Dynamic Materialized Views

Jingren Zhou
Microsoft Research
jrzhou@microsoft.com

Per-Åke Larson
Microsoft Research
palarson@microsoft.com

Jonathan Goldstein
Microsoft Research
jongold@microsoft.com

Luping Ding
Worcester Polytechnic Institute
lisading@cs.wpi.edu

Abstract

A conventional materialized view blindly materializes and maintains all rows of a view, even rows that are never accessed. We propose a more flexible materialization strategy aimed at reducing storage space and view maintenance costs. A dynamic materialized view selectively materializes only a subset of rows, for example, the most frequently accessed rows. One or more control tables are associated with the view and define which rows are currently materialized. The set of materialized rows can be changed dynamically, either manually or automatically by an internal cache manager using a feedback loop. Dynamic execution plans are generated to decide whether the view is applicable at run time. Experimental results in Microsoft SQL Server show that compared with conventional materialized views, dynamic materialized views greatly reduce storage requirements and maintenance costs while achieving better query performance with improved buffer pool efficiency.

1 Introduction

Judicious use of materialized views can speed up the processing of queries by several orders of magnitude. The idea of using materialized views is more than twenty years old [17, 22] and all major database systems (DB2, Oracle, SQL Server) now support materialized views [2, 23, 5]. The support included in those systems consists of computing, materializing, and maintaining *all* rows of the view definition result, which we refer to as *static* materialized views.

However, storage costs may be high for large static views and maintenance can also be costly if the views are frequently updated. If only a small subset of the full view result 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 this paper we introduce *dynamic materialized views* which selectively materialize only some of the rows in the view, for example, only the most frequently accessed rows. 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 by an example.

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

```
Q1:select p_partkey, p_name, p_retailprice, s_name,
    s_suppkey, s_acctbal, l_quantity, l_extendedprice
   from part, lineitem, supplier
   where p_partkey=l_partkey and s_suppkey=l_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.

```
create view V1 as
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_acctbal, l_quantity, l_extendedprice
from part, lineitem, supplier
where p_partkey=l_partkey and s_suppkey=l_suppkey
```

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

V_1 materializes the complete join, so it may be quite large. On a database at scale factor 1, there would be 200,000 parts and the view would contain 6 million 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 on any given day but 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. However, this is not possible with today's materialized view technology because the seasonally changing contents of the materialized view cannot be specified by a static predicate.

Dynamic materialized views are ideally suited for situations like this. To handle our example query, we create a control table *hotspot* and a dynamic materialized view DV_1 whose content is controlled by *hotspot*.

```
create table hotspot(hotpartkey int primary key)

create view DV1 as
select p_partkey, p_name, p_retailprice, s_name,
 s_suppkey, s_acctbal, l_quantity, l_extendedprice
from part, lineitem, supplier
```

```
where p_partkey=l_partkey and s_suppkey=l_suppkey
and exists(select * from hotspot hs
    where p_partkey=hs.hotpartkey)
```

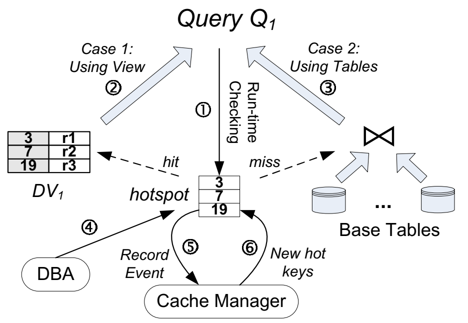


Figure 1. Overall Architecture

While the static view V_1 materializes information about all parts, DV_1 only materializes information about the parts listed in the control table $hotspot$, that is, parts satisfying the *exists* clause. The control table is *invisible* – queries do not need to explicitly reference the control table to exploit the view.

Figure 1 outlines the overall architecture of using dynamic materialized views in a database system. Query Q_1 can be answered from the view if the key of the desired part is found in $hotspot$. To exploit the view safely, the optimizer produces a query plan that first checks *at run-time* whether the desired part key exists in $hotspot$, shown as step ① in Figure 1. If it does, the plan evaluates the query using a simple select against DV_1 (step ②). Otherwise, the query is evaluated using the base tables (step ③).

Upon updates to the base tables, only changes affecting the hot parts need to be propagated to DV_1, which greatly reduces the view maintenance cost. The content of DV_1 can be changed dynamically by updating the control table $hotspot$. Inserting a new part key into the control table automatically adds its information to the view. The deletions happen in a similar way. We delay discussion of incrementally maintaining dynamic materialized views until Section 2.

Figure 1 also shows two possible ways to manage the content of DV_1. A DBA can manually change the contents of the control table according to new business requirements (step ④). Or, the control table can be automatically managed by an internal cache manager using a feedback loop. During execution of Q_1, the cache manager records whether the query could be answered from the view or not (step ⑤). The cache manager implements some caching policy and recommends admitting or evicting rows in the control table based on its policy (step ⑥). The recommended updates of the control table are done *asynchronously* so that normal query execution is not affected[1].

Dynamically materializing only part of a view can be useful in many scenarios. For example, dynamic views can be extremely useful for a mid-tier database cache [16, 1, 8], where the replicated data can be treated as dynamic materialized views and contain only the most frequently accessed rows. A dynamic view can also be used for incremental view materialization. Conventionally, materialized views cannot be exploited before the materialization finishes. The process can be very lengthy for an expensive view. Before the view gets fully materialized, we can treat it as a dynamic materialized view and the contents of the control table represent the current materialization progress. As a result, the view can be exploited even before it is fully materialized! We discuss other potential applications in Section 4.

The main contributions of our paper are as follows:

- We provide a new mechanism to dynamically adapt and exploit the contents of materialized views. Compared with traditional approaches, dynamic materialized views significantly reduce view maintenance costs and storage requirements.
- We extend conventional view matching and maintenance algorithms to dynamic materialized views. Dynamic query execution plans determine at run time whether a view can be used or not.
- We introduce a feedback loop with a novel caching policy to automatically adapt the contents of dynamic materialized views.
- We outline several potential applications of dynamic materialized views across different areas in database systems.

The rest of this paper is organized as follows. In Section 2, we introduce the general form of a dynamic materialized view, and present view matching and maintenance algorithms. We describe several types of control schemas and dynamic views with more complex control designs in Section 3. In Section 4, we explain how to adapt the contents of the dynamic view using a feedback loop and outline other applications. Experimental results in Microsoft SQL Server are presented in Section 5. We review related work in Section 6 and conclude in Section 7.

2 Dynamic Materialized Views

In this section, we define dynamic materialized views and describe how to extend regular view matching and maintenance algorithms to work with dynamic materialized views. For ease of presentation, we use a dynamic materialized view with a single control table as an example. The techniques presented here are also applicable to more advanced dynamic materialized views in Section 3.

2.1 View Definitions

Let V_b denote the query expression defining a standard SPJG (select, project, join and an optional group-by) view and P_v its select-join predicate. We refer to V_b as the base view. Borrowing from SQL, we use the shorthand $V_b.*$ to denote all columns of view V_b.

A dynamic materialized view V_d is defined over the base view V_b but has materialization controlled by a *control table* T_c and a *control predicate* $P_c(V_b, T_c)$.

[1]The update process is not shown in the figure.

```
create view $V_d$ as
select $V_b$.* from $V_b$
where exists (select 1 from $T_c$ where $P_c$($V_b$, $T_c$))
```

Control table T_c can be a regular table or even another materialized view. Control predicate P_c references columns from T_c and only *non-aggregated output columns* from V_b. This restriction is important for view matching and for view maintenance as described in Section 2.2 and Section 2.3.

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

The dynamic materialized view DV_1 defined earlier has the following components. We omit the full column list.

```
Vb: select ...
   from part, lineitem, supplier
   where p_partkey=l_partkey and s_suppkey=l_suppkey

Pv: (p_partkey=l_partkey) ∧ (l_suppkey=s_suppkey)
Tc: hotspot(hotpartkey int)
Pc(Vb,Tc): (p_partkey=hotpartkey)
```

2.2 View Matching

A view matching algorithm for regular materialized views is described in [5]. A view can be used to answer the whole query or some subexpressions. To determine whether a query expression can be computed from a view, the query and view expressions are first converted into normal form. Next containment is tested, that is, whether all rows required by the query are contained in the view. Finally, additional requirements such as whether the view supplies all required columns and has the correct duplication factor are checked. In this section, we show how to extend this algorithm to handle dynamic materialized views.

For regular views, containment of the query in the view can be tested at optimization time but for dynamic materialized views, part 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 values exist in the control table*. If the desired values are found in the control table, then all tuples associated with those values are currently materialized.

At optimization time, we construct the guard condition so that the query is guaranteed to be contained in the view if the guard condition evaluates to true. The 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.

Figure 2 shows a possible dynamic query plan for Q_1. The *ChoosePlan* operator first evaluates the guard condition shown on the right. (The operator tree for evaluating the guard condition is not shown.) If it evaluates to true, the dynamic 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.

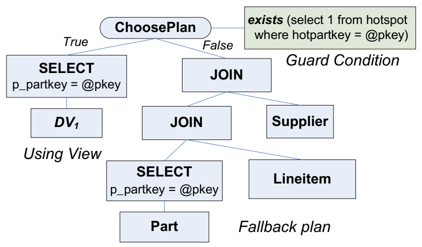


Figure 2. Dynamic execution plan for Q_1

More formally, let V_d be a dynamic materialized SPJ view with base view V_b and control predicate P_c. Denote the select-join 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. Due to space limitation, all the proofs are omitted and can be found in [24].

Containment checking requires that $P_q \Rightarrow (P_v \wedge P_c)$ in order to answer Q from V_d. The control table is invisible to the query so P_q does not reference the control table. The implication cannot be proven at compile time because it depends on the contents of the view at run time. To deal with 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 dynamic materialized view if it is not even contained in the corresponding fully materialized view.

For the second part, we add a *guard predicate* P_g, if possible, to the antecedent, obtaining the condition $(P_g \wedge P_q) \Rightarrow (P_v \wedge P_c)$. This condition asks the question: *"If the additional condition P_g 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 $(P_g \wedge P_q) \Rightarrow P_c$.

The third part of the test consists of verifying, at execution time, that a tuple satisfying the guard predicate exists in the control table, that is, $\exists t \in T_c : P_g(t)$.

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

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

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

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

Example 2 For our example view PV_1 and query Q_1, it is easy to see that the first test is true. Choosing the guard predicate as (hotpartkey=@pkey), the second test $(P_g \wedge P_q) \Rightarrow P_c$ is simplified to

```
(hotpartkey=@pkey)∧(p_partkey=@pkey)⇒
                                    (p_partkey=hotpartkey)
```

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

```
∃ t∈ hotspot:(t.hotpartkey=@pkey)
```

This condition, expressed in SQL, is shown in Figure 2.

Whether the dynamic materialized view is guaranteed to contain all required rows depends on whether P_g, with known parameters, evaluates to true at execution time. The following theorem considers queries with non-conjunctive predicates.

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 dynamic materialized SPJ view V_d 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_d if, for each disjunct $i = 1, 2, \cdots, n$, there exists a predicate P_g^i such that the following three conditions are satisfied.*

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

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

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

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'_1:select ...
   from part, lineitem, supplier
   where p_partkey=l_partkey and s_suppkey=l_suppkey
   and p_partkey in (12, 25)
```

```
P^1_q:(p_partkey=l_partkey) ∧
   (s_suppkey=l_suppkey) ∧ (p_partkey=12)
P^2_q:(p_partkey=l_partkey) ∧
   (s_suppkey=l_suppkey) ∧ (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∈ hotspot:(t1.hotpartkey=12) ∧
∃ t2∈ hotspot:(t2.hotpartkey=15)
```

which can be expressed in SQL most efficiently as

```
  2=(select count(*) from hotspot
     where hotpartkey in (12,15))
```

An aggregation query or view is treated 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 dynamic aggregation view the grouping-compatibility test is the same as for a regular view because of our requirement that the control predicate P_c of the 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.

2.3 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 dynamic 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 dynamic materialized view. The general observation is that if the base view V_b is maintainable, the corresponding dynamic view V_d 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 dynamic materialized view V_d that satisfies this requirement can, for maintenance purposes, be treated as the regular view V'_d shown below.

```
create view V'_d as
select V_b.* from V_b, T_c where P_c(V_b,T_c)
```

The view V'_d is a regular SPJG view and can be incrementally maintained. For example, the view DV_1 is of this type because hotpartkey is a primary key of the control table *hotspot*. Converting the subquery to an inner join produces the following equivalent definition

```
create view DV'_1 as
select ...
from part, lineitem, supplier, hotspot
where p_partkey=l_partkey and s_suppkey=l_suppkey
and p_partkey=hotpartkey
```

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.

Case 1: First consider the case when V_b is a SPJ view. If the output columns of V_b contain a unique key [2], we can convert the view V_d into the following aggregation view V'_d to make it incrementally maintainable.

```
create view V'_d as
select V_b.*, count(*) as cnt
from V_b, T_c where P_c(V_b, T_c) 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'_d simply removes the duplicated rows and the count is added for view maintenance. The view V'_d contains exactly the same rows as the view V_d; 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. We rewrite V_d using a self-join for maintenance purposes.

[2]In Microsoft SQL Server, a materialized view is required have a unique key.

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

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

Case 2: 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_d as follows for maintenance purposes. The inner query removes duplicate rows before applying the aggregation in the outer query.

```
create view $V'_d$ as
select $V_b$.*
from (select $V_b^{spj}$.* from $V_b^{spj}$, $T_c$
      where $P_c$($V_b$, $T_c$) 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.

In summary, these rewrites show that dynamic views can be efficiently maintained incrementally in the same way as regular views.

3 Control Schemes

So far we show dynamic views with equality control predicates. But many other types of control predicates and control tables are also possible. In this section, we cover a few important types, discuss what type of queries they can support and show how to construct the guard predicate P_g.

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. The control table *hotspot* and the dynamic materialized view DV_1 in Section 1 are of this type.

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

Example 4 Consider the following parameterized range query that finds information about all suppliers for a given range of parts, e.g. $(p\_partkey > @pkey1 \wedge p\_partkey < @pkey2)$. To support the query we create a dynamic materialized view with a range control table.

```
create table hotrange(lowerkey int, upperkey int)

create view $DV_2$ as
select ...  from part, lineitem, supplier
where p_partkey=l_partkey and s_suppkey=l_suppkey
and exists (select * from hotrange
  where p_partkey>lowerkey and p_partkey<upperkey)
```

For efficiency, one would ensure that *hotrange* contains only non-overlapping ranges. This can be done by adding a suitable check constraint or trigger to the table.

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

and the guard condition, expressed in SQL, becomes

```
exists(select * from hotrange
    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 5 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, e.g. $ZipCode(s\_address) = @zip$.

To support this query we define a control table *hotzipcode* and a dynamic view DV_3 as shown below.

```
create table hotzipcode(zipcode int primary key)

create view $DV_3$ as
select ...  from part, lineitem, supplier
where p_partkey=l_partkey and s_suppkey=l_suppkey
and exists (select * from hotzipcode zc
    where ZipCode(s_address)=zc.zipcode)
```

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

P_g: hotzipcode.zipcode=@zip

More Elaborate Control Designs: A dynamic materialized view can have multiple control tables, and the control predicates for each table can be combined in different ways. For example, a dynamic materialized view can store information only for hot parts defined in a control table *hotpart* and hot suppliers defined in another control table *hotsupplier*. A query that ask information for a given part and a given supplier can exploit the view with a run-time guard condition

```
exists(select 1 from hotpart where hotpartkey=@pkey)
and exists(select 1 from hotsupplier
        where hotsuppkey=@skey)
```

More interestingly, different dynamic materialized views may share a common control table. The same control table controls the contents of all the views. Moreover, a (dynamic) materialized view can be used as a control table to define another dynamic materialized view.

Example 6 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.

```
create table hotsegments(segm varchar[25] primary key)

create view DV_4 as
select ...  from customer
where exits(select * from hotsegments
   where c_mktsegment=segm)

create view DV'_4 as
select ...  from orders
where exists (select * from DV_4
   where o_custkey=c_custkey)
```

The two views can of course be used independently, that is, DV_4 for queries against the customer table where the market segment is specified and DV'_4 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.

```
select ...  from customer, orders
where c_custkey=o_custkey and c_mktsegment='Household'
```

Dynamic views with other powerful and flexible control schemes are possible; further details can be found in [24].

4 Control Table Management

Control table updates are treated no differently than normal base table updates. As detailed in Section 2.3, a dynamic materialized view can be properly maintained without distinguishing whether the update applies to a control table or a base table.

A materialization policy are simply rules for deciding which rows to materialize and when. The choice of materialization policy depends on the applications. A policy can be manually performed by a DBA or automatically deployed by the system. In this section we describe a few important applications and their corresponding policies.

Automatic Caching: A materialized policy can be implemented using some form of caching policy. For example, we implemented a feedback loop in Microsoft SQL Server to automatically manage the contents of the control table of a dynamic materialized view. An in-memory cache controller is associated with each control table. During execution of a query that uses a dynamic materialized view we record in its cache controller whether the required rows were covered by the control table, that is, whether the query could be answered by the view or not. A cache controller implements some caching policy and recommends admitting or evicting rows in the control table based on its policy. The recommended updates to the control table are done asynchronously so that normal query execution is not affected. A cache controller's data structures are stored in memory only and not persisted to disk. The cache controller is created the first time a dynamic materialized view is used after system startup.

An analysis of commonly used caching policies (LRU, LRU-k, Clock, GClock, 2Q) revealed that none of them satisfied all our requirements. All of them always admit a new key on its first access, ignoring the cost of the admission and possible an eviction as a consequence. In the case of dynamic views, admissions to the control table results in execution of a query and insertion of the result into the view. Evictions from the control table cause deletions from the view. The costs of admissions cannot be ignored. Therefore, we need a cache policy that has a low admission/eviction rate combined with a high hit rate. Because the cache controller is in memory, the cache policy needs to be very space efficient too.

We designed a novel cache policy based on the 2Q algorithm [14]. Due to space limitation, we can only highlight the important features of the algorithm.

- Similar to the 2Q algorithm, we make use of two queues, one for admissions and one for evictions.
- We consider admission of a new row on its second and subsequent accesses.
- We keep track of the approximate temperature (time since last access) of every row in both queues.
- If the cache is full, a row is admitted only of it is hotter than the coolest row already in the cache. This reduces the admission/eviction rate and keeps the hit rate high.
- We use bitmaps and hashing to significantly reduce memory requirements.

Experiments showed that our new policy is much more space efficient than competing policies and achieves a competitive hit rate coupled with a low admission/eviction rate and rapid response to changing access patterns.

Automatic caching can also be extremely useful for a mid-tier database cache, such as Microsoft's MTCache [16, 15] 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 to achieve improved scale-out. MTCache models local data as materialized views [16] that are updated asynchronously. Sometimes it would be preferable to automatically materialize only the most frequently accessed rows and change the set of rows in response to the access pattern of queries. We describe various models using dynamic views in [8].

Incremental View Materialization: A dynamic 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 dynamic 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 is to mark the view as being a fully materialized view and abandon the fallback plans.

Dynamic views can also be useful for clustering hot items, views with non-distributive aggregates, and view support for parameterized queries and queries with parameters in complex subqueries; the details can be found in [24].

5 Experimental Results

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

All experiments were performed on a workstation with dual 3.2 GHz Xeon processor, 2GB of memory and two 70GB SCSI disks, running Windows Server 2003. All queries were against a 10GB version (SF=10) of the TPC-H database. Automatic control table management as described in Section 4 was used throughout our experiments. We fixed the maximum size of the dynamic view and the cache controller determined which rows to materialize based on observed access patterns.

Dynamic materialized views are intended for applications with skewed access patterns. One main benefit of a dynamic view is reduced maintenance overhead, in case of frequent updates with either uniformly distributed or skewed update patterns. In this section, we focus in particular on the following areas.

- Due to partial materialization, a dynamic view cannot answer every possible query. With skewed access patterns and a fixed size, we want to verify that its query performance is no worse than that of a fully materialized view, even if occasionally some queries have to use the fallback plan.
- Secondly, we want to examine the query performance of a dynamic materialized view when memory space (for the buffer pool) is limited.
- Finally, we want to investigate the maintenance overhead for a dynamic view. Different update patterns need to be considered, either quite different or the same as the query access pattern.

5.1 Query Performance

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

```
Q5:select p_partkey, p_name, p_retailprice, s_suppkey
     s_name, s_acctbal, sum(l_quantity) as sum_qty
     sum(l_extendedprice*(1-l_discount)*(1+l_tax))
       as revenue
   from part, supplier, lineitem
   where p_partkey=l_partkey and s_suppkey=l_suppkey
   and p_partkey=@partkey
   group by p_partkey, p_name, p_retailprice,
       s_suppkey, s_name, s_acctbal

create view V5 as
select Q5.* from part, supplier, lineitem
where p_partkey=l_partkey and s_suppkey=l_suppkey
group by p_partkey, p_name, p_retailprice,
    s_suppkey, s_name, s_acctbal

create table hotspot(hotpartkey int primary key)

create view DV5 as
select Q5.* from part, supplier, lineitem
where p_partkey=l_partkey and s_suppkey=l_suppkey
and exists (select * from hotspot hs
    where p_partkey=hs.hotpartkey)
group by p_partkey, p_name, p_retailprice,
    s_suppkey, s_name, s_acctbal
```

The workload consisted of query Q_5 with varying parameter values. Three different database designs were considered: using a dynamic materialized view (DMV) DV_5, using a fully static materialized view (FMV) V_5, and using no views. When using the fully materialized view, the query execution plan is a simple index lookup of V_5. When using the dynamic materialized view DV_5, the query execution plan is a dynamic plan with two branches. The fast branch consists of a simple index lookup against DV_5. The fallback branch consists of an index lookup against the part table followed by two indexed nested loop joins with the supplier table and the lineitem table, respectively, and a final aggregation.

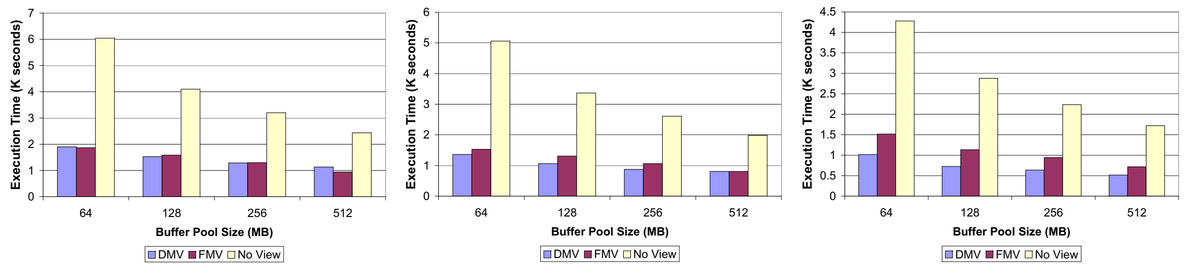
We ran the query one million times with randomly selected partkey values drawn from a Zipfian distribution with skew factor α. To avoid running a long series of queries to warm up the control table for every experiment, we preloaded the control table *hotspot* of view DV_5 with the most frequent accessed partkeys for each scenario [3].

The size of the fully materialized view V_5 is about 1GB. We fixed the size of the dynamic materialized view DV_5 to 5% of the size of V_5, 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 answered from DV_5[4].

We considered different skew factors. The larger the skew factor α, the more skewed the access pattern and the higher the hit rate for DV_5. We reported the results for three scenarios so that DV_5 covered 90%, 95%, and 97.5%, respectively, of the query executions. The guard condition was evaluated by an index lookup against the 1MB control table – the overhead was very small. The feedback loop is enabled so that new hot partkeys, if any, are automatically loaded or evicted from the view. The view's cache controller is called on every access and adds to the CPU load.

[3]We experimentally verified that preloading resulted in the correct initial state, that is, the same state as would have been reached by running a series of warm-up queries, and also that beginning with a cold cache controller did not materially affect the admission/eviction rate after warm-up.

[4]We also ran experiments for dynamic views with various sizes, up to 100% of the size of V_5. The results are similar; we do not report them here due to lack of space. As its size increased, the performance of the dynamic view converged to that of the full view.



(a) 90% ($\alpha = 1$) (b) 95% ($\alpha = 1.1$) (c) 97.5% ($\alpha = 1.115$)

Figure 3. Effect of Buffer Pool Size and Access Skew

In all the experiments, both the overhead of evaluating the guard condition and the cache controller overhead was too low to measure reliably.

For each scenario, we also varied the buffer pool size. 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 the hot part keys are randomly distributed over the key space, 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_5, no joins are needed and CPU time is saved. However, V_5 is still too large to fit completely in the buffer pool, resulting in some I/O. The dynamic view DV_5 is small enough to fit completely in the buffer pool.

As expected, it is much faster to use a materialized view than computing the query from scratch, as shown in Figure 3. All three plan types benefit from an increase in buffer pool size. Using the dynamic view DV_5 can be up to 33% faster than using the full view V_5 because of better buffer pool utilization; performance is slightly worse only when the buffer pool is very small. When the access pattern is highly skewed, as shown in Figure 3(c), the dynamic view can achieve about the same performance as the fully materialized view with only a quarter of the memory. When the access pattern is less skewed, as shown in Figure 3(a), the dynamic view achieves comparable performance as the fully materialized view. For the 10% of the queries that the dynamic view cannot answer, it is sufficiently expensive to compute the results from scratch with the limited memory available to cancel the savings resulting from better buffer pool utilization for the other 90% of the queries.

In summary, the performance benefits of using a dynamic view depends on several factors, such as the amount of memory available, the skew in the access pattern, and the relative cost of the fallback plan, etc. The benefit is higher if the access pattern is more skewed and the dynamic materialized view covers more queries, not taking into account buffer pool effects.

5.2 Processing Fewer Rows

In the previous experiment, both views were clustered on the control column p_partkey. Query Q_5 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 dynamic 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 dynamic view with an equality control predicate on s_nationkey and ran a query with selection predicates on p_type and s_nationkey.

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

create view DV6 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 hotnation ht
     where s_nationkey=ht.nationkey)

Q6: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
```

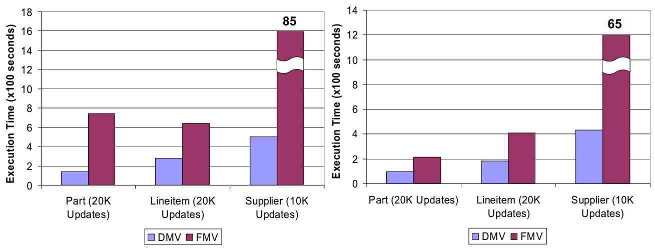
To speed up processing of the query, both DV_6 and the corresponding fully materialized view were clustered on (p_type, s_nationkey, p_partkey, s_suppkey). We varied the size of the dynamic view DV_6 by varying the number of rows in the control table. DV_6 always contained the nationkey for Argentina. We ran query Q_6 with @nkey=1 (Argentina) 100 times and computed the average elapsed time.

hotnation Size	*Full View*	*Dynamic 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%

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 DV_6 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 dynamic materialized view is small and increases linearly with the view size. The 3% increase when the dynamic materialized view contains all rows is caused by higher query startup cost and the the cost of evaluating the guard condition.

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.

5.3 Update Performance



(a) Uniform Update (b) Skewed Update

Figure 4. Maintenance Costs

Dynamic 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-H database, one with the dynamic view DV_5 and the other with the fully materialized view V_5. We chose the view configuration corresponding to Figure 3(b) (skew factor $\alpha = 1.1$, size of DV_5 5% of the size of V_5) and set the maximum buffer pool size of 512MB.[5]

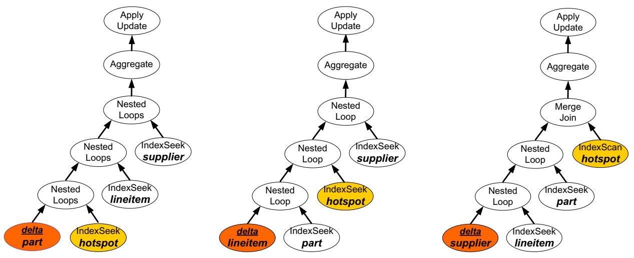
For each base table, we issued a series of updates, each one updating a single column of a single table row selected based on the primary key of the table. The updates modified p_retailprice in the part table, l_quantity in the lineitem table, and s_acctbal in the supplier table, respectively.

We ran experiments with two scenarios: uniform updates with random key value selection and skewed updates with skewed key value selection. We measured the total update time, including the time for the base table update and view maintenance and the time to flush updated pages to disk.

Figure 5 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 so it is applied as early as possible in each of the plans. The more significant savings, however, results from having far fewer updates to apply to the view.

In the uniform update case, the parameter values were uniformly distributed over their domains. Figure 4(a) shows

[5]The results for other configurations are similar, and are omitted due to space considerations.



(a) Update Part (b) Update Lineitem (c) Update Supplier

Figure 5. Update Plans

the total update costs for the uniform update scenario. As expected, the observed cost is much lower – up to 20 times – when using a dynamic view DV_5 than when using a fully materialized view V_5.

The gain when updating lineitem is much smaller than when updating supplier. The reason is that each update only affects one row in the full materialized view V_5. Even though we do much less maintenance work for the dynamic materialized view DV_5, 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 dynamic materialized view. However, when updating the supplier table, each update affects hundreds of rows in V_5 and those rows are not clustered together, which means that many disk pages are affected by each update. In this case, the reduced maintenance work for DV_5 makes a huge difference.

In the second scenario the rows to be updated were randomly selected using a skewed (Zipfian) distribution. The update columns are the same as the first scenario. The update plans generated were the same as shown in Figure 5.

When updating the part table, the updates had the same skewed distribution as queries. As DV_5 covered 95% of the query execution, DV_5 was also affected by 95% of the updates. This is the extreme case when the update skew is the exactly the same as the query skew. As shown in Figure 4(b), the benefits of using DV_5 is not as high as for uniformly distributed updates. Still, maintaining the dynamic view is cheaper by a factor of two. For 95% of the updates, the dynamic view and the fully materialized view were both affected by the update. However, the affected rows in the fully materialized view are scattered over the whole view. Maintaining them requires many more disk I/Os than maintaining the dynamic materialized view.

When updating the lineitem and supplier tables, the updates had a skewed distribution based on l_orderkey and s_suppkey with a skew factor $\alpha = 1.0$, respectively. However, the skew was not correlated to p_partkey. Again, maintaining the dynamic materialized view DV_5 is much cheaper and the improvement is up to a factor of 15.

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

- The cost of computing the delta rows for the view.
- How many rows in the view are affected by an update.

- Whether the affected rows are clustered or not.
- If an update affects very few rows, the benefit may not be significant because of the constant startup cost.

6 Related Work

The problems of view matching and view maintenance have received considerable attention in the research community for the last two decades. However, to the best of our knowledge, researchers have only considered fully materialized views.

Answering queries using views has been studied in [17, 22, 4, 20]. Larson and Yang [17, 22] were the first to describe view matching algorithms for SPJ queries and views. Srivastava et al. [20] 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 [12].

Incremental view maintenance has been studied in [3, 11, 7, 18]. They all use the *update delta* paradigm - compute a set of changed tuples (inserted or deleted) that are then used to update the materialized view. Gupta et al. [10] also described incremental view adaption techniques.

Materialized views have been adopted in all major commercial database systems. Oracle was the first commercial database system to support materialized views [2]. Zaharioudakis et al. [23] described a bottom-up view matching algorithm in IBM DB2. Goldstein and Larson [5] presented algorithms to matching views in Microsoft SQL Server.

Partial indexes that contain only some of the values in a column of a table were proposed in [21, 19]. The restriction is represented by a *static* predicate in the index definition. Similar to a regular materialized view, a partial index is defined over a single table by a select-project expression. Dynamic views are much more flexible.

Dynamic plans were proposed by Graefe and Ward in [6]. They have been used in the context of mid-tier caching in [1, 16, 9] and probably also in other contexts. At least one commercial system, Red Brick Warehouse [13], implements dynamic plans.

7 Conclusion

In current database systems, a view must be either fully materialized or not materialized at all. A dynamic view materializes only some of the rows and which rows are materialized can be easily and quickly modified by updating a control table. Adapting the contents can be done either manually by a DBA or automatically by an internal caching strategy that monitors which rows of the view are most frequently requested. We showed that dynamic views can be seamlessly integrated into the existing materialized view infrastructure.

Dynamic views offer significant performance benefits compared to regular materialized view for applications with skewed access patterns. The primary benefits are lower storage requirements, lower maintenance costs and lower buffer pool requirements. The new mechanism to dynamically adapt and exploit materialized views is ideal for many database applications.

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 *Proc. of VLDB Conference*, 2003.

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

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

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

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

[6] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of SIGMOD Conference*, 1989.

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

[8] H. Guo, P. Larson, and R. Ramakrishnan. Caching with 'good enough' currency, consistency, and completeness. In *Proc. of VLDB Conference*, 2005.

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

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

[11] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. of SIGMOD*, 1993.

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

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

[14] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. of VLDB Conference*, 1994.

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

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

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

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

[19] P. Seshadri and A. Swami. Generalized partial indexes. In *Proc. of ICDE Conference*, 1995.

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

[21] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4), 1989.

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

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

[24] J. Zhou, P.-Å. Larson, and J. Goldstein. Partially materialized views. Technical Report MSR-TR-2005-77, Microsoft Research, 2005.