Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'2011), pp. 60 - 73, Toronto, Canada, Nov. 7 - 10, 2011

A Materialized-View Based Technique to Optimize Progressive Queries via Dependency Analysis

Chao Zhu[†] Qiang Zhu[†] Calisto Zuzarte[‡] Wenbin Ma[‡]

[†]Department of Computer and Information Science
The University of Michigan, Dearborn, MI 48128, USA

[‡]IBM Toronto Laboratory, Markham, Ontario, Canada L6G 1C7

Abstract

Progressive queries (PQ) are a new type of query emerging from numerous contemporary database applications, including e-commerce, social network, business intelligence, and decision support. Such a PQ is formulated in several steps via a set of inter-related step-queries (SQ). How to optimize such PQs represents a new challenge in the development of a database management system. In our previous work, we introduced a materialized-view based technique to process a special type of PQ, called monotonic linear PQs. In this paper, we present a new materialized-view based technique to efficiently process generic PQs. This technique allows an SQ in a given PQ to utilize the results of previous SQs not only from the same PQ but also from other in-process and completed PQs. Due to the storage constraint, it is impossible to retain the results of all the SQs of a completed PQ. Hence, a crucial issue is how to select popular SQs from completed PQs to keep their results as materialized views for optimizing future PQs. To tackle this issue, we introduce a multiple query dependency graph (MQDG) to capture the data source dependency relationships among SQs from multiple PQs. We then present a model to estimate the benefit of an SQ in the MQDG and discuss a procedure to choose critical SQs in the MQDG for materializing their results. The strategies for constructing the MQDG and maintaining the set of materialized views are also suggested. Experimental results demonstrate that our technique is quite promising in efficiently processing PQs.

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

1 Introduction

As the rapid growth of numerous interactive web applications (e.g., Amazon, Google) and data-intensive applications (e.g., astronomy, biology), there is an increasing demand to efficiently process a new type of query, called progressive queries. A progressive query (PQ), which was introduced in [15], consists of a set of inter-related and incrementally formulated step-queries (SQs). An SQ is issued by a user based on the results obtained from the previous SQs of the same PQ.

A product searching at the Amazon web site is a simple example of the PQ. Assume that a user wants to buy a suitable laptop. First, he/she issues a search (step-query) to list all the laptop sales at the web site (database). After the result is returned, the user may realize that the result set is too large, and he/she does not want to scan all the returned sales by following several screens. Thus, the user adds a further condition on the brand name to restrict the laptop sales, say from Dell. However, the result set may still be too large. Therefore, the user further narrows down the result

set by adding another condition for the price limit. But, if the user finds that there is no sale found with the given price limit condition, he/she may go back a step to change the desired brand name of a laptop to HP or Lenovo and then continue his/her search. This example represents a very restricted PQ since a new SQ was selected from a set of pre-defined qualification conditions (hyperlinks) and could only use the result of its immediate previous SQ. For a general PQ, a new SQ can use as input the result(s) of any previously executed SQ(s) (not necessarily the immediate previous one) and adopt any qualification condition that the user likes (not being restricted to a pre-determined set of conditions).

The above example demonstrates a key characteristic of PQs; namely, a PQ cannot be formulated in advance because the user cannot predict what the next SQ is before the executions of its previous SQs are completed. A user typically needs to analyze the results returned by the previous SQs and makes a decision for the next SQ accordingly.

Such an unpredictability raises new challenges to efficiently process PQs. Some popular optimization techniques, such as index methods and materialized view techniques, which are frequently applied to the conventional queries may not be easily used in processing PQs. For example, there exists no indexes on the result table of a PQ, and selecting useful materialized views for optimizing future PQs is difficult. To tackle the challenges, [15] introduces an effective index technique, called the collective index method. The main idea is to construct a special index structure so that a collection of member indexes on an input table of an SQ can be efficiently transformed into indexes on the result table. This result index can be used to speed up the subsequent SQs. In [14], we introduced a materialized-view based approach to efficiently process a special type of PQ, called the monotonic linear PQs. The main idea is to construct a so-called superior-relationship graph based on the special containment properties of monotonic linear PQs and use it to dynamically select materialized views to speed up future PQs. But neither technique in [15] or [14] could deal with generic PQs, for instance, the ones with an SQ that is executed on more than one result table from other SQs. Hence, a new technique is required to efficiently process generic PQs.

In this paper, we present such a new materialized-view based technique to efficiently process generic PQs. In this technique, we allow users to specify their new SQs using not only the results of SQs from the same PQ but also the results of SQs from the other in-process PQs since they are all available in the system without additional cost. Furthermore, this technique also selects some popular (critical) SQs from completed PQs to retain their result tables as materialized views for processing future PQs. The data source dependency relationships among external tables, SQs of in-process PQs and critical SQs of completed PQs are captured in a directed graph, called the multiple query dependency graph (MQDG). A model to estimate the potential benefit of an SQ of a completed PQ based on the MQDG is developed for the technique. SQs with significant benefits are selected as critical ones. Since a user has more options in specifying his/her SQs, with the assistance (e.g., cost estimation) from the system, it is expected that an improved performance of his/her PQ can be achieved. Our experiments demonstrate that this approach is quite promising.

Many materialized view (MV) techniques have been reported in the literature [2, 5, 6, 7, 12, 13, 8, 1, 9, 11]. They are widely used in many different database areas, including distributed databases, data warehouses, and Web databases. A key issue addressed by many such techniques is how to select proper views to optimize future queries. [3] and [10] suggested to use genetic algorithms to generate views based on database tuning and restrict the solution space by taking the similarity of MV configurations into consideration. [4] proposed to cluster possible candidate views first and then adjust the view set according to specific requirements. However, most existing work focuses on how to process conventional queries, which are formulated in one time. Our previous work in [14] was the only one studying how to apply materialized views to optimize PQs. However, as mentioned earlier, that work was restricted to handle a special type of PQs. Applying materialized views to efficiently process generic PQs

is the new issue addressed in this paper.

The remainder of this paper is organized as follows. Background knowledge is introduced and the multiple query dependency graph is defined in Section 2. The main processing procedure of our approach is illustrated and the related algorithms and their functions are presented in Section 3. Experimental results are reported and analyzed in Section 4. Section 5 summarizes the conclusions and future work.

2 Preliminaries

In this section, some background knowledge of this work is provided. An overview of three types of PQs is given and the dependency graph is defined in Section 2.1. A view storage (VS) for the materialized views is introduced in Section 2.2.

2.1 Progressive Query Types and Multiple Query Dependency Graph

PQs are classified into three types in [15]. The first type is called the single-linear PQ. In such a PQ, each SQ (except the initial one) could only use the result returned by its immediate previous SQ as its input. The initial SQ uses an external relation from the database as its input. The second type is called the multiple-input linear PQ. In a PQ of this type, each SQ could not only make use of the result returned by its immediate previous SQ but also take advantage of some other external tables. For instance, a user may issue an SQ to join the result table of its immediate previous SQ with an external table. The third type is called the non-linear PQ, which is the most general one. In such a PQ, an SQ is allowed to make use the result tables of more than one previous SQ. For example, a user may issue an SQ sq_3 to join the result tables of two SQs sq_1 and sq_2. The domain $Domain(SQ)$ of an SQ is defined as the set of its input tables. An input table can be an external table or the result table of a previous SQ.

In our previous work [14], we considered a special type of the multiple-input linear PQ, called the monotonic linear PQ, where multiple inputs are allowed only for the initial SQ. In this paper, we consider general PQs, namely, all types of PQs mentioned above are allowed.

As mentioned earlier, the main characteristic of a PQ is that the user cannot predict what the next SQ is before the previous SQs are executed. Therefore, the result tables of previous SQs for a given PQ have to be made available (not discarded) since they may be used in a future SQ. Usually, multiple PQs are simultaneously processed in a DBMS. The result tables of the executed SQs for these in-process PQs are all kept in the system. We may consider these result tables as temporary materialized views. Conceptually, an SQ uses the result tables of previous SQs from the same PQ. In this paper, we also allow users to use the result tables of SQs from other in-process PQs rather than from the same PQ if a better performance can be achieved. Usually, the result tables for the SQs of completed (historical) PQs are no longer kept in the system. However, for popular result tables of some SQs, we choose to retain them as materialized views even after their corresponding PQs are completed. Such SQs are named critical SQs. The goal is to give users more flexibility (options) in specifying an efficient SQ.

In this work, we employ a so-called multiple query dependency graph (MQDG) to capture the data source dependency relationships among the SQs of the in-process PQs as well as the critical SQs in the system. Let SPQ be the set of the in-process PQs and the PQs that have at least one critical SQ. The multiple query dependency graph for SPQ is defined as a directed graph $MQDG(SPQ) = (V, E, P, F)$, where V is a set of nodes, E is a set of edges, P is a set of labels representing the id's for PQs in SPQ, and F is a function that maps a node in V to a label in P.

Let SSQ be the set of SQs of in-process PQs and critical SQs of completed PQs. Each node in V represents either an external table used by an SQ in SSQ or directly an SQ in SSQ. The former is called a table node, while the latter is called a temporary node (the result table of an SQ from an in-process PQ) or a critical node (the result table of a critical SQ). If a node v_2 representing an SQ uses as input the external table or the result table associated with node

v_1, we say v_2 depends on v_1, which is represented by a directed edge $e =< v_1, v_2 >$ from v_1 to v_2 in E. In this case, we also say that there exists a dependency relationship from v_1 to v_2. Set P in $MQDG(SPQ)$ consists of unique identifiers for all the PQs in SPQ. Function F in $MQDG(SPQ)$ maps (labels) each temporary node representing an SQ in SPQ to the id in P for the corresponding PQ to which the SQ belongs. A table node has no label. An MQDG dynamically grows as more SQs of current PQs or new PQs are issued. Figure 1 shows an example of the MQDG.

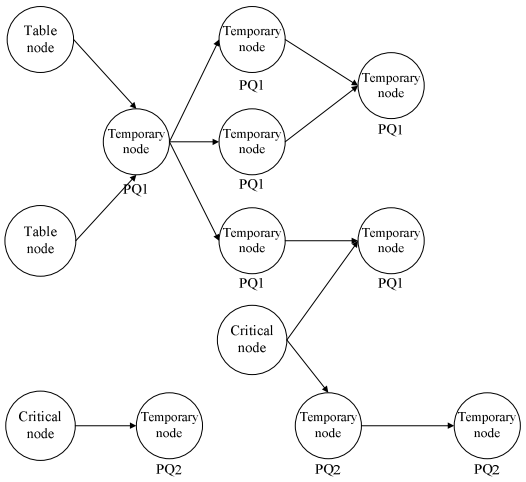

Figure 1: An example of the multiple query dependency graph (MQDG).

Several properties of an MQDG can be observed. First of all, there exist no directed circle in the graph. A directed edge from node v_1 to node v_2 in the graph implies that the SQ for v_2 makes use of the result table of v_1 if v_1 is a temporary node or v_2 is executed on the external table for v_1 if v_1 is a table node. In other words, v_2 is generated later than v_1. On the other hand, all the outgoing paths from v_2 are to connect the nodes which are generated later than v_2. Therefore, it is impossible to form a recursive cycle in the graph. Secondly, there may exist isolated subgraphs even among the SQs from the same PQ in an MQDG. As mentioned earlier, when a user issues an SQ, he/she can not only make use of the result tables for the previous SQs of the same PQ but also take advantage of the result tables of the SQs from other PQs. Hence, the result of an SQ in a PQ q may never be used by any subsequent SQs of q. Similarly, the SQs from different PQs may be connected together in the graph. Other properties of an MQDG include that each table node has no incoming edge and that there is a single sink (final) node for each PQ that returns the final result for the PQ.

Note that a dependency graph (DG) for a given PQ was defined in [15]. There are several differences between a DG and an MQDG. First of all, a DG is for a single PQ, while an MQDG is for multiple PQs. Secondly, a DG is used to illustrate the definition of a (complete) PG, while an MQDG is used to optimize multiple in-process PQs that are incomplete and growing. Finally, a DG does not include nodes for external tables, while an MQDG does.

2.2 View Storage

As mentioned earlier, the result table associated with a temporary node or a critical node is considered as a materialized view. The system needs to have a place to store them. We call such a place the view storage (VS). The VS is divided into two subspaces: temporary node view space (TNS) and critical node view space (CNS). TNS is to store the result table (view) of a temporary node, while CNS is to store the result table (view) of a critical node. For each stored view, the corresponding PQ id and query expression are also save. Figure 2 shows the structure of the view storage.

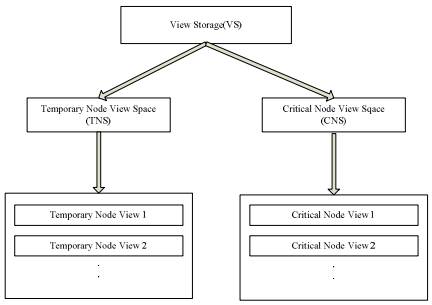

Figure 2: The structure of the view storage

If a space limit is given, it is observed that the size of the TNS determines how many in-process PQs are allowed, while the size of the CNS determines how many beneficial critical

SQ results can be retained. In this work, we make an assumption that the size of TNS is large enough to hold the result tables of all the issued SQs of in-process PQs. We only take the size of the CNS into consideration for our technique. This may be a reasonable assumption given that temporary results for the reasonably small set of currently executing queries might generally be needed to complete the queries although intermediate results could be pipelined without hitting the disk and as such need not be materialized

3 A Materialized-View Based Technique for Efficiently Processing PQs

In this section, we discuss how to construct the MQDG and how to use the MQDG to identify the critical nodes to help users efficiently process concurrently executing PQs. The main processing procedure is introduced in Section 3.1. The critical nodes finding algorithm is presented in Section 3.2. The removal of a node from the MQDG is described in Section 3.3 and the maintenance issue is discussed in Section 3.4.

3.1 Main processing procedure

As mentioned earlier, the result tables of the SQs of in-process PQs as well as the result tables of critical SQs are considered (temporary) materialized views to help users specify future SQs. Users may use the cost estimates provided by the system to decide whether to utilize the materialized views or not for their next SQs.

Since the result tables of all the SQs of in-process PQs are automatically stored in TNS and available to users, no further issue needs to be considered. However, it is clear that it is impossible to keep the result tables of the SQs for all the completed PQs. Hence a key issue that needs to be studied is how to properly choose the critical SQs from completed PQs and retain their results for future use. A technique to address this and other relevant issues is presented in this section. The main processing procedure is introduced in this subsection, and the details of its invoked functions are to be discussed in the following subsections.

Specifically, our technique will address the following four issues: (1) how to construct an MQDG; (2) how to use the MQDG to find the critical nodes from completed PQs; (3) how to remove the non-critical nodes of a completed PQ from the MQDG; and (4) how to remove some critical nodes and reconstruct the MQDG when the CNV overflows. The main procedure to carry out the tasks to address these issues is given below.

ALGORITHM 3.1 : **Selection of Materialized Views via Dependency Analysis**
Input: (1) newly arrived step-query (nsq); (2) multiple query dependency graph $mqdg = (V, E, P, F)$; (3) view storage (vs) including temporary node view space (tns) and critical node view space (cns).
Output: (1) revised $mqdg$; (2) revised vs.
Method:
1. execute nsq and save its result table in tr;
2. add tr and relevant information into $vs.tns$;
   /* revise $mqdg$ to include nsq */
3. **if** nsq is an initial SQ for a new PQ **then**
4. add the id of PQ into $mqdg.P$;
5. **end if**
6. create a temporary node tn labeled with the corresponding PQ id for csq in $mqdg.V$;
7. **for** each table r in $Domain(nsq)$ **do**
8. **if** r is an external table **then**
9. **if** r does not have a node in $mqdg.V$ **then**
10. create a table node m for r in $mqdg.V$;
11. **else**
12. find the table node m representing r in $mqdg.V$;
13. **end if**
14. **else**
15. find the corresponding temporary node m for r in $mqdg.V$;
16. **end if**
17. generate an edge $< m, tn >$ from m to tn in $mqdg.E$;
18. **end for**
    /* find critical nodes in a completed PQ */
19. **if** a PQ pq_1 in $mqdg$ is completed **then**
20. $cnset$ = FindCriticalNode($mqdg$, pq_1);
    /* remove non-critical nodes for a completed PQ */
21. **for** each non-critical node ncn in pq_1 **do**
22. RemoveAndTransfer($mqdg$, ncn);
23. **end for**
24. **for** each node cn in $cnset$ **do**
25. **if** cns has enough space to accommodate the result table (view) of cn **then**
26. transfer result table of (cn) from tns to cns;
    /* the maintainance issue for the CNS */
27. **else** /* cns overflows */
28. CriticalNodeRemove($mqdg$, cns, cn);
29. **end if**
30. **end for**
31. **end if**.

There are two phases in Algorithm 3.1. The first phase (lines 1 - 18) executes the newly arrived SQ and revises MQDG and VS to include this SQ. The second phase (lines 19 - 31) finds the critical nodes of a completed PQ, removes non-critical nodes of the PQ, updates

the MQDG, and maintains the VS. The second phase is done by invoking several functions.

In the first phase, the algorithm first executes the given SQ and saves its result table and relevant information in the VS (lines 1 - 2). If the given SQ is an initial (first) SQ of a new PG, the algorithm adds the PQ id into the MQDG (lines 3 - 5). It then adds relevant nodes and edges into the MQDG to include the given SQ (lines 6 - 18).

In the second phase, the algorithm first invokes function FindCriticalNode() to evaluate the benefit of each SQ of a completed PQ to identify and return a set of critical nodes (lines 19 - 20). After all the critical nodes of a PQ are identified, other non-critical nodes are removed from the MQDG by invoking function RemoveAndTransfer() (lines 21 - 23). For each critical node, its result table is transferred from TNS to CNS in the VS (lines 24 - 26) if there is enough space. If the CNS overflows, another function CriticalNodeRemove() is called to perform necessary maintenance (lines 27 - 29) before the new materialized view is added.

The invoked functions in this algorithm are to be discussed in the following subsections.

3.2 SQ benefit estimation and critical nodes selection

The main use of an MQDG constructed in Section 3.1 is to estimate the potential benefits of SQs of a completed PQ during the process of identifying critical nodes. As mentioned earlier, how to find the critical nodes from a completed PQ is a crucial issue in this work. In this subsection, we focus on discussing this issue and introducing an approach to estimating the potential benefits of SQs using the MQDG.

Let us first introduce some basic concepts for the MQDG, which will be used in the following discussion.

Direct parent node: if there exists an edge from node m to node n in an MQDG, then m is called a direct parent node of n in the MQDG.

Direct child node: if there exists an edge from node m to node n in an MQDG, then n is called a direct child node of m in the MQDG.

Indirect child node: if there exists a (directed) path p from node m to node n and p consists of more than one edge in an MQDG, then n is called an indirect child node of m in the MQDG.

Note that node n in an MQDG could be both a direct child node and an indirect child node of node m. See the example in Figure 3, there exists an edge from sq_1 to sq_6, and there also exists a directed path from sq_1 to sq_6 which consists two edges $\{< sq_1, sq_2 >, \ < sq_2, sq_6 >\}$. Consequently, sq_6 is both a direct child node and an indirect child node of sq_1.

Internal node: if there exists a (directed) path from node m to node n and the PQ id's of both n and m are the same, then n is called an internal node of m.

External node: if there exists a (directed) path from node m to node n, and the PQ id of m is different from that of n, then n is called an external node of m.

Now let us discuss the details about how to estimate the potential benefit of an SQ and how to find the critical nodes by using the MQDG. The main idea to estimate the potential benefit of an SQ sq_1 for future queries is to consider the benefit that sq_1 has already brought to its direct and indirect child nodes in the MQDG.

We have developed a model to quantitatively capture the benefit that an SQ (i.e., a temporary node) in an MQDG has brought to its direct and indirect child nodes. Assume that we want to calculate the benefit that node (SQ) sq_1 has brought to its direct/indirect child node (SQ) sq_2 using the MQDG. The following affecting factors are considered.

(1) *The distance*: it is the number of edges in a path from node sq_1 to node sq_2. The larger the distance is, the smaller the benefit sq_1 could bring to sq_2. As an illustration, we consider the following two scenarios: 1) sq_2 is a direct child node of sq_1. In this case, sq_2 could directly make use of the result of sq_1. 2) sq_2 is an indirect child node of sq_1 and the path from sq_1 to sq_2 that is under consideration is $\{< sq_1, sq_3 >, < sq_3, sq_2 >\}$. In this case, sq_2 could not directly take advantage of the result of sq_1. It is clear that sq_1 could make more contribution to executing sq_2 in the first scenario than in the second scenario. In other words, sq_1 could bring more benefit to sq_2 if the distance from sq_1 to sq_2 along the path that is under consideration is shorter. Note that, if there are multiple paths from sq_1 to sq_2, the benefits

gained by sq_2 through them are accumulated.

(2) *The node type (internal or external)*: it also makes a significant difference whether sq_2 is an internal node or an external node of sq_1. Obviously, the SQs of a PQ have a much higher chance to use the results of previous SQs from the same PQ. However, after the PQ is completed, most internal SQ nodes may never be used by other queries. Thus, an PQ may have many internal nodes, but they may not bring any benefits for future queries. On the other hand, future SQs can be considered as external nodes of sq_1 if they make use of the result of sq_1. Therefore, external nodes are more relevant than internal nodes to represent future SQs. In other words, an external node of sq_1 could have a higher benefit than an internal node of sq_1.

(3) *The number of inputs*: it represents the number of incoming edges of sq_2, assuming sq_2 is a direct child node of sq_1. The reason why this factor matters is that an SQ may only make a partial contribution to the evaluation of its direct child nodes. The larger the number of inputs that sq_2 has, the less the benefit that sq_1 could bring to sq_2. Consider the following two different scenarios. The first scenario is that sq_2 has only one incoming edge, which is from sq_1. In this case, sq_2 is evaluated totally based on the result table of sq_1. The second scenario is that sq_2 has n ($n > 1$) incoming edges, one of which is from sq_1. In this case, sq_2 is evaluated based on several SQ result tables and maybe some external tables as well. It is obvious that sq_1 makes more contribution (bring more benefit) to sq_2 in the first scenario. If sq_2 is an indirect child node of sq_1, the situation becomes more complicated because many intermediate SQs on the path from sq_1 to sq_2 may also have more than one incoming edge. In this case, sq_1 makes even less contribution to sq_2 and the incoming edges of all the intermediate nodes need also to be considered.

To estimate the potential benefit of a temporary node (SQ) sq in an MQDG for fugure queries, we can use the accumulated benefit that sq has brought to all its direct and indirect child nodes along all the possible paths. Let $ChdS(sq)$ be the set of direct and indirect child nodes of sq, $PthS(sq, c)$ be the set of paths from sq to its child node c, $NdeS(p)$ be the set of child nodes (including c) of sq on the path p from sq to c, $|p|$ denote the length of path p, $NE(x)$ is the number of incoming edges that node x has, $Ex(sq, c)$ is a function having value 1 if c is an external node of sq and having value 0 if c is an internal node of sq, and $In(sq, c) = 1 - Ex(sq, c)$. Assume that, if sq' is a direct child node of sq, sq' has only one incoming edge (from sq), and sq' and sq belong to the same PQ, then sq brings 1 unit[1] of benefit to sq'. The following model/formula is derived to estimate the potential benefit of sq:

$$Benefit(sq) = \sum_{c \in ChdS(sq)} \sum_{p \in PthS(sq,c)} \frac{(W_d)^{|p|-1} * [W_E * Ex(sq, c) + W_I * In(sq, c)]}{\prod_{x \in NdeS(p)} NE(x)},$$

where $W_d \in (0, 1)$, $W_E > 0$ and $W_I > 0$ are real number constant coefficients.

The formula essentially calculates the total benefit that sq has brought to all its direct and indirect child nodes along all possible paths. W_d represents the benefit reducing rate as the distance increases. For example, for a typical value $W_d = 0.5$ (it will be used in our remaining discussion), the relevant benefit contribution $(W_d)^{|p|-1}$ becomes 1.0, 0.5, 0.25, 0.125 ... for distance 1, 2, 3, 4, ..., respectively. We could see that the larger the distance is, the smaller the benefit is. W_E and E_I are the constant coefficients to differentiate the benefit impact from an external node or an internal node. For example, we can set $W_E = 2$ and $W_I = 1$ (they will be used in our remaining discussion), which implies that an external node is twice as important as an internal node. The factor $1/\prod_{x \in NdeS(p)} NE(x)$ represents how the benefit for node c from sq is affected by the number of incoming edges for all the child nodes of sq along path p from sq to c.

Let us consider the example in Figure 3. Assume that we want to calculate the benefit that sq_1 could bring to sq_6. First, all possible paths from sq_1 to sq_6 are listed:

[1]We could take the effect of the result table size $|sq|$ of sq into consideration by assigning a benefit value proportional to $|sq|$. However, we choose to use a constant unit to simplify our discussion.

(1) $p_1 = \{< sq_1, sq_6 >\}$;
(2) $p_2 = \{< sq_1, sq_2 >, < sq_2, sq_6 >\}$;
(3) $p_3 = \{< sq_1, sq_2 >, < sq_2, sq_3 >, < sq_3, sq_5 >, < sq_5, sq_6 >\}$.
Clearly, $Ex(sq_1, sq_6) = 0$, $In(sq_1, sq_6) = 1$.

For path p_1, $|p_1| = 1$, $NE(sq_6) = 3$. Thus, the benefit that sq_1 could bring to sq_6 through p_1 is:

$$Benefit\_p_1 = \frac{(0.5)^0 * 1}{3} = \frac{1}{3} .$$

For path p_2, $|p_2| = 2$, $NE(sq_2) = 1$, $NE(sq_6) = 3$. Thus, the benefit that sq_1 could bring to sq_6 through p_2 is:

$$Benefit\_p_2 = \frac{(0.5)^1 * 1}{1 * 3} = \frac{1}{6} .$$

For path p_3, $|p_3| = 4$, $NE(sq_2) = 1$, $NE(sq_3) = 1$, $NE(sq_5) = 2$ and $NE(sq_6) = 3$. Thus, the benefit that sq_1 could bring to sq_6 via p_3 is:

$$Benefit\_p_3 = \frac{(0.5)^3 * 1}{1 * 1 * 2 * 3} = \frac{1}{48} .$$

Therefore, the total benefit that sq_1 could bring to sq_6 is to add the above three benefit values together, i,e., $Benefit\_sq_6 \approx 0.52$. If we want to estimate the total benefit of sq_1, we just need to add all the benefit values that sq_1 could bring to all its direct and indirect child nodes via all the possible paths together.

Let us give an algorithm to estimate the benefit of a temporary node in an MQDG using the above model. The main idea of the algorithm is to traverse all the paths from the given node in a deep-first fashion to accumulate the benefit values that the node has brought to each of its direct and indirect child nodes.

ALGORITHM 3.2 : **CalculateBenefit**($mqdg$, t)
Input: (1) multiple query dependency graph $mqdg$; (2) temporary node t.
Output: benefit value b_t of t.
Method:
1. $b_t = 0$;
2. **for** each direct child node n of t **do**
3. N = number of incoming edges of n;
4. **if** n.pqid = t.pqid **then**
   /* n is an internal node */
5. $b_n = W_I$/N;
6. **else if** n.pqid != t.pqid **then**
   /* n is an external node */
7. $b_n = W_E$/N;
8. **end if**
9. $b_t = b_t + b_n$;
10. b_t = RecursiveAcc($mqdg, t, n, b_t, b_n$);
11. **end for**
12. return b_t.

In Algorithm 3.2, b_t denotes the total benefit that t could bring to all its direct and indirect child nodes, and b_n denotes the benefit that t could bring to its current individual (direct or indirect) child node n along one path. They are calculated for each direct child node n of t (lines 2 - 11), and they are calculated in different ways based on if n is an internal or external node of t (lines 4 - 8). A recursive function RecursiveAcc() is called to calculate the benefit that t could bring to the children of n along the current path.

ALGORITHM 3.3 : **RecursiveAcc**($mqdg, t, n, b_t, b_n$)
Input: (1) multiple query dependency graph $mqdg$; (2) temporary node t; (3) child node n of t; (4) current accumulative benefit value b_t of t; (5) current individual benefit value that t has brought to n.
Output: benefit value b_t of t.
Method:
1. **for** each direct child node m of n **do**
2. N = number of incoming links of m;
   /* n and m are both internal nodes or both external nodes*/
3. **if** (n.pqid = t.pqid and n.pqid = m.pqid) or (n.pqid! = t.pqid and m.pqid != t.pqid) **then**
4. $b_m = W_d$ * b_n * (1/N);
   /* n is an internal node and m is an external node*/
5. **else if** n.pqid = t.pqid and n.pqid != m.pqid **then**
6. $b_m = W_d$ * b_n * (1/N)* W_E/W_I;
   /* n is an external node and m is an internal node*/
7. **else if** n.pqid != t.pqid and m.pqid =t.pqid **then**
8. $b_m = W_d$ * b_n * (1/N) * W_I/W_E;
9. **end if**
10. $b_t = b_t + b_m$;
11. b_t = RecursiveAcc (dg, t, m, b_t, b_m);
12. **end for**
13. return b_t;

Algorithm 3.3 is a recursive function to traverse all the (direct and indirect) child nodes of an input node n in the depth-first fashion. The benefit b_n that t has brought to n along a traversed path is known as an input. The benefit b_m that t has brought to each direct child node m of n is computed based on b_n (lines 4, 6, 8) and the total benefit b_t of t is accumulated (line 10). If the node type (internal or external) of m is the same as that of n (line 3), the relevant coefficient (W_I or W_E) used in the benefit calculation for b_m does not change (line 4). If the node type changes from internal to external (line 5), the relevant coefficient used in the benefit calculation for b_m needs to change from W_I to W_E (line 6). If the node type changes from external to internal (line 7), the relevant

coefficient used in the benefit calculation for b_m needs to change from W_E to W_I (line 8).

Using function CalculateBenefit(), we identify all the critical nodes from a completed PQ as follows.

ALGORITHM 3.4 : **FindCriticalNode(*mqdg*, *fpq*)**
Input: (1) multiple query dependency graph *mqdg*; (2) a completed PQ *fpq*.
Output: a set of critical nodes for *fpq*.
Method:
1. Initialize *cnset* to empty;
2. **for** each SQ fsq in *fpq* **do**
3. $benefit$ = CalculateBenefits(*mqdg*, fsq);
4. **if** $benefit > THRESHOLD$ **then**
5. add fsq into *cnset*;
6. **end if**
7. **end for**
8. return *cnset*.

Algorithm 3.4 selects each SQ fsq from the completed PQ fpq whose potential benefit is sufficiently large as a critical SQ (node).

3.3 Non-critical node removal

After the critical SQs (nodes) of a completed PQ are identified, all the non-critical nodes of the PQ need to be removed from the MQDG. However, when a node n is removed from the MQDG, how to deal with the edges associated with n need to be properly addressed. Edges in an MQDG represent the dependency relationships on which our benefit calculation relies. We need to maintain the dependency relationships among the remaining nodes in the MQDG after the removal, including those went through the removed node n. Hence non-critical nodes should be removed carefully and the relevant dependency relationships should be transferred to the remaining nodes.

The following algorithm removes the non-critical nodes and transfers the dependency relationships properly.

ALGORITHM 3.5 : **RemoveAndTransfer(*mqdg*, *n*)**
Input: (1) multiple query dependency graph *mqdg*; (2) a node n that needs to be removed.
Output: a revised *mqdg* with n removed.
Method:
1. let r be the result table of n;
2. let q be the query expression of n;
3. **for** each direct child node m of n **do**
4. replace r in the query expression of m by q;
5. **for** each direct parent node t of n **do**
6. create a directed edge from t to m in *mqdg*;
7. **end for**
8. **end for**
9. remove all the incoming and outgoing edges for n from *mqdg*;
10. remove n from *mqdg*;
11. return *mqdg*.

In Algorithm 3.5, the given node n is safely removed and all dependency relationships are transferred in four steps. In the first step, the query expressions for all the direct child nodes of n are changed (lines 3 - 4). We know that the result table r of n is used in each of its direct child nodes. Since node n is to be removed, r will no longer exist. Hence, we replace r in the query expression of each direct child node of n by the query expression of n. As a result, r is removed from the domain of each direct child SQ (node). For example, consider sq_1: $\sigma_{c_1=v1}$ (R1); sq_2: $\sigma_{c_2=v2}$ (Result(sq_1)); where σ is the selection operation in the relational algebra. When node sq_1 is removed, the query expression of sq_2 has to be changed to: $\sigma_{c_2=v2}$ ($\sigma_{c_1=v1}$ (R1)). In the second step, new directed edges are generated from each direct parent node t of n to each of its direct child node (lines 5 - 7). Essentially, the tables represented by the direct parent nodes of n are added to the domain of each of its direct child nodes. In the third step, all the edges associated with n are safely removed (line 9). In the last step, n is finally removed (line 10).

Let us use an example to illustrate how to remove a node and transfer all its dependency relationships in an MQDG using Algorithm 3.5. Assume that we are given an MQDG as shown in Figure 3. The set of SQs in the figure includes:

1. sq_1: $\pi_{c_1,c_2,c_3}(\sigma_{c_1=v1}(R1 \bowtie R2 \bowtie R3))$,
2. sq_2: $\sigma_{c_2=v2}(R(sq_1))$,
3. sq_3: $\sigma_{c_3=v3}(R(sq_2))$,
4. sq_5: $\sigma_{c_5=v5}(R(sq_3) \bowtie R(sq_4))$,
5. sq_6: $\sigma_{c_6=v6}(R(sq_1) \bowtie R(sq_2) \bowtie R(sq_5))$,
6. sq_7: $\sigma_{c_2=v7}(R(sq_1))$,
7. sq_8: $\sigma_{c_7=v8}(R(sq_6) \bowtie R(sq_7))$,

where $R(sq_i)$ denotes the result table of sq_i.

Let us try to remove sq_6 from the graph. In the first step, the query expressions of the nodes that use the result table of sq_6 are rewritten. In this example, sq_8 is changed and rewritten to: sq_8: $\sigma_{c_7=v8}((\sigma_{c_6=v6}(R(sq_1) \bowtie R(sq_2) \bowtie R(sq_5))) \bowtie R(sq_7))$.

Next, directed edges are generated from each direct parent node of sq_6 to each direct child node of sq_6. In this example, the edges are generated from sq_1 to sq_8, sq_2 to sq_8 and sq_5 to sq_8. After that all edges associated with sq_6

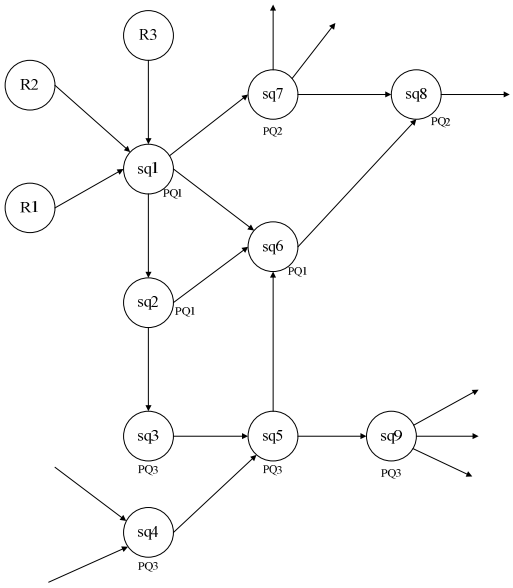

Figure 3: The MQDG for the example

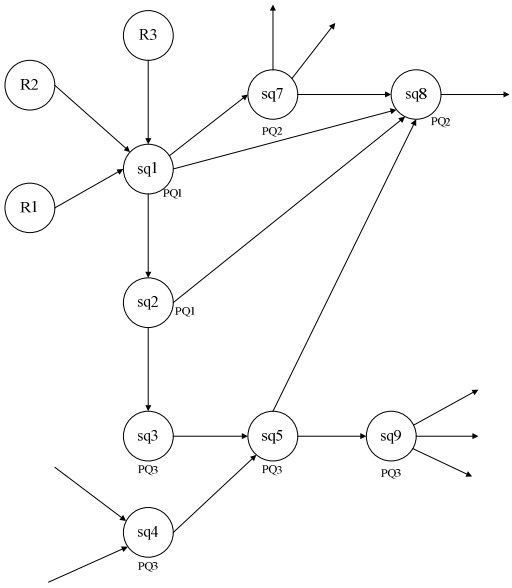

Figure 4: The MQDG of the example after sq_6 is removed

are removed and finally, sq_6 is removed. The resulting MQDG is shown in Figure 4.

3.4 Critical node view space maintenance

The last issue we want to discuss in this section is the maintenance of the critical node view space (CNS). As we mentioned in Section 2.2, the CNS stores all the materialized views for the critical nodes in the VS. The size of CNS is constrained. Therefore, when the CNS overflows, we have to make a decision to remove some views and free some space for accommodating new critical nodes (views). A natural way to do this is to sort the critical nodes in the CNS according to their potential benefits. The node (view) with the smallest benefit is removed first. However, the potential benefit of a critical node is dynamically changing since the nodes and edges in the MQDG are updated very frequently. Therefore, when the CNS overflows under a given space constraint, we adopt an approach to re-estimating the potential benefit for each critical node (view) in the CNS by using the current MQDG. All the critical nodes are then sorted according to their new benefit values. The critical nodes (views) with the small benefit values are removed from the CNS until sufficient space is freed to accommodate a new critical node (view). This approach is described in the following algorithm, which is invoked in Algorithm 3.1.

ALGORITHM 3.6 :
CriticalNodeRemove($mqdg, cns, cn$**)**
Input: (1) multiple query dependency graph $mqdg$; (2) the critical node view space cns; (3) the new critical node cn.
Output: (1) revised cns; (2) revised $mqdg$.
Method:
1. initialize $benefitlist$ to empty;
2. **for** each critical node n in cns **do**
3. bft = CaculateBenefit($mqdg, n$);
4. add (n, bft) into $benefitlist$;
5. **end for**
6. sort all nodes in $benefitlist$ in the ascending order by their benefit values;
7. **while** cns does not have enough space to accommodate the result table (view) of cn **do**
8. get the next node m from $benefitlist$;
9. RemoveAndTransfer($mqdg, m$);
10. **end while**
11. add the result table (view) of cn into cns.

Algorithm 3.6 first initializes a benefit list $benefitlist$ (line 1). For each critical node in the CNS, the algorithm re-estimates its benefit by invoking function CalculateBenefit() and saves the node along with its benefit value in

benefitlist (lines 2 - 4). It then sorts the nodes in *benefitlist* according to their benefit values (line 6). The replacement procedure repeatedly picks the critical node (view) with the smallest benefit from *benefitlist* and removes it by calling function RemoveandTransfer() until enough free space is obtained for the new critical nodes (line 7 - 10). The new node (view) is finally added to CNS (line 11). We assume that the CNS has enough space to store at least one critical node view.

4 Experiments

To evaluate the performance of our technique, we conducted simulation experiments. Experiment programs were implemented in Matlab 2007 on a PC with Intel® dual core (1.5 GHz) CPU and 4 GB memory running the Windows ® 7 operating system.

Specifically, 100 random generic progressive queries (PQ) and 10 external tables were used in our experiments. The sizes for all external tables ranged from 1 to 1000 disk blocks and each disk block contained 4096 bytes. Each PQ was composed of one or more step-queries (SQ), where the number of steps was randomly chosen between 2 and 10. Each SQ could have one or more input tables (external tables or previous SQ result tables) and the number of inputs was also randomly generated between 1 and 5. The result table size of an SQ was calculated by multiplying the product of all the input table sizes with a selectivity. The I/O cost was approximated by the product of the input table sizes of the SQ. Such I/O cost was used as the performance measure for our experiments.

In addition, each input table of an SQ could be either an external table or a (previous) SQ result table (including a critical node table). The probabilities to choose an external table or an SQ result table were not kept the same in our experiments. It was assumed that users had a preference to choose SQ result tables (including the result tables of SQs of other in-process PQs and critical SQs of completed PQs) over external tables for their new SQs. This was because a user tended to utilize their previous results in their new SQs. Hence, the SQ result tables were assigned a larger probability to be chosen.

To build the relevant multiple query dependency graph (MQDG), we recorded the starting and ending times for each PQ and the execution time for each SQ. The maximum number of PQs allowed to be executed simultaneously in the system was set to 10. The MQDG and the critical node view space (CNS) were initially set to empty. When the processing of a new PQ started, its executed SQs were added into the MQDG gradually. Each SQ not only had a chance to use the results of previous SQs from the same or other in-process PQs in the MQDG but also had a chance to use the results of critical SQs in the CNS. When a PQ pq_i was completed, we applied the model/formula introduced in Section 3.2 to estimate the potential benefit of each SQ in pq_i. In the experiments, W_d was set to 0.5, W_E was set to 2 and W_I was set to 1. After the benefits of all the SQs in pq_i were estimated, we choose those SQs which could bring sufficient potential benefits to its direct and indirect children as critical SQs (nodes).

The first experiment was to compare the performance between the separately executed PQ (SEPQ) processing approach and the critical nodes based PQ (CNPQ) processing approach. The separately executed PQ processing approach executes PQs separately, i.e., an SQ of a PQ pq_1 could only use as input the results of the previous SQs within pq_1 or external tables. The critical nodes based PQ processing approach allows an SQ to use any node tables (external tables and result tables of any SQs) as its input. In other words, the CNPQ approach has more temporary SQs results available to reuse. Note that, although an SQ (sq_1) in CNPQ may be formulated differently from the corresponding SQ (sq_2) in SEPQ, the results of sq_1 and sq_2 are the same. The performance comparison is shown in Figure 5. The X-axis represents the total number of SQs in the test, and the Y-axis represents the I/O cost (i.e., the number of disk block accesses). From the figure, we can see that CNPQ always outperforms SEPQ. Initially, the performance difference is not very significant. As more and more PQs were executed, more and more critical nodes were selected to optimize the future SQs. As a result, at the right end of the figure, a very significant performance improvement can be ob-

served.

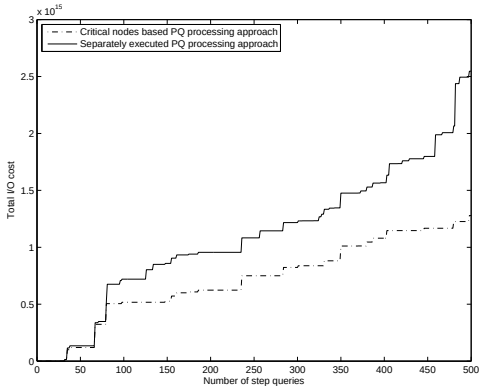

Figure 5: Performance comparison between SEPQ and CNPQ

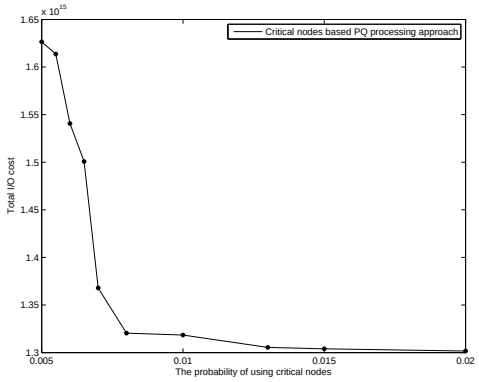

Figure 6: Performance behavior for different probabilities of critical node use

The second experiment was to discover the relationship between the performance of CNPQ and the probability of using a critical node. In the first experiment, the probability of using a critical node was set to 0.01, namely, each critical node had 1% chance to be used by an SQ. In this experiment, we varied this probability to see how it would affect the performance. The discovered relationship is shown in Figure 6. The X-axis represents the probability of using a critical node in the test, and the Y-axis represents the I/O cost. From the figure, we can observe that, as the probability increases, the I/O cost of CNPQ becomes increasingly smaller. In other words, the performance of CNPQ becomes better and better. This is because, when the probability increases, a critical node has a higher chance to be used to optimize the SQs.

In the third experiment, we varied the upper limit for the number of the critical nodes allowed in the CNS and wanted to see how it would affect the performance of CNPQ. For a given upper limit for the number of critical nodes, say 50, when the CNS overflowed, the handling strategy discussed in Section 3.4 was used to maintain the CNS. The motivation for doing this experiment was as follows. We wanted to know how many critical nodes could work well for our technique and how many critical nodes were sufficient. This study would help us find an appropriate solution to balance the time complexity and the space complexity. In this experiment, the upper limit for the number of the critical nodes was varied between 0 and 100. The performance behavior was shown in Figure 7. From the figure, we can see that, except some individual cases, the general trend of the performance curve is that, as the upper limit for the number of the critical nodes increases, the performance becomes increasingly better. Furthermore, we noted that, at the beginning, the cost decreases sharply. It means that increasing a small number of critical nodes could bring a dynamically improved performance. However, as the upper limit continues to increase, the performance improvement becomes more and more flat. It is observed that a trade-off solution for our experiment case is about 30.

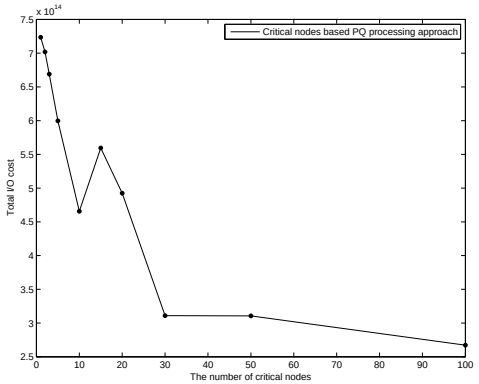

Figure 7: Performance behavior for different maximum numbers of critical nodes allowed

5 Conclusions

Efficiently processing PQs is demanded by numerous contemporary applications but is challenging. In this paper, we have presented a new materialized-view based technique to efficiently process generic PQs. The main contributions of the paper are summarized as follows.

We have introduced a multiple query dependence graph to capture the data source dependency relationships among the external tables, SQs of in-process PQs and critical SQs of completed PQs. This graph is used to discover the critical (popular) SQs of a completed PQ to keep their result tables as materialized views.

We have identified the important factors that affect the benefit that an SQ can bring to its subsequent SQs (children) and developed a model/formula incorporating these factors to estimate the potential benefit of an SQ in a completed PQ using the multiple query dependency graph. A critical SQ is selected if its estimated benefit is sufficiently large.

We have presented the relevant algorithms to process the SQs of a given PQ, to dynamically construct a multiple query dependency graph, to evaluate the potential benefit of an SQ of a completed PQ, to identify critical SQs and store their results in a view storage.

We suggested a strategy to safely remove nodes from the multiple query dependency graph while all the dependency relationships are transferred and retained. We have also presented a solution to the maintenance issue so that the critical (node) view space can be efficiently used under a space constraint.

Our experimental results demonstrate that the proposed technique is quite promising in processing the generic PQs.

Our future work includes further improving the benefit evaluation model by incorporating more factors such as the input table size, developing multi-layered materialized view technique to efficiently process PQs, and evaluate the proposed techniques in a real DBMS environment.

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 PhD research assistant with an IBM CAS fellowship. His research interests include database query processing and query optimization.

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 refereed journals and conference proceedings including TODS, TOIS and VLDBJ. 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. 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 Staff Member at the IBM Canada Laboratory. He serves on the database architecture board and manages the DB2 Compiler continuous engineering development team. His main expertise is in the area of query optimization. Calisto is also a research staff member at the Center for Advanced Studies (CAS) and oversees projects related to information management.

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

both.

Intel and Celeron are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

References

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

[2] 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.

[3] Chaves, L., E. Buchmann, F. Hueske, K. Bhm: Towards materialized view selection for distributed databases. In *Proc. of ACM*, NY, USA, 2009.

[4] Gong, A. and W. Zhao: Clustering-Based Dynamic Materialized View Selection Algorithm. In *Proc. of IEEE Conf.*, Washington, DC, USA, 2008.

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

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

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

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

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

[10] Talebian, S. and S. A. Kareem: Using Genetic Algorithm to Select Materialized Views Subject to Dual Constraints. In *Proc.of ACM Conf.*, 2009.

[11] Zaharioudakis, M., R. Cochrane, G. Lapis, H. Pirahesh, M. Urata: Answering Complex SQL Queries Using Automatic Summary Tables. In *Proc.of SIGMOD Conf.*, pp. 105-116, 2000.

[12] Zhang, C., X. Yao and J. Yang: An Evolutionary Approach to Materialized Views Selection in a Data Warehouse Environment. In *IEEE Transaction on Systems, Man and Cybernetics*, 31(3): 282-294, 2001.

[13] Zilio, D., C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. J. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, D. Liang, G. Valentin: Recommending Materialized Views and Indexes with IBM DB2 Design Advisor, In *Proc.of ICAC*, pp.180-188, 2004.

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

[15] 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.