B^+ = \{1\}$. Thus, $A \times B = \{(1,1), (1,1)\} = (A \times B)^v$. However, $m = \{(1,1)\}$ and $p = \{(1,1)\}$.

According to Theorem 3.1, $(m - p, p - m)$ is a change. To obtain the change propagation expressions for cartesian product, we further simplify $m - p$ and $p - m$. Before we do this let us rewrite m as follows:

$$\begin{aligned}
m &= A \times \delta_B^- \cup (\delta_A^- \times B - \delta_A^- \times \delta_B^-) \\
&= A \times \delta_B^- \cup \delta_A^- \times del(B) \\
&= (del(A) \cup \delta_A^-) \times \delta_B^- \cup \delta_A^- \times del(B) \\
&= \delta_A^- \times \delta_B^- \cup del(A) \times \delta_B^- \cup \delta_A^- \times del(B)
\end{aligned}$$

Then,

$$\begin{aligned}
m - p &= [\delta_A^- \times \delta_B^- \cup del(A) \times \delta_B^- \cup \delta_A^- \times del(B)] - \\
&\quad\; [\delta_A^+ \times del(B) \cup del(A) \times \delta_B^+ \cup \delta_A^+ \times \delta_B^+] \\
&= \delta_A^- \times \delta_B^- \cup ([del(A) \times \delta_B^- \cup \delta_A^- \times del(B)] - \\
&\quad\; [\delta_A^+ \times del(B) \cup del(A) \times \delta_B^+ \cup \delta_A^+ \times \delta_B^+]) && \text{by } P_{19} \\
&= \delta_A^- \times \delta_B^- \cup ([del(A) \times \delta_B^- \cup \delta_A^- \times del(B)] - \\
&\quad\; [\delta_A^+ \times del(B) \cup del(A) \times \delta_B^+]) && \text{by } P_{20} \\
&= \delta_A^- \times \delta_B^- \cup ([[del(A) \times \delta_B^- \cup \delta_A^- \times del(B)] - \delta_A^+ \times del(B)] - \\
&\quad\; del(A) \times \delta_B^+) && \text{by } P_3 \\
&= \delta_A^- \times \delta_B^- \cup ([[del(A) \times \delta_B^- - \delta_A^+ \times del(B)] \cup \delta_A^- \times del(B)] - \\
&\quad\; del(A) \times \delta_B^+) && \text{by } P_{19} \\
&= \delta_A^- \times \delta_B^- \cup [del(A) \times \delta_B^- - \delta_A^+ \times del(B)] \cup \\
&\quad\; [\delta_A^- \times del(B) - del(A) \times \delta_B^+] && \text{by } P_{19}
\end{aligned}$$

and,

$$\begin{aligned}
p - m &= [\delta_A^+ \times del(B) \cup del(A) \times \delta_B^+ \cup \delta_A^+ \times \delta_B^+] - \\
&\quad\; [(A \times \delta_B^- \cup \delta_A^- \times B) - \delta_A^- \times \delta_B^-] \\
&= [\delta_A^+ \times \delta_B^+ \cup \delta_A^+ \times del(B) \cup del(A) \times \delta_B^+] - \\
&\quad\; [A \times \delta_B^- \cup \delta_A^- \times B] && \text{by } P_{16} \\
&= \delta_A^+ \times \delta_B^+ \cup [(\delta_A^+ \times del(B) \cup del(A) \times \delta_B^+) - \\
&\quad\; (A \times \delta_B^- \cup \delta_A^- \times B)] && \text{by } P_{19} \\
&= \delta_A^+ \times \delta_B^+ \cup ([[\delta_A^+ \times del(B) \cup del(A) \times \delta_B^+] - \\
&\quad\; \delta_A^- \times B] - A \times \delta_B^-) && \text{by } P_3 \\
&\quad \delta_A^+ \times \delta_B^+ \cup [[(del(A) \times \delta_B^+ - \delta_A^- \times B) \cup \\
&\quad\; \delta_A^+ \times del(B)] - A \times \delta_B^-] && \text{by } P_{19}
\end{aligned}$$

$$\begin{aligned} = \; & \delta_A^+ \times \delta_B^+ \cup [del(A) \times \delta_B^+ - \delta_A^- \times B] \cup \\ & [\delta_A^+ \times del(B) - A \times \delta_B^-] \qquad \text{by } P_{19} \end{aligned}$$

Selection

If $Q = \sigma_\theta(A)$, then:

$$\begin{aligned} Q^v &= \sigma_\theta(A^v) \\ &= \sigma_\theta((A - \delta_A^-) \cup \delta_A^+) \\ &= [\sigma_\theta(A) - \sigma_\theta(\delta_A^-)] \cup \sigma_\theta(\delta_A^+) \quad \text{by } P_9 \text{ and } P_{10} \end{aligned}$$

Let $m = \sigma_\theta(\delta_A^-)$ and $p = \sigma_\theta(\delta_A^+)$. It is clear that (m, p) is a change since both properties of Definition 3.1 are trivially satisfied.

Projection

If $Q = \pi_X(A)$, then:

$$\begin{aligned} Q^v &= \pi_X(A^v) \\ &= [\pi_X(A) - \pi_X(\delta_A^-)] \cup \pi_X(\delta_A^+) \quad \text{by } P_{11} \text{ and } P_{12} \end{aligned}$$

Let $m = \pi_X(\delta_A^-)$ and $p = \pi_X(\delta_A^+)$. Unlike the case of selection, (m, p) is not a change. It is easy to see that $m \subseteq Q$. However, $m \cap p \neq \emptyset$. To see why, consider $A(X, Y) = \{(1, 1)\}, \delta_A^- = \{(1, 1)\}, \delta_A^+ = \{(1, 2)\}$. Supposing the projection retains the first attribute, X, of relation A, both $m = p = \{1\}$.

According to Theorem 4.1, $(m - p, p - m)$ is a change.

Duplicate Elimination

If $Q = e(A)$, then

$$\begin{aligned}
Q^v &= e((A-\delta_A^-)\cup\delta_A^+) && \\
&= e(A-\delta_A^-)\cup[e(\delta_A^+)-(A-\delta_A^-)] && \text{by } P_{13}\\
&= e(A-\delta_A^-)\cup[e(\delta_A^+)-A] && \text{by } P_{16}\\
&= [e(A)-(\delta_A^- -(A-\delta_A^-))]\cup[e(\delta_A^+)-A] && \text{by } P_{14}\\
&= [e(A)-e(\delta_A^- -(A-\delta_A^-))]\cup[e(\delta_A^+)-A] && \text{by } P_{15}\\
&= [e(A)-[e(\delta_A^-)-((A-\delta_A^-)-(\delta_A^- -(A-\delta_A^-)))]]\cup[e(\delta_A^+)-A] && \text{by } P_{14}\\
&= [e(A)-[e(\delta_A^-)-(A-\delta_A^-)]]\cup[e(\delta_A^+)-A] && \text{by } P_{17}
\end{aligned}$$

Let $m = e(\delta_A^-) - (A - \delta_A^-)$ and $p = e(\delta_A^+) - A$. Intuitively, m contains a unique copy of those tuples deleted from A that lost all the duplicates they had in A, while p contains those tuples inserted in A that did not exist in A before. The derived (m, p) is a change.

Set Difference

If $Q = A - B$, let $del(A) = A - \delta_A^-$ and $ins(A) = A \cup \delta_A^+$. Obviously, $A^v = del(A) \cup \delta_A^+ = ins(A) - \delta_A^-$. The following equivalences can easily be proven

$T_1:\ ins(A) - B = (A - B) \cup [\delta_A^+ - (B - A)]$ by P_4

$T_2:\ A - ins(B) = (A - B) - \delta_B^+$ by P_3

$T_3:\ del(A) - B = (A - B) - \delta_A^-$ by P_3 (twice)

$T_4:\ A - del(B) = (A - B) \cup [\delta_B^- - (B - A)]$ by P_5

We use the equivalences T_1 to T_4 and the equivalences $A^v = del(A) \cup \delta_A^+$ and $B^v = ins(B) - \delta_B^-$ to get the following:

$$\begin{aligned}
Q^v &= A^v - B^v && \\
&= (del(A)-B^v)\cup[\delta_A^+ -(B^v - del(A))] && \text{by } T_1\\
&= (del(A)-ins(B))\cup[\delta_B^- -(ins(B)-del(A))]\cup[\delta_A^+ -(B^v-del(A))] && \text{by } T_4\\
&= (del(A)-ins(B))\cup[\delta_B^- -(ins(B)-del(A))]\cup && \\
&\quad [\delta_A^+ -((ins(B)-del(A))-\delta_B^-)] && \text{by } T_3\\
&= (del(A)-ins(B))\cup[(\delta_B^-\cup\delta_A^+)-(ins(B)-del(A))] && \text{by } P_4
\end{aligned}$$

Let $X = \delta_A^- \cup \delta_B^+$ and $Y = \delta_B^- \cup \delta_A^+$.

And let,

$$
\begin{aligned}
p &= (\delta_B^- \cup \delta_A^+) - (ins(B) - del(A)) \\
&= Y - (ins(B) - del(A)) \\
&= Y - [(B - del(A)) \cup (\delta_B^+ - (del(A) - B))] && \text{by } T_1 \\
&= Y - [(B - A) \cup (\delta_A^- - (A - B)) \cup [\delta_B^+ - ((A - B) - \delta_A^-)]] && \text{by } T_4 \text{ and } T_3 \\
&= Y - [(B - A) \cup ((\delta_A^- \cup \delta_B^+) - (A - B))] && \text{by } P_4 \\
&= (Y - (B - A)) - (X - (A - B)) && \text{by } P_3
\end{aligned}
$$

Then,

$$
\begin{aligned}
Q^v &= (del(A) - ins(B)) \cup p \\
&= [(del(A) - B) - \delta_B^+] \cup p && \text{by } T_2 \\
&= [(A - B) - (\delta_A^- \cup \delta_B^+)] \cup p && \text{by } T_3 \text{ and } P_3 \\
&= [(A - B) - X] \cup p \\
&= [(A - B) - [X - (X - (A - B))]] \cup p && \text{by } P_6
\end{aligned}
$$

Let $m = X - [X - (A - B)]$.

Then,

$$
Q^v = [(A - B) - m] \cup p
$$

The first question that we must ask is whether the derived (m, p) is a change. The answer is no. Obviously, $m \subseteq Q$. However, $m \cap p \neq \emptyset$. To see why, let $A = \{1, 1\}, B = \{1\}, \delta_A^- = \delta_B^- = \{1\}$. Thus, $A - B = A^v - B^v = \{1\}$ and $B - A = \emptyset$. Also, $X = Y = \{1\}$. Then $m = \{1\} - (\{1\} - \{1\}) = \{1\}$ and $p = (\{1\} - \emptyset) - [\{1\} - \{1\}] = \{1\}$.

According to Theorem 3.1, $(m - p, p - m)$ is a change.

Let us simplify $m - p$:

$$
\begin{aligned}
m - p &= [X - (X - (A - B))] - [(Y - (B - A)) - (X - (A - B))] \\
&= [(A - B) - ((A - B) - X)] - [(Y - (B - A)) - (X - (A - B))] && \text{by } P_{18} \\
&= (A - B) - [((A - B) - X) \cup [(Y - (B - A)) - (X - (A - B))]] && \text{by } P_3 \\
&= (A - B) - [[(A - B) \cup (Y - (B - A))] - X] && \text{by } P_4 \\
&= (A - B) - [[(Y \cup (A - B)) - (B - A)] - X] && \text{by } P_{19} \\
&= (A - B) - [(Y \cup (A - B)) - (X \cup (B - A))] && \text{by } P_3 \\
&= (A - B) - [((A - B) - (X \cup (B - A))) \cup \\
&\quad [Y - ((X \cup (B - A)) - (A - B))]] && \text{by } P_4 \\
&= (A - B) - [((A - B) - X) \cup (Y - (X - (A - B)))] && \text{by } P_4 \\
&= (A - B) - [((A - B) \cup Y) - X] && \text{by } P_4 \\
&= [(A - B) - ((A - B) \cup Y)] \cup \\
&\quad [(X - (X - ((A - B) \cup Y))) - [(Y \cup (A - B)) - (A - B)]] && \text{by } P_5 \\
&= [X - (X - ((A - B) \cup Y))] - Y \\
&= (X - Y) - (X - ((A - B) \cup Y)) && \text{by } P_3 \\
&= (X - Y) - [(X - Y) - (A - B)] \\
&= (X - Y) \cap (A - B)
\end{aligned}
$$

Also, let us simplify $p - m$:

$$
\begin{aligned}
p - m &= [(Y - (B - A)) - (X - (A - B))] - [X - (X - (A - B))] \\
&= (Y - (B - A)) - [(X - (A - B)) \cup [X - (X - (A - B))]] && \text{by } P_3 \\
&= [Y - (B - A)] - X && \text{by } P_7 \\
&= (Y - X) - (B - A) && \text{by } P_3 \text{ (twice)}
\end{aligned}
$$

Union

If $Q = A \cup B$, then:

$$
\begin{aligned}
Q^v &= A^v \cup B^v \\
&= (A - \delta_A^-) \cup (B - \delta_B^-) \cup \delta_A^+ \cup \delta_B^+
\end{aligned}
$$

Let $p = \delta_A^+ \cup \delta_B^+$.

Then,

$$\begin{aligned} Q^v &= ((A \cup (B - \delta^- B)) - \delta_A^-) \cup p & \text{by } P_8 \\ &= (((A \cup B) - \delta_B^-) - \delta_A^-) \cup p & \text{by } P_8 \\ &= ((A \cup B) - (\delta_A^- \cup \delta_B^-)) \cup p & \text{by } P_3 \end{aligned}$$

Let $m = \delta_A^- \cup \delta_B^-$.

Obviously $m \subseteq (A \cup B)$. However (m, p) is not a change, because $m \cap p \neq \emptyset$, when $\delta_A^- \cap \delta_B^+ \neq \emptyset$ or $\delta_A^+ \cap \delta_B^- \neq \emptyset$. According to Theorem 3.1, $(m - p, p - m)$ is a change. In fact, $m - p, p - m$ are the change propagation expressions for union.

Aggregation

Aggregation is the only operator in the algebra for which we do not adopt the algebraic approach. Rather, for aggregation, we describe algorithmically how the updates to views with aggregates are defined. We do that first for views where aggregation is computed with respect to all the tuples in the relation (one group), and then for views where aggregation is computed with respect to some grouping attributes (many groups).

Aggregation over one group

COUNT

Let $Q = ag_{[\texttt{COUNT}(X) \text{ as } Y;\emptyset]}(A)$. Let us abbreviate with F the aggregate item "COUNT(X) as Y". For COUNT, the aggregation column Y holds the number of different tuples appearing in A. Let,

- $C = \pi_Y^d(Q)$, i.e., $c \in C$ is the number of tuples in A before the update,
- $C^+ = \pi_Y^d(ag_{[F;\emptyset]}(\delta_A^+))$, i.e., $c^+ \in C^+$ is the number of tuples inserted into A, and
- $C^- = \pi_Y^d(ag_{[F;\emptyset]}(\delta_A^-))$, i.e., $c^- \in C^-$ is the number of tuples deleted from A.

We have two cases:

1. If $c^- = c^+$, then the number of tuples in A before the update is the same as the number of tuples in A after the update. The value of the aggregate value of the view is not affected. The change to the view is

$$\begin{aligned}\delta_Q^- &= \delta_A^- \times \{c\}\\ \delta_Q^+ &= \delta_A^+ \times \{c\}\end{aligned}$$

2. If $c^- \neq c^+$, the number of tuples in A before the update is not the same as the number of tuples in A after the update, which is $c - c^- + c^+$. Therefore all the view tuples must change, and the update to the view is:

   $$\begin{aligned}\delta_Q^- &= Q\\ \delta_Q^+ &= A^v \times \{c - c^- + c^+\}\end{aligned}$$

   This shows a limitation of our approach to model only insertions and deletions. In this case, it is better to say that $\delta_Q^- = \delta_A^- \times \{c\}$ are the tuples that are deleted from the view, $\delta_Q^+ = \delta_A^+ \times \{c - c^- + c^+\}$ are the tuples that are inserted into the view, and the other tuples in the view are modified by changing the aggregate value from c to $c - c^- + c^+$. However, here we are concerned with the change propagation problem and not with the incremental maintenance problem.

We can test for conditions 1 and 2 by running the tests $Test_1$ and $Test_2$ respectively[3],

$$\begin{aligned}Test_1 &= C^- \cap C^+\\ Test_2 &= C^- - C^+\end{aligned}$$

Then, we can summarize the changes to Q as follows:

$$\begin{aligned}\delta_Q^- &= \pi_{\text{attrs of Q}}([Test_1 \times \delta_A^- \times C] \cup [Test_2 \times Q])\\ \delta_Q^+ &= \pi_{\text{attrs of Q}}([Test_1 \times \delta_A^+ \times C] \cup [Test_2 \times A^v \times \pi_{\$1-\$2+\$3}(C \times C^- \times C^+)])\end{aligned}$$

The role of the dollar sign in the expression for δ_Q^+ is to give the value of the corresponding attribute. Thus, \$1 is the value of the first attribute, and so on.

SUM

The aggregate `SUM` is treated identically to `COUNT`.

MIN

Let $Q = ag_{[\texttt{MIN}(X) \text{ as } Y;\emptyset]}(A)$. Let us abbreviate with F the aggregate item "`MIN`(X) as Y". For `MIN`, the aggregation column Y holds the minimum value of the attribute X. Let,

[3] The test is successful if and only if the relation corresponding to the test is non-empty.

- $M = \pi^d_Y(Q)$, i.e., $m \in M$ is the minimum value for the attribute X in A before the update,
- $M^+ = \pi^d_Y(ag_{[F;\emptyset]}(\delta^+_A))$, i.e., $m^+ \in M^+$ is the minimum value for X from all the inserted tuples, and
- $M^- = \pi^d_Y(ag_{[F;\emptyset]}(\delta^-_A))$, i.e., $m^- \in M^-$ is the minimum value for X from all the deleted tuples.

We have four cases:

1. If $m^+ < m$, a new minimum is inserted and all the view tuples must change. The update to the view is:

$$\begin{aligned} \delta^-_Q &= Q \\ \delta^+_Q &= A^v \times \{m^+\} \end{aligned}$$

2. If $m^+ \geq m$ and $m^- > m$, the minimum X value is not affected. The update to the view is:

$$\begin{aligned} \delta^-_Q &= \delta^-_A \times \{m\} \\ \delta^+_Q &= \delta^+_A \times \{m\} \end{aligned}$$

3. Let $C = \pi^d_Z(ag_{[\texttt{COUNT}(X) \text{ as } Z;\emptyset]}(\sigma_{X=m}(A)))$ and $C^- = \pi^d_Z(ag_{[\texttt{COUNT}(X) \text{ as } Z;\emptyset]}(\sigma_{X=m}(\delta^-_A)))$. Thus, $c \in C$ counts the number of tuples from A with the minimum value for X and $c^- \in C^-$ counts the number of tuples deleted from A that give the minimum value for X. Then, if $m^+ \geq m$, $m^- = m$ and $c > c^-$, some tuples with the minimum X value are deleted but not all of them, and, therefore, the minimum value is not affected. The update to the view is:

$$\begin{aligned} \delta^-_Q &= \delta^-_A \times \{m\} \\ \delta^+_Q &= \delta^+_A \times \{m\} \end{aligned}$$

4. If $m^+ \geq m$, $m^- = m$ and $c = c^-$, all the tuples with the minimum X value are deleted. The new minimum must be found. The update to the view is:

$$\begin{aligned} \delta^-_Q &= Q \\ \delta^+_Q &= ag_{[F;\emptyset]}(A^v) \end{aligned}$$

We can test for conditions 1, 2, 3 and 4 by running the tests $Test_1$, $Test_2$, $Test_3$, and $Test_4$ respectively,

$$
\begin{aligned}
Test_1 &= \pi_{\$1}(\sigma_{\$2<\$1}(M \times M^+)) \\
Test_2 &= \pi_{\$1}(\sigma_{\$2\geq\$1\wedge\$1<\$3}(M \times M^+ \times M^-)) \\
Test_3 &= \pi_{\$1}(\sigma_{\$2\geq\$1\wedge\$1=\$3\wedge\$4>\$5}(M \times M^+ \times M^- \times C \times C^-)) \\
Test_4 &= \pi_{\$1}(\sigma_{\$2\geq\$1\wedge\$1=\$3\wedge\$4=\$5}(M \times M^+ \times M^- \times C \times C^-))
\end{aligned}
$$

Then, the changes to Q are described by the following:

$$
\begin{aligned}
\delta_Q^- &= \pi_{\text{attrs of Q}}([(Test_1 \cup Test_4) \times Q] \cup [(Test_2 \cup Test_3) \times \delta_A^- \times M]) \\
\delta_Q^+ &= \pi_{\text{attrs of Q}}([Test_1 \times A^v \times M^+] \cup [(Test_2 \cup Test_3) \times \delta_A^+ \times M] \cup [Test_4 \times ag_{[F;\emptyset]}(A^v)])
\end{aligned}
$$

MAX

The aggregate `MAX` is treated identically to `MIN`.

Aggregation over many groups

Let $Q = ag_{[F;G]}(A)$, where F is an aggregate item involving one of the aggregate functions `COUNT, SUM, MIN` or `MAX`. The groups that are affected by the update to A can be determined by $\pi_G^d(\delta_A^- \cup \delta_A^+)$. For each affected group $g \in \pi_G^d(\delta_A^- \cup \delta_A^+)$, we have

- $\sigma_{G=g}(A)$, is the group in the relation A,
- $\sigma_{G=g}(\delta_A^-)$, are the deletions to this group, and,
- $\sigma_{G=g}(\delta_A^+)$, are the insertions to this group.

We can use the techniques discussed for the case of one group to find the updates to each group. The update to the view is the union of the updates for each group.

Other Operators

The change propagation expressions for the rest of the operators are derived by rewriting their expressions. Thus,

- if $Q = A -^d B$, then Q can be rewritten as $Q = e(A) - B$;
- if $Q = \pi_X^d(A)$, then Q can be rewritten as $Q = e(\pi_X(A))$;

- if $Q = A \cup^d B$, then Q can be rewritten as $Q = e(A \cup B)$;
- if $Q = A \cap B$, then Q can be rewritten as $Q = A - (A - B)$;
- if $Q = A \cap^d B$, then Q can be rewritten as $Q = e(A) \cap B = A \cap e(B) = e(A \cap B)$;
- if $Q = A \bowtie_\theta B$, then Q can be rewritten as $Q = \sigma_\theta(A \times B)$.

Chapter 4

Optimizations

In this chapter we propose a variety of optimization strategies in order to simplify change propagation and incremental expressions. There are three main categories of such optimizations: optimization specific to incremental maintenance, optimization in the presence of key constraints, and optimization in the presence of foreign key references.

4.1 The Incremental Maintenance Problem

The incremental maintenance problem is related to the change propagation problem discussed in the previous chapter. The change propagation problem pertains to the definition of changes to the value of queries. The incremental maintenance problem pertains to the definition of the new value of the query using the old value of the query and the changes to it. In the previous chapter we showed how equations for change propagation are derived for each operator in the multiset algebra. In this chapter, we introduce the concept of *over-estimations* of changes that have the potential to improve the performance of incremental view maintenance.

The methodology used in the previous chapter to derive the change propagation equations was the following:

1. we start with the new value of a query Q^v,

2. we algebraically manipulate the expressions for Q^v to bring it into the form $[Q - m] \cup p$, in such a way as to ensure that $\boxed{\text{Property 1: } m \subseteq Q}$ is satisfied,

3. we use theorem 3.1 to define $(m-p, p-m)$ as the change propagation expressions, when the generated m and p do not satisfy $\boxed{\text{Property 2: } m \cap p = \emptyset}$, and

4. we simplify $(m-p, p-m)$, if possible.

One question that we might want to ask is: what is the significance of the properties that the change propagation expressions satisfy? Why do we insist on them? Both these properties are very important when the change propagation expressions are used for integrity constraint maintenance, or reasoning about change. In other words, when the change of the old and new value of a query is necessary. Griffin and Libkin refer to satisfaction of both these properties as the *minimality condition* [GL95]. We make the observation that we can relax the second property when incrementally maintaining the query. We can do this, because $\forall m, p, Q$ such that $m \subseteq Q$,

$$[Q - (m-p)] \cup (p-m) \quad = \quad [Q-m] \cup p$$

We cannot relax the first property, as the above equation no longer holds.

The tradeoff between using (m, p) and using $(m-p, p-m)$ is analogous to duplicate removal. In the same way that early duplicate removal may improve the performance of subsequent operations (because it reduces the sizes of the relations involved in subsequent operations), computing minimum deltas must be done as early as possible. On the other hand, like duplicate removal, minimum delta computation is expensive, and expensive operations should be avoided if possible.

Definition 4.1. For each query Q, database D and database update $\delta(D)$, any (Δ_Q^-, Δ_Q^+) that satisfy

$$\begin{array}{ll} 1. & Q^v(D) = [Q(D) - \Delta_Q^-(D)] \cup \Delta_Q^+(D) \\ 2. & \Delta_Q^-(D) \subseteq Q(D) \end{array}$$

specify *over-estimations* of the change to the query answer $Q(D)$.

In the previous chapter, in Section 3.4.1, we showed how we define the incremental changes δ_Q^- (the equation for $m-p$) and δ_Q^+ (the equation for $p-m$) for each query expression Q. Tables 4.1 and 4.2 summarize these definitions.

$$
\begin{array}{|lcl|}
\hline
\delta^-(\sigma_\theta(A)) & = & \sigma_\theta(\delta_A^-) \\
\delta^-(\pi_X(A)) & = & \pi_X(\delta_A^-) - \pi_X(\delta_A^+) \\
\delta^-(e(A)) & = & e(\delta_A^-) - (A - \delta_A^-) \\
\delta^-(A \times B) & = & \delta_A^- \times \delta_B^- \cup [(A - \delta_A^-) \times \delta_B^- - \delta_A^+ \times (B - \delta_B^-)] \cup \\
 & & \qquad [\delta_A^- \times (B - \delta_B^-) - (A - \delta_A^-) \times \delta_B^+] \\
\delta^-(A - B) & = & [(\delta_A^- \cup \delta_B^+) - (\delta_B^- \cup \delta_A^+)] \cap (A - B) \\
\delta^-(A \cup B) & = & (\delta_A^- \cup \delta_B^-) - (\delta_A^+ \cup \delta_B^+) \\
\hline
\end{array}
$$

Table 4.1: Equations for computing δ^-

$$
\begin{array}{|lcl|}
\hline
\delta^+(\sigma_\theta(A)) & = & \sigma_\theta(\delta_A^+) \\
\delta^+(\pi_X(A)) & = & \pi_X(\delta_A^+) - \pi_X(\delta_A^-) \\
\delta^+(e(A)) & = & e(\delta_A^+) - A \\
\delta^+(A \times B) & = & \delta_A^+ \times \delta_B^+ \cup [(A - \delta_A^-) \times \delta_B^+ - \delta_A^- \times B] \cup \\
 & & \qquad [\delta_A^+ \times (B - \delta_B^-) - A \times \delta_B^-] \\
\delta^+(A - B) & = & ((\delta_B^- \cup \delta_A^+) - (\delta_A^- \cup \delta_B^+)) - (B - A) \\
\delta^+(A \cup B) & = & (\delta_A^+ \cup \delta_B^+) - (\delta_A^- \cup \delta_B^-) \\
\hline
\end{array}
$$

Table 4.2: Equations for computing δ^+

The proofs of correctness of Section 3.4.1 for the definitions of the incremental changes $m-p$ and $p-m$ also provide *one* possible set of over-estimations, m for deletions and p for insertions. Thus, m gives the over-estimation of deletions Δ^- and p gives the over-estimation of insertions Δ^+. These Δ^- and Δ^+ are expressed in terms of the δ^- and δ^+ of the inputs in Section 3.4.1. Except for the case of duplicate elimination, in deriving the equations for them, we have not used the fact that the δ^- and δ^+ of each input are disjoint multisets and therefore we can safely substitute Δ^- and Δ^+ for the inputs in place of their δ^- and δ^+. In the case of duplicate elimination, it is not possible to define over-estimations of the changes. The actual incremental changes are considered as their trivial over-estimations.

Tables 4.3 and 4.4 summarize the definitions of the over-estimations Δ^- and Δ^+. From now on, when we say over-estimations Δ^- and Δ^+, we will mean the over-estimations defined by the equations in these tables.

$\Delta^-(A)$	$=$	δ^-_A	for database relation A
$\Delta^-(\sigma_\theta(A))$	$=$	$\sigma_\theta(\Delta^-_A)$	
$\Delta^-(\pi_X(A))$	$=$	$\pi_X(\Delta^-_A)$	
$\Delta^-(e(A))$	$=$	$\delta^-(e(A))$	
$\Delta^-(A \times B)$	$=$	$[\Delta^-_A \times B \cup A \times \Delta^-_B] - \Delta^-_A \times \Delta^-_B$	
$\Delta^-(A - B)$	$=$	$(\Delta^-_A \cup \Delta^+_B) \cap (A - B)$	
$\Delta^-(A \cup B)$	$=$	$\Delta^-_A \cup \Delta^-_B$	

Table 4.3: Equations for computing Δ^-

$\Delta^+(A)$	$=$	δ^+_A	for database relation A
$\Delta^+(\sigma_\theta(A))$	$=$	$\sigma_\theta(\Delta^+_A)$	
$\Delta^+(\pi_X(A))$	$=$	$\pi_X(\Delta^+_A)$	
$\Delta^+(e(A))$	$=$	$\delta^+(e(A))$	
$\Delta^+(A \times B)$	$=$	$[\Delta^+_A \times (B - \Delta^-_B) \cup (A - \Delta^-_A) \times B] \cup \Delta^+_A \times \Delta^+_B$	
$\Delta^+(A - B)$	$=$	$[(\Delta^-_B \cup \Delta^+_A) - (B - A)] - [(\Delta^-_A \cup \Delta^+_B) - (A - B)]$	
$\Delta^+(A \cup B)$	$=$	$\Delta^+_A \cup \Delta^+_B$	

Table 4.4: Equations for computing Δ^+

The relationships between the operators discussed in this section are as follows:

1. the incremental changes are the net effect of their over-estimations, i.e., $\delta^-_Q = \Delta^-_Q - \Delta^+_Q$ and $\delta^+_Q = \Delta^+_Q - \Delta^-_Q$;

2. each over-estimation contains the incremental change and an excess of tuples Δ (possibly empty), i.e., there exists a multiset of tuples Δ such that $\Delta^-_Q = \delta^-_Q \cup \Delta$, $\Delta^+_Q = \delta^+_Q \cup \Delta$, and $\Delta = \Delta^-_Q \cap \Delta^+_Q$.

4.2 Optimization in the Absence of Duplicates

The equations for the delta and the over-estimation of the delta of a relation can be simplified, if the relation is known not to contain duplicates. To verify that a relation does not contain duplicates, we can use the following sufficient (but not necessary) conditions:

> A relation does not contain duplicates, if
>
> - the relation is generated by $\pi^d, e, \cap^d, \cup^d, -^d$, or
> - the relation always contains at most one tuple, or
> - the relation contains at least one key.

Like in relational algebra, relations in our multiset algebra may have sets of one or more attributes serving as keys. We say that a non-empty set S of attributes of relation R is a *key* for R, if no instance of R can have two tuples that agree in all the attributes of S. Therefore, it follows from the definition that a relation with at least one key does not contain duplicates. For database relations that contain at least one key, exactly one of them is designated as the *primary key*; the others are called *alternate* or *candidate* keys.

If R is generated by one of the operators $\pi^d, e, \cap^d, \cup^d$ or $-^d$, then R has a key. The key is formed by taking all the attributes of R. Also, if every instance of R is known to contain at most one tuple, then each attribute of R is sufficient to determine a unique tuple in the relation (if one exists), and any attribute can serve as a key. This allows us to revise the sufficient condition for checking duplicates to:

> A relation does not contain duplicates, if the relation contains at least one key.

From the above discussion it follows that a relation may have more than one key. The set *keys(R)* contains the keys for the relation R. Next, we describe how the keys for a relation computed by a query expression are generated from the inputs to the top operator in the query expression and their keys.

4.2.1 Generating Keys for Query Expressions

The set of keys of a relation specified using a query expression can be generated recursively by applying the following algorithm (bottom-up) to all operators in the query expression.

We use the boolean condition $01tuple(R)$ to be true when R is known to contain at most one tuple. It is easy to check whether a relation contains at most one tuple (without computing the value of the relation) by examining its keys: a relation contains at most one tuple if the empty set of attributes is a key. The algorithm for key derivation is based on the following inference rules[1]:

- $A \times B$: The set of keys of the cartesian product contains all possible combinations of the keys of A with the keys of B, if both A and B have non-empty sets of keys and both A and B contain more that one tuple. In particular:

$$keys(A \times B) = \begin{cases} \emptyset & \text{if } keys(A) = \emptyset \text{ or } keys(B) = \emptyset \\ keys(A) & \text{if } \neg 01tuple(A) \wedge 01tuple(B) \\ keys(B) & \text{if } 01tuple(A) \wedge \neg 01tuple(B) \\ keys(A) \cup keys(B) & \text{if } 01tuple(A) \wedge 01tuple(B) \\ \{k_1 \cup k_2 : k_1 \in keys(A), k_2 \in keys(B)\} & \text{otherwise} \end{cases}$$

- $\sigma_\theta(A)$: The key set of the selection is the same as the key set of the input, if the selection filters more than one tuple from A:

$$keys(\sigma_\theta(A)) = \begin{cases} \{\{X\} : X \in attrs(A)\} & \text{if } \theta \text{ selects 0 or 1 tuples} \\ keys(A) & \text{if } \theta \text{ selects } > 1 \text{ tuples} \end{cases}$$

If the selection filters more than one tuple and the selection condition θ *bounds* some attributes of A, the keys can be simplified by removing the bound attributes from them. An attribute of a relation is said to be *bound*, if all tuples in the relation contain the same value for that attribute. Let S be the set of attributes of A that the condition θ bounds. Also, let X^+ be the set of attributes that θ directly or indirectly equates with the X attribute. In every key containing X, we can substitute X with $Y \in X^+$. The simplification process is:

[1]This is a sound but not complete set of inference rules.

Repeat the following until no new keys are added into $keys(\sigma_\theta(A))$:

1. Replace each $k \in keys(\sigma_\theta(A))$ with $k - S$; if $k - S = \emptyset$, stop and set $01tuple(\sigma_\theta(A))$ to true – the selection returns at most one tuple because all attributes in a key are bound.
2. For each attribute X, for each $Y \in X^+$ and each $k \in keys(\sigma_\theta(A))$ such that $X \in k$, add $k - \{X\} \cup \{Y\}$ into $keys(\sigma_\theta(A))$.

- $A \bowtie_\theta B$: the set of keys for the join is the same as the set of keys for the selection $\sigma_\theta(A \times B)$. However, an additional simplification is possible due to the join. Again, let S be the set of bound attributes of θ. If θ is a conjunctive condition which contains an equality between the X attribute of A and the Y attribute of B, then:

  1. if $\{X\} \in keys(A)$, then $\forall k_2 \in keys(B)$ add $k_2 - S$ into $keys(A \bowtie_\theta B)$, and
  2. if $\{Y\} \in keys(B)$, then $\forall k_1 \in keys(A)$ add $k_1 - S$ into $keys(A \bowtie_\theta B)$.

- $\pi_X(A)$: if the list of attributes X maintains some keys from A then these keys are the keys for the projection. If X maintains no key from A, the projection does not have a key:

$$keys(\pi_X(A)) = \{X \cap k : k \in keys(A) \text{ and } X \cap k \in keys(A)\}$$

- $\pi^d_X(A)$ (and duplicate elimination e): the projection distinct, because it generates no duplicates in the output has the set of all attributes in the output relation as a key, if no other key can be found for it:

$$keys(\pi^d_X(A)) = \begin{cases} keys(\pi_X(A)) & \text{if } keys(\pi_X(A)) \neq \emptyset \\ \{X\} & \text{otherwise} \end{cases}$$

- $ag_{[F;G]}(A)$: the grouping attributes serve as a key for the aggregation, if the relation does not have a key:

$$keys(ag_{[F;G]}(A)) = \begin{cases} keys(A) & \text{if } keys(A) \neq \emptyset \\ \{G\} & \text{otherwise} \end{cases}$$

- $A - B$: the keys for the set difference is the same as the keys of the left input:

$$keys(A - B) = keys(A)$$

- $A -^d B$: the set of attributes in the output relation serve as a key if the left input does not have a key:

$$keys(A -^d B) = \begin{cases} keys(A - B) & \text{if } keys(A - B) \neq \emptyset \\ \{\{X : X \in attrs(A)\}\} & \text{otherwise} \end{cases}$$

- $A \cup B$: even if both input relations have keys there is no guarantee that any of them can serve as a key for the output relation, therefore:

$$keys(A \cup B) = \emptyset$$

- $A \cup^d B$: the set of attributes in the output relation can serve as keys:

$$keys(A \cup^d B) = \{\{X : X \in attrs(A)\}\}$$

- $A \cap B$: a key from each of the inputs can serve as key in the output relation:

$$keys(A \cap B) = keys(A) \cup keys(B)$$

- $A \cap^d B$: the set of attributes in the output relation serve as a key, if $A \cap B$ does not have a key:

$$keys(A \cap^d B) = \begin{cases} keys(A \cap B) & \text{if } keys(A \cap B) \neq \emptyset \\ \{\{X : X \in attrs(A)\}\} & \text{otherwise} \end{cases}$$

4.2.2 Simplifications

If a relation does not contain duplicates, the equations for defining its incremental changes can be simplified significantly. Next, we describe, for each basic operator, what these simplified versions of the change propagation expressions are.

- $A \times B$: If the output relation of a cartesian product $A \times B$ does not contain duplicates, the inputs A and B to the cartesian product also do not contain duplicates and,

$$
\begin{aligned}
\delta^-(A \times B) &= \text{(by definition)} \\
&\quad \delta_A^- \times \delta_B^- \cup [(A - \delta_A^-) \times \delta_B^- - \delta_A^+ \times (B - \delta_B^-)] \cup \\
&\quad [\delta_A^- \times (B - \delta_B^-) - (A - \delta_A^-) \times \delta_B^+] \\
&= \text{(because } A \text{ and } B \text{ do not have duplicates and, i.e., } (A - \delta_A^-) \cap \delta_A^+ = \emptyset) \\
&\quad \delta_A^- \times \delta_B^- \cup [(A - \delta_A^-) \times \delta_B^-] \cup [\delta_A^- \times (B - \delta_B^-)] \\
&= \text{(property of multisets)} \\
&\quad [\delta_A^- \cup (A - \delta_A^-)] \times \delta_B^- \cup [\delta_A^- \times B - \delta_A^- \times \delta_B^-] \\
&= A \times \delta_B^- \cup [\delta_A^- \times B - \delta_A^- \times \delta_B^-]
\end{aligned}
$$

  Similarly, we can prove that if A and B do not contain duplicates, then,

$$\delta^+(A \times B) = \delta_A^+ \times (B - \delta_B^-) \cup (A - \delta_A^-) \times \delta_B^+ \cup \delta_A^+ \times \delta_B^+$$

  Note the similarity between these two expressions and the ones used for the over-estimations of the changes for the cartesian product.

- $\sigma_\theta(A)$: No further simplification is possible because the equations for δ^- and δ^+ are already in the simplest form that they can be.

- $\pi_X(A)$: If the input relation to the projection does not contain duplicates, the output relation may or may not contain duplicates. If the output relation does not contain duplicates (because the projection retains at least one key from the input relation), then $\pi_X(\delta_A^-) \cap \pi_X(\delta_A^+) = \emptyset$, and,

$$\delta^-(\pi_X(A)) = \pi_X(\delta_A^-) - \pi_X(\delta_A^+) = \pi_X(\delta_A^-)$$
$$\delta^+(\pi_X(A)) = \pi_X(\delta_A^+) - \pi_X(\delta_A^-) = \pi_X(\delta_A^+)$$

- $A - B$: If both input relations do not contain duplicates, in order to find the incremental deletions from the set difference, we must look at two situations. The first is whether deleted tuples from A were in the set difference before (or equivalently whether they were not in B before). The second is whether tuples inserted into B were in the set difference before (or equivalently whether they were in A before). That is,

$$\delta^-(A - B) = (\delta_A^- - B) \cup (\delta_B^+ \cap A)$$

Similarly, we can argue that insertions into A result in insertions into the set difference as long as these insertions are not in the new value of B. Also, deletions from B result in insertions into the set difference as long as these deleted tuples also are in the new value of A.

$$\delta^+(A - B) = (\delta_A^+ - B^v) \cup (\delta_B^- \cap A^v)$$

- $A \cup B$: No simplification is possible for union because even if the inputs A and B do not contain duplicates, there is no guarantee that their union $A \cup B$ does not have duplicates.

- $e(A)$: If the input to the duplicate elimination does not contain duplicates, then duplicate elimination is a redundant operation and, obviously,

$$\delta^-(e(A)) = \delta_A^-$$
$$\delta^+(e(A)) = \delta_A^+$$

4.3 Optimization due to Foreign Keys

Like in the relational model, in our multiset algebra, a foreign key is an attribute (or a combination of attributes) X_A in a database relation A that is required to match values of the designated primary key X_B in some other database relation B, i.e., $\pi^d_{X_A}(A) \subseteq \pi_{X_B}(B)$.

Suppose that database relations A and B are joined and the join condition is a conjunctive condition containing $X_A = X_B$, where X_A is an attribute of A and X_B is the primary key attribute of B[2]. If there is a foreign key reference from X_A to B, tuples inserted into B do not join with tuples from the old value of A (before A is updated). This is because the X_A values in A before the update already appear in the domain of

[2]Of course, X_A and X_B may be sets of attributes.

X_B in B before the update. Since B has at least one key, if $t \in \delta_B^+$, then $t \notin B$. Then, the following equivalences can easily be proven (θ stands for a join condition of the form discussed above):

$$\begin{array}{lcl} A \bowtie_\theta \delta_B^+ & = & \emptyset \\ \delta_A^- \bowtie_\theta \delta_B^+ & = & \emptyset \\ (A - \delta_A^-) \bowtie_\theta \delta_B^+ & = & \emptyset \\ A^v \bowtie_\theta \delta_B^+ & = & \delta_A^+ \bowtie_\theta \delta_B^+ \end{array}$$

Also, tuples deleted from B either do not join with the value of A before the update, or they join with tuples deleted from A, or else the foreign key constraint would not be satisfied after the update[3]. For the same reason, tuples deleted from B do not join with tuples inserted into A. Therefore, the following equivalences also hold:

$$\begin{array}{lcl} \delta_A^+ \bowtie_\theta \delta_B^- & = & \emptyset \\ \delta_A^+ \bowtie_\theta (B - \delta_B^-) & = & \delta_A^+ \bowtie_\theta B \\ A \bowtie_\theta \delta_B^- & = & \delta_A^- \bowtie_\theta \delta_B^- \\ A^v \bowtie_\theta \delta_B^- & = & \delta_A^- \bowtie_\theta \delta_B^- \end{array}$$

These simplifications allow a database optimizer to transform a change propagation or incremental query into a simpler one that does not need to access database relations as many times and, thus, may be more efficient to evaluate.

4.4 Example

Let us assume the following database relations:

P : PART(P_PARTKEY, ...).

S : SUPPLIER(S_SUPPKEY, ...).

A : PARTSUPP(PS_PARTKEY, PS_SUPPKEY, ...).

[3]The constraint may be violated before the transaction commits, but we assume that view maintenance occurs at the commit point of a transaction when the constraints are known to be satisfied.

The relation P records information about parts; the relation S records information about suppliers; and the relation A relates suppliers with the part that each one supplies. For referential integrity, all the values appearing in the PS_PARTKEY of A must appear in the P_PARTKEY of P and all the values appearing in the PS_SUPPKEY of A must appear in the S_SUPPKEY of S, so that all the parts supplied by a supplier are valid parts, and all the suppliers supplying parts are valid suppliers.

Let us assume that we have a view V defined using the following SQL query

```
select *
from P, S, A
where A.PS_PARTKEY = P.P_PARTKEY and
        A.PS_SUPPKEY = S.S_SUPPKEY
```

Equivalently, we can specify the same view as $V = P \bowtie S \bowtie A$ (ignoring join arguments). Suppose now that each of the P, S and A relations lose a number of tuples specified by the system-defined relations δ_P^-, δ_S^- and δ_A^-, respectively. Let δ_{PA}^- be the deletions from the join $P \bowtie A$ (as if this join were materialized) and δ_V^- the deletions from V. One way to compute the deletions from V is to find the deletions to $P \bowtie A$ and propagate them to V, i.e.,

$$\begin{aligned}\delta_V^- &= \delta_{PA}^- \bowtie S \;\cup\; P \bowtie A \bowtie \delta_S^- \;-\; \delta_{PA}^- \bowtie \delta_S^-, \quad \text{where}\\ \delta_{PA}^- &= \delta_P^- \bowtie A \;\cup\; P \bowtie \delta_A^- \;-\; \delta_P^- \bowtie \delta_A^-.\end{aligned}$$

Tuples that are deleted from A can only join with tuples deleted from P or tuples deleted from S because, otherwise, the foreign key references would not be satisfied after the database update. Thus, we can use the following equivalences to simplify the above equations

$$A \bowtie \delta_P^- = \delta_A^- \bowtie \delta_P^-$$
$$A \bowtie \delta_S^- = \delta_A^- \bowtie \delta_S^-$$

We can rewrite δ_{PA}^- as

$$\delta_{PA}^- = P \bowtie \delta_A^-$$

and we can simplify the change propagation expression as

$$
\begin{aligned}
\delta_V^- &= \delta_{PA}^- \bowtie S \;\cup\; P \bowtie A \bowtie \delta_S^- \;-\; \delta_{PA}^- \bowtie \delta_S^- \\
&= \delta_{PA}^- \bowtie S \;\cup\; P \bowtie \delta_A^- \bowtie \delta_S^- \;-\; \delta_{PA}^- \bowtie \delta_S^- \\
&= P \bowtie \delta_A^- \bowtie S \;\cup\; P \bowtie \delta_A^- \bowtie \delta_S^- \;-\; P \bowtie \delta_A^- \bowtie \delta_S^- \\
&= P \bowtie \delta_A^- \bowtie S
\end{aligned}
$$

Thus, using this optimization, we were able to reduce accesses to the database relations from five to two; accesses to the delta relations from eight to one; and, the total number of joins from seven to two. Consequently, we increased the likelihood that the performance of the change propagation query would be very good.

This concludes our presentation of all proposed optimizations for simplification of change propagation and incremental expressions.

Chapter 5

The RHODES Database Optimizer

In this chapter we describe the design and implementation of the RHODES query optimizer. We discuss the extensions to RHODES to support the optimization of change propagation and incremental expressions. We also introduce the visual browser that accompanies RHODES.

5.1 Query Optimization

A query expressed in a high level language is parsed by the DBMS to produce an intermediate form of the query known as the *query parse tree*. Before any further processing, the query parse tree is validated, so that all of the relation and attribute names appearing in it exist in the DBMS. Figure 5.1 outlines all of the different phases involved in executing a high level query.

After the parsing and validation of the high-level query, the query optimizer of the DBMS examines the query parse tree in order to find an efficient way to implement it. The optimizer uses algebraic transformation rules to transform the query parse tree into one or more equivalent parse trees, that produce the same result as the original one but may give better performance [EN94]. The order in which the algebraic transformations are applied and the choice of which equivalent forms are further examined are heuristics. After pruning the space of equivalent parse trees, the optimizer determines the least expensive algorithms to implement each operator in the chosen parse tree(s). This optimization phase is usually cost-based, since the optimizer uses statistical information stored in a "mini-database", called the *system catalog*, to estimate the cost of choosing different algorithms and decide which choice yields the cheapest *execution plan*.

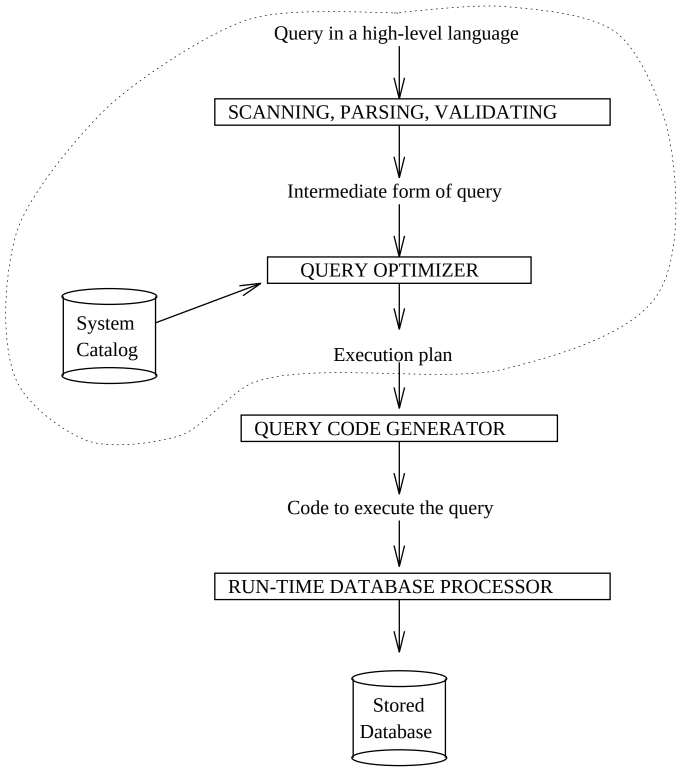


Figure 5.1: The different phases in executing a high level query

As follows from the discussion above, the execution plan generated by the optimizer is not machine executable code but, instead, an intermediate form from which code can be generated. If the optimizer can identify common subexpressions, the execution plan is a directed acyclic graph, otherwise it is a tree. The query execution plan is traversed by a machine-specific component of the DBMS that generates the code. The generated code is then executed (immediately or not) by the run-time processor of the *DBMS* which is the operating system of the *DBMS*: it is responsible for transferring memory blocks to and from disk, buffering, scheduling, and so on.

5.2 The Volcano Optimizer Generator

In this chapter, we present the design and implementation of the RHODES relational database query optimizer built using the Volcano Optimizer Generator [Gra94, GM93a].

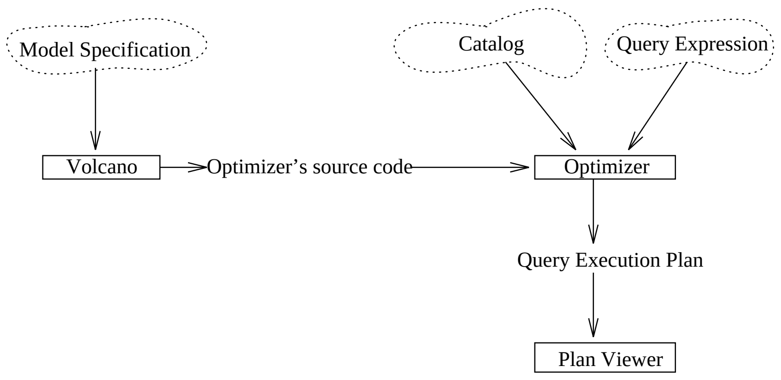


Figure 5.2: Using the Volcano optimizer generator

Figure 5.2 shows how Volcano is used to generate RHODES. Input to the Volcano Generator is a *model specification* of what the intended functionality of the generated optimizer should be. The model specifies what query expressions are being optimized, what algorithms are available to the DBMS for execution, what cost is being minimized when searching for the cheapest execution plan, and so on. The output from the Volcano Generator is the optimizer's source code which is compiled to produce the optimizer.

There are two inputs to RHODES: the *system catalog* and the *query expression* to be optimized. The catalog contains all statistical information necessary for plan cost estimation. The query is an expression (parse tree) over the algebra of logical operators. The output from the optimizer is a *query execution plan*, an expression (dag) over the algebra of algorithms. In our working framework, the query evaluation plan is subsequently fed to a plan visualization tool which allows us to view details of the chosen execution plan, a functionality similar to DB2's `explain` facility [DB2]. We present this tool in more detail later in the chapter.

The Volcano generator provides a search engine to be used by all created optimizers with an exploration and optimization strategy called *directed dynamic programming* [GM93a]. Therefore, RHODES, unlike other optimizers that use dynamic programming, such as the System R optimizer [SAC+94] or the Starburst optimizer [LFL88, Loh88], uses dynamic programming optimization with general algebraic query structures and not just select-project-join queries. Also, these other optimizers generate the space of equivalent expressions *bottom-up*, by creating all expressions that seem useful to create (the *query rewrite* phase) and then estimating all resulting expressions (the *cost-based* optimization). RHODES's search engine creates equivalent expressions and execution plans *top-down* in a goal-oriented way, since it explores and optimizes only those subexpressions that participate in the actual query to be optimized. It also uses the cost model and allows for some pruning of the search space during the query rewrite phase.

5.3 The Catalog

The catalog is a "mini-database" and its function is to store the *schema* and *statistics* of the database that the DBMS maintains. Although several components of the DBMS use the catalog, it is the query optimizer whose operation is interwoven with the use of the catalog, especially when the optimizer estimates the costs of different query execution strategies.

The catalog used by RHODES specifies the following:

- For each relation, the catalog contains the name, arity and cardinality of the relation and a list of the relation's attributes. Each relation must also have a (unique) primary key, i.e., one or more attributes that uniquely determine any tuple within the relation.

- For each attribute in a relation, the catalog contains the name, type, and size, in bytes, of the attribute. There are two data types currently supported by the presented optimizer: `string` and `integer`. The `string` type may use any predefined number of bytes to hold the string value, while the `integer` type requires four bytes.

  If the attribute is an ordering attribute, if it is part of the primary key, or if it has an index defined on it, this fact is also recorded in the catalog. There are three alternatives for index specification: 1) a *primary index* can be defined on an ordering key attribute of the relation[1]; 2) a *clustering index* can be defined on

[1]The key must be a single-attribute key for any index to be defined on it.

an ordering non-key attribute; and, 3) a *secondary index* can be defined on any non-key attribute, whether ordering or not.

Other important information about attributes is the number of distinct values appearing in the domain of the attribute and, if the attribute is of type `integer`, the minimum and maximum value in the domain.

- Foreign key references from attributes of one relation to the key attributes of other relations are also recorded in the catalog.

- Finally, the catalog contains the name and definition of all user-defined materialized views.

5.4 Model Specification

The *model specification* describes what the intended behavior of an optimizer generated by Volcano is. The specification is semi-declarative: some parts of it are provided using definitions and rules and some parts are provided using C code. In this section, we outline the components of the model specification. In the following sections we present each component in more detail.

To understand the outline of the model specification, it is important to know that there is a distinct separation between the logical and the physical view of a DBMS's functionality. The input to the optimizer is an expression in the *logical algebra* of the DBMS. The logical algebra is either the DBMS's query language or some convenient intermediate representation of it. The output of the optimizer, which is a plan to evaluate the logical expression, is an expression in the *physical algebra* of the DBMS. The physical algebra is the collection of algorithms that the DBMS is capable of executing when evaluating a logical expression.

The specification of the model for our optimizer, as with any optimizer built with Volcano, includes [GM93a]:

1. *The set of logical operators* and a definition of the structure of their arguments. These constitute the operators of the logical algebra of the DBMS. For example,

`JOIN` is a logical operator whose argument is an equality condition, such as $R_1.X = R_2.Y$, where R_1 is a relation with attribute X and R_2 is a relation with attribute Y.

2. *An abstract data type "LOGICAL_PROPERTY"* with associated functions for this type. Each expression in the logical algebra has a set of associated logical properties. Equivalent logical expressions share the same logical properties. For example, a logical property is the arity of the relation that the logical expression represents.

3. For each operator in the logical algebra, a *function to derive the logical properties* of an expression, given this logical operator as the top operator, from the logical properties of the inputs.

4. *Algebraic transformation rules*, possibly with condition and/or application code, used to generate equivalent logical expressions in the logical algebra. For example, commuting a select with a join is expressed as a transformation rule.

5. *The set of physical operators (algorithms and enforcers)* implementing the logical operators[2] and a definition of the structure of their arguments. These constitute the operators of the physical algebra of the DBMS. For example, `MERGE` is an algorithm whose argument is an equality condition, such as $R_1.X = R_2.Y$.

6. *Implementation rules* for logical operators describing which algorithm implements each logical operator. For example, the fact that `MERGE` implements `JOIN` is expressed as an implementation rule.

7. *An abstract data type "PHYSICAL_PROPERTY"* with associated functions for this type. Each expression in the physical algebra has a set of associated physical properties. Equivalent physical expressions computing the same relation do not necessarily share the same physical properties. For example, a physical property is the estimated cardinality of the relation that the physical expression computes.

[2] An enforcer does not really implement a logical operator but, instead, is used in conjunction with physical algorithms to guarantee certain properties in the result.

Another physical property is the name of the attribute(s) on which the computed relation is sorted on.

8. For each operator in the physical algebra, a *function to derive the physical properties* of an expression, given this physical operator as the top operator, from the physical properties of the inputs.

9. *An applicability* function for each operator in the physical algebra which determines whether the operator can be used to implement a given logical operator provided that a set of requested physical properties must be present in the output.

10. *A function for each operator in the physical algebra* to determine what physical properties are required from the operator's inputs. For example, this function for `MERGE` specifies that both inputs to the `MERGE` must be sorted.

11. *An abstract data type "COST"* with associated functions for this type.

12. *A cost function* for each operator in the physical algebra.

5.5 Logical Operators

The set of logical operators in the logical algebra of RHODES is shown in the next table. These are all the operators defined in Chapter 3, except that there is no operator corresponding to aggregation.

The table contains the name of each operator, the number of inputs it accepts and its standard abbreviating symbol. These logical operators were chosen so that the logical algebra of the optimizer would be consistent with SQL, as was discussed in Chapter 3.

Except for `PROJECT`, `PROJECT_D`, `GET`, `SELECT` and `JOIN`, the logical operators do not have operator arguments. The argument to `PROJECT` and `PROJECT_D` is a list of attribute names. The `GET` operator is used to retrieve a stored relation. The name of the relation is given as an argument to `GET` (which justifies why `GET` accepts zero inputs). The argument to `SELECT` is a list of selection criteria in conjunctive form. Finally, the argument to `JOIN` is an equality condition between an attribute of the first input and an attribute of the second input.

Name	# of Args	Symbol	Name	# of Args	Symbol
CARTESIAN	2	$\times$	JOIN	2	$\bowtie$
DIFF	2	$-$	PROJECT	1	π
DIFF_D	2	$-^d$	PROJECT_D	1	π^d
GET	0		SELECT	1	σ
INTERSECT	2	$\cap$	UNION	2	$\cup$
INTERSECT_D	2	$\cap^d$	UNION_D	2	$\cup^d$

Table 5.1: Logical Operators

5.6 Logical Properties and Logical Property Derivation

Each logical expression has an associated set of *logical properties*. These include:

- the *arity* of the relation represented by the expression;
- the *set of logical attributes* of that relation (each logical attribute having a name, a type, and, perhaps, a foreign key reference);
- the *set of keys* for the relation (each key being a set of one or more attribute names); and,
- the *set of bound attributes*. An attribute of a relation is strongly bound, if all the relation's tuples contain the same value for that attribute[3].

The derivation of logical properties for an expression, given any logical operator as the expression's top operator, from the logical properties of the inputs is rather simple. Key derivations follow the algorithm for generating the keys for query expressions presented in Chapter 4.

[3]Bound attributes are useful because keys can be simplified by removing the bound attributes from them.

5.7 Algebraic Transformation Rules

The goal of a query optimizer is to find the best possible plan to evaluate a given query. However, there are many logical expressions that are semantically equivalent to the one that the user provides. Two expressions are *semantically equivalent*, if and only if, for every possible database instance, the two expressions represent the same relation in the instance. In order to find the best plan, an optimizer must know how to generate equivalent expressions for any expression provided by the user. In this section, we describe these algebraic rules.

Laws involving `JOIN`:

- Join commutativity, i.e., $A \bowtie_{A1=B1} B \;=\; B \bowtie_{B1=A1} A$.
- Join associativity, i.e., $(A \bowtie_{A1=B1} B) \bowtie_{X=C1} C \;=\; A \bowtie_{A1=B1} (B \bowtie_{X=C1} C)$

  This transformation is not applicable if the argument of the join operator to be moved into the subtree, i.e., X, is not an attribute of B.

Laws involving `SELECT`:

- Commuting selects, i.e., $\sigma_{\theta_1}(\sigma_{\theta_2}(A)) = \sigma_{\theta_2}(\sigma_{\theta_1}(A))$.
- Combining a select with a get, i.e., $\sigma_\theta(get(A)) = get_\theta(A)$.
- Cascades of selects (one direction), i.e., $\sigma_{\theta_1}(\sigma_{\theta_2}(A)) = \sigma_{\theta_1 \wedge \theta_2}(A)$.
- Cascades of selects (other direction), i.e., $\sigma_{\theta_1 \wedge \theta_2}(A) = \sigma_{\theta_1}(\sigma_{\theta_2}(A))$.
- Commuting a select with a join, i.e., $\sigma_\theta(A \bowtie B) = \sigma_\theta(A) \bowtie B$. This transformation can only be applied if θ involves only attributes of A.
- Commuting a select with a cartesian product, i.e., $\sigma_\theta(A \times B) = \sigma_\theta(A) \times B$.

  This transformation can be applied only if θ involves only attributes from A.
- Commuting a select with a project or a project distinct, i.e., $\sigma_\theta(\odot(A)) = \odot(\sigma_\theta(A))$, where $\odot \in \{\pi, \pi^d\}$.

- Combining a select and a cartesian product into a join, i.e., $\sigma_\theta(A \times B) = A \bowtie_\theta B$

  This transformation can be applied, only if θ is of the form $A1 = B1$, $A1$ is an attribute of A and $B1$ is an attribute of B.

- Commuting a select with a set operator, i.e., $\sigma_\theta(A \odot B) = \sigma_\theta(A) \odot \sigma_\theta(B)$, where $\odot \in \{-, -^d, \cup, \cup^d, \cap, \cap^d\}$.

Laws involving `CARTESIAN`:

- Cartesian product commutativity, i.e., $A \times B = B \times A$.
- Cartesian product associativity, i.e., $A \times (B \times C) = (A \times B) \times C$.

Laws involving `PROJECT` and `PROJECT_D`:

- Replacing a project distinct with a project, i.e., $\pi^d_X(A) = \pi_X(A)$.

  This transformation can only be applied if X maintains one key from A.

- Cascades of projects with project distinct's, i.e., $\pi_X(\pi_Y(A)) = \pi_X(A)$ and $\pi^d_X(\pi_Y(A)) = \pi^d_X(A)$ and $\pi^d_X(\pi^d_Y(A)) = \pi^d_X(A)$.

  These transformations can only be applied if the list of attributes X is a subset of the list of attributes Y.

- Cascades of a project and a project distinct, i.e., $\pi_X(\pi^d_Y(A)) = \pi_X(A)$.

  This transformation can only be applied if the list of attributes Y maintains a key from A and X is a subset of Y.

- Commuting a project or project distinct with a select, i.e, $\odot_X(\sigma_\theta A) = \sigma_\theta(\odot_X(A))$, for $\odot \in \{\pi, \pi^d\}$.

  This transformation can only be applied if θ involves only attributes that appear in the attribute list X.

- Commuting a project or project distinct with a cartesian product, i.e., $\odot_X(A \times B) = \odot_{X1}(A) \times \odot_{X2}(B)$.

  This transformation can only be applied if $X1 \cup X2 = X$ and $X1$ contains only attributes of A and $X2$ contains only attributes of B.

- $\pi^d_X(A \times B) = \pi^d_X(A)$.

  This transformation can only be applied if X involves attributes of A only.

- Commuting a project or project distinct with a join (special case), i.e.,

  $\odot_X(A \bowtie_{A1=B1} B) = \odot_{X1}(A) \bowtie_{A1=B1} \odot_{X2}(B)$, where $\odot \in \{\pi, \pi^d\}$.

  This transformation can only be applied if X includes both $A1$ and $B1$ (among possibly other attributes) and $X = X1 \cup X2$, where $X1$ involves only attributes of A and $X2$ involves only attributes of B.

- Commuting a project or project distinct with a join (more general case), i.e.,

  $\odot_X(A \bowtie_{A1=B1} B) = \odot_X(\odot_{X1}(A) \bowtie_{A1=B1} \odot_{X2}(B))$, where $\odot \in \{\pi, \pi^d\}$.

  This transformation can be applied only if X does not contain both $A1$ and $B1$. Then, $X1$ contains the attributes of A that appear in X plus $A1$, if $A1$ does not appear in X, and $X2$ contains the attributes of B that appear in X plus $B1$, if $B1$ does not appear in X.

- Commuting a project or a project distinct with a union or union distinct, i.e.,

  $\odot(A \oplus B) = ((\odot(A)) \oplus (\odot(B)))$, where $\odot \in \{\pi, \pi^d\}$ and $\oplus \in \{\cup, \cup^d\}$.

Laws involving set operators:

- Commutativity of set operators, i.e., $A \odot B = B \odot A$, where $\odot \in \{\cup, \cup^d, \cap, \cap^d\}$.
- Associativity of set operators, i.e., $A \odot (B \odot C) = (A \odot B) \odot C$, where $\odot \in \{\cup, \cup^d, \cap, \cap^d\}$.

5.8 Physical Operators, Enforcers and Implementation Rules

In Volcano, a physical operator is either an algorithm that implements one or more logical operators or an algorithm that does not directly implement a logical operator but is, instead, used to deliver a required physical property. The second kind of physical operators are called *enforcers*. There are two enforcers in `RHODES`:

- `SORT` which is used to sort the input relation on a given attribute; and,

- DUPLICATE_ELIMINATION which is used to remove duplicates from the input relation.

The other physical operators and their implementation rules are:

CARTESIAN	→	CARTESIAN_ALGO	GET	→	INDEX_SCAN
DIFF	→	SET_DIFF	GET	→	FILE_SCAN
DIFF_D	→	SET_DIFF_D	GET	→	BINARY_SEARCH
INTERSECT	→	SET_INTERSECT	JOIN	→	MERGE
INTERSECT_D	→	SET_INTERSECT_D	JOIN	→	NESTED_LOOP
UNION	→	SET_UNION	PROJECT	→	PROJECT_ALGO
UNION_D	→	SET_UNION_D	PROJECT_D	→	PROJECT_ALGO
			SELECT	→	SELECT_ALGO

Table 5.2: Physical Operators and Implementation Rules

As we can see, there are logical operators (e.g., JOIN) that are implemented by more that one physical operator (MERGE and NESTED_LOOP). Also, the same physical operator (e.g., PROJECT_ALGO) may implement more than one logical operator (PROJECT and PROJECT_D). It is also possible that a logical operator is not implemented by any physical algorithm at all. In fact, we use this third feature considerably when extending RHODES with the knowledge of how to optimize incremental expressions.

5.9 Physical Properties and Physical Property Derivation

Each expression in the algebra of algorithms has a number of physical properties associated with it. The physical properties of expressions include:

- the cardinality of the relation that the physical expression computes;
- the size, in bytes, of each tuple in the relation;

- the set of physical attributes (each physical attribute having a name, a type, the number of distinct values in the attribute domain, the minimum and maximum value, and the size, in bytes, of the attribute);

- the attribute names the relation is sorted on, if any; and,

- a boolean variable specifying whether the relation contains duplicates or not.

For simplicity of the presentation, we omit the description of physical property derivations. Appendix A contains the derivation of the cardinality of a relation from the cardinality (as well as other information) of the inputs to the operator used to compute the relation. Join selectivities that determine the cardinality of `JOIN` and predicate selectivities that determine the cardinality of `SELECT` are also discussed in Appendix A.

5.10 Applicability Functions and Input Requirements

For each algorithm of RHODES, there is a function that describes whether the algorithm applies for a given logical operator, given certain physical properties that are requested from the result of the algorithm. For example, for the join $A \bowtie_{A1=B1} B$, if the requested properties include sorted-ness on attribute $A1$ of the output relation, the `MERGE` algorithm applies (because `MERGE` delivers the output sorted on A_1). However, if the requested properties include sorted-ness on attribute $A3$ of the join, `MERGE` does not apply[4].

In addition to the applicability functions, for each algorithm, there is a function that specifies the required properties from the inputs of the algorithm. For example, for `MERGE` to implement a given `JOIN`, each input to `MERGE` must be sorted on the corresponding attribute of the join condition.

For simplicity of presentation, we do not describe the applicability and input requirements functions in any further detail.

[4]In this case, the `SORT` enforcer applies, or the `NESTED_LOOP` algorithm may apply if the relation A is already sorted on $A3$.

5.11 Cost Model and Cost Estimation

When optimizing a user's query, RHODES systematically estimates the cost of different execution strategies and chooses the one with the lowest cost estimate. In order to find the least expensive plan, it performs an exhaustive search over all possible equivalent expressions of the query and, for each such expression, all possible implementations of logical operators by physical algorithms, estimating each one in turn. The cost that is being minimized is an abstract data type in Volcano, that we defined, in RHODES, as the expected I/O in executing the query, i.e., the number of block transfers between memory and disk.

The estimation of the cost to apply each of RHODES's algorithms is described in Appendix B.

5.12 Extensions to RHODES

We extended the basic RHODES presented so far with a) view maintenance support, b) view maintenance specific optimization and c) query rewrite using materialized views. Next, we present each one of these extensions in more detail.

5.12.1 View Maintenance Support

We extended RHODES with the ability to decide, for each view, what the best way to evaluate the view's new value is, after some update to the database has occurred. Also, if a view is to be maintained incrementally, RHODES can decide which incremental expression to use for view maintenance. Before presenting the extensions to RHODES to support this, we should explain how we envision the use of RHODES during view maintenance. We perform view maintenance at the commit point of a transaction that updates the database. At this point, the state of the database has not yet changed and the updates to the database are available through system-defined tables (the delta tables). Also, the catalog of the DBMS (or some in-memory portion of it) contains information and statistics about these delta tables. Then,

1. for each view maintained by the system, RHODES decides what the best way to maintain the view is;

2. for each view that is maintained incrementally, the DBMS computes the changes to the view;

3. the DBMS merges these changes with the old materialized view;

4. the database updates are merged in the database;

5. for each view that is not maintained incrementally, the view's expression is re-evaluated; and,

6. the updating transaction commits.

To support optimization of view maintenance, the set of logical operators for RHODES is extended with five new operators:

- `NEW`(V): the `NEW` logical operator takes a query expression V as its argument and returns the value of the expression V under the database that results from incorporating any non-committed updates into the current database. If no updates are recorded in the catalog, the result of `NEW`(V) is the same as the value of V. However, if some updates have been recorded in the catalog, the result of `NEW`(V) is the value that V will have once the updates commit.

- `DELTAMINUS`(V): the `DELTAMINUS` logical operator takes a query expression V as its argument and returns (exactly) the set of tuples that must be deleted from the old value of V (as if V was materialized), when the database changes are merged with the old database.

- `DELTAPLUS`(V): the `DELTAPLUS` logical operator takes a query expression V as its argument and returns (exactly) the set of tuples to be inserted into the old value of V (as if V was materialized), when the database changes are merged with the old database.

- `OVERMINUS`(V): the `OVERMINUS` logical operator takes a query expression V as its argument and returns one over-estimation of the set of tuples to be deleted from the value of V (as if V was materialized), when the database changes are merged with the old database. The over-estimations computed are those defined in Chapter 4.

- `OVERPLUS`(V): the `OVERPLUS` logical operator takes a query expression V as its argument and returns one over-estimation of the set of tuples to be deleted from the value of V (as if V was materialized), when the database changes are merged with the old database. The over-estimations computed are those defined in Chapter 4.

We symbolize each of `DELTAMINUS, DELTAPLUS, OVERMINUS`, and `OVERPLUS` logical operators with $\delta^-, \delta^+, \Delta^-$ and Δ^+, respectively. When supporting these new operators in the optimizer, the optimizer is not only responsible for optimizing incremental and change propagation expressions for view maintenance, but also for generating the change propagation expressions necessary for view maintenance. Supporting these new operators in RHODES does not require any change in the physical algebra of the database system, neither does it require special algorithms or specialized data structures to be built on top of an existing DBMS. In fact, transformation rules are used in RHODES to *expand* the definition of each of the new logical operators. Let us see what these transformation rules look like:

The `NEW` operator:

There are three different ways to compute the new value of a query expression V:

- by re-evaluating V; the transformation rules applied in this case are:

  $$\begin{array}{lll} \texttt{NEW}(V) = \texttt{NEW}(A) \odot \texttt{NEW}(B), & \text{if } V = A \odot B, & \odot \in \{\times, \bowtie, -, -^d, \cup, \cup^d, \cap, \cap^d\} \\ \texttt{NEW}(V) = \odot(\texttt{NEW}(A)), & \text{if } V = \odot(V), & \odot \in \{\texttt{GET}, \pi, \pi^d, \sigma\} \end{array}$$

- by incremental computation with the use of the change propagation expressions of V; the transformation rule applied in this case is:

  $$\texttt{NEW}(V) = [V - \delta^-(V)] \cup \delta^+(V)$$

- and, by incremental computation with the use of the expressions for the over-estimations of the changes of V; the transformation rule applied in this case is:

  $\texttt{NEW}(V) = [V - \Delta^-(V)] \cup \Delta^+(V)$

The `DELTAMINUS` operator

There are two different ways to compute the incremental deletions of a query expression V:

- by using the non-simplified change propagation expression given by the transformation rule:

  $\delta^-(A) = \Delta^-(A) - \Delta^+(A)$

- and, by using the simplified change propagation expression. In this case, there is one transformation rule per logical operator that defines the incremental deletions for the operator. Table 4.1 of Chapter 4 contains all of them.

The `DELTAPLUS` operator

There are two different ways to compute the incremental insertions of a query expression V:

- by using the non-simplified change propagation expression given by the transformation rule:

  $\delta^+(A) = \Delta^+(A) - \Delta^-(A)$

- and, by using the simplified change propagation expression. In this case, there is one transformation rule per logical operator that defines the incremental deletions for the operator. Table 4.2 of Chapter 4 contains all of them.

The `OVERMINUS` operator

There is one transformation rule per logical operator that defines the over-estimation of incremental deletions for the operator. Table 4.3 of Chapter 4 contains these transformations.

The `OVERPLUS` operator

There is one transformation rule per logical operator that defines the over-estimation of incremental insertions for the operator. Table 4.4 of Chapter 4 contains these transformations.

5.12.2 View Maintenance Specific Optimization

In the previous section, we described what transformation rules are applied in order for RHODES to decide, at the time of view maintenance, what the best way to maintain a view is. If a view is to be maintained incrementally, RHODES determines the best incremental query expression to use for view maintenance. However, as we showed in Chapter 4, change propagation and incremental expressions are amenable to a number of simplifications. Each one of the simplifications specified in Chapter 4 is expressed in RHODES as a transformation rule.

In addition to the simplifications that we defined in Chapter 4, there are simplifications due to the fact that certain expressions during view maintenance evaluate to empty. For example, if one database relation A is not updated during a transaction, both its $\delta^-(A)$ and $\delta^+(A)$ are empty. Since change propagation expressions may evaluate to empty, RHODES uses transformation rules governing the empty set. The empty set conforms to the following rules:

- $\odot(\emptyset) = \emptyset$, for each $\odot \in \{\sigma, \pi, \pi^d, \delta^+, \delta^-, \texttt{NEW}\}$;
- $\emptyset \odot A = \emptyset$, for each $\odot \in \{-, -^d, \times, \bowtie\}$;
- $A \odot \emptyset = A$, for each $\odot \in \{\cup, -\}$;
- $A \odot \emptyset = \pi^d_{[\text{all attrs of } A]}(A)$, for each $\odot \in \{\cup^d, -^d\}$; and
- $\delta^-(A) = \emptyset$ (or $\delta^+(A) = \emptyset$), for each database relation A that the catalog does not contain statistics about its deletions (or its insertions).

5.12.3 Query Rewriting using Views

During the optimization of a query, RHODES examines the possibility of using already materialized views, in order to speed-up the execution of the query. This optimization has been recognized recently as one of the promising advantages of materialized views [LMSS95, FRV96]. RHODES, however, is, to the best of our knowledge, the only optimizer that supports it. There are two transformation rules that implement this idea:

- $A \odot B = V$, for each binary operator $\odot \in \{\times, \bowtie -, -^d, \cup, \cup^d, \cap, \cap^d\}$. This transformation rule is valid only if the query expression $A \odot B$ `matches` the definition of view V.

- $\odot(A) = V$, for each unary operator $\odot \in \{\mathtt{GET}, \pi, \pi^d, \sigma\}$. This transformation rule is valid only if the query expression $\odot(A)$ *matches* the definition of view V.

The algorithm for checking if a query expression *expr* matches a view definition v first checks to see if the top operator of v is the same as the top operator of *expr*, and, then, recursively applies the algorithm to each of the inputs of the expression and the view.

Example

Suppose two materialized views have been defined as $v_1 : A \bowtie B$ and $v_2 : B \bowtie C$ and suppose we are interested in evaluating the query expression $A \bowtie B \bowtie C$. During the transformation of this query into its equivalent forms, all possible join orderings are produced[5]. Since the subexpressions $A \bowtie B$ and $B \bowtie C$ match the definitions of the views v_1 and v_2, respectively, there are two additional equivalent forms for the same query: $v_1 \bowtie C$ and $A \bowtie v_2$, and RHODES would examine the possibility of using either of these equivalent forms in order to generate an efficient plan for this three-way join.

Another possible rewriting of the same query that uses only the materialized views is $\pi_{[\text{attrs of } A\bowtie B\bowtie C]}(v_1 \bowtie v_2)$. This equivalent form of the query is *not* examined by the current version of `RHODES`.

[5]For simplicity of the presentation, we omit the join arguments here.

After having presented the design and functionality of the RHODES database query optimizer, we introduce and describe the graphical browser that accompanies it.

5.13 The Browser

We use a general visualization tool [Noi96] to display the output of RHODES graphically. The nodes in the graph of an execution plan visualization correspond to the database relations, the intermediate results and the output relation of the execution plan. The edges in the graph relate a node v with all nodes corresponding to relations necessary to compute the relation of v.

Each node is labelled with the algorithm (physical operator) used to derive the relation of the node. Database relations are accessed using file scans, binary search, or index scans. Other relations are produced by executing one of (other) the physical operators of RHODES. Each node in the visualization is also identified by an icon specifying the type of the node. Figure 5.3 shows all the icons used by the browser and the physical operator(s) to which they correspond.

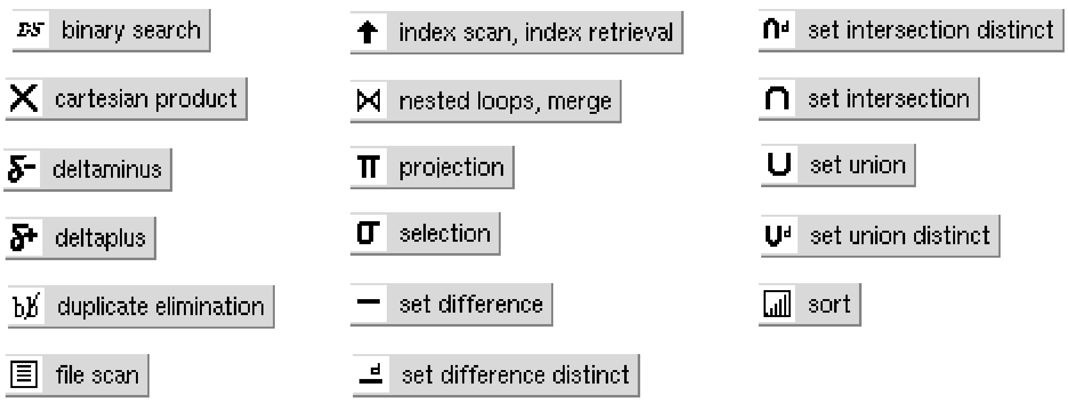


Figure 5.3: The icons of the browser

Textual representations of the generated plans of an optimizer are rather difficult to manage or understand, especially for relatively large plans where the textual description may be dozens or even hundreds of pages long. Our graphical tool provides the database administrator, the anticipated user of the browser, with qualitative and quantitative understanding of the execution plan. It can be used to help understand the output of the optimizer and to facilitate tuning of the database system for better performance. The database administrator could, for example, modify some physical aspect of the database environment and visually monitor its effect in the execution plans for queries of interest.

Dynamic Mapping

Our browser supports the technique of dynamic mapping [Bar92, Noi96, War], which permits the dynamic binding of the elements in the visualization (nodes and edges) into visual properties. We use this technique to dynamically map the relative size of each node into the estimated cardinality of the relation corresponding to the node. Nodes that correspond to large relations, thus, appear larger in the visualization. We also map the color of each node into the estimated cost to compute the relation of the node. Nodes that are expensive to compute, thus, appear more red (hot) in the visualization. It has been argued elsewhere[6] that our ability to perceive is greatly facilitated by the use of such graphic properties as color and size.

Dynamic Querying and Manipulation

Our browser supports a limited form of querying, so that the database administrator can better understand the output of the optimizer by selectively paying attention to only parts of it. To support dynamic querying, each node being visualized has a number of associated numeric attributes. These are:

- the estimated cardinality of the relation of the node;
- the estimated size, in blocks, of that relation;
- the estimated cost to compute the relation of the node from the relations of the inputs (the node cost);
- the total cost to compute the relation of the node (the subtree cost);

[6] See Noik's thesis [Noi96] for references.

- the size, in bytes, of records in the relation; and,
- a boolean attribute describing whether the relation contains duplicates or not.

With each numeric attribute there is an associated histogram representing the distribution of values for this attribute. The range of these values is divided into a number of equal subranges. Each subrange corresponds to a histogram bar, the height of which is proportional to the number of nodes for which the value of the corresponding attribute lies in the associated subrange. Clicking over a node in the graph results in the highlighting of the histogram bars corresponding to each numeric attribute for that node. The actual values for these attributes appear at the bottom of the histogram. Brushing over the histogram bars results in highlighting of the nodes in the graph having a value for the corresponding numeric attribute within the subrange that the histogram represents. Clicking over a histogram results in hiding (or showing) the corresponding nodes in the visualization.

These techniques have also been discussed elsewhere [Noi96, Shn83, Shn94].

5.13.1 Visual Explain Facility of DB2

The Visual Explain facility of DB2 provides a functionality similar to that of our browser [DB2]. However, the visual explain facility does not provide any form of dynamic querying of the visualized output or any manipulation of the visualization. Color in Visual Explain has a predefined meaning: each node of a certain type has an associated (configurable) color. However, color, as we use it in our browser, could be mapped to a number of different attributes dynamically (the most common of which is the cost of the node but it could be the type of the node).

5.13.2 Examples of Plans Generated by RHODES

Join algorithms and index selection

Let us assume the following database relations, as in Section 4.4:

$PART(P\_PARTKEY, P\_NAME, P\_RETAILPRICE, \ldots)$
$PARTSUPP(PS\_PARTKEY, PS\_SUPPKEY, PS\_SUPPLYCOST, \ldots)$
$SUPPLIER(S\_SUPPKEY, \ldots)$

The relation *PART* contains information about parts. A portion of this information is the name and retail price for the part. The relation *PARTSUPP* contains information about supplied parts. Such information relates suppliers with the parts they supply as well as the cost of each part from each supplier. Finally, the relation *SUPPLIER* contains information about each supplier.

Now, suppose we are interested in evaluating the following SQL query:

`select` *P_NAME, PS_SUPPLYCOST*
`from` *PART, PARTSUPP*
`where` $P\_PARTKEY = PS\_PARTKEY$ `and`
$PS\_SUPPLYCOST < 100$ `and`
$P\_RETAILPRICE > 120$

This query requests the name and prices of all products that are being sold at a price greater than 120 but for which the retail price is not more than 120. Suppose that a secondary dense index has been defined on the *PS_SUPPLYCOST* attribute of the *PARTSUPP* relation. This index can be used to retrieve the tuples from *PARTSUPP* with the specified supply cost. Also, suppose that no index has been defined on the *P_RETAILCOST* attribute of the *PART* relation. Thus, this relation must be scanned entirely before all the tuples that have the specified retail price are found. The `from` clause of the query, together with the condition $P\_PARTKEY = PS\_PARTKEY$, specify a join on the two relations. RHODES examines all different join orderings (there are two) and all different implementations of the join, in order to decide that, in this case, a merge join algorithm must be used. The merge algorithm requires its inputs to be sorted before it can be applied. The sort enforcer guarantees that the two relations to be joined are sorted before being joined.

Figure 5.4 shows the visualized optimized execution plan generated by RHODES for this query.

Join orderings

In the same database, let us now examine the query:

> `select *`
> `from` *PART, PARTSUPP, SUPPLIER*
> `where` *P_PARTKEY* = *PS_PARTKEY* `and` *PS_SUPPKEY* = *S_SUPPKEY*

The `from` clause of the query, together with the selection conditions, specify a 3-way join among the three relations. The order of executing a series of joins may have a significant impact on the performance of queries. RHODES examines all join orderings and chooses the one with the lowest cost estimate. For this example, the generated join ordering appears in Figure 5.5. In this example, we have also dynamically mapped the size of each node to the estimated cardinality of the relation of the node and its color to the estimated cost to compute the relation of the node. As we can see from the picture, the relation *PARTSUPP* is bigger than either of the other relations. One merge join and one nested-loops join are chosen in this execution plan. The merge join requires both of its inputs to be sorted. A secondary dense index defined on the *P_PARTKEY* of the *PART* relation is used to produce this relation sorted. The other relation is the result of evaluating the join of *PARTSUPP* and *SUPPLIER*. There are two different ways to produce this relation as sorted. The first is to compute the join and then use the sort enforcer. The other is to sort the first input of the nested-loops algorithm. If the first input of nested-loops is sorted, the result of the join is also sorted. In this particular example, the size of the result of the join has as many tuples as the *PARTSUPP* relation, and, therefore, it is better to sort the *PARTSUPP* relation instead of the result of its join with *SUPPLIER*.

Reasoning about keys and duplicates

Suppose $A(A_1, A_2, A_3)$ and $B(B_1, B_2)$ are two database relations. Let attribute A_1 be the key of A and let attribute B_1 be the key of B. Suppose now that we are interested in evaluating the following query:

> `select distinct` A_3, B_2
>
> `from` A, B
>
> `where` $A_1 = B_2$ `and` $A_1 < 10$

The generated output of RHODES for this plan is shown in Figure 5.6. As before, the `from` clause of the SQL query, together with the $A_1 = B_2$ predicate condition, results in a join between A and B.

This example demonstrates the use of keys and reasoning about duplicates in RHODES. In the generated plan for this query, RHODES *pushes the selection down* and uses a file scan with the predicate $A_1 < 10$. Since no index is defined on the A_1 attribute of A, scanning the relation is necessary. Also, RHODES *pushes the projection down* and uses two projections before joining A and B, one for each of the join inputs. Pushing the projection inside the join operands has the desirable effect of reducing the sizes of the join operands. Each tuple in a projection is smaller that a tuple in the original relation and therefore many more tuples can fit into one memory page. Thus, the size of the projection, in pages, is reduced. The cost estimation used by RHODES is such that the size, in pages, of a join's inputs is the most dominant factor in the estimated cost for the join.

In a similar spirit to pushing projection, RHODES pushes duplicate elimination too, because duplicate elimination reduces the number of tuples in the operands. The projection on the A relation maintains the key of A and therefore no duplicate elimination is necessary in this projection (redundant operator). The projection on the B relation, however, does not maintain the key of B. In this case, duplicate elimination is enforced.

Now consider the operands of the join. The left operand has A_1 as a key. The right operand has B_2 as a key. Therefore, $\{A_1, B_2\}$ is a key for the join. However, because the join condition equates these two attributes the keyset for the join can be simplified to $\{\{A_1\},\{B_2\}\}$. (See Chapter 4 for an algorithm to generate keys.) Either A_1 or B_2 alone is sufficient to uniquely determine each tuple in the output of the join. Therefore, the final projection, the one that maintains A_3, B_2 does not contain duplicates and no duplicate elimination operation is necessary.

View maintenance

Suppose a view V has been defined which must be maintained under deletions (but not insertions).

$$V : \mathit{PART} \bowtie \mathit{PARTSUPP} \bowtie \mathit{SUPPLIER}$$

To maintain the view, RHODES requires to sort the delta relations. External sorting is not used for the database relations. If these relations need to be accessed in a sorted manner, indices defined on them are used, instead. Figure 5.7 shows the RHODES generated plan.

Figure 5.4: Choosing indices and join algorithms

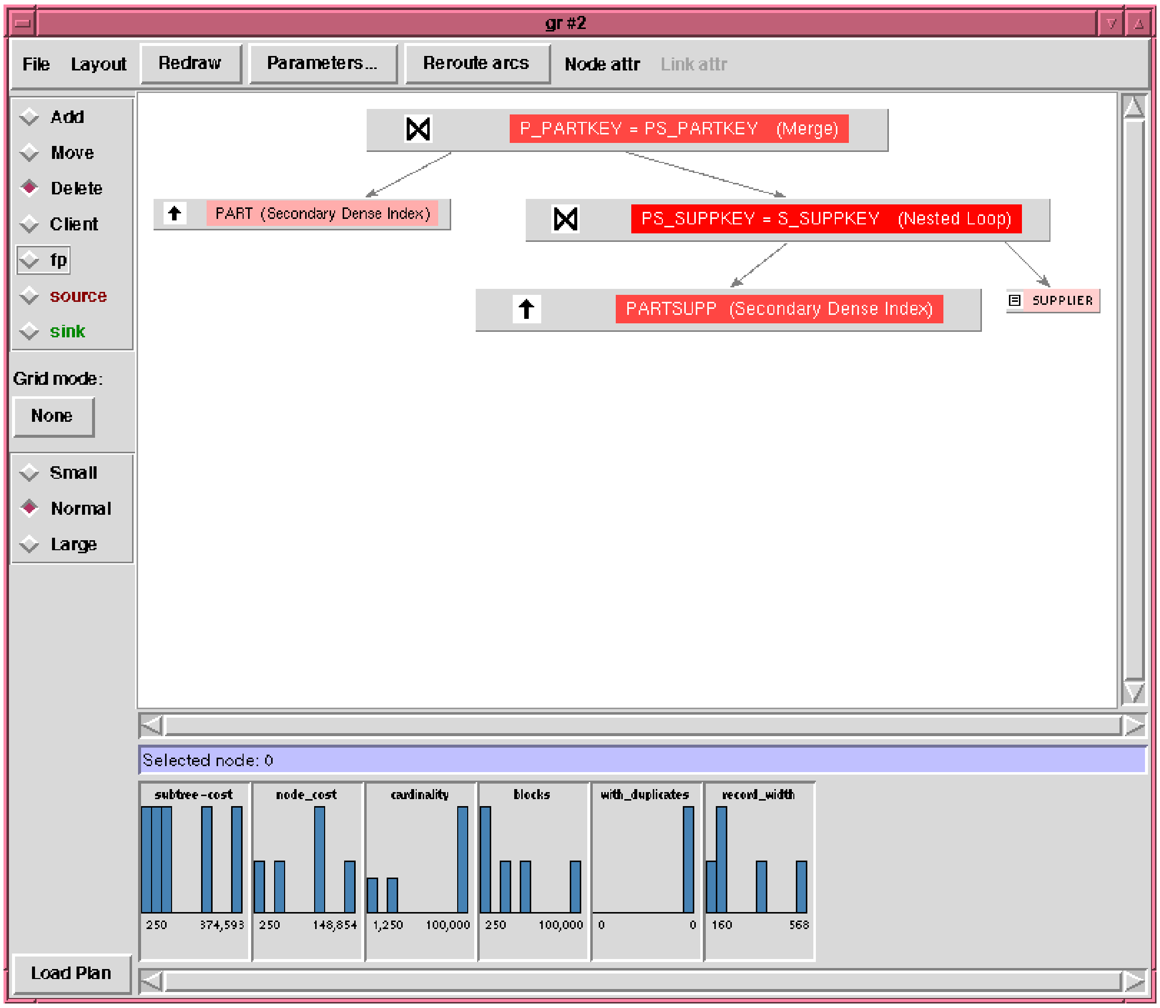


Figure 5.5: Choosing join orderings

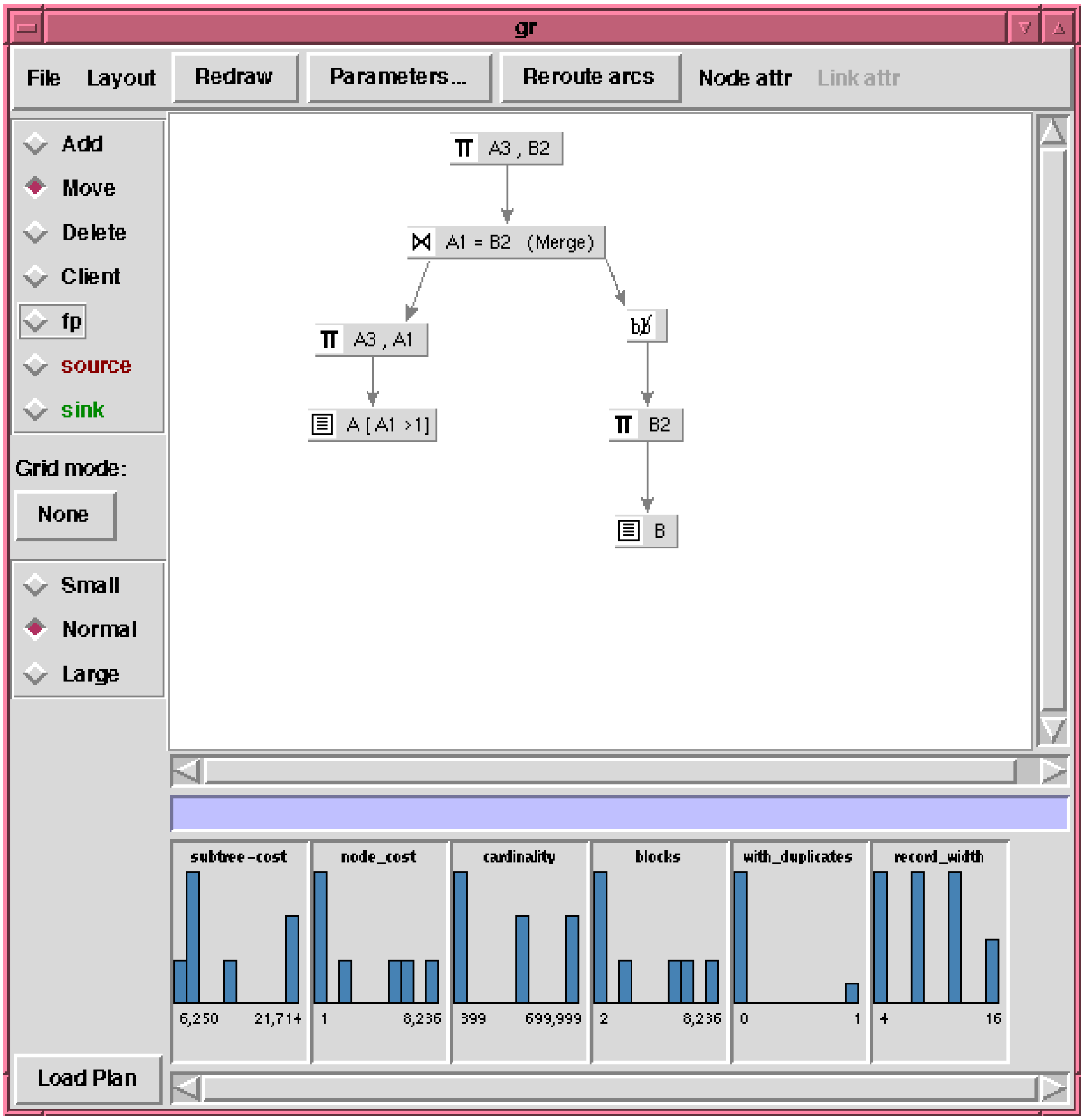


Figure 5.6: Reasoning with keys and duplicates

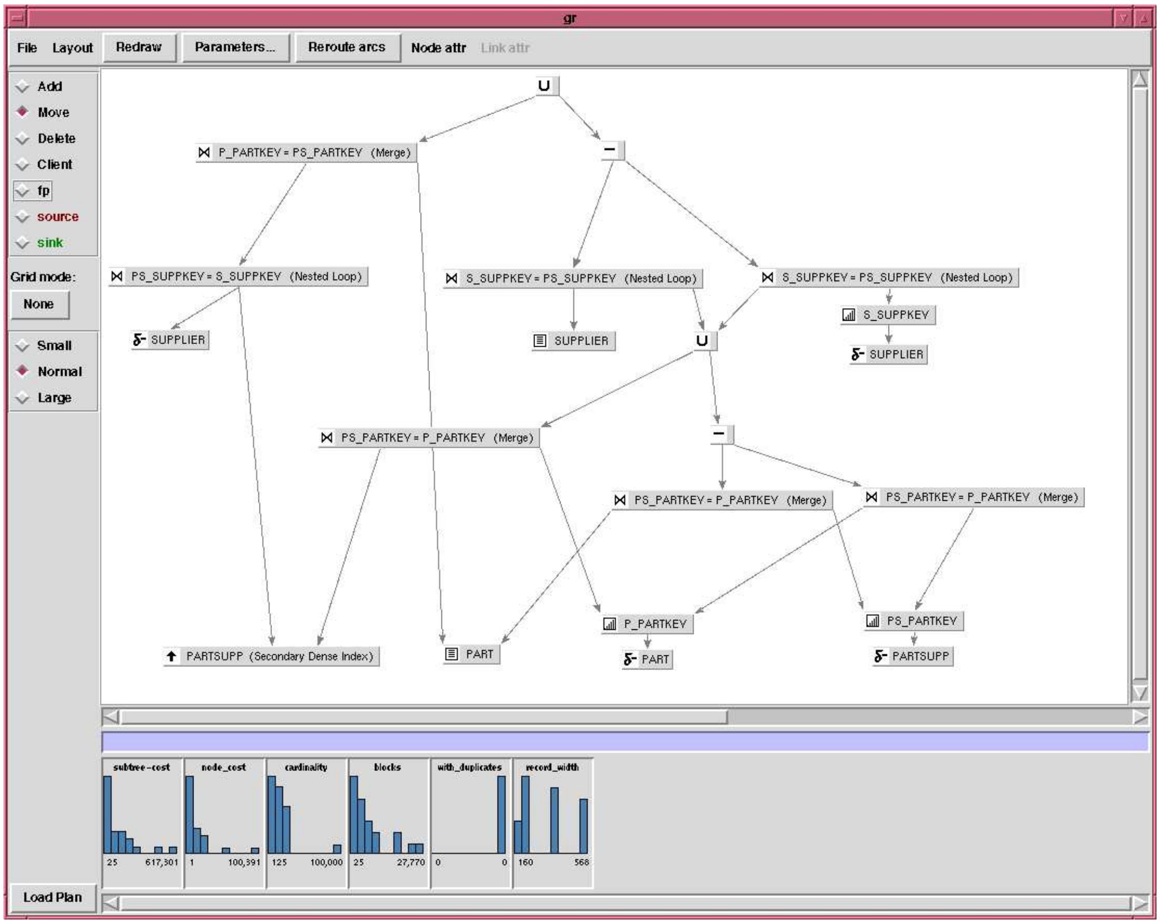


Figure 5.7: Maintaining a view

Chapter 6

Experiments

In this chapter, we present some examples of change propagation queries that provide experimental evidence about the validity of the claims we make in this dissertation.

6.1 Stating the Questions

The questions that we are interested in addressing experimentally are the following:

1. Do different change propagation expressions for defining the incremental changes to a view result in differences in the performance of computing these changes? Are these performance differences big enough to justify our claim that an "intelligent" component of the DBMS is needed to decide among the different choices?

2. Can we demonstrate experimentally that our optimization of change propagation expressions in the presence of key constraints is a reasonable optimization?

3. Can we demonstrate experimentally that our optimization of change propagation expressions in the presence of foreign key references is a reasonable optimization?

4. Can we demonstrate experimentally that the cost of computing incremental changes to a view may be about the same as the cost of view re-computation?

To address questions 1-3 above, we have designed three different sets of experiments, one for each question. These experiments are conducted using the DB2 PE DMBS (DB2 Parallel Edition). To address question 4 we use the results of these same experiments in addition to other experiments conducted using the RHODES optimizer.

6.2 Collecting Data: the Database

All experiments are run on the TPC-D database benchmark relations [TPC95]. In particular, we assume the following schema in the database, as in Section 4.4:

$$
\begin{array}{ll}
\mathtt{P}: & \mathtt{PART(P\_PARTKEY,\ldots)}. \\
\mathtt{PS}: & \mathtt{PARTSUPP(PS\_PARTKEY, PS\_SUPPKEY,\ldots)}. \\
\mathtt{S}: & \mathtt{SUPPLIER(S\_SUPPKEY,\ldots)}.
\end{array}
$$

The relation PART, abbreviated with P, records information about specific parts in a decision support environment. The relation SUPPLIER, abbreviated with S, records information about suppliers of those parts. Finally, the relation PARTSUPP, abbreviated with PS, relates suppliers with the parts that they supply. The distribution of values in the attributes of each relation is uniform.

Relation	Primary Key	Index Type	No. tuples	No. pages
P	P_PARTKEY	P	25000	955
PS	{PS_PARTKEY, PS_SUPPKEY}	S, S	100000	4015
S	S_SUPPKEY	P	1250	61

Table 6.1: Information about the TPC-D relations

We store the three relations using the DB2 PE parallel edition. We have configured the system so that it uses only one node in order to simulate a centralized database. A *node* in a DB2 PE DBMS is one processor in the parallel database. By configuring the system to use only one node, all relations reside in a single memory disk and only one node participates in query evaluation. Thus, the resulting system behaves as a centralized system. We use the TPCD-D database with a scale factor of 0.125. This means that the sizes of each database relation is one eighth of the suggested size. Table 6.1 contains information about some of the relations in the database such as primary keys, cardinality, memory pages and indices. A primary index (denoted in the table with P) is defined on the P_PARTKEY attribute of P and on the S_SUPPKEY attribute of S. A dense secondary index (denoted with S) has been defined on each of the PS_PARTKEY and PS_SUPPKEY of the PS relation.

To answer the questions 1-3 presented above we use this database. Then, we conduct a number of experiments and perform a number of measurements. Each experiment is done independently. This means that before an experiment is conducted, the buffer of the database is cleared from its contents. In this way, the results of each experiment do not depend on the contents of the buffer and the hit-ratio that results from previously cached pages.

On top of this database we define three views:

J1 : select * from P, PS where P_PARTKEY = PS_PARTKEY

J2 : select * from PS, S where PS_SUPPKEY = S_SUPPKEY

J3 : select * from P, PS, S where P_PARTKEY = PS_PARTKEY and PS_SUPPKEY = S_SUPPKEY

In each experiment, we specify an update to the underlying database and we monitor propagation of the incremental changes to these views. We measure the logical and physical I/O necessary to perform change propagation and, sometimes, view evaluation. *Logical I/O* refers to the number of memory pages accessed during computation (independently of whether these pages must be brought from the disk or are already cached in main memory). *Physical I/O* refers to the number of memory pages transferred from the disk into main memory during computation. To get the logical and physical I/O we have used a monitor program that gets "snapshots" of the state of the DBMS. The state contains, among other things, counters that record the activity of the DBMS since the last time a "reset" of the counters was issued.

6.3 Collecting Data: The Experiments

6.3.1 Using Different Incremental Queries

These experiments compare the performance of different queries computing incremental changes. We take different equivalent queries corresponding to computing the incremental changes to a view and, for a number of database updates, we run these queries on DB2.

We, then, measure the results (logical and physical I/O) of each experiment and compare these results.

Experiment 1.1: In this experiment, we delete one tuple from `S`, a portion of `P` ranging from 0.1% to 10% of `P`, and all related `PS` facts and we monitor the propagation of incremental deletions to `J3`.

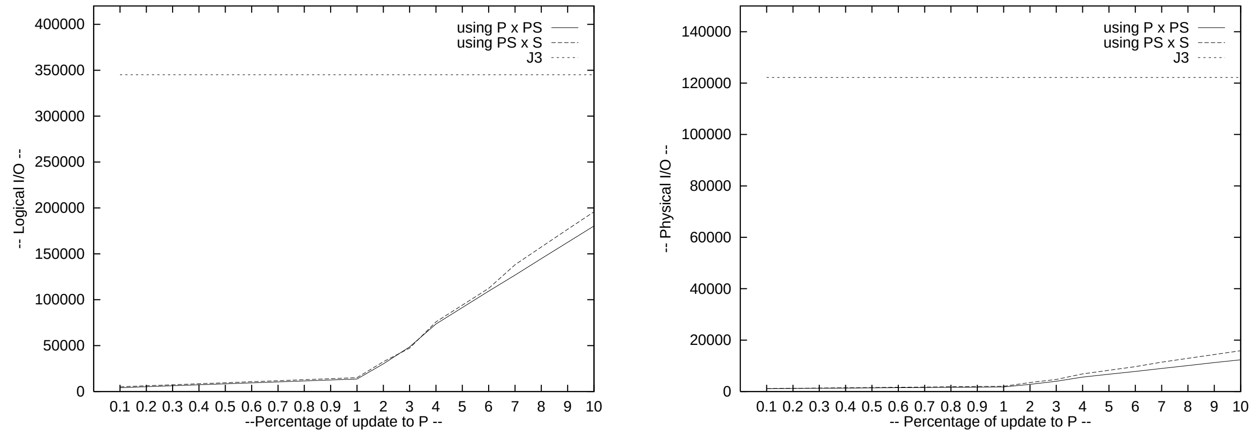


Figure 6.1: Experiment 1.1: Incremental deletions to `J3` with the index on `PS_PARTKEY`

We use two different queries to compute the incremental deletions: the first is to propagate the database deletions through `P` $\bowtie$ `PS` and the second is to propagate the database deletions through `PS` $\bowtie$ `S`. We also compare the results with computing `J3` before the update. Figure 6.1 shows the results of this experiment. As we can see, the performance of the incremental methods is much better that evaluating the view initially. This is especially true when considering the physical I/O, which is the dominating factor affecting performance.

The plan chosen by DB2 to compute `J3` in this experiment is as follows: first the join between `PS` and `S` is performed by scanning the `S` relation and looking-up, using the index on the `PS_SUPPKEY` of `PS`, the corresponding tuples in `PS` (nested loops join). Then, for each tuple in the intermediate join, the corresponding tuples on the `P` relation are found (merge join). As we see, the index on `PS_PARTKEY` of `PS` is not used during the evaluation.

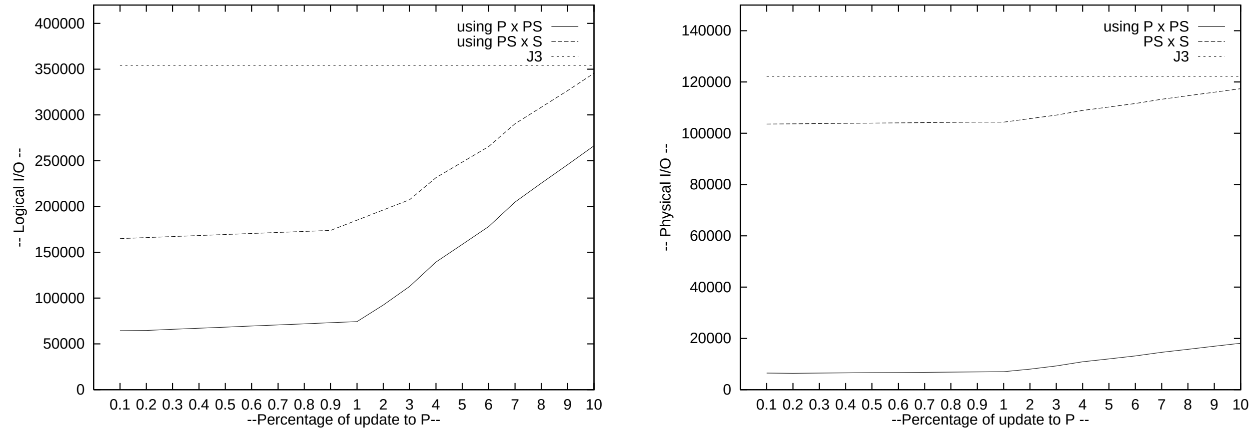


Figure 6.2: Experiment 1.2: Incremental deletions to J3 without the index on PS_PARTKEY

Next, we repeat the same experiment, only in this case, we drop the index on the PS_PARTKEY attribute of PS.

Experiment 1.2: We use the same updates as before and monitor the propagation of incremental deletions to J3 using the two different ways described above. This time, no index exists on the PS_PARTKEY attribute of P. Figure 6.2 shows the results. Note that to compute J3 in this and the previous experiment, the same amount of logical and physical I/O is necessary (because the index we dropped is not used in the evaluation of J3). As we see, the performance difference between choosing to propagate through P $\bowtie$ PS or through PS $\bowtie$ S is rather big. In fact, to compute the incremental changes through P $\bowtie$ PS the time[1] ranges from 54 sec to 3 min, while through PS $\bowtie$ S the time ranges from 26 min to 28 min. Computing J3 requires approximately 34 min.

Experiment 1.3: We add a portion of P and related PS facts as well as one new supplier. We monitor propagation of incremental insertions to J3. Figure 6.3 shows the results. All defined indices are available during evaluation. Note that, in contrast with propagating the deletions, the two methods to propagate insertions have a slight difference in performance.

[1] This refers to real time, not cpu or system time.

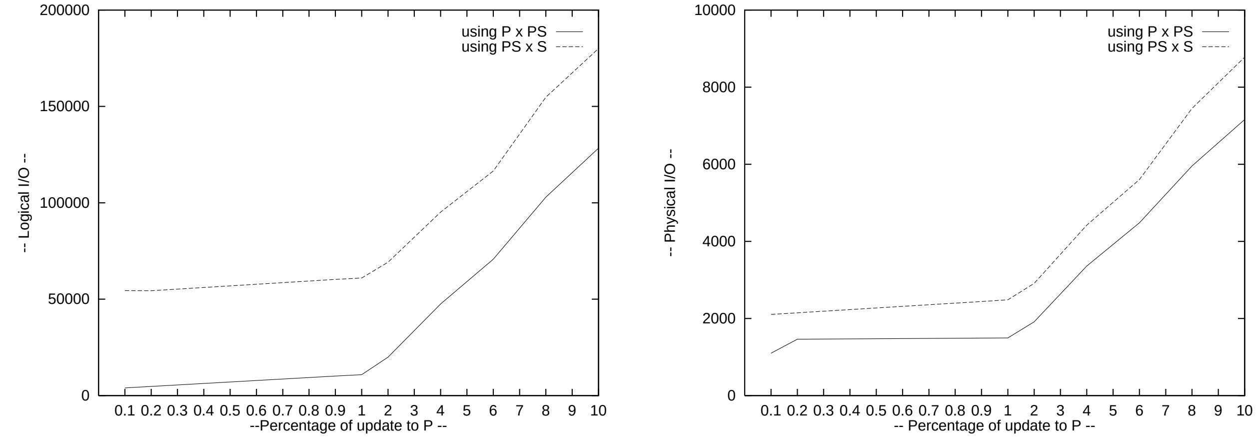


Figure 6.3: Experiment 1.3: Incremental insertions to `J3` (due to insertions only)

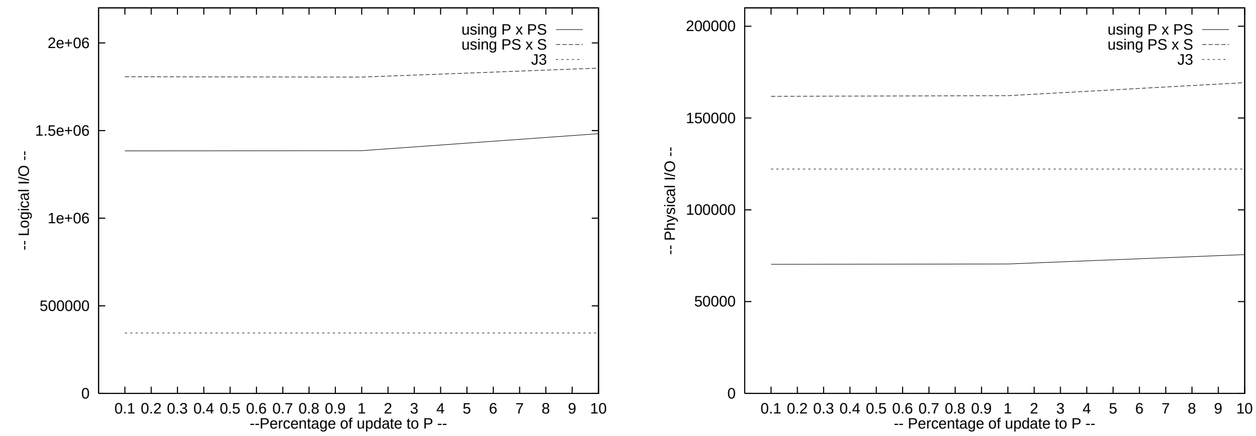


Figure 6.4: Experiment 1.4: Incremental insertions to `J3` (due to insertions and deletions)

Experiment 1.4: We delete a portion of `P`, one tuple from `S` and all related `PS` facts. We also add a portion of `P`, one tuple in `S` and related `PS` facts. We monitor propagation of incremental insertions to `J3`. Figure 6.4 shows the results. As we see, one incremental method is better than view evaluation but the other is not. In fact, the time to compute the incremental insertions through `P` $\bowtie$ `PS` ranges from 12.5 min to 14 min while to compute them through `PS` $\bowtie$ `S` the time ranges from 53 min to 54 min. Note that when considering the logical I/O, view computation seems to outperform both change propagation methods. However, when considering the physical I/O (which is more representative of the actual time a query takes to execute) propagation through PS $\bowtie$ P is, in fact, a lot better.

In this set of experiments, we used optimized queries to propagate incremental changes through P $\bowtie$ PS and through PS $\bowtie$ S. The only optimization that was *not* applied during these experiments is the optimization due to foreign key references. As we see later on, this optimization greatly improves the performance of change propagation.

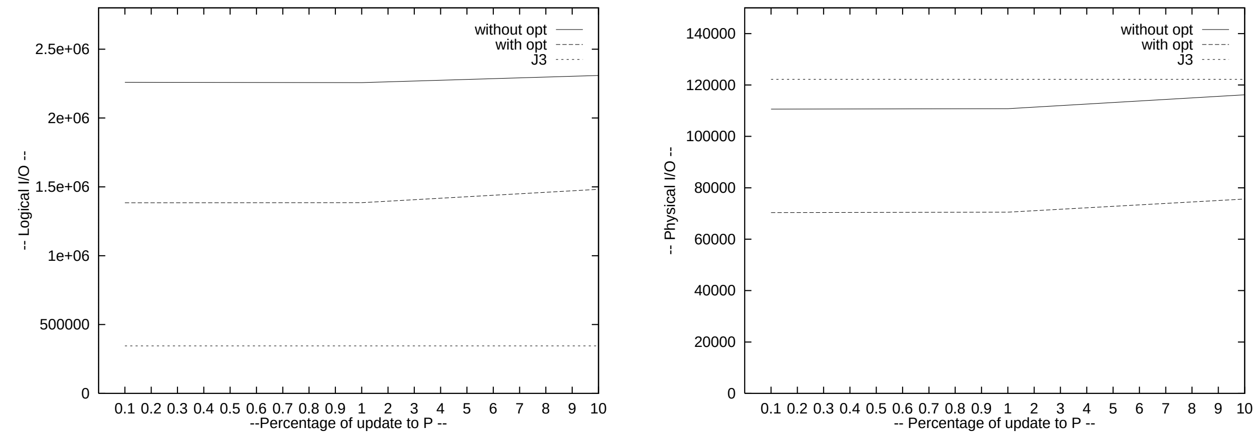


Figure 6.5: Experiment 2.1: Incremental insertions to J3 with and without key optimization

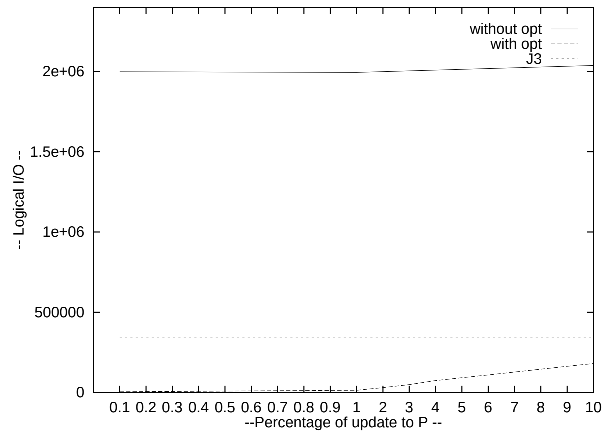
6.3.2 Using the Key Constraint Optimizations

In this section, we present examples where we compare the optimized change propagation queries (where all optimizations except the foreign key reference optimization are active) to the non-optimized change propagation queries (the ones derived by the method of Griffin and Libkin [GL95]). These experiments demonstrate the benefit from using the new key constraint optimizations proposed in the thesis. We take the non-optimized and the optimized change propagation queries and we run these queries on DB2. Then, we measure the logical and physical I/O necessary to compute the incremental changes and we compare the results.

Experiment 2.1: We add and delete a portion of P ranging from 0.1% to 10% of P, we add and delete one tuple from S and we add and delete related PS facts. We monitor propagation of incremental insertions to J3. Figure 6.5 shows the results. We see that the proposed optimization reduces the physical I/O necessary to about half the I/O needed to compute the original join. Also, note how the logical I/O is much larger than the physical I/O.

Experiment 2.2: We use the same update as before but we measure the propagation of incremental deletions to join J3. Figure 6.6 shows the results.

For the above two examples, the optimized queries to maintain the view coincide with the over-estimations of the changes for these views. This also shows that using over-estimations of changes in place of their actual changes has the potential to improve the performance of change propagation.



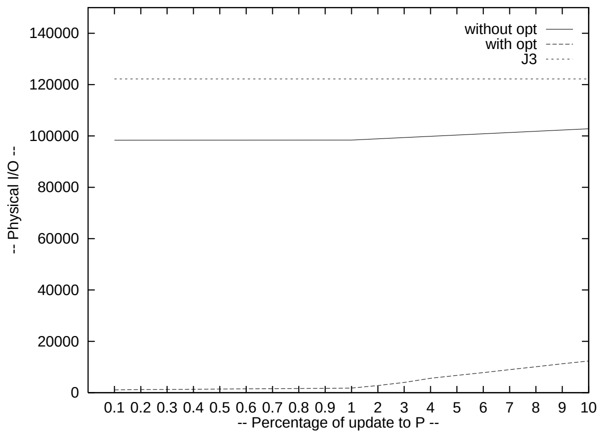
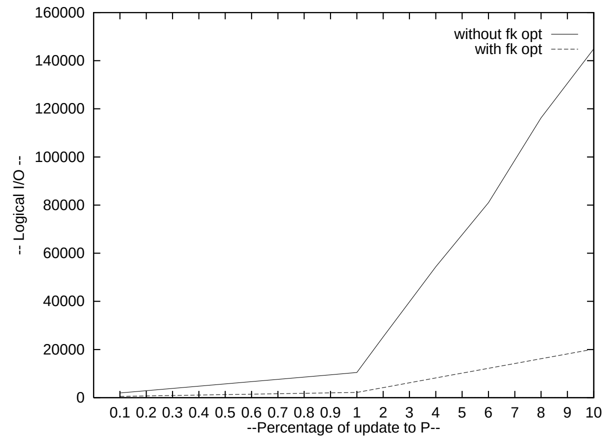


Figure 6.6: Experiment 2.2: Incremental deletions to J3 with and without key optimization

6.3.3 Using the Foreign Key Constraint Optimizations

The experiments presented in this section demonstrate the benefit from using the new foreign key constraint optimizations proposed in the thesis. We take a number of updates and monitor propagating these updates to each of the three defined views J1, J2 and J3. In this database schema, there is a foreign key reference from PS_PARTKEY of PS to P_PARTKEY of P and one foreign key reference from PS_SUPPKEY of PS to S_SUPPKEY of S.

We use two different ways to propagate the updates: one uses all available optimizations except the optimization due to the foreign key references and the other also uses this additional optimization. In both cases, we use the RHODES optimizer to optimize the expressions for computing the incremental updates. First, we use RHODES without declaring the foreign key references to derive an execution plan to propagate the updates. From this plan, we construct an SQL query and execute it in DB2. Then, we define in RHODES the foreign key reference and optimize the change propagation queries again. From the execution plan that RHODES generates, we derive another SQL query which, again, we execute in DB2. We compare the results. In the figures that follow, "without fk opt" means that all optimizations but the one due to the foreign key references are on, while "with fk opt" refers to all optimizations being active.



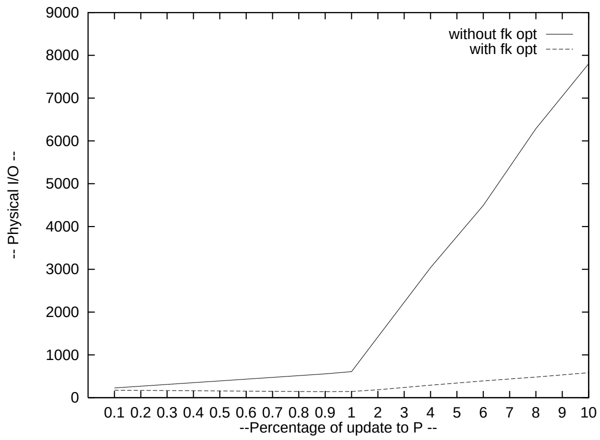


Figure 6.7: Experiment 3.1: Incremental deletions to `J1` with and without foreign key optimization

Experiment 3.1: We delete a portion of `P`, one tuple from `S` and all related `PS` facts and monitor the changes to `J1`. Figure 6.7 shows the results. As we see, the additional optimization greatly improves the performance of the incremental method. The optimized query due to the foreign key is $\delta^{-}(\mathtt{PS}) \bowtie \mathtt{P}$.

Experiment 3.2: We use the same updates as before, only now we look at the propagation of incremental changes to `J2`. Figure 6.8 shows the results. The optimized

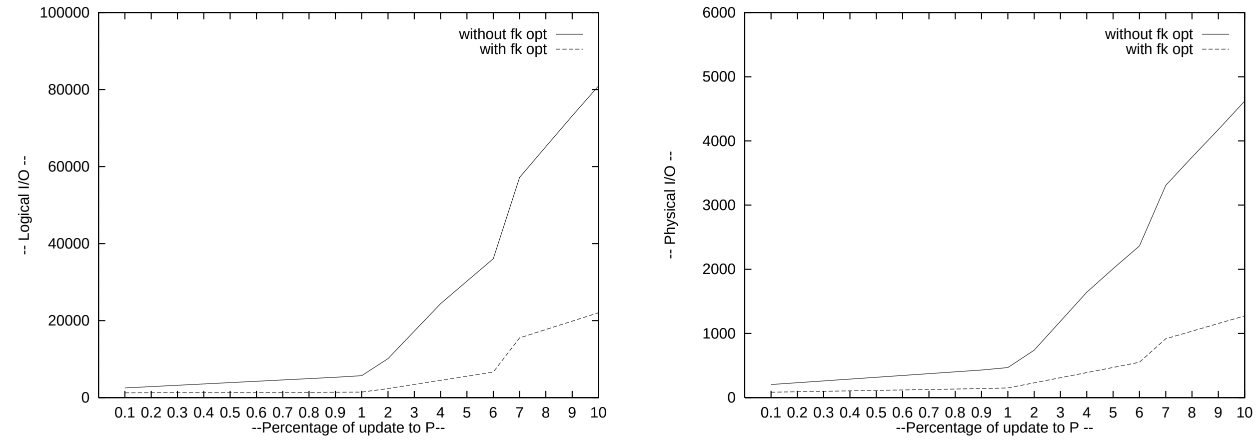


Figure 6.8: Experiment 3.2: Incremental deletions to J2 with and without foreign key optimization

query is $\delta^{-}(\texttt{PS}) \bowtie \texttt{S}$. Note the difference between this experiment and the previous one, where we had a very smooth behavior of both the optimized and the non-optimized method. We were not able to satisfactorily explain the irregularities in the graphs. The execution plan for both queries remain the same throughout the example (for all updates) and the join selectivity between the changes to PS and S is constant. Also, one may be wondering why the cost is so high for propagating changes to this join since only one tuple changes from S (which in this case joins with about 80 tuples from PS). With no index on the changes of PS, scanning the changes to PS is the most important factor that affects the performance. Finally, note that the benefit of using the foreign key optimization in this case is relatively smaller. This is mainly because the non-optimized version for this example takes much less time to execute than the non-optimized version in the previous example.

Experiment 3.3: We use the same updates as before, but we monitor propagation of incremental changes to J3. The optimized query due to the foreign keys is $\texttt{P} \bowtie \delta^{-}(\texttt{PS}) \bowtie \texttt{S}$. Figure 6.9 shows the results. As we see, in this case the plots are smooth again. One can think of these plots as approximately the *sum* of the plots presented for J1 and J2.

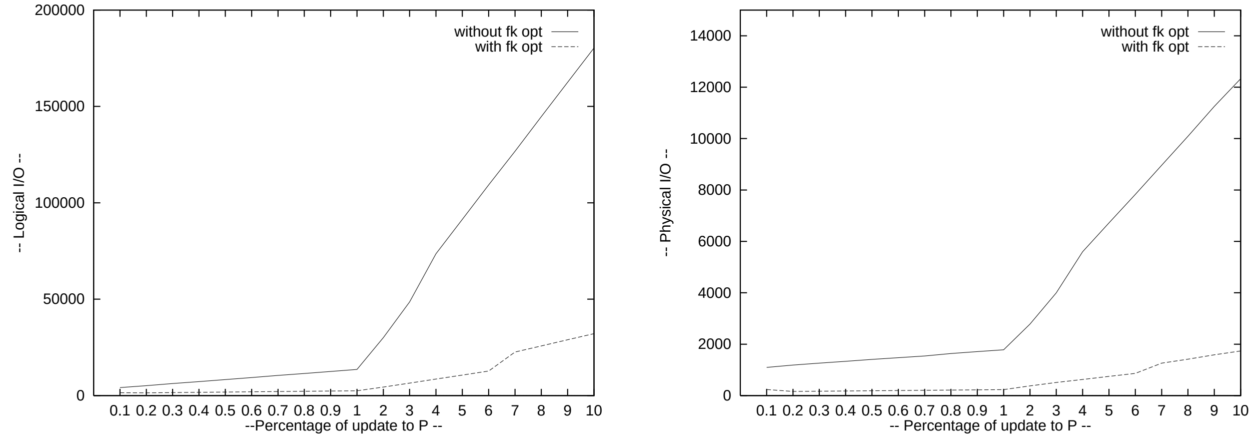


Figure 6.9: Experiment 3.3: Incremental deletions to `J3` with and without foreign key optimization

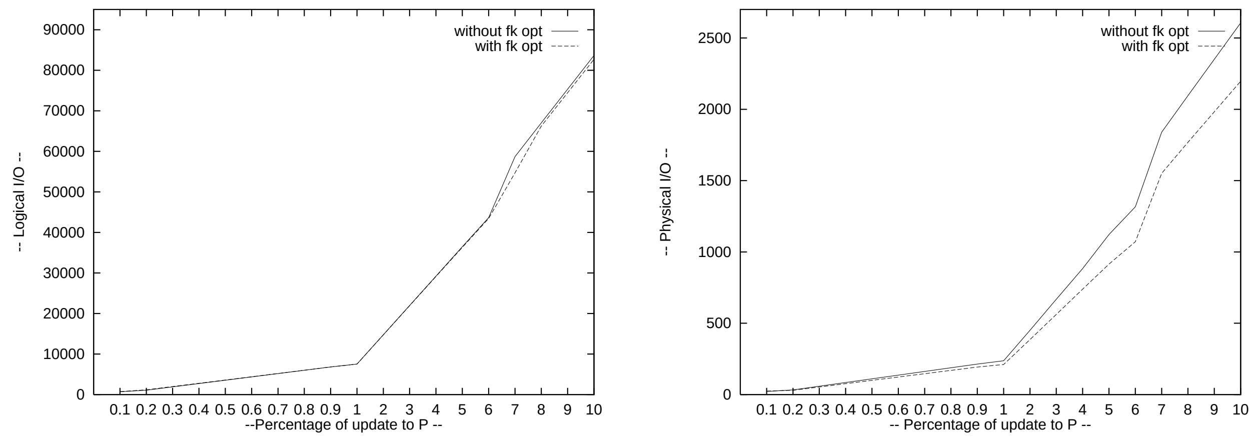


Figure 6.10: Experiment 3.4: Incremental insertions to `J1` when both `P` and `PS` get insertions, with and without foreign key optimization

Experiment 3.4: We add a portion to the `P` relation ranging from 0.1% to 10% of `P` and related related `PS` facts. We monitor propagating the incremental insertions to `J1`. Figure 6.10 shows the results. The non-optimized change propagation query is $\delta^+(\mathtt{P}) \bowtie \mathtt{PS} \cup \mathtt{P} \bowtie \delta^+(\mathtt{PS}) \cup \delta^+(\mathtt{P}) \bowtie \delta^+(\mathtt{PS})$. The optimized query is $\delta^+(\mathtt{PS}) \bowtie \mathtt{P} \cup \delta^+(\mathtt{PS}) \bowtie \delta^+(\mathtt{P})$. As we see the only difference between the two queries is the extra factor of $\delta^+(\mathtt{P}) \bowtie \mathtt{PS}$ which evaluates to empty. The DB2 system was able to understand this by simply accessing the index on PS_PARTKEY of PS without accessing the PS data at all. Thus, the optimized and the non-optimized queries have almost the same run time performance. However, this is because the two relations change by insertions only. Next, we repeat the same experiment, only in this case we allow deletions as well as insertions to the two relations.

Experiment 3.5: We delete a portion to the P relation and add another one. We also delete all related PS tuples and add PS facts. We monitor the incremental insertions into J1. Figure 6.11 shows the results.

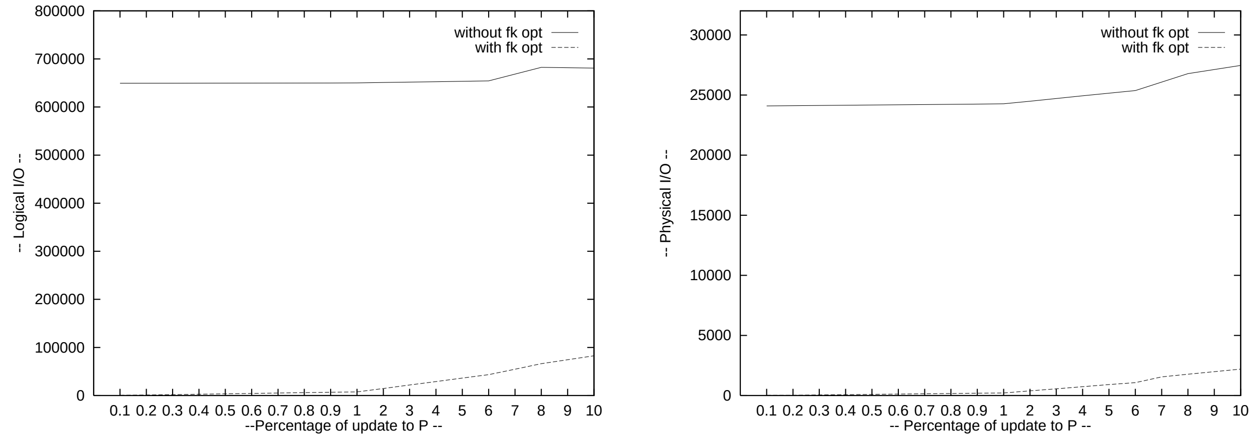


Figure 6.11: Experiment 3.5: Incremental insertions to J1 when both P and PS get insertions and deletions, with and without foreign key optimization

The optimized query for this case is the same as before (Experiment 4). The non-optimized query is $\delta^{+}(\texttt{P}) \bowtie (\texttt{PS}-\delta^{-}(\texttt{PS})) \cup (\texttt{P}-\delta^{-}(\texttt{P})) \bowtie \texttt{PS} \cup \delta^{+}(\texttt{P}) \bowtie \delta^{+}(\texttt{PS})$. As one can see, DB2 was not able to efficiently evaluate the set differences required in this query. The performance difference between the optimized and the non-optimized change propagation queries is very big.

6.4 Other Experiments

All the examples of the previous sections involved views computed using joins on the database relations. This section contains a few more examples involving other operators in the algebra. For these experiments, the logical I/O necessary to compute the changes to the views is estimated using the RHODES optimizer.

6.4.1 Project Distinct

Let us consider two views defined as

V1 : select distinct PS_PARTKEY from PS

V2 : select distinct PS_SUPPKEY from PS

Figure 6.12 shows the estimated I/O necessary to compute V1 and V2 originally (shown with the line "original" in the charts) as well as the estimated I/O to compute their incremental deletions (shown as "deletions") and insertions (shown as "insertions") when PS is updated. The sizes of the updates range from 0.1% to 10% of PS. When we say *updated* we mean that some portion of PS is deleted from PS, and another portion, of the same size, is added to PS. As we see, even for very small updates the I/O necessary to propagate these changes to the views is rather high compared to computing the views.

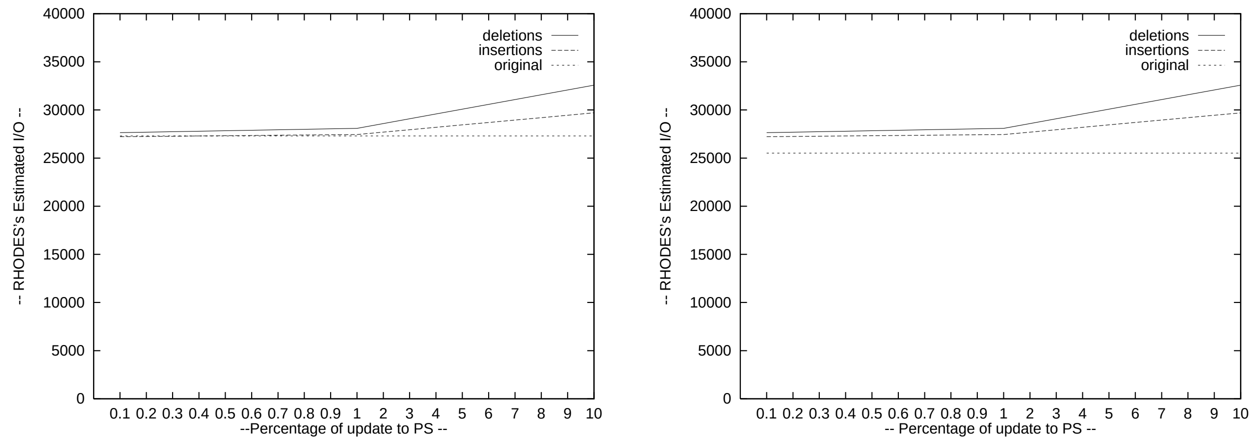


Figure 6.12: Incremental changes to V1 and V2

6.4.2 Set Difference

Let us consider a view defined as

V3 : select P_PARTKEY from P except all select PS_PARTKEY from PS

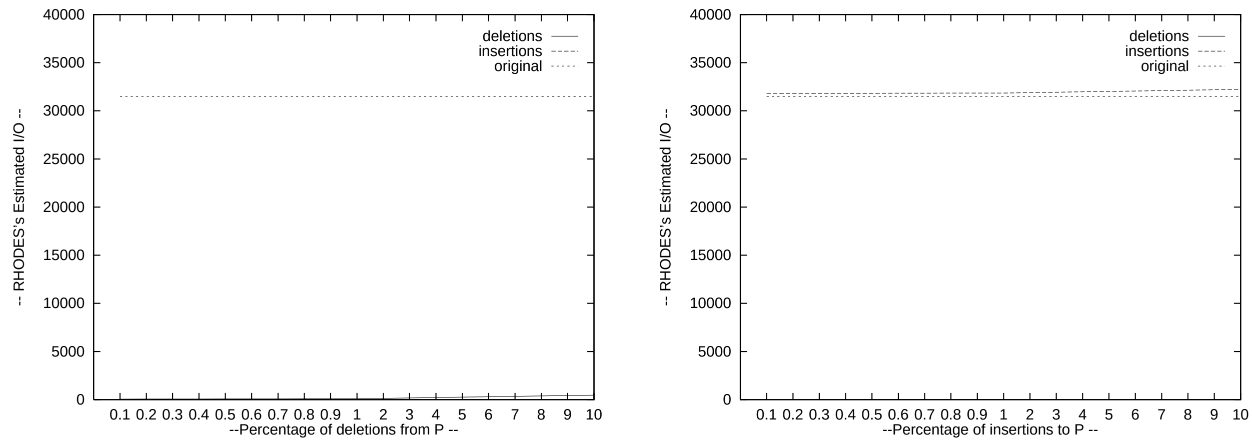


Figure 6.13: Incremental changes to `V3` due to deletions from `P` (first figure) and due to insertions to `P` (second figure)

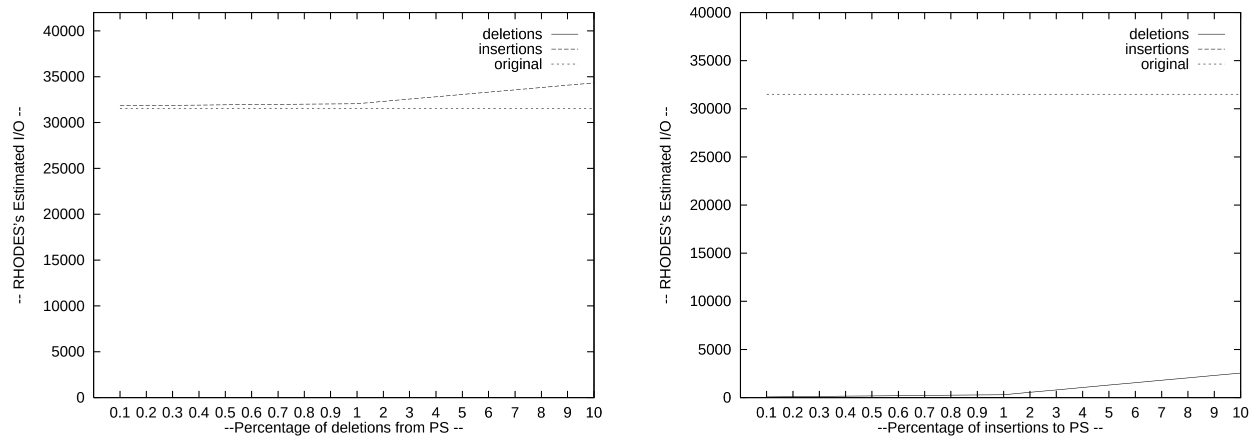


Figure 6.14: Incremental changes to `V3` due to deletions from `PS` (first figure) and due to insertions to `PS` (second figure)

Figure 6.13 shows the estimated logical I/O of propagating incremental changes to `V3` when tuples are either only deleted from or only inserted to `P`. As we see in these figures, the cost of propagating the deletions from `P` is very low while the cost of propagating the insertions to `P` is rather high and comparable to the cost of computing `V3`. Exactly the opposite happens when, instead of `P`, it is the relation `PS` that accepts the changes. Figure 6.14 shows the results in this case. Note however that propagating insertions in this case is slightly more costly than propagating deletions in the previous case.

Finally, Figure 6.15 show two more experiments. The first plot in the figure shows what logical I/O is necessary to compute incremental changes when parts of both `P` and `PS` are deleted. The second plot shows what logical I/O is necessary to compute incremental changes when all relations incur a number of deletions and insertions, that is they simultaneously change by both insertions and deletions. Note that computing the incremental deletions is rather beneficial. But computing the incremental insertions is comparable to computing the view from scratch.

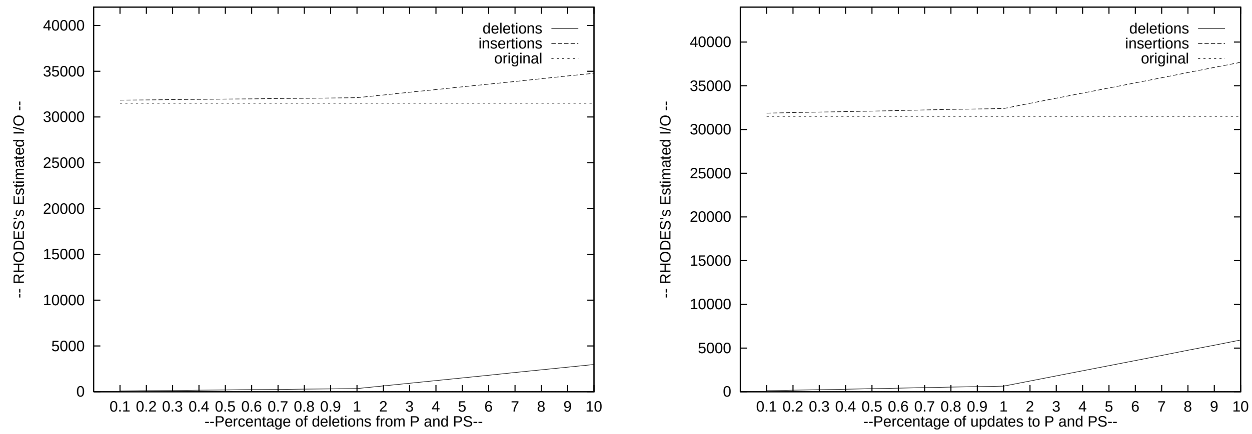


Figure 6.15: Incremental changes to `V3` due to deletions from both `P` and `PS` (first figure) and due to updates to both `P` and `PS` (second figure)

6.5 Making Inferences

The examples presented so far demonstrate a number of interesting points. First, they show that it is indeed possible for different change propagation strategies to result in significant performance differences. Second, they show that having indices defined on the database data is very helpful during change propagation, even in the case where these indices do not participate in view creation (see example 2 of set 1). Although, this is not a surprising result, it is still interesting to see how the performance of change propagation is affected by the creation and use of the index.

We can see that in most cases examined here, propagating deletions seems to be less time consuming than propagating insertions. For database data with keys (such as the ones used here), the deletions can be propagated independently of the database insertions. However, the same is not true for insertions, where access and manipulation of the deletions is also necessary in order to correctly propagate the insertions.

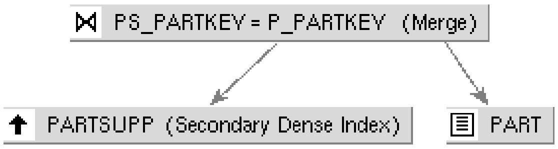
Another interesting point that we can make from these experiments is that the foreign key optimizations greatly improve the performance of change propagation and, thus, of view maintenance. As these optimizations are generally applicable only to certain change propagation expressions but not to others, our claim that "an intelligent component of the DBMS, such as the query optimizer, should be responsible for the generation as well as the optimization of incremental and change propagation queries" is strongly supported by the results of these experiments.

Looking at the results of these experiments, we can also see that even for small updates, where relations change by no more that 10% of the relation's original sizes, the performance of change propagation may be comparable to the performance of view computation. Incremental view maintenance involves the computation of *both* insertions and deletions before incremental changes can be incorporated into the old values of the views. If one adds the cost to compute both insertions and deletions and the cost of incorporating these insertions and deletions to the old value of the view, one can see that it is not at all clear that incremental view maintenance is going to be better than view re-evaluation.

6.6 Validation of RHODES

The examples of this chapter demonstrate that our proposed optimizations have the potential to improve the performance of incremental view maintenance and change propagation. This is especially true for the optimizations due to the foreign key references. The optimized queries that we run on DB2 were generated using the RHODES optimizer. The non-optimized queries were generated by applying the generation algorithm of Griffin and Libkin [GL95] (by hand). These examples, thus, provide an indication that the optimizations that RHODES is capable of executing are very useful.

To further validate the basic RHODES optimizer, we can examine and compare the execution plans generated by RHODES and the DB2 optimizer, for each of the three views `J1`, `J2` and `J3`. The DB2 output is generated by the **`dynexpln`** command of DB2.




```
SQL Statement:

      SELECT *
      FROM PART, PARTSUPP
      WHERE P_PARTKEY = PS_PARTKEY

Coordinator Subsection:
      Distribute Subsection #1
            Directed to Single Node
            Partition Map ID = 1, Nodegroup = IBMDEFAULTGROUP, #Nodes = 1
      Access Table Queue ID = q1 #Columns = 14

Subsection #1:
      Access Table Name = VISTA.PARTSUPP ID = 23 #Columns = 5
            Scan Direction = Forward
            Relation Scan
            Lock Intent Share
            Sargable Predicate(s)
                  #Predicates = 1
                  Create/Insert Into Sorted Temp Table ID = t1
                        Sort #Columns = 1
                        Not Piped
                  Sorted Temp Table Completion ID = t1

      Access Table Name = VISTA.PART ID = 21 #Columns = 9
            Scan Direction = Forward
            Relation Scan
            Lock Intent Share
            Sargable Predicate(s)
                  #Predicates = 1
                  Create/Insert Into Sorted Temp Table ID = t2
                        Sort #Columns = 1
                        Piped
                  Sorted Temp Table Completion ID = t2

      Access Temp Table ID = t2 #Columns = 9
            Scan Direction = Forward
            Relation Scan

      Merge Join
            Join Strategy: Collocated

      Access Temp Table ID = t1 #Columns = 5
            Scan Direction = Forward
            Relation Scan
            Create/Insert Into Table Queue ID = q1,
```

Figure 6.16: Execution plans for `J1` chosen by RHODES and DB2

6.6.1 Execution Plans for `J1`

Figure 6.16 shows the plans chosen by RHODES and DB2 for the view `J1`.

Both RHODES and the DB2 optimizer chose the same join algorithm to implement J1. RHODES knows that the relation P is sorted on P_PARTKEY (because a primary index has been defined on it). To retrieve the tuples from P, RHODES reasons that no (external) sorting is necessary. A simple relation scan is sufficient to generate the tuples in a sorted order. DB2, however, sorts the P tuples before further processing.

However, there is a second difference between the two plans. It is in how the PS relation is accessed. RHODES retrieves the tuples of PS sorted using the secondary dense index because it thinks that sorting such a big relation is rather expensive. DB2 instead sorts the relation using external sorting (we know that this relation is not sorted in memory because of the "not piped" keyword that appears in the access to PS in the DB2 plan, which means that insufficient memory exists during execution).

The data for this example were created in such a way that the PS relation is in fact sorted according to the PS_PARTKEY. It seems that the DB2 optimizer was able to recognize this fact, while RHODES could not.

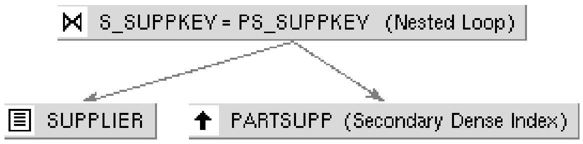
RHODES estimates that the logical I/O to compute J1 is 154,558. The actual logical I/O is 135,013. The actual physical I/O is only 9,972. We believe that the physical I/O is so low because the sorting of PS on PS_PARTKEY (estimated to be rather expensive) is in fact very fast because the data is already sorted on this attribute.

6.6.2 Execution Plans for J2

Figure 6.17 shows the plans chosen by RHODES and DB2 for the view J2.

For this view, the two execution plans chosen by the two optimizers are identical. A nested loops join algorithm is chosen to perform the necessary join. All tuples of the S relation are scanned and, for each one of them, the corresponding tuples in the PS relation are found by using the index on the PS_SUPPKEY.

RHODES estimates that the logical I/O to compute J2 is 150,641. The actual logical I/O is 101,278. The actual physical I/O is 101,260. We believe that the physical I/O is so high, in this case, because PS is *not* clustered on PS_SUPPKEY and accessing each PS tuple (and we need to access them all) results in one new page I/0.




```
SQL Statement:
    SELECT *
    FROM PARTSUPP, SUPPLIER
    WHERE PS_SUPPKEY = S_SUPPKEY

Coordinator Subsection:
    Distribute Subsection #1
        Directed to Single Node
        Partition Map ID = 1, Nodegroup = IBMDEFAULTGROUP, #Nodes = 1
    Access Table Queue ID = q1 #Columns = 12

Subsection #1:
    Access Table Name = VISTA.SUPPLIER ID = 22 #Columns = 7
        Scan Direction = Forward
        Relation Scan
        Lock Intent Share

    Nested Loop Join
        Join Strategy: Collocated
        Access Table Name = VISTA.PARTSUPP ID = 23 #Columns = 5
            Scan Direction = Forward
            Index Scan: Name = VISTA.PSSUPPKEYIND ID = 2 #Key Columns = 1
            Lock Intent Share
        Create/Insert Into Table Queue ID = q1, Broadcast
```

Figure 6.17: Execution plan for `J2` chosen by RHODES and DB2

6.6.3 Execution Plans for `J3`

Figure 6.18 shows the plans chosen by RHODES and DB2 for the view `J3`.

RHODES chooses the join ordering (`PS` $\bowtie$ `S`) $\bowtie$ `P` while the DB2 optimizer chooses (`S` $\bowtie$ `PS`) $\bowtie$ `P`. Both RHODES and the DB2 choose the same join algorithms to execute these joins. A nested loops join algorithm is used to implement the join between `S` and `PS` and a merge join algorithm is used to implement the join between this intermediate result and `P`.

RHODES reasons that since a primary key index has been defined on the `P` relation, this relation is already sorted on the key attribute `P_PARTKEY` and no (external) sorting is necessary. The DB2 optimizer scans the `P` relation and creates a sorted intermediate table holding the tuples of the `P` relation sorted.

Also, let us call `I` the join between `PS` and `S`. The merge algorithm to implement the join between `I` and `P` requires `I` to be sorted on `PS_PARTKEY` and `P` to be sorted on `P_PARTKEY`. There are two different ways to sort `I` on the `PS_PARTKEY` attribute and the two plans differ on how they deliver `I` sorted.

The way chosen by the DB2 optimizer is to compute the join `I` first and, then, to sort `I` on `PS_PARTKEY`. Because of the referential integrity, however, every tuple in the `PS` relation joins with a tuple from the `S` relation and the join `I` has as many tuples as the `PS` relation has. The difference is that the size of `I` in bytes is much bigger that the size of `PS` in bytes because each `I` tuple also is contains the supplier information. So, why should we sort `I` and not sort `PS` instead?

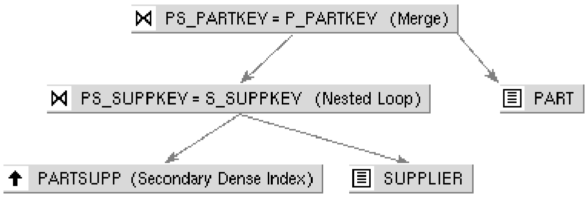
RHODES chooses to sort the `PS` relation before using the nested loops algorithm with `S`. If a nested loops algorithm is performed the output of the nested loops algorithm is sorted on whatever attribute the first input is sorted on. To retrieve `PS` sorted on `PS_SUPPKEY`, the secondary dense index defined on it is used.

There is also another reason why this is a better plan. Scanning the `S` relation first and then using the index on `PS_SUPPKEY` results in finding for each `S` tuple all corresponding `PS` tuples. But all `PS` tuples must be accessed. Since the `PS` relation is *not* clustered on the `PS_SUPPKEY` attribute, finding the `PS` tuples requires as many page I/O's as tuples in `PS` approximately. Accessing the `PS` relation using the index defined on `PS_SUPPKEY` also results in as many page I/O's as tuples in `PS`, but it completely saves the cost of sorting `I`.

RHODES estimates that the logical I/O to compute `J3` is 353,662. The actual logical I/O is 345,099. The actual physical I/O is 122,194. Table 6.2 summarizes the estimated I/O from both RHODES and DB2 for all three execution plans.

View	RHODES I/O	DB2 Log I/O	DB2 Phys I/O
$J1:$	$154,558$	$135,013$	$9,972$
$J2:$	$150,641$	$101,278$	$101,260$
$J3:$	$353,662$	$345,099$	$122,194$

Table 6.2: Logical and physical I/O




```
SQL Statement:

      SELECT *
      FROM PART, PARTSUPP, SUPPLIER
      WHERE PS_PARTKEY = P_PARTKEY AND PS_SUPPKEY = S_SUPPKEY

Coordinator Subsection:
      Distribute Subsection #1
      Directed to Single Node
      Partition Map ID = 1, Nodegroup = IBMDEFAULTGROUP, #Nodes = 1
      Access Table Queue ID = q1 #Columns = 21

Subsection #1:
      Access Table Name = VISTA.SUPPLIER ID = 22 #Columns = 7
            Scan Direction = Forward
            Relation Scan
            Lock Intent Share
      Nested Loop Join
            Join Strategy: Collocated
            Access Table Name = VISTA.PARTSUPP ID = 23 #Columns = 5
                  Scan Direction = Forward
                  Index Scan: Name = VISTA.PSSUPPKEYIND ID = 2 #Key Columns = 1
                  Lock Intent Share
                  Sargable Predicate(s)
                        #Predicates = 1
            Create/Insert Into Sorted Temp Table ID = t1
                  Sort #Columns = 1
                  Not Piped
      Sorted Temp Table Completion ID = t1
      Access Table Name = VISTA.PART ID = 21 #Columns = 9
            Scan Direction = Forward
            Relation Scan
            Lock Intent Share
            Sargable Predicate(s)
                  #Predicates = 1
                  Create/Insert Into Sorted Temp Table ID = t2
                        Sort #Columns = 1
                        Piped
      Sorted Temp Table Completion ID = t2
      Access Temp Table ID = t2 #Columns = 9
            Scan Direction = Forward
            Relation Scan
      Merge Join
            Join Strategy: Collocated
            Access Temp Table ID = t1 #Columns = 12
                  Scan Direction = Forward
                  Relation Scan
      Create/Insert Into Table Queue ID = q1
```

Figure 6.18: Execution plan for J3 chosen by RHODES and DB2

Chapter 7

Conclusions

This chapter concludes the dissertation with a presentation of the research contributions along with a discussion on the limitations of the approach and an outline for further research. This chapter also includes a discussion of practicality and limitations of materialized views.

7.1 Research Contributions

One primary contribution of our thesis is that we provide a different perspective to address view maintenance and, consequently, change propagation. We can summarize this perspective with:

> "both the choice of incremental view maintenance versus non-incremental view maintenance as well as the choice of an appropriate propagation strategy are best left to the database query optimizer to make."

Another primary contribution is that we provide a repertoire of original optimizations specific to incremental view maintenance and change propagation.

In particular:

1. In this dissertation, we experimentally demonstrate that the performance of incremental view maintenance depends on the physical aspects of the underlying database management system, such as the availability of index structures, the sizes of the relations involved, as well as the sizes of the database updates. For this reason, we argue that incremental maintenance strategies should not be adopted by

a database system without first taking these system properties into consideration. We also argue that the database query optimizer is a reasonable component of the database system to decide, at the point of view maintenance, whether a view is to be maintained incrementally or not, because the optimizer has knowledge of, and access to, all of the parameters that may affect this choice. To the best of our knowledge, this is the first work that does not commit to the a priori usage of incremental view maintenance due to assuming that in "typical" situations incremental view maintenance is very efficient.

2. We demonstrate how one can take an algorithm for change propagation and incremental view maintenance and incorporate it into a database query optimizer. We have built the RHODES relational query optimizer that supports both change propagation and incremental view maintenance. Our approach does not require significant changes in the DBMS, other than the proposed extension to the query optimizer and some bookkeeping about the database updates (which is necessary in any incremental maintenance technique). Therefore, using our approach, no additional software must be written, and no special purpose evaluation component must be integrated into the DBMS.

3. Incorporating change propagation and view maintenance into the query optimizer allows the optimizer to be responsible for the generation of the queries to be executed in order to support change propagation or incremental view maintenance. In incremental view maintenance, for example, there may be more than one different strategy to maintain a view incrementally. Choosing among the different strategies is not an easy task and cannot always be done independently of the system aspects of the database.

4. Incorporating the generation of change propagation and view maintenance into the query optimizer also allows the query optimizer to use, in addition to traditional optimizations, incremental maintenance specific optimizations in order to find the best possible way to maintain a view. A repertoire of maintenance-specific optimizations are provided in the thesis. These proposed optimizations are also

validated experimentally. For example, when the updates affect only part of the database, some view maintenance expressions may evaluate to empty and the optimizer may be able to recognize this in order to avoid extra computation involving the database, and, thus, to decide that incremental view maintenance is more efficient than re-evaluation.

Apart from the above contributions, we also make two secondary contributions:

1. The research of this dissertation has lead to the implementation of a reusable relational query optimizer. The design of the query optimizer is such that it can be extended into, for example, a query optimizer for a parallel database rather easily [Zil96]. Another novel feature of RHODES is that it considers the alternative to use already materialized views in order to optimize the execution of general queries. This has recently been recognized as a potential for query optimization but RHODES is, to the best of our knowledge, the first optimizer that supports this.

2. Textual representations of the generated plans of a query optimizer are rather difficult to manage and understand, especially for *relatively large* plans where the textual description of the plan may be hundreds of pages long. The query plan generated by RHODES is supplied to a plan visualization tool generated by appropriately configuring a general visualization tool for graphical presentation of structured information [Noi96]. The browser allows us to view the chosen plan for any given conventional or incremental query and to view details of the plan, including statistics, access structures, and so on. This functionality is similar to DB2's `visual explain` facility [DB2].

Next we describe some of the limitations of our approach.

7.2 Limitations

In this thesis, we showed how one can take their favorite algorithm for incremental view maintenance and incorporate it into the database query optimizer. This allows the query optimizer to be responsible for deciding whether a view is to be maintained incrementally as well as which change propagation expressions to use to compute the incremental changes. As a proof of concept that it is easy to extend an optimizer to support this, we have built the RHODES database optimizer that supports both incremental view maintenance and change propagation. The implementation of RHODES is such that each view is examined independently of the other views. That is, if the views depend on each other, RHODES *does not* try to find the best way to maintain the set of views. We consider this as a major disadvantage of our work. However, we also consider it an implementation problem. It is possible to extend RHODES in order to optimize the set of available views in some topological ordering. Then, RHODES can use the fact that some of these views could have already been updated at the time views that depend on it are examined by it.

Another limitation of our technique is the increase in time and system resource usage during view maintenance due to optimization. Many query expressions are examined for the maintenance of each view, and, if there are a lot of views to be maintained, this may result in significant performance degradation. The view maintenance optimization time and resource consumption is influenced by the complexity of each view expression, especially by the number of joins and subqueries the view expressions contain. Ideally, the optimizer could be configured to use default view maintenance expressions (obtained at view compilation time or at the first time view maintenance is performed) for views that require short time to be computed. Decision support queries or end-of-the-month queries, however, are good examples of complex queries, where the increase in the optimization time may not affect the overall performance of the system very much.

All experiments conducted to support our thesis were on data with uniform distributions. We consider this as a limitation of our thesis, since it is believed that uniform distribution are not very natural. It would be interesting to see how the performance results would be affected under different distributions of data values.

The impact on the overall performance of the database system is a limitation of view maintenance, in general, both incremental and not. View maintenance at the end of each updating transaction slows down transactions. It is an open problem to determine how view maintenance and the overall performance of the database system are related. We believe that it would probably depend on each application whether the performance impact of materialized views is beneficial for the application or not.

7.3 Discussion and Open Problems

We conclude this dissertation with a discussion on the practicality of materialized views along with a discussion of some open problems.

Almost every commercial database system supports views (Oracle, Sybase, DB2, etc.). Materialized views exist mostly during the execution of a single query (in the form of materialized intermediate results) but are destroyed right after the completion of the query. Oracle is now implementing materialized views and incremental view maintenance support. Active database systems, such as those described by Widom and Ceri [WC96], can support materialized views but it is (mostly) up to the user to specify how view maintenance is realized by specifying appropriate active rules in the system's rule language. Among those systems supporting materialized views, only the Starburst system [HCL+90, Wid96] has automatically generated active rules for the views' incremental maintenance. The ARIEL system [HBH+95, Han96] supports automatic incremental maintenance of (certain) materialized views by exploiting the specialized data structures used by the system.

According to representatives of major database companies at the workshop on Materialized Views associated with the SIGMOD '96 conference, most database vendors are considering incorporating materialized views into their products because of the demand for materialized views by new applications such as data replication, decision support, data mining, and so on. The requirements set by the vendors are that materialized views work well with the other components of the database system, that they do not have a negative effect on the overall performance of the system, and that they be used for optimization of general queries. Domain specific knowledge of each application will probably be used to justify the use of materialized views.

In the SIGMOD '96 workshop on Views there was a debate on whether materialized views should be part of the SQL-93 standard. This would of course mean that all database products would support materialized views and perhaps their incremental maintenance. The majority of participants agreed that materialized views, like B-trees, are optimization techniques and should not be part of the standard. Others hoped that they will soon become part of the standard because of their potential for optimization.

There are some problems associated with materialized views and these must be solved or appropriately addressed before materialized views become part of commercial database products. The maintenance of materialized views, for example, slows down update transactions, reduces query throughput and interferes with concurrency control. Materialized views require more disk space and, sometimes, special algorithms and data structures. Other problems associated with materialized views is what views to materialize, how to store these views as well as how to keep them consistent with the database, and, as discussed above, how to do all this without affecting the performance of the rest of the database system.

In the current state of incremental view maintenance research, there seems to be an over-formulation of how to do incremental view maintenance: there are too many proposed algorithms on how to do incremental view maintenance for a number of different data models. What seems to be missing, though, is a thorough evaluation of the problems discussed above and implementation-specific proposals of how to incorporate materialized views and their incremental view maintenance into a database system without affecting performance unacceptably. Finally, database query optimizers must be extended to detect and use materialized views automatically, as they do with B-trees and join indices, for instance, which also need to be consistent with respect to the database.

Bibliography

[ABW88] K. R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

[AISN90] G. Ausiello, G. Italiano, A.M. Spaccamela, and U. Nanni. Incremental Algorithms for Minimal Length. In *1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 12–21, 1990.

[Alb91] J. Albert. Algebraic Properties of Bag Data Types. In *17th International Conference on Very Large Databases*, pages 211–219, 1991.

[Ana96] T.K. Anand. Incremental Maintenance of Views in Database Systems. Master's thesis, Department of Computer Science, University of Toronto, 1996.

[AV95] T.K. Anand and D. Vista. Incremental Query Evaluation for Programs with Duplicate Semantics. Unpublished Manuscript, 1995.

[BA93] D. Barbará and R. Alonso. Answering Continuous Queries in General Environments. Technical report, Matsushita Information Technology Laboratory, 1993.

[Ban85] F. Bancilhon. Naive Evaluation of Recursively Defined Relations. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*. Springer-Verlag, 1985.

[Bar92] D. Bardon. Management of Color Usage in Dynamic Mapping Environments: Balancing Semantics, Visual Ordering and Discernability. In T. Catarci, M.F. Costabile, and S. Levialdi, editors, *Proc. of Advanced Visual Interfaces 1992*, volume 36 of *Series in Computer Science*, pages 50–67. World Scientific, 1992.

[BC79] O.P. Buneman and E.K. Clemons. Efficiently Monitoring Relational Databases. *ACM Transactions on Data Base Systems*, 4(3):368–382, 1979.

[BCL89] J.A. Blakeley, N. Coburn, and P-A. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Transactions on Data Base Systems*, 14(3):369–400, 1989.

[BKV90] A.L. Buchsbaum, P.C. Kanellakis, and J.S. Vitter. A Data Structure for Arc Insertion and Regular Path Finding. In *1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 22–31, 1990.

[BLT86] J.A. Blakeley, P-A. Larson, and F.W. Tompa. Efficiently Updating Materialized Views. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 61–71, 1986.

[BM91] C. Beeri and T. Milo. A Model for Active Object Oriented Database. In *Proceeding of the 17th International Conference on Very Large Data Bases*, pages 337–349, 1991.

[BR86] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 16–52, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proceedings of Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 269–283, 1987.

[BS81] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Transactions on Data Base Systems*, 6(4):557–575, 1981.

[BW93] E. Baralis and J. Widom. Using Delta Relations to Optimize Condition Evaluation in Active Databases. Technical report, Department of Computer Science, 1993. Technical Report Number Stan-cs-93-1495.

[CC82] G.A. Cheston and D.G. Corneil. Graph Property Update Algorithms and their Applications to Distance Matrices. *INFOR*, 20(3):178–201, 1982.

[CG85] S. Ceri and G. Gottlob. Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, 1985.

[CG96] L.S. Colby and T. Griffin. An Algebraic Approach to Supporting Multiple Deferred Views. In *Workshop on Views*, pages 103–109, 1996.

[CGL+96] L.S. Colby, T. Griffin, L. Libkin, I.S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 469–480, 1996.

[CH91] J-P. Cheiney and Y-N. Huang. Set-Oriented Propagation of Updates Into Transitively Closed Relations. In *Proceeding of DOOD*, pages 503–523, 1991.

[CM96] L.S. Colby and I.S Mumick. Staggered Maintenance of Multiple Views. In *Workshop on Views*, pages 119–128, 1996.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proceeding of the 16th International Conference on Very Large Data Bases*, pages 566–577, 1990.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceeding of the 17th International Conference on Very Large Data Bases*, pages 577–589, 1991.

[DB82] U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Transactions on Data Base Systems*, 8(3):381–416, 1982.

[DB2] DB2. On line documentation. IBM RS6000.

[DD93] C. J. Date and H. Darwen. *A Guide to the SQL Standard (Third Edition)*. Addison-Wesley, 1993.

[DLW95] G. Dong, L. Libkin, and L. Wong. On Impossibility of Decremental Recomputation of Recursive Queries in Relational Calculus and SQL. In *Proceedings of International Workshop on Database Programming Languages*, 1995.

[DT92] G. Dong and R. Topor. Incremental Evaluation of Datalog Queries. In J. Bishup and R. Hull, editor, *Proceeding of 4th International Conference on Database Theory*, pages 282–296, 1992.

[Elk90] C. Elkan. Independence of Logic Database Queries and Updates. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 154–160, 1990.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems (Second Edition)*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[FRV96] D. Florescu, L. Raschid, and P. Valduriez. Answering Queries Using OQL View Expressions. In *Workshop on Views*, pages 84–90, 1996.

[FSDS79] A.L. Furtado, K.C. Sevcik, and C.S. Dos Santos. Permitting Updates Through Views of Data Bases. *Information Systems*, 4:269–283, 1979.

[GHJ92] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Implementation of Delayed Updates in Heraclitus. In *Proceedings of the 3rd International Conference on Extending Database Technology*, pages 261–276, 1992.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs, J. Castillo, M. Escobar-Molano, S. Lu, J. Luo, C. Tsang, and G. Zhou. On Implementing a Language for Specifying Active Database Execution Models. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 441–454, August 1993.

[GJS92] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event Specification in an Active Object Oriented Database. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 81–90, 1992.

[GKM92] A. Gupta, D. Katiyar, and I.S. Mumick. Counting Solutions to the View Maintenance Problem. In *Workshop on Deductive Databases, JICSLP*, pages 185–194, 1992.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 328–339, 1995.

[GLT] T. Griffin, L. Libkin, and H. Trickey. A Correction to "Incremental Recomputation of Active Relational Expressions" by Qian and Wiederhold. To Appear in IEEE Transactions on Knowledge and Data Engineering.

[GM93a] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the 9th International Conference on Data Engineering*, pages 209–218. IEEE Computer Society Press, 1993.

[GM93b] S. Grumbach and T. Milo. Towards Tractable Algebras for Bags. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 49–58, 1993.

[GM95] A. Gupta and I.S. Mumick. Maintenance of Materialized Views: Problems, Techniques and Applications. *Data Engineering, Special Issue on Materialized Views and Data Warehousing, IEEE Computer Society*, 18(2):3–18, 1995.

[GMR95] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting Materialized Views after Redefinitions. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 211–222, 1995.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 157–166, 1993.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[Gra94] G. Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, February 1994.

[Han96] E.N. Hanson. The Ariel Project. In J. Widom and S. Ceri, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[Has95] M. Hasan. An Active Temporal Model for Network Management Databases. In *Proceedings of the IEEE/IFIP Fourth International Symposium on Integrated Network Management*, pages 524–535. Chapman and Hall, May 1995.

[Has96] M.Z. Hasan. Active Temporal Rules and Declarative Visualization for Network Management. Ph.D. Thesis to be submitted, 1996.

[HBH+95] E.N. Hanson, S. Bodagala, M. Hasan, G. Kulkarni, and J. Rangarajan. Optimized Rule Condition Testing in Ariel using Gator Networks. Technical report, CISE Department, University of Florida, October 1995. Technical Report Number TR-95-027.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HD92] J.V. Harrison and S. Dietrich. Maintenance of Materialized Views in Deductive Databases: An Update Propagation Approach. In *Workshop on Deductive Databases, JICSLP*, pages 56–65, 1992.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *Data Engineering, Special Issue on Materialized Views and Data Warehousing, IEEE Computer Society*, 18(2):41–48, 1995.

[Huy96] N. Huyn. Efficient View Self-Maintenance. In *Workshop on Views*, pages 17–25, 1996.

[Ita91] G.F. Italiano. Distributed Algorithms for Updating Shortest Paths (EXTENDED ABSTRACT). In *Workshop on Distributed Algorithms and Graphs*, pages 200–211, 1991.

[Jag90] H.V. Jagadish. A Compression Technique to Materialize Transitive Closure. *ACM Transactions on Data Base Systems*, 15(4):558–598, 1990.

[Jak92] H. Jakobsson. On Materializing Views and On-Line Queries (Extended Abstract). In J. Bishup and R. Hull, editor, *Proceeding of 4th International Conference on Database Theory*, pages 407–420, 1992.

[Kel85] A. M. Keller. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 154–163, 1985.

[Kim80] W. Kim. A New Way to Compute the Product and Join of Relations. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 179–187, 1980.

[KP81] S. Koenig and R. Paige. A Transformational Framework for the Automatic Control of Derived Data. In *7th International Conference on Very Large Databases*, pages 306–318, 1981.

[Küc91] V. Küchenhoff. On the Efficient Computation of the Difference Between Consecutive Database States. In *Proceeding of DOOD*, pages 478–502, 1991.

[LFL88] M. Lee, J.C. Freytag, and G.M. Lohman. Implementing an Interpreter for Functional Rules in a Query Optimizer. In *Proceeding of the 14th International Conference on Very Large Data Bases*, pages 218–229, 1988.

[LHM+86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A Snapshot Differential Refresh Algorithm. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 53–60, 1986.

[LMSS95] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 95–104, 1995.

[Loh88] G.M. Lohman. Grammar-Like Functional Rules for Representing Query Optimization Alternatives. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 18–27, 1988.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. The World Wide Web as a Collection of Views: Query Processing in the Information Manifold. In *Workshop on Views*, 1996.

[LS93] A.Y. Levy and Y. Sagiv. Queries Independent of Updates. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 171–181, August 1993.

[MP94] I.S. Mumick and H. Pirahesh. Implementation of Magic-Sets in a Relational Database System. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 103–114, 1994.

[MPR90] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan. The Magic of Duplicates and Aggregates. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 264–277, 1990.

[Noi96] E.G. Noik. *Dynamic Fisheye Views: Combining Dynamic Queries and Mapping with Database Views.* PhD thesis, Dept. of Comp. Sci., U. of Toronto, April 1996.

[NY83] J-M. Nicolas and K. Yazdanian. An Outline of BDGEN: A Deductive DBMS. In *Information Processing*, pages 711–717, 1983.

[Ple93] D. Plexousakis. Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 146–157, August 1993.

[Qua96] D. Quass. Maintenance Expressions for Views with Aggregation. In *Workshop on Views*, pages 110–118, 1996.

[QW91] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[RCK95] N. Roussopoulos, C.M. Chen, and S. Kelley. The ADMS Project: Views "R" Us. *Data Engineering, Special Issue on Materialized Views and Data Warehousing, IEEE Computer Society*, 18(2):19–28, 1995.

[Rou91] N. Roussopoulos. An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *ACM Transactions on Data Base Systems*, 16(3):535–563, 1991.

[SAC+94] 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 M. Stonebraker, editor, *Readings in Database Systems*. Morgan Kaufmann Publishers, 1994.

[SH91] R.S. Sundaresh and P. Hudak. Incremental Computation via Partial Evaluation. In *Synposium on Principles of Programming Languages*, pages 1–13, 1991.

[Shn83] B. Shneiderman. Direct manipulation: A step beyond programming. *IEEE Computer*, 16(8):57–69, August 1983.

[Shn94] B. Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6):70–77, November 1994.

[SI84] O. Shmueli and A. Itai. Maintenance of Views. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 240–255, 1984.

[SK96] E. Simon and J. Kiernan. The A-RDL System. In J. Widom and S. Ceri, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceeding of the 17th International Conference on Very Large Data Bases*, pages 469–478, 1991.

[SSU95] A. Silberschatz, M. Stonebraker, and J. Ullman. Database Research: Achievements and Opportunities Into the 21st Century. Report of an NFS Workshop on the Future of Database Systems Research, 1995.

[Sto75] M. Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 65–78, 1975.

[TPC95] Transaction Processing Performance Council TPC. Benchmark D. Standard Specification, Revision 1.0, 1995.

[TR81] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge Base Systems I*. Computer Science Press, 1988.

[UO92] T. Urpí and A. Olivé. A Method for Change Computation in Deductive Databases. In *Proceeding of the 18th International Conference on Very Large Data Bases*, pages 225–237, 1992.

[Vis94] D. Vista. View Maintenance in Relational and Deductive Databases by Incremental Query Evaluation. In *IBM CASCON Conference, CD-ROM,*, 1994.

[War] M. Ward. http://cs.wpi.edu/ matt/courses/cs563/talks/datavis.html. WPI CS Department.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Trigers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, Inc., 1996.

[WDSY91] O. Wolfson, H.M. Dewan, S.J. Stolfo, and Y. Yemini. Incremental Evaluation of Rules and its Relationship to Parallelism. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 78–87, 1991.

[Wid96] J. Widom. The Starburst Rule System. In J. Widom and S. Ceri, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[Wie92] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, March 1992.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 316–327, 1995.

[ZH96] G. Zhou and R. Hull. A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. In *Proceeding of ACM-SIGMOD Conference on Management of Data*, pages 481–492, 1996.

[ZHKF95] G. Zhou, R. Hull, R. King, and J. Franchitti. Supporting Data Integration and Warehousing Using H20. *Data Engineering, Special Issue on Material-*

ized Views and Data Warehousing, IEEE Computer Society, 18(2):29–40, 1995.

[Zil96] D. Zilio. Personal Communication. University of Toronto, 1996.

Appendix A

Cardinality Estimation

In this appendix, we present the derivation of one physical property, the cardinality, from the physical properties of the inputs for each available physical operator. In what follows, we use n as the cardinality of the derived relation, and n_1, n_2 as the cardinality of the input relations.

A.1 Cardinality Estimation

BINARY_SEARCH, FILE_SCAN, INDEX_SCAN, SELECT_ALGO

The number of tuples in the output relation of these operators is a fraction of the number of tuples in the input relation. We call this fraction the *selectivity* of the selection conditions that appear as arguments to the algorithms. At the end of the appendix, we describe how we estimate selectivities.

$$n = selectivity \times n_1$$

CARTESIAN_ALGO

The number of tuples in the output relation is always

$$n = n_1 \times n_2$$

DUPLICATE_ELIMINATION

Let $image(a_i)$ denote the number of distinct values that the i-th attribute in the input relation may have. This information is stored in the catalog. The output relation's size cannot be smaller than the maximum image size. Thus, a lower bound for the size of the output is

$$\textit{lower bound} = \max_{i=1}^{arity} \{image(a_i)\}$$

The output cannot be larger than the input or the product of all attributes' image size, whichever is less. Thus, an upper bound for the size is

$$\textit{upper bound} = \min \left\{ n_1, \prod_{i=1}^{arity} (image(a_i)) \right\}$$

We estimate the size of the output as

$$n = \left\lceil \frac{\textit{lower bound} + \textit{upper bound}}{2} \right\rceil$$

MERGE, NESTED_LOOP

The number of tuples in the output relation is a fraction of the cartesian product of the two relations. We call this fraction the *join selectivity*. At the end of the appendix, we describe how we estimate join selectivities.

$$n = \textit{join selectivity} \times n_1 \times n_2$$

PROJECT_ALGO, SORT

For these operators, the input size does not change.

$$n = n_1$$

SET_DIFF

$$n = \max \left\{ \left\lceil \frac{n_1}{2} \right\rceil, \left\lceil n_1 - \frac{n_2}{2} \right\rceil \right\}$$

SET_INTERSECT

$$n = \min \left\{ \left\lceil \frac{n_1}{2} \right\rceil, \left\lceil \frac{n_2}{2} \right\rceil \right\}$$

SET_UNION

$$n = n_1 + n_2$$

For SET_DIFF_D, SET_INTERSECT_D and SET_UNION_D, we estimate the number of duplicates in each input, and then we use the formulas presented above (thus, n_1 and n_2, in these cases, are the sizes of the two inputs after duplicate elimination). Duplicate estimation follows the formula presented for duplicate elimination.

A.2 Selectivity Estimation

The fraction of tuples from a relation satisfying a given selection condition is called the *selectivity* of the condition. The smaller the selectivity of a condition, the fewer tuples the condition selects and the larger the desirability of using this condition first to retrieve tuples. The selectivity of conjunctive condition $\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_k$ is the product of the selectivities of each individual selection condition θ_i [EN94]. The different forms of θ_i known to RHODES are: $(X_i \;\; op \;\; \texttt{val})$ and $(X_i \;\; op \;\; X_j)$ where $op \in \{=, >, <\}$. The selectivity of each θ_i is defined according to the type of θ_i [SAC+94]:

- For condition $X_i = \texttt{val}$
$$selectivity = \frac{1}{image(X_i)}$$
This formula assumes an even distribution of tuples among the different values in the domain. However, if $\texttt{val} < \min(X_i)$, or $\texttt{val} > \max(X_i)$, the selectivity is 0.

- For condition $X_i > \texttt{val}$, we do a linear interpolation of the value `val` within the range of values of attribute X_i from $\min(X_i)$ to $\max(X_i)$, and we estimate
$$selectivity = \frac{\max(X_i) - \texttt{val}}{\max(X_i) - \min(X_i)}$$
If $\texttt{val} < \min(X_i)$, the selectivity is 1, and if $\texttt{val} > \max(X_i)$, the selectivity is 0.

- For condition $X_i < \texttt{val}$

$$selectivity(X_i < \texttt{val}) = 1 - selectivity(X_i = \texttt{val}) - selectivity(X_i > \texttt{val})$$

- For condition $X_i = X_j$

$$selectivity = \frac{1}{\max\{image(X_i), image(X_j)\}}$$

This formula assumes that each value in the domain of the attribute with the smaller image size has a matching value in the other attribute. If X_i is the same attribute name as X_j, the selectivity is 1. Also, if $\max(X_j) < \min(X_i)$, or $\max(X_i) < \min(X_j)$, the selectivity is 0.

- For condition $X_i > X_j$, if $image(X_j) > image(X_i)$, then

$$selectivity = 1 - \frac{image(X_j) - image(X_i)}{image(X_j)}$$

else

$$selectivity = \frac{image(X_i) - image(X_j)}{image(X_i)}$$

If X_i is the same attribute name as X_j, the selectivity is 0. If $\max(X_i) < \min(X_j)$, the selectivity is also 0. If $\min(X_i) > \max(X_j)$, the selectivity is 1.

- For condition $X_i < X_j$

$$selectivity(X_i < X_j) = 1 - selectivity(X_i = X_j) - selectivity(X_i > X_j)$$

- If the attribute(s) appearing in the selection condition are not arithmetic or if, for some reason, the required statistics are not available in the catalog, for an equality condition

$$selectivity = \frac{1}{10}$$

and for a comparison condition

$$selectivity = \frac{1}{3}$$

There is no significance to these default numbers, other that an equality condition is more selective than a comparison condition.

A.3 Join Selectivity Estimation

A join $A \bowtie_{A_1=B_1} B$ is a selection whose condition is the join condition, $A_1 = B_1$, from the cartesian product, $A \times B$, of the two relations being joined. The fraction of tuples from the cartesian product satisfying the join condition is called the *join selectivity*. The only join condition that is allowed in RHODES is of the form $A_1 = B_1$ where A_1 is an attribute of relation A and B_1 is an attribute of relation B. There are two cases for estimating the join selectivity:

- If no foreign key constraint is known between attribute A_1 of A and attribute B_1 of B, then
  - If A_1 and B_1 are not arithmetic, then
  $$join\ selectivity = \frac{1}{1000}$$
  - Otherwise, let the range of the domain of the A_1 attribute is $d_A = \max(A_1) - \min(A_1) + 1$ and the range of the domain of B_1 is $d_B = \max(B_1) - \min(B_1) + 1$ with an overlap d. (We assume that the distinct values are uniformly distributed within each range and that the tuples are uniformly distributed in the distinct values.) If no overlap exists, then
  $$join\ selectivity = 0$$
  else, let v_A be the total number of values for the A_1 attribute in the overlap and v_B the total number of values for the B_1 attribute in the overlap, i.e., $v_A = \lceil d/d_A * image(A_1) \rceil$ and $v_B = \lceil d/d_B * image(B_1) \rceil$.
  We define
  $$join\ selectivity = \frac{\min\{v_A, v_B\}}{image(A_1) * image(B_1)}$$
- If, however, $B_1 1$ is the key attribute of B and there is a foreign key reference from A_1 to $B_1 1$, then
$$join\ selectivity = \frac{1}{image(B_1 1)}$$
In fact this selectivity is identical to the selectivity of selection condition $X_i = X_j$ only that we know that $image(B_1 1)$ is at least as big as $image(A_1)$ in this case.

Appendix B

Cost Estimation

In this appendix we present the formulas for cost estimation used by RHODES. In order to estimate the cost of different expressions, RHODES must know what the cost of choosing each algorithm is. In presenting our cost model, we use the following symbols:

- B is the size in bytes of one memory block (usually 1024 bytes);
- M is the size in pages of memory available to the optimizer;
- n is the cardinality (number of tuples) of the output relation and n_1, n_2 are the cardinalities of each of the input relations;
- r is the size in bytes of each tuple in the output relation and r_1, r_2 are the record sizes of the input relations;
- bf is the blocking factor of the output relation, that is, the number of tuples of the output relation that fit into one memory page, and bf_1, bf_2 are the blocking factors of the input relations. The blocking factor is defined as $bf = \lfloor B/r \rfloor$;
- b is the size in blocks of the output relation and b_1, b_2 are the sizes in blocks of the input relations. The size in blocks is defined as $b = \lceil n/bf \rceil$; and,
- I is the size in bytes of one index tuple (we assume this number to be constant for each index).

Next, we present the cost for each algorithm used by RHODES.

FILE_SCAN

This algorithm can be used for two purposes: a) to retrieve all tuples of the relation, and b) to retrieve those tuples satisfying a conjunctive selection condition. When no condition is specified, or when the selection conditions need to be checked against all tuples in the relation, then,

$$cost = b_1$$

If one selection condition is specified equating the key attribute of the relation with a constant, only half of the blocks are reached on the average before finding the (unique) tuple in the result. Then,

$$cost = \left\lceil \frac{b_1}{2} \right\rceil$$

BINARY_SEARCH

This algorithm can be used when a single selection condition is specified equating the ordering attribute of a relation with a constant and the relation is contiguous. Then,

$$cost = \max\left\{ \lceil \log_2(b) \rceil + \left\lceil \frac{n}{bf} \right\rceil - 1, 1 \right\}$$

This cost reduces to $\lceil \log_2(b) \rceil$, if the condition is an equality condition on a key attribute, because $n = 1$ in this case.

INDEX_SCAN

This algorithm can be used when a single condition is specified equating the indexing attribute of a relation with a constant. Let b_i be the number of pages the index itself is stored into. The cost of an index scan depends on the type of the index used [EN94].

In a primary index, there is one index tuple per relation page and $b_i = \lceil b / bf_i \rceil$, where the blocking factor for the index is (always constant) $bf_i = \lfloor B/I \rfloor$. Then,

$$cost = \lceil \log_2(b_i) \rceil + 1$$

In a clustering index, there is one index tuple per distinct value in the indexing attribute and so $b_i = \lceil i/bf_i \rceil$, where i is the number of distinct values. Then,

$$cost = \lceil \log_2(b_i) \rceil + b$$

Finally, in a secondary index, there is one index tuple per relation tuple and so $b_i = \lceil n_1/bf_i \rceil$. Then,

$$cost = \lceil \log_2(b_i) \rceil + \left\lfloor \frac{n}{bf_i} \right\rfloor + n$$

CARTESIAN_ALGO, NESTED_LOOP

This algorithm performs a block-nested-loop join where the inner input is scanned once for each page of the outer input. The memory is completely filled with the outer input except from one page that is reserved for the inner input. In addition, scans of the inner input are made a little faster by scanning the inner input once forwards and once backwards, thus reusing the last page of the previous scan [Kim80]. Then,

$$cost = \left\lceil \frac{b_1}{M-1} \right\rceil \times (b_2 - 1) + 1 + b$$

MERGE, SELECT_ALGO, PROJECT, PROJECT_D, SET_DIFF, SET_DIFF_D SET_INTERSECT, SET_INTERSECT_D, SET_UNION, SET_UNION_D

If the output relation can fit in memory, i.e., if $b < M$, then,

$$cost = 0$$

else the result must be written into disk, and

$$cost = b$$

SORT

To sort a relation on a given attribute, the relation is written into initial sorted runs, each about the size of available memory. These runs are merged into larger and larger ones, two at a time, until only one run file, the final output, is produced. The number of initial runs, in this algorithm, is

$$initial\_runs = \lceil b_1/M \rceil$$

The number of merge levels necessary to complete the task is

$$merge\_levels = \lceil \log_2(initial\_runs) \rceil$$

Using a factor of two for reading and writing, then

$$cost = 2 \times b_1 \times merge\_levels$$

DUPLICATE_ELIMINATION

This algorithm is based on sorting to bring duplicates close together. The cost in sort-based duplicate removal is, thus, dominated by the cost of the sorting but it is smaller than it, because of the effect of early duplicate removal on each merge level. The total number of merge levels is unaffected by duplicate removal and is defined in terms of the number of initial runs that the input file is split into. As in the case of sorting, $initial\_runs = \lceil b_1/M \rceil$ and $merge\_levels = \lceil \log_2(initial\_runs) \rceil$.

In the first merge levels, it is unlikely that duplicates of the same tuple are in the same run file, and therefore we can assume that the sizes of run files are unchanged until the last merge levels, where we can assume that each run file has the same size as the final output. The total number of merge levels with run file sizes equal to the output size (the later merge levels) is, according to Graefe [Gra93],

$$affected\_levels = \lceil \log_2(b_1/b) \rceil - 1$$

The merge levels where each run file has the same size as the input is

$$unaffected\_levels = merge\_levels - affected\_levels$$

Using a factor of two for reading and writing, then [Gra93]

$$cost = \left\lceil 2 \times b_1 \times unaffected\_levels + 2 \times b \times \sum_{unaffected\_levels}^{affected\_levels - 1} \frac{initial\_runs}{2^i} \right\rceil$$