Dirk Habich, Wolfgang Lehner, Michael Just

Materialized Views in the Presence of Reporting Functions

Erstveröffentlichung in / First published in:
18th International Conference on Scientific and Statistical Database Management (SSDBM'06). Vienna, 03.-05.07.2006. IEEE, S. 159-169. ISBN 0-7695-2590-3

DOI: https://doi.org/10.1109/SSDBM.2006.32

Final edited form was published in "18th International Conference on Scientific and Statistical Database Management (SSDBM'06), Vienna, 2006".S. 159-169. ISBN 0-7695-2590-3.
https://doi.org/10.1109/SSDBM.2006.32

Materialized Views in the Presence of Reporting Functions

Dirk Habich, Wolfgang Lehner, and Michael Just
Dresden University of Technology
Database Technology Group
01062 Dresden, Germany
dbgroup@mail.inf.tu-dresden.de

Abstract

Materialized views are a well-known optimization strategy with the potential for massive improvements in query processing time, especially for aggregation queries over large tables. To realize this potential, the query optimizer has to know how and when to exploit materialized views. Reporting functions represent a novel technique to formulate sequence-oriented queries in SQL. They provide a column-wise ordering, partitioning, and windowing mechanism for aggregation functions and therefore extend the well-known way of grouping and applying simple aggregation functions. Up to now, current work has not considered the frequently used reporting functions in data warehouse environments. In this paper, we introduce materialized reporting function views and show how to rewrite queries with reporting functions as well as aggregation queries to this new kind of materialized views. We demonstrate the efficiency of our approach with a large number of experiments.

1 Introduction

Due to the tremendous increase in the amount of data efficiently managed by current database systems, optimization is still one of the most challenging issues in database research. From a structural perspective, database optimization can be done on two different levels. On the physical level, highly specialized index structures and other methods like partitioning support the access to individual records or restrict the search space (e.g. partition pruning). On the logical level, [10] already introduced the notion of 'logical access paths,' nowadays well-known under the concept of materialized views [11]. The general idea of materialized views (MVs) is as simple as it is beneficial for query optimization: pre-compute certain query fragments once and then re-use them during query runtime; this process is of special interest for aggregation queries.

In addition to regular aggregation queries, queries with reporting functions are becoming more and more important, especially in the context of decision support on top of data warehouse systems. Reporting functions basically operate on a single column expression and perform a column-wise sorting (usually according to a time line), a locally defined partitioning of rows, and the execution of regular aggregation functions defined over an either implicitly or explicitly specified window. Listing 1 illustrates an example where the query partitions the LINEITEM table of a TPCD database according to $l\_shipmode$ (AIR, SHIP, etc.) and computes the cumulative sum of $l\_quantity$ with respect to $l\_shipdate$ (order criterion) in each partition. Such a query is used to capture the trend of $l\_quantity$ over time for each single $l\_shipmode$ instance.

```
SELECT l_shipmode, l_shipdate,
       SUM(l_quantity) OVER(
         PARTITION BY l_shipmode ORDER BY l_shipdate)
FROM TPCD.LINEITEM
```

Listing 1. Example Query with Reporting Function

Although this type of query is being used more and more often—especially by SQL generators of commercial OLAP tools—there is no known support for queries with reporting functions exploiting MVs.

Contribution of This Paper:

In this paper, we extend the way of dealing with MVs to a completely new class of queries in a seamless way so that regular aggregation queries as well as queries with reporting functions are able to exploit this new class of materialized views and benefit from a performance speedup. The main target of our optimization strategy proposed in this paper is to eleminate or to reduce sort operations. Therefore, we discuss whether there is a possibility and a benefit of using

MVs in the context of reporting functions. In detail, we discuss the following issues:

- Does it make sense to define materialized views with reporting functions? This question is particularly interesting because—in comparison to simple aggregation queries—reporting function queries enable a column-wise ordering, partitioning and window-mechanism for aggregation functions.
- How can we describe the existence of reporting functions in relational operator trees? This perspective is a necessary prerequisite for query rewrite patterns.
- How can we exploit materialized views with reporting functions? We will discuss this issue in very detail from multiple perspectives starting with simple query patterns and then develop more sophisticated derivation strategies.
- What are further research issues? We will present some ideas of our ongoing work.

Organization of This Paper:

The paper starts with a discussion of related work in the context of MV support to clarify the need for supporting reporting function views. Thereafter, we outline the necessary formal prerequisite for the subsequent derivation rules described in Section 4. In Section 5, we present a large number experiments demonstrating the performance benefit when using MVs with reporting functions. The paper closes with a discussion of our ongoing work and a conclusion.

2 Related Work

Using materialized views to speed up query processing in data warehouse environments is a well-known optimization strategy. To realize the potential of materialized views, efficient solutions to the following issues are required and have been studied intensively within the research community over the last years:

View design: This issue deals with the selection process of views that are subject to materialization, including the question of how to store and index them [1,5]. In particular, the distinction between *SPJ views* (select-project-join) and *SPJG views* (SPJ with a final group-by) is usually considered because of space constraints. Within the scope of this paper, we will leave this issue aside—an adaptation of existing workload-based selection criteria can be achieved in a straight-forward manner.

View maintenance: Research work in this area provides algorithms to update materialized views when base tables are subject to Update, Insert, and Delete (UDI) operations [6]. Above all, incremental maintenance strategies have been studied intensively. Obviously, the overhead of maintenance has an impact on the selection process. Again, we do not focus on the maintenance problem of MVs with reporting functions because they are either a trivial extension of existing work, described in [7], or far beyond the scope of this paper.

View exploitation: The view exploitation issue deals with the transparent use of materialized views to speed up query processing. Query rewriting techniques for simple aggregation queries were proposed by [2–5, 9, 14]. Simple extensions for sequence-based views are described in [7]. However, we put our focus on this issue and do not propose only an extension of existing work but open the possibility of exploiting the benefit of MVs for a completely different class of queries and type of MVs.

Related work for this paper has to be considered not only with regard to MVs but also from the perspective of sequence processing, which is also an intensively studied problem within the database research community. Different proposals range from special data models like SEQ [12] to the introduction of special sequence operators [13] to complex algorithms like similarity search or pattern recognition that operate in a special sequence-oriented environment. However, we focus strictly on database system technology and propose extensions which might be used within these more application-oriented scenarios.

3 Reporting Functions: SQL and Internal Representation

Before we are able to discuss the benefit of MVs with reporting functions in detail, we outline the SQL semantics and the internal representation which is used throughout the paper and provides a semi-formal way to describe reporting function queries in the context of a relational query graph.

3.1 SQL Representation

Queries with reporting functions are generally executed in three steps. In a first step, all joins, *WHERE*, *GROUP BY* and *HAVING* clauses are performed and the resulting tuple stream is made available for the sequential computation of reporting functions with locally defined ordering and partitioning conditions in the second step. Within a third step, finally, the optional global *ORDER BY* clause is applied to the set of resulting rows. From an optimization point of view, we have to note that ordering is (a) heavily required when executing reporting functions (e.g. sort-merge join, local sorts for reporting functions, global sort) and (b) one of the most expensive operations other than joins and therefore an interesting candidate for query optimization.

From an SQL perspective, reporting functions are defined with the *OVER()* clause (see Figure 1) following a reg-

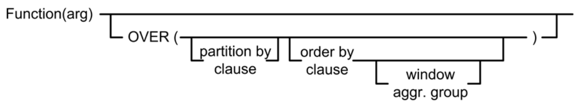


Figure 1. Syntax Diagram for OLAP Functions

ular aggregation function, thus algorithmically declaring the scope of the aggregation function for each single incoming tuple. The query can optionally partition the result set into groups of rows with the *PARTITION BY* clause acting as a reset command for window definitions. These partitions are created on top of the groups defined within the *GROUP BY* clause, so that they are available to any aggregate result from a potential global GROUP BY. Moreover, partitions may be based upon any desired columns or expressions. The *ORDER BY* clause within the *OVER* construct determines the local sort order of the rows within the partitions independent of any global *ORDER BY* clause. Local ordering may be based upon any desired columns or expressions.

Finally, the scope of the aggregation function is determined by a window specification defined either implicitly (cumulative semantics) or explicitly within the OVER clause. The window definition determines the range of rows used to perform the calculations for the currently processed row. Window definitions can be based either on physical number of rows (ROWS) or on logical value-based intervals (RANGE). The window has a starting row $\omega_s(k)$ and an ending row $\omega_e(k)$ relative to the current row k. In general, we can distinguish two different types of window definitions:

1. ***Cumulative Windows:*** A cumulative window definition has its starting row fixed at the first row of the current partition. The ending row slides from the starting point all the way to the last row of the partition. The range of cumulative windows can be algorithmically defined with k as the position of the current row by $\omega_s(k) = 1$ and $\omega_e(k) = k$.

2. ***Sliding Windows:*** A sliding window definition has both its starting and end points slide relatively to the current row and therefore maintain a constant physical or logical range. In the former case, the range of sliding windows can be algorithmically defined by $\omega_s(k) = k - j$ and $\omega_e(k) = k + i$, where i and j are numerical values determining the range.

More formally, we are able to define a reporting function as follows:

Definition 1 (Reporting Function): A reporting function RF is defined by a quadruple ($PBColSet$, $OBColList$, $WSpec$, $AFunc(AColExpr)$), where $PBColSet$ is a set of partitioning attributes, $OBColList$ is a list of ordering attributes, $WSpec$ is a window definition consisting of a definition of the starting and ending point, and $AFunc(AColExpr)$ is a regular aggregation function like SUM, $COUNT$, AVG applied to a column expression. Furthermore, the intersection between the set of partitioning attributes $PBColSet$ and the list of ordering attributes $OBColList$ is empty, i.e. $PBColSet \cap OBColList = \oslash$.

The notion of reporting functions, as it is defined within the SQL standard, does not consider two properties which are crucial for the context of materialized views. First of all, the standard does not enforce the empty intersection property of partitioning and ordering attributes; our restriction is of pure syntactical nature and does not restrict the functionality. The second property addresses the handling of duplicates within the input stream. For example, the query of Listing 1 refers to the LINEITEM table with many duplicates according to $l\_shipmode$ and $l\_shipdate$. Within the SQL standard, duplicates are considered members of the current value-based window definition. The aggregation function is applied to all members of each window and the resulting value is assigned to each input tuple, i.e. every duplicate holds the same value. Therefore, a semantically equivalent expression can be given by explicitly stating the number of original duplicates by additionally requiring the unique property of all partitioning and ordering attributes. Within our approach, we expect exactly this syntactical constraint:

Definition 2 (Reporting Function for MVs): A reporting function within a query defining a materialized view is a reporting function according to Definition 1 with the additional uniqueness property according to $PBColSet \cup OBColSet$.

The query in Listing 2 illustrates this transformation by applying a nested aggregation with regard to the original query given in Listing 1.

```
SELECT l_shipmode, l_shipdate,
  SUM(SUM(l_quantity)) OVER (
    PARTITION BY l_shipmode ORDER BY l_shipdate),
  COUNT(*) as number_of_duplicates
FROM TPCD.LINEITEM
GROUP BY l_shipmode, l_shipdate
```

Listing 2. Example Query with Uniqueness Property

3.2 Internal Representation

Although almost every commercial database system provides (some) support for reporting functions, there is no

Final edited form was published in "18th International Conference on Scientific and Statistical Database Management (SSDBM'06), Vienna, 2006".S. 159-169. ISBN 0-7695-2590-3.
https://doi.org/10.1109/SSDBM.2006.32

published extension of a relational query graph model to capture the required functionality. For the sake of a better illustration of our derivation patterns presented in the following sections, we use the following operators:

- *ACCESS(Pred [ColSet:Cond], RelExpr)*: The ACCESS operator reflects the base operator to read tuples from an underlying tuple producer, i.e. either from a base table or from other relational operators satisfying the given global predicate $Pred$. As a logical extension of the classical database scan operator, our ACCESS-operator checks an additional condition which must be met for each group defined by $ColSet$. For example, the expression $ACCESS([A, B : k = last(C, D)], R)$ returns only the single tuples holding the last position according to the scheme of (C,D) for each valid (A,B)-combination. This can be efficiently achieved by defining a primary/secondary B-tree index structure over the composite index attribute (A,B,C,D). Based on this index structure, a modified search algorithm has to take care of the condition according to the position requirements within each different (A,B)-combination. An index lookup would then return only the RowIDs of the required tuples.

- *PROJ(ColSet:AFunc(AColExpr))*: The PROJ operator eliminates duplicates and optionally computes the aggregation function $AFunc()$ and therefore implements the group-by operator. The PROJ operator requires a preceding SORT operator over the set of grouping columns ($GBColSet$).

- *SORT(ColSet) or SORT(ColList)*: The SORT operator sorts the incoming tuples according to the sorting criterion. In the first case ($ColSet$), the sort order can be determined freely by the optimizer. In the second case, the sort order is fixed and has to be applied to the incoming tuple stream.

- *WINDOW(ColSet:AFunc(AColExpr)[ω_s,ω_e])*: The WINDOW operator, finally, is similar to the PROJ operator with $ColSet$ denoting the partitioning attributes, $AFunc(AColExpr)$ referring to a aggregation function, and [ω_s,ω_e] being the window specification within each partition.

- *JOIN*: Regular logical join operators; the type of join implementation depends on the physical plan.

- *UNPACK(Col)*: The additional UNPACK-operator (similar to [8]) generates duplicates of each incoming tuple. The number of duplicates is retrieved from the values of column Col, which is pruned for the final output.

Figure 2 illustrates the use of the proposed operators. The combination of SORT and PROJ implements the logical group-by operator in $GBColSet$; the combination of SORT and WINDOW represents a reporting function if the partitioning columns ($PBColSet$) build the prefix of the SORT operator followed by the list of explicitly given ordering attributes $OBColList$. Additionally, we are able to model nested aggregations (motivated as described above to eliminate duplicates) by inserting a PROJ operator between the SORT and the WINDOW operator (see Figure 2c).

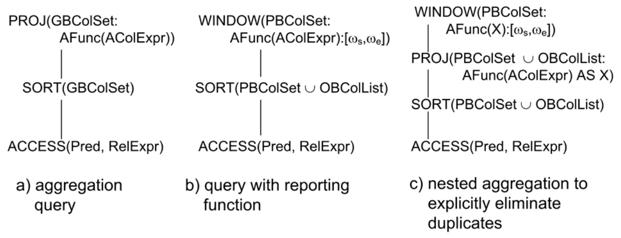

PROJ(GBColSet: AFunc(AColExpr))
SORT(GBColSet)
ACCESS(Pred, RelExpr)
a) aggregation query
WINDOW(PBColSet: AFunc(AColExpr):[ωs,ωe])
SORT(PBColSet ∪ OBColList)
ACCESS(Pred, RelExpr)
b) query with reporting function
WINDOW(PBColSet: AFunc(X):[ωs,ωe])
PROJ(PBColSet ∪ OBColList: AFunc(AColExpr) AS X)
SORT(PBColSet ∪ OBColList)
ACCESS(Pred, RelExpr)
c) nested aggregation to explicitly eliminate duplicates

Figure 2. Examples of Internal Representations

4 Derivability in the Presence of Reporting Functions

Previous work on materialized views has shown that a significant performance gain in query execution can be achieved by utilizing pre-computed data. This section consists of two main components: the first is a brief outline of several well-studied aspects we rely on (Section 4.1) and the second component is a presentation of those aspects we have to extend in order to support reporting function MVs and queries (Sections 4.2 and 4.3).

4.1 Derivability of Aggregation Queries and MVs

The largest benefit of using aggregation-based MVs, i.e. SPJG-MVs, is that only the reduced set of rows, one of each group, has to be stored. In general, the following additional constraints have to be considered.

- *group-by compatibility*: The set of group-by columns of the query Q, $GBColSet_Q$ is equal or a subset of group-by columns $GBColSet_{MV}$ of the MV. The condition also holds for complex grouping expressions like grouping sets, cube, or rollup.

- *join compatibility*: In general, the query and the MV are supposed to have the same set of base tables with the same join predicates. The MV may have additional tables added via lossless joins (N:1 joins); the query may require a back-join to additional tables not being referenced in the MV.

- *selection compatibility*: The selection criterion of the query must be equal or more restrictive with respect to MVs.
- *compatibility of the aggregate function and column expression*: The aggregate function of the query must be compatible (or algebraically computable) with the MV aggregation result.

This list of derivability conditions is included here because all conditions also apply to MVs and queries with reporting functions. We therefore do not explicitly consider these constraints but focus on the additional requirements for this new class of queries and MVs with reporting functions.

4.2 Derivation Patterns for Reporting Function Views Without Partitioning Attributes

The first patterns target MVs with reporting functions defined without any partitioning attributes ($PBColSet_{MV} = \oslash$) and with no explicit window specification, yielding global cumulative aggregate values (usually sum values) according to the given order criterion $OBColList_{MV}$.

Derivation Pattern 1 - Ordering Reduction:

A reporting function query with a reduced order criterion regarding the MV can be answered within this pattern by picking the largest value of the cumulative sum per required sort criterion without considering the omitted ordering attributes. Figure 3 illustrates this pattern: the query only has to take advantage of the conditional ACCESS operator to retrieve the values with the highest index according to the ordering attributes available in the MV but not specified in the query. For brevity, this set of columns $OBColList_{MV} - OBColList_Q$ is denoted as $\overline{OBColList_Q}$.

For example (see Figure 4), with the query and the MV referring to the same set of base tables and sharing a compatible selection criterion, the query expression

```
... SUM(SUM(x)) OVER (ORDER BY A) AS sum_x_a
... GROUP BY A
```

can be derived from an MV expression

```
... SUM(SUM(X)) OVER (ORDER BY A,B) AS sum_x_ab
... GROUP BY A,B
```

by logically rewriting the query to an aggregation query over the materialized reporting function view:

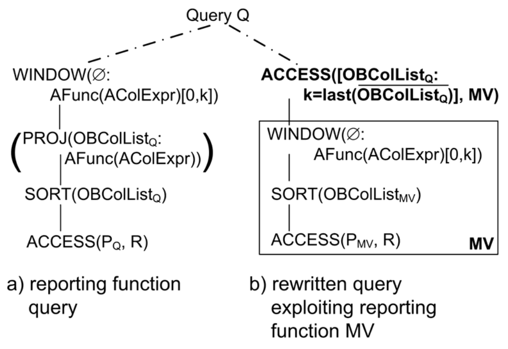
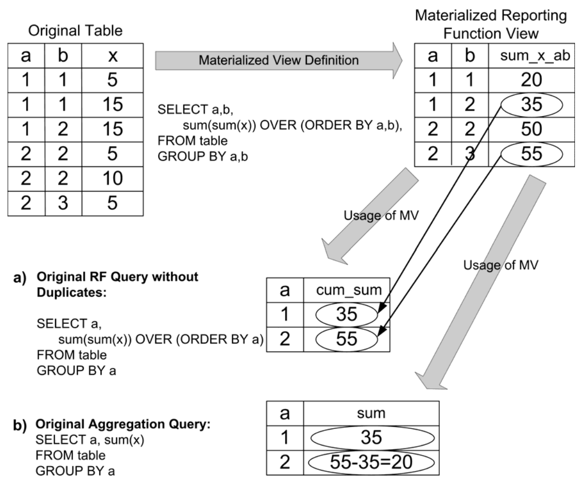

Query Q
WINDOW(∅: AFunc(AColExpr)[0,k])
PROJ(OBColList_Q: AFunc(AColExpr))
SORT(OBColList_Q)
ACCESS(P_Q, R)
a) reporting function query
ACCESS([OBColList_Q: k=last(OBColList_Q)], MV)
WINDOW(∅: AFunc(AColExpr)[0,k])
SORT(OBColList_MV)
ACCESS(P_MV, R)
MV
b) rewritten query exploiting reporting function MV

Figure 3. Derivability Conditions for Ordering Reduction


Original Table
Materialized View Definition
Materialized Reporting Function View
SELECT a,b, sum(sum(x)) OVER (ORDER BY a,b), FROM table GROUP BY a,b
Usage of MV
Usage of MV
a) Original RF Query without Duplicates: SELECT a, sum(sum(x)) OVER (ORDER BY a) FROM table GROUP BY a
b) Original Aggregation Query: SELECT a, sum(x) FROM table GROUP BY a

a	b	x
1	1	5
1	1	15
1	2	15
2	2	5
2	2	10
2	3	5

a	b	sum_x_ab
1	1	20
1	2	35
2	2	50
2	3	55

a	cum_sum
1	35
2	55

a	sum
1	35
2	55-35=20

Figure 4. Example for Patterns 1 and 2

```
... MAX(sum_x_ab)
... GROUP BY A
```

More generally, we are able to state the following rule:

Derivability by Ordering Reduction: A reporting function $RF_Q = (\oslash, OBColList_Q, WSpec, AFunc(AColsExpr))$ is derivable from a reporting function $RF_{MV} =(\oslash, OBColList_{MV}, WSpec, AFunc(AColsExpr))$ if $OBColList_Q$ is a prefix of the ordering scheme of $OBColList_{MV}$. Additionally, $WSpec$ is required to represent a cumulative window ($[\omega_s = 0, \omega_e = k]$), and the aggregation expressions are compatible according to the derivability and compensation rules.

In the above description, we assumed that the reporting function queries avoid duplicates in the results. If a query does not satisfy the uniqueness property, the same derivation pattern can be applied. However, the resulting data stream has to be extended with the right number of dupli-

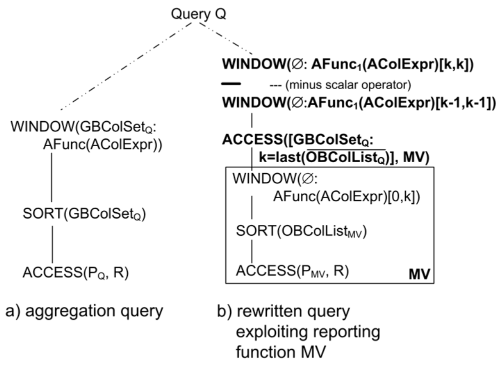

Query Q
WINDOW(GBColSet$_Q$: AFunc(AColExpr))
SORT(GBColSet$_Q$)
ACCESS(P$_Q$, R)
WINDOW(∅: AFunc$_1$(AColExpr)[k,k])
--- (minus scalar operator)
WINDOW(∅:AFunc$_1$(AColExpr)[k-1,k-1])
ACCESS([GBColSet$_Q$: k=last($\overline{\text{OBColList}_Q}$)], MV)
WINDOW(∅: AFunc(AColExpr)[0,k])
SORT(OBColList$_{MV}$)
ACCESS(P$_{MV}$, R)
MV
a) aggregation query
b) rewritten query exploiting reporting function MV

Figure 5. Derivability Conditions for Aggregation Queries

cates by applying the UNPACK operator as the last operator in the query execution graph.

Derivation Pattern 2 - Aggregation Query:

In contrast to the previous pattern, we want to answer a regular aggregation query Q from a reporting function view, where the set of group-by attributes $GBColSet_Q$ is a prefix of $OBColList_{MV}$. Figure 5 illustrates this pattern: the aggregation query can be rewritten to the materialized reporting function view with the help of reporting functions. Again, the set of columns $OBColList_{MV} - GBColSet_Q$ is denoted as $\overline{OBColList_Q}$. Also, the rewritten query takes advantage of the conditional ACCESS operator in order to retrieve the values with the highest index according to $\overline{OBColList_Q}$.

For example (see Figure 4(b)), the query expression

```
... SUM(x) AS sum_x_a
... GROUP BY A
```

can be derived from an MV expression

```
... SUM(SUM(X)) OVER (ORDER BY A,B) AS sum_x_ab
... GROUP BY A,B
```

by logically rewriting the query to a reporting function query over the materialized reporting function view:

```
... MAX(sum_x_ab) OVER (ORDER BY A
        ROWS BETWEEN CURRENT ROW AND CURRENT ROW)
    -    --- minus scalar operation
    MAX(sum_x_ab) OVER (ORDER BY A
        ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
... GROUP BY A
```

In other words, to answer an aggregation query Q from a materialized reporting function view, the first thing we have to do is to select the values with the highest index according to $\overline{OBColList_Q}$ in groups defined by $GBColSet_Q$. Subsequently we have to subtract the value of the preceding tuple from the value of the currently considered tuple by applying the regular scalar minus operator. To avoid a null value for the first tuple, we obviously have to extend the derivation pattern with a COALESCE statement. More formally, we are able to state the following rule:

Derivability of Aggregation Queries: An aggregation query Q with a set of group-by attributes $GBColSet_Q$ is derivable from a reporting function RF_{MV} $=(\oslash, OBColList_{MV}, WSpec, AFunc(AColsExpr))$ if $GBColSet_Q$ is a prefix of the ordering scheme of $OBColList_{MV}$.

4.3 Derivation Patterns for Reporting Function Views With Partitioning Attributes

The next step to more complex derivation patterns is to consider partitioning attributes within the reporting function. In this case, we can specify derivation patterns for (i) queries with reduced partitioning attributes and (ii) aggregation queries referring to partitioning attributes. Again we consider cumulative windows, compatible selection and aggregation expressions.

Derivation Pattern 3 - Partitioning Reduction:

For simplicity only, we assume for now that the query and the MV exhibit the same list of ordering columns. The resulting scenario is different from Pattern 1 because we have to combine values from different partitions to retrieve reporting function values of coarser partitioning granularity in this case.

Figure 6 illustrates the problem of partitioning reduction. The partitioning attributes of the MV are A,B with values A_1, A_2, B_1, and B_2; attribute C is the ordering attribute and consists of four values. Additionally, suppose that all 16 combinations exist for the corresponding reporting function values. Since the partitioning scheme of the MV is finer (A, B) than the query Q (A), we have to combine values according to the more coarse-grained partitioning scheme of the query. A typical situation is the reduction of $(year, quarter)$ to $(year)$ with daily numeric values.

We can now state the more general derivation pattern illustrated in Figure 7. A query can be answered within this pattern by generating an aggregation query with the following group-by attributes: $PBColSet_Q, OBColSet_{MV}$. For example, the query expression (following the example in Figure 6)

```
... SUM(SUM(X)) OVER (PARTITION BY A
                      ORDER BY C)
... GROUP BY A,C
```

can be derived from an MV

Final edited form was published in "18th International Conference on Scientific and Statistical Database Management (SSDBM'06), Vienna, 2006".S. 159-169. ISBN 0-7695-2590-3.
https://doi.org/10.1109/SSDBM.2006.32

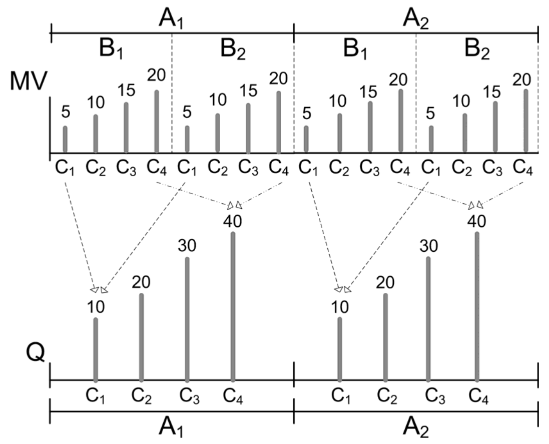


Figure 6. Example for Partitioning Reduction

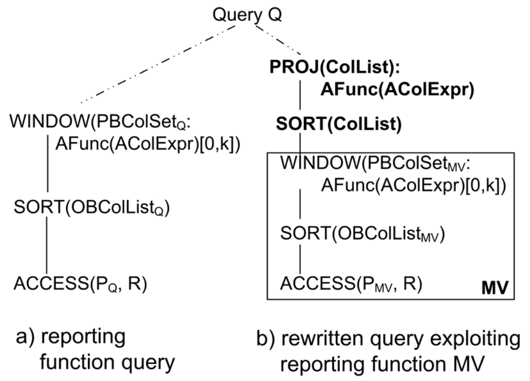

Query Q
WINDOW(PBColSetQ: AFunc(AColExpr)[0,k])
SORT(OBColListQ)
ACCESS(PQ, R)
PROJ(ColList): AFunc(AColExpr)
SORT(ColList)
WINDOW(PBColSetMV: AFunc(AColExpr)[0,k])
SORT(OBColListMV)
ACCESS(PMV, R)
MV
a) reporting function query
b) rewritten query exploiting reporting function MV

Figure 7. Derivability Conditions for Partitioning Reduction in Case of Ordering Completeness.

```
... SUM(SUM(X)) OVER (PARTITION BY A,B
                      ORDER BY C) AS rf_value
... GROUP BY A,B,C
```

by rewriting the query to an aggregation query over the materialized reporting function view:

```
... SUM(rf_value)
... GROUP BY A,C
```

Unfortunately, this derivation pattern is only valid if the same set of instances of the ordering attributes occurs in every partition (as illustrated in Figure 6). We call this requirement the *completeness property* regarding the ordering attributes. In following we would like to illustrate a scenario in which the completeness property is not fulfilled: assume that the tuple A_1, B_2, C_3 does not exist in the example from Figure 6. In this case, only the value of the tuple A_1, B_1, C_3 would go into the cumulative sum to compute tuple A_1, C_3 even though one value of the partition A_1, B_2 would have to

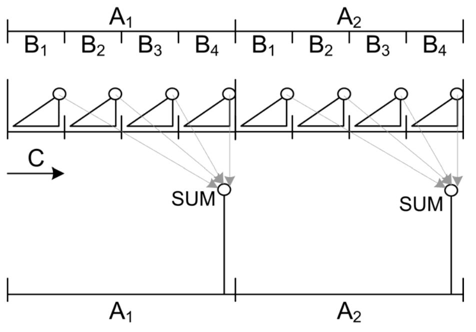


Figure 8. Example for a Group-By Reduction

be considered within this computation. Due to the fact that the combination A_1, B_2, C_3 does not exist, the fine-grained cumulative sum of the tuple A_1, B_2, C_2—the value of the preceding tuple regarding the ordering attributes—must be included. The trick to overcome this limitation is to apply a regular aggregation operation with a group-by condition over the attributes $PBColSet_Q \cup OBColList_{MV}$ as a preparatory step (see next pattern). Subsequently, the compensation itself corresponds directly to the original reporting function of the incoming query.

Moreover, it is interesting to note that this pattern can be easily extended to cover reduced ordering schemes according to Pattern 1. This observation leads to the following observation.

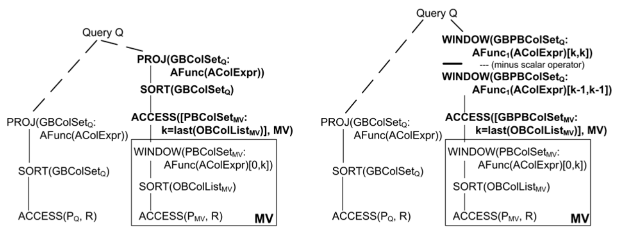
Derivability by Partitioning Reduction: A reporting function $RF_Q = (PBColSet_Q, OBColList_Q, WSpec, AFunc(AColsExpr))$ is derivable from a reporting function $RF_{MV} = (PBColSet_{MV}, OBColList_{MV}, WSpec, AFunc(AColsExpr))$ if $PBColSet_Q$ is a subset of $PBColSet_{MV}$ and $OBColList_Q$ is a prefix of the ordering scheme of $OBColList_{MV}$. Additionally, $WSpec$ is required to represent a cumulative window and the aggregation expressions are compatible according to the derivability and compensation rules.

Derivation Pattern 4 - Aggregation Queries:

In this part, we investigate the question of how to answer a regualr aggregation query where the group-by attributes $GBColSet_Q$ are a combination of partitioning and ordering attributes. Figure 8 illustrates the required task of a compensation query on top of an MV. In this example, the materialized reporting function view is defined with A, B as partitioning attributes and C as ordering column. The group-by attribute of the aggregation query Q is A.

In contrast to Pattern 2, we only have to pick the largest value of each partition if the sets of group-by and partitioning columns are equivalent, i.e. $GBColSet_Q =$

Final edited form was published in "18th International Conference on Scientific and Statistical Database Management (SSDBM'06), Vienna, 2006".S. 159-169. ISBN 0-7695-2590-3.
https://doi.org/10.1109/SSDBM.2006.32


Query Q
PROJ(GBColSet_Q: AFunc(AColExpr))
SORT(GBColSet_Q)
ACCESS([PBColSet_MV: k=last(OBColList_MV)], MV)
PROJ(GBColSet_Q: AFunc(AColExpr))
SORT(GBColSet_Q)
ACCESS(P_Q, R)
WINDOW(PBColSet_MV: AFunc(AColExpr)[0,k])
SORT(OBColList_MV)
ACCESS(P_MV, R)
MV
Query Q
WINDOW(GBPBColSet_Q: AFunc_1(AColExpr)[k,k])
--- (minus scalar operator)
WINDOW(GBPBColSet_Q: AFunc_1(AColExpr)[k-1,k-1])
ACCESS([GBPBColSet_MV: k=last(OBColList_MV)], MV)
PROJ(GBColSet_Q: AFunc(AColExpr))
SORT(GBColSet_Q)
ACCESS(P_Q, R)
WINDOW(PBColSet_MV: AFunc(AColExpr)[0,k])
SORT(OBColList_MV)
ACCESS(P_MV, R)
MV

a) aggregation query based on MV with reporting function (subset relationship of group by and partitioning columns)

b) aggregation query based on MV with reporting function (group by is an union of an prefix of partitioning columns and the entire ordering columns)

Figure 9. Derivability Conditions for Aggregation Queries

$PBColSet_{MV}$. Derivability is also possible if the set of group-by columns is a prefix of $PBColSet_{MV}$. In this case, we require the compensation query given in Figure 9(a), where we first pick the largest value of each fine-grained partition according to $PBColSet_{MV}$ and then apply the group-by condition of the query Q. In both cases, we must select the largest value of each fine-grained partition. We can also answer an aggregation query if the group-by attributes $GBColSet_Q$ are a combination of a prefix of the partitioning attributes $PBColSet_{MV}$ and the entire ordering attributes $OBColList_{MV}$ of the materialized reporting function view. The necessary compensation query is depicted in Figure 9(b); this pattern can be seen as a seamless extension of Pattern 2. Furthermore, we can also consider any combination of a prefix of the partitioning attributes and a prefix of the ordering attributes.

4.4 Summary

To summarize, we are able to give a broad spectrum of derivation rules for different situations with reporting functions in materialized views, i.e. in the case of ordering or partition reduction. Moreover, we support queries with reporting functions and typical group-by aggregation operations. All situations are based on cumulative sum as the underlying materializing strategy and as the windowing semantics for all derived queries. However, deriving reporting function views with sliding window semantics can also be done within the same framework. Due to space restrictions, we just point out the basic idea: each sliding window $w_s(k)[k-i, k+j]$ can be reconstructed by referring to two cumulative windows $w_c(k)[0, k+j]$ and $w_c(k)[0, k-i]$. The given patterns have to be adjusted in order to keep track of two cumulative window values in order to compute the value for sliding windows.

5 Evaluation

In this section, we explore the performance gain of our materialized reporting function views and the proposed patterns. All experiments are conducted on a single-disk Pentium 4 PC with 1.5 GB main memory running Linux. All patterns are evaluated with synthetic data sets and with TPCD Benchmark data.

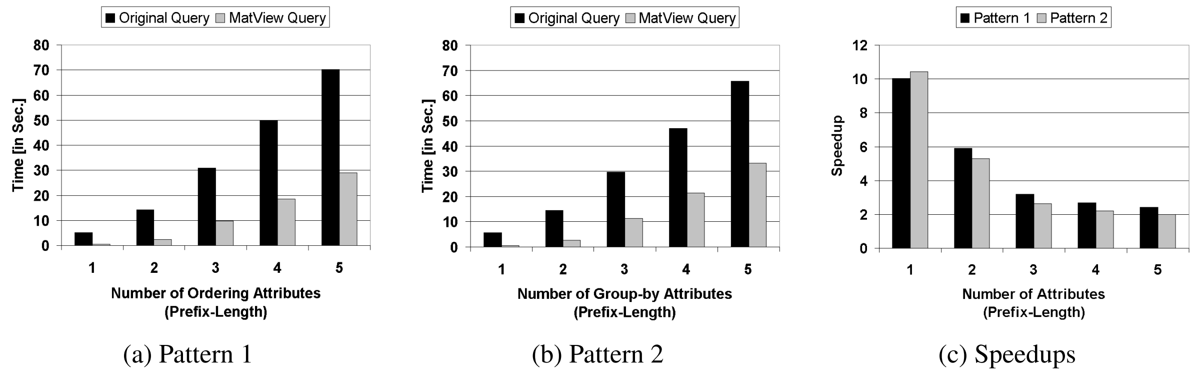
Synthetic Data Sets:

The first set of experiments is conducted on a single table storing randomly generated tuples. For these experiments, we developed a JAVA-Programm executing the corresponding SQL queries according to the patterns on a commercial relational database and writing the results to a flat file. Each query is executed ten times, and the average execution time is then used as the time measure.

For the first experiments, we defined a materialized view with a reporting function including five ordering columns, an empty partitioning scheme and cumulative window definition. The underlying raw data table consisted of $500,000$ rows, where 10 percent were distinct with regard to the five ordering columns. Figure 10 shows the evaluation of the applicable Patterns 1 and 2. Pattern 1 rewrites reporting function queries with a reduced ordering scheme to the MV. The runtimes of the original queries and the rewritten queries with one to five ordering columns (prefixes) are depicted in Figure 10(a). Pattern 2 is responsible for answering regular aggregation queries, where the group-by attributes are a prefix of the ordering attributes of the MV. Figure 10(b) shows the runtimes of the corresponding queries depending on the prefix length of the ordering scheme. The resulting speedups depicted in Figure 10(c) show that our Patterns 1 and 2 achieve a high performance gain. The performance speedup increases with the decreasing prefix length of the ordering scheme. We also observe that the speedups of both patterns are nearly identical.

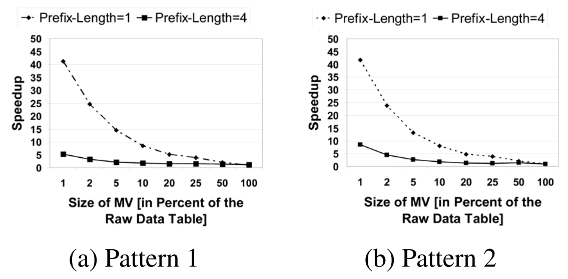
In the second experiment depicted in Figure 11, we investigated the influence of the data reduction of the materialized reporting function view on the raw data table, that means we do not store duplicates in the materialized views. In the previous experiment, the MV stored only 10% of the raw data table and we achieved a high performance gain for Patterns 1 and 2. Figure 11 shows the resulting speedups of Pattern 1 and 2 depending on the data reduction. Again, the raw data table consisted of $500,000$ randomly generated rows and in the experiment we varied the distinct tuples regarding the five ordering columns. In the diagrams, we illustrate only the performance gain for queries with attribute prefix length 1 and 4. The results show that the larger the data reduction, the larger the speedup for both patterns.

In the next experiment, we defined a materialized re-



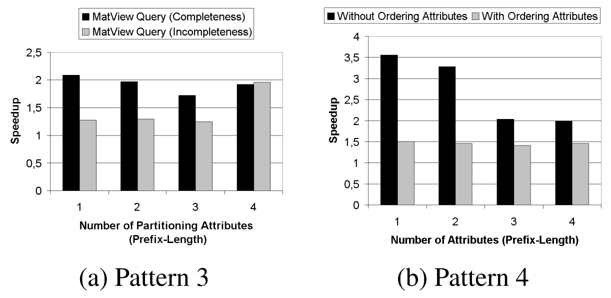
(a) Pattern 1 (b) Pattern 2 (c) Speedups

Figure 10. Evaluation of Patterns 1 and 2 on Synthetic Data Sets.



(a) Pattern 1 (b) Pattern 2

Figure 11. Influence of the Data Reduction on the Speedup of Patterns 1 and 2.



(a) Pattern 3 (b) Pattern 4

Figure 12. Evaluation of Patterns 3 and 4 on Synthetic Data Sets.

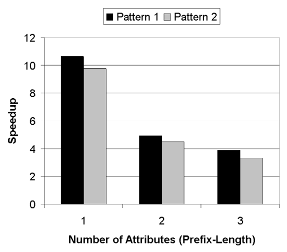
porting function view with 4 partitioning and 2 ordering attributes and cumulative window definition with the same data parameters as in the first experiment to evaluate Patterns 3 and 4. Figure 12(a) shows the performance gain of Pattern 3, where we have to consider the completeness regarding the ordering scheme. We generated a data sets which satisfies this requirement and evaluated both valid rewriting techniques. In general, Pattern 3 is responsible for reporting function queries with a reduced partitioning scheme. The diagram shows that both techniques are efficient and that the second (incomplete case) is slower than the first (complete case).

Figure 12(b) shows the speedups for Pattern 4 for two kinds of regular aggregation queries for this pattern. In both cases we varied the number of partitioning attributes as group-by attributes (prefix condition). In one case, we added the entire ordering attributes of the MV to the group-by condition, while in the other case we used only the partitioning attributes. The diagram shows that a higher speedup can be achieved for queries without the ordering attributes. The performance gain of both Patterns 3 and 4 cannot be as high as for Patterns 1 and 2 because they always work on the largest value of the fine-grained partitions in the MV. To increase the speedup, we require a more sophisticated index structure, which allows direct access to the largest value in the partitions according to $PBColSet_{MV}$.

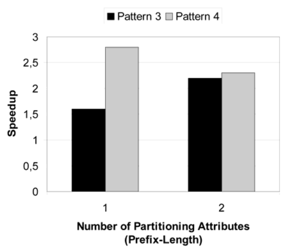
TPCD Benchmark Data:

In the second part of the experiment, we used the TPCD Benchmark with scaling factor 0.3 and $skew = 2.0$ resulting in 1.8 million rows within the lineitem table. In the first experiment here, we consider a materialized view defined with a reporting function including 3 ordering columns, an empty partitioning scheme and no explicit window definition (e.g. cumulative sum values). The MV query is depicted in Listing 3.

Figure 13(a) shows the performance speedup which is achieved for the corresponding queries according to Patterns 1 and 2. The speedup behavior is comparable to the experiment with the synthetic data set. For the evaluation of Patterns 3 and 4, we changed the reporting function of the previous MV query as follows: c_name and $l\_shipmode$ as partitioning attributes and $year$ and $month$ of $l\_shipdate$ as ordering scheme. The performance speedup of Patterns 3 and 4 are depicted in Figure 13(b). The ordering completeness for Pattern 3 was not fulfilled. For Pattern 4, we only consider regular aggregation queries where the group-



(a) Pattern 1 and 2



(b) Pattern 3 and 4

Figure 13. Evaluation on TPCD Benchmark

by attributes are a prefix of the partitioning attributes. The experiments on the TPCD Benchmark confirm the results of the evaluation on the synthetic data sets.

```
SELECT ..., SUM(SUM(l.l_extendedprice)) OVER (
                 ORDER BY c.c_name ,YEAR(l.l_shipdate) ,
                          MONTH(l.l_shipdate))
FROM lineitem l, order o, customer c
WHERE l.l_orderkey = o.o_orderkey AND
      o.o_custkey = c.c_custkey
GROUP BY c.c_name , YEAR(l.l_shipdate) ,
         MONTH(l.l_shipdate)
```

Listing 3. MV Query

6 Conclusion

Reporting functions reflect a feature which is quite heavily used in complex decision support queries issued to large data warehouse databases. Surprisingly, there is no work considering the possibility and benefit of materialized views based on reporting functions. We therefore give a semi-formal extension of relational query graphs to capture the semantics of reporting functions and discuss a broad variety of derivation patterns. The conducted experiments confirm that the existence of MVs defined over reporting functions are of great benefit and provide great performance improvements for regular aggregation queries as well as for queries with reporting functions under the condition of ordering and/or partitioning reduction.

Our next research topics are view design and view maintenance in the presence of reporting functions. At the moment, we are working on an extension of regular B-tree indexes from a syntactical as well as an implementational point of view. To increase the performance speedup of our patterns, we require an efficient method for direct access to tuples with certain positional conditions. Our core idea is to use a secondary index structure where the specific key values are no longer an unordered set, but reflect a list which is ordered according to a condition. Adding a sort order to the secondary index structure enables a search algorithm to consider additional positional constraints more efficiently. In this way, the search algorithm can evaluate additional constraints like FIRST() or LAST() or even more complex positional tests like MOD() et cetera, which are frequently required in our patterns.

References

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, 2000.

[2] J.-Y. Chang and S.-G. Lee. Query reformulation using materialized views in data warehousing environment. In *Proceedings of the 1st International Workshop on Data Warehousing and OLAP (DOLAP)*, 1998.

[3] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the 11th International Conference on Data Engineering(ICDE)*, 1995.

[4] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2001.

[5] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB)*, 1995.

[6] E. N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1987.

[7] W. Lehner, W. Hümmer, and L. Schlesinger. Processing reporting function views in a data warehouse environment. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, 2002.

[8] G. Özsoyoglu, Z. M. Özsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. Database Syst.*, 12(4):566–592, 1987.

[9] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, 2000.

[10] N. Roussopoulos. View indexing in relational databases. In *ACM Transactions on Database Systems, 7(2)*, 1982.

[11] N. Roussopoulos. Materialized views and data warehouses. In *ACM SIGMOD Record, 27(1)*, 1998.

[12] P. Seshadri, M. Livny, and R. Ramakrishnan. Seq: A model for sequence databases. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, 1995.

[13] B.-K. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, 2000.

[14] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2000.