Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'2012), pp. 42 - 56, IBM, Toronto, Canada, Nov. 5 - 7, 2012

DMVI: A Dynamic Materialized View Index for Efficiently Discovering Usable Views for Progressive Queries *

Chao Zhu† Qiang Zhu†⋆ Calisto Zuzarte‡⋆ Wenbin Ma‡

†Department of Computer and Information Science
The University of Michigan, Dearborn, MI 48128, USA
⋆IBM Canada CAS Research, Markham, Ontario, Canada
‡IBM Canada Software Lab, Markham, Ontario, Canada

Abstract

Progressive queries (PQ) are a new type of query emerged from numerous data intensive applications. A user formulates a PQ in several steps using a set of inter-related step-queries (SQ). Efficiently processing PQs in a DBMS is crucial in supporting these applications. In our previous work, we introduced a materialized view based approach to efficiently processing PQs, where our focus was on selection of promising materialized views. The problem of how to efficiently find usable views to answer SQs for a PQ remained open. In this paper, we present a new index technique, which is called the dynamic materialized view index (DMVI), to rapidly discover usable views to answer a given SQ. The structure of the index and the strategies to construct, maintain and use the DMVI are discussed. The Experimental results demonstrate that our technique is quite promising in improving the performance of the materialized view based query processing for PQs.

Keywords: Database, query processing, query optimization, progressive query, materialized view, index

*Research was partially supported by the IBM Canada Software Laboratory and The University of Michigan.

1 Introduction

The problem of analyzing a large amount of data in databases has recently received significant attention because of the rapid growth of numerous data intensive applications (e.g., astronomy, biology, and social media). In such data intensive applications, a new type of query, which is so-called progressive queries (PQ) [25], is demanded. Unlike a conventional query, a PQ consists of a set of inter-related step-queries (SQs). Each SQ is formulated by a user based on the result(s) returned by the previous SQ(s). Hence, the user gradually approaches his/her desired result by issuing several SQs to the database.

As an illustration, let us consider the following example. Assume that a user wants to buy a car. In the first step, he/she searches all the cars available at a website. However, the result returned is too large. Hence, in the second step, he/she adds a search condition to only return cars which were manufactured in the USA. After analyzing the result, he/she prefers to select a Ford car. Therefore, in the third step, he/she checks the details (e.g., prize, configuration, etc.) of all the Ford cars to make a final decision.

From the above example, we can see that the main characteristic of a PQ is that the SQs of a PQ cannot be known beforehand. Each SQ is formulated based on the result(s) of the previous SQ(s). Hence, to execute such unpredictable SQs, the results of previous SQs of

each in-process PQ have to be kept in the system (as one type of (temporary) materialized views) until the PQ is completed. On the other hand, it is desirable to retain some popular results (i.e., those being utilized frequently) of SQs in the system (as another type of (critical) materialized views) even after their corresponding PQs are completed so that the SQs of future PQs can utilize these results to improve their processing efficiency. To achieve this goal, we introduced a dynamic materialized view based approach to processing PQs in [24]. A model was developed to determine if the result table for an SQ of a completed PQ should still be kept in the system as a (critical) materialized view. Both the popular results of SQs from completed PQs and the results of previous SQs from in-process PQs are kept as materialized views in a view storage (VS).

In our previous work, we mainly focused on how to select promising materialized views for processing PQs. A straightforward linear scan method was assumed to search/match a desired materialized view for answering a given SQ in a PQ. To improve the searching efficiency, in this paper, we present a new index technique, called the dynamic materialized view index (DMVI), designed specifically for indexing materialized views for PQs. The DMVI is dynamically built for all the (materialized) views in the VS. When a new view v is added to the VS, a search path is created in a tree structure of the DMVI for v and the relevant information (including some bitmaps for refined pruning) of v is stored at the end (leaf) of the path. Using the $DMVI$, the views in the VS can be efficiently managed and searched to answer SQs of PQs. Algorithms/strategies to construct the DMVI, maintain the DMVI, and search desirable views using the DMVI are presented. As demonstrated in our experimental results, the DMVI can be utilized to efficiently discover usable views for answering a given SQ and effectively remove many undesirable views from consideration for view matching. Since checking if a view matches a given query is computationally expensive, removing undesirable views from consideration for view matching can significantly improve the performance of query processing based on materialized views.

Indexing and view materialization are two important techniques for improving query performance. Extensive work has been reported for each of them in the literature. There is also a substantial body of work exploring these two techniques together. Roussopoulos [18] presented a method to select a set of views and maintain an index for each of them to support efficient query processing. The index of each view contains pointers to the tuples of the base tables used to construct the view. Kimura *et al.* [10] adopted a form of Integer Linear Programming (ILP) to select the best set of materialized views and indexes for a given workload under given database size constraints, taking into consideration of the effect of correlated attributes. Bellatreche *et al.* [3] introduced a technique to select optimal or near optimal join indexes for a given set of OLAP queries, where the indexes can be built on materialized views as well as dimension and fact base tables. Talebi *et al.* [21] examined the exact and inexact methods for selecting materialized views and indexes to efficiently process OLAP queries. Aouiche and Darmont [2] applied a data mining process to select candidate materialized views and indexes in data warehouse environments. Graefe and Zwilling [7] studied techniques for transaction support for indexed summary views. Kuno and Graefe [11] proposed a deferred technique to maintain indexes and materialized views. However, all the above work considered indexes that were built on base tables and/or materialized views to accelerate the processing of queries on the database in conjunction with materialized views. In contrast, the index technique we introduce in this paper directly uses materialized views, instead of the underlying data, as indexed objects, with a goal of removing as many undesirable views as possible from consideration for view matching during query processing based on materialized views.

The most related work in the literature is a so-called collective index method introduced by Zhu *et al.* [25]. It is the only existing index technique specifically designed for processing PQs. The main idea of this technique is to construct a special index structure so that a collection of member indexes on an input table of an SQ in a PQ can be efficiently transformed into indexes on the result table of the SQ, which

can be utilized to process the subsequent SQs of the PQ. However, like many other existing indexes, the collective index is also built for the underlying data objects rather than for materialized views used directly as indexed objects, which is different from our index in this paper.

Our previous work includes one studying how to apply materialized views to optimize a special type of PQs, called monotonic PQs, in [23] and another one studying how to apply materialized views to optimize generic PQs in [24]. However, the focus of our previous work was on how to select materialized views, while the focus of this work is on how to efficiently find desirable materialized views to answer an SQ in a given PQ. Our indexing technique can be used in conjunction with existing view matching [4, 8, 12, 13, 15, 16, 20, 22] to identify desirable views for answering a given SQ. To our knowledge, no similar work has been reported in the literature.

The rest of this paper is organized as follows. The preliminaries and background knowledge are introduced in Section 2. The dynamical materialized view index (DMVI) and related algorithms/strategies are presented in Section 3. Experimental results are reported in Section 4. The conclusions are summarized in Section 5.

2 Preliminaries

In this section, an overview of some related concepts that were introduced in our earlier work [24] is given. Specifically, Section 2.1 discusses two types of materialized views for processing progressive queries, and Section 2.2 presents the concept of a materialized view storage.

2.1 Two types of materialized views

As mentioned in Section 1, the main characteristic of a PQ is the unpredictability of its SQs. The user cannot know what the next SQ is before the result(s) of the previous SQ(s) is returned. Therefore, the result tables for all the finished SQs of an in-process PQ have to be kept in the system because they may be used to execute the following SQs. Since multiple PQs are allowed to execute simultaneously, the result tables for the finished SQs of all in-process PQs in the system are kept as materialized views, which we call them the temporary materialized views (TMV).

Usually, after a PQ is completed, all the TMVs for its SQs are removed from the system. However, it is possible that the result tables of some SQs of a completed PQ are popular, e.g., they are frequently used by the SQs of other PQs. In such a case, they may also have a high chance to be used to optimize future SQs. Thus, the popular results of SQs of completed PQs are still kept in the system and constitute another type of materialized views, which we call them the critical materialized views (CMV). Generally, the lifetime of a TMV is shorter, which is removed after its corresponding PQ is completed, than a CMV, which is kept in the system until the allocated view space overflows.

2.2 View storage

To store the materialized views, the system allocates a space, called the view storage (VS). At the logical level, we divide the VS into two subspaces according to the two different view types, i.e., the temporary view space (TVS) for the TMVs, and the critical view space (CVS) for the CMVs.

However, at the physical level, the TMVs and the CMVs are mixed and stored together in the VS. TMVs and CMVs are differentiated by their view type identifiers, which are stored in our new index. Figure 1 shows the structure of the view storage.

3 Dynamic materialized view index

To efficiently find the usable views to answer SQs, we present a so-called dynamic materialized view index (DMVI) to index all the views in the VS in this section. The structure of the DMVI is introduced in Section 3.1. An encoding-based matching technique is presented in Section 3.2. The DMVI construction issue is discussed in Section 3.3. The view

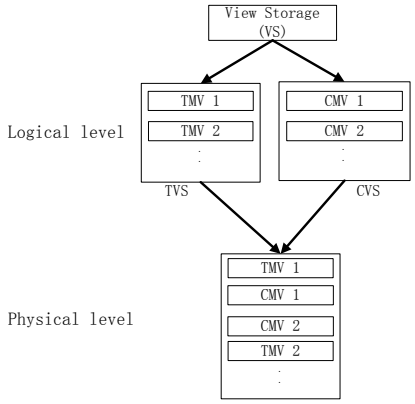

Figure 1: The structure of the view storage

search by using the DMVI and the maintenance issue for the DMVI are discussed in Section 3.4.

3.1 Index structure

In this work, we mainly focus on addressing the issue on how to efficiently answer SQs using the available TMVs and CMVs to reduce the SQ processing cost. In other words, we want to develop an efficient method to search for possibly usable views from the VS to answer a given SQ. A straightforward way to do this is to apply a sequential scan to the VS for checking each view to see if it is usable to answer the given SQ. However, the overhead of this approach is usually high, especially when the number of views is large. Note that, in general, matching a view with a given query (i.e., to check if the former can be used to answer the latter) is computationally expensive. Hence, developing a view search technique to rapidly identify desirable views to answer a given SQ (without having to check all the views) is crucial in achieving efficient optimization for PQs.

However, the special characteristics of the materialized views for PQs raise some new challenges to develop such an access method for the views. The first challenge is that all the materialized views are dynamically generated while PQs are processed. Therefore, the view access method has to support how to dynamically add new views. The second challenge is the high complexity of maintaining the VS since the TMVs are created and removed with a high frequency. Furthermore, all of the CMVs in the VS are transformed/selected from the TMVs. Therefore, the view access method has to support efficient maintenance of the VS.

In this paper, we develop a dynamic materialized view index to efficiently find the views that are possibly usable for answering the SQs. The main idea is to dynamically build an index for all the materialized views in the VS. For each materialized view v, its corresponding query expression contains the input tables of v. Unlike a conventional index on a table, which uses the attribute values as search keys to find the satisfied rows of the table, the DMVI uses the identifiers of the view input tables as the search keys to find the desirable views. We call the set of all the input tables of an SQ (view) as the domain of the SQ (view). Hence, we also call an input table as a domain table. The criterion used to search the DMVI is that the domain of a desirable view is the same as that of the given SQ. Note that, although a view may also be usable if its domain is a superset of the domain of the given SQ, such a view usually does not match the given SQ as closely as a view whose domain equals to that of the SQ. To reduce the number of views returned from the index search, we do not consider the superset criterion. On the other hand, it does not guarantee that the views returned from the equal-domain criterion are always usable for answering the given SQ. Hence, a refined checking on the usability of the returned views is required. Therefore, the DMVI is an approximate index with an objective to return a set S of materialized views for a given SQ such that (1) the views in S match the given SQ as closely as possible; (2) the size of S is as small as possible. The views in S are then examined to see if they can be used to efficiently process the given SQ.

The data structure of the DMVI is an ordered tree in which there is an order among the children of a node. Each leaf node of the tree represents a materialized view and keeps the relevant view information. Each internal node n (except the root) represents a domain table used by the materialized views for the leaf nodes of the corresponding subtree rooted at n, namely, n is associated with the identifier of the

corresponding domain table. The root node of the tree is the starting point for a search. The domain tables labeled on the path between the root and a leaf node for a materialized view v are all the domain (input) tables for v. Note that, for simplicity, we will use a domain table and a domain table identifier interchangeably. Figure 2 shows an example of the dynamic materialized view index, where four materialized views v_1, v_2, v_3 and v_4 are indexed and four domain tables t_1, t_2, t_3 and t_4 are used by the views. The domains of v_1 and v_2, for example, are t_1, t_2 and t_1, t_3, respectively. The tables in a domain are used as a search key for the corresponding materialized view in the DMVI.

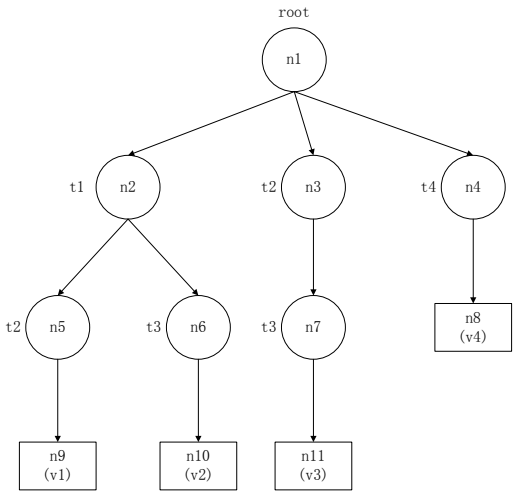

Figure 2: An example of the DMVI

As we mentioned earlier, the first challenge to create an index for the views is that all the materialized views are dynamically generated. To tackle the challenge, the DMVI must support how to dynamically incorporate new views. In the previous example, assume that we have another materialized view v_5 whose domain tables are: t_1 and t_4. v_5 can be indexed in the tree in two alternative ways: (1) create an internal node n_{12} labeled with t_4 and connect n_{12} to n_2 as a child, then create a leaf node n_{13} for v_5 and connect n_{13} to n_{12} as a child; (2) create an internal node n_{14} labeled with t_1 and connect n_{14} to n_4 as a child, then create a leaf node n_{15} for v_5 and connect n_{15} to n_{14} as a child. To avoid ambiguity, we need to define a priority order for the nodes for insertions. The DMVI is constructed by following the priority order defined as follows.

Suppose we want to index a new materialized view v in the DMVI. Assume that node n_i is either the root or the last internal node that has been chosen on the search path of v in the DMVI, and n_i has internal nodes n_1, n_2, ..., n_m as m ordered (from the left to the right) children. The domain tables represented by these internal nodes are t_1, t_2, ..., t_m, respectively. The priority order used to determine the next node on the search path for v is given by the following:

Case 1: n_1 is chosen to be the next node on the search path for v if the domain of v contains table t_1.

Case 2: n_2 is chosen to be the next node on the search path of v if the domain of v contains t_2 but not t_1.

.....

Case m: n_m is chosen to be the next node on the search path of v if the domain of v contains t_m but not t_1, t_2, ..., or t_{m-1}.

If none of node n_j ($1 \leq j \leq m$) has its labeled table contained in the domain of v and the domain of v still has tables that have not been labeled on the search path of v, a new internal node for each unlabeled table is created one at a time until all domain tables of v are labeled on its search path. If all the domain tables of v have been labeled on the search path of v, a leaf node representing v is created and connected to the last internal node on the search path.

The above rule implies that, in the DMVI, a left child node has a higher priority to produce descendants. In the previous example, two candidate nodes considered are n_2 and n_4. n_2 is on the left of n_4. Hence, n_2 rather than n_4 is chosen as the next node on the search path of view v_5. n_3 is excluded because table t_1 represented by n_3 is not in the domain of v_5.

From the rule, we observe two properties of the DMVI.

(1) At the first level of the tree (i.e., the level just below the root), if the leftmost internal node n is labeled with a domain table t, then all the indexed views whose domains contain t can be found in the subtree rooted at n.

(2) At the first level of the tree, if an in-

ternal node n that is not the leftmost node is labeled with a domain table t, then all the indexed views whose domains contain t can be found in the subtree rooted at n or the subtrees rooted at the left brothers of n. Note that the internal node with t as a label must be at a level higher than one in the latter case.

In general, given an internal node n_m labeled with a domain table t at the m-th level of the tree, if an indexed view v is under the subtree rooted at n_m, t must be the m-th domain table of v; if v is under a subtree rooted at a left brother of n_m, the node with t as a label can only be found at a level higher ($>$) than m; if v is under a subtree rooted at a right brother of n_m, the node with t can only be found at a level lower ($<$) than m.

The second challenge for developing a method for accessing views for PQs, which we mentioned earlier, is the high complexity of maintaining the views in the VS. To tackle the challenge, the DMVI has to support a way to efficiently maintain the TMVs and CMVs.

On one hand, the DMVI should support a logical level transformation from a TMV to a CMV (not physically move the view). As we mentioned earlier, each leaf node of the DMVI stores the information of a view. The information of a view includes the view name, a view type indicator to differentiate the view types (TMV or CMV), etc. We will discuss the details of the view information structure in Section 3.2. When a transformation occurs, the view itself and all its related information are kept unchanged except the view type indicator.

On the other hand, it is not trivial to maintain the TMVs and CMVs in the DMVI. In general, an SQ can be formulated using as inputs the external (base) tables, the result tables of the previous SQs of in-process PQs (i.e., TMVs) and the result tables of the SQs of historical PQs (i.e., CMVs). Therefore, the TMVs and CMVs can also be the domain tables of an SQ besides the external tables. This implies that the CMVs and TMVs can appear in the search keys for the views indexed in the DMVI. For CMVs, their long lifetimes make them relatively easy to handle. However, for TMVs, the frequent updates make their management more challenging.

Let us consider an example. After an SQ of a PQ is executed, its result table is saved as a TMV v_1 and indexed in the DMVI. Assume that v_1 is used by some other SQs. When the PQ is completed, the SQ needs to be discarded or transformed into a CMV. In the former case, view v_1 should be removed from the VS. As a result, the search keys (i.e., the search paths) of all the views whose domains include v_1 become invalid.

To tackle this challenge, we have designed an algorithm to rebuild the invalid search paths in the DMVI. The details of the algorithm will be discussed in Section 3.4.

To apply the algorithm, we have to find all the views whose domains contain the discarded view. A straightforward way to do this would be to traverse the DMVI. However, we observe that we can make use of the two properties of the DMVI to improve such a search. According to the first property of the DMVI, at the first level of the tree, if an internal node n is the leftmost child of the root and its represented domain table t becomes invalid, all the views whose domains contain t can be found in the subtree rooted at n. Furthermore, according to the second property of the DMVI, at the first level of the tree, no matter where node n whose domain table t becomes invalid is, all the views whose domains contain t can be found in the subtree rooted at n or the subtrees rooted at the left brothers of n. Therefore, there is no need to search the right brothers. This is one of the reasons why the priority order for node insertions was defined as such.

Since TMVs need to be removed and transformed frequently while CMVs and external tables are quite stable, the search keys which involve TMVs are easy to become invalid. When a view v is indexed in the DMVI, if the domain of v contains some TMVs, then the TMVs (internal nodes) are picked up first and inserted into more left branches than its brother nodes which represent CMVs and/or external tables. The tree construction details will be discussed in Section 3.3.

Let us consider the following example. Assume that, in a given DMVI, four TMVs $tmv_1 \sim tmv_4$ and one CMV cmv_1 are indexed; four external tables $et_1 \sim et_4$ are used as domain tables; tmv_1, tmv_2, and cmv_1 are also used as

domain tables. The DMVI is shown in Figure 3.

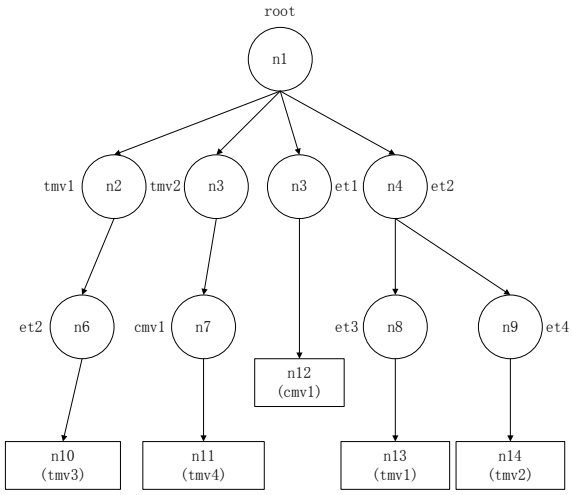

Figure 3: An example of the DMVI with views as domain tables

In the figure, we can see that tmv_1 and tmv_2 are assigned to the first level nodes n_2 and n_3 in the DMVI. If tmv_1 becomes invalid, to find all the views whose domains contain tmv_1, only the subtree rooted at n_2 needs to be searched. If tmv_2 becomes invalid, to find all the views whose domains contain tmv_2, only the subtrees rooted at n_2 and $n3$ need to be searched. How to rebuild an invalid search path after it is found will be discussed in Section 3.4.

3.2 View bitmap-based matching in the DMVI

The main purpose of introducing the DMVI is to efficiently find usable views to answer the SQs. Using the above structure of the DMVI, the system filters out undesirable views in the VS and only returns the views which share the same domain with the SQ to be processed. However, as mentioned earlier, the returned views are not guaranteed to be usable for answering the SQ. To reduce the number of cases in which we have to directly examine a returned view for its usability, which is computationally expensive, we adopt an efficient refined filtering, which is called the bitmap-based matching.

The query expression of a view is encoded as several bitmaps in a special way. The bitmaps are saved in the DMVI. As mentioned before, each leaf node stores the information of a view. The information includes: the view name/identifier, the view type, the query expression of the view, the view bitmaps and encoding method, and the view location. Hence, the view bitmaps can be accessed in the leaf nodes of the tree. As we will see, the encoding method depends on the domain of a query expression (for a view or an SQ). In other words, the encoding method is the same for those query expressions that share the same domain. When an SQ *sq* arrives, the system uses the DMVI tree discussed in Section 3.1 to find all the leaf nodes whose associated views share the same domain with *sq*. The query expression of *sq* is then encoded by using the same encoding method for these views. For each view in the returned set, its bitmaps are compared with those for *sq*. If the view is found not usable for *sq*, it is filtered out. Note that our bitmap matching is different from a conventional view matching. As we will see, even if a view passes the bitmap matching, it still may not be usable for answering *sq*. A final direct view matching examination is needed. However, using the bitmap matching technique, the number of candidate views for the direct view matching examination is further reduced.

Like most related work in the literature, we consider the common select-project-join query expressions (for SQs and views) and assume that the (qualification) conditions for the select and join operations are in the conjunctive normal form (CNF) in the following discussion.

To encode a query expression (for an SQ or view), an encoding method is required. As mentioned above, our encoding method depends on the domain of the query expression. Specifically, the encoding method creates three bitmaps: one for each operation (i.e., project, select and join) of the query expression. Given a domain T (consisting of input tables), its encoding method is described as follows:

(1) *The project bitmap.* For each domain table t in T, the bitmap for a project operation contains a bit for each attribute a of t. If a appears in the target attribute list of the project operation, the bit for a in the bitmap is set to 1. Otherwise, the

bit for a is set to 0. Let us consider a simple example. Assume that T contains only one domain table t. t has four attributes: a_1, a_2, a_3, and a_4. The encoding method allocates a bit for each of a_1, a_2, a_3 and a_4. If the given query expression is: *select* a_1, a_2 *from* t, then the bits for a_1 and a_2 are set to 1 and the bits for a_3 and a_4 are set to 0. Hence, the bitmap for this query is 1100.

(2) *The select bitmap.* For each domain table t in T, every attribute a of t is analyzed and its range of values is divided into n subranges. n can be 1 if the range of a cannot be divided. For example, it is difficult to divide the range of an attribute a_1 representing the paper title for a paper table. A bit segment which contains n bits is assigned for a, one bit for each subrange.

If the range of a are restricted by one or more clauses in the CNF of the condition of the select operation, the bits in the bit segment for a are set accordingly. Let us consider a simple example. Assume that attribute a_1 represents the age of a person and its range is from 0 to 150. The encoding method divides the range of a_1 into 5 subranges: [0, 30], [31, 60], [61, 90], [91, 120] and [120, 150]. Then the encoding method assigns a bit segment which contains 5 bits to a_1. If the given query expression is: *select* a_1 *from* t *where* $a_1 > 70$. In the bit segment for a_1, the bits for the subranges with at least one value satisfying the condition are set to 1, and the other bits are set to 0. Note that, although only some (not all) values in the subrange [61, 90] satisfy the query condition, its corresponding bit in the bit segment is set to 1. Hence, the bit segment for a_1 in this example is 00111.

If the range of a is not restricted by any clause in the CNF of the condition of the select operation, all the bits of the bit segment of a are set to 1. In other words, we take a conservative approach by keeping all the subranges.

The bitmap for the select operation consists of all the bit segments for the attributes of the domain tables in T.

(3) *The join bitmap.* To generate a bitmap for a join operation, all the possible attribute pairs that can be used for a join operation are discovered from the domain tables in T first. Such an attribute pair is called a join pair. For each join pair, it is assigned with a bit in the bitmap. If a join pair appears in the join condition of the query expression (with any comparison such as $=, <, >$), then its bit in the bitmap is set to 1. Otherwise, the bit is set to 0.

Now let us discuss how to use the bitmaps to compare a given SQ sq with a view v. As we mentioned earlier, the main purpose of the bitmap matching is to filter out unusable views that are returned by the searching on the DMVI tree. The key idea is to prune some views which has no containment relationship with the SQ, namely, the views do not contain the result of the SQ.

Assume that v and sq are encoded using the same encoding method (i.e., v and sq have the same domain). The process of the bitmap matching can be done in three stages.

In the first stage, the project bitmap pbm_1 for v and the project bitmap pbm_2 for sq are compared. The bit value of 1 represents that its corresponding attribute appears in the result of the relevant query. Therefore, if the bit for an attribute a in pbm_1 is 0 but in $bpbm_2$ is 1, it means that a is in the result of sq but not in v. Hence, the system can conclude that v cannot contain the result of sq. To compare pbm_1 and pbm_2, the system performs a bitwise complement on pbm_2 first and then a bitwise OR on pbm_1 and pbm_2. If the result contains 0, it means v cannot contain the result of sq. For example, if pbm_1 is 00111, pbm_2 is 10011. First, a bitwise complement is performed on pbm_2 to get 01100. Then, a bitwise OR is applied on pbm_1 and pbm_2, resulting in 01111. Since the result contains 0, v cannot contain the result of sq. Hence, v is filtered out.

In the second stage, the select bitmap sbm_1 for v that has passed the first stage test and the select bitmap sbm_2 for sq are compared. If the bit segment for an attribute a in sbm_1 indicates a narrower range (i.e., missing some 1's) than the bit segment for a in sbm_2, it implies that v restricts the range of a in its select

operation more, i.e., the select operation filters out more rows than sq. Hence, there may exist some rows in the result of sq but not in v. In other words, v cannot contain the result of sq. In this case, v should be filtered out. Similar to the first stage, a bitwise complement is performed on sbm_2 first, then a bitwise OR is applied on sbm_1 and sbm_2. If the result contains 0, it implies v cannot contain the result of sq. For example, assume that each of the two bitmaps for v and sq consists of only one bit segment, say, sbm_1 is 00011 and sbm_2 is 00001. First, a bitwise complement is performed on sbm_2, resulting in 11110. Then, a bitwise OR is applied on sbm_1 and sbm_2, resulting in 11111. Thus, the containment relationship between v and the result of sq is still unknown. v needs to be further examined.

In the third stage, the join bitmap jbm_1 for v that has passed the second stage test and the join bitmap jbm_2 for sq are compared. Each bit in the join bitmap indicates the occurrence of a pair of join attributes in the condition of the join operation for a given query. If the join bitmaps of v and sq are different, it is very difficult to determine if the containment relationship between v and sq holds, which makes the view matching examination difficult. To reduce the view matching cost, we exclude such views from consideration. Hence, we require jbm_1 and jbm_2 to be the exactly same. Otherwise, v is filtered out.

From the above discussion, we can see that our complete DMVI consists of the tree structure discussed in Section 3.1 and the bitmaps presented in this section. The objective of the DMVI is to efficiently filter out those views that are clearly unusable for answering the given SQ or very difficult to perform view matching, resulting in a small set of candidate views. The candidate views are then further examined (i.e., perform view matching) to see if they are indeed usable for answering the given SQ.

3.3 The DMVI construction

In this section, we discuss the details of the DMVI construction. The DMVI is dynamically created. If no view is indexed, the DMVI contains only a root node. When the result table of an SQ becomes a materialized view, the view is added into the VS and indexed in the DMVI. The main idea is to build a search path p for v in the DMVI using the domain tables of v as the elements of the search key. Each internal node in p represents a domain table (i.e., a search key element) of v. The leaf node which is at the end of p represents v (including all its relevant information).

In Section 3.1, we defined a priority order for the existing internal nodes of the DMVI tree so as to determine the next node on the search path for a new view that is being inserted into the tree. However, there are two other related orderings that need to be decided when indexing a new view v, that is, the order of the domain tables of v and the order of the domain tables for the entire workload. The former determines which domain table (internal node) of v is inserted (created) first. The latter determines where to insert a domain table of v in the tree in relation to other domain tables in the DMVI. Let us consider the following example. Assume that a domain table t of v is to be added as a child node of n. n has one existing child node cn. How to insert t is ambiguous because t can appear as the left brother or the right brother of cn. In this case, the order of the domain tables for the entire workload is required. The policy we use is that the domain table with a higher priority appears on the left. We have mentioned the idea of this second order briefly in Section 3.1.

To solve the above two ordering issues, we assign different priorities to different domain tables. A two-level priority rule is used to order the domain tables. At the first level, the domain tables are recognized only by their types (TMVs, CMVs, or external tables). The priority order for these three domain table types from high to low are: TMVs, CMVs, and the external tables. At the second level, within each table type, an older domain table is given a higher priority. With the two-level priority rule, the order of the domain tables of v and the order of the domain tables in the entire workload can be determined.

Let us consider an example: given a set of domain tables: $T = \{cmv_1, et_2, et_1, tmv_2, tmv_1\}$, where cmv_1 is a CMV; tmv_1 and tmv_2 are TMVs; et_1 and et_2 are external tables. To determine the order of the tables in T, the ta-

bles are first sorted by the table types: tmv_2, tmv_1, cmv_1, et_2, et_1. After that, the tables are further sorted by their time order within each type. The ordered list is: tmv_1, tmv_2, cmv_1, et_1, et_2.

Now let us discuss how to index a new view v in the DMVI. The basic process is described as follows. All the domain tables of v are sorted by the two-level priority rule. The domain tables in the entire workload are also sorted by the rule and saved in a workload list. The domain tables of v are picked up one at a time in the given order. For the first picked domain table t_1, each node of the root at the first level of the tree is checked. If there exists an internal node n_1 labeled with t_1, then n_1 is picked as the first node in the search path for v and used to lead the rest of the domain tables (internal nodes) of v. In this case, the insertion process is recursively applied to n_1 and the next domain table for v. Otherwise, a new internal node n_2 is created for t_1. In the latter case, we need to decide where to insert n_2. Clearly, n_2 must be a child node of the root. In the DMVI, if a node has multiple child nodes, the order (from the left to the right) of these child nodes is determined by the order in the workload list. Thus, each child node of the root is compared with n_2 one by one from the left to the right. If a node n_3 in the DMVI is found to be the first node to appear after n_2 based on the order from the workload list, n_2 is inserted as the immediate left brother of n_3 and an edge from the root to n_3 is created. If no node after n_2 is found in the DMVI, it means all the child nodes of the root should appear before n_2. Hence, n_2 is inserted as the rightmost child node of the root, and an edge from the root to n_2 is created. n_2 is used to lead the rest search path with a new internal node created for each remaining domain table of v. After all the domain tables of v are associated with the search path, a leaf node which contains the information (including bitmaps) of v is created and linked to the last domain table of the search path. If multiple leaf nodes (views) share the same search path, the order of the leaf nodes is unimportant, which can be given, for example, on the first-come-first-serve base.

The following recursive algorithm describes the formal procedure for inserting (indexing) a new view (v) into the DMVI ($dmvi$). At the beginning, the algorithm is invoked using the root node of the DMVI (for $cnode$) and the complete list of domain tables of the view (for $cdomainlist$). It assumes that the input lists of the domain tables for both the view ($cdomainlist$) and the workload ($workloadlist$) have been sorted using the two-level priority rule.

ALGORITHM 3.1 : **ViewInsertion(**v**,** $dmvi$**,** $cnode$**,** $cdomainlist$**,** $workloadlist$**)**
Input: (1) the new view v for indexing; (2) the DMVI $dmvi$; (3) the node $cnode$ in the DMVI that leads the remaining search path of the view; (4) the list $cdomainlist$ of current (unprocessed) domain tables of the view; (5) the list $workloadlist$ of domain tables in the workload;
Output: the revised DMVI.
Method:

```
1.  if cdomainlist is empty then
2.    create a leaf node lnode with view info for v;
3.    link lnode to cnode as the rightmost child;
4.    return;
5.  else
6.    let ftable be the first domain table in cdomainlist;
7.    remove ftable from cdomainlist;
8.    if there exists a child node dnode of cnode in dmvi
        associated/labeled with ftable then
9.      ViewInsertion(v, dmvi, dnode, cdomainlist,
          workloadlist);
10.   else
11.     create an internal node inode for ftable;
12.     if cnode has no child node then
13.       link inode to cnode as the only child;
14.       ViewInsertion(v, dmvi, inode, cdomainlist,
            workloadlist);
15.     else
16.       find the right position for inode among the
            ordered children of cnode based on
            the order given in workloadlist;
17.       link inode to cnode as a child at the right
            position;
18.       ViewInsertion(v, dmvi, inode, cdomainlist,
            workloadlist);
19.     end if
20.   end if
21. end if.
```

Using the above algorithm, we can build the DMVI dynamically by inserting every new view when it becomes available.

3.4 The view searching using the DMVI and the DMVI maintenance issues

Let us first describe how to apply the DMVI to find the desirable views in the VS for the view matching. When an SQ sq is issued, the set T of domain tables of sq is extracted and sorted using the two-level priority rule. According to T, a proper encoding method is applied to generate the bitmaps for sq. The ordered tables in T are then used as a search key to find the set

lns of leaf nodes whose views share the same domain with *sq*. For each node *ln* in *lns*, the bitmaps of the view for *ln* are extracted and compared with those of *sq*. If the view is not filtered out by the three-stage bitmap matching, the view is found and returned. Using the DMVI, as we will see in Section 4, the number of the candidate views that are used to perform the final direct view matching for an SQ is significantly reduced.

The last issue we want to discuss is how to maintain the DMVI when a view v is removed from the VS. First, the domain of v is used as the search key to find the set *lns* of all the leaf nodes whose corresponding views have the same domain in the DMVI. Second, the leaf node *ln* in *lns* which was assigned for v is found. After that, the parent node *pn* of *ln* in the DMVI is found and checked. If *pn* contains multiple child nodes, it means *pn* is still on the search path(s) for other views. Hence, only *ln* is removed. If *pn* has *ln* as the only child, *pn* becomes useless after *ln* is removed. Thus, both *ln* and *pn* are removed. This node removal may be propagated up the root in the DMVI.

However, only removing the useless nodes on the search path for v in the DMVI is not sufficient since the search paths of all the views that use v in their search keys also become invalid in the DMVI. To solve this problem, we adopt an approach to efficiently rebuild all the invalid search paths in the DMVI. The work is done in two stages. In the first stage, every view v' which includes v in its domain is found. The main idea is to discover all the search paths which contain a node labeled with v. The desired views can be found at the end of these search paths. To improve the performance, we utilize the properties of the DMVI. As mentioned in Section 3.1, if the internal node n for v appears as at the first level (i.e., a child of the root), all v's can be found in the subtree rooted at n or the subtrees rooted at the left brothers of n. Therefore, the branches for the right brothers are pruned.

In the second stage, the search path for each discovered v' in the first stage is rebuilt. The old search path for v' is removed first. The removing process is similar to the one for deleting the search path of a removed view in the DMVI, as described above. Then, the query expression *qe* of v' is rewritten by merging it with the query expression of v after the occurrence(s) of v is removed, and the domain T of v' is updated by replacing v in T with the domain tables of v. After that, the domain tables in T are sorted using the two-level priority rule, and v' with the updated T is inserted back to the DMVI.

Let us consider the following example. Assume that the domain T_1 of a view v_1 is: $\{ec_1, ec_2\}$; the query expression of v_1 is: *select* a_1 *from* ec_1, ec_2 *where* $ec_1.a_1 = ec_2.a_2$; the domain T_2 of a view v_2 is: $\{ec_3, v_1\}$; the query expression of v_2 is: *select* a_3 *from* v_1, ec_3 *where* $v_1.a_1 = ec_3.a_3$. In this example, v_2 uses v_1 in its domain. Hence, if v_1 is removed, the search path for v_2 in the DMVI becomes invalid and has to be rebuilt. The old search path for v_2 is removed first. Based on T_1 and the query expression of v_1, T_2 is changed to $\{ec_1, ec_2, ec_3\}$ and the query expression of v_2 is rewritten to: *select* a_3 *from* ec_1, ec_2, ec_3 *where* $ec_1.a_1 = ec_2.a_2$ *and* $ec_1.a_1 = ec_3.a_3$. After that, the new query expression is saved and the domain tables in T_2 are sorted and used to build a new search path for v_2.

4 Experiments

In this section, we report the results of our simulation experiments to demonstrate the efficiency of our technique. The experiment programs are implemented in Matlab 2010 on a PC with Intel® dual core (1.5 GHz) *CPU* and 4 *GB* memory running the Windows ® 7 operating system.

100 random progressive queries were generated for our experiments. The starting and ending times for each PQ and the execution time for each SQ were recorded. The maximum number of PQs allowed to be executed simultaneously in the system was set to 10. Each PQ was formulated by several SQs, where the step number was randomly chosen from 2 to 20. 50 random external tables were used as domain tables by SQs. Each external table consisted of several attributes, where the attribute number was randomly chosen from 2 to 10. Each attribute was assigned with a random range, where the number of subranges for the

select bitmap was also randomly chosen from 1 to 5. For each SQ sq, a corresponding TMV was created when sq was completed and removed when the corresponding PQ of sq was completed. When a PQ was completed, the results of its SQs had a chance (probability) to become CMVs. Both TMVs and CMVs were also used as domain tables by SQs.

Each SQ sq was generated in two steps. First, the domain of sq was determined. The domain of sq contains one or more domain tables, where the domain size is randomly chosen between 1 and 5. Each domain table of sq could be either an external table or a materialized view (TMV or CMV). The probabilities to choose an external table and a materialized view were different in our experiments. We assumed that users preferred to choose previous SQ results (i.e., TMVs and CMVs) over external tables for their new SQs if possible. Hence, CMVs and TMVs were assigned a larger probability to be chosen. Second, the query expression of sq was built. According to the domain T of sq, attributes were randomly chosen from the domain tables in T to determine the project operations (target attributes), select operations (attributes whose ranges were restricted), and join operations (pairs of joining attributes).

In our experiments, the DMVI started with a single root node. When a materialized view v (TMV or CMV) was determined, its domain tables were sorted by the two-level priority rule, and a search path was created for v in the DMVI. At the end of the path, the bitmaps of v was saved in a leaf node. Before each SQ sq was executed, the DMVI was searched and the desirable views (both the domains and the bitmaps matched) were returned.

In the first experiment, the performance of the DMVI based view searching technique (DMVIT) was compared with that of the sequential scan based view searching technique (SST). The main idea of the SST is as follows. The views are picked from the VS one by one sequentially. Each view is compared with the given SQ sq. If the view v contains the result of sq, then v is considered as a candidate view. After all the views are checked, a best view is chosen from the candidate views to answer sq. In contrast to the SST, our technique first uses the DMVI to filter out the views that do not share the same domains with the given SQ sq. It then prunes the views whose bitmaps do not match the ones for sq. After that, our technique processes the desirable views in the same way as the SST, i.e., the view matching technique is applied to examine each view against sq to find the candidate views and then the best view for answering sq. Figure 4 shows the number of view matching comparisons saved by the DMVIT over the SST. The X-axis represents the total number of SQs in the test, and the Y-axis represents the saved number of view matching comparisons. From the figure, we can see that as the number of SQs increases, the view matching cost saving is increasingly larger, which demonstrates the efficiency of our proposed technique.

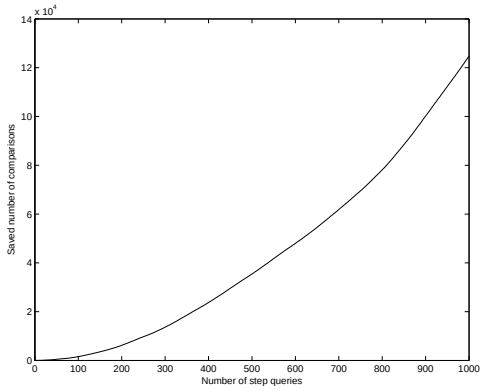

Figure 4: Saved view matching comparisons between DMVIT and SST

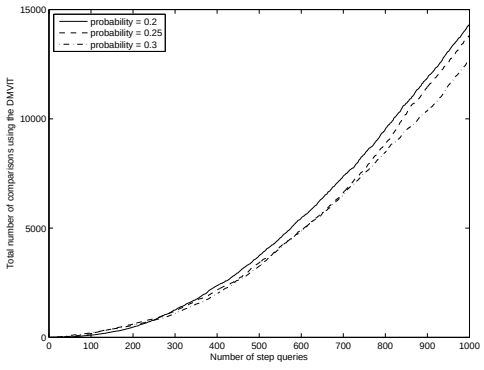

Figure 5: The performance of the DMVIT with various probabilities of selecting CMVs

The second experiment was to examine the effect of various probabilities of choosing the

results of SQs as CMVs on the performance of the DMVIT and SST. As mentioned earlier, TMVs are removed after their corresponding PQs are completed. Thus, the size of the VS is mainly determined by the number of CMVs kept in the system. A higher probability of choosing CMVs usually leads to a larger VS. Therefore, we wanted to see the performance of the DMVIT and SST for different sizes of the VS. The view matching cost comparisons for the DMVIT with various probabilities are shown in Figure 5, and the view matching cost comparisons for the SSTs are shown in Figure 6. From Figure 5, we observe that the three performance curves are very close to each other, which implies that the view matching costs by using the DMVIT with various probabilities (i.e., VS sizes) are nearly the same. However, from Figure 6, we can see significant differences among the three performance curves. The reason for this phenomenon is as follows. Using the SST, the number of views that need to be compared grows quickly with the size of the VS. From this experiment, we can see that the performance of our technique is stable relative to the size of the VS.

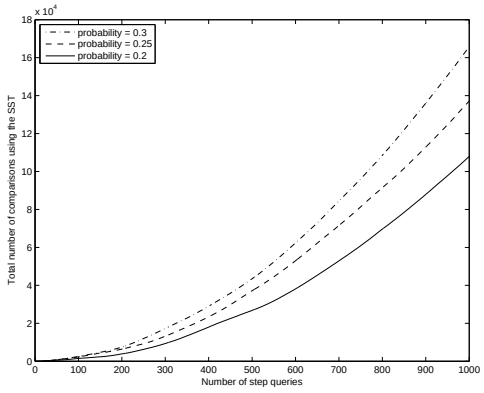

Figure 6: The performance of the SSTs with various probabilities of selecting CMVs

In the third experiment, the view deletion cost for the two-level priority ordering based DMVI constructing technique (TPDMVI) and the no-priority ordering DMVI constructing technique (NPDMVI) were compared. The main idea of the TPDMVI is as follows. When a view v needs to be inserted into the DMVI, the domains of v are sorted by the two-level priority rule. After that, each domain table is picked up in turn and inserted into the DMVI. Furthermore, if a node has multiple child nodes, the order of its child nodes is determined by the two-level priority order of the domain tables in the workload. The NPDMVI, on the other hand, assigns no priority to any domain table, which causes the DMVI to be created randomly. In our technique, when a view v becomes invalid, the system has to discover all the views which uses v in their domains. Using the TPDMVI, if v appears as a first level node of the DMVI, the searching process only occurs on the subtree of v and the subtrees of the left brothers of v (if any). However, using the NPDMVI, the entire tree of the DMVI always needs to be traversed. The performance of searching the views with invalid search paths was compared between the TPDMVI and the NPDMVI and the results are shown in Figure 7, where X-axis represents the number of SQs in the test and Y-axis represents the total number of nodes visited in the DMVI. In this experiment, to make the figure clearer, we set the upper bound for the number of SQs to 200. From the figure, we can see that the view deletion cost for the TPDMVI is significantly smaller than that for the NPDMVI. Note that the view deleting process only happens after a PQ is completed.

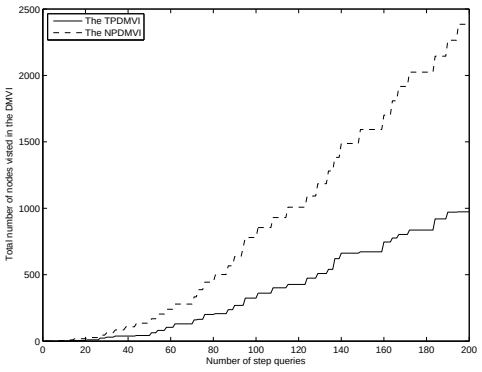

Figure 7: The deletion cost comparisons between TPDMVI and NPDMVI

In the last experiment, the effectiveness of the DMVIT was compared with that of the SST. As mentioned earlier, the DMVIT filters out the views whose domains are different from that of the given SQ. However, some views that are filtered out by the DMVIT may be

usable for answering the given SQ. The main purpose of this experiment is to find out that what percentage of usable views can be discovered by the DMVIT. We utilized the view matching mechanism (i.e., materialized query table matching) of DB2 to conduct the experiment and calculated the percentages of usable views discovered by the DMVIT and the SST for the tested cases. The probability of selecting CMVs was set to 0.25. The experiment results we obtained are as follows. The percentages of usable views found by the SST and the DMVIT were 100% and 83%, respectively; while the numbers of views checked by the SST and the DMVIT were 163546 and14648, respectively. From the experiment, we can see that, comparing to the SST, the DMVIT dramatically reduces the number of checked views while keeping a high hitting rate for its discovered usable views.

5 Conclusion

The rapid growth of data intensive applications leads to an increasing demand to efficiently process progressive queries (PQ). A materialized view based approach to processing PQs was proposed in our previous work. However, how to efficiently discover usable materialized views to answer a given SQ in a PQ was challenging and remained open.

In this paper, we have presented a new index technique, called the dynamic materialized view index (DMVI), to efficiently discover usable views for a given SQ from the available views in the view storage. Domain table based search paths are dynamically created in a tree structure of the DMVI for the arriving new views. A two-level priority ordering is adopted to achieve efficient construction and maintenance of the DMVI tree for a dynamically changing view set. Encoding methods to generate bitmaps for target attributes, selection condition and join condition are suggested. The relevant bitmaps are stored at the leaf nodes of the DMVI tree together with other view information to support a refined pruning for undesirable views. Since matching a view with a given query is computationally expensive, using the DMVI to efficiently discover usable views for the query can improve the performance of view based query processing.

Our experimental results demonstrate that our DMVI is quite promising in reducing the view matching cost and the proposed tree structure supports efficient view management for a dynamic view set.

Our future work includes extending the method for handling aggregate queries and evaluating the relevant techniques in real database management systems.

About the Authors

Chao Zhu is a PhD student in the Department of Computer and Information Science at The University of Michigan, Dearborn, USA. He is a graduate research assistant with an IBM CAS fellowship. His research interests include query processing and optimization, data mining, and Web services.

Qiang Zhu is a Professor in the Department of Computer and Information Science at The University of Michigan, Dearborn, MI, USA. He received his Ph.D. in Computer Science from the University of Waterloo in 1995. Dr. Zhu is a principal investigator for a number of database research projects funded by highly competitive sources including NSF and IBM. He has numerous research publications in various top journals and conference proceedings in the database field including TODS, TOIS, VLDBJ and VLDB. Some of his research results have been included in several well-known database research/text books. Dr. Zhu served as a program/organizing committee member for numerous international conferences and an editor-in-chief/associate-editor for a number of international journals. His current research interests include query optimization, data stream processing, multidimensional indexing, self-managing databases, Web information systems, and data mining.

Calisto Zuzarte is a senior technical manager at the IBM Canada Software Laboratory. He has been involved in several projects leading and implementing many features related to the IBM® DB2® SQL compiler. His main expertise is in the area of query optimization including cost-based optimizer technology and automatic query rewriting for performance. Calisto

is also a research staff member at the IBM Center for Advanced Studies (CAS).

Wenbin Ma is a Software Engineer at the IBM Canada Laboratory. He received his first master degree in Software Engineering from the Bei Hang of P. R. China and his second master degree in Algorithms and Theory from the University of Alberta. He works in the DB2 SQL compiler team. His main expertise is in the areas of query rewrite and MQT technology.

Trademarks

References

[1] Agrawal, S., S. Chaudhuri and V. Narasayya: Automated selection of materialized views and indexes in SQL databases. *Proc.of VLDB Conf.*, pp. 391-398, 2000.

[2] Aouiche, K. and J. Darmont: Data mining-based materialized view and index selection in data warehouses. *J. Intell. Inf. Syst.*, 33(1):65-93, 2009.

[3] Bellatreche, L., K. Karlapalem and Q. Li: Evaluation of Materialized View Indexing in Data Warehousing Environments. *Proc.of DaWaK Conf.*, pp. 5766, 2000.

[4] Calvanese, D., G. D. Giacomo, M. Lenserini and M. Y. Vardi: View-based query process: on the relationship between rewriting, answering and losslessness. *Theor.Comput.Sci.*, 371(3): 169-182, 2007.

[5] Comer, D.: The ubiquitous B-tree. *ACM Computing Survey*, 11(2): 121-137, 1979.

[6] Gou, G., M. Kormilitsin and R. Chirkova: Query evaluation using overlapping views: completeness and efficiency. *Proc.of SIGMOD Conf.*, pp. 37-48, 2006.

[7] Graefe, G. and M. J. Zwilling: Transaction support for indexed views. *Proc.of SIGMOD Conf.*, 2004.

[8] Halevy, A. Y.: Answering queries using views: a survey. *The VLDB Journal*, 10(4): 270-294, 2001

[9] Himanshu, G. and I. S. Mumick: Selection of Views to Materialize in a Data Warehouse. *IEEE Transaction on Knowledge and Data Engineering*, 17(1): 24-43, 2005.

[10] Kimura, H., G. Huo, A. Rasin, S. Madden and S. B. Zdonik: CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. *PVLDB*, 3(1): 1103-1113, 2010.

[11] Kuno, H. A. and G. Graefe: Deferred Maintenance of Indexes and of Materialized Views. *Proc. of DNIS Conf.*, pp. 312-323, 2011.

[12] Larson, P.-Å. and H. Z. Yang: Computing queries from derived relations. *Proc. of VLDB Conf.*, pp. 259-269, 1985.

[13] Larson, P.-Å. and J. Zhou: View matching for outer-join views. *VLDB Journal*, pp. 29-53, 2007.

[14] Lehner, W., R. J. Cochrane, H. Pirahesh and M. Zaharioudakis: Fast Refesh using Mass Query Optimization. *Proc.of ICDE Conf.*, pp. 391-398, 2001.

[15] Liu, Z. and Y. Chen: Answering Keyword Queries on XML Using Materialized Views. *Proc. of ICDE Conf.*, pp. 1501-1503, 2008.

[16] Park, C.-S., M.-H. Kim and Y.-J. Lee: Rewriting OLAP Queries Using Materialized Views and Dimension Hierarchies in Data Warehouses. *Proc. of ICDE Conf.*, pp. 515-523, 2001.

[17] Pottinger, R. and A. Levy: A Scalable Algorithm for Answering Queries Using Views. *Proc. of VLDB Conf.*, pp. 484-495, 2000.

[18] Roussopoulos, N.: View indexing in relational databases. *ACM Trans. on Database Systems*,7(2):258-290, 1982.

[19] Roy, P., S. Sudarshan, K. Ramamrithaml: Materialized View Selection and Maintenance Using MultiQuery Optimization Hoshi Mistry. *Proc. of SIGMOD Conf.*, pp.307-318, 2001.

[20] Srivastava, D., S. Dar, H.V. Jagadish, A. Levy: Answering Queries with Aggregation Using Views. *Proc. of VLDB Conf.*, pp. 318-329, 1996.

[21] Talebi, Z. A., R. Chirkova, Y. Fathi and M. Stallmann: Exact and inexact methods for selecting views and indexes for OLAP performance improvement. *Proc. of EDBT Conf.*, pp. 311-322, 2008.

[22] Xu, W. and Z. M. Ozsoyoglu: Rewriting XPath Queries Using Materialized Views. *Proc. of VLDB Conf.*, pp. 121-132, 2005.

[23] Zhu, C., Q. Zhu and C. Zuzarte: Efficient Processing of Monotonic Linear Progressive Queries via Dynamic Materialized Views. *Proc.of CASCON Conf.*, pp. 224 - 237, 2010.

[24] Zhu, C., Q. Zhu, C. Zuzarte and W. Ma: A Materialized-View Based Technique to Optimize Progressive Queries via Dependency Analysis. *Proc.of CASCON Conf.*, 2011.

[25] Zhu, Q., B. Medjahed, A. Sharma and H. Huang: The Collective index: A Technique for Efficient Processing of Progressive Queries. *The Computer Journal*, 51(6): 662-676, 2008.