IBM Research Report

Partial Materialized Views

Gang Luo
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
Partial Materialized Views

Gang Luo
IBM T.J. Watson Research Center
luog@us.ibm.com

Abstract
Early access to partial query results is highly desirable during exploration of massive data sets. However, it is challenging to provide transactionally consistent, immediate partial results without significantly increasing queries’ run-to-completion time. To address this problem, this paper proposes a partial materialized view method to cache some of the most frequently accessed results rather than all the possible results. Compared to traditional materialized views, the proposed partial materialized views do not require maintenance during insertion into base relations, and have much smaller storage and maintenance overhead. Upon the arrival of a query, the RDBMS first searches the partial materialized view and returns to the user the cached partial results. Since a large portion of the partial materialized view is cached in memory, this usually finishes within a millisecond. Then the RDBMS continues to execute the query to find the remaining results. The proposed techniques can also be extended to rank query result tuples according to their popularity, which addresses the information overflow problem. The efficiency of our partial materialized view method is evaluated through a simulation study, a theoretical analysis, and an initial implementation in PostgreSQL.

1. Introduction
Large data sets are common in practice, and the sizes of these data sets are becoming larger and larger. As a result, the capability of efficiently exploring massive data sets is urgently needed [HHW97]. It has been widely recognized that early access to partial query results can provide the following benefits and greatly facilitate the exploration of massive data sets:

Benefit 1: The RDBMS becomes more user-friendly.
Benefit 2: Early termination of those queries with unsatisfactory partial results (e.g., if users would like to refine them) can greatly reduce the load on the RDBMS and significantly speed up the exploration process.

In practice, it is important to provide transactionally consistent, immediate partial results without significantly increasing queries’ run-to-completion time, while statistical guarantee for the partial results is often not necessary.

For example, consider a retailer’s customer service call center. When a customer calls in, the call center operator can offer him on-sale items that are of his interest. The operator first obtains all the items \( I_p \) that the customer recently purchased and then performs a query \( Q \) on two relations. From the first relation, \( Q \) retrieves all the items \( I \), that are related to at least one of the items in \( I_p \). From the second relation \( R_{sales} \), \( Q \) finds all the items in \( I \), that are currently on sale with a discount of at least \( p \% \), where \( p \) is determined based on the loyalty of the customer. The operator only needs to see partial results of \( Q \) in order to start making offers to the customer and no statistical guarantee is needed for these partial results. Nevertheless, these partial results have to be obtained quickly (before the customer hangs up). In commercial databases, a common practice is to use a separate \( R_{sales} \) for each store or each department. Consequently, many query templates are needed to support this application.

Database researchers have spent much effort on investigating techniques for providing partial query results. However, none of the existing techniques is completely satisfactory. These techniques fall into three categories:

1. Use non-blocking query processing to generate output tuples continuously [CCD’03, HH99, HHW97, IFF’99, RH02].
2. Use special optimization techniques to find the first or top-\( k \) output tuples quickly [BCG02, CK97, DR99, IAE04].
3. Use asynchronously updated replicas to provide output tuples quickly [BAK’03, GLR’04].

Non-blocking query processing often increases a query’s run-to-completion time significantly. During exploration of massive data sets, knowing beforehand whether or not the user wants to see all the query results is often infeasible. Hence, it is difficult to decide in advance whether traditional (blocking) query processing should be used to optimize the query’s run-to-completion time, or non-blocking query processing should be used to generate output tuples continuously.

The optimization techniques for quickly finding the first or top-\( k \) output tuples are based on traditional (blocking) query processing and often require expensive I/Os. Hence, it can take much time (e.g., a few minutes) to generate the first or top-\( k \) output tuples. Moreover, in order to use these optimization techniques, the user needs to specify \( k \), which can be difficult to know beforehand. If the user is not satisfied with the first or top-\( k \) output tuples and would like to see all the results, he has to re-execute the query. This re-execution wastes system resources and is slower than requiring all the results at the first time.

In general, there is a delay before the data in the master copy is transferred to an asynchronously updated replica. Thus, the query results provided by the replica can be...
transactionally inconsistent with the data in the master
copy (e.g., a tuple is deleted from the master copy but still
exists in the replica). This is unacceptable to many
applications.

In this paper, we propose a partial materialized view
(PMV) method that can provide immediate partial results
without increasing queries’ run-to-completion time much.
These partial results are transactionally consistent and
suitable for those applications that do not require
statistical guarantee. Our idea is to reuse previous “hot”
results. More specifically, from previous queries’
execution, some of the most frequently accessed results
are remembered in the so-called PMVs. When a new
query comes, the corresponding PMV is first searched
and the found partial results are returned to the user. This
often finishes within a millisecond, because a large portion
of PMV is cached in memory. Then Q is executed to find
the remaining results.

Compared to traditional materialized views (MVs) that
store all possible results, our PMVs only store some of
the most frequently accessed results and have smaller sizes.
This saves most of the storage and maintenance overhead
of traditional MVs while many queries can still have early
access to partial results. Since PMVs are not used to
provide all the query results, no maintenance of PMV is
needed during insertion into base relations. To ensure that
a large portion of PMVs is cached in memory and thus the
return of partial results is quick, the size of each PMV has
an upper bound. To increase our chance of using a PMV
to provide partial results with only a limited storage, we
continuously update the content in the PMV to adapt to
the current query pattern, and restrict the maximum number
of tuples that can be stored in the PMV for any single,
so-called basic condition part. Whenever possible, both
the maintenance and the update of PMVs are coupled
with query execution for free. We investigate the
performance of the PMV method with a simulation study,
a theoretical analysis, and an initial implementation in
PostgreSQL. Our results show that PMVs have minor
overhead and can often provide partial results almost
instantly. Also, the RDBMS can afford storing many
PMVs.

Our proposed techniques can be extended to solve
the frequently encountered information overflow problem
[CCH04], where users get overwhelmed by the large
number of result tuples returned from SQL queries. Our
method is to rank result tuples according to their
popularity. More specifically, some data structure DS is
used to record the frequencies of “basic” query selection
conditions. These frequencies approximate the popularity
of result tuples. When a new query comes, the information
in DS is used to rank result tuples. An advantage of this
ranking method is that as query pattern changes, the
information in DS gets continuously updated. Hence, the
ranking result always reflects the current status.

The rest of the paper is organized as follows. Section 2
discusses the limitations of traditional MV method.
Section 3 presents the details of the proposed PMV
method. Section 4 shows how to extend the proposed
techniques to deal with the information overflow problem.
Section 5 investigates the performance of the PMV
method. Finally, we discuss related work in Section 6 and
conclude in Section 7.

2. Limitations of Materialized Views

A traditional method of speeding up query execution is
to use MVs [GM99]. In this section, we first describe the
queries that will be considered by the PMV method and
then discuss the limitations of traditional MV method.

2.1 Query Specification

In this work, we consider the following type of queries
that are frequently encountered in practice (e.g., in form-
based applications) – queries coming from templates of
the following form:

\[ q: \quad \text{select } L_a \text{ from } R_1, R_2, \ldots, R_n \text{ where } C_{\text{join}} \text{ and } C_{\text{select}} \]

Here, \( L_a \) is the select list, \( C_{\text{join}} \) includes both the join
condition among the \( n \geq 1 \) relations \( R_1, R_2, \ldots \) and \( R_n \), and
the selection conditions on a single relation that have no
parameters (e.g., \( R_{i,k} \cdot b=100 \)). \( C_{\text{select}} = \bigwedge_{i=1}^{m} C_i \), where \( m \) is a
number. Each \( C_i (1 \leq i \leq m) \) is a selection condition on a
single relation \( R_i \). \( 1 \leq h \leq n \). \( C_i \) takes one of the
following two disjunctive forms, which accept one or
more parameters:

**Equality form:** \[ v_{r=1}^{u} \left( R_{i,k} \cdot a_h = v_{i,r} \right) \]

**Interval form:** \[ v_{r=1}^{u} (v_{i,r} < R_{i,k} \cdot a_h < w_{i,r}) \]. The intervals
\( (v_{i,r}, w_{i,r}) \), \( (v_{i,r}, w_{i,r}) \), \ldots, and \( (v_{i,r}, w_{i,r}) \) are disjoint from
each other.

Different queries from the same template can have
different \( u_i \)’s \( (1 \leq i \leq m) \). In Section 3.6, we will show how
to extend our techniques to handle other forms of queries
(e.g., aggregate queries, nested queries).

In the interval form case, \( R_{i,k} \cdot a_h \) is not restricted to
being a numerical attribute. For example, \( R_{i,k} \cdot a_h \) can be a
string attribute. Also, \( v_{i,r} (1 \leq r \leq u) \) can be \( -\infty \) while \( w_{i,r} \)
can be \( +\infty \), and “<” can be replaced by “\( \leq \)”. In other
words, the intervals can be either bounded or unbounded,
open or closed. For ease of presentation, in the remainder
of this paper, we always write an interval as an open
bounded one, with the understanding that it can be closed
and/or unbounded if necessary.

\[ \text{select } R.a, S.e \text{ from } R, S \]

where \( R.c=S.d \) and \( (R.f=f_1 \) or \( R.f=f_2 \) or \ldots or \( R.f=f_h ) \)
and \( (S.g=g_1 \) or \( S.g=g_2 \) or \ldots or \( S.g=g_l ) \).

Figure 1. An example query template \( E_q \).
For a large subset of queries of the \( q_i \) form, traditional query processing cannot produce output tuples quickly and continuously. In this work, we focus specifically on such type of queries. These queries include both queries whose query plans are not fully pipelined and some queries whose query plans are fully pipelined. To illustrate the latter case, let us consider the template \( E_{\emptyset} \) in Figure 1. Suppose that an index exists on each selection/join attribute. The query plan fetches tuples from \( R \) using the index on \( R.f \). For each retrieved tuple \( t_r \), the index on \( S.d \) is used to search \( S \) for matching tuples. If the selectivity of \( S.g \) is low, the index on \( S.d \) needs to be searched many times before the first result query tuple is obtained. This can take a few seconds in a lightly loaded RDBMS, and a few minutes in a heavily loaded RDBMS.

### 2.2 Limitations of Large Materialized Views

create materialized view \( V_{\emptyset} \) as

\[
\text{select } R.a, S.e, R.f, S.g \text{ from } R, S \text{ where } R.c = S.d;
\]

**Figure 2. An example large materialized view.**

Existing techniques for automatically selecting MVs from query traces are based on “merging,” where the definition of each suggested MV is based on the common part of some of the queries [ACN00, DDD'04, ZRL'04]. For example, for the template \( E_{\emptyset} \) in Figure 1, existing automatic MV selection tools may suggest a materialized view \( V_M \) as shown in Figure 2. (The search procedure in \( V_M \) needs attributes \( R.f \) and \( S.g \).) As \( V_M \) needs to keep all the possible results for queries from \( E_{\emptyset} \), \( V_M \) is fairly large. In general, due to the extreme storage and maintenance overhead of MVs [GM99], the RDBMS cannot keep a MV for each frequently used query template.

### 2.3 Limitations of Small Materialized Views

For the template \( E_{\emptyset} \) in Figure 1, instead of using the big \( V_M \) in Figure 2 for all possible \((f_i, g)\) pairs, one might wonder whether we could create multiple small MVs, one for each “hot” \((f_i, g)\) pair, and use them to speed up query processing. These small MVs have the following advantages. First, a hot \((f_i, g)\) pair appears frequently in queries from \( E_{\emptyset} \). Therefore, the RDBMS can use a small MV that is built for a hot \((f_i, g)\) pair to partially answer a lot of queries from \( E_{\emptyset} \). Second, the combined size of these small MVs is a small percentage of that of the big \( V_M \). Thus, the combined storage and maintenance overhead of these small MVs is smaller than that of \( V_M \). Also, compared to \( V_M \), these small MVs can be accessed more quickly, as they are more likely to be cached in memory.

However, existing techniques for answering queries using MVs [CKP'95, GL01, Hal01, PL00] focus on shortening queries’ run-to-completion time. In a large number of cases, they cannot use these small MVs to shorten the run-to-completion time of queries from the template \( E_{\emptyset} \). This is because typically, a query from \( E_{\emptyset} \) contains both several hot \((f_i, g)\) pairs and several cold \((f_j, g)\) pairs. During the process of obtaining the results corresponding to the cold \((f_j, g)\) pairs, the results corresponding to the hot \((f_i, g)\) pairs can be computed inexpensively without using these small MVs.

For example, suppose that \((R.f=1, S.g=2)\) is the only hot \((f_i, g)\) pair. We create a small materialized view \( V_{SM} \) for \((R.f=1, S.g=2)\) as follows:

create materialized view \( V_{SM} \) as select \( R.a, S.e \) from \( R, S \) where \( R.c = S.d \) and \( R.f = 1 \) and \( S.g = 2 \);

Consider the following query that comes from the template \( E_{\emptyset} \) in Figure 1:

\[
\text{select } R.a, S.e \text{ from } R, S \\
\text{where } R.c = S.d \text{ and } (R.f = 1 \text{ or } R.f = 3) \text{ and } (S.g = 2 \text{ or } S.g = 4);
\]

In order to obtain the results corresponding to the cold pair \((R.f=1, S.g=4)\), tuple(s) \( t_8 \) of \( R \) where \( R.f=1 \) are fetched. Similarly, to obtain the results corresponding to \((R.f=3, S.g=2)\), tuple(s) \( t_3 \) of \( S \) where \( S.g=2 \) are retrieved. After fetching \( t_8 \) and \( t_3 \), computing the (possibly in-memory) join between them is not expensive and thus there is no need to use \( V_{SM} \).

### 3. The Partial Materialized View Method

In this section, we present our PMV method for providing partial query results, which can overcome the limitations of traditional MV method. We first describe the main ideas. Then we go into the details of the method.

All discussions in Section 2.3 about small MVs are from the viewpoint of minimizing queries’ run-to-completion time. The main goal of our PMV method is to minimize the time of generating partial results. In this case, these small MVs for the hot \((f_i, g)\) pairs become useful, as they can quickly provide partial results to a large number of queries from the template \( E_{\emptyset} \).

For example, consider a query \( Q \) from the template \( E_{\emptyset} \) in Figure 1. \( Q \) contains both several hot \((f_i, g)\) pairs and several cold \((f_j, g)\) pairs. The RDBMS answers \( Q \) in the following way:

**Step 1:** These small MVs are used to quickly obtain the partial results corresponding to the hot \((f_i, g)\) pairs. These partial results are returned to the user and recorded in a temporary in-memory data structure \( D_3 \).

**Step 2:** \( Q \) is executed to obtain all the results. For each result tuple \( t \), we check whether \( t \in D_3 \). If so (i.e., the user has already obtained \( t \) at Step 1), \( t \) is removed from \( D_3 \) and not returned to the user. Otherwise if \( t \notin D_3 \), the RDBMS knows that \( t \) corresponds to some cold \((f_j, g)\) pair and returns \( t \) to the user. In this way, each result tuple is returned to the user once and only once. (Query results can contain duplicate tuples. In the case that \( t \notin D_3 \), if \( t \) is not removed from \( D_3 \) and later another tuple \( t'=t \) comes, the RDBMS can end up returning fewer result tuples to the user than it should.)

The above method will slightly increase query \( Q \)’s run-to-completion time, as neither Step 1 nor the checking at Step 2 is needed in traditional query processing. However, this extra overhead is minor compared to the two benefits (user-friendliness, load reduction) of providing partial results that are mentioned in the introduction. Thus, for those applications of exploring massive data sets, it is
worth to make this tradeoff. For the purpose of easy management, all the small MVs are combined into a single so-called PMV. This becomes our PMV method. More details of our method are described in the following subsections.

3.1 Definitions

We first introduce some definitions.

Partial materialized view. Consider a MV definition \( V_M \). \( V_M \) may or may not exist in the RDBMS. Any subset of \( V_M \) is a partial materialized view \( V_{PM} \). \( V_{PM} \) is the containing materialized view of \( V_M \). The base relations of \( V_M \) are also called the base relations of \( V_{PM} \). (Both MVs and PMVs are treated as multi-sets and thus can contain duplicate tuples.) For the materialized view \( V_M \) in Figure 2, we show an example partial materialized view \( V_{PM} \) in Figure 3.

![Figure 3. An example partial materialized view.](image)

**Condition part.** Consider the query template \( q_i \) in Section 2.1. A condition part is an m-tuple \((d_1, d_2, \ldots, d_m)\), where for each \( i \) \((1 \leq i \leq m)\):

1. If the selection condition \( C_i \) is of equality form, \( d_i \) is of the form \( R_{k_i}.a_{k_i} = b_i \).
2. If \( C_i \) is of interval form, \( d_i \) is of the form \( b_i < R_{k_i}.a_{k_i} < c_i \).

A query result tuple \( t \) belongs to a condition part \((d_1, d_2, \ldots, d_m)\) if \( t \) satisfies all conditions \( d_i \) \((1 \leq i \leq m)\). A condition part \((d_1', d_2', \ldots, d_m')\) is contained in another condition part \((d_1, d_2, \ldots, d_m)\) if whenever conditions \( d_i \) \((1 \leq i \leq m)\) are true, conditions \( d_i' \) \((1 \leq i \leq m)\) are also true.

For each selection condition \( C_i \) \((1 \leq i \leq m)\) that is of interval form, let \( E_i \) denote the entire range of all possible intervals in \( C_i \) (e.g., \( E_i = (-\infty, +\infty) \)). We assume that the RDBMS knows multiple “dividing” values that can divide \( E_i \) into multiple non-overlapping “basic” intervals and these basic intervals fully cover \( E_i \). Each basic interval is assigned a different id. The purpose of this division is discretization so that the problem becomes more tractable. The criterion for choosing dividing values is that the resulting basic intervals can be used to differentiate hot results from cold results.

In a large number of form-based applications, for each selection condition \( C_i \) \((1 \leq i \leq m)\) that is of interval form, the user is provided with both a list of from values and a list of to values. Each (from value, to value) pair chosen by the user forms an interval \((v_r, w_r)\), where \( 1 \leq r \leq u \). In this case, these from values and to values can serve as dividing values. In other cases, we assume that either the person (e.g., DBA) who defines the PMV for the query template will specify the dividing values, or the continuous feature discretization technique [DKS95] in machine learning can be used to automatically learn dividing values from query traces.

**Basic condition part.** A condition part \((d_1, d_2, \ldots, d_m)\) is a basic condition part, if for each selection condition \( C_i \) \((1 \leq i \leq m)\) that is of interval form, \( d_i \) is of the form \( b_i < R_{k_i}.a_{k_i} < c_i \), where \((b_i, c_i)\) is a basic interval.

A basic condition part \((d_1, d_2, \ldots, d_m)\) is stored in the following way:

1. If \( d_i \) is of the form \( R_{k_i}.a_{k_i} = b_i \), value \( b_i \) is stored.
2. If \( d_i \) is of the form \( b_i < R_{k_i}.a_{k_i} < c_i \), where \((b_i, c_i)\) is a basic interval, the id of \((b_i, c_i)\) is stored.

3.2 Organization of Partial Materialized Views

Consider a frequently used query template \( q_i \) (see Section 2.1). Suppose that the RDBMS cannot afford to keep a materialized view \( V_{PM} = \{ \text{select } L_i' \text{ from } R_1, R_2, \ldots, R_n \text{ where } C_{\text{join}} \} \). Here, \( L_i' \) is the expanded select list that includes all the attributes in both \( C_{\text{select}} \) and the original select list \( L_i \). The search procedure in \( V_{PM} \) needs the attributes in \( C_{\text{select}} \).

We build a partial materialized view \( V_{PM} \) for \( q_i \) as follows:

- create partial materialized view \( V_{PM} \) as subset of \( \{ \text{select } L_i' \text{ from } R_1, R_2, \ldots, R_n \text{ where } C_{\text{join}} \} \) with selection condition template \( C_{\text{select}} \).
- \( V_{PM} \) is the containing MV of \( V_{PM} \). All the tuples in \( V_{PM} \) satisfy the condition \( C_{\text{join}} \).

The person who defines \( V_{PM} \) specifies an upper bound \( U_B \) for the size of \( V_{PM} \). This \( U_B \) is used to constrain the storage and maintenance overhead of \( V_{PM} \) and ensure that a significant portion of \( V_{PM} \) is cached in memory so that \( V_{PM} \) can be accessed quickly. Initially, \( V_{PM} \) is empty. Our goal is to use \( V_{PM} \) to provide immediate partial results to as many queries from the template \( q_i \) as possible.

In the template \( q_i \), the original select list \( L_i \) is replaced with the expanded select list \( L_i' \). This is to let all the attributes in \( C_{\text{select}} \) appear in query result tuples. As will be shown later, some result tuples are stored in \( V_{PM} \). The attributes in \( C_{\text{select}} \) are needed to find partial results in \( V_{PM} \). When the RDBMS obtains a query result tuple, it only returns the attributes in \( L_i \) to the user. Hence, the user still receives the same answer, as if \( V_{PM} \) did not exist and \( L_i \) in \( q_i \) had not been replaced by \( L_i' \).

![Figure 4. Data structure of a partial materialized view \( V_{PM} \).](image)
attribute index. For example, for the template \( E_y \) in Figure 1, Figure 4 shows the corresponding PMV.

Our goal is to use \( V_{PM} \) to provide immediate partial results to as many queries as possible. Hence, it is preferable to have a large number of basic condition parts stored in \( V_{PM} \). In general, many query result tuples can belong to a single basic condition part, and it is not desirable to flood \( V_{PM} \) with all these tuples. Therefore, the person who defines \( V_{PM} \) specifies a constant \( F \). For a basic condition part \( bcp \), the RDBMS stores at most \( F \) result tuples (rather than all the possible result tuples) that belong to \( bcp \) in \( V_{PM} \). This is different from the case of traditional MVs, where a materialized view \( V_M \) stores all the result tuples that satisfy the definition of \( V_M \). Given the storage limit \( U_B \) of \( V_{PM} \), for a query \( Q \), this \( F \) makes a tradeoff between (a) the probability that \( V_{PM} \) can provide some partial results to \( Q \), and (b) in the case that \( V_{PM} \) contains some partial results of \( Q \), the number of partial result tuples that \( V_{PM} \) can provide to \( Q \).

Let \( L \) denote the number of basic condition parts in \( V_{PM} \). \( A \) denotes the average size of the tuples in \( V_{PM} \). We have \( U_B \leq L \times F \times A \). If \( L=10K \), \( F=2 \), and \( A=50B \), then the size of \( V_{PM} \) is no more than 1MB and thus the memory can hold many PMVs. As will be shown in Section 5.1, \( L=10K \) can lead to a hit probability of 95%.

The design principles of our algorithm are as follows. The storage budget \( U_B \) is limited. Hence, \( V_{PM} \) should store hot basic condition parts. (A hot basic condition part appears in a large number of queries.) This is to maximize the chance that \( V_{PM} \) can provide partial results to a query.

The query pattern can change from time to time. That is, the basic condition parts that are hot can keep changing. We want to automatically keep track of this change and update \( V_{PM} \) accordingly. Hence, all the basic condition parts in \( V_{PM} \) are managed by the CLOCK algorithm [SGG02]: when \( V_{PM} \) is full, the RDBMS replaces the basic condition parts in \( V_{PM} \) that are no longer hot with the currently hot basic condition parts.

\( V_{PM} \) is initially empty. Before \( V_{PM} \) becomes full, content is filled into \( V_{PM} \). When \( V_{PM} \) becomes full, the content in \( V_{PM} \) is updated as query pattern changes. Both the fill in process and the update process of \( V_{PM} \) should be as efficient as possible. Therefore, in the case that there is no change to the base relations of \( V_{PM} \), the RDBMS only fills content into \( V_{PM} \) (if \( V_{PM} \) is not full) or changes the content of \( V_{PM} \) (if \( V_{PM} \) is full) for free when it obtains result tuples from query execution. There is no separate process for examining the base relations of \( V_{PM} \).

Similarly, in the case that the basic relations of \( V_{PM} \) get changed, the maintenance of \( V_{PM} \) should be as efficient as possible. Hence, whenever possible, the RDBMS couples the maintenance of \( V_{PM} \) with the execution of subsequent queries for free. Lastly, the use of \( V_{PM} \) needs to have minor influence on queries’ run-to-completion time.

### 3.3 Handling Queries

When a query \( Q \) comes, the RDBMS performs the following operations:

**Operation \( O_1 \):** The \( C_{select} \) of \( Q \) is broken into one or more non-overlapping condition parts. Each condition part is either a basic condition part itself or contained in a basic condition part.

**Operation \( O_2 \):** For each generated condition part, the RDBMS checks whether there is a corresponding entry in \( V_{PM} \). If so, the related tuples in \( V_{PM} \) are returned to the user as partial results. In this way, the RDBMS finds all the result tuples of \( Q \) that are in \( V_{PM} \).

**Operation \( O_3 \):** \( Q \) is executed to obtain all the result tuples. For those tuples that the user does not receive in Operation \( O_2 \), the RDBMS returns them to the user now. Also, the content in \( V_{PM} \) is updated to reflect the observed change in the hot basic condition parts.

**Operation \( O_4 \):** \( C_{select} \Rightarrow \) Condition Parts

\[ C_{select} = \bigwedge_{i=1}^{n} C_i \]

For each \( i \ (1 \leq i \leq m) \), there are two possible cases:

1. \( C_i \) is of equality form \( \forall_{r \in i}(R_{a_i} = v_{r,i}) \). Let set
   \[ S_i = \{R_{a_i} = v_{r,i} \mid 1 \leq r \leq u_i \} \]

2. \( C_i \) is of interval form \( \forall_{r \in i}(v_{r,i} < R_{a_i} < w_{r,i}) \). For each \( r \) \((1 \leq r \leq u_i)\), the RDBMS finds all the basic intervals \( I_{i,r} \) that overlap with the interval \((v_{r,i},w_{r,i})\). Let set
   \[ S_i = \bigcup_{r \in i} \{R_{a_i} \in (v_{r,i},w_{r,i}) \cap I_{i,r} \} \]

**C_{select}** is broken into a number \( (h \geq 1) \) of “non-overlapping” condition parts \( \prod_{i=1}^{h} S_i \). For each condition part \( cp_j \ (1 \leq j \leq h) \), there are two possible cases:

1. \( cp_j \) is a basic condition part \( bcp \), itself.
2. \( cp_j \) is contained in a basic condition part \( bcp_x \).

In either case, \( bcp_j \) is called the containing basic condition part of \( cp_j \).

Suppose that in the template \( E_y \) in Figure 1, the selection condition on \( S.g \) is of interval form rather than of equality form. Figure 5 shows an example of breaking the \( C_{select} \) of a query from \( E_y \) into condition parts. The outer rectangle represents the entire query space, which is partitioned into non-overlapping basic condition parts as shown by the dashed lines. The gray rectangle represents the query. The \( C_{select} \) of this query is broken into nine condition parts. Each condition part is represented by the intersection of the gray rectangle and a dashed rectangle that is filled with either upward or downward diagonals.

![Figure 5. An example of breaking the C_{select} of a query from E_y into condition parts.](image-url)
A temporary in-memory data structure $D_s$ is kept. For each condition part $bcp_j$ ($1 \leq j \leq h$) generated in Operation $O_1$, a counter $c_j$ is kept for its containing basic condition part $bcp_j$. Initially, $D_s$ is empty and $c_j = 0$ ($1 \leq j \leq h$). For each $bcp_j$ ($1 \leq j \leq h$), the index $I$ on $bcp$ is used to check whether $bcp_j$’s containing basic condition part $bcp_j$ exists in $V_{PM}$. There are two possible cases:

1. $bcp_j$ exists in $V_{PM}$, $c_j$ is set to be the number of tuples in $V_{PM}$ that belong to $bcp_j$. For each tuple $t$ in $V_{PM}$ that belongs to $bcp_j$, the RDBMS checks whether $t$ belongs to $bcp_j$. This is equivalent to checking whether $t$ satisfies the $C_{select}$ of query $Q$. If $bcp_j$ is a basic condition part itself, $t$ must belong to $c_j$. In contrast, if $c_j$ is contained in a basic condition part, $t$ may or may not belong to $bcp$. All the tuples in $V_{PM}$ satisfy the condition $C_{join}$. Hence, if $t$ satisfies $C_{select}$, $t$ is returned to the user as a partial result, and recorded in $D_s$.

2. $bcp_j$ does not exist in $V_{PM}$. Nothing is done in this case.

**Operation $O_2$: Returning Remaining Result Tuples and Updating Partial Materialized View**

Query $Q$ is executed to obtain all the result tuples. For each result tuple $t$, the data structure $D_s$ is checked to see whether the user has already obtained $t$ in Operation $O_2$. If $t \in D_s$, $t$ is removed from $D_s$. If $t \notin D_s$, the RDBMS performs the following operations:

1. Return $t$ to the user.
2. Find the containing basic condition part $bcp_j$ ($1 \leq j \leq h$) that $t$ belongs to. For each basic condition part $bcp$, at most $F$ query result tuples that belong to $bcp$ can be stored in $V_{PM}$. If the counter $c_j < F$, $t$ is added into $V_{PM}$ and $c_j$ is incremented by 1. This can require purging some basic condition part (and the associated query result tuples) from $V_{PM}$ if $V_{PM}$ has already been full. This case of $c_j < F$ is possible, e.g., as $V_{PM}$ is not maintained immediately during insertion into the base relations of $V_{PM}$ (see Section 3.4). In the case that $c_j = 0$, a new basic condition part $bcp_j$ is added into $V_{PM}$.

After all the result tuples have been processed, the data structure $D_s$ must be empty. $D_s$ is freed.

**3.4 Maintaining Partial Materialized Views**

When the base relations of $V_{PM}$ get changed, $V_{PM}$ is maintained in a different way from traditional MVs. This is because $V_{PM}$ is only a subset of its containing materialized view $V_M$. $V_{PM}$ is not used to provide all the query results. As long as $V_{PM}$ does not provide incorrect partial results, there is no need to change $V_{PM}$ immediately. Rather, the maintenance of $V_{PM}$ is deferred to when the RDBMS obtains result tuples from the execution of future queries for free. This minimizes the influence of $V_{PM}$ on transactions that change the base relations of $V_{PM}$.

Upon a change $\Delta R_i$ to a base relation $R_i$ ($1 \leq i \leq n$) of $V_{PM}$, there are three possible cases:

1. The change is an insert. This insert may generate new query result tuples. However, existing tuples in $V_{PM}$ are not affected by this insert. Hence, $V_{PM}$ is not maintained immediately.
2. The change is a delete. The join between $\Delta R_i$, and the other base relations $R_j$ ($1 \leq j \leq n, j \neq i$) of $V_{PM}$ is computed. For each join result tuple $t$, the index $I$ on $bcp$ is used to check whether $t$ exists in $V_{PM}$’s containing MV $V_M$. However, since $V_{PM} \subseteq V_M$, $t$ may or may not exist in $V_{PM}$. If $t \in V_{PM}$, $t$ is removed from $V_{PM}$.
3. The change is an update. Recall that all the attributes in $C_{select}$ appear in the expanded select list $L_i'$. If this update does not change the attributes of $R_i$ that appear in either $L_i'$ or the condition $C_{join}$, it will not affect the existing tuples in $V_{PM}$. Hence, there is no need to maintain $V_{PM}$. (Deletion influences all the attributes of $R_i$ and thus does not have this optimization.) Otherwise we proceed in a way similar to that in the case of deletion.

**3.5 Refinements**

In order to improve performance, we present several refinements to our approach.

**Using Better Cache Management Method**

Consider a basic condition part $bcp$ that exists in the partial materialized view $V_{PM}$. Tuples in $V_{PM}$ often have either a large number of attributes or some long attributes (e.g., detailed description). As a result, the combined size of all the tuples in $V_{PM}$ that belong to $bcp$ is usually much larger than the size of $bcp$. If we treat $bcp$ as the page id, and all the tuples in $V_{PM}$ that belong to $bcp$ as the page, then $V_{PM}$ looks much like a buffer pool. Hence, instead of using the CLOCK algorithm, the RDBMS can use other better buffer pool management algorithms (e.g., 2Q [JS94]) to manage $V_{PM}$. This will increase the probability that $V_{PM}$ can provide partial results to queries from the template $q_r$. The experimental section 5.1 gives a performance comparison between CLOCK and 2Q.

**Speeding Up Partial Materialized View Maintenance**

To speed up the maintenance of the partial materialized view $V_{PM}$ when some base relation of $V_{PM}$ gets changed, we can build indices on some attributes of $V_{PM}$. For example, suppose that tuple $t$ is deleted from base relation $R_i$ ($1 \leq i \leq n$) of $V_{PM}$. Assume that the index $I$ on $bcp$ is the only index on $V_{PM}$. Then in general, as mentioned in Section 3.4, in order to see whether any tuple in $V_{PM}$ is affected by this delete, the RDBMS needs to first compute the join between $t$ and the other base relations $R_j$ ($1 \leq j \leq n, j \neq i$) of $V_{PM}$. This join computation can be costly.

Now suppose that attribute $R_a$ exists in $V_{PM}$ and an index $I_a$ is built on $R_a$. $I_a$ is first searched to see whether there are tuples $t'$ in $V_{PM}$ such that $t''.a = t.a$. If no such tuple exists, there is no need to maintain $V_{PM}$. Otherwise the RDBMS deletes all the tuples $t'$ in $V_{PM}$ such that $t''.a = t.a$. 

In either case, the expensive join between \( t \) and the other base relations \( R_i \) \((1 \leq j \leq n, j \neq i)\) is waived. In the latter case, more tuples can be deleted from \( V_{PM} \) than necessary. However, this is acceptable, as \( V_{PM} \) only needs to maintain the property that it is a subset of its containing materialized view \( V_m \). Also, deleting tuples from the (possibly in-memory) \( V_{PM} \) is often cheaper than computing the join between \( t \) and \( R_i \)’s \((1 \leq j \leq n, j \neq i)\). The RDBMS can get back (some of) the unnecessarily deleted tuples from the execution of subsequent queries for free.

**Ignoring Queries Whose \( C_{select} \) is Complex**

In Operation \( O_1 \), the \( C_{select} \) of a query is broken into a number \((h \geq 1)\) of condition parts. It is not desirable to use the PMV method to handle queries whose \( C_{select} \) can be broken into too many condition parts, as it can be costly to check all these condition parts. Hence, we have a threshold \( h_t \). The PMV method is not used to handle those queries whose \( h > h_t \). As will be shown in Section 5.2 below, \( h_t \) can be quite large.

### 3.6 Discussions and Summary of Advantages

Like traditional MVs, the standard locking protocol is used on PMVs to ensure serializability. When a query \( Q \) reads a partial materialized view \( V_{PM} \) in Operation \( O_2 \), \( Q \) puts an S lock on \( V_{PM} \). Then between Operations \( O_2 \) and \( O_3 \), no other transaction can change the correct \( V_{PM} \) read result of \( Q \) by updating some base relation, as that would require updating \( V_{PM} \) with the acquisition of an X lock on \( V_{PM} \). Hence, \( Q \) would not have read anomaly.

With minor changes in our algorithm, PMVs can be used to handle queries with distinct clauses. In Operation \( O_2 \), only distinct tuples in the partial results obtained from the PMV are returned to the user and stored in the data structure \( D_S \). In Operation \( O_3 \), all distinct result tuples are first obtained from query execution. Then only those tuples that are not in \( D_S \) are returned to the user.

The above discussion focuses on non-aggregate queries, which are common these days. For example, both the call center scenario in the introduction and deep analytical tasks in real-time data warehouses require detailed data. With minor changes in the user interface, PMVs can also be used to handle aggregate queries (e.g., group by) or queries with order by clauses. In Operation \( O_2 \), the partial results obtained from the PMV are first aggregated or sorted and then presented to the user as intermediate results, with the user’s understanding that (a) these intermediate results are used to get a feeling of the final results and (b) the final aggregate values or order sequence can be different. In Operation \( O_1 \), after all the results are obtained, the intermediate results obtained in \( O_2 \) are invalidated and the final results are presented to the user.

In certain cases, with some extension, PMVs can be used to handle nested queries. For example, consider a two-level nested query. The subquery appears in the where clause of the main query after an EXISTS operator. Suppose that we can quickly obtain tuples from the main query but checking the EXISTS condition is time-consuming. In this case, a PMV can be used to quickly generate partial results of the subquery. Then for some tuples from the main query, the process of checking the EXISTS condition can be sped up. Consequently, we can rapidly produce some partial results for the entire query.

The partial materialized view \( V_{PM} \) has the following advantages:

1. \( V_{PM} \) has small storage and maintenance overhead.
2. \( V_{PM} \) can provide immediate partial results to a large number of queries from the template \( q_t \).
3. A large portion of provided partial results are hot results – they are frequently accessed by other queries from \( q_t \). This is desirable for those applications where users care more about hot results than cold results. (For applications that users want to see random partial results, this can be a disadvantage. However, as shown in [CMN99], in general it is difficult to provide random partial results.)
4. \( V_{PM} \) has minor influence on queries’ run-to-completion time.

The proposed techniques are not limited to providing early access to partial results. In the next section, we demonstrate the generality of our techniques by applying them to the problem of ranking query result tuples according to popularity.

### 4. Ranking Query Result Tuples

During exploration of massive data sets, users often get overwhelmed by the large number of result tuples returned from SQL queries, also known as the information overload problem [CCH04]. In this case, unless an order by clause is specified in the SQL query, it is desirable to rank result tuples according to their popularity (i.e., the frequencies that users query them). For example, both AOL’s Shopping Search & Browse tool [AOL03] and the Direct Hit search engine [Fag02] rank search results based on popularity. As a second example, [Joa02, Zwi03] show that by considering popularity in the search result ranking algorithm, the performance of search engines is improved. (An RDBMS can be regarded as a search engine in the sense that both RDBMS and search engine do search.) In fact, due to lack of system support, a large number of web sites implement their own methods of ranking SQL query result tuples according to popularity [AOL03].

#### 4.1 Overview of Our Approach

We propose a new method for ranking query result tuples according to their popularity. The main idea of our method is as follows. SQL queries do associative search (search by value). For all tuples with the same selection attribute values, a SQL query selects either all of them or none of them. That is, all result tuples with the same selection attribute values have the same popularity. Therefore, popularity could be tracked based on selection attribute values. To reduce the space overhead, popularity is tracked continuously based on basic condition parts. To minimize the burden of ranking result tuples on the RDBMS, the data structure that is used for tracking
popularity is kept in memory. Hence, the exact popularity cannot be tracked for all the possible basic condition parts. Rather, approximate popularity is tracked.

In the remainder of Section 4, we focus on queries coming from the same template \( q_t \) in Section 2.1. Irrespective of query execution time, as long as a query returns a large number of result tuples, it is desirable to rank these tuples.

### 4.2 Ranking Method

Suppose that we want to rank result tuples for queries from the template \( q_t \). As will be shown later, in the ranking process, the attributes in \( C_{select} \) are needed to decide which result tuple belongs to which basic condition part. Therefore, as in Section 3, in \( q_t \), the original select list \( L_o \) is replaced with the expanded select list \( L_o' \). After all the result tuples have been ranked, their attributes in \( L_o \) are returned to the user. In this way, the user still receives the same answer (but ranked by popularity), as if \( L_o \) in \( q_t \) had not been replaced by \( L_o' \).

The RDBMS builds an in-memory data structure \( DS \) that is a table. The number of rows in \( DS \) has an upper bound \( U_b \). This \( U_b \) is specified by the person who requires ranking result tuples for queries from the template \( q_t \). The criterion for choosing \( U_b \) is to ensure that \( DS \) can be kept in memory all the time (or at least most of the time). Each row of \( DS \) is of the form (basic condition part \( bcp \), count), where count represents the popularity of \( bcp \). We build an index on \( bcp \). Initially, \( DS \) is empty.

The same techniques in Section 3 are used to divide the entire query space into basic condition parts. The data structure \( DS \) is used to continuously keep track of the (approximate) popularity of basic condition parts. If each basic condition part is treated as a value, this is the hot list query problem that is studied in [GM98]. [GM98] gives two solutions to this problem: the concise sample method and the counting sample method. The first solution has lower overhead while the second solution is more accurate. Either solution can be used for our purpose.

When a new query \( Q \) comes, the same techniques in Section 3.3 are used to break the \( C_{select} \) of \( Q \) into one or more condition parts and obtain the corresponding containing basic condition parts. The concise/counting sample method in [GM98] is used to update the data structure \( DS \) accordingly. Then \( Q \) is executed to obtain all the result tuples. For each such result tuple \( r \), the RDBMS finds the containing basic condition part \( bcp \) that \( r \) belongs to. If \( bcp \) exists in \( DS \), the count of \( bcp \) in \( DS \) is used to approximate the popularity of \( r \). Otherwise the popularity of \( r \) is approximated as zero. Finally, all the result tuples of \( Q \) are ranked according to their (approximate) popularity. The core of our ranking method is the concise/counting sample method. The interested reader can find the performance study of both sample methods in [GM98].

### 5. Performance Evaluation of Partial Materialized View

The performance of our PMV method has been evaluated from three perspectives:

1. The probability that a PMV can provide partial results to a query.
2. The influence of the PMV method on queries’ run-to-completion time.
3. The maintenance overhead of a PMV when its base relations get changed.

#### 5.1 Probability of Being Useful

We first perform a simulation study to show that in a large number of cases, PMVs can provide partial results to a query. Consider a read-only database. We focus on those queries that come from the same template \( q_t \). Assume that a partial materialized view \( V_{PM} \) is built for \( q_t \). In Operation \( O_i \), the \( C_{select} \) of each query is broken into the same number \( h \geq 1 \) of condition parts, where each condition part is a basic condition part itself. The entire query space contains \( 1M \) basic condition parts \( bcp_i \) (\( 1 \leq i \leq 1M \)). For each basic condition part, the number of query result tuples that belong to it is greater than \( F \). As a result, for each basic condition part that exists in \( V_{PM} \), \( F \) query result tuples are stored in \( V_{PM} \). For each basic condition part in the \( C_{select} \) of a query, the probability that it is \( bcp_i \) (\( 1 \leq i \leq 1M \)) is \( e_i \). All the \( e_i \)'s (\( 1 \leq i \leq 1M \)) follow a Zipfian distribution with parameter \( \alpha \). That is, \( e_i \propto 1/i^\alpha \).

We compare the following two methods of managing all the basic condition parts in \( V_{PM} \):

1. The \( CLOCK \) algorithm. \( V_{PM} \) is a queue with \( L \) entries that is managed by the \( CLOCK \) algorithm. Each entry can store one basic condition part \( bcp \) and \( F \) query result tuples that belong to \( bcp \).
2. A simplified version of the \( 2Q \) algorithm [JS94]. \( V_{PM} \) is composed of two queues: \( Am \) and \( AI \). \( Am \) has \( N \) entries and is managed by the \( CLOCK \) algorithm. Each entry can store one basic condition part \( bcp \) and \( F \) query result tuples that belong to \( bcp \). \( AI \) has \( N' = 50\% \times N \) entries and is a FIFO queue. Each entry stores one basic condition part. Upon the first time that a basic condition part \( bcp \) appears in the \( C_{select} \) of a query, \( bcp \) is put into \( AI \). If during its stay in \( AI \), \( bcp \) appears again in the \( C_{select} \) of another query, both \( bcp \) and \( F \) query result tuples that belong to \( bcp \) are moved to \( Am \). \( Am \) is used to provide partial results to a query.

We assume that the storage requirement of a basic condition part is \( 4\% \) of that of \( F \) query result tuples. Thus, given the same storage budget \( U_b \) of \( V_{PM} \) for both the \( CLOCK \) and the \( 2Q \) algorithms, we have \( L = 1.02 \times N \).

The purpose of the comparison between the \( CLOCK \) algorithm and the \( 2Q \) algorithm is to show that in a large number of cases, the simple \( CLOCK \) algorithm performs well. Also, \( CLOCK \) is not the best algorithm for managing all the basic condition parts in \( V_{PM} \). In many cases, \( 2Q \) performs better than \( CLOCK \). We leave it as an interesting area for future work to identify other algorithms that perform better than both \( CLOCK \) and \( 2Q \).
We performed the following two experiments:

**Number of bcps experiment.** We fixed \( N=20K \) and tested two cases:

(i) \( \alpha=1.07 \). This is the high skew case. 10% of all the 1M basic condition parts get 90% of the chance of appearing in the \( C_{select} \) of a query.

(ii) \( \alpha=1.01 \). This is the moderate skew case. 21% of all the 1M basic condition parts get 90% of the chance of appearing in the \( C_{select} \) of a query.

In either case, we varied \( h \) from 1 to 5. Recall that \( h \) is the number of basic condition parts in the \( C_{select} \) of a query.

**PMV size experiment.** We fixed \( \alpha=1.07 \) and \( h=2 \). We varied \( N \) from 10K to 30K. Recall that \( N \) determines the size of \( V_{PM} \).

The hit probability is defined as the probability that \( V_{PM} \) can provide some partial results to a query \( Q \). That is, if any of the \( h \) basic condition parts in the \( C_{select} \) of \( Q \) exists in \( V_{PM} \), \( Q \) is “hit.” This definition is different from that in traditional caching [JS94], as our case is about “partial hit” while traditional caching is about “full hit.” In each test case, 1M queries were used to “warm up” \( V_{PM} \). Then the hit probability was reported over the next 1M queries. (We also tested other numbers of “warm up” queries. The results were similar and thus omitted.)

![Hit probability (number of bcps experiment).](image)

For the number of bcps experiment, Figure 6 shows the hit probability results. The y-axis starts from 50%. \( h \) is the number of basic condition parts in the \( C_{select} \) of a query \( Q \). If any basic condition part in the \( C_{select} \) of \( Q \) is “hit,” \( Q \) is “hit.” Hence, the hit probability approaches 100% quickly as \( h \) increases. The larger the \( \alpha \), the more queries focus on a few basic condition parts and thus the more likely these basic condition parts are cached in \( V_{PM} \). Therefore, for a fixed algorithm (either CLOCKS or 2Q) and a fixed \( h \), the hit probability increases with \( \alpha \). For a fixed \( \alpha \) and a fixed \( h \), 2Q performs better than CLOCKS, which is consistent with the results in [JS94].

![Hit probability (PMV size experiment).](image)

Figure 7 shows the hit probability results from the PMV size experiment. The y-axis starts from 70%. The larger the \( N \), the more basic condition parts and their corresponding query result tuples can be stored in \( V_{PM} \), and thus the more likely \( V_{PM} \) can provide some partial results to a query. Therefore, the hit probability approaches 100% quickly as \( N \) increases. Again, for a fixed \( N \), 2Q performs better than CLOCKS.

### 5.2 Influence on Queries’ Run-to-completion Time

In order to show that the PMV method has negligible influence on queries’ run-to-completion time, we did a prototype implementation of our techniques in PostgreSQL Version 7.3.4 [Pos05] for read-only database. Our measurements were performed with the PostgreSQL client application and server running on a computer with one 2.2GHz processor, 512MB main memory, one 40GB disk, and running the Microsoft Windows XP operating system. The default setting of PostgreSQL was used, where the buffer pool size is 1,000 pages. (We also tested larger buffer pool sizes. The results were similar and thus omitted.)

The relations used for the experiments followed the schema of the standard TPC-R Benchmark relations [TPC]:

- customer (custkey, nationkey, …),
- orders (orderkey, custkey, orderdate, …),
- lineitem (orderkey, suppkey, …).

<table>
<thead>
<tr>
<th>s</th>
<th>number of tuples</th>
<th>total size</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer</td>
<td>0.15s M</td>
<td>23s× MB</td>
</tr>
<tr>
<td>orders</td>
<td>1.5s M</td>
<td>114s× MB</td>
</tr>
<tr>
<td>lineitem</td>
<td>6s M</td>
<td>755s× MB</td>
</tr>
</tbody>
</table>

\( s \) is the scale factor of the database. In our experiments, on average, each customer tuple matches ten orders tuples on the attribute custkey. Each orders tuple matches 4 lineitem tuples on the attribute orderkey.

We used the following two query templates:

**Template \( T_1 \):** Find lineitems whose parts were provided by certain suppliers and sold on certain days.

select * from orders o, lineitem l

where o.orderkey=l.orderkey and o.orderdate=\( d_1 \) or … or o.orderdate=\( d_n \)

and l.suppkey=\( s_j \) or … or l.suppkey=\( s_j \);  

**Template \( T_2 \):** Find lineitems whose parts were provided by certain suppliers and sold to certain customers on certain days.

select * from orders o, lineitem l, customer c

where o.orderkey=l.orderkey and o.custkey=c.custkey and o.orderdate=\( d_1 \) or … or o.orderdate=\( d_n \)

and l.suppkey=\( s_j \) or … or l.suppkey=\( s_j \)

and c.nationkey=\( n_1 \) or … or c.nationkey=\( n_1 \);

We built an index on each selection/join attribute. Before we ran queries, we ran the PostgreSQL statistics collection program on all the relations. For either template, due to the low selectivity of the selection attributes, the query plan is not fully pipelined and thus traditional query execution cannot provide any result until it almost finishes.

For the template \( T_1 \), each basic condition part is a 2-tuple (\( d_i, s_j \)). For the template \( T_2 \), each basic condition part is a 3-tuple (\( d_i, s_j, n_k \)). We built two PMVs, one for \( T_1 \) and the other for \( T_2 \). Either PMV contains 20K entries. Each entry can store one basic condition part \( bcp \) and \( F \) query result tuples that belong to \( bcp \). (For each basic condition part, the number of query result tuples that belong to it is greater than \( F \).)

For the template \( T_1 \), its combination factor is defined as \( h=e^{xf} \). For the template \( T_2 \), its combination factor is defined as \( h=e^{xf}x^g \). In operation \( O_1 \), the \( C_{select} \) of each query from \( T_1/T_2 \) is broken into the same number (\( h \)) of condition parts, where each condition part is a basic
condition part itself, and one of these \( h \) basic condition parts exists in the PMV. We performed three experiments. Each experiment was repeated a large number of times (a large number of runs). All the reported numbers are averaged over these runs.

### Number of Tuples

In this experiment, we fixed \( h=4 \) and \( s=1 \). We varied \( F \), the number of query result tuples stored in each entry of the PMV, from 1 to 5. Figure 8 shows the impact of \( F \) on the overhead of our techniques. For a fixed \( F \), the overhead of our techniques for the template \( T_2 \) is greater than that for the template \( T_1 \). This is because \( T_2 \) is more complex than \( T_1 \): \( T_2 \) joins three relations, while \( T_1 \) joins two relations. As a result, the basic condition parts generated from \( T_2 \) are more complex than those generated from \( T_1 \). Also, the query result tuples of \( T_2 \) are longer than that of \( T_1 \). Recall that in our PMV method, both basic condition parts and query result tuples are checked.

The overhead of our techniques increases with \( F \). This is easy to understand, as for each entry of the PMV, \( F \) query result tuples stored there are checked.

### Combination Factor

In this experiment, we fixed \( F=3 \) and \( s=1 \). We varied the combination factor \( h \) from 1 to 10. Figure 9 shows the impact of \( h \) on the overhead of our techniques. The larger the \( h \), the more basic condition parts a query generated. Then the RDBMS needs to spend more time on dealing with all these basic condition parts. As a result, the overhead of our techniques increases with \( h \). Again, for a fixed \( h \), the overhead of our techniques for the template \( T_2 \) is greater than that for the template \( T_1 \).

### Database Scale Factor

In this experiment, we fixed \( h=4 \) and \( F=3 \). We varied the database scale factor \( s \) from 0.5 to 2. The purpose of this experiment is to show that our techniques have negligible influence on queries’ run-to-completion time.

Figure 10 shows both the overhead of our techniques and the query execution time. The lines for “execute \( T_j/T'_j \)” represent the overhead of our techniques. The lines for “execute \( T_j/T'_j \)” represent the query execution time. The y-axis uses logarithmic scale.

Our techniques examine query result tuples rather than the data set. Also, our techniques mainly perform fast in-memory operations (recall that a significant portion of the PMV is cached in memory). Hence, compared to the query execution time, the overhead of our techniques is more than five orders of magnitude smaller. Since the cost of Operations \( O_1 \) and \( O_2 \) is less than the overhead of our techniques, the RDBMS can use the PMV to provide partial query results within a millisecond.

### 5.3 Maintenance Overhead

We use an analytical model to gain insight into the maintenance overhead of PMVs vs. MVs when their base relations get changed. A similar analytical model for MV maintenance has been validated in a commercial RDBMS (NCR Teradata) in [LNE’03]. The goal of this model is not to accurately predict exact performance numbers in specific scenarios. Rather, it is to identify and explore some of the main trends that dominate in the PMV method. (PostgreSQL currently does not support MVs. As a result, we were not able to compare the actual maintenance overhead of PMVs vs. MVs in PostgreSQL.)

Consider the template in Figure 1 and its corresponding partial materialized view \( V_{PM} \). The materialized view \( V_M \) in Figure 2 is the containing MV of \( V_{PM} \). The maintenance overhead of \( V_M \) and \( V_{PM} \) is compared. We make the following simplifying assumptions in this model:

1. \( V_{PM} \) has an index \( I_u \) on \( R.a \). \( V_M \) has an index. Relation \( R (S) \) has an index \( I_o (\delta_S) \) on the join attribute. All the indices are non-clustered.

2. In a single transaction \( T \), \( p \times \Delta R \) tuples are inserted into \( R \) and \((1-p) \times \Delta R \) tuples are deleted from \( R \). These \( \Delta R \) tuples are uniformly distributed on the join attribute. For each tuple \( t_R \), there are \( M \) matching tuples \( t_S \) in \( S \) that satisfy \( t_R.c=t_S.d \). Index nested loops is used for the join with \( S \).

3. The overhead of searching the index \( I_t \) once is a constant \( SEARCH \). If \( M \) tuples \( t_S \) of \( S \) are found to match a tuple \( t_R \) through index search, the overhead of first fetching these \( M \) tuples \( t_S \) and then joining them with \( t_R \) is \( MXFETCH \).

4. The overhead of inserting a tuple into \( V_M \) is \( INSERT_{VM} \). The overhead of deleting a tuple from \( V_M \) is \( DELETE_{VM} \). The overhead of searching the index \( I_u \) on \( V_{PM} \) once is a constant \( SEARCH \).

5. For each tuple \( t_R \) that is to be removed from \( R \), with probability \( q \), no tuple exists in \( V_{PM} \) that has the same \( a \) attribute value as \( t_R \) and thus there is no need to maintain \( V_{PM} \). With probability \( 1-q \), one or more tuples exist in \( V_{PM} \) that have the same \( a \) attribute value.
as \( t_R \). Removing these tuples from \( V_{PM} \) has overhead
\[ \text{DELET}_{V_{PM}}. \]

For each tuple \( t_R \), the total workload \( TW \), which is
defined as the total work done, is used as the cost metric.
For both \( V_M \) and \( V_{PM} \), the same updates must be performed
on base relation \( R \). Because of this, our model omits the
cost of these updates and focuses on the maintenance cost
of \( V_M/V_{PM} \). The total workload for transaction \( T \) is
\( |\Delta R| \) times the average \( TW \) for a tuple \( t_R \).

We first consider the materialized view \( V_M \). For each
tuple \( t_R \), there are two possible cases:
1. With probability \( p \), \( t_R \) is inserted into \( R \). In this case:
   (a) Searching the index \( I_S \) once has overhead
   \( \text{SEARCH} \). (b) Fetching the \( M \) matching tuples \( t_S \) of \( S \)
   and then joining them with \( t_R \) has overhead
   \( M \times \text{FETCH} \). (c) \( M \) join result tuples are obtained.
   Inserting them into \( V_M \) has overhead \( M \times \text{INSERT}_{VM} \).
   Thus the total workload \( TW \) for \( t_R \) is
   \[ \text{SEARCH} + M \times \text{FETCH} + M \times \text{INSERT}_{VM}. \]

2. With probability \( 1-p \), \( t_R \) is removed from \( R \). Compared
to the case of insertion, the \( M \) join result tuples needs to be
deleted (rather than inserted) from \( V_M \), which has overhead
\( M \times \text{DELETE}_{VM} \). Thus the \( TW \) for \( t_R \) is
\[ \text{SEARCH} + M \times \text{FETCH} + M \times \text{DELETE}_{VM}. \]

So for \( V_M \), the average total workload \( TW \) for each \( t_R \) is
\[ \text{SEARCH} + M \times \text{FETCH} + M \times \text{INSERT}_{VM} + (1-p) \times \text{DELETE}_{VM}. \]

Now we consider the partial materialized view \( V_{PM} \). For each
tuple \( t_R \), there are two possible cases:
1. With probability \( p \), \( t_R \) is inserted into \( R \). In this case,
   there is no need to maintain \( V_{PM} \). The total workload
   \( TW \) for \( t_R \) is 0.
2. With probability \( 1-p \), \( t_R \) is removed from \( R \). In this
case (see Section 3.5): (a) The overhead of searching
   the index \( I_P \) on \( V_{PM} \) once is \( \text{SEARCH} \). (b) With
   probability \( 1-q \), one or more tuples with the same \( a \)
   attribute value as \( t_R \) are found in \( V_{PM} \). Removing these
   tuples from \( V_{PM} \) has overhead \( \text{DELET}_{V_{PM}} \).

So for \( V_{PM} \), the average total workload \( TW \) for each \( t_R \) is
\[ (1-p) \times \text{SEARCH} + (1-q) \times \text{DELETE}_{V_{PM}}. \]

In the following, we assume that \( \text{SEARCH} \) takes 0.02
I/O. \( \text{FETCH} \) takes one I/O. \( \text{INSERT}_{VM} \) takes 0.02 I/O
(in-memory append). (A page can contain a large number
of tuples. Hence, the average logging overhead for inserting
a tuple into \( V_M \) is a small percentage of one I/O.)
\( \text{DELETE}_{VM} \) takes two I/Os (one read plus one write).
\( \text{DELETE}_{V_{PM}} \) takes 0.03 I/O (a significant portion of \( V_{PM} \)
cached in memory). Our conclusion would remain
unchanged by small variations in these assumptions.

Setting \( q=95\% \), \( M=1 \), and \( |\Delta R|=1,000 \), we present in
Figures 11 and 12 the resulting performance of both the
MV method and the PMV method. Figure 11 shows the
total workload for transaction \( T \). The y-axis uses
logarithmic scale. The maintenance of \( V_{PM} \) mainly
performs cheap in-memory operations, while the
maintenance of \( V_M \) requires a large number of expensive
I/Os. Hence, for a fixed percentage of insertion \( p \),
maintaining \( V_{PM} \) is at least two orders of magnitude
cheaper than maintaining \( V_M \).

Inserting a tuple into \( V_M \) is less expensive than deleting
a tuple from \( V_M \). Also, there is no need to maintain \( V_{PM} \)
in the presence of insertion into base relation \( R \). As a result,
both the maintenance overhead of \( V_M \) and the maintenance
overhead of \( V_{PM} \) decrease as \( p \) increases. When \( p=100\% \),
the maintenance overhead of \( V_{PM} \) is 0. However, this
cannot be shown in Figure 11, as the y-axis is on
logarithmic scale.

Figure 12 shows the speedup ratio gained by maintaining
\( V_{PM} \) over maintaining \( V_M \). This speedup ratio
increases with the percentage of insertion \( p \), as there is no
need to maintain \( V_{PM} \) during insertion into base relation \( R \).

We can use the techniques in [LNE03] to extend the
above analytical model so that it can handle the situation
that indices are clustered, and/or transaction \( T \) is large
enough for hash/sort-merge join to become the join
algorithm of choice for the join with base relation \( S \). Also,
its straightforward to apply the above analytical model to
the situation of a (partial) MV defined on multiple base
relations, and/or \( T \) contains updates. In either case,
experiments with the extended model did not provide any
insight not already given by the above two-relation model,
so we omit them here.

6. Related Work

Partial Materialized Views

To facilitate exploration of massive data sets, [HH99,
HHW97] proposed using online aggregation to return
approximate answers to the user immediately after a query
is submitted to the RDBMS. Online aggregation focuses
on aggregate queries. In contrast, our PMV method works
for both aggregate and non-aggregate queries.

[AC99, BCG01] proposed building histograms “for
free” by analyzing query results rather than checking the
relation. In our case, if base relations do not change, the
RDBMS both fills content into and updates PMVs “for
free” by analyzing query results.

[SS95, Sto89] use partial indices to reduce index
maintenance overhead. Upon an insertion into a relation \( R \),
the partial index \( I_p \) on \( R \) needs to be maintained
immediately if this insertion satisfies the selection
condition in the definition of \( I_p \). In contrast, the PMV
defined on \( R \) is not maintained immediately.

[OR92] proposed using sample MVs to support
approximate query answering. A sample MV is a random
sample of tuples in a MV. The maintenance of sample
MV's is more expensive than that of PMVs, as randomness needs to be maintained in sample MVs. Also, since a sample MV does not focus on hot query result tuples, the probability that it can provide partial results to a query is low. In a read-only environment, [GLR00] proposed using icicles samples to support approximate query answering for key-foreign key join queries. In contrast, PMVs work in a general environment that allows updates.

[DRS98] uses chunks to cache OLAP query results in the middle tier. [DRS98] focuses on aggregate queries in a read-only environment, and imposes an order on each dimension if no implicit order exists. In contrast, our method works for both aggregate and non-aggregate queries in a general environment that allows updates, and does not impose non-natural orders on attribute values.

In a data stream environment, to speed up the processing of continuous multi-way windowed join queries, [BMW05] proposed caching a subset of the join result tuples of some of the streams. If a key value $v$ exists in the cache, all the join result tuples related to $v$ must also exist in the cache. This requires maintaining the cache immediately upon arrival of new tuples from the streams. In contrast, upon insertion into base relations, PMVs are not maintained immediately.

In a distributed data integration environment, [HZ96] and [Ora00] define a PMV as a MV whose definition contains a subset of all the attributes and a where clause, respectively. Both PMV definitions are different from the one used in this paper.

**Ranking Query Result Tuples**

[CDH04] uses attributes that are not specified in the query to rank result tuples. Before it can take effect, the ranking method in [CDH04] needs to first gain some knowledge by analyzing both some previous workload and the data set. This is the start-up cost. The gained knowledge is static and can become imprecise if either the query pattern or the data set changes. In contrast, our ranking method is more dynamic. It does not have a start-up cost while the popularity information kept in the data structure gets continuously updated. This is especially advantageous if either no previous workload is available or the workload pattern changes significantly over time.

To address the information overload problem, [CCH04] proposed categorizing query result tuples. This is orthogonal to our approach of ranking query result tuples.

In the case that no tuple satisfies the query condition completely, [ACD03, BCG02, Fuh90, Mot88] proposed ranking tuples according to their “proximity” to the query. In a data integration environment, [BM02] ranks query result tuples according to the credibility of data sources, while [Coh98] ranks query result tuples according to textual similarity. [ACD02, HP02] proposed keyword search in RDBMS. All these work focus on a different environment from ours.

7. Conclusion

We have presented a partial materialized view method that can provide transactionally consistent, immediate partial query results to the user without increasing queries’ run-to-completion time much, by caching hot query results in PMVs. Our experiments with a simulation study, a theoretical analysis, and a prototype implementation in PostgreSQL show that PMVs have low storage and maintenance overhead. In a large number of cases, they can provide partial results almost instantly. Many PMVs can reside in the RDBMS simultaneously. Also, our method has negligible influence on queries’ run-to-completion time. Furthermore, our techniques are extended to address the information overflow problem, the result of which is a method for ranking query result tuples according to their popularity. Both the PMV method and the query result ranking method can facilitate the exploration of massive data sets.

**Acknowledgements**

We would like to thank Jiuxing Liu, Jeffrey F. Naughton, Ying-li Tian, Michail Vlachos, Haijing Wang, Michael W. Watzke, and Hao Yang for helpful discussions.

**References**
