Incremental Maintenance for Non-Distributive Aggregate Functions

Themistoklis Palpanas* Richard Sidle Roberta Cochrane Hamid Pirahesh

University of Toronto
10 King's College Rd.
Toronto ON, Canada M5S 3G4
themis@cs.toronto.edu

IBM Almaden Research Center
650 Harry Rd.
San Jose CA, USA 95120-6099
{rsidle, bobbiec, pirahesh}@almaden.ibm.com

Abstract

Incremental view maintenance is a well-known topic that has been addressed in the literature as well as implemented in database products. Yet, incremental refresh has been studied in depth only for a subset of the aggregate functions. In this paper we propose a general incremental maintenance mechanism that applies to all aggregate functions, including those that are not distributive over all operations. This class of functions is of great interest, and includes MIN/MAX, STDDEV, correlation, regression, XML constructor, and user defined functions. We optimize the maintenance of such views in two ways. First, by only recomputing the set of *affected* groups. Second, we extend the incremental infrastructure with work areas to support the maintenance of functions that are algebraic. We further optimize computation when multiple dissimilar aggregate functions are computed in the same view, and for special cases such as the maintenance of MIN/MAX, which are incrementally maintainable over insertions. We also address the important problem of incremental maintenance of views containing super-aggregates, including materialized OLAP cubes. We have implemented our algorithm on a prototype version of IBM DB2 UDB, and an experimental evaluation proves the validity of our approach.

*Work done while the author was visiting IBM Almaden Research Center.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

1 Introduction

Materialized views, or Automatic Summary Tables (ASTs)[1], are increasingly being used to facilitate the analysis of the large amounts of data being collected in relational databases. The use of ASTs can significantly reduce the execution time of a query, often by orders of magnitude, which is particularly significant for databases with sizes in the terabyte to petabyte range, whose queries are designed by Business Intelligence tools or Decision Support Systems. Such queries tend to be extremely complex, involving a large number of join and grouping operations. The focus of this paper is on ASTs defined with aggregate functions, including those defined for OLAP cubes. This class of ASTs is extremely important in practice.

The advantage of ASTs is that they are precomputed once and subsequently used multiple times to quickly answer complex queries. When base relations are modified, these modifications must be propagated to the affected ASTs. Using current techniques, the systems can only incrementally update a restricted set of ASTs; those only containing distributive aggregate functions. The remainder must be fully recomputed. Prior work [LHM+86, CGL+96, BLT86, QW91, Qua96, MQM97, LSPC00], has studied the problem of incremental view maintenance in which all the necessary changes for the AST are computed based only on the modifications to the base table (and the corresponding values in the AST). This process is called *incremental view maintenance*, and many commercial products support it. Due to the complexity of the queries and the magnitude of the data, recomputation of ASTs in large-scale databases is prohibitive. Since the set of updates to the base tables is usually only some small percentage of those tables, incremental maintenance of an AST is usually much quicker than full recomputation. For example, a typical ware-

[1] The term "AST" is used in the IBM DB2 database instead of "materialized view". For the rest of this paper we will use the terms materialized view and AST interchangeably.

house can contain up to 6 years of data. Daily inserts into a fact table in this warehouse may constitute only about 0.05% of the entire size of the table, while an associated AST can grow up to a billion rows.

Figure 1 depicts the process of incrementally maintaining ASTs. When updates occur in the base data,

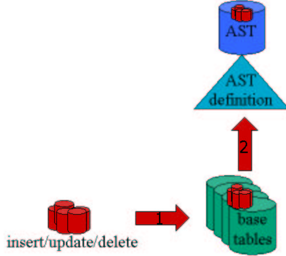

Figure 1: The AST incremental maintenance process.

the system determines which ASTs are affected and propagates the changes through the AST definitions to produce the delta changes. It then applies these deltas to their respective ASTs. If an AST is automatically refreshed in the same unit of work as the changes to the underlying base data are applied, then we say that the maintenance is *immediate*. Otherwise, it is *deferred*. In this paper, we consider the problem of immediate incremental maintenance. A variation of our solution is applicable to deferred maintenance, but is beyond the scope of this paper.

1.1 Classes of Aggregate Functions

All previous studies restrict the aggregate functions allowed in incrementally maintained ASTs. Such functions must have the property that when the underlying tables change, the system can compute the new value of the aggregate function from its old value and the changes themselves, for both insertions and deletions (we can always view updates as a series of deletions and insertions). We call these aggregate functions incrementally computable. To understand the above characterization, we briefly review the classification of aggregate functions over INSERT and DELETE operations.

The behavior of an aggregate function with respect to an operation can be classified into one of three categories [GBLP96]. A function is *distributive* for an operation if the new result of the function can be computed using only the existing value of the aggregate and the values of the operation (new values for insert, old values for delete). SUM and COUNT are distributive for both INSERT and DELETE. MIN and MAX are distributive for INSERT, but not for DELETE. A function is *algebraic* for an operation if the new result of the function, as a result of the operation, can be computed using some small, constant size storage (work area) that accompanies the existing value of the aggregate. AVG and STDDEV are algebraic for INSERT and DELETE. For AVG, the work-area consists of simply the COUNT. In fact, most products and studies include AVG in their set of incrementally computable functions, and for the rest of this paper we take the same position. A function is *holistic* for an operation if there is no constant bound on the amount of storage needed to compute the new result of the function for any instance of the database. This is the case for MIN and MAX over DELETE. They may require revisiting all the records for the affected groups.

Aggregate functions that are distributive over both INSERT and DELETE (henceforth referred to simply as *distributive aggregate functions*) are incrementally computable. This is not true for algebraic and holistic functions. The class of non-distributive functions is of great interest, and is commonly used in practice. It includes MIN, MAX, STDDEV, CORRELATION, REGRESSION functions, XML constructor functions [SQL02], and others. However, the problem of efficiently maintaining views with non-distributive aggregate functions has not been sufficiently studied in the literature.

1.2 Maintaining Non-Distributive Aggregate Functions

In this work we extend the current framework to efficiently support the incremental maintenance of ASTs defined with non-distributive aggregate functions. We present a method for *selectively recomputing* only the affected groups as well as supporting incremental maintenance of distributive aggregate functions that occur in the same view definition as non-distributive aggregate functions. The recomputation step is indispensable since the new value for the non-distributive aggregate functions cannot be derived from the old value and the changes to the base relations alone. We also describe the maintenance of work-areas that make algebraic functions incrementally computable. The algorithm we propose is an efficient way to perform selective recomputation, and to the best of our knowledge, is the first such algorithm proposed in the literature that deals with the intricacies of this problem and accounts for super-aggregates.

Our work focuses on identifying the affected groups of the AST, and efficiently recomputing them. In order to improve performance we apply a series of optimizations on the query plan generated for the maintenance of the affected ASTs, and we make sure that only the non-distributive aggregate functions for the affected groups are recomputed. The rest of the aggregate functions are incrementally maintained and not recomputed. The aforementioned optimizations are specific to the problem at hand, and therefore, could not have been applied by the optimizer module of the database system. We also optimize separately for some special cases, which enables us to save much computational effort. Note that our approach is not confined to any

particular set of functions, but works for arbitrary aggregate functions as well. Our contributions can be summarized as follows.

- We enhance the incremental view maintenance framework with a selective recomputation step that significantly expands the set of supported aggregate functions. The newly supported aggregate functions are non-distributive, and include MIN, MAX, STDDEV, CORRELATION, REGRESSION functions, XML constructor functions, and others. The framework can also accommodate any user-defined aggregate functions.
- We present several optimizations incorporated in the algorithm, which lead to an efficient solution of the problem. We describe the necessary rewrites that improve the execution of the maintenance expression, and we augment the query rewrite rules to handle materialized views with super-aggregates.
- For the aggregate functions that are algebraic for INSERT and DELETE we extend the incremental infrastructure to support materialization and maintenance of sub aggregates.
- We discuss in detail a practical algorithm that can handle a large variety of real-life scenarios. We implemented our method in a prototype version of IBM DB2 UDB, and the experimental evaluation proves the validity of our approach.

The rest of the paper is organized as follows. In Section 2 we give some background necessary for the rest of the paper. In Section 3 we present a detailed discussion of our technique, and in Section 4 we elaborate on the use of work areas for algebraic aggregates. We present an experimental evaluation of our method in Section 5, in Section 6 we review the related work, and finally we conclude in Section 7.

2 Background

2.1 Notation

The notation we are going to use to illustrate how an SQL query changes in each step of the algorithm is based on the Query Graph Model (QGM) [HFLP89], which is a structural representation of SQL statements. The choice of QGM only helps in the presentation of the material and in no case does it affect the generality of our solutions.

At a high level description, a QGM graph consists of (rectangular) boxes, and edges between the boxes. Each box implements one or more relational operators on its input columns, and also specifies the output columns. The edges merely denote the flow of tuples from the output columns of one box to the input columns of another. For the rest of this paper we will use the term "QGM" to refer both to the model and a given instance graph of the model.

For illustration purposes in this paper we use a simplified form of the QGM representation. We are only interested in the way SQL Update, Delete, and Insert statements (*UDI statements*) are transformed along the various steps of the incremental maintenance compilation algorithm. Therefore, we ignore the detailed representation of the operators inside each QGM box. For clarity of presentation, we substitute an entire subgraph of the UDI statement QGM graph with a trapezoid box. Finally, we use the cylinder to depict materialized tables (either base relations, or ASTs), whereas a rectangular or trapezoid box for an AST refers to the QGM graph of its definition query.

2.2 Incremental Maintenance for Distributive Aggregate Functions

The work of this paper has been implemented as an extension to the compilation algorithm in IBM DB2 UDB for the incremental maintenance of ASTs. We present an overview of this algorithm here as background to our extensions. It is important to note however, that the techniques we describe in this paper are not specific to a particular database product, but are applicable to any database management system.

The algorithm supports incremental maintenance for a wide range of aggregation queries. It supports aggregation over any combination of select, project, inner join (including correlated joins and self-joins) and union. The super-aggregation operators CUBE, ROLLUP and GROUPING SETS are also supported [LSPC00], as are the distributive aggregate functions SUM and COUNT. The base objects that are referenced by the AST (after view expansion) may be either the underlying tables of the AST query or deterministic table functions.

The restrictions on the queries that define the ASTs are: (a) a unique key must be derivable (i.e. the AST cannot contain duplicate rows), (b) only one level of aggregation is permitted (GROUP BY on GROUP BY is not supported), and (c) the AST query must contain a COUNT(*) function.

The algorithm supports all possible SQL update statements, including those that modify more than one of the underlying tables via cascading referential integrity constraint or trigger actions. The algorithm implements immediate AST maintenance by compiling the constructions needed to maintain the affected ASTs into the QGM graphs of the UDI statements. These constructions employ standard relational operations whose inputs are: (a) the changes to the underlying tables, (b) the post-update images of the underlying tables, and (c) the AST.

As in previous incremental maintenance algorithms [MQM97], this algorithm decomposes incremental computation into two steps, namely *propagate* and *apply*. At execution time, *propagate* computes the set of changes (the *final delta*) that must be made to an AST

given the changes to its underlying tables (the *underlying table deltas*), and *apply* updates the AST with the results from *propagate*. The propagate phase generated by the algorithm uses only the changes to the underlying tables and the post-update images of the underlying tables. Conversely, the apply phase uses only the results of propagate and the AST. This compilation algorithm collects the set of ASTs that depend on the tables that are the targets of the modification operations in the UDI QGM graph. For each dependent AST it then constructs the propagate phase and the apply phase (see Figure 2), for which we provide further details in the following sections.

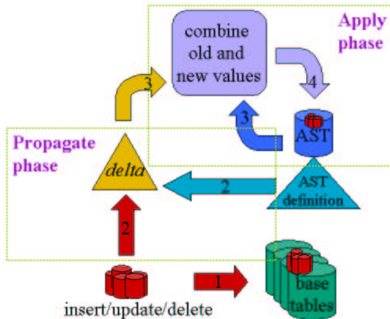

Figure 2: Current incremental maintenance framework.

2.2.1 Propagate Phase Compilation

The algorithm performs a depth-first traversal of the AST QGM graph, and employs compilation rules specific to each box in the graph. The result of each of these rules is a modified QGM for the operation that computes the delta for that portion of the AST query. The propagate phase has rules for underlying table references, select-project, union, inner-join and group-by operations. In the remainder of this section, we describe rules for underlying table references, select-project and union to give the flavor of the algorithm.

Underlying table reference rule: On encountering a reference to an underlying table, we create a union operation of all of the UDI operations in the UDI statement QGM that affect this table reference. If there is exactly one UDI operation that affects the table reference, then a selection operation is constructed instead of a union.

When a union is required, the union may compute a mixture of update, delete and insert rows. The delete rows contain old values, insert rows contain new values and update rows contain both. To distinguish between these rows a tag column is added to the union operation with values -1, +1, and 0, for deletes, inserts and updates respectively. If there is a combination of update operations with delete/insert operations in the resulting union, the delete and insert operands are padded with NULL-valued columns for union compatibility with the update operands, which may contain twice as many columns (since they carry both the old and new values). Table 1 shows the general form of a delta for an operation (or underlying table reference) that returns n columns $C_1, C_2, \ldots, C_n$.

action	**delta** (with tag column)
inserted row	(newC_1, ..., newC_n, +1, null, ..., null)
deleted row	(oldC_1, ..., oldC_n, -1, null, ..., null)
updated row	(oldC_1, ..., oldC_n, 0, newC_1, ..., newC_n)

Table 1: Delta format.

Select-Project Rule: If there is a delta for the input operand, then the select-project operation is modified to propagate the tag column and if applicable, the extra columns for updates. Otherwise, the operation is unchanged.

Union Rule: If none of the operands have deltas, then the union is unchanged. Otherwise, the operands without deltas (union is an n-ary operation in QGM) are pruned from the union. If only one operand remains after pruning, then the operation is transformed into a select-project operation (without predicates) and the corresponding rule is applied. When more than one operand remains, the output tag column is derived from the tag column of all of the operands.

The result of applying each rule to an operation is the modified QGM for the operation and a number of properties characterizing the result of the operation. The properties are required to construct the propagate phase for subsequent operations and to communicate requirements for constructing the apply phase.

2.2.2 Apply Phase Compilation

Apply phase constructs a join between the propagate graph and the AST, on the unique key of the AST (derived by *propagate*). The join computes the rows of the AST that must be modified. If an insert operation is required to apply the final delta, then a left outer join is constructed, where the left operand is the propagate graph. The left outer join is required to preserve rows of the final delta that do not exist in the AST (i.e., rows for new groups). Otherwise, only update and/or delete operations are required to apply the final delta, and an inner join is constructed. When an update is required, expressions are built in the result of the join to compute the new values of the affected aggregate functions by combining the old values from the AST with the corresponding values of the final delta. Finally, the needed AST update, delete and insert operations are constructed above the join. A predicate is built for each one, that defines which UDI operation applies to any given row.

Figure 3 is a high level depiction of the QGM after the construction of *propagate* and *apply*. The box marked "prop" encapsulates the results of *propagate*. The join built by *apply* is marked "LOJ" (a Left Outer Join in general) and the box marked "UDI" shows the UDI operations on the AST constructed by *apply*.

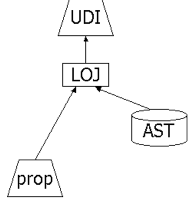

Figure 3: The QGM representation of the augmented UDI statement.

3 Selective Recomputation for Non-Distributive Aggregate Functions

We now present our algorithm for incrementally maintaining ASTs whose definition includes *any* non-distributive aggregate functions (even user defined functions), which is based on selective recomputation. The algorithm we propose is an efficient way to recompute only the affected groups, and is the first such general algorithm proposed in the literature that deals with the intricacies of this problem and takes into account ASTs with super-aggregates.

The new method we propose for supporting incremental AST maintenance is depicted in Figure 4. Step 3 in the figure (arrows labeled with "3"), represent the change in the propagate phase that handles the non-distributive aggregate functions. The delta for the distributive functions is computed as before in step 2. Thus, the propagate phase computes the new values for all aggregate functions. The changes in the apply phase simply involve the construction and manipulation of predicates for the new aggregate functions.

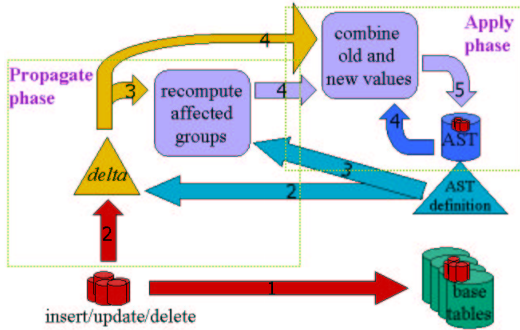

Figure 4: The new incremental maintenance framework.

Our approach can be decomposed into five steps. Each step is a refinement of the previous one, in the sense that it cuts back on the amount of computational effort needed to achieve our goal. Note that this is the description of the algorithm at the logical level, and does not reflect the exact order in which actions are taken when the algorithm is executing. The five steps of our method are the following.

1. Construct a graph, G_{maint}, that computes the new values for each group (i.e., the values after the changes have been applied to the underlying tables) both for the distributive and for the non-distributive functions.
2. Modify G_{maint} to recompute only the non-distributive functions.
3. Modify G_{maint} to recompute only those groups that are affected by the changes.
4. Eliminate any unnecessary operations in G_{maint}.
5. Optimize G_{maint} for special cases.

In the next sections we elaborate on each of the above steps, out of which the first four relate to the propagate phase, and only the last one refers to the apply phase.

For the rest of this paper we will refer to the current approach, namely, incremental maintenance of distributive aggregate functions, as the "old" approach, and to the new method as the "new".

3.1 Compute New Values

In the old setting, it is sufficient to compute the delta and then apply those changes to the AST by matching the corresponding tuples (in the delta and the AST). The new framework must do additional work before applying the changes, because the delta does not provide all the information needed to compute the new values for the non-distributive functions. This is achieved using the definition of the AST, as illustrated in Figure 5 with the lower AST box, which nevertheless, results in recomputing all the groups of the AST. Clearly, this is not desirable and we show how to remove this constraint later on.

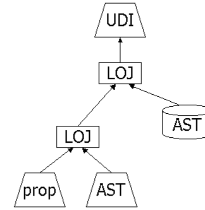

Figure 5: The G_{maint} graph that computes the new aggregate values.

After the recomputation of the groups, we are only interested in keeping the new values for those groups that are affected by the changes to the base relations. It is these values that will produce the new delta. To that effect, we inject a join operation on the unique key columns of the AST query that joins the top of the old propagate phase with the top of the AST graph, so as to get the new values only for the affected groups. We can efficiently identify these groups, because the propagate phase carries all the necessary information, i.e., the changes that took place and which groups they affected. It may be the case that the changes cause some groups to be deleted from the AST. To handle this situation, the join that we construct here is a left outer join, where the result of the propagate phase is the left operand. As we use the post-update image of the underlying tables there will be no matching

group in the right operand for a group that is to be deleted. The left outer join preserves these groups in its result. We will refer to this operation as the "new join". Figure 5 depicts the G_{maint} graph after the above modifications.

3.2 Change Column Derivation

As a result of the previous step of the algorithm, the new values for the distributive aggregate functions are computed twice. Once during the computation of the propagate delta, and a second time during the recomputation of the AST. To eliminate this inefficiency we choose to obtain the new aggregate values from the delta, since the computation of the values for the distributive aggregate functions in the AST is based on the entire set of relevant data in the base relations, and is significantly less efficient than computing the values based only on the changes, which is the case with the delta. Thus, we selectively set the column derivation for each one of the columns in the new join box to come from the AST computation leg if the column is non-distributive, and from the propagate phase leg for all the other cases, as shown in Figure 6.

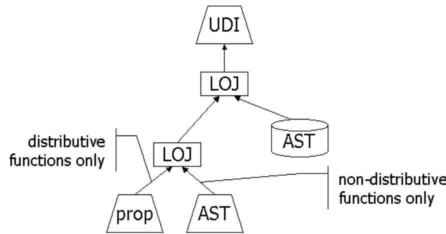

Figure 6: The G_{maint} graph after changing the column derivations.

At this point we have modified the join operation to choose one operand as the source for each of the aggregate function columns in its result. The remaining unreferenced aggregate function columns of each operand will be removed later during the normal course of query optimization.

Example 1 *Consider* AST-1 *defined as follows.*

```
SELECT dept_id, COUNT(emp_id), MAX(age), STDDEV(salary)
FROM employees
GROUP BY dept_id
```

The result of the `COUNT(emp_id)` *function (distributive) will be determined from the old propagate phase, while* `MAX(age)` *and* `STDDEV(salary)` *(non-distributive) will be computed from the AST definition. But none of them will be computed twice.*

3.3 Recompute Only Affected Groups

So far we do not avoid recomputing the entire AST (minus the evaluation of the distributive aggregate functions, because of the previous step). This step pushes the new join predicate down to the lowest operations of the AST query graph. Note that the pushed down join need only be an inner join. We employ the query rewrite engine of the DBMS to push down the predicate. If this pushdown is successful then the join will be applied between the top of the old propagate phase and the underlying tables of the AST. We are now able to select from the AST's underlying tables only those values that contribute to the affected groups (because at the top of the propagate phase we already know which groups are affected), and consequently, it is only the affected groups that will be recomputed. The G_{maint} graph after the pushdown is depicted in Figure 7. In this figure, we use $T_1, ..., T_k$ to represent

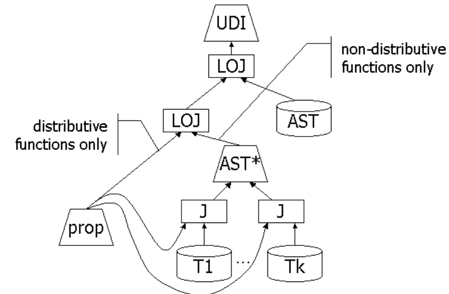

Figure 7: The G_{maint} that will selectively recompute only the affected groups.

the base relations over which the AST is defined. By "AST*" we denote the QGM graph of the AST after removing the references to the underlying tables $T_1, ..., T_k$.

The functionality described above is a prime component of the incremental maintenance procedure we propose, and defines the class of ASTs that our method supports. First, we require that there exists a key that uniquely identifies the tuples in the AST. Second, given a predicate on the aforementioned unique key, we require that it is possible to push this predicate down through the AST QGM graph to the underlying tables of the AST. If these two requirements are met then the AST is accepted as incrementally maintainable, since the selective recomputation step can be efficiently supported.

The basic predicate pushdown engine does not support predicate pushdown through super-aggregates. In the next section we present extensions that support this important class of warehouse ASTs.

3.4 Super-Aggregate Predicate Pushdown

When the AST involves a super-aggregate, we must devise special predicate pushdown rules to avoid erroneous results. In order to describe the issues and our algorithm, we first give a brief overview of super-aggregates. A super-aggregate is a SQL language extension to the group-by clause [GBLP96] that supports the computation of measures for different levels of a

hierarchy. The language extensions are very versatile, and allow the specification of all of the following cases:

- Rollup of Dimension Hierarchies:
  `GROUP BY ROLLUP(year,month,day)`
- Simple Cubes: `GROUP BY CUBE(year,product,region)`
- OLAP-cubes:`GROUP BY year,month,ROLLUP(prodline, prodgroup,product),ROLLUP(country,state,city)`
  This example specifies a hierarchical cube per month.
- Sparse Cubes: `GROUP BY GROUPING SETS((year, prodline),(year,country),(year))`

The result of a super-aggregate is a table that contains the union of many simple group-by operations, not all of which contain the same set of grouping columns. In the resulting table one can identify which rows belong to each simple group-by. We refer to a column in the group-by clause of a super-aggregate as a *dimension column*. If the dimension column is not nullable, then a NULL value for this column indicates that it is not one of the grouping columns for a given row. For nullable columns, SQL provides a "GROUPING" function whose value is 0 when the dimension is one of the grouping columns and 1 when it is not.

To satisfy our unique key requirement for incremental maintenance, a dimension column in a super-aggregate AST must either be non-nullable or contain a corresponding *indicator column* that computes the GROUPING function for the dimension. In the following description, we will assume that all dimension attributes are nullable, and rely solely on the existence of the corresponding indicator columns. Note that all of the expressions can be simplified with "IS NULL" tests when the column is non-nullable.

When the AST involves a super-aggregate operation we alter the predicate pushdown procedure as follows. First, recall that the predicate formed between the propagate phase and the AST query is a join on the unique key. In the case of super-aggregates, the unique key contains indicator columns. Prior to pushing this join predicate down, we remove terms containing these indicator columns since they are manufactured during the computation of the super-aggregate and are not available from the base tables.

The second alteration is more fundamental and part of the pushdown procedure itself. We must take care to appropriately recompute all of the higher-level aggregations. Like the super-aggregate itself, the propagate phase contains deltas for many simple group-by operations. It contains a row for each affected row of the AST. Consider the OLAP-cube: GROUP BY year, month, ROLLUP(prodline, prodgroup, product), ROLLUP(country, state, city), and a modification that inserts sales data for Waynesboro, VA for June, 2002. The propagate phase contains deltas for all groups that must be updated in the AST: these are the June, 2002 aggregates per each level of the product-dimension for the city of Waynesboro, the state of VA, the entire country, and for all countries.

Pushing predicates through a grouping expression that contains a super-aggregate must alter the predicate in such a way as to appropriately recompute the aggregate values for each of the affected groups. If the super-aggregate computes values for different levels in a dimension hierarchy (e.g. state and country subtotals), the results of propagate will contain rows for each level in the hierarchy that must be modified. The predicate pushdown must ensure that subtotals for higher levels in the hierarchy do not double count the contributions from the lower levels. For example, insertions of rows for Waynesboro, VA and Staunton, VA could inadvertently cause a duplication of the values for VA. Furthermore, we must ensure that subtotals for higher levels of the hierarchy revisit all contributing rows from the base tables when necessary.

To this effect, a pushdown through a super-aggregate will mark the predicate as a super-aggregate predicate. When such a predicate encounters a simple group-by operation, we construct a new predicate as shown in Figure 8.

Let A be a dimension column, and $g(A)$ the corresponding indicator column.

```
for each A in the simple group-by:
  preserve all terms in the predicate containing A
  add term g(A) = 0 to the predicate
for each A not in the simple group-by:
  remove all terms in the predicate containing A
  add term g(A) = 1 to the predicate
```

Figure 8: New predicate construction.

Example 2 *Consider the sparse cube:* `GROUP BY GROUPING SETS((year,prodline),(year,country), (year))`. *It contains the union of three simple group-bys. Let D represent the results of the propagate, and t be the input to the super-aggregate group-by operation. The predicate pushdown will break down into 3 predicates, one for each group-by as shown in Table 2.*

Recall that we are working with non-distributive functions, and in general, computations of sub-totals cannot be combined to compute higher-level aggregates. Although our approach requires recomputing higher-level aggregates from scratch, it eliminates the computation of the subgroups that are unaffected. These savings can be significant in the case of non-distributive aggregate functions. In our example, we need only recompute the aggregates for the cities of Waynesboro and Staunton, the state of VA, and for USA. Aggregate results for other cities, states and countries are not recomputed.

3.5 Eliminate Unnecessary Operations

We identify two distinct categories of changes to the base data. The first category includes changes that involve deletions. Deletions occur not only when the changes explicitly specify some tuples deleted, but also when values in the base relations are updated. For the

Group-by	Altered predicate
(year, prodline)	g(D.year) = 0 AND g(D.prodline) = 0 AND g(D.country) = 1 AND D.year = t.year AND D.prodline = t.prodline
(year, country)	g(D.year) = 0 AND g(D.prodline) = 1 AND g(D.country) = 0 AND D.year = t.year AND D.country = t.country
(year)	g(D.year) = 0 AND g(D.prodline) = 1 AND g(D.country) = 1 AND D.year = t.year

Table 2: Predicates resulting from pushdown through super-aggregate.

latter scenario to be true, assume an AST with an aggregate group consisting of a single tuple. Then, an update statement that changes the grouping values of this tuple will result in the deletion of the group from the AST. The second category includes changes that involve no deletions from the AST, either explicit or implicit. This translates to either allowing insertions to the underlying tables, or updates that do not affect the grouping expressions. When the changes fall under this category we call them *deletion-free.* The deletion-free case is an important class of changes. It is very common in real life applications, where it represents the accumulation of new data in data warehouses.

When the changes are deletion-free there is no reason for the new join to be an outerjoin, whose only purpose is to capture the cases where some groups of the AST are deleted. Therefore, we transform the new join into an inner join operation in order to speedup the process. In addition, we observe that even an inner join operation does not serve a real purpose, other than propagating the incrementally maintainable columns (coming from the old propagate phase) up in the G_{maint} graph. This is apparent by looking at Figure 7, and substituting the lower outerjoin operator by an inner join. It turns out that we are able to

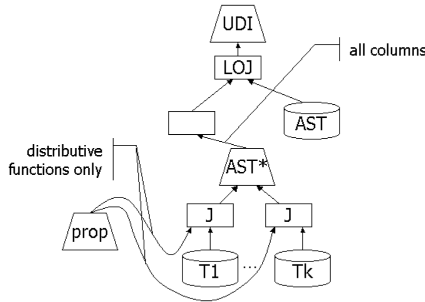

Figure 9: The G_{maint} graph after removing the new join operation.

reroute all the columns going from the old propagate phase to the new join through the pushed down join, up the AST QGM graph, and finally back into the new join box, as depicted in Figure 9. By doing that we completely eliminate the need for the new join. Thus, we remove it from the QGM graph, saving an expensive operation. Note that by rerouting the columns we are not introducing additional grouping columns for the aggregations computed in the AST, because the rerouted columns are functionally determined by the grouping expressions. Therefore, no extra computation will take place for them when they go through the grouping operators. Note that this optimization applies to super-aggregates as well.

Another interesting scenario that we optimize are updates to the underlying tables that do not affect the group-by columns or predicates of the AST. In this case there is no need to decompose the update operations into deletions and insertions. The apply phase detects that only updates are in the data flow, and consequently builds clauses that update only those aggregate functions of the AST which are affected by the changes to the underlying tables. The remaining aggregate function columns that are unaffected, and not referenced in the apply phase, will be removed from the query plan later during query optimization, and will not be recomputed.

Example 3 *Consider* AST-1 *defined in Example 1. Then, the following modification to the* employees *table will* not *cause the outerjoin operation to be built.*

```
UPDATE employees SET salary = 10 WHERE age > 40
```

This is because the update operation will not be decomposed into deletions and insertions, since the update does not refer to the grouping expression (`dept_id`*) and no predicate in* AST-1 *refers to the updated column (*`salary`*). Therefore it falls under the category of* deletion-free *changes. Furthermore,* only *the function* `STDDEV(salary)` *will be recomputed, since the specified changes do not affect the other aggregate functions.*

3.6 Optimize for Special Cases

We now describe two special cases, for which we are able to avoid the selective recomputation step altogether. We notice that when the changes are only insertions, and the AST involves only the MIN/MAX functions from the class of non-distributive aggregate functions, then there is no reason to recompute the affected groups. Indeed, in this case it suffices to build a predicate that at the apply phase time will check whether the new MIN/MAX value should replace the old one. The rules for building this predicate are straightforward and are given in Table 3. Under these circumstances the G_{maint} graph becomes very simple as shown in Figure 10. In fact the graph is virtually identical to the case when there are no non-distributive aggregate functions present (Figure 3). The only difference is the additional predicate in the apply phase that updates the values for the MIN/MAX functions.

The second case handles changes that are only deletions, and furthermore, the delete predicates only refer to the grouping expressions of the AST. Clearly, this

conditions for MIN	conditions for MAX
if (oldMIN is NULL) then newValue elseif (newValue<oldMIN) then newValue else oldMIN	if (oldMAX is NULL) then newValue elseif (newValue>oldMAX) then newValue else oldMAX

Table 3: Apply phase predicates for MIN and MAX.

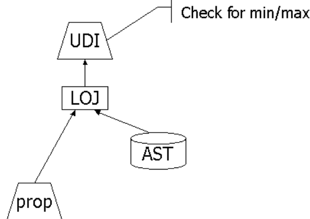

Figure 10: The G_{maint} graph for updates involving insertions and the MIN/MAX aggregate functions.

situation can only result in one or more of the groups of the AST to be deleted in their entirety. Thus, there is no need to recompute any of the aggregate functions for these groups. The selective recomputation step is once again not necessary, and we do not construct it at all. As we show in the experimental evaluation, this optimization can have tremendous performance benefits.

Example 4 *Consider* AST-1 *defined in Example 1. The following modification will* not *cause the selective recomputation step to be constructed.*

```
DELETE FROM employees WHERE dept_id > 40
```

This is because the above operation will result in entire groups to be deleted from AST-1 (those with `dept_id>40`*), and will not change any other group.*

4 Using Work Areas

Our focus in this paper thus far has been on the incremental maintenance of materialized views, using selective recomputation for queries containing non-distributive aggregate functions. This technique is applicable to all aggregate functions, and we have explored how to apply it efficiently.

However, there is a class of non-distributive functions that can be further optimized by maintaining a summary of sub-aggregation in the materialized view that is distributive, and hence incrementally maintainable. Such functions are classified as *algebraic*, and many important aggregate functions fall into this category. We refer to the maintained sub-aggregates as *work areas*. The maintenance of algebraic functions is optimized by incrementally maintaining the information in the work area and computing the resulting aggregate function of the query from the work area. Some standard SQL functions that are algebraic are AVG, CORRELATION, COVARIANCE, the REGRESSION functions, STDDEV and VARIANCE.

Note that MIN and MAX are not algebraic. However, it is possible to reduce the frequency of recomputation using work areas, by recording the bottom (top) N values in the work area each time the recomputation is performed. Deleting tuples from the materialized view may cause some of these values stored in the work area to be removed as well. When the last value is deleted from the work area for a group, the function must be recomputed for that group. Such work areas must also be maintained for insertions, which can be done by computing the delta for the work area from the insertions and merging with the existing value for the work area in the materialized view.

Queries containing the aforementioned functions can be evaluated in parallel using work areas [SN95], which is precisely what is done in the DB2 MPP (shared nothing parallelism) system. A computation is performed per data partition in parallel to produce a work area. The work areas from each partition are then combined into a final work area, and the aggregate function is computed from the final work area.

We apply the same algorithms to incrementally maintain these functions for insertions, and similar algorithms can also be used for deletions. The final work area for each of these functions must be kept in the materialized view as an additional, hidden attribute. Given any function from the above list, the incremental maintenance proceeds as follows. The propagate phase of incremental maintenance computes two work areas for the function per affected group: one for insertions and the other for deletions. Parallelism is fully exploited in this phase. For each affected group, the apply phase first combines the work area computed from the insertions with the work area in the materialized view for the corresponding group, in order to compute an intermediate result work area. Then, the work area computed from the deletions is combined with this intermediate result work area to compute the final new work area and new function value to be stored in the materialized view.

5 Experimental Evaluation

In order to evaluate our framework for selective recomputation, we implemented our technique in a prototype version of the IBM DB2 UDB DBMS. The hardware platform is an RS/6000 44P model 270, with 2-way 64bit Power-3 processors, and 1GB of RAM.

In the following experiments we use a star schema (see Figure 11) data warehouse that stores information about products and sales for a period of 5 years. In the figure we only show the part of the schema relevant to our experiments. The fact table (*transitems*) contains 10 million rows of sales data. This data was generated randomly over the products, product groups, and locations. The product dimension contains 5 product

groups and 2,000 products, of which 1,000 were sold in this 5 year time period. The location dimension contains data for 1,000 stores. The *transitems* table includes attributes describing the product sold, the location of the sale, the quantity and price, and the time of the sale.

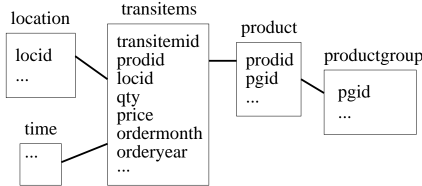

Figure 11: The schema of the database.

We create two ASTs (see Table 4), one with distributive aggregate functions and one with a non-distributive aggregate function. The first, *AST-d*, is a summary of sales. It contains aggregate functions that are distributive over insertions and deletions. The second, *AST-nd*, determines the relationship of sales quantity to product price. It computes the linear regression slope for a 2-dimensional set of points, which is computed by a non-distributive aggregate function. Both ASTs summarize data by product group, location, year and month. They have a cardinality on the order of 240,000 tuples. In our experiments, the first AST is maintained purely incrementally, whereas the second utilizes selective recomputation.

AST-d: distributive AST

```
CREATE TABLE sales_summary AS
  (SELECT p.pgid, t.locid, t.orderyear AS year,
          t.ordermonth AS month, COUNT(*) AS count
          SUM(t.qty * t.price) AS sales,
   FROM transitems AS t, product AS p
   WHERE  t.prodid = p.prodid
   GROUP BY pgid, locid, orderyear, ordermonth)
```

AST-nd: non-distributive AST

```
CREATE TABLE sales_summary AS
  (SELECT p.pgid, t.locid, t.orderyear AS year,
          t.ordermonth AS month, COUNT(*) AS count
          REGR_SLOPE(t.qty, t.price) AS qtyonprice,
   FROM transitems AS t, product AS p
   WHERE t.prodid = p.prodid
   GROUP BY pgid, locid, orderyear, ordermonth)
```

Table 4: The definitions of the two ASTs.

In the experiments we measure the cost of maintaining the two ASTs. Incremental maintenance (with and without selective recomputation) is compared with full refresh of the ASTs. In all cases we report only the elapsed time (in seconds) required for the maintenance of the ASTs, excluding the time needed for the update of the underlying tables.

Our performance scenarios model nightly updates to the warehouse. The cost of adding and deleting one day's worth of sales data to the fact table is evaluated using two different scenarios. The first scenario corresponds to adding data for the first day of the month, which results in adding groups for the new month to the ASTs. Conversely, deleting this day's worth of data results in deleting all such groups. The second scenario adds and deletes sales data for the second day of the month, resulting primarily in updates of existing groups with possible insertion of new groups. Finally, we experiment with the addition and deletion of a full month of sales data.

Sales data is added using set-oriented insertions. Each day's worth of data is collected in a staging table *transitems_delta* and applied using an insert statement (*ins* workload). We also test two different delete workloads that demonstrate our delete optimization described in Section 3.6. The two queries, which are shown in Table 5, perform exactly the same modifications. The second query, *del-opt*, contains a predicate on year and month, from which we are able to prove that full groups will be deleted from the ASTs. The first query, *del*, does not have such a predicate. Instead the *transitems_delta* table in this case contains only the data for the aforementioned year and month, implicitly satisfying the same predicates.

example for **del** workload

```
DELETE FROM transitems WHERE transitemid IN
   (SELECT transitemid FROM transitems_delta)
```

example for **del-opt** workload

```
DELETE FROM transitems
WHERE orderyear = 2002 AND ordermonth = 1
```

Table 5: Examples for the deletion workloads.

Table 6 depicts the results for the first day workload, which results in 5,481 new/deleted rows in the *transitems* table (0.05%), 2,348 new/deleted groups in the ASTs, and no updates to existing groups. Note

	ins	**del**	**del-opt**
incr. AST-nd	**151**	**286**	**3**
full AST-nd	**702**	**699**	
incr. AST-d	2	2	3
full AST-d	779	757	

Table 6: Elapsed time for the 1st day workload (secs).

that our algorithm is able to produce an optimized plan for the *del-opt* workload, which leads to significant performance improvements. In Table 7 we show the results for the second day workload, which results in 5,481 new/deleted rows in the *transitems* table (0.05%), no new/deleted groups in the ASTs, and 2,348 updates to existing groups. In this case the *del-opt* workload is not applicable since it is not possible

	ins	**del**	**del-opt**
incr. AST-nd	**158**	**294**	N/A
full AST-nd	**702**	**702**	
incr. AST-d	4	2	N/A
full AST-d	783	780	

Table 7: Elapsed time for the 2nd day workload (secs).

to optimize the deletion of only the second day of the month if the AST contains only year and month. Table 8 presents the results for the full month workload, which results in 166,667 new/deleted rows in the *transitems* table (1.7%), 3,994 new/deleted groups in the ASTs, and no updates to existing groups. In all cases

	ins	**del**	**del-opt**
incr. AST-nd	**180**	**420**	**31**
full AST-nd	**721**	**692**	
incr. AST-d	7	200	31
full AST-d	809	762	

Table 8: Elapsed time for the month workload (secs).

of the *AST-nd* maintenance, the performance for the *del* workload is noticeably worse than that of the *ins*. This is due to the outerjoin operation that is necessary in order to identify the deleted groups, which is only built when deletions are present (see Section 3.5). Note that the numbers for *AST-d* are presented to provide a baseline. These numbers should not be compared with the *AST-nd* numbers, as *AST-d* is for distributive aggregate functions and *AST-nd* is for non-distributive ones.

The experiments show that our method runs in 20%-60% of the time required by full refresh, which is the only available alternative. Note that for the insertion workloads, which are more common in real-life situations, the corresponding numbers are 20%-25%, or 4-5 times faster than full refresh. In addition, there are still cases which our method identifies and optimizes by avoiding the selective recomputation step. This holds for changes in the underlying tables that result in the deletion of entire groups. The above situation, represented by the *del-opt* workload, results in running times 1%-4% of the time required by full refresh. The performance for maintaining ASTs with distributive and ASTs with non-distributive aggregate functions is in this case identical. The same is also true for the maintenance of MIN/MAX over insertions.

6 Related Work

Much research has been devoted to the problem of incremental view maintenance. A differential refresh algorithm is described by Lindsay et al. [LHM+86] for views restricted to selection and projection of a single base table. Colby et al. [CGL+96] discuss an algorithm for deferred incremental maintenance. Salem et al. [SBCL00] propose a technique that allows the system to control the resources dedicated to view maintenance. The study of Blakeley et al. [BCL86] presents sufficient and necessary conditions for detecting when an update of a base table cannot affect a materialized view, and when a view can be incrementally maintained, for SPJ views. The affected views can then be updated using a differential algorithm [BLT86, QW91]. Mumick et al. [MQM97] describe the *summary-delta table* method, which first computes a summary of all the changes, and then applies the summarized changes to the view. The same model is followed in a subsequent study [LSPC00].

Ceri and Widom [CW91] propose the use of the trigger mechanism of the database system for the maintenance of the views, but do not handle aggregates. Griffin and Libkin [GL95] describe an algorithm for deriving incremental view maintenance expressions for views with duplicates, but do not consider aggregate operators. Gupta et al. [GMS93] consider a more general class of views. However, the algorithms do not handle updates that cause new tuples to be inserted in, or deleted from the result of the aggregate operators. Moreover, they do not handle non-distributive aggregate functions either. A work that generalizes the maintenance expressions to additionally handle views with a GROUP BY operator, as well as the MIN and MAX aggregate functions, is presented by Quass [Qua96]. In a sense, our work is taking the aforementioned approach a step further by exploring the characteristics and the intricacies of the selective recomputation procedure, and by providing a general and efficient solution.

In some cases, given a materialized view V, it is beneficial to materialize an additional set of views, called *auxiliary views*, that will help in the incremental maintenance of V [RSS96, QGMW96, SKM99]. A subsequent study [MK00] extends the above work by considering any view definition based on relational algebraic operators and the aggregate operators. Nevertheless, this approach becomes inefficient for the non-distributive aggregate functions.

Other related work includes rendering the optimizer aware of the incremental view maintenance process [Vis98, MRSR01], dealing with the case where the views and the base data are decoupled [ZGMHW95], and maintaining views under structural changes [GMRR01].

7 Conclusions

Incremental view maintenance is an extremely important aspect of the modern database management systems. It enables the fast execution of complex queries without sacrificing the freshness of the data. However, the maintenance of views defined with non-distributive aggregate functions was not sufficiently explored.

In this paper we describe an efficient method for maintaining materialized views with non-distributive aggregate functions, even in the presence of super-aggregates. This class of functions is of great interest, and includes MIN, MAX, STDDEV, CORRELATION, REGRESSION functions, XML constructor functions, as well as any user-defined functions. In order to support the aforementioned functions we need to recompute some portions of the materialized views, and our work provides an efficient way for achieving

this goal. We perform a series of optimizations on the query plan generated for the maintenance of the affected views, which result in better performance. We also optimize separately for some special cases, which enables us to avoid applying some steps of the algorithm, thus saving much computational effort. Note that our algorithm applies naturally to deferred incremental maintenance as well, since the propagate phase does not access the AST and the apply phase does not access any underlying tables or the deltas.

We experimentally evaluated our technique by simulating a data warehouse environment. The experiments show that incremental maintenance of views defined with non-distributive aggregate functions using selective recomputation is a viable and promising solution, offering significant performance improvements compared to full refresh.

References

[BCL86] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. In *VLDB International Conference*, pages 457–466, Kyoto, Japan, August 1986.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently Updating Materialized Views. In *ACM SIGMOD International Conference*, pages 61–71, Washington, D.C., USA, May 1986.

[CGL+96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for Deferred View Maintenance. In *ACM SIGMOD International Conference*, pages 469–480, Montreal, QC, Canada, June 1996.

[CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB International Conference*, pages 577–589, Barcelona, Spain, September 1991.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *International Conference on Data Engineering*, pages 152–159, New Orleans, LO, USA, March 1996.

[GL95] Timothy Griffin and Leonid Libkin. Incremental Maintenance of Views with Duplicates. In *ACM SIGMOD International Conference*, pages 328–339, San Jose, CA, USA, May 1995.

[GMRR01] Ashish Gupta, Inderpal S. Mumick, Jun Rao, and Kenneth A. Ross. Adapting materialized views after redefinitions: Techniques and a performance study. *Information Systems*, 26(5):323–362, July 2001.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *ACM SIGMOD International Conference*, pages 157–166, Washington, D.C., USA, May 1993.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible Query Processing in Starburst. In *ACM SIGMOD International Conference*, pages 377–388, Portland, OR, USA, May 1989.

[LHM+86] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Hamid Pirahesh, and Paul F. Wilms. A Snapshot Differential Refresh Algorithm. In *ACM SIGMOD International Conference*, pages 53–60, Washington, D.C., USA, May 1986.

[LSPC00] Wolfgang Lehner, Richard Sidle, Hamid Pirahesh, and Roberta Cochrane. Maintenance of Cube Automatic Summary Tables. In *ACM SIGMOD International Conference*, pages 512–513, Dallas, TX, USA, May 2000.

[MK00] Mukesh K. Mohania and Yahiko Kambayashi. Making aggregate views self-maintainable. *Data & Knowledge Engineering*, 32(1):87–109, January 2000.

[MQM97] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *ACM SIGMOD International Conference*, pages 100–111, Tuscon, AZ, USA, June 1997.

[MRSR01] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *ACM SIGMOD International Conference*, pages 307–318, Santa Barbara, CA, USA, May 2001.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing (Extended Abstract). In *International Conference on Parallel and Distributed Information Systems*, pages 158–169, Miami Beach, FL, USA, December 1996.

[Qua96] Dallan Quass. Maintenance Expressions for Views with Aggregation. In *VIEWS*, pages 110–118, Montreal, QC, Canada, June 1996.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[RSS96] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *ACM SIGMOD International Conference*, pages 447–458, Montreal, QC, Canada, June 1996.

[SBCL00] Kenneth Salem, Kevin S. Beyer, Roberta Cochrane, and Bruce G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *ACM SIGMOD International Conference*, pages 129–140, Dallas, TX, USA, May 2000.

[SKM99] Sunil Samtani, Vijay Kumar, and Mukesh K. Mohania. Self Maintenance of Multiple Views in Data Warehousing). In *ACM International Conference on Information and Knowledge Management*, pages 292–299, Kansas City, MI, USA, November 1999.

[SN95] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive Parallel Aggregation Algorithms. In *ACM SIGMOD International Conference*, pages 104–114, San Jose, CA, USA, May 1995.

[SQL02] SQLX Group for SQL/XML Specifications Document. http://www.sqlx.org/, 2002.

[Vis98] Dimitra Vista. Integration of Incremental View Maintenance into Query Optimizers. In *International Conference on Extending Database Technology*, pages 374–388, Valencia, Spain, March 1998.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *ACM SIGMOD International Conference*, pages 316–327, San Jose, CA, USA, May 1995.