Optimizing Queries Using Materialized Views:
A Practical, Scalable Solution

Jonathan Goldstein and Per-Åke Larson

Microsoft Research, One Microsoft Way, Redmond, WA 98052

{jongold,palarson}@microsoft.com

Abstract

Materialized views can provide massive improvements in query processing time, especially for aggregation queries over large tables. To realize this potential, the query optimizer must know how and when to exploit materialized views. This paper presents a fast and scalable algorithm for determining whether part or all of a query can be computed from materialized views and describes how it can be incorporated in transformation-based optimizers. The current version handles views composed of selections, joins and a final group-by. Optimization remains fully cost based, that is, a single "best" rewrite is not selected by heuristic rules but multiple rewrites are generated and the optimizer chooses the best alternative in the normal way. Experimental results based on an implementation in Microsoft SQL Server show outstanding performance and scalability. Optimization time increases slowly with the number of views but remains low even up to a thousand.

Keywords

Materialized views, view matching, query optimization.

1. Introduction

Using materialized views to speed up query processing is an old idea [10] but only in the last few years has the idea been adopted in commercial database systems. Recent TPC-R benchmark results and actual customer experiences show that query processing time can be improved by orders of magnitude through judicious use of materialized views. To realize the potential of materialized views, efficient solutions to three issues are required:

- **View design:** determining what views to materialize, including how to store and index them.
- **View maintenance:** efficiently updating materialized views when base tables are updated.
- **View exploitation:** making efficient use of materialized views to speed up query processing.

This paper deals with view exploitation in transformation-based optimizers. Conceptually, an optimizer generates all possible rewritings of a query expression, estimates their costs, and chooses the one with the lowest cost. A transformation-based optimizer generates rewritings by applying local transformation rules on subexpressions of the query. Applying a rule produces substitute expressions, equivalent to the original expression. View matching, that is, computing a subexpression from materialized views, is one such transformation rule. The view-matching rule invokes a view-matching algorithm that determines whether the original expres-

sion can be computed from one or more of the existing materialized views and, if so, generates substitute expressions. The algorithm may be invoked many times during optimization, each time on a different subexpression.

The main contributions of this paper are (a) an efficient view-matching algorithm for views composed of selections, joins and a final group-by (SPJG views) and (b) a novel index structure (on view definitions, not view data) that quickly narrows the search to a small set of candidate views on which view-matching is applied. The version of the algorithm described here is limited to SPJG views and produces single-view substitutes. However, these are not inherent limitations of our approach; the algorithm and the index structure can be extended to a broader class of views and substitutes. We briefly discuss possible extensions but the details are beyond the scope of this paper.

Our view-matching algorithm is fast and scalable. Speed is crucial because the view-matching algorithm may be called many times during optimization of a complex query. We also wanted an algorithm able to handle thousands of views efficiently. Many database systems contain hundreds, even thousands, of tables. Such databases may have hundreds of materialized views. Tools similar to that described in [1] can also generate large numbers of views. A smart system might also cache and reuse results of previously computed queries. Cached results can be treated as temporary materialized views, easily resulting in thousands of materialized views. The algorithm was implemented in Microsoft SQL Server, which uses a transformation-based optimizer based on the Cascades framework [6]. Experiments show outstanding performance and scalability. Optimization time increases linearly with the number of views but remains low even up to a thousand.

Integrating view matching through the optimizer's normal rule mechanism provides important benefits. Multiple rewrites may be generated; some exploiting materialized views, some not. All rewrites participate in the normal cost-based optimization, regardless of whether they make use of materialized views. Secondary indexes, if any, on materialized views are automatically considered. The optimization time may even be reduced. If a cheap plan using materialized views is found early in the optimization process, it tightens cost bounds resulting in more aggressive pruning.

The rest of the paper is organized as follows. Section 2 describes the class of materialized views supported and defines the problem to be solved. Section 3 describes our algorithm for deciding if a query expression can be computed from a view. Section 4 introduces our index structure. Section 5 presents experimental results based on our prototype implementation. Related work is discussed in section 6. Section 7 contains a summary and a brief discussion of possible extensions.

2. Defining the problem

SQL Server 2000 supports materialized views. They are called indexed views because a materialized view may be indexed in multiple ways. A view is materialized by creating a unique clustered

index on an existing view. Uniqueness implies that the view output must contain a unique key. This is necessary to guarantee that views can be updated incrementally. Once the clustered index has been created, additional secondary indexes can be created. Not all views are indexable. An indexable view must be defined by a single-level SQL statement containing selections, (inner) joins, and an optional group-by. The FROM clause cannot contain derived tables, i.e. it must reference base tables, and subqueries are not allowed. The output of an aggregation view must include all grouping columns as output columns (because they define the key) and a count column. Aggregation functions are limited to sum and count. This is the class of views considered in this paper.

Example 1: This example shows how to create an indexed view, with an additional secondary index, in SQL Server 2000. All examples in this paper use the TPC-H/R database.

```
create view v1 with schemabinding as
  select p_partkey, p_name, p_retailprice,
    count_big(*) as cnt,
    sum(l_extendedprice*l_quantity) as gross_revenue
  from dbo.lineitem, dbo.part
  where p_partkey < 1000
    and p_name like '%steel%'
    and p_partkey = l_partkey
  group by p_partkey, p_name, p_retailprice

create unique clustered index v1_cidx on v1(p_partkey)
create index v1_sidx on v1( gross_revenue, p_name)
```

The first statement creates the view v1. The phrase "with schemabinding" is required for indexed views. A count_big column is required in all aggregation views so deletions can be handled incrementally (when the count becomes zero, the group is empty and the row must be deleted). Output columns defined by arithmetic or other expressions must be assigned names (using the AS clause) so that they can be referred to. The second statement materializes the view and stores the result in a clustered index. Even though the statement specifies only the (unique) key of the view, the rows contain all output columns. The final statement creates a secondary index on the materialized view. □

As outlined in the introduction, a transformation-based optimizer generates rewrites by recursively applying transformation rules on relational expressions. View matching is a transformation rule that is invoked on select-project-join-group-by (SPJG) expressions. For each expression, we want to find every materialized view from which the expression can be computed. In this paper, we require that the expression can be computed from the view alone. The following is the view-matching problem considered in this paper.

View Matching with Single-View Substitutes: *Given a relational expression in SPJG form, find all materialized (SPJG) views from which the expression can be computed and, for each view found, construct a substitute expression equivalent to the given expression.*

No restrictions are imposed on the overall query. Even though we consider only single-view substitutes, different views may be used to evaluate different parts of a query. Whenever the optimizer finds a SPJG expression the view-matching rule is invoked. All substitutes produced by view matching participate in cost-based optimization in the normal way. Furthermore, any secondary indexes defined on a materialized view will be considered automatically in the same way as for base tables.

The algorithm explained in this paper is limited to SPJG subexpressions and single-table substitutes. However, this is not an in-herent limitation of our approach. The algorithm can be extended to a broader class of input and substitute expressions, for example, expressions containing unions, outer joins or aggregation with grouping sets.

3. Computing a query expression from a view

In this section, we describe the tests applied to determine whether a query expression can be computed from a view and, if it can, how to construct the substitute expression. The first subsection deals with join-select-project (SPJ) views and queries assuming that the view and the query reference the same tables. Views with extra tables and views with aggregation are covered in separate subsections. There is no need to consider views with fewer tables than the query expression. Such views can only be used to compute a subexpression of the query expression. The view-matching rule will automatically be invoked on every subexpression.

Our algorithm exploits four types of constraints: not-null constraints on columns, primary key constraints, uniqueness constraints (either explicitly declared or implied by creating a unique index), and foreign key constraints. We assume that the selection predicates of view and query expressions have been converted into conjunctive normal form (CNF). If not, we first convert them into CNF. We also assume that join elimination has been performed so query and view expressions contain no redundant tables. (The SQL Server optimizer does this automatically.)

3.1 Join-select-project views and queries

For a SPJ query expression to be computable from a view, the view must satisfy the following requirements.

1. *The view contains all rows needed by the query expression.* Because we are considering only single-view substitutes, this is an obvious requirement. However, this is not required if substitutes containing unions of views are considered.

2. *All required rows can be selected from the view.* Even if all required rows exist in the view, we may not be able to extract them correctly. Selection is done by applying a predicate. If one of the columns required by the predicate is missing from the view output, the required rows cannot be selected.

3. *All output expressions can be computed from the output of the view.*

4. *All output rows occur with the correct duplication factor.* SQL is based on bag semantics, that is, a base table or the output of a SQL expression may contain duplicate rows. Hence, it is not sufficient that two expressions produce the same set of rows but any duplicate rows must also occur exactly the same number of times.

Equivalences among columns play an important role in our tests so we cover this topic first. We then discuss how we ensure that the requirements above are met, devoting a separate subsection to each requirement.

3.1.1 Column equivalence classes

Let $W = P_1 \wedge P_2 \wedge ... \wedge P_n$ be the selection predicate (in CNF) of a SPJ expression. By collecting the appropriate conjuncts, we can rewrite the predicate as $W = PE \wedge PNE$ where PE contains column equality predicates of the form $(T_i. C_p = T_j.C_q)$ and PNE contains all remaining conjuncts. T_i and T_j are tables, not necessarily distinct, and C_p and C_q are column references.

Suppose we evaluate the SPJ expression by computing the Cartesian product of the tables, then applying the column-equality predicates in PE, then applying the predicates in PNE, and finally computing the expressions in the output list. After the column-

equality predicates have been applied, some columns are interchangeable in both the PNE predicates and the output columns. This ability to reroute column references among equivalent columns will be important later on.

Knowledge about column equivalences can be captured compactly by computing a set of equivalence classes based on the column equality predicates in PE. An equivalence class is a set of columns that are known to be equal. Computing the equivalence classes is straightforward. Begin with each column of the tables referenced by the expression in a separate set. Then loop through the column equality predicates in any order. For each $(T_i.C_p = T_j.C_q)$, find the set containing $T_i.C_p$ and the set containing $T_j.C_q$. If they are in different sets merge the two sets, otherwise do nothing. The sets left at the end is the desired collection of equivalence classes, including trivial classes consisting of a single column.

3.1.2 Do all required rows exist in the view?

Assume that the query expression and the view expression reference the tables $T_1, T_2, ..., T_m$. Let W_q denote the predicate in the where-clause of the query expression and W_v the predicate of the view expression. Determining whether the view contains all rows required by the query expression is, in principle, easy. All we need to show is that the output of the expression (select * from $T_1, T_2, ..., T_m$ where W_q) produces a subset of the output of (select * from $T_1, T_2, ..., T_m$ where W_v) for all valid instances of tables $T_1, T_2, ..., T_m$. This is guaranteed to hold if $W_q \Rightarrow W_v$, where '$\Rightarrow$' denotes logical implication.

Therefore we need an algorithm to decide whether $W_q \Rightarrow W_v$ holds. We rewrite the predicates as $W_q = P_{q,1} \wedge P_{q,2} \wedge ... \wedge P_{q,m}$ and $W_v = P_{v,1} \wedge P_{v,2} \wedge ... \wedge P_{v,n}$. A simple conservative algorithm is to check that every conjunct $P_{v,i}$ in W_v, matches a conjunct $P_{v,i}$ in W_q. There are several ways to decide whether two conjuncts match. For instance, the matching can be purely syntactic.This can be implemented by converting each conjunct into a string, i.e., the SQL text version of the conjunct, and then matching the strings. The drawback with this approach is that even minor syntactic differences result in different strings. For example, the two predicates $(A > B)$ and $(B < A)$ would not match. To avoid this problem, we must interpret the predicates and exploit equivalences among expressions. Exploiting commutativity is a good example, applicable to many types of expressions: comparisons, addition, multiplication, and disjunction (OR). We can design matching functions at different levels of sophistication and complexity depending on how much knowledge about equivalences we build into the function. For example, a simple function might only understand that $(A+B) = (B+A)$, while a more sophisticated function might also recognize that $(A/2 + B/5)*10 = A*5 + B*2$.

Our decision algorithm exploits knowledge about column equivalences and column ranges. We first divide the predicates W_q and W_v into three components and write the implication test as

$$(PE_q \wedge PR_q \wedge PU_q \Rightarrow PE_v \wedge PR_v \wedge PU_v).$$

PE_q consists of all column equality predicates from the query, PR_q contains range predicates, and PU_q is the residual predicate containing all remaining conjuncts of W_q. W_v is divided similarly. A column-equality predicate is any atomic predicate of the form $(T_i.C_p = T_j.C_r)$, where C_p and C_r are column references. A range predicate is any atomic predicate of the form $(T_i.C_p \; op \; c)$ where c is a constant and op is one of the operators "<", "≤", "=", "≥", ">". The implication test can be split into three separate tests:

$(PE_q \wedge PR_q \wedge PU_q \Rightarrow PE_v) \wedge$
$(PE_q \wedge PR_q \wedge PU_q \Rightarrow PR_v) \wedge$
$(PE_q \wedge PR_q \wedge PU_q \Rightarrow PU_v).$

An implication test can be strengthened by dropping conjuncts in the antecedent. (Expressed in formal terms, the formula $(A \Rightarrow C) \Rightarrow (AB \Rightarrow C)$ holds for arbitrary predicates A, B, C. In words, if we can deduce that A by itself implies C then certainly A and B together imply C.) Our final tests are strengthened versions of the three tests. To determine whether all rows required by a query exist in the view, we apply the following three tests:

$(PE_q \Rightarrow PE_v)$ (Equijoin subsumption test)

$(PE_q \wedge PR_q \Rightarrow PR_v)$ (Range subsumption test)

$(PE_q \wedge PU_q \Rightarrow PU_v)$ (Residual subsumption test)

The first test is called the equijoin subsumption test because, in practice, most column equality predicates come from equijoins. However, all column equality predicates are included in PE, even those referencing columns in the same table. Recall that the predicates in PE_q are the column equality predicates used for computing the query equivalence classes. Since PE_q is in the antecedent in the latter two implications, we can reroute a column reference to any column within its query equivalence class.

The tests are clearly stronger than minimally required and may cause some opportunities to be missed. For instance, by dropping PR_q from the antecedent of the equijoin test we will miss cases when the query equates two columns to the same constant, say, $(A=2) \wedge (B=2)$ and the view contains the weaker predicate $(A=B)$. A similar problem may arise in the residual subsumption test. For instance, if the query contains $(A=5) \wedge (B=3)$ and the view contains the predicate $(A+B) = 5$, we would safely but incorrectly conclude that the view does not provide all required rows. It is a tradeoff between speed and completeness.

Check constraints can be readily incorporated into the tests. The key observation is that check constraints on the tables of a query can be added to the where-clause without changing the query result. Hence, check constraints can be taken into account by including them in the antecedent of the implication $W_q \Rightarrow W_v$. Whether or not the check constraints will actually be exploited depends on the algorithm used for testing.

Equijoin subsumption test.
The equijoin subsumption test amounts to requiring that all columns equal in the view must also be equal in the query (but not vice versa). We implement this test by first computing column equivalence classes, as explained in the previous section, both for the query and the view, and then checking whether every non-trivial view equivalence class is a subset of some query equivalence class. Just checking that all column equality predicates in the view also exist in the query is a much weaker test because of transitivity. Suppose the view contains ($A=B$ and $B=C$) and the query contains $(A=C$ and $C=B)$. Even though the actual predicates don't match, they are logically equivalent because they both imply that $A=B=C$. The effect of transitivity is correctly captured by using equivalence classes.

If the view passes the equijoin subsumption test, we know that it does not contain any conflicting column equality constraints. We can also easily compute what, if any, compensating column equality constraints must be enforced on the view to produce the query result. Whenever some view equivalence classes $E_1, E_2 ... E_n$ map to the same query equivalence class E, we create a column-equality predicate between any column in E_i and any column in E_{i+1} for $i=1, 2, ..., n-1$.

Range subsumption test.
When no ORs are involved, there is an easy algorithm for the range subsumption test. We associate with each equivalence class

in the query a range that specifies a lower and upper bound on the columns in the equivalence class. Both bounds are initially left uninitialized. We then consider the range predicates one by one, find the equivalence class containing the column referenced, and set or adjust its range as needed. If the predicate is of type $(T_i.C_p <= c)$, we set the upper bound to the minimum of its current value and c. If is of type $(T_i.C_p >= c)$, we set the upper bound to the maximum of its current value and c. Predicates of the form $(T_i.C_p < c)$ are treated as $(T_i.C_p <= c\text{-}\Delta)$ where $c\text{-}\Delta$ denotes the smallest value preceding c in the domain of column $T_i.C_p$. Predicates of the form $(T_i.C_p > c)$ are treated as $(T_i.C_p <= c\text{+}\Delta)$. Finally, predicates of the form $(T_i.C_p = c)$ are treated as $(T_i.C_p >= c) \wedge (T_i.C_p <= c)$. The same process is repeated for the view.

The view cannot produce all required rows if it is more tightly constrained than the query. To check this, we consider the view equivalence classes with ranges where at least one of the bounds has been set. We find the matching equivalence class in the query, the query equivalence class that has at least one column in common with the query equivalence class, and check whether the range of the query equivalence class is contained in the range of the view equivalence class. (Uninitialized bounds are treated as $+\infty$ or $-\infty$.) If it is not, the range subsumption test fails and the view is rejected.

During this process we can determine what compensating range predicates must be applied to the view to produce the query result. If a query range matches precisely the corresponding view range, no restriction is needed. If the lower bound doesn't match exactly, we must restrict the view result by enforcing the predicate $(T.C >= lb)$ where $T.C$ is a column in the (query) equivalence class and lb is the lower bound of the query range. If the upper bounds differ, we need to enforce the predicate $(T.C <= ub)$.

This range coverage algorithm can be extended to support disjunctions (OR) of range predicates. Due to space limitations, we will not discuss the extension here. Our prototype does not support disjunctions.

Residual subsumption test.
Conjuncts that are neither column-equality predicates nor range predicates form the residual predicates of the query and the view. The only reasoning applied to these predicates is column equivalence. We test the implication by checking whether every conjunct in the view residual predicate matches a conjunct in the query residual predicate. Two column references match if they belong to the same (query) equivalence class. If the match fails, the view is rejected because the view contains a predicate not present in the query. Any residual predicate in the query that did not match anything in the view must be applied to the view.

As discussed in the beginning of this section, whether two conjuncts are found to match depends on the matching algorithm. Our prototype implementation uses a shallow matching algorithm: except for column equivalences, the expressions must be identical. An expression is represented by a text string and a list of column references. The text string contains the textual version of the expression with column references omitted. The list contains every column reference in the expression, in the order they would occur in the textual version of the expression. To compare two expressions, we first compare the strings. If they are equal, we scan through the two lists comparing column references in the same positions in the two lists. If both column references are contained in the same (query) equivalence class, the column references match, otherwise not. If all column pairs match, the expressions match. We chose this shallow algorithm for speed, fully aware that it may cause some opportunities to be missed.

In summary, here are the steps of our procedure for testing whether a view contains all the rows needed by the query.

1. Compute equivalence classes for the query and the view.
2. Check that every view equivalence class is a subset of a query equivalence class. If not, reject the view
3. Compute range intervals for the query and the view.
4. Check that every view range contains the corresponding query range. If not, reject the view.
5. Check that every conjunct in the residual predicate of the view matches a conjunct in the residual predicate of the query. If not, reject the view.

Example 2:

View:
```
Create view V2 with schemabinding as
Select l_orderkey, o_custkey, l_partkey,
      l_shipdate, o_orderdate,
      l_quantity*l_extendedprice as gross_revenue
From dbo.lineitem, dbo.orders, dbo.part
Where l_orderkey = o_orderkey
  And l_partkey  = p_partkey
  And p_partkey >= 150
  And o_custkey >= 50 and o_custkey <= 500
  And p_name like '%abc%'
```

Query:
```
Select l_orderkey, o_custkey, l_partkey,
l_quantity*l_extendedprice
From lineitem, orders, part
Where l_orderkey = o_orderkey
  And l_partkey  = p_partkey
  And l_partkey >= 150 and l_partkey <= 160
  And o_custkey = 123
  And o_orderdate = l_shipdate
  And p_name like '%abc%'
  And l_quantity*l_extendedprice > 100
```

Step 1: Compute equivalence classes.
View equivalence classes: {l_orderkey, o_orderkey}, {l_partkey, p_partkey}, {o_orderdate}, {l_shipdate}

Query equivalence classes: {l_orderkey, o_orderkey}, {l_partkey, p_partkey}, {o_orderdate, l_shipdate}

Not all trivial equivalence classes are shown; {o_orderdate} and {l_shipdate} are included because they are needed later in the example.

Step 2: Check view equivalence class containment.
The two non-trivial view equivalence classes both have exact matches among the query equivalence classes. The (trivial) equivalence classes {o_orderdate} and {l_shipdate} map to the same query equivalence class, which means that the substitute expression must create the compensating predicate (o_orderdate=l_shipdate).

Step 3: Compute ranges.
View ranges: {l_partkey, p_partkey} ∈ (150, +∞),
                  {o_custkey} ∈ (50, 500)

Query ranges: {l_partkey, p_partkey} ∈ (150, 160),
                  {o_custkey} ∈ (123, 123)

Step 4: Check query range containment.
The range (150, 160) on {l_partkey, p_partkey} is contained in the corresponding view range. The upper bounds do not match so we have to enforce the predicate ({l_partkey, p_partkey} <= 160). The range (123, 123) on {o_custkey} is also contained in the corresponding view range. The bounds don't mach so we must en-

334

force the predicates (o_custkey >= 123) and (o_custkey <= 123), which can be simplified to (o_custkey = 123).

Step 5: Check match of view residual predicates.
View residual predicate: p_name like '%abc%'

Query residual predicate: p_name like '%abc%',
                               l_quantity*l_extendedprice > 100

The view has only one residual predicate, p_name like '%abc%', which also exists in the query. The extra residual predicate, l_quantity*l_extendedprice > 100 must be enforced.

The view passes all the tests so we conclude that it contains all the required rows. The compensating predicates that must be applied to the view are (o_orderdate = l_shipdate), ({p_partkey, l_partkey} <= 160), (o_custkey = 123), and (l_quantity*l_extendedprice > 100.00). The notation {p_partkey, l_partkey} in the second predicates means that we can choose either p_partkey or l_partkey. □

3.1.3 Can the required rows be selected?
We explained in the previous section how to determine the compensating predicates that must be enforced on the view to reduce it to the correct set of rows. They are of three different types.

1. Column equality predicates obtained while comparing view and query equivalence classes. In our example above, there was one predicate of this type: (o_orderdate = l_shipdate).

2. Range predicates obtained while checking query ranges against view ranges. There were two predicates of this type: , ({p_partkey, l_partkey} <= 160) and (o_custkey = 123).

3. Unmatched residual predicates from the query. There was one predicate of this type: (l_quantity*l_extendedprice > 100).

All compensating predicates must be computable from the view's output. We exploit equalities among columns by considering each column reference to refer to the equivalence class containing the column, instead of referencing the column itself. The query equivalence classes are used in all but one case, namely, the compensating column equality predicates (point one in the list above). These predicates were introduced precisely to enforce additional column equalities required by the query. Each such predicate merges two view equivalence classes and, collectively, they make the view equivalence classes equal to the query equivalence classes. Hence, a column reference can be redirected to any column within its view equivalence class but not within its query equivalence class.

Compensating predicates of type 1 and type 2 above contain only simple column references. All we need to do is check whether at least one of the columns in the referenced equivalence class is an output column of the view and route the reference to that column. Compensating predicates of type 3 may involve more complex expressions. In that case, it may be possible to evaluate the expression even though some of the columns referenced cannot be mapped to an output column of the view. For example, if l_quantity*l_extendedprice is available as a view output column, we can still evaluate the predicate (l_quantity*l_extendedprice > 100) without the columns l_quantity and l_extendedprice. However, our prototype implementation ignores this possibility and requires that all columns referenced in compensating predicates be mapped to (simple) output columns of the view.

In summary, we determine whether all rows required by the query can be correctly selected from a view as follows.

1. Construct compensating column equality predicates while comparing view equivalence classes against query equivalence classes as described in the previous section. Try to map every column reference to an output column (using the view equivalence classes). If this is not possible, reject the view.

2. Construct compensating range predicates by comparing column ranges as described in the previous section. Try to map every column reference to an output column (using the query equivalence classes). If this is not possible, reject the view.

3. Find the residual predicates of the query that are missing in the view. Try to map every column reference to an output column (using the query equivalence classes). If this is not possible, reject the view.

3.1.4 Can output expressions be computed?
Checking whether all output expressions of the query can be computed from the view is similar to checking whether the additional predicates can be computed correctly. If the output expression is a constant, just copy the constant to the output. If the output expression is a simple column reference, check whether it can be mapped (using the query equivalence classes) to an output column of the view. For other expressions, we first check whether the view output contains exactly the same expression (taking into account column equivalences). If so, the output expression is just replaced by a reference to the matching view output column. If not, we check whether the expression's source columns can all be mapped to view output columns, i.e. whether the complete expression can be computed from (simple) output columns. If the view fails these tests, the view is rejected.

This algorithm will miss some cases. For instance, we do not consider whether some part of an expression matches a view output expression. Neither do we consider the case when it can be deduced that a query column is constant because of constraints in the where-clause, possibly taking into account check constraints on the column.

3.1.5 Do rows occur with correct duplication factor?
When the query and the view reference exactly the same tables, this condition is trivially satisfied if the view passes the previous tests. The more interesting case occurs when the view references additional tables, which is covered in the next section.

3.2 Views with extra tables
Suppose we have a SPJ query that references tables T_1, T_2 ,..., T_n and a view that references one additional table, that is, tables T_1, T_2 ,..., T_n, S. Under what circumstances can the query still be computed from the view? Our approach is based on recognizing cardinality-preserving joins (sometimes called table extension joins). A join between tables T and S is cardinality preserving if every row in T joins with exactly one row in S. If so, we can view S as simply extending T with the columns from S. An equijoin between all columns in a non-null foreign key in T and a unique key in S has this property. A foreign key constraint guarantees that, for every row t of T, there exists at least one row s in S with matching column values for all non-null foreign-key columns in t. All columns in t containing a null are ignored when validating the foreign-key constraint. It can be shown that all requirements (equijoin, all columns, non-null, foreign key, unique key) are essential.

Now consider the case when the view references multiple extra tables. Suppose the query references tables T_1, T_2 ,..., T_n and the view references m extra tables, that is, it references tables T_1, T_2 ,..., T_n , T_{n+1}, T_{n+2} ,..., T_{n+m}. To determine whether tables T_{n+1},

$T_{n+2}, ..., T_{n+m}$ are joined to tables T_1, $T_2, ..., T_n$ through a series of cardinality preserving joins we build a directed graph, called the foreign-key join graph. The nodes in the graph represent tables T_1, $T_2, ..., T_n$, T_{n+1}, $T_{n+2}, ..., T_{n+m}$. There is an edge from table T_i to table T_j if the view specifies, directly or transitively, a join between tables T_i and T_j and the join satisfies all the five requirements listed above (equijoin, all columns, non-null, foreign key, unique key). To capture transitive equijoin conditions correctly we must use the equivalence classes when adding edges to the graph. Suppose we are considering whether to add an edge from table T_i to table T_j and there is an acceptable foreign key constraint going from columns F_1, F_2, ..., F_n of table T_i to columns C_1, C_2, ..., C_n of T_j. For each column C_i, we locate the column's equivalence class and check whether the corresponding foreign key column F_i is part of the same equivalence class. If the join columns pass this test, we add the edge.

Once the graph has been built, we try to eliminate nodes T_{n+1}, $T_{n+2}, ..., T_{n+m}$ by a sequence of deletions. We repeatedly delete any node that has no outgoing edges and exactly one incoming edge. (Logically, this performs the join represented by the incoming edge.) When a node T_i is deleted, its incoming edge is also deleted, which may make another node deletable. This process continues until no more nodes can be deleted or the nodes T_{n+1}, $T_{n+2}, ..., T_{n+m}$ have been eliminated. If we succeed in eliminating nodes T_{n+1}, $T_{n+2}, ..., T_{n+m}$, the extra tables in the view can be eliminated through cardinality-preserving joins and the view passes this test.

The view must still pass the tests detailed in the previous section (subsumption tests, required output columns available). However, these test all assume that the query and the view reference the same tables. To make them the same, we conceptually add the extra tables T_{n+1}, $T_{n+2}, ..., T_{n+m}$ to the query and join them to the existing tables T_1, $T_2, ..., T_n$ through exactly the same foreign-key joins that were used to eliminate them from the view. Because the joins are all cardinality preserving, this will not change the result of the query in any way. In practice, we merely simulate the addition of extra tables by updating query equivalence classes. We first add a trivial equivalence class for each column in tables T_{n+1}, $T_{n+2}, ..., T_{n+m}$. (We have now added the tables to the from clause of the query.) Next, we scan the join conditions of all foreign-key edges deleted during the elimination process above and apply them to query equivalence classes. This will cause some query equivalence classes to merge. (We have now added the join conditions to the where-clause.) At the end of this process, the (conceptually) modified query references the same tables as the view and the query equivalence classes have been updated to reflect this change. After this modification, all tests described in the previous section can be applied unchanged.

Example 3: This example illustrates views with extra tables.

View:
```
Create view v3 with schemabinding as
Select c_custkey, c_name, l_orderkey,
       l_partkey, l_quantity
From dbo.lineitem, dbo.orders, dbo.customer
Where l_orderkey = o_orderkey
  And o_custkey = c_custkey
  And o_orderkey >= 500
```

Query:
```
Select l_orderkey, l_partkey, l_quantity
From lineitem
Where l_orderkey between 1000 and 1500
  And l_shipdate = l_commitdate
```

We obtain the following equivalence classes and ranges for the view and the query.

View: {l_orderkey, o_orderkey}, {o_custkey, c_custkey}
        { l_orderkey, o_orderkey} $\in$ (500, +∞)

Query: {l_shipdate, l_commitdate}
        {l_orderkey} $\in$ (1000, 1500)

The foreign-key join graph for the view consists of three nodes (lineitem, orders, customer) with an edge from lineitem to orders and an edge from orders to customer. The customer node can be deleted because it has no outgoing edges and one incoming edge. This also deletes the edge from orders to customer. Now orders has no outgoing edges and can be removed.

We then conceptually add orders and customer to the query. The join predicate for the lineitem-to-orders edge is l_orderkey = o_orderkey, which generates the equivalence class {l_orderkey, o_orderkey}. The join predicate for the orders-to-customer edge is o_custkey = c_custkey, which generates the equivalence class {o_custkey, c_custkey}. The updated query equivalence classes and ranges for the query are

Query: {l_shipdate, l_commitdate}, {l_orderkey, o_orderkey},
        {o_custkey, c_custkey};
        { l_orderkey, o_orderkey} $\in$ (1000, 1500)

We then apply the subsumption tests. The view passes the equijoin subsumption test because every view equivalence class is a subset of a query equivalence class. It also passes the range subsumption test because the view range { l_orderkey, o_orderkey} $\in$ (500, +∞) contains the corresponding query range { l_orderkey, o_orderkey} $\in$ (1000, 1500). The compensating predicates are l_orderkey >= 1000 and l_orderkey <= 1500, which can be enforced because l_orderkey is available in the view output. Finally, every output column of the view can be computed from the view output. $\square$

The procedure above ensures that we can "prejoin", directly or indirectly, each extra table in the view to some input table T of the query and the resulting, wider table will contain exactly the same rows as T. This is safe but somewhat restrictive because we only need to guarantee it for the rows actually consumed by the query, not all rows. Here is an example of such a case. Suppose we have a view consisting of tables T and S joined on $T.F=S.C$ where F is declared as a foreign key referencing C and C is the primary key of S. Now consider a selection query on table T with the predicate $T.F > 50$. If $T.F$ is not declared with "not null", the view will be rejected by our procedure. The join of T and S does not preserve the cardinality of T because rows with a null in column $T.F$ are not present in the view. However, for the subset of rows with a non-null $T.F$ value, it does preserve cardinality, which is all that matters because of the null-rejecting predicate $T.F > 50$ in the query. On other words, any row in T containing a null value in $T.F$ will be discarded the query predicate in any case. The algorithm can be modified to handle this case (not yet implemented). All that is required is an additional check when considering whether to add an edge to the foreign-key join graph. A foreign key column allowing nulls is still acceptable if the query contains a null-rejecting predicate on the column (other than the equijoin predicate).

3.3 Aggregation queries and views

In this section, we consider aggregation queries and views. SQL Server allows expressions, as opposed to just columns, in the group-by list both in queries and materialized views. In a materialized view, all group-by expressions must also be in the output

list to ensure a unique key for each row. In addition, the output list must contain a count_big(*) column so deletions can be handled incrementally. The only other aggregation function currently allowed in materialized views is sum.

We treat aggregation queries as consisting of a SPJ query followed by a group-by operation and similarly for views. An aggregation query can be computed from a view if the view satisfies the following requirements.

1. The SPJ part of the view produces all rows needed by the SPJ part of the query and with the right duplication factor.
2. All columns required by compensating predicates (if any) are available in the view output.
3. The view contains no aggregation or is less aggregated than the query, i.e, the groups formed by the query can be computed by further aggregation of groups output by the view.
4. All columns required to perform further grouping (if necessary) are available in the view output.
5. All columns required to compute output expressions are available in the view output.

The first two requirements are the same as for SPJ queries. The third requirement is satisfied if the group-by list of the query is a subset of the group-by list of the view. That is, if the view is grouped on expressions A, B, C then the query can be grouped on any subset of A, B, C. This includes the empty set, which corresponds to an aggregation query without a group-by clause. We currently require that each group-by expression in the query match exactly some group-by expression in the view (taking into account column equivalences). This can be relaxed. As shown in [16], it is sufficient that the grouping expression of the view functionally determine the grouping expressions of the query.

If the query group-by list is equal to the view group-by list, no further aggregation is needed so the fourth requirement is automatically satisfied. If it is a strict subset, then we must add a compensating group-by on top of the view. The grouping expressions are the same as for the query. Because they are a subset of the view grouping expressions and all grouping expressions of the view must be part of the view output, they are always available in the view output. Hence, the third requirement is automatically satisfied.

Testing whether the fourth requirement is satisfied is virtually identical to what was discussed for SPJ queries. The only slight difference occurs when dealing with aggregation expressions. If the query specifies a count(*) and the view is an aggregation view, the count(*) must be replaced by a SUM over the view's count_big(*) column. If the query output contains a SUM(E) where E is some scalar expression, we require that the view contain an output column that matches exactly (taking into account column equivalences). AVG(E) is converted to SUM(E)/count_big(*).

Example 4: This may sound too easy, perhaps causing some readers to wonder whether we handle situations illustrated by the following example view and query.

View:
```
Create view v4 with schemabinding as
Select o_custkey, count_big(*) as cnt
       sum(l_quantity*l_extendedprice)as revenue
From dbo.lineitem, dbo.orders
Where l_orderkey = o_orderkey
Group by o_custkey
```

Query:
```
Select c_nationkey,
       sum(l_quantity*l_extendedprice)
From lineitem, orders, customer
Where l_orderkey = o_orderkey
  And o_custkey = c_custkey
Group by c_nationkey
```

It is easy to see that the query can be computed from the view by joining it with the customer table and then aggregating the result on c_nationkey. However, the view satisfies none of the conditions listed above so one might conclude that we will miss this opportunity. Not so – this is a case where integration with the optimizer helps. For queries of this type, the SQL Server optimizer also generates alternatives that include preaggregation. That is, it will generate the following form of the query expression.

```
Select c_nationkey, sum(rev)
From customer,
     (select o_custkey,
             sum(l_quantity*l_extendedprice) as rev
      From lineitem, orders
      Where l_orderkey = o_orderkey
      Group by o_custkey) as iq
Where c_custkey = o_custkey
Group by c_nationkey
```

When the view-matching algorithm is invoked on the inner query, it easily recognizes that the expression can be computed from v4. Substituting in the view then produces exactly the desired expression, namely,

```
Select c_nationkey, sum(revenue)
From customer, v4
Where c_custkey = o_custkey
Group by c_nationkey
```

4. Fast filtering of views

To speed up view matching we maintain in memory a description of every materialized view. The view descriptions contain all information needed to apply the tests described in the previous section. Even so, it is slow to apply the tests to all views each time the view-matching rule is invoked if the number of views is very large. In this section, we describe an in-memory index, called a *filter tree*, which allows us to quickly discard views that cannot be used by a query.

A filter tree is a multiway search tree where all the leaves are on the same level. A node in the tree contains a collection of (key, pointer) pairs. A key consists of a <u>set</u> of values, not just a single value. A pointer in an internal node points to a node at the next level while a pointer in a leaf node points to a list of view descriptions. A filter tree subdivides the set of views into smaller and smaller partitions at each level of the tree.

A search in a filter tree may traverse multiple paths. When the search reaches a node, it continues along some of the node's outgoing pointers. Whether to continue along a pointer is determined by applying a search condition on the key associated with the pointer. The condition is always of the same type: a key qualifies if it is a subset (superset) of or equal to a given search key. The search key is also a set. We can always do a linear scan and check every key but this may be slow if the node contains many keys. To avoid a linear scan, we organize the keys in a lattice structure, which allows us to find all subsets (supersets) of a given search key easily. We call this internal structure a *lattice index*.

We describe the lattice index in more detail in the next section and then explain the partitioning conditions applied at different levels of the tree.

4.1 Lattice index

The subset relationship between sets imposes a partial order among sets, which can be represented as a lattice. As the name indicates, a lattice index organizes the keys in a graph that correspond to the lattice structure. In addition to the (key, downward pointer) pair described above, a node in the lattice index contains two collections of pointers, superset pointers and subset pointers. A superset pointer of a node *V* points to a node that represents a minimal superset of the set represented by *V*. Similarly, a subset pointer of *V* points to a node that represents a maximal subset of the set represented by *V*. Sets with no subsets are called roots and sets without supersets are called tops. A lattice index also contains an array of pointers to tops and an array of pointers to roots. Figure 1 shows a lattice index storing eight key sets.
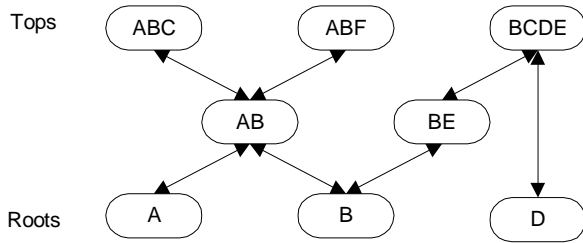
Figure 1: Lattice index storing eight key sets:
A, B, D, AB, BE, ABC, ABF, BCDE

Searching a lattice index is a simple recursive procedure. Suppose we want to find supersets of AB. We start from the top nodes, where we find that ABC and ABF are supersets of AB. From each qualifying node, we recursively follow the subset pointers, at each step checking whether the target node is subsets of AB. AB is acceptable but none if its subset nodes are. The search returns ABC, ABF, and AB. Note that AB is reached twice, once from ABC and once from ABF. To avoid visiting and possibly returning the same node multiple times, the search procedure must remember which nodes have been visited. The algorithm for finding subsets is similar; the main difference is that the search starts from root nodes and proceeds upwards following superset pointers.

Due to space constraints, we will not describe insertion and deletion algorithms for lattice indexes. They are not complex but some care is required with the details.

4.2 Partitioning conditions

A filter tree recursively subdivides the set of views into smaller and smaller non-overlapping partitions. At each level, a different partitioning condition is applied. In this section, we describe the partition conditions used in our prototype.

4.2.1 Source table condition

Views that lack some of the tables required by the query can be discarded. This in captured by the following condition.

Source table condition: A query cannot be computed from a view unless the view's set of source tables is a superset of the query's set of source tables.

We build a lattice index using the view's set of source tables as the key. Given a query's set of source tables, we search the lattice index for partitions containing views that satisfy this condition.

4.2.2 Hub condition

Recall the algorithm explained in section 3.2 for eliminating extra tables from a view. In that section, it was sufficient to have the algorithm reduce the set of source tables to the same set as that of the query. However, we can let the algorithm run until no further tables can be eliminated from the view, thereby reducing the remaining set of tables to the smallest set possible. We call the remaining set the *hub* of the view. As discussed in section 3.2, a view cannot be used to answer a query unless we can eliminate all extra tables through cardinality-preserving joins. The hub cannot be reduced further so clearly we can disregard any view whose hub is not a subset of the query's set of source tables. This observation gives us the following condition.

Hub condition: A query cannot be computed from a view unless the hub of the view is a subset of the query's set of source tables.

The previous condition gave us a lower bound on a view's set of source tables while this condition gives us an upper bound. We again use a lattice index structure but this time with view hubs as the key instead of the complete set of source tables. Given a query, we then search the index for nodes whose key is a subset of the query's set of source tables.

The algorithm outlined above for computing view hubs tends to produce hubs that are unnecessarily small because it takes into account only foreign-key joins. It can be improved by also taking into account other predicates. Suppose *T* is a table that would be eliminated from the hub by the algorithm above. Let *T.C* be a column not participating in any non-trivial equivalence class. If *T.C* is referenced in a range or other predicate, we can leave *T* in the hub. The join is no longer guaranteed to be cardinality preserving because the predicate on *T.C* may reject some rows in *T*. The only way the reference to *T.C* can be rerouted to another column is if the query contains a column equality predicate involving *T.C*. However, if that is the case, then *T* must also be among the query's source tables so leaving *T* in the hub will not cause the view to be rejected.

4.2.3 Output column condition

Assume for the moment that the output lists of queries and views are all simple column references. We will deal with more complex output expressions separately. As stated earlier, a query cannot be computed from a view unless all its output expressions can be computed from the output of the view. However, this does not mean that a query output column has to match an output column because of equivalences among columns. The following example illustrates what is required.

Example 6:
Suppose the query outputs columns A, B, and C. Furthermore, suppose the query contains column equality predicates generating the following equivalence classes: {A, D, E}, {B, F} {C}. The columns within an equivalence class all have the same value so they can be used interchangeably in the output list. We indicate this choice by writing the output list as {A, D, E}, {B, F} {C}.

Now consider a view that outputs {<u>A</u>, D, G}, {<u>E</u>}, {<u>B</u>} and {<u>C</u>, H} where the column equivalences have been computed from the column equality predicates of the view. The columns that are actually included in the output list are indicated by underlining. Logically, we can then treat the view as outputting all of the columns, that is, as if its output list were extended to A, D, G, E, B, C, H.

The first output column of the query can be computed from the view if at least one of the columns A, D, or E exists in the view's extended output list. In this case, all of them do. Similarly, the second column can be computed if B or F exists in the extended output list. B does (but F does not). Finally, the third output column requires C, which also exists in the extended output list. Consequently, the query output can be computed from the view. □

As this example illustrated, to correctly test availability of output columns we must take into account column equivalences in the query and in the view. We do this by replacing each column reference in the output list by a reference to the columns equivalence class. For a view, we then compute the extended output list by including every column in the referenced equivalence classes. We can now state the condition that must hold.

Output column condition: A view cannot provide all required output columns unless, for each equivalence class in the query's output list, at least one of its columns is available in the view's extended output list.

To exploit this condition, we build lattice indexes using the extended output lists of the views as keys. Given a query, we search the index recursively beginning from the top nodes. A node qualifies if its extended output list satisfies the condition above. If so, we follow its subset pointers. If not, it is not necessary to follow the pointers because, if a view *V* does not provide all required columns, neither does any view whose extended output list is a subset of *V*'s extended output list.

4.2.4 Grouping columns
An aggregation query cannot be computed from an aggregation view unless the query's grouping columns are a subset of the view's grouping columns, again taking into account column equivalences. This is exactly the same relationship as the one that must hold between the output columns of the query and the view. Consequently, we can extend the views' grouping lists in the same way as for output columns and build lattice indexes on the extended grouping lists. For completeness, here is the condition that must hold between grouping columns.

Grouping column condition: An aggregation query cannot be computed from an aggregation view unless, for each equivalence class in the query's grouping list, at least one of its columns is present in the view's extended grouping list.

4.2.5 Range constrained columns
A query cannot be computed from a view that specifies range constraints on a column that is not range constrained in the view, again taking into account column equivalences. We associate with each query and view a range constraint list, where each entry references a column equivalence class. A column equivalence class is included in the list if it has a constrained range, that is, at least one of the bounds has been set. Next, we compute an extended constraint list in the same way as for output columns but this time for the query but not the view. We can now state the condition that must hold.

Range constraint condition: A query cannot be computed from a view unless, for each equivalence class in the view's range constraint list, at least one of its columns is present in the query's extended range constraint list.

Note that the extended range constraint list is associated with the query, and not the view. Hence, lattice indexes on the extended constraint lists of views, mimicking the indexes on output columns and grouping columns, cannot be used. However, we can build lattice indexes based on a weaker condition involving a reduced range constraint list. The reduced range constraint list contains only columns that reference trivial equivalence classes, i.e. columns that are not equivalent to any other columns.

Weak range constraint condition: A query cannot be computed from a view unless the view's reduced range constraint list is a subset of the query's extended range constraint list.

When building a lattice index based on this condition, the complete constraint list of a view is included and used as the key of a node but the subset-superset relationship is computed solely based on reduced constraint lists. A search starts from the roots and proceeds upwards along superset edges. If a node passes the weak range constraint condition, we follow its superset pointers but the node is returned only if it also passes the range constraint. If a node fails the weak range constraint condition, all of its superset nodes will also fail so there is no need to check them.

4.2.6 Residual predicate condition
Recall that we treat all predicates that are neither column-equality predicates nor range predicates as residual predicates. The residual subsumption test checks that every residual predicate in the view also exists in the query, again taking into account column equivalences. Our implementation of the test uses a matching function that compares predicates converted to text strings, omitting column references, and then matches column references separately. We associate with each view and query, a residual predicate list containing just the text strings of the residual predicates. Then the following condition must hold.

Residual predicate condition: A query cannot be computed from a view unless the view's residual predicate list is a subset of the query's residual predicate list.

For filtering purposes, we then build lattice indexes using the residual predicate lists of the views as keys. Given a query's residual list, we search for nodes whose key is a subset of the query's residual list.

4.2.7 Output expression condition
Output expressions are handled in much the same way as residual predicates. We convert the expressions to text strings, omitting column references, and associate with each view and query an output expression list consisting of the text strings of its output expressions. We can then build lattice indexes based on the following condition.

Output expression condition: A query cannot be computed from a view unless its (textual) output expression list is a subset of the view's (textual) output expression list.

A search then looks for nodes whose key is a superset of the query's (textual) output expression list. The condition is conservative in the sense that we ignore the possibility of computing an expression from "scratch" using plain columns or precomputed parts. The condition can be weakened to cover this possibility but the details are beyond the scope of this paper.
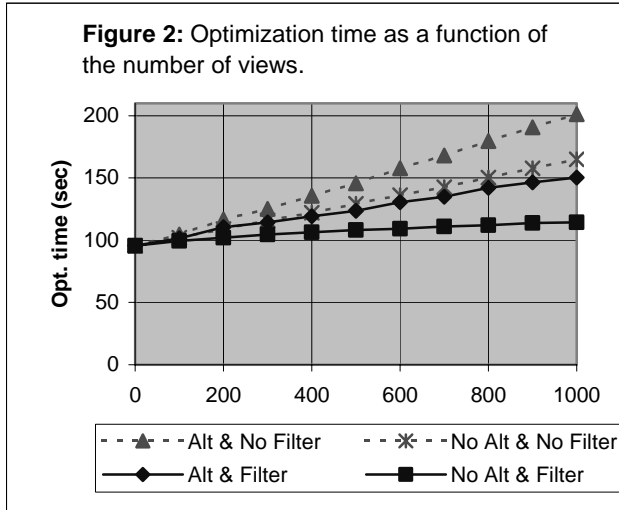
4.2.8 Grouping expression condition
Expressions in the grouping clause can be handled in the same way as expressions in the output list. For completeness, we state the condition.

Grouping expression condition: An aggregation query cannot be computed from an aggregation view unless its (textual) grouping expression list is a subset of the view's (textual) grouping expression list.

4.3 Summary
As we saw, each condition above can be the basis for a lattice index subdividing a collection of views. The conditions are independent and can be composed in any order to create a filter tree. For instance, we can create a filter tree where the root node partitions the views based on their hubs and the second level nodes further subdivide each partition according to the views' extended

Figure 2: Optimization time as a function of the number of views.

Figure 3: Total increase in optimization time and time spent in view-matching rule

output column lists. We can stop there or add more levels using any other conditions. Our implementation uses a filter tree with eight levels. From top to bottom, the levels are: hubs, source tables, output expressions, output columns, residual constraints, and range constraints. For aggregation views, there are two additional levels: grouping expressions and grouping columns.

5. Experimental results

We have implemented the view-matching algorithm, including the filter tree, in SQL Server. One goal was to keep optimization time low, even when the number of materialized views is high. We ran a series of experiments that measured the increase in optimization time and the benefits of using the filter tree.

The views used in the experiments were randomly generated SPJG views over the TPC-H/R database. Each view was generated by randomly selecting a table and joining in additional tables randomly through foreign key equijoins. Then range predicates were added on randomly selected columns until the estimated cardinality of the SPJ part of the result was within 25-75% of the largest table included. Output columns were also selected randomly. About 75% of the views were aggregation views, using randomly selected output columns for grouping. Any remaining (numerical) output column was used as the argument for a SUM. A parameter file specified the frequency with which a table was chosen as the initial table, the frequency with which a foreign key was select for a join, the frequency with which a column received a range predicate, and the frequency with which a column was chosen as an output column. We generated a total of 1000 materialized views. Queries were generated in the same way but with a different seed for the random number generator and restricting the cardinality of the result to within 8-12% of the largest table used in the query. 40% of the queries referenced two tables, 20% referenced three, 17% referenced four, 13% referenced five, 8% referenced six, and 2% referenced seven tables.

All experiments were run on a machine with 128 MB of memory and a 700MHz Pentium processor. The database was TPC-H at scale factor 0.5 (500MB) with primary keys and foreign keys defined. The scale factor does not affect optimization time.

Figure 2 shows the total optimization time for 1000 randomly generated queries varying the number of materialized views. The bottom two lines were measured with the filter tree enabled. The line labeled Alt&Filter shows the total optimization time. The optimization time increases linearly with the number of views. For 1000 views, it increased by about 60%. The absolute time is still
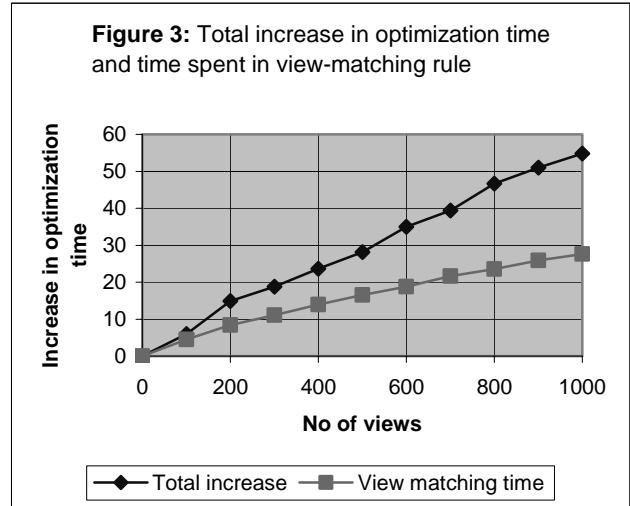
low; on average, only 0.15 seconds per query. The increase in optimization time is caused by two factors: the time for finding and checking candidate views and the time spent in the optimizer processing the substitutes produced. In an attempt to separate the two components, we also ran the optimization without introducing substitutes. That is, the view-matching algorithm performed its normal analysis but always returned without producing substitutes. The bottom line shows the optimization time when no substitutes were produced. This shows that approximately 35% of the time was spent finding and checking views and the rest spent on further processing of the substitutes produced.

The top two lines (dashed) show the optimization time with the filter tree disabled. Using the filter tree to find candidate views reduces the optimization time significantly. With 1000 views, the optimization time increases by about 110% when the filter tree is disabled, as opposed to about 60% when it is enabled.

Estimating the time spent on view matching simply by never producing substitutes is not entirely accurate. Introducing substitutes increases the amount of time spent on view matching because the substitutes will result in additional expressions being generated, on which view matching will also be attempted. In other words, introducing substitutes increases the number of invocations of the view-matching rule. We instrumented the code to measure more accurately the total time spent in the view-matching rule. The results are plotted in Figure 3. For 1000 views, about half of the increase in the optimization time originated in the view-matching code. When there are fewer views, fewer substitutes are generated on average and a larger fraction of the time is spent on view matching. When there are very few views, most invocations will produce no substitutes at all, in which case all of the time is spent on (unsuccessful) view matching.

We also measured how effectively the filter tree reduces the number of views that have to be tested more carefully. The view-matching rule may be invoked on expressions that are not acceptable. The expression may contain, for example, a subquery. If so, the rule returns immediately without checking any views. These unsuccessful invocations are not counted in the subsequent summary. If this first test is successful, we begin looking for acceptable views. Without the filter tree, every view would then have to be checked. Using the filter tree, most views are discarded quickly and only a small set of candidate views remain. The filter tree was highly effective on our workload; it consistently reduced the candidate set to less than 0.4% of the views on average. The actual numbers were 0.29% for 100 views and 0.36% for 1000 views.

On average, between 15% and 20% of the candidate views passed the further checking and produced substitutes. The average number of substitutes produced per invocation was low: 0.04 for 100 views, rising to 0.59 for 1000 views. This means that most invocations found no matching views at all. However, the view-matching rule is invoked multiple times per query. On our workload, the average number of invocations was between 17.8 and 17.9 times. The average number of substitutes produced *per query* ranged from 0.7 for 100 views to 10.5 for 1000 views. In other words, on average, over ten different rewrites of the query using views were considered by the optimizer.

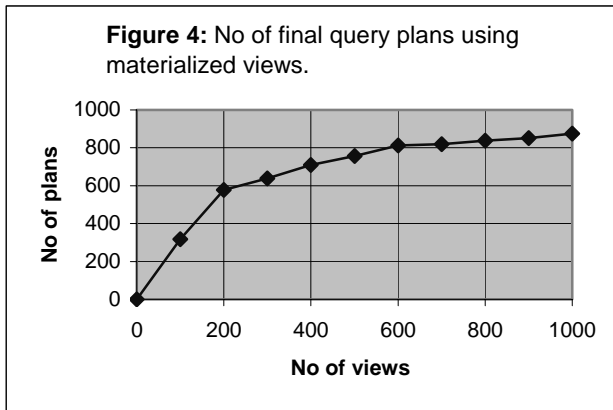**Figure 4:** No of final query plans using materialized views.

Figure 4 plots how many of the final execution plans actually used materialized views. Considering that views and queries were randomly generated, the curve behaves as expected, namely, the benefit of additional views decreases with the number of views already in the system. Already with 200 views, the best execution plan used views for about 60% of the queries. With 1000 views, this increased to about 87%. All this shows is that view matching is working, finding applicable views and producing improved plans.

6. Related work

Algorithms proposed in the literature for solving the view-matching problem differ in several respects:

- Class of query expressions considered.
- Class of views supported.
- Class of substitute expressions considered.
- Whether and how multiple substitutes are handled.
- Whether bag semantics or set semantics is assumed.
- Which constraints, if any, are exploited.

However, a view-matching algorithm solves only part of the optimization problem. The algorithm must also be integrated into the overall optimization process so that all valid rewrites using views are generated and evaluated in the same way as rewrites not using views. Many papers on query optimization using views ignore this aspect completely, some consider only the issue of finding all valid rewrites, and some propose selecting a best rewrite by heuristic rules.

Larson and Yang [10] were the first to describe a view-matching algorithm for SPJ queries and views. They assumed set semantics, considered only single-table substitutes and did not consider constraints. Their algorithm sometimes produced substitutes not expressible in SQL or standard relation algebra. Our view-matching algorithm has much in common with the approach taken in this paper. In a subsequent paper [17], Larson and Yang extended their algorithm to consider substitutes consisting of joins

of views. The focus was on finding all valid rewrites but the paper did not discuss how to select the best rewrite.

Chaudhuri et al. [4] published the first paper on incorporating the use of materialized views into query optimization, in their case, a System-R style optimizer using dynamic programming. They dealt with a limited form of SPJ queries and views and assumed set semantics. Their matching algorithm does not consider predicate subsumption or constraints. They make use of a map table that specifies which subexpressions of the query can be substituted by one of the available views. The map table is computed before join enumeration begins. During join enumeration, rewrites of the expression under consideration are located in the map table and their cost estimated in the normal way. A small experimental study on ten queries showed a 50% increase in optimization time with only six materialized views.

Levy, Mendelzon and Sagiv [11] studied the complexity of rewriting SPJ queries using views and proved that many related problems are NP-complete. This is true even for the simplest version of the problem, determining whether there exists a rewrite of a conjunctive query with no built-in predicates. (In SQL terms: the where-clause is limited to a conjunction of column-equality predicates and predicates equating a column to a constant.)

Gupta, Harinaryan and Quass [9] introduced a generalized projection operator that captures duplicate elimination, aggregation, grouping, and duplicate-preserving projection in a unified framework. The paper focuses on transformation rules for this new operator. They use the transformation rules to generate expressions that can be rewritten using aggregation views. Substitute expressions are limited to single views and do not take into account subsumption of selection predicates and grouping expressions.

Srivastava et al. [15] present a view-matching algorithm for SPJG queries and views. They assume bag semantics but do not consider constraints. They take into account subsumption of selection predicates, though to what extent is not clear because the algorithm for testing subsumption is not fully described. In addition to single-view substitutes, they also consider substitutes consisting of unions of views (but only for SPJ views). Not surprisingly, our view-matching algorithm is based on conditions similar to theirs. However, they require that the view and the expression it replaces reference exactly the same tables, which we do not. The paper does not discuss how to select among multiple rewrites nor other issues related to optimizer integration. No experimental results are provided. Chang and Lee [3] recognized that a view can sometimes be used even if it contains extra tables. However, we cannot tell from their paper what tests they apply to verify this.

The requirement that the grouping columns of a query must be a subset of the grouping columns of a view is sufficient but not necessary. Yan and Larson [16] proved that it is sufficient that the grouping columns of the view functionally determine the grouping columns of the query. This property is important in OLAP environments because lower levels in a dimension hierarchy functionally determine higher levels in the hierarchy. Park, Kim and Lee [13] exploited these functional dependencies to define a partial order (lattice) among OLAP queries and views. The key observation is that a view is potentially useful only if it is at the same or lower level than the query in the lattice. Their algorithm may produce substitutes containing unions and joins. They use heuristic rules for selecting the best rewrite. No experimental results are reported. Our algorithm can easily be extended to consider functional dependencies in a dimension hierarchy but SQL Server does not currently have any way to specify such functional dependencies. However, if a dimension hierarchy is implemented

as a set of tables connected by foreign keys, the functional dependencies are implied by foreign keys and will be exploited.

Oracle was the first commercial database system to support materialized views[2]. Views may contain joins and aggregation but no selections. The query rewrite algorithm is briefly described in the Oracle 8i documentation. The algorithm considers replacing part or all of a PSJG query by a materialized view. Views with extra tables joined in through cardinality-preserving joins are considered and functional dependencies among levels in dimension hierarchies are taken into account. Multiple views may be used in a substitute expression. If multiple rewrites are possible, one is selected by heuristic rules based on the size of the views involved. The selected rewrite is then optimized and compared to the best plan with no materialized views. No information is provided about the speed or scalability of the query rewrite algorithm.

Zaharioudakis et al. [18] describe a view-matching algorithm implemented in DB2 UDB. The algorithm performs a bottom-up matching of query graphs but does not require an exact match. Compensating nodes, for instance, selections and further aggregation, may be introduced along the way. Cardinality-preserving joins are recognized as are subsumption of selection predicates (though the exact algorithm is not described). Complex aggregations (rollup, cube, grouping sets) are also handled. The paper does not specify precisely what substitute expressions are considered nor does it say whether multiple rewrites are considered. Integration with the cost-based optimizer is not addressed and no experimental results are reported.

Pottinger and Levy [14] considered the view-matching problem for conjunctive SPJ queries and views in the context of data integration. In that context, a view describes data available from some data source and a query must be answered using only the set of views. It may not be possible to find a rewriting that is equivalent to the query. Instead, the objective is to find a maximally-contained rewriting, which provides the best answer possible (most rows). The rewriting consists of a union of combinations of views joined together, where each such combination contributes some rows to the result. They present a new algorithm, called the MiniCon algorithm, and compare its performance to two earlier algorithms. The MiniCon algorithm reduces the number of view combinations by taking into account availability of join columns. That is, if a view V must be joined to a table T (or another view containing T) to provide complete rows but the required join columns are not among its output columns, then V can be discarded. They then show experimentally that exploiting this observation provides significant speedup. It is not entirely clear to us how to translate their findings into a query optimization context because of the difference in objectives.

7. Summary and extensions

Materialized views can reduce query processing time very significantly but only if the query optimizer is able to find applicable views quickly. We presented an efficient view-matching algorithm for SPJG views and described its integrated into a transformation-based optimizer. We also presented an index structure, called a filter tree, which greatly speeds up the search for applicable views. Experimental results obtained from an implementation in Microsoft SQL Server showed that the algorithm is fast and scales to very large numbers of views. With 1000 views in the system, average optimization time remained as low as 0.15 seconds per query, even though an average of ten different substitutes using views were generated. To the best of our knowledge, no other algorithm has achieved this level of speed and scalability.

We are planning to extend the algorithm to cover a broader class of views and substitute expressions. Some extensions are easy, requiring only local changes. For instance, the tests applied to determine whether a view contains all rows needed by a query are somewhat conservative and do not fully exploit constraints. The tests can be refined in several ways. We indicated how to take into account check constraints. Support for predicates containing ORs can also be added relatively easily. Improved reasoning about when a scalar expression can be computed from other scalar expressions would also be desirable.

Some extensions are a wee bit more complex. The current algorithm produces only single-table substitutes. We plan to add union substitutes and substitutes with base table backjoins to the repertoire. Union substitutes cover the case when all rows needed are not available from a single view but can be collected from several views. Overlapping views together with SQL's bag semantics complicate the issue. If the same rows can be obtained from multiple views, we have to make sure that they appear in the result with the right duplication factor. Base table backjoins cover the case when a view contains all tables and rows needed but some columns are missing. In that case, it may be worthwhile backjoining the view to a base table to pull in the missing columns, especially of the missing columns are available from an index.

8. References

[1] S. Agrawal, S. Chaudhuri, V. R. Narasayya: Automated Selection of Materialized Views and Indexes in SQL Databases. VLDB 2000: 496-505

[2] R.G. Bello, K. Dias, J. Feenan, J. Finnerty, W.D. Norcott, H. Sun, A. Witkowski, M. Ziauddin, Materialized Views in Oracle, VLDB 1998, 659-664.

[3] J. Chang and S. Lee, Query Reformulation Using Materialized Views in Data Warehousing Environment, First ACM Int'l Workshop on Data Warehousing and OLAP (DOLAP), 1998, 54-59.

[4] S. Chaudhuri, S. Krishnamurthy, S. Potamianos, K. Shim, Optimizing Queries with Materialized Views, ICDE 1995, 190-200.

[5] S. Cohen, W. Nutt, A. Serebrenik, Rewriting Aggregate Queries Using Views, PODS, 1999, 155-166.

[6] G. Graefe, The Cascades Framework for Query Optimization, Data Engineering Bulletin, 18(3), 1995, 19-29.

[7] G. Graefe and W. J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search, ICDE 1993, 209-218.

[8] S. Grumbach, M. Rafanelli, L. Tininini, Querying Aggregate Data, PODS 1999, 174-184.

[9] A. Gupta, V. Harinarayan, D. Quass, Aggregate Query Processing in Data WareHousing Environments, VLDB 1995, 358-369.

[10] P.-Å. Larson and H. Z. Yang, Computing Queries from Derived Relations, VLDB 1985, 259-269.

[11] A. Levy, A. O. Mendelzon, Y. Sagiv, D. Srivastava, Answering Queries Using Views, PODS 1995, 95-104.

[12] W. Nutt, Y. Sagiv, S. Shurin, Deciding Equivalence Among Aggregate Queries, PODS 1998, 214-223.

[13] C.-S. Park, M. H. Kim and Y.-J. Lee, Rewriting OLAP Queries Using Materialized Views and Dimension Hierarchies in Data Warehouses, Korea Advanced Institute of Science and Technology, CS/TR-2000-156.

[14] R. Pottinger, A. Levy, A Scalable Algorithm for Answering Queries Using Views, VLDB 2000, 484-495.

[15] D. Srivastava, S. Dar, H.V. Jagadish, A. Levy, Answering Queries with Aggregation Using Views, VLDB 1996, 318-329.

[16] W. P. Yan and P. -Å. Larson, Eager Aggregation and Lazy Aggregation, VLDB 1995, 345-357.

[17] H. Z. Yang and P. -Å. Larson, Query Transformation for PSJ Queries, VLDB 1987, 245-254.

[18] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata, Answering Complex SQL Queries Using Automatic Summary Tables, SIGMOD 2000, 105-116