J Inf Process Syst, Vol.9, No.4, pp.511-537, December 2013
http://dx.doi.org/10.3745/JIPS.2013.9.4.511

pISSN 1976-913X
eISSN 2092-805X

Developing a Dynamic Materialized View Index for Efficiently Discovering Usable Views for Progressive Queries

Chao Zhu*, Qiang Zhu*, Calisto Zuzarte**, and Wenbin Ma**

Abstract—Numerous data intensive applications demand the efficient processing of a new type of query, which is called a progressive query (PQ). A PQ consists of a set of unpredictable but inter-related step-queries (SQ) that are specified by its user in a sequence of steps. A conventional DBMS was not designed to efficiently process such PQs. In our earlier work, we introduced a materialized view based approach for efficiently processing PQs, where the focus was on selecting promising views for materialization. The problem of how to efficiently find usable views from the materialized set in order to answer the SQs for a PQ remains open. In this paper, we present a new index technique, called the Dynamic Materialized View Index (DMVI), to rapidly discover usable views for answering a given SQ. The structure of the proposed index is a special ordered tree where the SQ domain tables are used as search keys and some bitmaps are kept at the leaf nodes for refined filtering. A two-level priority rule is adopted to order domain tables in the tree, which facilitates the efficient maintenance of the tree by taking into account the dynamic characteristics of various types of materialized views for PQs. The bitmap encoding methods and the strategies/algorithms to construct, search, and maintain the DMVI are suggested. The extensive experimental results demonstrate that our index technique is quite promising in improving the performance of the materialized view based query processing approach for PQs

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

1. INTRODUCTION

The rapid growth of numerous data intensive applications (e.g., astronomy, biology, and social media) has led to significant research being focused the problem of analyzing a large amount of data in databases. In such data intensive applications, a new type of query, which is called a progressive query (PQ), is required [49]. Unlike a conventional query, a progressive query is defined as a query that is formulated in more than one step (progressively), where each step is called a step-query (SQ) [49]. A user submits his/her first SQ sq_1 on one or more external (existing) tables in the database. Based on the result R_1 of sq_1, the user submits a second SQ sq_2

※ The preliminary results of this work were presented at the 2012 Conference of the IBM Centre for Advanced Studies on Collaborative Research -- CASCON'12 [48] in Toronto between Nov. 5-7, 2012. Research was partially supported by the IBM Canada Software Lab and The University of Michigan.
Manuscript received February 26, 2013; first revision July 26, 2013; accepted September 3, 2013.
Corresponding Author: Chao Zhu (zhuchaon1@gmail.com)
* Dept. of Computer and Information Science, the University of Michigan, Dearborn, MI 48128, USA (zhuchaon1@gmail.com, qzhu@umich.edu)
** IBM Canada Software Lab, Markham, Ontario, Canada (calisto@ca.ibm.com, wenbinm@ca.ibm.com)

using R_1 and/or additional external tables as its input. In general, an SQ may use the result(s) of its previous SQ(s) and/or external tables as its input. After the last SQ is submitted, a PQ is completed. Hence, the user gradually approaches his/her desired result by issuing a number of related SQs to the database. The relationships among the SQs and referenced external tables for a PQ can be depicted as a dependency graph (DG) [49]. Based on the structures of their DGs, PQs can be classified as single-input linear, multiple-input linear, and non-linear ones.

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 of the cars that are available on a website (i.e., SQ 1). However, there are too many results that show up. Hence, in the second step, he/she adds a search condition to only show cars that were manufactured in the USA (i.e., SQ 2). After analyzing the result, he/she selects 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 (i.e., SQ 3).

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 the previous SQs of each in-process PQ have to be kept in the system (as one type of [temporary] materialized view) until the PQ is completed. On the other hand, it is desirable to retain some popular results (i.e., those that are frequently utilized) for SQs in the system (as another type of [critical] materialized view) 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 for processing PQs in [46]. A benefit estimation model was developed to determine if the result table for the 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/set (VS).

In [46], we mainly focused on discussing how to select promising materialized views for processing PQs (i.e., studying the view materialization selection problem). However, as more and more materialized views are selected and saved in the VS, how to efficiently discover and use the relevant materialized views from the VS for answering the SQs of a PQ becomes an important issue. Since view matching (searching) is a time consuming task, if the saved materialized views are not properly managed (e.g., each view in the VS is examined sequentially to match the currently executing SQ) the view matching/searching cost is very expensive and the performance of the PQ seriously suffers. Therefore, a new PQ oriented materialized view managing technique is required. Our target is to develop a view access method to efficiently discover usable views for answering SQs.

In this paper, we present a new index technique, which is called the dynamic materialized view index (DMVI), to index the materialized views in the VS and we used this index to efficiently discover/search for usable materialized views for answering a PQ (i.e., solving the view searching problem). When a new view v is added to the VS, a search path for v is created in the DMVI and the relevant information (address, query expression, bitmaps, etc.) of v is stored at the end (leaf) of the path. Many views may share the same search path. When an SQ sq arrives, the system goes through the proper search path to discover usable views for answering sq. In addition, a bitmap based method is applied at the leaf of the path in the DMVI for further pruning unusable views. By using the DMVI, the views in the VS can be efficiently managed and searched for answering the SQs of the PQs. Algorithms and strategies for constructing the DMVI,

maintaining the DMVI, and searching for desirable views by using the DMVI that are presented. As demonstrated in our experimental results, the DMVI can be utilized to efficiently discover usable views for answering SQs. Since checking if a view matches a given query is computationally expensive, filtering out undesirable views from consideration for view matching can significantly improve the overall query performance.

Many index techniques, which are used to efficiently access data objects in the database, are reported in the relevant literature. Robinson presented a dynamic index, called the K-D-B tree, to retrieve multi-key records via range queries [30]. Guttman *et al.* described a dynamic index structure, called the R-tree, to efficiently handle spatial data [13]. Berchtold, Katayama, Chakrabarti, Sakurai, *et al.* proposed index structures to access high dimensional data sets [4,19,6,34]. Kuo *et al.* proposed methodologies to control the access of B-tree indexed data in a batch and in real-time fashion [22]. Chan *et al.* presented the RE-tree, which is an index structure for large databases of Regular Expression (RE) specifications [7]. Wang, Jiang, *et al.* proposed index structures for searching XML documents [18,41,40]. He *et al.* introduced an index structure, called a Closure-tree, to support subgraph queries and similarity queries [15]. Zhang *et al.* proposed the Bed tree, which is a B+-tree based index structure, for doing string similarity searches [44]. Although how to efficiently access data objects in different types of databases have been well studied in the relevant literature, no index was designed to find usable materialized views for answering SQs.

A bitmap index is a special type of index that uses bitmaps and that answers queries by performing bitwise logical operations on these bitmaps. Bitmap indexes are known as the most effective indexing methods for conducting range queries on append-only data. Many different bitmap indexes have been proposed in the relevant literature [35,27,42,36,16,43,10,8]. Chan *et al.* presented a general framework to study the design space of bitmap indexes for selection queries and for examining the disk-space and time characteristics that the various alternative index choices offer [8]. Nitsos *et al.* reported a hybrid-indexing scheme (Bitmap-Tree) that integrates the advantages of bitmap indexing and file inversion to improve the efficiency of query processing and to reduce the storage overhead [27]. Yoon *et al.* proposed a bitmap-indexing scheme for speeding up the access control for the XML documents [43]. Sinha *et al.* proposed a multi-resolution, parallelizable bitmap index, which supports a fine-grained trade-off between storage requirements and query performance [36]. Sinha *et al.* introduced adaptive bitmap indexes, which conform to space limits while dynamically adapting to the query load and these indexes provide excellent performance [35]. He *et al.* developed a bitmap pruning strategy for processing the iceberg query, which is a special type of aggregation query that computes aggregate values above a user-provided threshold [16]. Fusco *et al.* proposed a compressed bitmap index approach that significantly reduces both CPU load and disk consumption [10]. The differences between the conventional bitmap indexes and our bitmap-based technique are as follows: first, a conventional bitmap index is applied to answer queries, while our bitmap-based approach is used to filter undesirable views. Second, our bitmap-based method is simpler than a conventional bitmap index since our method assigns only 1-bit in a bitmap for an attribute in most cases, while a conventional bitmap index usually builds one bitmap vector for each attribute.

The view materialization is another important type of technique for improving query performance. There is a substantial body of work that explores the index techniques and the view materialization techniques together. Roussopoulos [31] presented a method to select a set of

views and to 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 that are used to construct the view. Kimura *et al.* [20] 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. They did so by taking into consideration 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 on dimension and fact base tables. Talebi *et al.* [38] 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 materialized view candidates and indexes in data warehouse environments. Graefe and Zwilling [12] studied techniques for providing transaction support for indexed summary views. Kuno and Graefe [21] proposed a deferred technique to maintain indexes and materialized views. However, all of 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 the materialized views. In contrast, the index technique we introduce in this paper directly uses materialized views, instead of the underlying data, as indexed objects. It does so with the goal of removing as many undesirable views as possible from consideration for view matching during the materialized view based query processing.

The work that is most related to this paper in the literature is a so-called collective index method introduced by Zhu *et al.* [49]. It is the only existing index technique that was 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 the input table of a 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 the materialized views that are directly used as indexed objects, which is different from our index in this paper.

Other related work includes one study on how to apply materialized views to optimize a special type of PQ, which is called a monotonic PQ, in [45,47] and another study on how to apply materialized views to optimize the generic PQs in [46]. However, the focus of these earlier works was on how to select which materialized views to keep in the system, while the focus of this work is on how to efficiently find/search for the desirable materialized views that are available in the system for answering an SQ in a given PQ. Our indexing technique can be used in conjunction with existing view matching techniques [5,14,23,24,26,28,37,39] 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. The experimental results are reported in Section 4 and 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 [46] 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) from the previous SQ(s) is/are 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. We call these the temporary materialized views (TMVs).

Usually, after a PQ is completed, all of the TMVs for its SQs are removed from the system. However, it is possible that the result tables of some of the SQs of a completed PQ are popular (e.g., they are frequently used by the SQs of other PQs). In such cases, 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 view, which we call critical materialized views (CMVs). Generally, the lifetime of a TMV is shorter than a CMV and the TMV is removed after its corresponding PQ is completed. However, the CMV 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 in memory or disk, 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. The TMVs and CMVs are differentiated by their view identifiers, which are stored in our new index. Figure 1 shows the structure of the view storage.

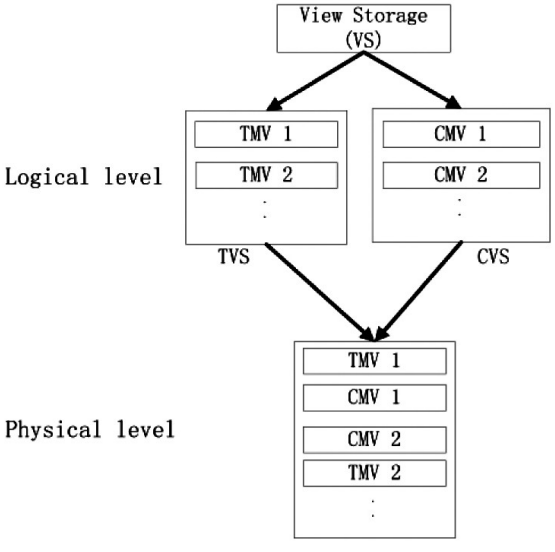

Fig. 1. The structure of the view storage

3. DYNAMIC MATERIALIZED VIEW INDEX

To efficiently find the usable views for answering SQs, we present a dynamic materialized view index (DMVI) in this section. The structure of the DMVI is introduced in Section 3.1. A bitmap matching technique, which is considered as part of the DMVI technique, is presented in Section 3.2. The DMVI construction issue is discussed in Section 3.3. The view search by using the DMVI is described in Section 3.4, and the view maintenance issue for the DMVI is discussed in Section 3.5.

3.1 Index structure

In this work, we want to develop an efficient method to search for possibly usable views (TMVs or CMVs) in the VS to answer the given SQs. A straightforward way to do this is to apply a sequential scan on the VS. Each view in the VS is checked to see if it is a usable view for answering 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., checking if the former can be used to answer the latter) is computationally expensive. Hence, developing an efficient view access method to rapidly identify usable views for answering SQs is crucial in achieving efficient optimization for PQs.

In this paper, we develop a dynamic materialized view index (DMVI) to efficiently find the views that are possibly usable for answering the SQs. However, the special characteristics of the materialized views for PQs raise some new challenges. The first challenge is that all the materialized views are dynamically generated while the PQs are being processed. Therefore, the DMVI has to be dynamically updated to access 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 DMVI has to be efficiently maintained in accord with the VS.

The main idea of the DMVI is to dynamically build an index for all of the materialized views in the VS. For each view v in the VS, 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 usable views. We call the set of all the input tables of an SQ (view) the domain of the SQ (view). Hence, we also call an input table a "domain table". The criterion used to search the DMVI is that the domain of a usable 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 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, the DMVI does not guarantee that the views returned from the equal-domain criterion are always usable for answering the given SQ. Hence, a refined checking (bitmap based method) of the usability of the returned views is required. Therefore, the DMVI is an approximate index with the objective to return set S of materialized views for a given SQ such that (1) the views in S match the given SQ as closely as possible; and (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 one or multiple materialized views, which

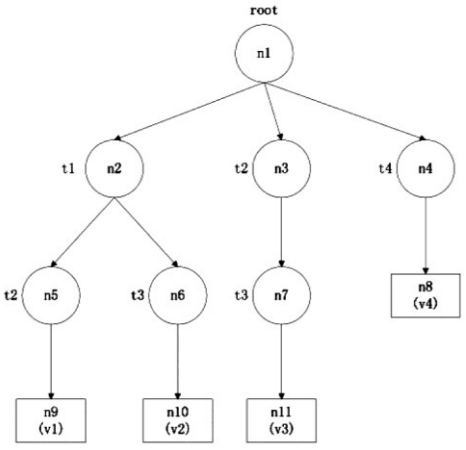

Fig. 2. An example of the DMVI

share the same search path in the DMVI. Each internal node n (except the root) represents a domain table t. In other words, n is associated with the identifier of the domain table t. For any view v in a leaf node whose search path contains n, its domain must contain t. 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 of the domain (input) tables for v. Note that, for simplicity, we will use a domain table and a domain table identifier interchangeably in our discussion. 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. For example, the domains of v_1 and v_2 are $\{t_1, t_2\}$ and $\{t_1, t_3\}$, respectively. t_1 and t_2 are used as a search key for v_1 in the DMVI.

As mentioned earlier, the first challenge in creating an index for the views in PQ processing is that all the materialized views are dynamically generated. To tackle this challenge, the DMVI must support a mechanism 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, and 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, and 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 a priority order for the nodes for insertions in the DMVI.

Our priority order is given as follows: for two internal nodes n and m in the DMVI that share the same direct parent node, if n is on the left to m in the DMVI, n is assigned with a higher priority than m for building search paths for views. If the domain of a view v contains two tables labeled by n and m, and n has a higher priority than m, then, n rather than m is selected as the next node on the search path of v.

More specifically, suppose we want to index a new materialized view v in the DMVI. Assume that node ni is either the root or the current chosen internal node for building the search path of v in the DMVI, and that ni has m ordered (from the left to the right) children (internal nodes): $n_1, n_2, \ldots, n_m$. The domain tables represented by these internal nodes are $t_1, t_2, \ldots, t_m$, respectively. The criterion to decide the next internal node for building the search path for v is

given by the following rules:

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, one of the unlabeled domain table(s) t is selected (the order for such a selection is to be discussed in Section 3.3) and a new internal node representing t is created as a child node of ni. If all domain tables of v have been labeled on the search path of v, a leaf node is created or chosen (if already exists) at the end of the path.

In the previous example, the two candidate nodes that are considered are: n_2 and n_4. n_2 is on the left to n_4. Hence, n_2, rather than n_4, is chosen to build the search path of v_5. n_3 is excluded because table t_1 , which is represented by n_3 is not in the domain of v_5.

From the rule, we can 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 the domain table t, then all of the indexed views whose domains contain t can be found in the subtree that is rooted at n.

(2) At the first level of the tree, if the internal node n, which is not the leftmost node is labeled with the domain table t, then all of the indexed views whose domains contain t can be found in the subtree that is rooted at n or in the subtrees that are rooted at the nodes on the left of n. Note that the internal node that has t as a label must be at a higher level in the latter case.

In general, given that an internal node $n[m]$ is labeled with a domain table t at the m-th level of the tree, if the 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 node on the left to $n[m]$, 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 node on the right to $n[m]$, the node with t can only be found at a level lower (〈) than m.

The second challenge of creating an index for views in the VS is the high complexity of the view maintenance in the VS. As we mentioned earlier, the DMVI has to be efficiently maintained in accord with the VS.

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 about one or more views. The information about a view includes the name of the view and a view indicator to differentiate between the types of views (TMV or CMV), etc. We will discuss the details of the view information structure in Section 3.2. When a view transformation occurs, the view itself and all if its related information is kept unchanged, except for the view type indicator.

On the other hand, it is required to safely and efficiently remove the search paths in the DMVI, which are associated with invalid TMVs or CMVs. Furthermore, since in general, an SQ can be formulated using 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) as inputs, the TMVs and CMVs can also be used as the domain tables of an SQ besides the external tables. This

implies that CMVs and TMVs can appear in the search keys for the views indexed in the DMVI. Therefore, how to adapt search paths in the DMVI that contain invalid TMVs or CMVs is also an important issue. We will discuss the details of the DMVI maintenance in Section 3.5.

3.2 View bitmap-based matching in the DMVI

The main purpose of introducing the DMVI is to efficiently find usable views for answering the SQs. Using the structure of the DMVI introduced in Section 3.1, the system filters out unusable views in the VS and only returns the views that share the same domain as the SQ that is 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 adopted an efficient refined filtering method, 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 of the DMVI stores the information of a view. The information includes: the view name/identifier, , the type of view (indicator), the query expression of the view, the view bitmaps, and the location of the view. Hence, the view bitmaps can be accessed in the leaf nodes of the tree. As we will demonstrate later on, the bitmap encoding method depends on the domain of a query expression (for a view or an SQ). In other words, the bitmap encoding method is the same for those query expressions that share the same domain. When an SQ sq arrives, the system creates the bitmaps for the query expression of sq using a certain bitmap encoding method. Next, the index discussed in Section 3.1 is searched to find all the leaf nodes whose associated views share the same domain with sq. For each view in the returned set, its bitmaps are compared with those for sq. If the bitmaps for a view do not match with the view for sq, the view is filtered out. Note that our bitmap matching is different from conventional view matching. As we will demonstrate later on, 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, by 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), a bitmap encoding method is required. As mentioned above, our encoding method depends on the domain of the query expression. Specifically, the bitmap encoding method creates three bitmaps: one for each operation (i.e., project, select and join) of the query expression. Given the domain T (consisting of input tables), its encoding method is described as laid out below.

(1) **The project bitmap:** it is the bitmap for the project operation (π) of the query expression.

For each domain table t in T, the project bitmap assigns 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: $\pi_{a1,a2}(t)$, 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 project bitmap for this query is 1100.

(2) **The select bitmap:** it is the bitmap for the select operation (σ) of the query expression.

For each domain table t in T, each attribute of t is analyzed and its value range is divided into n subranges. n can be 1 if the range of a is difficult to divide. For example, it is difficult to divide the range of an attribute a_1 that represents the title of a paper in a paper table. A bit segment that contains n bits is assigned for a, one bit for each subrange of a.

If the range of a is 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 bitmap 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 for a_1. Assume that the given query expression is: $\sigma_{a1>70}(t)$. In the bit segment for a_1, the bits for the subranges with at least one value that satisfies 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 of the bits of the bit segment of a are set to 1. In other words, we took a conservative approach by keeping all of the subranges. The select bitmap consists of all the bit segments for the attributes of the domain tables in T.

(3) **The join bitmap:** it is the bitmap for the join operation ($\bowtie$) of the query expression.

To generate a bitmap for a join operation, all of the possible attribute pairs that can be used for a join operation are discovered from the domain tables in T first. This type of 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 =, $\langle$, $\rangle$), 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 the 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 that do not have containment relationship with the SQ, namely, that the views do not contain the result of the SQ.

Assume that the query expressions of v and sq are encoded using the same bitmap 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 $\mathbf{pbm}_1$ for v and the project bitmap $\mathbf{pbm}_2$ 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 $\mathbf{pbm}_1$ is 0 but in $\mathbf{pbm}_2$ 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 $\mathbf{pbm}_1$ and $\mathbf{pbm}_2$, the system performs a bitwise complement on $\mathbf{pbm}_2$ first and then a bitwise OR on $\mathbf{pbm}_1$ and $\mathbf{pbm}_2$. If the result contains 0, it means v cannot contain the result of sq. For example, if $\mathbf{pbm}_1$ is 00111, $\mathbf{pbm}_2$ is 10011. First, a bitwise complement is performed on $\mathbf{pbm}_2$ to get 01100. Then, a bitwise OR is applied to $\mathbf{pbm}_1$ and $\mathbf{pbm}_2$, 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 $\mathbf{sbm}_1$ for v that has passed the first stage test and the

select bitmap $\mathbf{sbm}_2$ for sq are compared. If the bit segment for an attribute a in $\mathbf{sbm}_1$ indicates a narrower range (i.e., missing some 1's) than the bit segment for a in $\mathbf{sbm}_2$, 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, some rows may exist 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 $\mathbf{sbm}_2$ first, then a bitwise OR is applied to $\mathbf{sbm}_1$ and $\mathbf{sbm}_2$. If the result contains 0, it implies that 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, $\mathbf{sbm}_1$ is 00011 and $\mathbf{sbm}_2$ is 00001. First, a bitwise complement is performed on $\mathbf{sbm}_2$, resulting in 11110. Then, a bitwise OR is applied to $\mathbf{sbm}_1$ and $\mathbf{sbm}_2$, 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 $\mathbf{jbm}_1$ for v, which has passed the second stage test and the join bitmap $\mathbf{jbm}_2$ for sq are compared. Each bit in the join bitmap indicates the occurrence of a pair of join attributes under 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 conduct. To reduce the view matching cost, we excluded such views from consideration. Hence, we require $\mathbf{jbm}_1$ and $\mathbf{jbm}_2$ 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 the views that are clearly unusable for answering the given SQ or that are very difficult to use for performing view matching. All of which results in a small set of candidate views. The candidate views are then further examined (i.e., perform view comparison) to see if they are indeed usable for answering the given SQ.

Let us consider the following example: assume that there are two tables $T1\ (t1\_id, name, age)$ and $T2\ (t2\_id, annual\_salary)$. The bitmap encoding method for the domain $\{T1, T2\}$ is defined as follows: the project bitmap contains five bits that correspond to five attributes in $T1$ and $T2$: $t1\_id$, $name$, age, $t2\_id$, and $annual\_salary$. Since only the ranges of age and $annual\_salary$ can be easily divided, the age range is divided into five subranges: (0, 30), (30, 60), (60, 90), (90, 120), (120, 150), and the range of $annual\_salary$ is also divided into five subranges: (0, 50000), (50000, 100000), (100000, 500000), (500000, 1000000), (1000000, ∞). Hence, the select bitmap contains thirteen bits: five for age, five for $annual\_salary$ and three for other attributes (i.e., one for each). The join bitmap contains only one bit that corresponds to the join pair ($t1\_id$, $t2\_id$).

Assume that we want to check if a view v can match an SQ sq. The query expressions of sq and v are shown as follows:

$$sq: \pi_{name}(\sigma_{age>60 \quad and \quad annual\_salary>5000}(T_1 \overset{\triangleright\triangleleft}{}_{t1\_id=t2\_id} T_2))$$

$$v: \pi_{name,age}(\sigma_{age>30}(T_1 \overset{\triangleright\triangleleft}{}_{t1\_id=t2\_id} T_2))$$

First of all, the query expressions of both sq and v are encoded and six bitmaps are generated: $ProjectBitmap\_sq$ is: 01000; $SelectBitmap\_sq$ is: 0011101111111; $JoinBitmap\_sq$ is:1; $ProjectBitmap\_v$ is: 01100; $SelectBitmap\_v$ is: 0111111111111; $JoinBitmap\_v$ is:1.

Next, the system performs a bitwise complement on $ProjectBitmap\_sq$ (10111) first and

then a bitwise OR on $ProjectBitmap\_sq$ and $ProjectBitmap\_v$. The result is 11111 and no 0 is contained. After that, the system applies a bitwise complement again on $SelectBitmap\_sq$ (1100010000000) and a bitwise OR on $SelectBitmap\_sq$ and $SelectBitmap\_v$. The result is also all 1s. Finally, the $JoinBitmap\_sq$ and $JoinBitmap\_v$ are compared and they are exactly the same. Therefore, v is considered to match sq and v is returned as one of the results.

3.3 The construction of the DMVI

In this section, we discuss the details of the construction of the DMVI. 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 the materialized view v, it 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 stores the information of v.

In Section 3.1, we defined a priority order for the existing internal nodes of the index tree to determine the unique search paths of new views. In this section, we discuss how to construct the DMVI as an ordered tree. We need two orderings for the domain tables, i.e., the order of the domain tables of new view 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 the domain table t is selected and that the internal node in, which represents t is inserted into the DMVI as a child node of n. n has one existing child node cn. How to insert in is ambiguous because in can appear on the left or the right of cn. In this case, the order of the domain tables for the entire workload is required. The policy we used is that the domain table with a higher priority appears on the left.

To solve the above two ordering issues, we assigned different priorities to different domain tables. A two-level priority rule was 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 type of table, an older (i.e., created earlier) 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 where we are 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. Assume that an older table has a smaller subscript index. To determine the order of the tables in T, the tables 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 of the domain tables of v are sorted by the two-level priority rule. The domain tables in the entire workload are also sorted by this rule and are saved in a workload list. The domain tables of v are picked up one at a time in the given order. For the domain table t1 that was chosen first, each child 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 element on the search path for v. Otherwise, a new internal node n_2, which represents t_1 is created. In this case, we

need to decide where to insert n_2 in relation to other existing first level nodes. 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 of their labeled domain tables in the workload list. Thus, the labeled domain table of each child node of the root is compared with that of n_2 one by one from the left to the right according to the order in the workload list. In this way, the system can find a unique place to insert n_2 in relation to other first level nodes. Next, the insertion process is recursively applied to incorporate other domain tables of v into the search path of v. After all the domain tables of v are labeled on the search path, a leaf node is created/chosen to save the information of v.

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

The algorithm recursively builds the search path for the new view v in the DMVI. If all domain tables of v have been picked out to build the search path of v (line 1), a leaf node $lnode$

is used (if $lnode$ exists) or created (if $lnode$ does not exist) at the end of the path to save the information of v (lines 2 - 6). Otherwise, a domain table $ftable$ from the domain table list of v is picked up, an internal node $inode$ that labels $ftable$ is used (if $inode$ exists) or created (if $inode$ does not exist), and $inode$ is added into the search path of v (lines 11, 13 - 16, 18 - 20). After that, the function invokes itself to insert the remaining domain tables of v into the search path (lines 12, 17, 21). Using the above algorithm, we can build the DMVI dynamically by inserting every new view when it becomes available.

Assume that the number of materialized views is N and that the maximum number of domain tables for each view is M. Usually, $M \ll N$. The worst-case time complexity to construct a DMVI for N materialized views is $O(MN + N(N+1)/2)$=$O(N^2)$.

3.4 View searching using the DMVI

Let us describe how to apply the DMVI to find the usable views in the VS for 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 bitmap encoding method is applied to generate the bitmaps for sq. The ordered tables in T are used as the search key to find the leaf node ln. For each view v in ln, the bitmaps are extracted and compared with those of sq. If the view is not filtered out by the three-stage bitmap matching, the view is returned. By As we will see in Section 4, by using the DMVI, the number of the candidate views that are used to perform the final direct view comparison for an SQ is significantly reduced. The details of the view searching algorithm are specified as listed below.

ALGORITHM 2: **SeachViews**($sq, dmvi$)
Input: (1) a new SQ sq; (2) the DMVI $dmvi$.
Output: a set of matched views.
Method:

```
1.  domain_sq = Domain of sq;
2.  sort domain tables in domain_sq using the two level priority rule;
3.  encode the query expression of sq using the bitmap encoding method for domain_sq;
4.  ProjectBitmap_sq = the project bitmap of sq;
5.  SelectBitmap_sq = the select bitmap of sq;
6.  JoinBitmap_sq = the join bitmap of sq;
7.  search dmvi using domain_sq as the search key;
8.  if no leaf node found then
9.    return ∅; /*return empty*/
 /*some views which share the same domain with sq are found.*/
10. else
11.   n = the reached leaf node;
12.   for each view v in n do
13.     ProjectBitmap_v = the project bitmap of v;
14.     SelectBitmap_v = the select bitmap of v;
15.     JoinBitmap_v = the join bitmap of v;
16.     if (-ProjectBitmap_sq) | (ProjectBitmap_v) contains 0 then
          /*' |' represents the bitwise OR; '-' represents the bitwise complement */
17.       continue;
18.     else if (-SelectBitmap_sq) | (SelectBitmap_v) contains 0 then
19.       continue;
20.     else if JoinBitmap_sq == JoinBitmap_v then
```

```
21.       continue;
22.     else
23.       add v into ViewList;
24.     end if
25.   end for
26.   return ViewList;
27. end if.
```

In this algorithm, lines 1 - 3 extract the domain of the given SQ sq, sort its domain by using the two-level priority rule, and encode the query expression of sq by using the corresponding bitmap encoding method. Three bitmaps for sq are made available for later view comparison (lines 4 - 6). Line 7 searches the DMVI by using the domain of sq as the search key. If no leaf node is reached, then the empty view set is returned (lines 8 - 9). Otherwise, each view v in the discovered leaf node is checked (lines 11 - 12). Three bitmaps for v are also made available (lines 13 - 15). If all of the bitmaps for sq are matched with those for v, then v is added into the found view set (lines 16 - 24). Finally, the view set that contains all of matched views is returned (line 26).

Assume that the number of materialized views is N and that the number of leaf nodes of a DMVI is M. To search a usable view using the DMVI, the worst-case time complexity (number of views searched) is $O(N)$ (all the views are in one leaf node), which is the same as that of the view sequential search. However, the average time complexity of searching a view using the DMVI is $O(N/M)$, which is usually much better than that $O((1+N)/2)$ of the view sequential search since M is usually much greater than 2. When the bitmap matching is applied, the actual number of view comparisons can be further reduced.

3.5 The DMVI maintenance issues

The last issue we want to discuss is how to maintain the DMVI when a view v (TMV or CMV) is removed from the VS. The work can be done in two stages. In the first stage, we focus on how to update the search path for invalid view v in the DMVI. The main idea is shown as follows: the domain of v is used as the search key to find its representing leaf node n. If n contains some other views besides v, it means that, although v is removed, its search path is still used by other views. Thus, the search path of v remains unchanged, and only the view information of v in n is removed. Otherwise, the search path of v is directly removed (remove n and some useless internal nodes). The algorithm runs as described below.

ALGORITHM 3: **PathRemove**$(v, dmvi)$}
Input: (1) the view v to be removed; (2) the DMVI $dmvi$.
Output: the revised DMVI.
Method:

```
1.  domain_v = Domain of v;
2.  sort domain tables in domain_v using the two-level priority rule;
3.  search dmvi using domain_v as the search key;
4.  if no leaf node found then
5.    return;
    /*The leaf node which contains v is found.*/
6.  else
7.    n = the reached leaf node;
```

```
      /*The search path of v is shared by other views in the DMVI.*/
8.    if n contains multiple views then
9.      remove the information of v in n and return;
      /*The search path of v becomes invalid.*/
10.   else
        /*Remove the search path of v in the DMVI.*/
11.     RecursiveRemove(n, dmvi);
12.   end if
13. end if.
```

In this algorithm, lines 1 and 2 extract the domain of the removed view v and sort that domain using the two-level priority rule. Line 3 searches the DMVI by using the domain of v as the search key. If no leaf node is found, it means that the view is not indexed in the DMVI, thus, no further work needs to be done (lines 4 - 5). Otherwise, the discovered leaf node is checked. If the leaf node contains other views except v, then only the information of v is n is removed (lines 8 - 9). If the leaf node contains only v, then function $RecursiveRemove()$ is called to recursively move the search path of v in the DMVI (line 11).

The following function RecursiveRemove() recursively removes the search path of an invalid view in the DMVI. The input of the function is a node n that is either a leaf node or an internal node in the DMVI. Line 1 finds the direct parent node m of n, and line 2 safely removes n and its associated links. If m still has direct child nodes, it means that m is shared by the search paths of other views and then the function stops here and the search path that removes work is completed (lines 3 - 4). Otherwise, the function calls itself to recursively remove m (lines 5 - 6).

ALGORITHM 4: **RecursiveRemove**($n, dmvi$)
Input: (1) a node n in the DMVI; (2) the DMVI $dmvi$.
Output: the revised DMVI.
Method:

```
1.  m = the direct parent node of m in dmvi;
2.  remove n and its associated links;
3.  if m is the root or has other direct child nodes then
4.    return;
5.  else
6.    RecursiveRemove(m, dmvi);
7.  end if.
```

However, only updating the search path of v in the DMVI is not sufficient. Let us consider an example. After an SQ of a PQ is executed, its result table is saved as a TMV v_1 and it is 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. Therefore, in the second stage, for all the views whose domains contain v, their search paths need to be rebuilt.

First, we have to find all the views whose domains contain v. A straightforward way to do this is to traverse the DMVI. However, we can make use of the properties of the DMVI to improve this type of search. According to the first property of the DMVI mentioned in Section 3.1, at the first level of the tree, if an internal node n is the leftmost child node of the root and its labeled

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 that is rooted at n or the subtrees that are rooted at the nodes on the left to n. Therefore, there is no need to search the nodes on the right to n.

Since TMVs need to be removed and transformed frequently while CMVs and external tables are relatively stable, the search paths that include TMVs have a high chance to become invalid. Furthermore, the search paths that contain elder views also have a high chance to become invalid. By using the two-level priority rule, the TMVs or elder views, which are used as search keys in the DMVI, are picked first and are then inserted into more left branches than its brother nodes, which represent CMVs/external tables or newer views. Therefore, it is easier to search views whose domains contain invalid TMVs or elder views. As a result, the overall DMVI maintenance performance is improved. This is one of the reasons why the priority order (two-level priority rule) for node insertions was defined as such.

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.

In the figure, we can see that tmv_1 and tmv_2 are labeled by 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, only the subtrees that are rooted at n_2 and n_3 need to be searched to find all the views whose domains contain tmv_2.

After all the views v's whose domains contain the invalid view v are discovered, the search path for each v' is rebuilt. The main idea to rebuild the search path for v' is shown as follows: the old search path for v' is removed first. The removing process is similar to that of deleting the search path of a removed view in the DMVI, as described before. Next, the query expression qe of v' is rewritten by merging it with the query expression of v after the occurrence(s) of v is/are removed, and the domain T of v' is updated by replacing v in T with the domain tables

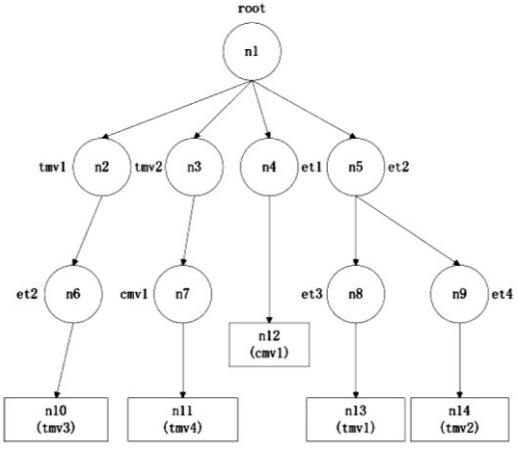

Fig.3. An example of the DMVI with views as domain tables

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: $\pi_{ec_1.a_1}(ec_1 \overset{\triangleright\triangleleft}{}_{ec_1.a_1=ec_2.a_2} ec_2)$; the domain T_2 of a view v_2 is:$\{ec_3, v_1\}$; the query expression of v_2 is: $\pi_{ec_3.a_3}(v_1 \overset{\triangleright\triangleleft}{}_{v_1.a_1=ec_3.a_3} ec_3)$. In this example, the domain of v_2 contains v_1. 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 as: $\pi_{ec_3.a_3}((\pi_{ec_1.a_1}(ec_1 \overset{\triangleright\triangleleft}{}_{ec_1.a_1=ec_2.a_2} ec_2), \overset{\triangleright\triangleleft}{}_{ec_1.a_1=ec3.a3} ec_3)$ 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.

The algorithm for rebuilding the search paths for all the views whose domain include the invalid views is summarized as explained below.

ALGORITHM 5: **RebuildPath**($v, dmvi, workloadlist$)
Input: (1) a removed view v; (2) the DMVI $dmvi$; (3) the list $workloadlist$ of domain tables in the workload.
Output: the revised DMVI.
Method:

```
    /*find all the views whose domains contain v.*/
1.   initialize firstlevel_node and leaf_node;
    /*find all the first level nodes in the DMVI.*/
2.   firstlevel_node = the direct child nodes of the root in dmvi from left to right;
   /*v is labeling a first level node in the DMVI and some unnecessary branches are pruned.*/
3.   if v is labeling a node n in firstlevel_node then
4.    traverse the sub-tree rooted at n and add the reached leaf nodes into leaf_node;
5.    remove n and right brothers of n in firstlevel_node;
6.    for each node t in firstlevel_node do
7.     for each path p rooted at t do
8.      if v is labeling a node in p then
9.       find the leaf node of p and add into leaf_node;
10.     end if
11.    end for
12.   end for
    /*v is not labeling any first level node in the DMVI and the whole tree is traversed.*/
13.  else
14.   for each path p in dmvi do
15.    if v is labeling a node in p then
16.     find the leaf node of p and add into leaf_node;
17.    end if
18.   end for
19.  end if
    /*rebuild the search path for each view whose domain contains v*/
20.  for each node n in leaf_node do
21.   for each view v' in n do
22.    domain_v' = Domain of v';
23.    domain_v = Domain of v;
24.    replace v in domain_v' by domain_v;
25.    rewrite the query expression of v';
26.    sort domain tables in domain_v' using the two level priority rule;
27.    root = root node of dmvi;
```

28. ViewInsertion(v', $dmvi$, $root$, $domain\_v'$, $workloadlist$);
29. **end for**
30. **end for**
/*remove the search path for each view whose domain contains v.*/
31. **for** each node n in $leaf\_node$ **do**
32. **if** n exists in $dmvi$ **then**
33. m = direct parent node of n;
34. **while** m is labeled by v **do**
35. m = direct parent node of m;
36. **end while**
37. remove the subtree rooted at m;
38. RecursiveRemove(m, $dmvi$);
39. **end if**
40. **end for**

In this algorithm, the work is done in two phases. In the first phase, given the invalid view v, all the views whose domains contain v are discovered (lines 1 - 19). In the second phase, for each view discovered from the first phase, its search path is rebuilt (lines 20 - 40).

In the first phase, based on the properties of the DMVI, some unnecessary searches are pruned. If v is represented by a first level node n in the DMVI (line 3), then the properties of the DMVI can be used and only the subtree of n and the subtrees of the left brothers of n are checked. First, the subtree of n is traversed and the reached leaf nodes are directly added into a leaf node set (line 4). Next, the subtree of each left brother of n is traversed. If a search path p contains a node representing v, then the leaf node in p is added into the leaf node set (lines 6 - 12). Otherwise, v does not appear as a first level node in the DMVI. Then the whole tree is traversed and the leaf nodes whose search paths contain nodes representing v are added into the leaf node set (lines 13 - 19).

In the second phase, the search patch for each view v' in the discovered leaf nodes from the first phase is rebuilt. Since v becomes invalid and the domain of v is still available, to make the search path of v' usable, the algorithm updates the domain of v' by replacing v with the domain of v (lines 23 - 24). After that, the query expression of v' is rewritten (line 25), the updated domain of v' is sorted (line 26) using the two-level priority rule, and a new search path is built for v' in the DMVI by using the ordered domain of v' as the search key (lines 27 - 28). Next, all the search paths that contain v are removed from the DMVI (lines 31 - 40).

Assume that the number of views indexed in the DMVI is N and that the maximum number of domain tables for each view is M ($M \ll N$). To delete a view from the DMVI, the worst-case time complexity is $O(4MN - 2M + (N-1)(N-2)/2) = O(N^2)$.

4. EXPERIMENTS

In this section, we report on the results of our experiments to demonstrate the efficiency of our technique.

4.1 Experiment setup

The experiment programs were implemented in Matlab 2010 on a PC with on Intel® dual core (1.5 GHz) CPU and 4 GB memory that was running the Windows® 7 operating system.

The underlying DBMS we used to run the SQs of a PQ was MySQL.

In our experiments, 100 random progressive queries and 50 random external tables with uniformly distributed data were generated. The number of SQs in each PQ was randomly chosen between 2 and 20. The sizes for external tables ranged from 1 to 1,000 disk blocks with each disk block containing 4,096 bytes. The experiments were begun by running (the SQs of) the first PQ and ended after having completed all (100) PQs on MySQL. The timestamps were used to record the starting and ending times for the SQs of the PQs. Multiple PQs were executed simultaneously. The maximum number of PQs that could be run at the same time was set to 10. A DMVI was dynamically constructed to index the materialized views, which were generated by the system. The DMVI was also used to efficiently search usable views for answering the SQs.

Each SQ sq was generated in two steps. First, the domain of sq was determined. The domain of sq contained one or more domain tables, where the domain size was randomly chosen between 1 and 5. Each domain table of sq could either be an external table or a materialized view (TMV or CMV). The probabilities for choosing 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 (i.e., 0.75) of being chosen than that for external tables (i.e., 0.25). 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 addition, to construct the VS and the DMVI, some parameters were set. The VS had two subspaces: CVS (for CMVs) and TVS (for TMVs). Since the number of simultaneously executing PQs was constrained (≤ 10), which implies that the maximum number of TMVs in the TVS was restricted (only the results of SQs of in-process PQs were saved in the TVS as TMVs), we did not set a space limit for the TVS. But, for the CVS, the space limit was set to 25,000 disk blocks. The CVS maintenance strategy is mentioned in [46]. The main idea is that when the CVS overflows, its CMVs are re-estimated and sorted by using their potential benefits. The CMVs with the smallest benefit are removed first. This process continues until the CVS can accommodate the new CMV. The DMVI construction started with a single root node. When a materialized view v (TMV or CMV) was saved in the VS, its domain tables were sorted by the two-level priority rule, and a search path was created for v in the DMVI. The bitmaps of v were generated and saved at the end of the path (i.e., in a leaf node). Before each SQ sq was executed, the DMVI was searched and the usable views were returned. When a view (CMV or TMV) v was removed, its corresponding search path in the DMVI was also removed and the search paths of all the views whose domains included v were rebuilt.

4.2 The performance of the DMVI based view searching technique

The first set of experiments were conducted to evaluate the efficiency of our DMVI based view searching technique (DMVIT). To make the comparison, we used the straightforward technique, which is called the sequential scan based view searching technique (SST). The main idea of the SST is as follows: to find a usable view from the VS to answer the given SQ sq, views are checked one by one from the VS sequentially. If the view v contains the result of sq, then v is

considered to be a candidate view. After examining all the views in the VS, the best (smallest) 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 domain with the given SQ sq. Next, it prunes the views whose bitmaps do not match with those for sq. After that, the discovered views are processed in the same way as the SST (i.e., examining each view against sq to find the best view for answering sq).

In this set of experiments, the I/O costs for processing PQs using the two view searching techniques were first compared. Figure 4 shows the comparison of the total I/O cost of running 100 progressive queries between using the SST and the DMVIT. From the figure, we can see that two performance curves were very close, which indicates that two view searching techniques only have a small effect on the total I/O cost. Since the I/O cost reflects the quality of views used in the PQ processing, the two techniques are comparable in terms of the quality of views found.

To capture both the I/O and view matching costs, Figure 5 shows the comparison between the execution time of running 100 progressive queries and using the SST and the DMVIT. During the processing of PQs, the materialized views were dynamically generated and indexed into the DMVI. Hence, the DMVI was constructed in parallel with the processing of the PQs. The execution time for the DMVI construction was also included in the cost of executing the PQs. From the figure, we can see that two curves are very close at the beginning. However, as the total number of SQs being executed increases, the performance of the DMVIT becomes increasingly better. The reason for this is explained as follows: as more and more SQs being executed, more and more SQ results are materialized and kept in the VS. The cost for view searching by the SST (i.e., examining each view in the VS) increases sharply, but the cost for view searching by the DMVIT (i.e., examining only the views discovered by the DMVI) is relatively stable. As a result, the total PQ execution time of the DMVIT is significantly better than that of the SST. Note that, compared to the SST; the DMVIT utilizes an index to significantly reduce the cost of view matching. However, since the cost of view matching is only reflected in the execution time rather than the I/O cost, the benefit of using the DMVIT is observed in Figure 5 rather than in Figure 4.

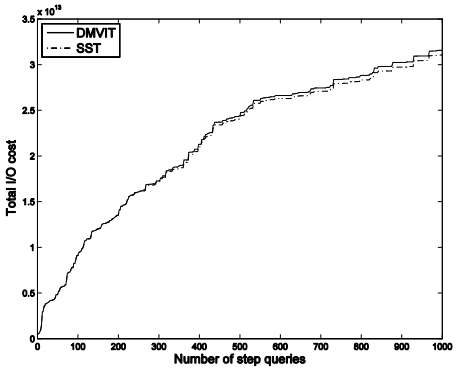

Fig.4. The I/O cost comparison of running 100 progressive queries between the SST and the DMVIT

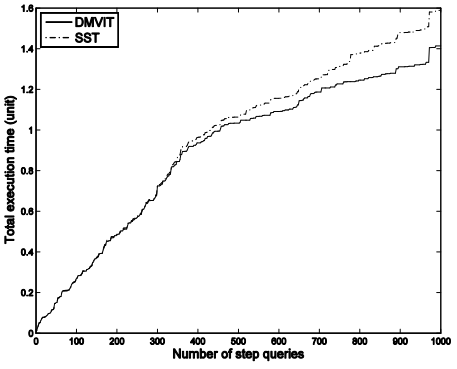

Fig.5. The time cost comparison of runing 100 progressive queries between the SST and the DMVIT

4.3 The scalability of the DMVI based view searching technique

The second set of experiments was conducted to examine the scalability of the DMVIT. The performance of using the DMVIT and the SST with different CVS space limits was compared. Since the maximum number of simultaneously executing PQs was fixed, the size of the TVS was controlled within a certain range. Thus, the size of the VS was dominated by the size of the CVS. The numbers of view comparisons (examining the cost) for the D MVIT with different CVS space limits are shown in Figure 6, and the numbers of view comparisons for the SSTs are shown in Figure 7. From Figure 6, we observe that the three performance curves are very close to each other, which implies that the cost of view matching. by using the DMVIT with various CVS space limits (thus VS sizes) are relatively stable. However, from Figure 7, we can observe significant differences among the three performance curves. On the other hand, from Section 4.2, we know that the PQ performance improves as more materialized views are available. In other words, as more materialized views are made available; the DMVIT gains more performance and incurs less searching overhead, as compared to the SST. Hence, the scalability of the eDMVIT is better than that of the SST.

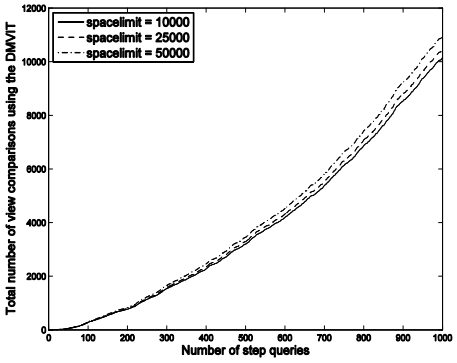

Fig. 6. The performance of the DMVIT with different CVS space limits

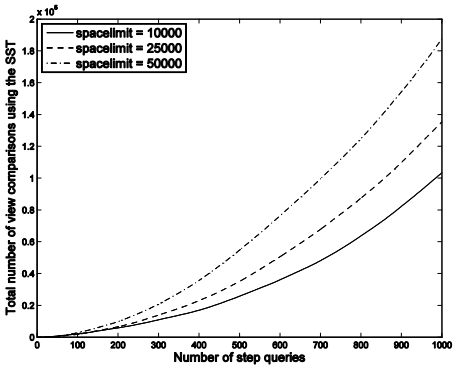

Fig. 7. The performance of the SST with different CVS space limits

4.4 The performance of the bitmap matching

The third set of experiments was conducted to examine the importance of the bitmap matching in the DMVI. Figure 8 shows the performance of the DMVIT with or without the bitmap matching. It is obvious that using the bitmap matching, saves much unnecessary direct view comparison costs. Note that, compared to the saved view comparison cost; the bitmap matching overhead can be ignored.

4.5 The performance of maintaining the DMVI with different DMVI construction techniques

In this set of experiments, the DMVI maintenance cost for the two-level priority ordering based DMVI constructing technique (TPDMVI) and the DMVI maintenance cost for the non-priority ordering DMVI constructing technique (NPDMVI) were compared. The main idea of the TPDMVI is as follows: when a view v is ready to index in the DMVI, the domain tables of v are

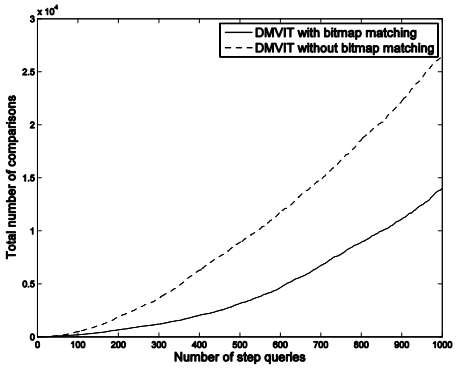

Fig. 8. The performance of the DMVIT with or without the bitmap matching

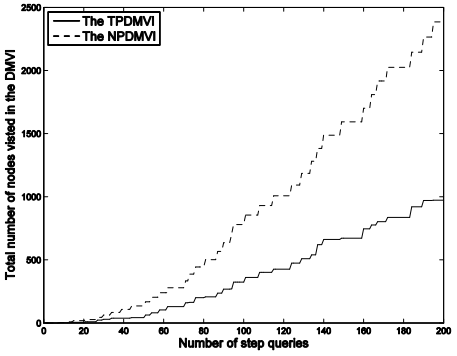

Fig. 9. The performance of the view searching with invalid search paths between the TPDMVI and the NPDMVI

sorted by using the two-level priority rule.

Next, each domain table of v is picked in (ascending) order and inserted into the DMVI. Furthermore, if a node has multiple child nodes, the order of its child nodes is also 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 employ a random order. The purpose of employing the two-level priority order rule is to improve the efficiency of discovering the invalid search paths that related to a deleted view, which is the major component of work for deleting a view from the DMVI (i.e., maintaining the DMVI). When a view v is to be deleted, the system has to discover all the other views whose domains contain v (i.e., having invalid search paths). If an internal node n representing v appears at the first level of the DMVI (child of the root), only the subtree of n and the subtrees of the left brothers of n (if any) are searched by using the TPDMVI, while the whole tree has to be traversed by using the NPDMVI. The performance results of searching the views with invalid search paths by using the TPDMVI and the NPDMVI is compared in Figure 9, where the X-axis represents the total number of SQs in the test and the Y-axis represents the total number of nodes visited in the DMVI during the search. From the figure, we can see that compared to the NPDMVI, the TPDMVI can save a lot on costs in terms of discovering views with invalid search paths during DMVI maintenance.

4.6 The effectiveness of the DMVI based view searching technique

In the last set of experiments, 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. For example, a view $\mathbf{v_1}$ that has a different domain from a given SQ sq is filtered out by the DMVIT when searching the usable views for sq. Assume that a view $\mathbf{v_2}$ is returned by the DMVIT. However, it is possible that $\mathbf{v_1}$ can be used for answering sq and that the size of $\mathbf{v_1}$ is smaller than $\mathbf{v_2}$. In other words, $\mathbf{v_1}$ is more suitable than $\mathbf{v_2}$ for answering sq. In this case, we consider that the most usable view is missed by the DMVIT. We define a hitting rate for the DMVIT as being the percentage of the most usable views that can be discovered by it. In this set of experiments, we utilized a commonly used view matching mechanism in a commercial DBMS and calculated the hitting rates of the most usable views discovered by the

DMVIT and the SST for the tested cases, respectively.

Our results are described as follows: the hitting rates of the SST and the DMVIT were 100% and 83%, respectively; while the numbers of views checked/compared by the SST and the DMVIT during the search were 163,546 and 14,648, respectively. From the experiments, we can see that, compared to the SST, the DMVIT can dramatically reduce the number of checked views (by 91%) while keep a high hitting rate (at 83%) when discovering usable views. These results are consistent with the conclusion observed from Figures 4 and 5. Hence, our proposed DMVI technique is quite effective.

5. CONCLUSION

The rapid growth of data intensive applications has led to an increasing demand to efficiently process progressive queries (PQs). Materialized view based approaches for processing PQs have been proposed in previous work [45,46,47]. However, how to efficiently find usable materialized views in the view storage. for answering the SQs of PQs was challenging and has remained open.

In this paper, we have presented a new index technique, which is called the dynamic materialized view index (DMVI), to efficiently discover usable views for answering SQs from 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 the efficient construction and maintenance of the DMVI tree for a dynamically changing view set. Bitmap encoding methods for generating bitmaps for the query expressions of views (SQs) are suggested. The relevant bitmaps are stored in the leaf nodes of the DMVI tree to support a refined pruning of unusable views. In this paper we have presented the algorithms for constructing, searching, and maintaining the DMVI. 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 extensive experimental results demonstrate that our DMVI is quite promising in reducing the overall query cost. The proposed tree structure supports efficient view management for a dynamic view set.

Our future work includes extending the method for handling more types of SQs, such as those that involve aggregate functions, and incorporating our technique into a real DBMS.

REFERENCES

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

[2] K. Aouiche, and J. Darmont, "Data mining-based materialized view and index selection in data warehouses." *J. Intell. Inf. Syst.*, vol.33, no. 1, 2009, pp. 65-93.

[3] L. Bellatreche, K. Karlapalem and Q. Li, "Evaluation of Materialized View Indexing in Data Warehousing Environments." *Proc. of DaWaK Conf.*, 2000, pp. 57-66.

[4] S. Berchtold, D. A. Keim and H. Kriegel, "The X-tree: An Index Structure for High Dimensional Data." *Proc. of VLDB Conf.*, 1996, pp. 28-39.

[5] D. Calvanese, G. D. Giacomo, M. Lenserini and M. Y. Vardi, "View-based query process: on the

relationship between rewriting, answering and losslessness." *Theor. Comput. Sci.*, vol.371, no. 3, 2007, pp. 169-182.

[6] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces." *Proc. of ICDE Conf.*, 1999, pp. 440-447.

[7] C. Y. Chan, M. N. Garofalakis and R. Rastogi, "RE-Tree: An Efficient Index Structure for Regular Expressions." *Proc. of VLDB Conf.*, 2002, pp. 263-274.

[8] C. Y. Chan and Y. E. Ioannidis, "Bitmap Index Design and Evaluation." *Proc. of SIGMOD Conf.*, 1998, pp. 355-366.

[9] D. Comer, "The ubiquitous B-tree." *ACM Computing Survey*, vol.11, no.2, 1979, pp. 121-137.

[10] F. Fusco, M. Vlachos and M. Stoecklin, "Real-time creation of bitmap indexes on streaming network data." *The VLDB Journal*, vol. 21, no. 3,2012, pp. 287-307.

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

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

[13] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching." *Proc. of SIGMOND Conf.*, 1984, pp. 47-57.

[14] A. Y. Halevy, "Answering queries using views: a survey." *The VLDB Journal*, vol.10, no. 4, 2001, pp. 270-294.

[15] H. He and A. K. Singh, "Closure-Tree: An Index Structure for Graph Queries." *Proc. of ICDE Conf.*, 2006, pp. 38.

[16] B. He, H. Hsiao, Z. Liu, Y. Huang and Y. Chen, "Efficient Iceberg Query Evaluation Using Compressed Bitmap Index." *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 9, 2012, pp. 1570-1583.

[17] G. Himanshu, and I. S. Mumick, "Selection of Views to Materialize in a Data Warehouse." *IEEE Transaction on Knowledge and Data Engineering*, vol. 17, no. 1, 2005, pp. 24-43.

[18] H. Jiang, H. Lu, W. Wang and B. C. Ooi, "XR-Tree: Indexing XML Data for Efficient Structural Joins." *Proc. of ICDE Conf.*, 2003, pp. 253-264.

[19] N. Katayama and S. Satoh: The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. *Proc. of SIGMOND Conf.*, 1997, pp. 369-380.

[20] H. Kimura, G. Huo, A. Rasin, S. Madden and S. B. Zdonik, "CORADD: Correlation Aware Database Designer for Materialized Views and Indexes." *PVLDB*, vol. 3, no. 1, 2010, pp. 1103-1113.

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

[22] T. W. Kuo, C. H. Wei and K. Lam, "Real-Time Data Access Control on B-Tree Index Structures." *Proc. of ICDE Conf.*, 1999, pp. 458-467.

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

[24] P.-A. Larson and J. Zhou, "View matching for outer-join views." *VLDB Journal*, 2007, pp. 29-53.

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

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

[27] I. Nitsos, G. Evangelidis and D. Dervos, "Bitmap-Tree Indexing for Set Operations on Free Text." *Proc. of ICDE Conf.*, 2004, pp. 837.

[28] C.-S. Park, M.-H. Kim and Y.-J. Lee, "Rewriting OLAP Queries Using Materialized Views and Di-

mension Hierarchies in Data Warehouses." *Proc. of ICDE Conf.*, 2001, pp. 515-523.

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

[30] J.T. Robinson, "The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes." *Proc. of SIGMOND Conf.*, 1981, pp. 10-18.

[31] N. Roussopoulos, "View indexing in relational databases." *ACM Trans. on Database Systems*, vol.7, no.2, 1982, pp. 258-290.

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

[33] M. Sadoghi and H. A. Jacobsen, "BE-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space." *Proc. of SIGMOND Conf.*, 2011, pp. 637-648.

[34] Y. Sakurai, M. Yoshikawa, S. Uemura and H. Kojima, "The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation." *Proc. of VLDB Conf.*, 2000, pp. 516-526.

[35] R. R. Sinha, M. Winslett, K. Wu, K. Stockinger and A. Shoshani, "Adaptive Bitmap Indexes for Space-Constrained Systems. *Proc. of ICDE Conf.*, 2008, pp. 1418-1420.

[36] H. H. Sinha and M. Winslett, "Multi-resolution bitmap indexes for scientific data." *ACM Trans. Database Syst.*, vol. 32, no. 3, 2010, pp. 16, 2010.

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

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

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

[40] H. Wang, S. Park, W. Fan and P. S. Yu, "ViST: A Dynamic Index Method for Querying XML Data by Tree Structures." *Proc. of SIGMOND Conf.*, 2003, pp. 110-121.

[41] H. Wang and X. Meng, "On the Sequencing of Tree Structures for XML Indexing." *Proc. of ICDE Conf.*, 2005, pp. 372-383.

[42] K. Wu, A. Shoshani and K. Stockinger, "Analyses of multi-level and multi-component compressed bitmap indexes." *ACM Trans. Database Syst.*, vol. 35, no. 1, 2010.

[43] Jong P. Yoon, "Presto Authorization: A Bitmap Indexing Scheme for High-Speed Access Control to XML Documents." *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 7, 2006, pp. 971-987.

[44] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi and D. Srivastava, "Bed-tree: an all-purpose index structure for string similarity search based on edit distance." *Proc. of SIGMOND Conf.*, 2010, pp. 915-926.

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

[46] C. Zhu, Q. Zhu, C. Zuzarte and W. Ma, "A Materialized-View Based Technique to Optimize Progressive Queries via Dependency Analysis." *Proc. of CASCON Conf.*, 2011, pp. 60 - 73.

[47] C. Zhu, Q. Zhu and C. Zuzarte, "Optimization of Monotonic Linear Progressive Queries Based on Dynamic Materialized Views." *The Computer Journal*, to appear, 2013.

[48] C. Zhu, Q. Zhu, C. Zuzarte and W. Ma: DMVI, "A Dynamic Materialized View Index for Efficiently Discovering Usable Views for Progressive Queries." *Proc. of CASCON Conf.*, 2012, pp. 42 - 56.

[49] Q. Zhu, B. Medjahed, A. Sharma and H. Huang, "The Collective index: A Technique for Efficient Processing of Progressive Queries." *The Computer Journal*, vol. 51, no. 6, 2008, pp. 662-676.

Chao Zhu

He is a PhD candidate 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

He 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, TKDE, 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

He is a Senior technical Staff Member (STSM) in the IBM Canada Lab. He is also a Research Staff Member (RSM) in the Centre for Advanced Studies (CAS) at the lab overseeing collaborative Information Management projects between IBM and academia. He serves on the Distributed Database Architecture Board (DDAB) as a DB2 architect and manages the DB2 Compiler continuous engineering team. He specializes in Database Query Optimization.

Wenbin Ma

He is a Senior Software Engineer in the IBM Toronto Lab. He has worked in IBM DB2 Compiler group for 12 years. He obtained Master degree of Computer Science at University of Alberta in Canada in 2001 and Master degree of Computer Engineering in Bei Hang University in China in 1997. He actively participates in IBM CAS (Center for Advanced Studies) projects.