Partially Materialized Views

Jingren Zhou Per-Åke Larson Jonathan Goldstein
{jrzhou, palarson, jongold}@microsoft.com

June 2005

Technical Report
MSR-TR-2005-77

We propose a new type of materialized view called a *partially materialized view*. A partially materialized view only materializes some of the rows, for example, the most frequently accessed rows, which reduces storage space and view maintenance effort. One or more control tables are associated with the view and define which rows are currently materialized. As a result, one can easily change which rows of the view are stored and maintained. We show how to extend view matching and maintenance algorithms to partially materialized views and outline several potential applications of the new view type. Experimental results in Microsoft SQL Server show that compared with fully materialized views, partially materialized views have lower storage requirements, better buffer pool efficiency, better query performance, and significantly lower maintenance costs.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Judicious use of materialized views can speed up the processing of some queries by several orders of magnitude. The idea of using materialized views to speed up query processing is more than twenty years old [19, 23] and all major database systems (DB2, Oracle, SQL Server) now support materialized views [2, 24, 5]. The support included in those systems consists of computing, materializing, and maintaining all rows of the view result and will be referred to as *fully* materialized views.

Since fully materialized views store and maintain all their rows, storage cost may be high for large views and maintenance can be costly for frequently updated views. If only a small subset of the fully materialized view is used over a period of time, disk storage is wasted for the unused records and many records that are never used are unnecessarily kept up to date.

In current systems, it is very expensive to modify the definition of a materialized view to adapt to a changing workload. The old view would have to be dropped and a new view created from scratch. Incremental view adaption techniques [11] could be applied in some circumstances but are not currently supported. However, even if such support were available, all query plans referencing the old materialized view would have to be recompiled. To not miss opportunities, query plans referencing any of the view's input tables would also need to be recompiled.

In this paper we introduce partially materialized views, that is, views where only some of the rows are materialized. For example, instead of materializing all rows of the view, only the most frequently requested records might be materialized. Which rows are currently materialized is specified by one or more *control tables* associated with the view. Changing which rows are materialized can be done dynamically (at run time) simply by modifying data in a control table. We illustrate the basic idea of partially materialized views by an example.

Example 1. Consider the following parameterized query against the TPC-H/R database that finds information about all suppliers for a given part.

Q_1:
```
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and p_partkey = @pkey
```

Suppose Q_1 is executed frequently but its current response time is deemed too high for the application's needs. To speed up the query, we could define a materialized view V_1 that precomputes the join.

V_1:
```
create view V1 as
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
```

If the view result is clustered on (p_partkey, s_suppkey), a three-table join is replaced by a very efficient index lookup of a clustered index.

V_1 materializes the complete join, so it may be quite large. If there are 200,000 parts and each part has on average four suppliers, the view would contain 800,000 rows. Now consider a scenario where the access pattern is highly skewed and, in addition, *changes* over time. Suppose 1,000 parts account for 90% of the queries and this subset of parts changes seasonally - some parts are popular during summer but not during winter and vice versa. In this scenario, we could get 90% of the benefit of the materialized view by materializing only 0.5% of the rows. This would both reduce overhead for maintaining the view during updates and also save storage space. It would also improve buffer pool utilization and possibly computational costs because the most frequently required rows are packed densely on a few pages. However, this is not possible with today's materialized view technology because static predicates are inadequate for describing the seasonally *changing* contents of the materialized view.

Partially materialized views are ideally suited for situations like this. To handle our example query, we create a control table `pklist` and a partially materialized view PV_1 whose content is controlled by `pklist`.

PV_1:
```
create table pklist(partkey int primary key)

create view PV1 as
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and exists (select * from pklist pkl
  where p_partkey = pkl.partkey)
```

PV_1 is initially empty. To materialize information about a part, all we need to do is to add its key to `pklist`. Interestingly, information about parts without suppliers can also be cached - the part key occurs in `pklist` but there are no matching tuples in PV_1. Normal incremental view maintenance will correctly update PV_1 – nothing special is required even though the view contains a subquery. The *exists* subquery can be converted to an inner join because partkey is the primary key of `pklist` and consequently the subquery will return at most one tuple.

Converting the subquery to an inner join produces the following equivalent version of the view definition. We now see that PV_1' is defined by a select-project-join (SPJ) expression and as such can be incrementally maintained.

PV_1':
```
create view PV1' as
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier, pklist pkl
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and p_partkey = pkl.partkey
```

The content of PV_1 can be changed dynamically by updating the control table. Q_1 can be answered from the view if the key of the desired part is found in `pklist`. To exploit the view safely, the optimizer produces a query plan that first checks whether the desired part key exists in `pklist` *at run-time*. If the part key exists, the plan evaluates the query using a simple select against PV_1. Otherwise, the query is evaluated using the base tables.

A possible dynamic query plan is illustrated in Figure 1. The *ChoosePlan* operator first evaluates the guard predicate shown on the right. (The operator tree for evaluating the guard predicate is not shown.) If it evaluates to true, the partial view contains the required rows and the left branch using the view is executed. Otherwise, the right branch computing the result from base tables is executed.

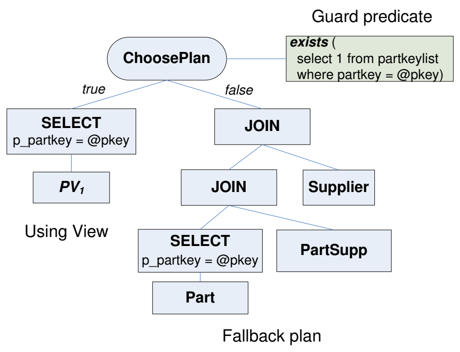


Figure 1: Dynamic execution plan for Q_1

The rest of this paper is organized as follows. After a brief review of related work in Section 2, we introduce the general form of a partially materialized view, and show how to extend view matching and maintenance algorithms to partially materialized views in Section 3. We describe several types of control tables and show how to create views with more complex control designs in Section 4.

We also outline four potential applications of partially materialized views in Section 5. Each application area will require its own policies for determining what rows to materialize and when but the design of such policies is outside the scope of this paper. This paper is focused on mechanisms for partially materialized views, not policies needed for using them effectively in different scenarios.

Experimental results in Section 6 show that partially materialized views have many benefits, such as better buffer pool efficiency, better query performance with fewer rows processed, lower maintenance costs and lower storage requirements. We conclude in Section 7.

2 Related Work

The problem of materialized view matching and maintenance has received considerable attention in the research community for the last two decades. However, virtually all researchers have only considered fully materialized views.

In his paper on executing nested queries [6], Graefe discusses the idea of caching the result of the inner operand in a nested-loop join. He describes the idea of caching the results of multiple invocations with different correlation values (parameter bindings). An additional control index remembers for which correlation values results are currently cached. Traditionally such caches have been temporary and discarded at the end of the execution of the current query. However, Graefe suggests that such caches could be made persistent so that they can be used (and populated) by multiple queries. Each cache would be exposed as a materialized view with a control table that describes its current content. This is precisely the simplest form of a partially materialized view discussed in this paper, namely, a partially materialized view with one equality control table. We also consider other types of control tables, views with multiple control table, and views that directly or indirectly share control tables.

Answering queries using views has been studied in [19, 23, 4, 21]. Larson and Yang [19, 23] were the first to describe view matching algorithms for SPJ queries and views. Srivastava et al. [21] proposed a view-matching algorithm for queries and views with aggregation. Chaudhuri et al. [4] considered using materialized views in a System-R style query optimizer. A thorough survey of work on answering queries using views can be found in [13].

Incremental view maintenance has been studied in [3, 12, 9, 20]. They all use the *update delta* paradigm - compute a set of changed tuples (inserted or deleted) that are then used to refresh the materialized view.

Materialized views have now been adopted in all major commercial database systems. Oracle was the first commercial database system to support materialized views [2]. Zaharioudakis et al. [24] described a bottom-up view matching algorithm implemented in IBM DB2. Goldstein and Larson [5] presented algorithms to matching SPJG views in Microsoft SQL Server.

Dynamic plans were proposed by Graefe and Ward in [8]. They have been used in the context of mid-tier caching in [1, 18, 10] and probably also in other contexts. At least one commercial system, Red Brick Warehouse [16], implements dynamic plans.

The term "partially materialized views" was also used in [14, 15] but there it referred to views where not all columns of the view's input tables are retained in the view. This meaning of the term is unrelated to our use of the term. Valluri proposed something called "partially materialized partitioned views" in [22] where a view is first statically partitioned into three parts and only two of the partitions are materialized. Although the name is similar, the approach is very different from our partially materialized views.

3 General Form of Partially Materialized Views

In this section, we define partially materialized views and describe how to modify existing view matching and view maintenance algorithms to work with partially materialized views. For simplicity of presentation, we use a partially materialized view with a single control table as an example. The techniques presented here are also applicable to other more complex partially materialized views in the following sections.

3.1 View Definitions

Let V_b denote the query expression defining a standard SPJG view and P_v its select-join predicate. V_b is assumed to satisfy all the restrictions imposed by the system on materialized views. We wish to create a partially materialized view with V_b as the base and have materialization controlled by a predicate $P_c(p_1, p_2, \ldots, p_n)$, called a *control predicate*, where $p_1, p_2, \ldots, p_n$ are parameters.

Control predicate P_c can only reference *non-aggregated output columns* of V_b. This restriction is important for view matching and for view maintenance as we shall see in Section 3.2.2 and Section 3.3.

We define a control table T_c with n columns, one for each parameter $p_1, p_2, \ldots, p_n$. The declarations of the control table and a partially materialized view V_p are shown below. The notation *typeof*(p_i) is shorthand for "of a type matching the type of parameter p_i".

```
create table Tc(col1 typeof(p1), col2 typeof(p2),
    ..., coln typeof(pn))

create view Vp as
select Vb.* from Vb
where exists (select 1 from Tc
    where Pc(Tc.col1, Tc.col2, ..., Tc.coln))
```

The ***exists*** clause in the definition restricts the rows actually materialized in V_p to those satisfying the control predicate P_c for some parameter combination currently stored in T_c. Hence, by adding and deleting rows from T_c, we control the contents of V_p.

The partially materialized view PV_1 defined earlier has the following components.

```
Vb:
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey

Pv:       (p_partkey = sp_partkey) ∧
          (sp_suppkey = s_suppkey)
Pc(p1):   (p_partkey = p1)
Tc:       pklist(partkey int)
```

3.2 View Matching

How can we determine whether a query expression can be computed from a partially materialized view? A view matching algorithm for fully materialized views is described in [5]; we show how to extend the algorithm to work with partially materialized views. Only one step in the algorithm needs to be modified, namely, the step showing that all rows required by the query exist in the view.

For regular views, this condition can be tested at optimization time. But for partially materialized views, some of the testing has to be postponed to execution time. We call the test evaluated at execution time a *guard condition.* In this paper, we assume that *guard conditions are limited to checking whether one or a few covering parameter values exist in the control table.* If the desired parameter values are found in the control table, then all tuples associated with those parameter values are currently materialized. At optimization time, we construct the guard condition so that the query is guaranteed to be covered by the view if the guard condition evaluates to true. The actual evaluation of the guard condition is delayed until execution time. The query plan must also contain an alternative subplan, called a *fallback plan*, that computes the query expression from other input sources in case the guard condition evaluates to false.

This may sound rather complicated, but in practice it is quite straightforward. Take query Q_1 and view PV_1 as an example. The actual value of `@pkey` is known at execution time. If the value of `@pkey` is found in `pklist`, we know that the supplier information for the desired part exists in PV_1, which is precisely what the query requires. So the guard condition is simply `exists(select * from pklist where partkey = @pkey)`.

We first deal with select-project-join (SPJ) views and queries in Section 3.2.1. Partially materialized views with aggregation are covered in Section 3.2.2.

3.2.1 SPJ Views and Queries

Let V_p be a partially materialized SPJ view with base view V_b and control predicate P_c. Denote the join-select predicate of V_b with P_v. Consider a SPJ query Q over the same tables as V_b and denote its combined select-join predicate by P_q.

If V_p were a regular view, containment would be tested simply by the condition $P_q \Rightarrow (P_v \wedge P_c)$. However, for a partially materialized view this condition would never be satisfied because the query doesn't reference the control table. To remedy this, we break up the test into three parts; the first two are evaluated at optimization time and the third one — the guard condition – is evaluated at execution time.

The first part is $P_q \Rightarrow P_v$, which tests whether the query is contained in the view if it is fully materialized. Clearly, the query cannot be contained in a partially materialized view if it is not contained in the corresponding fully materialized view.

For the second part, we add a *guard predicate* P_r to the antecedent, obtaining the condition $(P_r \wedge P_q) \Rightarrow (P_v \wedge P_c)$. This condition asks the question: *"If the additional condition P_r is satisfied, is the query then contained in the view?"* If the first condition, $P_q \Rightarrow P_v$, is satisfied, this second condition can be simplified to $(P_r \wedge P_q) \Rightarrow P_c$.

The third part of the test consists of verifying, at execution time, that a tuple satisfying the guard predicate exists in the control table. In other words, testing the condition $\exists t \in T_c : P_r(t)$.

Theorem 1. *Consider an SPJ query Q with a conjunctive predicate P_q and a partially materialized SPJ view V_p with base view predicate P_v and control predicate P_c referencing a control table T_c. Then query Q is covered by view V_p if there exists a predicate P_r such that the following three conditions are satisfied.*

$$P_q \Rightarrow P_v \tag{1}$$

$$(P_r \wedge P_q) \Rightarrow P_c \tag{2}$$

$$\exists t \in T_c : P_r(t) \tag{3}$$

Proof. We prove the theorem by contradiction. Assume that the three conditions are satisfied but, for some database instance, including an instance of control table T_c, there exists a row r such that $r \in Q$ but $r \notin V_p$. There are two cases to consider.

Case 1: Row r is not in V_p because $P_v(r)$ is false, that is, the row does not satisfy the base view predicate. But this contradicts the assumption that $P_q \Rightarrow P_v$ holds for all tuples.

Case 2: Row r is not in V_p because it has not been materialized. There are two reasons why this might happen. i) There is no tuple $t \in T_c$ such that $P_r(r, t)$ is true. But this contradicts the assumption that the condition $\exists t \in T_c : P_r(t)$ is satisfied. Or ii) r does not satisfy the control predicate, that is, $P_c(r)$ is false. But this contradicts the assumption that $(P_r \wedge P_q) \Rightarrow P_c$ holds for all tuples.

We have shown that all cases lead to contradictions and the proof is complete. $\square$

Example 2. For our example view V_1 and query Q_1 we have the following predicates.

P_v: (p_partkey=sp_partkey) $\wedge$ (sp_suppkey=s_suppkey)
P_c: (p_partkey=partkey)
P_q: (p_partkey=sp_partkey) $\wedge$
    (sp_suppkey=s_suppkey) $\wedge$ (p_partkey=@pkey)

Hence, the first test $P_q \Rightarrow P_v$ equals

(p_partkey = sp_partkey) $\wedge$ (sp_suppkey = s_suppkey)
$\wedge$ (p_partkey = @pkey)
  $\Rightarrow$
(p_partkey = sp_partkey) $\wedge$ (sp_suppkey = s_suppkey)

which clearly evaluates to true. Choosing the guard predicate as (partkey = @pkey), the second test $(P_r \wedge P_q) \Rightarrow P_c$ becomes

(partkey = @pkey) $\wedge$ (p_partkey = sp_partkey) $\wedge$
(sp_suppkey = s_suppkey) $\wedge$ (p_partkey = @pkey)
  $\Rightarrow$
(p_partkey = partkey)

After simplification to

(partkey = @pkey) $\wedge$ (p_partkey = @pkey)
  $\Rightarrow$
(p_partkey = partkey)

it is easy to see that this condition is also true. The last test, to be evaluated at execution time, equals

$\exists$ t $\in$ pklist: (t.partkey = @pkey)

Whether the partially materialized view is guaranteed to contain all required rows depends on whether P_r, with known parameters, evaluates to true at execution time. Using P_r and a ChoosePlan operator, we construct a dynamic execution plan as shown in Figure 1. If P_r evaluates to true at run time, we execute the branch of the plan that uses the view, otherwise the branch with the fallback plan is executed instead.

Theorem 2. *Consider an SPJ query Q with a non-conjunctive predicate P_q, which can be converted to disjunctive normal form as $P_q = P_q^1 \vee \cdots \vee P_q^n$ and a partially materialized SPJ view V_p with base view predicate P_v and control predicate P_c referencing a control table T_c. Then query Q is covered by view*

V_p if, for each disjunct $i = 1, 2, \cdots, n$, there exists a predicate P_r^i such that the following three conditions are satisfied.

$$P_q^i \Rightarrow P_v \tag{4}$$

$$(P_r^i \wedge P_q^i) \Rightarrow P_c \tag{5}$$

$$\exists t_i \in T_c : P_r^i(t_i) \tag{6}$$

Proof. For each disjunction P_q^i, the conditions are identical to those in Theorem 1. If the three conditions are satisfied, Theorem 1 guarantees that all rows of the query that qualify under P_q^i are covered by the view. If this holds for all disjuncts, every row that satisfies the query predicate $P_q = P_q^1 \vee \cdots \vee P_q^n$ is covered by the view. □

Example 3. The following query is similar to Q_1 but the equality predicate has been changed to an IN predicate. An IN predicate can be rewritten as a disjunction of equality predicates, which after conversion to disjunctive normal form, produces the two disjuncts shown below.

Q_2:
```
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and p_partkey in (12, 25)
```

P_q^1: `(p_partkey = sp_partkey)` $\wedge$
`(s_suppkey = sp_suppkey)` $\wedge$ `(p_partkey = 12)`
P_q^2: `(p_partkey = sp_partkey)` $\wedge$
`(s_suppkey = sp_suppkey)` $\wedge$ `(p_partkey = 15)`

The view matching tests for this example will be the same as in Example 2, except `@pkey` is replaced by 12 or by 15. The optimization-time tests still evaluate to true. For the query to be covered, both execution-time tests must be satisfied, which produces the following guard condition

```
∃ t1 ∈ pklist:  (t1.partkey = 12) ∧
∃ t2 ∈ pklist:  (t2.partkey = 15),
```

which can be expressed in SQL most efficiently as

```
2 = (select count(*) from pklist
       where partkey in (12,15))
```

3.2.2 Aggregation Views and Queries

We treat an aggregation query or view as an SPJ query followed by a group by operation. Aggregation adds one step to view matching that tests whether the grouping in the view is compatible with that in the query.

For a partially materialized view aggregation view, only the containment test of the view matching algorithm has to be modified as described in the previous section. The grouping-compatibility test is not affected because of our requirement that the control predicate P_c of a partially materialized view involves only *non-aggregated output columns* of the base view V_b. Hence, either all the rows in a group or none of them will satisfy the control predicate.

3.2.3 Types of Control Tables

This section describes different types of control predicates, their associated control tables, and how to choose the appropriate guard predicate P_r. We cover the most important types but do not cover all possible types.

Equality Control Tables: An equality control table is one where the control predicate specifies an equijoin between one or more columns in the base view and in the control table. This type of control table can only support queries with equality constraints on all join columns or queries that can be converted to this form.

Example 4. The control table `pklist` and the partially materialized view PV_1 in Section 1 are of this type. The control predicate is `(p_partkey = pklist.partkey)`. Query Q_1 contains a constraint equating `p_partkey` to a run-time constant, namely, `p_partkey = @pkey`. This run-time constant is then substituted for `p_partkey` in the control predicate to produce the guard predicate shown below.

P_r: `(pklist.partkey = @pkey)`

The guard condition, expressed in SQL, then becomes

```
exists(select * from pklist where partkey = @pkey)
```

Range Control Tables: A range control table is one that supports range control predicates. A partially materialized view with a range control table can support range queries or point queries.

Example 5. Consider the following parameterized range query that finds information about all suppliers for a given range of parts.

Q_3:
```
select p_partkey, p_name, p_retailprice, s_name,
  s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and p_partkey > @pkey1
and p_partkey < @pkey2
```

To support the query we create a partially materialized view with a range control table.

PV_2:
```
create table pkrange(lowerkey int, upperkey int)

create view PV2 as
select p_partkey, p_name, p_retailprice, s_name,
  s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and exists (select * from pkrange
     where p_partkey > lowerkey
     and p_partkey < upperkey)
```

Ensuring that `pkrange` contains only non-overlapping ranges can be done by adding a suitable check constraint or trigger to the table. The control predicate is

P_c: `(p_partkey > lowerkey)` $\wedge$ `(p_partkey < upperkey)`

A query must contain a range restriction on p_partkey for the view to be useful, which Q_3 does.

`(p_partkey > @pkey1)` $\wedge$ `(p_partkey < @pkey2)`

To guarantee that the view contains all required rows, the control table must contain a range that covers the query's range. Hence, the guard predicate becomes

P_r: `(lowerkey` $\leq$ `@pkey1)` $\wedge$ `(upperkey` $\geq$ `@pkey2)`

and the guard condition, expressed in SQL, becomes

```
exists(select * from pkrange
 where lowerkey <= @pkey1 and upperkey >= @pkey2)
```

Control tables specifying just an upper or a lower bound are feasible as well, and would support queries that specify a single bound, a range constraint, or an equality constraint. The control table would have only one row containing the current lower (or upper) bound.

Control Predicates on Expressions: The control predicate P_c is not limited to comparisons with "plain" columns from the base view. The comparison may instead be applied to the result of an expression or function over columns from the base view. Even a user-defined function can be used as long as it is deterministic.

Example 6. Suppose we have a user-defined function ZipCode that takes as input an address string and returns the zip code of the address. Consider the following query that finds information about all suppliers within a specified zip code.

Q_4:
```
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_address, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and ZipCode(s_address) = @zip
```

To support this query we define a control table `zipcodelist` and a partial view PV_3 as shown below.

PV_3:
```
create table zipcodelist(zipcode int primary key)

create view PV3 as
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_address, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and exists (select * from zipcodelist zcl
    where ZipCode(s_address) = zcl.zipcode)
```

The guard predicate is the same as for an equality control predicate referencing a "plain" column.

```
$P_r$: zipcodelist.zipcode = @zip
```

3.3 View Maintenance

Incremental maintenance of materialized views is a well-studied problem, and efficient maintenance algorithms are known for SPJG views. Compared with a fully materialized view, a partially materialized view can be maintained more efficiently, because only a small number of rows are actually materialized. However, current view maintenance algorithms are designed for SPJG views and do not support views containing *exist* subqueries. In this section, we outline how to incrementally maintain a partially materialized view. The general observation is that if the base view V_b is maintainable, the corresponding partial view V_p is also maintainable.

If the query expression in the *exists* clause returns at most one row for each possible value of the control columns, the subquery can be converted to a join. A partially materialized view V_p that satisfies this requirement can, for maintenance purposes, be treated as the regular view V'_p shown below.

```
create view $V'_p$ as
select $V_b$.*
from $V_b$, $T_c$
where $P_c$($T_c$.col1, $T_c$.col2, ..., $T_c$.coln)
```

The view V'_p is a regular SPJG view and can be incrementally maintained. For example, the view PV_1 is of this type because partkey is a primary key of the control table `pklist`.

If the query expression in the *exists* clause may return more than one row, converting the subquery into a join may produce duplicate rows. We consider two situations based on whether V_b contains aggregation or not.

First consider the case when V_b is a SPJ view. If the output columns of V_b contain a unique key [1], we can convert the view V_p into the following aggregation view V'_p to make it incrementally maintainable. View matching still treats the view as V_p.

```
create view $V'_p$ as
select $V_b$.*, count(*) as cnt
from $V_b$, $T_c$
where $P_c$($T_c$.col1, $T_c$.col2, ..., $T_c$.coln)
group by $V_b$.*
```

All the output columns of V_b have to be included as group-by columns so that they can be output. The group-by operation in V'_p simply removes the duplicated rows and the count is added for view maintenance. The view V'_p contains exactly the same rows as the view V_p; the only difference is that each row has an additional column `cnt`.

If the output columns of V_b do not contain a unique key, an extra join is required during maintenance so as not to introduce duplicates. We will show the rewrite assuming a single control column and denote this column by C_c. The generalization to multiple view columns is straightforward. In this case, we rewrite V_p using a self-join for maintenance purposes.

```
create view $V'_p$ as
select $V_b$.*
from $V_b$ v1 join
    (select $C_c$ from $V_b$, $T_c$
     where $P_c$($T_c$.col1, $T_c$.col2, ..., $T_c$.coln)
     group by $C_c$) v2
    on (v1.$C_c$ = v2.$C_c$)
```

The inner query removes duplicate rows. Even though V'_p is no longer a SPJG view, it can be maintained incrementally. During updates, the delta table of the inner query is computed first, including elimination of duplicates, and then used to update the outer view.

Now consider the case when V_b is an aggregation view. Let V_b^{spj} denote the SPJ part of the view and G denote the group-by columns of the view. If the output columns of V_b^{spj} contain a unique key, we can rewrite V_p as follows for maintenance purposes. The inner query removes duplicate rows before applying the aggregation in the outer query.

```
create view $V'_p$ as
select $V_b$.*
```

[1]In Microsoft SQL Server, a materialized view always has a unique key, so the output columns of V_b must form a unique key.

```
from (select $V_b^{spj}$.*
      from $V_b^{spj}$, $T_c$
      where $P_c$($T_c$.col1, $T_c$.col2, ..., $T_c$.coln)
      group by $V_b^{spj}$.*)
group by $G$
```

Similarly, if the output columns of V_b^{spj} do not contain a unique key, the inner query can by replaced by a self-join; the view can also be incrementally maintained.

3.4 Control Table Updates

Control table updates are treated no differently than normal base table updates. As detailed above, a partially materialized view can be properly maintained without distinguishing whether the update applies to a control table or a base table. One can design different strategies to decide which rows to materialize and when. The choice of materialization strategy, that is, which rows to materialize and when, depends entirely on the application. One example would be to use a caching policy like LRU or LRU-k that attempts to materialize the most frequently accessed rows. However, the problem of designing a materialization strategy for an application of partially materialized views is a separate issue and outside the scope of this paper.

4 More Complex Control Designs

In this section, we consider more complex uses of control predicates and control tables to design partially materialized views.

4.1 Multiple Control Tables

A partially materialized view can have multiple control tables, and the control predicates for each table can be combined in different ways. We will illustrate this scenario by creating a partially materialized view similar to PV_1 but using two control tables. In addition to the table `pklist` containing part keys, we have another control table `sklist` containing supplier keys.

We first create a partially materialized view PV_4 where the two control predicates (exists clauses) are ANDed together.

PV_4:
```
create table pklist (partkey int primary key)
create table sklist (suppkey int primary key)

create view $PV_4$ as
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and exists (select * from pklist pkl
    where p_partkey = pkl.partkey)
and exists (select * from sklist skl
    where p_partkey = skl.suppkey)
```

Q_1 cannot be answered from the view PV_4 because the view may not contain all the desired rows for a given part. For the view to be useful, the query must specify a set of part keys *and* a set of supplier keys, which must be found in `pklist` and `sklist`, respectively. Consider the following parameterized query that finds supplier information for a given part and a given supplier.

Q_5:
```
select p_partkey, p_name, p_retailprice, s_name,
   s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and p_partkey = @pkey
and s_suppkey = @skey
```

The guard predicate and the run-time guard condition equal

```
$P_r$: (partkey = @pkey) ∧ (suppkey = @skey)

    exists(select 1 from pklist
        where partkey = @pkey)
    and exists(select 1 from sklist
        where suppkey = @skey)
```

We can also create a partially materialized view PV_5 where the control predicates are ORed together.

PV_5:
```
create view PV5 as
select p_partkey, p_name, p_retailprice, s_name,
   s_suppkey, s_acctbal, sp_availqty, sp_supplycost
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and (exists (select * from pklist pkl
      where p_partkey = pkl.partkey)
      or exists (select * from sklist skl
      where p_partkey = skl.suppkey))
```

PV_5 can be treated as a union of two partially materialized views with control tables pklist and sklist, respectively. This immediately implies that queries that specify part keys and queries that specify supplier keys may be computable from the view. This holds also for queries that specify *both* part keys and supplier keys.

4.2 Views With A Common Control Table

Different partially materialized views may share a common control table. For instance, the table pklist controls the contents of the partially materialized view PV_1. The same table can be used to control other partially materialized views as well.

Example 7. Consider the following parameterized query that finds information about all lineitems for a given part.

Q_6:
```
select p_partkey, p_name, sum(l_quantity)
from part, lineitem
where p_partkey = l_partkey
and p_partkey = @pkey
group by p_partkey, p_name
```

Suppose that the access pattern is roughly the same as that for Q_1. To handle this query efficiently, we can create a partially materialized view PV_6, using the same control table pklist as Q_1.

PV_6:
```
create view PV6 as
select p_partkey, p_name, sum(l_quantity) qty
from part, lineitem
where p_partkey = l_partkey
and exists (select * from pklist
     where p_partkey = partkey)
group by p_partkey, p_name
```

The table pklist controls the contents of both the view PV_1 and the view PV_6. For both views, only rows with partkeys stored in the table pklist are materialized.

4.3 Using Another View As A Control Table

Another materialized view can be used as a control table. As we shall see in Section 5, this can be particularly useful for mid-tier caching.

Example 8. Suppose we wish to cache data about customers in the most frequently accessed market segments and also their orders. To do so, we would create a control table containing market segment ids and two views.

PV_7:
```
create table segments
   (segm varchar[25] primary key)

create view PV7 as
select c_custkey, c_name, c_address
from customer
where exits (select * from segments
   c_mktsegment = segm)
```

PV_8:
```
create view PV8 as
select o_custkey, o_orderkey, o_orderstatus,
   o_totalprice, o_orderdate
from orders
where exists (select * from PV7
   o_custkey = c_custkey)
```

The two views can of course be used independently, that is, PV_7 for queries against the customer table where the market segment is specified and PV_8 for queries against the orders table where the customer key is specified. In addition they can be used for queries joining customer and orders that specify a market segment, e.g. the following query.

Q_7:
```
select c_custkey, c_name, c_address,
   o_orderkey, o_orderstatus, o_totalprice
from customer, orders
where c_custkey = o_custkey
and c_mktsegment = 'Household'
```

4.4 View Groups

We say that two partially materialized views are (directly) related if they reference the same control table or one uses the other as a control table. A *partial view group* is a set of, directly or indirectly, related partially materialized views and control tables. We represent a partial view group as a directed graph, where nodes denote either control tables or partial views and edges denote control constraints (defined by control predicates). The direction of an edge for a control constraint is from a partial view to its control table(s).

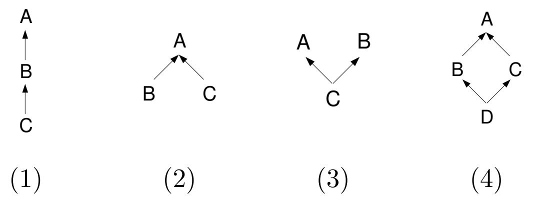


Figure 2: Partial View Graphs

Figure 2 shows some examples of partial view graphs. Figure 2(1) represents the partial view group of PV_8, PV_7 and the control table `segments` in Section 4.3. Figure 2(2) represents the case of PV_1, PV_6 and the control table `pklist` in Section 4.2. Figure 2(3) represents the case of PV_4 and its control tables `pklist` and `sklist`. Figure 2(4) shows a combination of different partially materialized views.

Within a partial view group, one or more control tables control the contents of a group of partially materialized views. Any update on the control tables has a cascading effect on all the views.

There are no cycles in a partial view group's connectivity graph. Views cannot reference themselves directly or indirectly because it could cause view expansion and view maintenance to fail. Partially materialized views also inherit this property.

5 Applications for Partially Materialized Views

Fully materialized views are defined statically and changing the contents of the view is cumbersome and slow. Partially materialized views are more flexible — changing the view contents is easy and fast. In practice, materializing only part of a view can be useful in many scenarios, five of which are briefly discussed in this section.

Mid-Tier Cache Containers

Partially materialized views can be extremely useful for a mid-tier database cache, such as Microsoft's MTCache [18, 10, 17] and IBM's DBCache [1]. A mid-tier cache replicates part of the data from a backend server and attempts to handle as many queries as possible from the replicated data. The goal is improved scale-out by offloading some of the query workload to such cache servers.

MTCache models local data as materialized views [18] that are updated asynchronously. These cached views are treated as regular materialized views and picked up by the optimizer transparently. Sometimes it would be preferable to materialize only some of the rows, for example, the most frequently accessed rows and be able to easily and quickly change which rows are materialized. Partially materialized views are ideal for this purpose.

Cache table is a new table type supported by DBCache to dynamically determine subsets of rows to be stored at a front-end server [1]. Cache tables and partially materialized views are similar in the sense that they can both be used as cache containers but there are many differences. Partially materialized views provide a general mechanism with many potential applications while cache tables are a special-purpose mechanism designed only for mid-tier caching. As cache containers, partially materialized views are much more flexible than cache tables. A cache table are limited to storing a subset of the rows from a backend table. Partially materialized views do not have this restriction; they can cache horizontal and vertical subsets, joins, and aggregations of tables or views on the backend server. Cache tables do not use explicit control tables and therefore do not offer the same flexibility in designing materialization strategies. A partially materialized view may have multiple control tables and can thus answer queries with different selection predicates.

Clustering Hot Items

Suppose we have a large table or materialized view with a very skewed access pattern. That is, a small fraction of rows account for the great majority of accesses. More often than not, these hot rows are scattered in what appears to be random order among the pages of the table or view. The buffer pool hit rate may be high but in reality significant memory space is wasted because each page contains only one or two hot rows. A partially materialized view can be used purely to cluster together the hot rows on fewer pages, thereby reducing memory requirements and improving buffer pool efficiency. The memory space freed up this way can then be used to bring into memory additional rows from the same or other tables, which will improve overall query performance.

Incremental View Materialization

A partially materialized view can be used to incrementally materialize an expensive view. This can be done using a range control table and slowly increasing the range covered. Having the control predicates range over the view's clustering key would materialize the view page by page and minimize overhead. Before the view gets fully materialized, we treat it as a partially materialized view and the contents of the control table represent the current materialization progress. The view can be exploited even before it is fully materialized! When materialization completes, all we need to do mark the view as being a fully materialized view and abandon the fallback plans.

Graefe [7] introduced partitioned B-trees and outlined several potential applications of this novel B-tree variant. As described in the paper, partitioned B-trees can be used to significantly speed up incremental index and view materialization.

Views with Non-Distributive Aggregates

Partially materialized views can also be used to expand the class of views supported by a DBMS to include view types that are not incrementally updatable. For instance, views that containing non-distributive aggregates like `min` and `max` that are not incrementally updatable, could be allowed. If the `min` or `max` for a particular group changes, the group could be removed from the view description and recomputed asynchronously later. In fact, it might be better to use the control table as an exception table, that is, an entry in the control table indicates that the corresponding group needs to be recomputed before it can be used.

View Support for Parameterized Queries

A parameterized query can typically be supported by a view with the same definition, except that the parameterized columns have to be added to the output columns of the view, allowing selection and possible re-aggregation to be applied. However, if the domain of a parameter is large, the resulting view may be very large because, in essence, we have materialized the view for all possible parameter values. If only a small number of the values are actually used, much effort is wasted to store and and maintain the view for parameter values never used in queries.

Example 9. This query computes the total value and number of orders by status for orders with a value range and a date.

Q_8:
```
select o_orderstatus, sum(o_totalprice), count(*)
from orders
where round(o_totalprice/1000,0) = @p1
and o_orderdate = @p2
group by o_orderstatus
```

A fully materialized view would provide very little benefit for this query. Following the standard approach, the view would be grouped on columns (round(o_totalprice / 1000,0), o_orderdate, o_orderstatus). The number of possible combinations of parameter values is so large that the materialized view would be as large as the order table.

Most likely, only a few combinations of actual parameter values would ever be used. To exploit this fact, we create an equality control table containing combinations of prices and dates of interest and a partially materialized view. The most commonly used combinations are added to the control table.

PV_9:
```
create table plist(price int, orderdate date)

create view PV9 as
select round(o_totalprice/1000,0) op, o_orderdate,
  o_orderstatus, sum(o_totalprice) sp, count(*) cnt
from orders
where exists (select * from plist pl
   round(o_totalprice/1000,0) = pl.price
   and o_orderdate = pl.orderdate)
group by round(o_totalprice/1000,0),
   o_orderdate, o_orderstaus
```

The query can be answered immediately by an index lookup of the view; no further aggregation is needed.

Partially materialized views can also be helpful for view support for queries with parameters in complex subqueries but the details are outside the scope of this paper.

6 Experimental Results

We have prototyped support for partially materialized views in Microsoft SQL Server 2005 Beta. We ran a series of experiments to compare the performance of a partially materialized view with that of a fully materialized view.

The main benefit of a partially materialized view is to avoid wasted maintenance efforts. However, before analyzing maintenance costs, we first verify (Section 6.2 and 6.1) that query performance when using a partially materialized view is no worse than when using a fully materialized view. In fact, it can even be better. In Section 6.3, we then compare update costs for two types of updates: large updates modifying all rows of a base table and small updates modifying a single row of a base table.

All experiments were performed on a workstation with a 3.2 GHz Pentium 4 processor, 1GB of memory and one 80GB disk, running Windows Server 2003. All queries were against a 10GB version (SF=10) of the TPC-R database.

6.1 Query Performance

With unlimited memory resources, the query performance of a partially materialized view improves if the view covers more queries because fewer queries will use the, presumably, more expensive fallback plan. The higher the hit rate, the closer its performance to that of a fully materialized view, assuming both views fit in memory. However, with limited memory resources, a larger fraction of a partially materialized view fits in memory, which reduces disk I/O. As a result, the overall query performance of a partially materialized view may be better than that of a fully materialized view, even if the view does not cover all the queries. The experiments reported in this section are designed to quantify the net effect and see how it is affected by skewness in the access pattern and buffer pool size.

The workload consisted of query Q_1 in Section 1 with varying parameter values. Three different database designs were considered: using no views, using fully materialized view V_1, and using partially materialized view PV_1. When using the fully materialized view V_1, the query execution plan is a simple index lookup of V_1. When using the partially materialized view PV_1, the query execution plan is a dynamic plan as shown in Figure 1. The fast branch consists of a simple index lookup against PV_1 while the fallback branch consists of an index lookup against the part table followed by two indexed nested loop joins with the partsupp table and the supplier table respectively.

We ran the query two million times with randomly selected partkey values drawn from a Zipfian distribution with a skew factor α. The control table `pklist` of view PV_1 contained the most frequent partkeys. The size of the fully materialized view V_1 is about 1GB. We fixed the size of the partially materialized view PV_1 to 5% of the size of V_1, that is, about 51 MB. By varying the skew factor α, we were able to change the view's hit rate, that is, the fraction of queries that can answered from PV_1.

We considered three different skew factors. The larger the skew factor α, the more skewed the access pattern and the higher the hit rate for PV_1. In this experiment, α was chosen so that PV_1 covered 90%, 95%, and 97.5%, respectively, of the query executions. The remaining query executions used the fallback plan. The guard condition was evaluated by an index lookup against the 1MB control table – the overhead was very small. For each scenario, we also explicitly varied buffer pool sizes.

Figure 3 shows the total execution time with different buffer pool sizes for the three scenarios. The buffer pool is too small to hold all three base tables (part, partsupp and supplier), which have a combined size of 1.5 GB. Because part keys are randomly distributed, we expect to have poor buffer pool usage and significant disk I/O. The smaller the buffer pool, the more severe the I/O problem. With the fully materialized view V_1, no joins are needed and CPU time is saved. However, V_1 is still too large to fit completely in the buffer pool, resulting in some I/O. With the partially materialized view, PV_1 is small enough to fit completely in the buffer pool.

As expected, it is uniformly faster to use a materialized view than computing the query from scratch, see Figure 3. All three plan types benefit from an increase in buffer pool size. Using the partially materialized view PV_1 can be up to 62% faster than using the fully materialized view V_1 because of better buffer pool utilization; performance is worse only when the buffer pool is very small. When the access pattern is more skewed, as shown in Figure 3(b) and 3(c), the partially materialized view can achieve about the same performance as the fully materialized view using only a quarter of the memory. When the access pattern is less skewed, as shown in Figure 3(a), it is slower to use PV_1 than to use V_1 when the memory size is extremely small. This is because the partially materialized view can only answer 90% of the queries. For the remaining 10%, it is sufficiently expensive to compute the results from scratch with the very limited memory available that it outweighs the savings on the other 90% of the queries.

In the experiments reported here, we arbitrarily limited the size of the partially materialized view to 5% of the fully materialized view. We have run additional experiments to determine the optimal size of the partially materialized view, how it varies with skewness and buffer pool size, and how sensitive query performance is to the size. The results indicate that, for our parameter settings, the optimal size is in the range 40-60% of the fully materialized view and that the performance curve is quite flat around the minimum. We also observed that even for the case of a 64 MB buffer pool and $\alpha = 1.0$, using the optimal partial materialized view is faster than the fully materialized view.

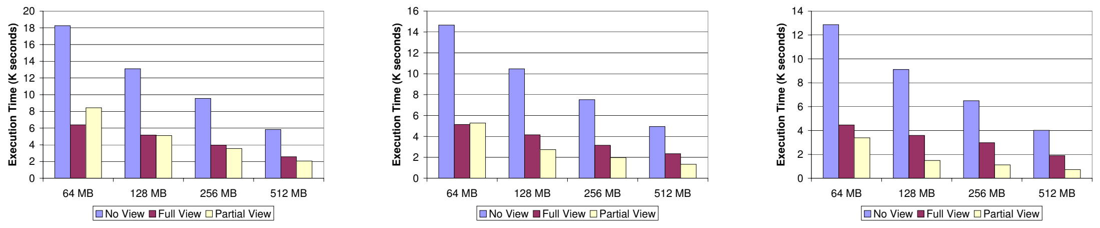
6.2 Processing Fewer Rows

In the previous experiment, both views were clustered on the control column p_partkey. Query Q_1 includes the very selective predicate (p_partkey = @pkey), so both plans included a small index scan using the view's clustering index. No matter which view is used, the number of rows scanned is the same, and so is the cost of computing the rest of the query. Therefore, the overall number of rows processed is the same for both views and the savings in elapsed time is due to improved buffer pool utilization.

What if the views are not clustered on the control column? In this case, fewer pages need to be fetched and fewer rows processed when using a partially materialized view instead of a fully materialized view. Simply put, there is less "junk" (non-qualifying rows) to wade through to find the target rows. Query performance should improve because less work needs to be done.

To investigate this effect, we created the following partially materialized view with an equality control predicate on s_nationkey and ran a query with selection predicates on p_type and s_nationkey.

PV_{10}:



(a) Skew factor $\alpha = 1$ (b) Skew factor $\alpha = 1.1$ (c) Skew factor $\alpha = 1.125$

Figure 3: Effect of Buffer Pool Size and Access Skewness

```
create table nklist(nationkey int primary key)

create view $PV_{10}$ as
select p_partkey, p_name, p_type, s_name,
  sp_supplycost, s_suppkey, s_name, s_nationkey
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and exists (select * from nklist nkl
     where s_nationkey = nkl.nationkey)

$Q_9$:
select p_partkey, p_name, p_type, s_name,
  sp_supplycost, s_suppkey, s_name, s_nationkey
from part, partsupp, supplier
where p_partkey = sp_partkey
and s_suppkey = sp_suppkey
and p_type like 'STANDARD POLISHED%'
and s_nationkey = @nkey
```

To speed up processing of the query, both PV_{10} and the corresponding fully materialized view were clustered on (p_type, s_nationkey, p_partkey, s_suppkey). We varied the size of the partially materialized view PV_{10} by varying the number of rows in the control table. PV_{10} always contained the nationkey for Argentina. We ran query Q_9 with @nkey = 1 (Argentina) 100 times and computed the average elapsed time.

The above table compares query Q_9 execution time with a cold buffer pool. For both view types, the main part of the execution consisted of an index scan using the view's clustering index. Because PV only contains rows from a subset of nations, fewer rows need to be read and processed compared with a fully materialized view. As expected, the savings is highest when the partially materialized view is small and increases linearly with the view size. The 3% increase when the partially materialized view contains all rows is caused by higher query startup cost and the the cost of evaluating the guard condition.

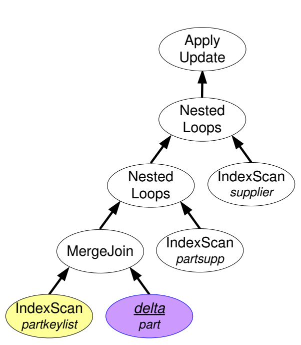
nklist Size	*Full View*	*Partial View*	*Savings(%)*
1	1.130	0.121	89%
5	1.130	0.294	74%
10	1.130	0.594	47%
25	1.130	1.170	-3%

Experiments with a warm buffer pool gave similar results but the savings were lower. With a warm buffer pool, no I/O is required to answer the query regardless of view type so the reduction in execution time is strictly due to reduced CPU time.

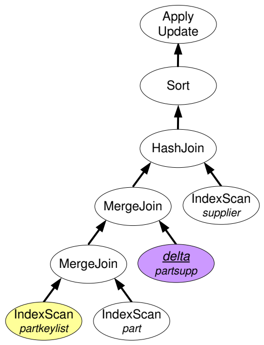
6.3 Update Performance

Partially materialized views are expected to have lower maintenance cost than the corresponding fully materialized view. To investigate this issue, we created two instances of the 10GB TPC-R database, one with the partially materialized view PV_1 and the other with the fully materialized view V_1. We chose the view configuration corresponding to Figure 3(b) (skew factor $\alpha = 1.1$, size of PV_1 5% of the size of V_1) and set the maximum buffer pool size of 512MB.

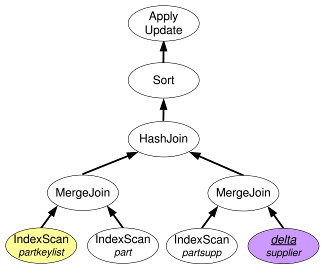
We ran experiments with two update scenarios: a large update that modified every row in a table and



(a) Update Part

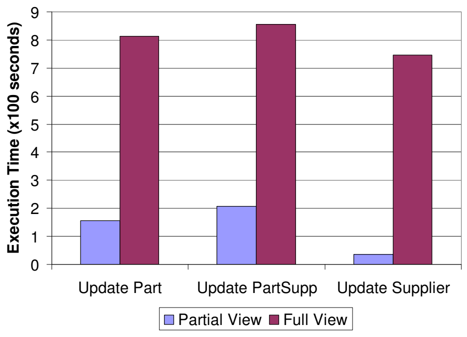


(b) Update PartSupp

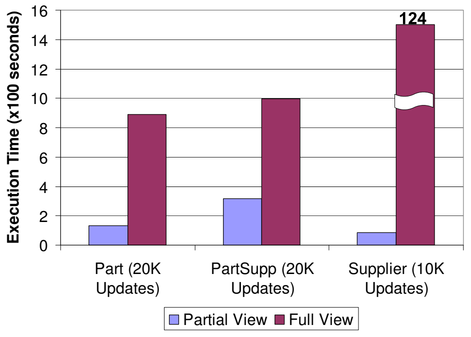


(c) Update Supplier

Figure 4: Update Plans



(a) Table Update



(b) Row Update

Figure 5: Maintenance Costs

small updates that modified a single base table row. We measured the total update time, including the time for the base table update and view maintenance and the time to flush all updated pages to disk.

In the large update case, a single update query was issued for each base table. The updates modified p_retailprice in the part table, ps_availqty in the partsupp table, and s_acctbal in the supplier table.

Figure 4 shows the corresponding update plans. Recall that the control table contains only 5% of the part keys (100,000 keys), so it is relatively small compared with the base tables. The join with the control table greatly reduces the number of rows causing it to be applied as early as possible in each of the plans. The more significant savings, however, results from having far fewer rows to apply to the view.

Figure 5(a) shows the total update cost for the large-update scenario. As expected, the observed cost is much lower – up to 43 times – when using a partially materialized view PV_1 than when using a fully materialized view. The gain when updating partsupp is much smaller than for the supplier table. We found that this is due to an optimization inefficiency for update plans. The optimizer produces an update plan that computes the full *delta* for the affected base table first and then runs update plans for any affected materialized view. When updating partsupp, which is the largest among the three tables, the delta itself is so large that much of it has to be flushed to disk. This adds significant overhead to the overall update cost. The same reasoning applies to updates of the part table. But update performance can be improved by immediately filtering the base table delta by semi-joining it with the partially materialized view or, when applicable, the smaller control table. This will be addressed in future work.

The second scenario we considered is a group of small updates, each one updating a single column of a single base table row based on a primary key selection. The updated columns are the same as the first scenario. The update plans generated were the same as shown in Figure 4.

We applied a large number of these small updates with randomly selected parameter values. The parameter values were uniformly distributed over their domains. The observed total update times are plotted in Figure 5(b). Again, maintaining the partially materialized view PV_1 is much cheaper and the reduction is as high as 124 times.

The reason for the smaller savings when updating partsupp table is that each update only affects one row in the full materialized view V_1. Even though we do much less maintenance work for the partially materialized view PV_1, the total execution cost is so low that the query initialization cost is a significant fraction of the overall cost. The initialization cost is the same whether we use a fully or partially materialized view. However, when updating the supplier table, each update affects 80 rows in V_1 and those 80 rows are unclustered, which means that close to 80 disk pages are affected by each update. In this case, the reduced maintenance work for PV_1 makes a huge difference.

Updating the control table pklist changes the content of the view PV_1. The update performance is similar to other base tables. The fourth column in Figure 5(b) shows the overall cost for updating the control table. These updates are cheap relative to V_1 updates because PV_1 is significantly smaller than V_1.

In summary, compared with maintaining a fully materialized view, the maintenance saving for a partially materialized view depends on the following factors.

- The cost of computing the delta rows for the view.
- How many rows in the view are affected by each update.
- Whether the affected rows are clustered or not.
- If an update affects very few rows, the benefit may not be that significant because of the constant startup cost.

7 Conclusion

In current database systems, a view must be either fully materialized or not materialized at all. We propose partially materialized views which materialize only some of the rows and can easily and quickly modify which rows to be stored. Partially materialized views are preferable in many database areas. We give a formal definition of a partially materialized view and extend regular view matching and maintenance algorithms. Experimental results in our database system show that partially materialized views have significant benefits of lower storage requirements, better buffer pool efficiency, better query performance, and lower maintenance costs.

References

[1] M. Altinel, C. Bornhovd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of VLDB Conference*, 2003.

[2] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *Proceedings of VLDB Conference*, 1998.

[3] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of ACM SIGMOD Conference*, 1986.

[4] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of ICDE Conference*, 1995.

[5] J. Goldstein and P. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of ACM SIGMOD Conference*, 2001.

[6] G. Graefe. Executing nested queries. In *Proceedings of BTW Conference*, pages 58–77, 2003.

[7] G. Graefe. Sorting and indexing with partitioned b-trees. In *Proceedings of CIDR Conference*, 2003.

[8] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proceedings of ACM SIGMOD Conference*, 1989.

[9] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedins of ACM SIGMOD Conference*, 1995.

[10] H. Guo, P. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency: how to say "good enough" in SQL. In *Proceedings of ACM SIGMOD Conference*, 2004.

[11] A. Gupta, I. S. Mumick, and K. A. Ross. Adapting materialized views after redefinitions. In *Proceedings of ACM SIGMOD Conference*, 1995.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD Conference*, 1993.

[13] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4), 2001.

[14] E. N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of ACM SIGMOD Conference*, 1987.

[15] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *Proceedings of ACM SIGMOD Conference*, 1996.

[16] IBM. *Red Brick Warehouse 6.3, Peformance Guide*, 2004.

[17] P. Larson, J. Goldstein, H. Guo, and J. Zhou. Mtcache: Mid-tier database caching for SQL server. *Data Engineering Bulletin*, 27(2), 2004.

[18] P. Larson, J. Goldstein, and J. Zhou. MTCache: Mid-tier database cache in SQL server. In *Proceedings of ICDE Conference*, 2004.

[19] P. Larson and H. Z. Yang. Computing queries from derived relations. In *Proceedings of VLDB Conference*, 1985.

[20] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of ACM SIGMOD Conference*, 1997.

[21] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *Proceedings of VLDB Conference*, 1996.

[22] S. R. Valluri. Partially materialized partitioned views. In *Proceedings of the 11th International Conference on Management of Data (COMAD)*, 2005.

[23] H. Z. Yang and P. Larson. Query transformation for psj-queries. In *Proceedings of VLDB Conference*, 1987.

[24] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *Proceedings of ACM SIGMOD Conference*, 2000.