Advance Access publication on 28 February 2013
doi:10.1093/comjnl/bxt021

Optimization of Monotonic Linear Progressive Queries Based on Dynamic Materialized Views

CHAO ZHU[1,*], QIANG ZHU[1] AND CALISTO ZUZARTE[2]

[1]*Department of Computer and Information Science, The University of Michigan, Dearborn, MI 48128, USA*
[2]*IBM Canada Software Laboratory, Markham, Ontario, Canada L6G 1C7*
[*]*Corresponding author: zhuchaon1@gmail.com*

There is an increasing demand to efficiently process emerging types of queries, such as progressive queries (PQs), from contemporary database applications including telematics, e-commerce and social media. Unlike conventional queries, a PQ consists of a set of step-queries (SQ). A user formulates a new SQ on the fly based on the result(s) from the previous SQ(s). Existing database management systems were not designed to efficiently process such queries. In this paper, we present a novel technique to efficiently process a special type of PQ, called monotonic linear PQs, based on dynamically materialized views. The key idea is to create a superior relationship graph for SQs from historical PQs that can be used to estimate the benefit of keeping the current SQ result as a materialized view. The materialized views are used to improve the performance of future SQs. A new storage structure for the materialized views set is designed to facilitate efficient search for a usable view to answer a given SQ. Algorithms/strategies to efficiently construct a superior relationship graph, dynamically select materialized views, effectively manage the materialized views set and efficiently search for usable views are discussed. Experiment results demonstrate that our proposed technique is quite promising.

Keywords: query optimization; progressive query; materialized view

Received 8 August 2012; revised 22 October 2012
Handling editor: Leonid Libkin

1. INTRODUCTION

In recent years, we have witnessed the emergence of many contemporary database applications such as telematics, e-commerce, bioinformatics, business intelligence and decision support. Such data-intensive applications raise new challenges to process advanced types of queries [1–5]. A new type of query, called the progressive query (PQ), was presented in Zhu *et al*. [5]. It was observed that in many applications, users routinely perform queries step by step. In each step, the query uses the result(s) returned from the previous step(s). The desired query result is gradually reached in multiple steps under the user's direction. Hence, unlike a conventional query, a PQ is formulated in several steps, i.e. a set of inter-related step-queries (SQ). A user formulates his/her SQs on the fly based on the result(s) returned by previous SQ(s).

Let us consider the following example. Assume that a traveler wanted to select a set of songs from a worldwide song database containing millions of songs and lyrics to burn some CDs to be played on his/her next trip. He/she first issued a query on the database to list all the songs released in the last 3 years. He/she found that there were too many such songs in the database. He/she then narrowed down the list by adding a condition on the genre. However, he/she found that the list was still too long. Thus, he/she further narrowed down the list by adding another condition to restrict songs to those sung by several his/her favorite singers and with a length <4 min. Finally, he/she found a reasonable (not too large) set of songs he/she liked to enjoy for his/her trip. Some other examples of PQs: an on-line shopper searches for a product to purchase from the web via several SQs (e.g. searching for related products, checking their reviews, comparing their features and prices, etc.) to optimize the quality/cost within his/her budget; a biologist identifies an unknown DNA sequence via a sequence of tasks (e.g. alignment, validation, and comparison); a geo-scientist accesses massive volumes of earth science data via a number of complex multi-step queries and a decision maker explores and

analyzes the relevant information from multiple data sets and in multiple steps.

The previous examples demonstrate two main characteristics of a PQ. First, the SQs of a PQ cannot be known beforehand. Each SQ, which can be a full-fledged query on its own, is formulated dynamically by the user. The user needs to know the result(s) of the previous SQ(s) to determine the next SQ. Second, a PQ is frequently used to access large data sets, and the intermediate result returned from a SQ may not be held in memory.

These characteristics of a PQ raise new challenges to processing such a query efficiently. For example, because of the second characteristic, an efficient access method such as an index-based one is desired. However, many conventional indexes (e.g. the B+-tree [6]) that are typically created on base relations may not be directly applicable because a SQ that is not for the first step of a PQ uses the intermediate result(s) from the previous SQ(s). To tackle this challenge, an effective collective index technique was introduced in Zhu *et al.* [5]. The main idea of this technique is to construct a special index structure that allows a collection of member indexes on an input relation of a SQ to be efficiently transformed into indexes on the result relation, which can be used to speed up the subsequent SQs. This work was the first to address efficient processing issues for PQs.

It is well known that utilizing materialized views to efficiently process queries is one of the important optimization techniques for conventional queries. However, the first characteristic of a PQ presents a challenge for relevant issues like selection of promising materialized views that can be used to efficiently process such queries. Although a non-initial step-query sq in a given progressive query pq is performed on the result(s) of the previous SQ(s) of sq in pq (and maybe on other external data sets as well), which appears to have some similarity with a conventional query performed on predefined views, the previous SQ(s) as well as sq itself in pq cannot be determined in advance before pq is started due to the dynamic nature of a PQ. Hence, SQs in a PQ are different from (predefined) views. On the other hand, if some views that can be used to answer some SQs of PQs are selected (defined) and materialized beforehand, a user may utilize such materialized views (instead of the previous SQs) to evaluate a relevant step-query sq'. There are two cases in which the efficiency of processing the corresponding PQ could be improved in this way. In the first case, evaluating sq' using the materialized views is more efficient than using the results of the previous SQs of sq'. In this case, the previous SQs are still issued and processed although some of them are replaced by materialized views when evaluating sq'. In the second case, some previous SQs (and their own possible previous ones) may never be issued since the user may be able to determine sq' based on the materialized views without those previous SQs. Therefore, processing of such previous SQs is avoided. Now the questions are: how to select promising materialized views for optimizing PQs and how to search for feasible materialized views for answering/evaluating SQs.

In this paper, we present a novel materialized-view-based technique to process a special type of PQ, called monotonic linear PQs. To tackle the challenge for selection of promising materialized views for PQ optimization, we utilize the unique properties of monotonic PQs to dynamically construct a so-called superior relationship graph (SRG) for SQs from the PQs that have been executed. When the execution of the current SQ of a PQ is completed, a chance for its result to be saved as a materialized view is given. The SRG is used to estimate the benefit of such a materialization. If it is beneficial, the result of the current SQ is kept as a materialized view. Since we utilize the results of executed SQs, there is not much cost for materialization. The materialized views are used to optimize the SQs of future PQs. To facilitate the search for a usable view for answering a given SQ, we suggest a new storage structure, called the relationship-linked structure (RLS), for the set of materialized views (SMVs). The RLS maintains the superior relationships among the materialized views. Utilizing the maintained relationships, our view search can achieve both efficiency in processing and quality for the result. Relevant algorithms and strategies/heuristics to efficiently create and maintain a SRG, dynamically select materialized views (SQs), search for a usable materialized view to answer a given SQ and manage the SMVs are presented. A framework to apply the presented technique to optimize PQs is also discussed.

The work that is most related to PQs in the literature includes query processing for continuous queries [7–10], adaptive (dynamic) query optimization [11–15] and ETL processing [16–19]. Continuous queries require the repeated execution of a query over a continuous stream of data [8]. The main difference with PQs is that a continuous query is formulated at once (although data are dynamic), while a PQ is formulated in several steps. The idea in adaptive query optimization is to exploit information that becomes available at query runtime and adapt the query plan to changing environments during execution. While the adaptive query optimization problem may be seen as 'progressive' (performed at compile-time and run-time), queries are, however, formulated at once ('non-progressive'). Extraction–Transformation–Loading (ETL) processes are used to extract data from multiple sources, cleanse them, integrate them and propagate them to a data warehouse incrementally. In an ETL workflow, activities/operators are chained together. One operator uses the results of previous operators. However, all the activities/operators in an ETL workflow are programmed in advance, which is different from a PQ, although new data are incrementally added to a data warehouse. In addition, an operator in an ETL workflow tends to be much simpler than a SQ in a PQ. The latter can be a full-fledged query on its own. The only previous work that directly studied efficient processing of PQs is the collective index technique proposed by Zhu *et al.* [5].

Applying materialized views to speed up query processing has been well studied in the literature [20–24]. Different

types of database systems were considered, including relational databases [23–25], object-oriented databases [26], data warehouses [27, 28], XML databases [29, 30] and others [31, 32]. Various issues were studied, including materialized view selection [33–36], materialized view maintenance [37–39], materialized view matching [40], materialized view concurrency control [41, 42] and materialized view indexing [43, 44]. For the materialized view selection problem, which is the main issue studied in this paper, a number of techniques have been suggested in the literature. Liang *et al.* [35] introduced heuristic-based algorithms to solve the view selection problem under the maintenance time constraint for data warehouses. Lee and Hammer [34] suggested a genetic algorithm to compute a near-optimal set of views to minimize the total query response time over all queries. Ezeife proposed a method for selecting and materializing views, which selects and horizontally fragments a view and recomputes the size of the stored partitioned view while deciding further views to select [45]. Mistry *et al.* [36] presented algorithms that can be used to efficiently select materialized views to speed up workloads by exploiting common subexpressions and indices. Hung *et al.* [46] derived a cost model and efficient view selection algorithms that effectively exploit the gain and loss metrics. Agrawal *et al.* [47] described an industry-strength tool for automated selection of materialized views for SQL workloads. Chirkova *et al.* [33] presented techniques for finding a minimum-size view set for a single query without self-joins by using the shape of the query and its constraints. Tang *et al.* [48] developed a heuristic method to identify a minimal view set for a given XPath query. Aouiche *et al.* [49] proposed a framework to exploit a clustering technique to solve the materialized view selection problem. Gupta and Mumick [50] presented polynomial-time heuristics for selection of views using an AND view graph, an OR view graph or an AND–OR view graph for different scenarios. Although much work on materialized view has been done in the past, no work on studying how to apply materialized views to efficiently process PQs, as we report in this paper, has been found in the literature. The approach to utilize the unique properties of monotonic PQs to effectively select materialized views and efficiently search for feasible views is our novel idea.

The remainder of this paper is organized as follows. The preliminaries and properties of PQs are introduced in Section 2. The materialized-view-based PQ processing procedure, two algorithms to construct the SRG, the strategy to select materialized views, the storage structure and management for the SMVs and the view-search algorithms are presented in Section 3. Experimental results are reported in Section 4. The conclusions and future work are summarized in Section 5.

2. PRELIMINARIES

In this paper, we focus on discussing how to apply a dynamic materialized view technique to process a specific type of PQ, called the monotonic linear PQ, on a relational database. In this section, an overview of different types of PQs is given. Especially, the monotonic linear PQ is introduced. A SRG that is used in our technique is defined. The main properties of the monotonic linear PQ are discussed.

2.1. Types of PQ

A PQ is formulated in several steps. Each step, referred to as a SQ, is executed over one or more relations and returns one relation as a result. Result(SQ) and Domain(SQ) represent the result relation of the SQ and the set of relations on which the SQ is executed, respectively. A SQ can be executed on either the result relation(s) returned by the previous SQ(s) and/or other external base relation(s). Zhu *et al.* [5] classified the PQs into the following three types:

Type 1: *single-input linear PQs*. A single-input linear PQ has the following characteristics. Each SQ in such a PQ uses a single relation as its input. If the SQ is the initial (first) SQ, then the input is an external relation. Otherwise, the input is the result relation returned by its previous SQ. The relationship among the SQs of such a PQ demonstrates a linear structure.

Type 2: *multiple-input linear PQs*. A multiple-input linear PQ has the following characteristics. At least one SQ takes more than one relation as its input. If this SQ is the initial SQ, its domain includes multiple external relations. Otherwise, its domain includes at least one external relation. Each step uses the result returned by its previous SQ. Hence, the relationship among SQs is also linear.

Type 3: *non-linear PQs*. A non-linear PQ has the following characteristic: at least one SQ has the results returned by more than two other SQs (and possibly external relations as well) as inputs. Thus, the relationship among SQs demonstrates a non-linear structure.

In this paper, we consider an extended type of single-input linear PQ that allows the initial SQ to have multiple external relations. Since the result size of each SQ is monotonically decreasing as the processing of the query progresses, we call this type of PQ the monotonic linear PQ.

2.2. Superior–inferior relationship and SRG

In our dynamic materialized view technique, we utilize a so-called SRG to determine whether the result of a SQ under consideration should be materialized as a view. A SRG captures the superior (or inferior) relationships among the SQs for historical PQs.

Let sq_1 and sq_2 be two (distinct) SQs belonging to the same or different historical PQs. The superior relationship from sq_1 to sq_2 is defined as follows. For every tuple t_2 in Result(sq_2), if there exists tuple t_1 in Result(sq_1) such that t_2 can be completely derived from t_1, we say there is a superior relationship from sq_1 to sq_2, where sq_1 is called a superior of sq_2 and sq_2 is called an inferior of sq_1.

Consider the following example. Let Result(sq_1) = $\{\langle a_1, a_2, a_3\rangle, \langle b_1, b_2, b_3\rangle, \langle c_1, c_2, c_3\rangle\}$, Result($sq_2$) = $\{\langle a_1, a_3\rangle, \langle b_1, b_3\rangle\}$ and Result(sq_3) = $\{\langle a_1, a_4\rangle\}$. Since every t_2 in Result(sq_2) can be derived from a tuple in Result(sq_1), sq_1 is a superior of sq_2 (i.e. sq_2 is an inferior of sq_1). However, a_4 of $\langle a_1, a_4\rangle$ in Result(sq_3) cannot be derived from any tuple in Result(sq_1). Hence, there is no superior or inferior relationship between sq_1 and sq_3.

Intuitively, a superior relationship indicates that, if we select the superior SQ as a materialized view, its inferior SQ can be evaluated by utilizing this materialized view. Hence each superior relationship represents a benefit case for the superior SQ to be materialized. However, there is an exception. When two SQs with a superior relationship belong to the same PQ, the inferior SQ usually does not directly use the result of its superior SQ unless the latter is its immediate previous step. The SRG captures those useful superior relationships among SQs for the historical PQs.

An SRG is defined as a digraph with three components $G = (V, E, B)$, where V is a set of nodes representing the set of SQs in the given historical PQs; E is a set of directed edges $\langle sq', sq''\rangle$ representing the superior relationships from SQ sq' to SQ sq'' with the constraint that either sq' and sq'' do not belong to the same PQ or sq' is the immediate previous step of sq''; B is a set of pairs $\langle n, id\rangle$ indicating the identifier id of the PQ to which the SQ represented by node n belongs. Note that the benefit of materializing the result of an SQ represented by a node in an SRG can be measured by the number w of out-going edges that n has. We call w the weight of n, which can be calculated for a given SRG.

Example 1. Given the following four relations:

PAPER(P#, Title, FirstAuthor, PublishYear),
AUTHOR(A#, A_Fname, A_Lname, Area),
EDITOR(E#, E_Fname, E_Lname, Area),
REVIEW(E#, P#, Date),

assume that every paper has been reviewed by an editor. Let us consider the following three PQs.

Progressive Query 1 (pq_1):

sq_1: $\pi_{Title,\ PublishYear,\ A\_Lname}(PAPER \overset{\bowtie}{}_{FirstAuthor=Aid}\ AUTHOR)$,
sq_2: $\pi_{Title,\ A\_Lname}(\sigma_{PublishYear=2009}(Result(sq_1)))$,
sq_3: $\pi_{Title}(\sigma_{A\_Lname='Smith'}(Result(sq_2)))$.

Progressive Query 2 (pq_2):

sq_4: $\pi_{E\_Lname,\ Title,\ PublishYear}(PAPER \overset{\bowtie}{}_{PAPER.P\#=REVIEW.P\#}$
$(REVIEW \overset{\bowtie}{}_{REVIEW.E\#=EDITOR.E\#}\ EDITOR))$,
sq_5: $\sigma_{PublishYear>2008}((Result(sq_4))$,
sq_6: $\pi_{Title}(\sigma_{PublishYear=2009}(Result(sq_5)))$.

Progressive Query 3 (pq_3):

sq_7: $\pi_{Title,\ PublishYear}(\sigma_{PublishYear>2008}(PAPER))$,
sq_8: $\pi_{Title}(\sigma_{PublishYear=2009}(Result(sq_7)))$.

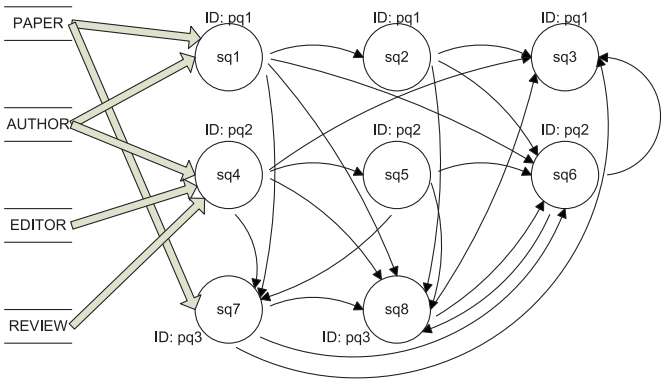

FIGURE 1. Superior relationship graph of Example 1.

Figure 1 shows the SRG for these three PQs. From the figure, we can see that four SQs would benefit from materializing the result of sq_1. The number of out-going edges for a node v is the weight of v, which is not shown in the figure. Clearly, the weights of the nodes in an SRG can be calculated once the graph is given.

2.3. Main properties of monotonic linear PQs

As we will see, the following two properties of the monotonic linear PQs are useful in developing an efficient processing technique.

Property 1. Result(sq_i) $\sqsupseteq$ Result(sq_j) if $i < j$ and sq_i, sq_j are two SQs belonging to the same PQ, where $\sqsupseteq$ indicates that the right operand can be completely derived from the left one. According to the definition, the current SQ only uses the result relation returned by the previous SQ. Hence, if sq_j is one of the subsequent SQs of sq_i, every tuple in Result(sq_j) must be derivable from Result(sq_i).

Property 2. Weight(sq_i) $\geq$ Weight(sq_j) if $i < j$ and sq_i, sq_j are two SQs belonging to the same PQ.

As defined earlier, the weight of an SQ is the number of out-going edges in the SRG, which represents the benefit of materializing the result of the SQ. Based on Property 1, sq_i must be a superior of sq_j. As mentioned before, we do not consider the superior relationships between two non-consecutive SQs within the same PQ when we construct the SRG. All the other superior relationships (out-going edges) for sq_j must also be valid for sq_i.

In the following section, we will discuss how to apply the above properties to improve the efficiency of processing this type of PQ.

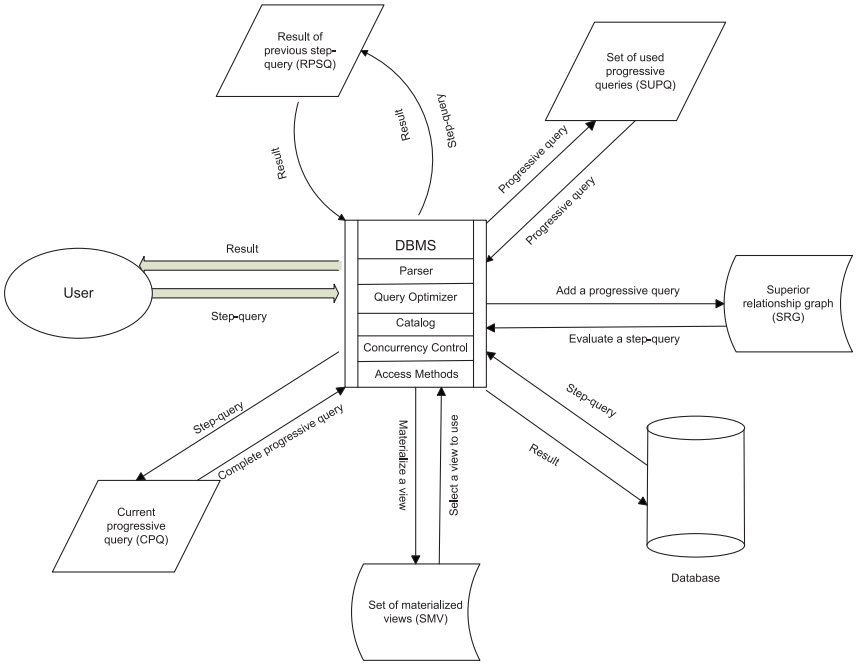

FIGURE 2. PQ processing procedure based on dynamic materialized views.

3. DYNAMIC MATERIALIZED-VIEW-BASED PQ PROCESSING

To efficiently process PQs, we introduce a dynamic materialized-view-based processing procedure for PQs in Section 3.1. Efficient strategies to create and update a SRG are discussed in Section 3.2. The algorithm to decide whether to materialize a view is discussed in Section 3.3. The storage structure and algorithms to manage the SMVs are given in Section 3.4. The view-search algorithms are described in Section 3.5.

3.1. PQ processing procedure

The view materialization techniques have become popular in query optimization, as mentioned in Section 1. The decision for view materialization is typically based on statistic information such as access frequency. However, unlike a conventional query, a PQ is formulated as a number of inter-related SQs. Each SQ cannot be known beforehand. No one can predict what the next SQ could be. Hence, there is no prior knowledge about future user (step) queries when deciding view materialization. This situation raises a challenge to applying a materialized-view-based technique to efficiently process PQs.

To tackle the challenge, we present a dynamic materialized-view-based approach to processing PQs. Figure 2 depicts the processing procedure. There are several components involved in the procedure. The user submits one SQ at each step for the current progressive query (CPQ). The current step-query (CSQ) is the one that is currently being processed in the system. The underlying database management system (DBMS) coordinates the PQ processing based on the dynamic materialized-view approach. This DBMS has all the typical modules such as the parser, catalog, query optimizer and concurrency control that a conventional DBMS has. However, these modules are enhanced to handle a PQ based on dynamically materialized views as follows. A SRG is dynamically constructed by the system. Initially, the SRG is empty. When more and more completed PQs are dynamically added to it, it grows larger and larger. This graph is used to determine whether materializing the result of the CSQ is beneficial. If so, the CSQ is materialized as a view to be used for future SQs. If an SQ of the CPQ is chosen to be materialized, the CPQ is put into a set of used PQs (SUPQ) rather than added into the SRG when it is completed. The reason for this is that, if one of the SQs of a PQ has been materialized, the SQs of this PQ should not be used in the SRG to estimate the benefits of materializing another SQ. Otherwise, the benefits of a materialized SQ may be double counted. A PQ in the SUPQ can be added to the SRG later on when its materialized SQ is removed from the set of the materialized views because of the space limitation. The result of the previous SQ (RPSQ) is always saved for the possible use of evaluating the CSQ. The CSQ is evaluated either on a materialized view (if beneficial) or on the base relation(s) in the database (for the first SQ) or on the RPSQ. The set of the materialized views (SMV) is managed. Each materialized view

mv is associated with its corresponding SQ $mv.sq$ as well as its access frequency $mv.freq$ (note that mv itself represents the materialized view).

The details of the PQ processing procedure are given in the following algorithm.

REVISED ALGORITHM SEGMENT 3.1. **Dynamic materialized-view based PQ processing procedure (DMVPQ)**
Input: (1) current step-query (csq); (2) current progressive query (cpq); (3) set of materialized views (smv); (4) result of previous step-query ($rpsq$); (5) set of used progressive queries ($supq$); (6) superior relationship graph (srg).
Output: (1) the result of csq; (2) a revised srg; (3) a revised cpq; (4) a revised smv; (5) a revised $supq$.
Method:

```
1. if the domain of csq consists of a base relation(s) then
   /* csq is the 1st SQ, i.e., user starts a new PQ */
2.  if cpq is not empty then   /* cpq contains a completed previous PQ */
3.   for each step-query sq_i of cpq from 2nd to the last do
4.    merge sq_i and sq_{i-1}, and replace sq_i by merged query;
5.   end for
6.   if any step-query sq_i in cpq is found as mv.sq for some view mv in smv
then
7.    add cpq into supq;
8.   else AddtoSRG(cpq, srg) end if
9.  end if
10. set cpq as a new PQ with csq as the 1st SQ;
11. mv=SearchView(csq, smv, size of Domain(csq) );
12. if mv is not null then
13.  evaluate csq on mv;
14.  mv.fc++;
15. else
16.  evaluate csq on base relation(s) in the database;
17. end if
18. let mcsq = csq;
19. else /* csq is not the 1st SQ and cpq is ongoing */
20.  add csq to cpq;
21.  merge csq with all its previous SQs in cpq and save the merged query in
mcsq;
22.  mv=SearchView(mcsq, smv, size of rpsq );
23.  if mv is not null then
24.   evaluate mcsq on mv;
25.   mv.fc++;
26.  else
27.   evaluate csq on rpsq;
28.  end if
29. end if
30. if (CheckWeight(srg, mcsq)) then
31. create a materailzed view mv for mcsq;
32. AddtoSMV(mv, smv, srg, supq);
33. end if.
```

There are two phases in Algorithm 3.1. The first phase (lines 1–29) evaluates the current SQ and updates the SRG. The second phase (lines 30–33) decides whether the result of the current SQ should be materialized for the future use and updates the SMVs.

In the first phase, the algorithm first checks whether the given SQ (csq) is the first (initial) SQ (line 1) of a new PQ. If so, the user is actually starting a new PQ and the previous PQ (i.e. the one saved in cpq if any) is completed. In this case, the previous PQ in cpq needs to be added into either the SRG srg or the set $supq$ of used PQs (lines 2–9). Lines 3–5 convert each SQ in cpq into one that is operated directly on the base relation(s) in the database, which can be then compared with the SQs for the materialized views. If one of SQs in cpq is found to have been materialized, cpq is put into $supq$ (lines 6–7). Otherwise, cpq is added into srg by algorithm $AddtoSRG()$ (line 8). After having processed the previous PQ in cpq, cpq is reset to a new PQ with csq as the first (initial) SQ (line 10). If a materialized view whose associated SQ is a superior of csq and whose size is smaller than the size of the relation(s)[1] in Domain(csq) is found from the materialized view set smv by algorithm SearchView(), we evaluate csq on the found materialized view instead of its (base) operand relation(s) (lines 11–14). Otherwise, we evaluate csq on its base operand relation(s) in the database directly (lines 15–16). If csq is not the first SQ, cpq holds the previous SQs of the current/ongoing PQ. In this case, csq is added to cpq (line 20). To check if csq can be evaluated on a materialized view, it needs to be converted into a SQ, $mcsq$, on the base relation(s) in the database (line 21). If there exists a materialized view whose associated SQ is a superior of $mcsq$ and whose size is smaller than the size of the result of the SQ directly preceding csq, we evaluate csq on the materialized view (lines 22–25). Otherwise, we evaluate csq on the result of its previous SQ ($rpsq$) (lines 26–28).

Note that $mcsq$ and csq have the same result. However, the former is specified on the base relation(s), while the latter is specified on the (temporary) RPSQ (if not the first SQ). For example, when merging SQs sq_1 and sq_2 from pq_1 in Example 1, we have the following merged SQ:

$$msq_2 : \pi_{Title,\ A\_Lname}(\sigma_{PublishYear=2009} \times (PAPER \overset{\bowtie}{_{FirstAuthor=Aid}} AUTHOR))$$

on base relations *PAPER* and *AUTHOR*, which has the same result as sq_2.

In the second phrase, the algorithm checks to see whether saving the result of the current SQ $mcsq$ (i.e. csq) as a materialized view is beneficial by invoking algorithm Checkweight() (line 30). If so, it creates an entry for the relevant information (e.g. result, query expression and access frequency) on the materialized view for $mcsq$ and invokes an algorithm $AddtoSMV()$ to add the entry into smv (lines 31–33).

The invoked algorithms: $AddtoSRG()$, $SearchView()$, $CheckWeight()$ and $AddtoSMV()$ are to be discussed in the following subsections.

3.2. SRG construction

The SRG is a key component for our dynamic materialized-view-based PQ processing technique. It allows us to dynamically accumulate information about executed PQs and effectively use it to select materialized views for efficient execution of future PQs. To efficiently construct such a graph,

[1]The Cartesian product is considered if there is more than one relation.

we apply several heuristic rules derived from the properties of the monotonic linear PQs that were discussed in Section 2.3.

We present two constructing algorithms: generating based and pruning based. The former automatically generates as many other superior (inferior) relationships as possible once one is found, while the latter prunes as many other impossible cases as possible once a superior (inferior) relationship is not found between two nodes. Both can significantly reduce the cost for testing the existence of superior (inferior) relationships among nodes.

An SRG starts from an empty one and is constructed in an incremental way as more and more PQs are added into the graph gradually. An isolated new PQ npq can be represented by a set of nodes (one for each SQ in npq), a set of edges (connecting interrelated SQs in npq) and a set of node-identifier pairs (one for each SQ in npq). To add npq into the SRG, the above nodes, edges and node-identifier pairs are inserted first. The system then finds the set of edges representing the superior or inferior relationships between the (new) SQs in npq and the (old) SQs in the current SRG. This can be done in two stages: the superior stage and the inferior stage. In the superior stage, all the superior relationships from the new SQs to the old SQs are identified. In the inferior stage, all the inferior relationships from the new SQs to the old SQs are identified. The edges representing these relationships are added into the SRG. The aforementioned two algorithms apply heuristic rules in the above two stages to improve the constructing performance.

The generating-based algorithm applies the following two heuristic rules:

Heuristic Rule 1: If there exists an edge from sq_i to sq_j (sq_i, sq_j are two SQs $\notin$ the same PQ) in the SRG, then there exist edges from sq_i to all sq_k's if sq_k satisfies the following conditions: (1) $k > j$; (2) $sq_k, sq_j \in$ the same PQ.

Heuristic Rule 2: If there exists an edge from sq_i to sq_j (sq_i, sq_j are two SQs $\notin$ the same PQ), then there exist edges from all sq_k's to sq_j if sq_k satisfies the following conditions: (1) $k < i$; (2) $sq_k, sq_i \in$ the same PQ.

The details of the algorithm are specified as follows.

REVISED ALGORITHM SEGMENT 3.2. **Generating-Based AddtoSRG1(*npg*, *srg*)**

Input: (1) new progressive query (npg); (2) superior relationship graph (srg).
Output: revised superior relationship graph (srg).
Method:

1. **if** srg is empty **then** startempty = true;
2. **else** startempty = false **end if**
   /* Adding an isolated PQ npq into srg */
3. add the node and the node-identifier pair for each SQ of npq into sets V and B of srg, respectively;
4. add an edge from each SQ of npq to its immediate subsequent SQ (if any) of npq into edge set E of srg;
5. **if** not startempty **then**
   /* Stage 1: finding external superior relationships */
6. **for** each PQ opq (other than npq) in srg **do**
7. **for** each SQ nsq of npq from the last to the first **do**
8. **for** each SQ osq of opq from the first to the last **do**
9. **if** there exists an edge from nsq to osq **then**
10. break;
11. **else if** there exists a superior relationship from nsq to osq **then**
12. add an edge from nsq to osq into edge set E of srg;
13. **for** each subsequent SQ osq' in opq **do**
14. **if** edge from nsq to osq' does not exist **then**;
15. add an edge from nsq to osq' into edge set E of srg;
16. **end if**
17. **end for**
18. **for** each previous SQ nsq' in npq **do**
19. **if** edge from nsq' to osq does not exist **then**;
20. add an edge from nsq' to osq into edge set E of srg;
21. **for** each subsequent SQ osq' in opq **do**
22. **if** edge from nsq' to osq' does not exist **then**;
23. add an edge from nsq' to osq' into edge set E of srg;
24. **end if**
25. **end for**
26. **end if**
27. **end for**
28. break;
29. **end if**
30. **end for**
31. **end for**
32. **end for**
    /* Stage 2: finding external inferior relationships */
33. **for** each PQ opq (other than npq) in srg **do**
34. exchange the roles of opq and npq in lines 7–31 to find the superior relaltionships from an SQ in opq to an SQ in npq;
    /* i.e., finding the inferior relationships from an SQ in npq to an SQ in opq */
35. **end for**
36. **end if**.

In this algorithm, lines 1 and 2 set a flag to indicate whether the given SRG is empty or not. If it is empty, neither stage 1 nor stage 2 needs to be considered. Lines 4–5 add the nodes, node-identifier pairs and internal edges for the SQs from the given PQ into the SRG. The edges between a node for the PQ and an external node that has already existed in the given SRG are added in two stages. Stage 1 adds the edges for the superior relationships (lines 6–32), while stage 2 adds the edges for the inferior relationships (lines 33–35).

In stage 1, the algorithm considers one old (existing) PQ in the SRG at a time (line 6). It then scans the SQs of the new PQ backwards and the SQs of the old PQ under consideration forwards and examines each pair of SQs from the two PQs (lines 7–8). If there exists a superior relationship between the pair, an edge connecting the corresponding nodes is added into the SRG (lines 11–12). The algorithm then automatically generates more superior relationships based on Heuristic Rule 1 (lines 13–17 and 21–25) and Heuristic Rule 2 (lines 18–20). The relevant edges representing these superior relationships are added into the SRG (see Fig. 3). Because of the above automatic generation, it is possible that a relevant edge has already been added when a pair of SQs from the two PQs under consideration is examined. Such situations are considered by the algorithm to avoid duplicate additions (lines 9, 14, 19 and 22).

In stage 2, the new PQ and the old PQ under consideration play the opposite roles, compared with stage 1, because an

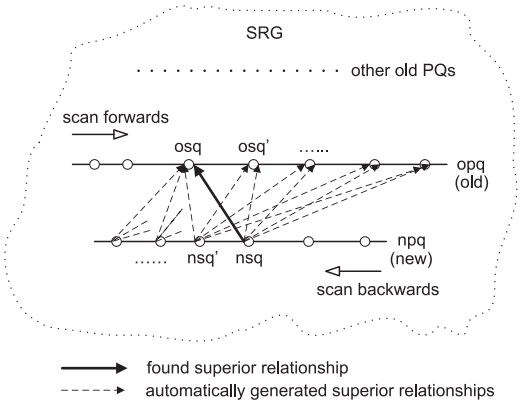

FIGURE 3. Superior relationships automatically generated in Stage 1 of AddtoSRG1().

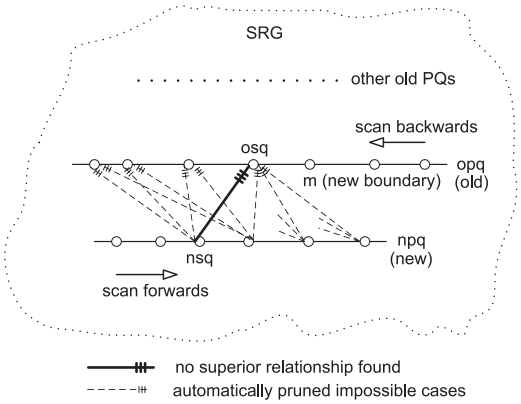

FIGURE 4. Impossible superior relationships automatically pruned in Stage 1 of AddtoSRG2().

inferior relationship is opposite to its superior counterpart. With this observation in mind, the algorithm behaves in a similar way.

In contrast to Algorithm 3.2, the pruning-based SRG construction algorithm applies the following two heuristic rules to eliminate the pairs of SQs that cannot have superior or inferior relationships, i.e. considering impossible cases rather than possible cases.

Heuristic Rule 3: If there exists no edge from sq_i to sq_j (sq_i, sq_j are two SQs $\notin$ the same PQ), then there exists no edge from sq_i to any sq_k if sq_k satisfies the following conditions: (1) $k < j$; (2) $sq_k, sq_j \in$ the same PQ.

Heuristic Rule 4: If there exists no edge from sq_i to sq_j(sq_i, sq_j are two SQs $\notin$ the same PQ), then there exists no edge from any sq_k to sq_j if sq_k satisfies the following conditions: (1) $k > i$; (2) $sq_k, sq_i \in$ the same PQ.

The details of the algorithm are given below.

REVISED ALGORITHM SEGMENT 3.3. **Pruning-Based AddtoSRG2(*npg*, *srg*)**
Input: (1) new progressive query (*npg*); (2) superior relationship graph (*srg*).
Output: revised superior relationship graph (*srg*).
Method:
1. **if** *srg* is empty **then** startempty = true;
2. **else** startempty = false **end if**;
/* Adding an isolated PQ *npq* into *srg* */
3. add the node and the node-identifier pair for each SQ of *npq* into sets *V* and *B* of *srg*, respectively;
4. add an edge from each SQ of *npq* to its immediate subsequent SQ (if any) of *npq* into edge set *E* of *srg*;
5. **if** not startempty **then**
/* Stage 1: finding external superior relationships */
6. **for** each progressive query *opq* in *srg* **do**
7. let $m = 1$;
8. **for** each SQ *nsq* of *npq* from the first to the last **do**
9. **for** each SQ *osq* of *opq* from the last to the *m*-th **do**
10. **if** there exists a superior relationship from *nsq* to *osq* **then**
11. add an edge from *nsq* to *osq* into edge set *E* of *srg*;
12. **else**
13. let *m* = index number of *osq* in *opq* + 1;
14. break;
15. **end if**
16. **end for**
17. **end for**
18. **end for**
/* Stage 2: finding external inferior relationships */
19. **for** each progressive query *opq* in *srg* **do**
20. exchange the roles of *opq* and *npq* in lines 7 - 17 to find the superior relationships from an SQ in *opq* to an SQ in *npq*;
/* i.e., finding the inferior relationships from an SQ in *npq* to an SQ in *opq* */
21. **end for**
22. **end if**.

Lines 1–4 are the same as those in Algorithm 3.2. There are also two stages in this algorithm. In stage 1, the algorithm considers one old (existing) PQ in the SRG at a time (line 6). It then scans the SQs of the new PQ forwards and the SQs of the old PQ under consideration backwards and examines each pair of SQs from the two PQs (lines 8–9). If there exists a superior relationship between the pair, an edge connecting the corresponding nodes is added into the SRG (lines 10–11). Otherwise, the algorithm prunes the remaining SQs of *opq* (Heuristic Rule 3) and resets the scan boundary of the SQs in the old PQ under consideration (Heuristic Rule 4). Figure 4 illustrates the ideas of pruning in this stage. In stage 2, the algorithm behaves similarly except that the new PQ and the old PQ under consideration play the opposite roles.

As an illustration, let us consider the example in Fig. 1. Assume that we already have pq_1 (containing sq_1, sq_2 and sq_3) and pq_2 (containing sq_4, sq_5 and sq_6) in the SRG. Our goal is to add pq_3 (containing sq_7 and sq_8) into the graph. Both algorithms first add the nodes, node-identifier pairs and internal edges for pq_3 into the graph. In the superior stage, the algorithms find all the out-going edges (representing superior relationships) from sq_7 or sq_8 to other nodes. In the inferior stage, the algorithms find all the incoming edges (representing inferior relationships) from other nodes to sq_7 or sq_8.

For Algorithm 3.2, in the first iteration, we pick up pq_1 from the graph and consider its SQs in the ascending order (from sq_1 to sq_3) while we consider SQs from pq_3 in the descending order. For the first pair $[sq_8, sq_1]$, we find that there is no superior

relationship from sq_8 to sq_1. We then move to consider the pair $[sq_8, sq_2]$. There exists no superior relationship either. Now we consider the pair $[sq_8, sq_3]$. In this case, we find a superior relationship here. We add an edge from sq_8 to sq_3. According to Heuristic Rule 1, another edge from sq_7 to sq_3 is automatically added. In such a way, we continue to process-remaining node pairs: $[sq_7, sq_1]$, $[sq_7, sq_2]$, $[sq_7, sq_3]$, but find no edges for the first two pairs and find that an edge already existed for the third pair. In the second iteration, we handle pq_2 in the same way and find the edges from sq_7 to sq_6 and sq_8 to sq_6. In the inferior stage, we add the incoming edges for sq_7 or sq_8 into the SRG. The details are omitted here due to the space limitation.

For Algorithm 3.3, in the first iteration, we pick up pq_1 from the graph and consider its SQs in the descending order (from sq_3 to sq_1) while we consider SQs from pq_3 in the ascending order. For the first pair $[sq_7, sq_3]$, there is a superior relationship from sq_7 to sq_3. So we add an edge from sq_7 to sq_3 and move to consider the pair $[sq_7, sq_2]$. There is no superior relationship in this case. According to Heuristic Rule 3, we remove $[sq_7, sq_1]$ from consideration, and according to Heuristic Rule 4, we remove $[sq_8, sq_2]$ and $[sq_8, sq_1]$ from consideration. We then directly move to consider pair $[sq_8, sq_3]$ and add an edge from sq_8 to sq_3, since such a superior relationship exists. In the second iteration, we handle pq_2 in the same way. We find edges from sq_7 to sq_6 and sq_8 to sq_6. In the inferior stage, we add the incoming edges for sq_7 or sq_8 into the SRG. The details are omitted here.

Assume that an SRG is composed of N PQs and each PQ is formulated by m SQs. When applying either the generating-based algorithm or the pruning-based algorithm to construct the SRG, the worst-case time complexity (i.e. the number of pair-wise SQ comparisons) is $O(N*(N-1)*m^2) = O(N^2*m^2)$, and the best-case time complexity is $O(N*(N-1)) = O(N^2)$. In general, the time complexity of constructing the SRG by applying either algorithm is between these two complexities. Since the complexities are polynomial, the algorithms are efficient.

To compare the two algorithms, let us consider two different situations, i.e. the given SRG is a dense graph or a sparse graph. In the dense graph case, Algorithm 3.2 could automatically generate many edges by applying Heuristic Rules 1 and 2. In this case, Algorithm 3.2 is more efficient. In the sparse graph case, Algorithm 3.3 efficiently prunes many useless pairs without checking them individually. In this case, Algorithm 3.3 is better. As a result, two algorithms can be used in different situations. This observation is validated through experiments reported in Section 4.

3.3. Weight checking

As mentioned before, the candidates for materialized views in our technique are those executed SQs from user PQs. After the current SQ for a given PQ is executed, we need to decide whether its result should be saved as a materialized view. The following strategy is adopted in our technique for this decision. The SRG provides the necessary information.

For a given SQ x, a node y in the SRG that satisfies the following conditions is searched for:

(1) The query represented by node y is an inferior of x.
(2) Node y has a sufficient weight (i.e. greater than a given threshold).

If such a node exists, x (its result) is selected as a materialized view.

As we know, the weight of a node in the SRG represents the benefit of materializing this node (i.e. how many SQs from historical PQs can be evaluated by using the result of the node). The above condition (1) ensures that any query that can benefit from node y can also benefit from x. Condition (2) guarantees a sufficient benefit.

The algorithm to search for node y can also utilize Heuristic Rule 3 to improve the search performance. It runs as follows:

REVISED ALGORITHM SEGMENT 3.4. **Checkweight**(srg, csq)
Input: (1) superior relationship graph srg; (2) current step-query csq.
Output: true or false.
Method:
1. **if** srg is empty **then**
2. return false;
3. **else**
4. **for** each progressive query pq in srg **do**
5. **for** each step-query sq of pq from the last to the first **do**
6. **if** sq is an inferior of csq **then**
7. $weight$ = number of out-going edges of sq;
8. **if** $weight$ exceeds a given threshold **then**
9. return true;
10. **end if**
11. **else** break **end if**
12. **end for**
13. **end for**
14. return false
15. **end if**.

In the algorithm, if it is found that no information is available in the SRG yet, the given SQ is not selected for materialization (lines 1–2). Otherwise, it checks each SQ in every PQ in the given SRG to see whether any of them satisfies Conditions (1) and (2) discussed above (lines 4–14). If so, return true (line 9). Otherwise, return false (line 14). Heuristic Rule 3 is applied to prune impossible cases (line 11).

3.4. Storage structure and management of materialized view set

As mentioned earlier, the materialized views and their relevant information (e.g. associated SQs and access frequencies) are kept in a SMVs. However, how to efficiently manage and search for the SMV becomes an important issue.

3.4.1. Storage structure

A straightforward way to implement the SMV is to store materialized views in a linear queue. A new materialized view is always added to the end of the queue. Thus, when an SQ to be evaluated arrives, the system has to scan the view set sequentially to search for an appropriate view to use for the SQ. Clearly, if the number of views in the SMV is large, the process to find a usable view can be slow, yielding a low system performance. On the other hand, the views in the SMV may have superior–inferior relationships among themselves, the linear structure cannot guarantee that the first usable view found is the best one for the given SQ. For example, assume that A and B are materialized views in the SMV, A's associated query is superior to B's associated query, and B's query is superior to the given SQ. If A contains 10 000 tuples and B contains 100 tuples, B is clearly a better view to use for the SQ than A. However, in the linear structure, if A is placed before B, the sequential scanning method may return A as a chosen view unless the entire queue is examined.

To overcome the limitations of the linear storage structure, we introduce a new storage structure, called the RLS, to store and manage the materialized views in order to improve the view-searching performance and quality.

In our new storage structure RLS, we classify views into four types[2]:

Type 1*: top-view.* A top-view satisfies the following conditions: (1) there exists no other view in the SMV that is superior to this view and (2) there exists at least one other view in the SMV that is inferior to this view.

Type 2*: middle-view.* A middle-view satisfies the following conditions: (1) there exists at least one other view in the SMV that is superior to this view and (2) there exists at least one other view in the SMV that is inferior to this view.

Type 3*: bottom-view.* A bottom-view satisfies the following conditions: (1) there exists at least one other view in the SMV that is superior to this view and (2) there exists no other view in the SMV that is inferior to this view.

Type 4*: independent-view.* An independent-view satisfies the following condition: there exists no other view in the SMV that is superior or inferior to this view.

As a result, four view sets (i.e. the top-view, middle-view, bottom-view and independent-view sets) are maintained within the SMV. Each view set is represented by a linked list.

For the storage structure RLS of the SMV, we also use the following concepts[3]. Node A is called a *direct parent node* of node B if the following conditions are satisfied: (1) A is superior to B and (2) there exists no node C that is superior to B and inferior to A. A *direct child node* A of node B can be defined in a similar way. Node A is called an *ancestor node* of node B if A is superior to B (allow transitive superior relationships). Node A is called a *descendant node* of node B if A is inferior to B (allow transitive inferior relationships). Two nodes A and B are *equivalent* if A is both superior and inferior to B. Note that we only need to keep one view/node among its equivalents in the SMV.

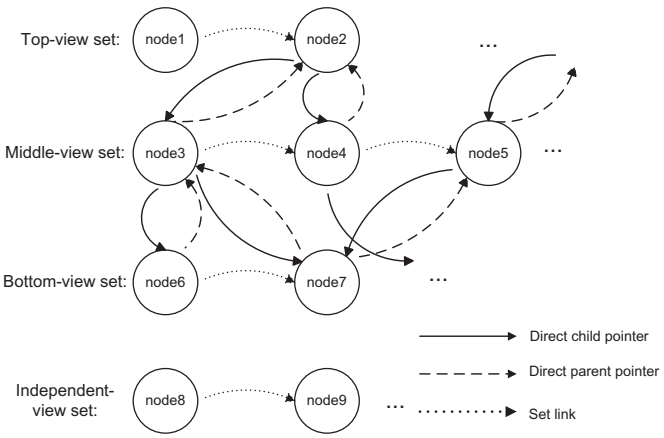

FIGURE 5. An example of the storage structure RLS of the SMV.

ID	SQ	NV	PPS	CPS	FC	STR	dataP

FIGURE 6. The data structure of each node in the SMV.

Figure 5 shows an example of the storage structure RLS of the SMV. In the figure, each node represents a view, which belongs to one of the four view sets. Nodes are connected by three types of links. Dotted links are used to connect views in the same view set. Dash links are used to represent direct (parent) superior relationships, while solid links are used to represent direct (child) inferior relationships. In other words, if node A is a direct parent of node B, then a solid link from A to B is assigned and, at the same time, a dashed link from B to A is also assigned.

In the RLS of the SMV, each node (view) has a special data structure (see Fig. 6) to keep the relevant information, which includes the node id (ID) to identify the node, the associated SQ expression for the represented view, the next view pointer (NV) to point to the next node in the same (top, middle, bottom or independent) set, the direct parent pointer set (PPS) to store the addresses/pointers of all the direct parent nodes of this node, the direct child pointer set (CPS) to store the addresses/pointers of all the direct child nodes of this node, a frequency counter (FC) to indicate the use frequency of the represented view, a STR to keep the previously discovered relationships and the address/pointer of the view (DataP) to point to the materialized view data.

When a new view/node N (corresponding to the current SQ) is to be added to the SMV, it is compared with the

[2]In the remaining discussion, we say a view is superior–inferior to another view if their associated queries have the corresponding superior–inferior relationship.

[3]In our discussion, we use terms 'view' and 'node' (in the SMV) interchangeably.

TABLE 1. Discovered superior–inferior relationship indicator.

N'.STR.REL value	meaning
00	N' has no relationship with M
01	N' is superior (but not inferior) to M
10	N' is inferior (but not superior) to M
11	N' is equivalent (both superior and inferior) to M

existing views/nodes in the SMV to discover its superior–inferior relationships with them. To improve the processing performance, as done before, we apply heuristic rules to automatically derive new relationships with more nodes in the SMV once a relationship with one node is discovered. We also try to avoid a duplicate comparison if the relationship of N with an existing node has already been discovered or derived previously. STR is a temporary storage for an existing node N' to record the previously discovered or heuristic-derived superior–inferior relationships with a new node being inserted. STR consists of a node id (ID) and an indicator (REL). The node id identifies the node M (i.e. N or a previously inserted node) with which the relationship(s) has been discovered/derived previously. The indicator is a two-bit binary value, where the lower bit indicates the existence of an inferior relationship from N' to M and the higher bit indicates the existence of a superior relationship from N' to M. The possible values of REL and their meanings are summarized in Table 1.

3.4.2. *RLS storage structure construction*

The following heuristic rules are applied by the algorithm to construct the SMV with the RLS structure:

Heuristic Rule 5 : If new node (view) N is superior to a node N' in the SMV, then N is superior to all descendant nodes of N'. If N is not superior to a node N' in the SMV, then N cannot be superior to any ancestor node of N'.

Heuristic Rule 6 : If new node (view) N is inferior to a node N' in the SMV, then N is inferior to all ancestor nodes of N'. If N is not inferior to a node N' in the SMV, then N cannot be inferior to any descendant nodes of N'.

Heuristic Rule 5 is similar to Heuristic Rules 1 and 3, while Heuristic Rule 6 is similar to Heuristic Rules 2 and 4. The only difference is that the ancestor and descendant nodes of a given node from the SMV in Heuristic Rules 5 and 6 may not belong to the same PQ.

Now let us discuss how to construct the SMV with the aforementioned RLS storage structure. In brief, we need to consider how to insert a new view/node N into an appropriate view set, discover all the direct child nodes of N in the SMV, and find all the direct parent nodes of N in the SMV. The insertion process can be done in three stages. In the first stage, all the direct child nodes of N in the bottom-view set, the middle-view set and the top-view set are discovered. In the second stage, all the direct parent nodes of N in the bottom-view set, the middle-view set and the top-view set are found. In the third stage, all the direct parent nodes or the direct child nodes of N in the independent-view set are discovered, and N is inserted into an appropriate view set based on its discovered relationships with existing nodes in the SMV.

TABLE 2. Status flag values and their indicated status.

$status\_flag$ value	determined status
-1	nothing determined yet
0	N belongs to the independent-view set
1	N belongs to the bottom-view set
2	N belongs to the middle-view set
3	N belongs to the top-view set
4	N has no direct child in the top-view, middle-view or bottom-view set
5	N has at least one direct child found
6	N has an equivalent view found in the current SMV

During the above process, we use a status flag ($status\_flag$) to indicate the status of determining the view set to which the new node N belongs. The flag is initially set to -1. The values of this status flag and their meanings are summarized in Table 2. Values 0–3 indicate that the view set to which N belongs to has been determined; while values 4–5 indicate that only partial information, which is insufficient to determine the view set membership of N, is obtained. In fact, when the flag value is 4, there are three cases. First, N has a direct child node M in the independent-view set. In this case, N belongs to the top-view set. Note that N cannot have a direct parent node in any view set in this case. Otherwise, M could not belong to the independent-view set in the first place due to the relationship transitivity. Second, N has a direct parent node in a (any) view set. In this case, N belongs to the bottom-view set. Note that N cannot have a direct child M in the independent-view set in this case. Otherwise, it violates the fact that M belongs to the independent-view set due to the transitivity. Third, N has no relationship with any view in the current SMV. In this case, N belongs to the independent-view set. When the flag value is equal to 5, there are two cases. First, N has a direct parent node in a (any) view set. In this case, N is a middle-view node, since it also has a direct child. Second, N has no direct parent node in any view set. In this case, N belongs to the top-view set. When the flag value equals to 6, there is no need to insert N into the SMV, since it has already been represented by an existing node in the SMV.

The construction algorithm that incorporates a new view/node into the RLS storage structure of the SMV runs as follows:

REVISED ALGORITHM SEGMENT 3.5. **InsertViewIntoSMV** (N, smv)
Input: (1) new materialized view node N; (2) set of materialized views (smv)

with the RLS structure.
Output: updated smv.
Method:
1. initialize $status\_flag$ to -1 and the fields of N to NULL or $\emptyset$;
/*Stage 1: find direct child nodes of N in the bottom-view, middle-view and top-view sets */
2. **if** the bottom-view set is not empty **then**
3. **for** each node S in the bottom-view set **do**
/* find direct child nodes of N that lie on each upward path of S and try to determine the view set membership of N from bottom up */
4. $status\_flag$=AddFromBottom(N, S, smv, $status\_flag$);
5. **if** $status\_flag$ == 6 **then** /* N already has an equivalent in smv */
6. return; /* no need to insert N */
7. **end if**
8. **end for**
9. **if** $status\_flag$ == -1 **then**
/* no direct child node was found for N in the bottom-view, middle-view or top-view set */
10. set $status\_flag$=4; /* record partial information */
11. **end if**
12. **end if**
/*Stage 2: find direct parent nodes of N in the bottom-view, middle-view and top-view sets */
13. **if** $status\_flag \neq 3$ **then**
/* N has not been determined to be in the top-view set */
14. **if** the top-view set is not empty **then**
15. **for** each node T in the top-view set **do**
/* find direct parent nodes of N that lie on each downward path of T and try to determine the view set membership of N from top down */
16. $status\_flag$=AddFromTop(N, T, smv, $status\_flag$);
17. **if** $status\_flag$ == 6 **then** /* N already has an equivalent in smv */
18. return; /* no need to insert N */
19. **end if**
20. **end for**
21. **if** N is not inferior to any node T in top-view set **then**
22. **if** $status\_flag$==5 **then** /* N is known to have at least one child */
/* undetermined situation can be determined now */
23. set $status\_flag$ = 3; /* N is determined to be in the top-view set */
24. **end if**
25. **end if**
26. **end if**
27. **end if**
/*Stage 3: find direct child or parent nodes of N in independent-view set, and place N in a proper view set * /
28. **if** $status\_flag \neq 2$ **then**
/* N is not in the middle-view set */
29. **if** the independent-view set is not empty **then**
30. **for** each node W in the independent-view set **do**
31. find the relationship between N and W and record the information in $W.STR$;
/* i.e., set $W.STR.REL$ to 00, 01, 10 or 11 accordingly and $W.STR.ID = N.ID$
32. **if** N and W are equivalent **then** /* i.e., $W.STR.REL = 11$ */
33. return; /* no need to insert N */
34. **else if** N is superior to W **then** /* $W.STR.REL$ == 10 */
35. set $status\_flag$ = 3; /* i.e., N is determined to be in the top-view set */
36. move W from the independent-view set to the bottom-view set;
37. link W and N together with a direct child/parent relationship;
/* i.e., update $W.PPS$ and $N.CPS$ to indicate W is a direct child of N */
38. **else if** N is inferior to W **then** /* $W.STR.REL$ == 01 */
39. set $status\_flag$ = 1;
/* i.e., N is determined to be in the bottom-view set */
40. move W from the independent-view set to the top-view set;
41. link N and W together with a direct child/parent relationship;
/* i.e., update $N.PPS$ and $W.CPS$ to indicate N is a direct child of W */
42. **end if**
43. **end for**
44. **end if**
45. **if** $status\_flag$ == 4 or $status\_flag$ == -1 **then**
/* N has no relationship with any existing view node or smv is empty */
46. set $status\_flag$ = 0;
/* N is determined to be in the independent-view set */
47. **end if**.
48. **end if**
49. **if** $status\_flag$ == 0 **then**
50. put N into the independent-view set and return;
51. **else if** $status\_flag$ == 1 **then**
52. put N into the bottom-view set and return;
53. **else if** $status\_flag$ == 2 **then**
54. put N into the middle-view set and return;
55. **else** /* $status\_flag$ == 3 */
56. put N into the top-view set and return;
57. **end if**.

In Algorithm 3.5, before the first stage, the relevant fields for new node N are initialized to be ready for the node data structure in smv, and flag $status\_flag$ is initialized to -1 (line 1).

In the first stage, if the bottom-view set B is not empty, this algorithm invokes a recurive function AddFromBottom() to discover all the direct child nodes of N in smv by following the ancestor (upward) paths of each node in B. The goal is to find the largest (highest) direct child of N along each upward path. Depending on how high the algorithm can climb up along the paths, the information about the view set membership of N may be obtained. For example, if the top node of a path is found to be a direct child of N, then N is determined to be in the top-view set. The details of AddFromBottom() will be discussed later on. It is possible that N is found to be equivalent to a node in smv during the procedure (line 5). In such a case, there is no need to add N into smv and the algorithm returns (line 6). If N is found not to be superior to any node in the bottom-view set, $status\_flag$ is set to be 4 (lines 9–11). At the end of the first stage, the possible values of $status\_flag$ are 3, 4, 5, 6 (algorithm exits) and -1 (only if the bottom-view set is empty).

At the beginning of the second stage, the algorithm first checks whether N has been determined to be a top-view (line 13). If it is true (i.e. $status\_flag = 3$), the second stage is skipped, since N has no direct parent node in such a case. Otherwise, if the top-view set D is not empty (line 14), the algorithm invokes a recurive function AddFromTop() to discover all the direct parent nodes of N in smv by following the descendant (downward) paths of each node in D. The goal is to find the smallest (lowest) direct parent of N along each downward path. Depending on how low the aglorithm can go down along the paths, the information about the view set membership of N may be obtained. For example, if the lowest node of a path is found to be a direct parent of N, then N is determined to be in the bottom-view set. In conjunction with some partial information obtained from the first stage, there are more cases in which the view set membership of N can be determined. The details of AddFromTop() will be discussed later on. It is possible that N is found to be equivalent to a node

in smv during the procedure (line 17). In such a case, there is no need to add N into smv and the algorithm returns (line 18). If N is found not to be inferior to any node in the top-view set and known to have at least one direct child (from the first stage), N is determined to be in the top-view set (lines 21–25). At the end of the second stage, the possible values of $status\_flag$ are 1, 2, 3, 4, 6 (program exits) and -1 (the bottom-view set—hence, the middle-view and top-view sets as well are empty).

At the beginning of the third stage, the algorithm first checks if N has already been determined to be in the middle-view set (line 28). If it is true (i.e. $status\_flag = 2$), the third stage is skipped, since N cannot have a superior or inferior relationship with any independent-view node in such a case due to the property of an independent-view. Otherwise, if the independent-view set is not empty (line 29), the algorithm compares N with each node W in the independent-view set (lines 30–43). The relationship between N and W can be discovered only on site (lines 31), since no derived relationships exist for an independent-view. If N is equivalent to any node W in the independent-view set, the algorithm returns (lines 32–33), since there is no need to add N into smv. Note that, in such a case, N must have not been linked to any node in smv (otherwise, W would not have belonged to the independent-view set). Hence, no clean-up work is needed before the return. If N is superior to any node W in the independent-view set, N is determined to be a top-view node (lines 34–37). This is because N cannot be inferior to any node in the bottom-view set, the middle-view set or the top-view set in this case. Otherwise, W would not have belonged to the independent-view set. Similarly, if N is inferior to any node W, N is determined to be a bottom-view node (lines 38–42). After N is compared with every independent-view, if $status\_flag = 4$, it implies that N has no direct child node found in the first stage (so *status_flag* was set to 4), N has no direct parent node found in the second stage (so *status_flag* was unchanged) and N is not superior or inferior to any independent-view node in the third stage (so *status_flag* remains the same). In this case, N must be an independent-view node (line 46). If $status\_flag = -1$ at line 45, it implies that all the top-view set, the middle-view set, the bottom-view set and the independent-view set are empty. N is clearly an independent-view node in this case (line 46). At the end of stage 3, N is inserted into a proper view set in smv according to the value of determined *status_flag*.

Two invoked functions AddFromBottom() and AddFromTop() are shown as follows:

REVISED ALGORITHM SEGMENT 3.6. **AddFromBottom**(N, S, smv, $status\_flag$)

Input: (1) new materialized view node (N); (2) a compared node (S); (3) set of materialized views (smv) with the RLS structure; (4) a status flag for the view set membership determination ($status\_flag$).

Output: (1) updated $status\_flag$; (2) updated smv.

Method:

1. $superior\_relationship$ = false;
2. **if** relationship between N and S was found before **then**
   /* i.e., $S.STR.ID == N.ID$ */
3. **if** N is superior to S **then** /* i.e., $S.STR.REL$==10 */
4. $superior\_relationship$ = true;
5. **end if**
6. **else** /* relationship between N and S has never been explored before */
7. find the relationship between N and S and record the information in $S.STR$;
   /* i.e., set $S.STR.REL$ to 00, 01, 10 or 11 accordingly and $S.STR.ID = N.ID$ */
8. **if** N and S are equivalent **then** /* i.e., $S.STR.REL = 11$ */
9. set $status\_flag$ = 6;
10. clean N from smv if N was linked in smv via a direct child/parent relationship previously;
11. return $status\_flag$; /* no need to insert N */
12. **else if** N is superior to S **then** /* i.e., $S.STR.REL$ == 10 */
13. $superior\_relationship$=true;
14. propagate the superior relationship to each descendant of S;
    /* i.e., set $X.STR.ID = N.ID$ and $X.STR.REL$ = 10 for each (unset) descendant X of S */
15. **else if** N is inferior to S **then** /* i.e., $S.STR.REL$ == 01 */
16. propagate the inferior relationship to each ancestor of S;
    /* i.e., set $X.STR.ID = N.ID$ and $X.STR.REL$ = 01 for each (unset) ancestor X of S;
17. **end if**
18. **end if**
19. **if** $superior\_relationship$ = true **then** /* N is superior to S */
20. **if** S is a top-view **then**
21. set $status\_flag$ = 3; /* N is determined to be in the top-view set */
22. move S from the top-view set to the middle-view set;
23. link S and N together with a direct child/parent relationship;
    /* i.e., update $S.PPS$ and $N.CPS$ to indicate S is a direct child of N */
24. **else** /* S is a middle-view or a bottom-view */
25. **for** each direct parent node K in $S.PPS$ **do**
    /* find direct child nodes of N on each upward path from S recursively and try to determine the view set membership for N from bottom up */
26. $status\_flag$=AddFromBottom(N, K, smv, $status\_flag$);
27. **if** $status\_flag$ == 6 **then** /* N already has an equivalent in smv */
28. return $status\_flag$; / * no need to insert N */
29. **end if**
30. **end for**
31. **if** N is not superior to any direct parent node of S **then**
32. **if** $status\_flag$ == -1 **then**
33. set $status\_flag$ = 5; /* N has at least S as a direct child node */
34. **end if**
35. link S and N together with a direct child/parent relationship;
    /* i.e., update $S.PPS$ and $N.CPS$ to indicate S is a direct child of N */
36. **end if**
37. **end if**
38. **end if**
39. return $status\_flag$.

Algorithm 3.6 is used to find the direct child nodes of N in the bottom-view set, the middle-view set and the top-view set that lie on the upward paths from S and determine the membership of a view set for N from bottom up if possible. It traverses up from S in smv by recursively following the parent links of S (lines 25–26). The algorithm first identifies the relationship between N and S, which could be found previously (lines 2–5) or is discovered in the current invocation (lines 6–18). If N and S are found to be equivalent, there is no need to insert N into smv (lines 8–11 and 27–29). Note that, when such an equivalence is found (line 8), the algorithm has to clean up the possible direct child–parent links added for N from its direct child nodes discovered so far before it returns. If N is

found to be superior or inferior to S for the first time, such a relationship needs to be propagated to the descendants or ancestors of S based on Heuristic Rule 5 or 6, respectively (lines 12–18). If N is superior to S, there are two cases in which S becomes a direct child of N. The first case is when S was in the top-view set before N is added (lines 20–23), i.e. S had at least one child but no parent. After N is added, N becomes the only (direct) parent of S. In this case, it is determined that N belongs to the top-view set (line 21), and S has to be moved to the middle-view set (line 22). The second case is when N is found to be not superior to any direct parent of S (lines 31–36). Since it is unknown whether N has its own direct parent in this case, the view set membership for N cannot be determined (line 33). When the algorithm returns, *status_flag* has one of the following values: 3, 5, 6 and −1 (no direct child so far).

REVISED ALGORITHM SEGMENT 3.7. **AddFromTop**($N, T, smv, status\_flag$)
Input: (1) new materialized view node (N); (2) a compared node (T); (3) set of materialized views (smv) with the RLS structure; (4) a status flag for the view set membership determination ($status\_flag$).
Output: (1) updated $status\_flag$; (2) updated smv.
Method:

```
1. inferior_relationship = false;
2. if relationship between N and T was found before then
   /* i.e., T.STR.ID == N.ID */
3. if N is inferior to T then   /* i.e., T.STR.REL==01 */
4.  inferior_relationship = true;
5. end if
6. else   /* relationship between N and T has never been explored before */
7. find the relationship between N and T and record the information in T.STR;
      /* i.e., set T.STR.REL to 00, 01, or 11 accordingly and T.STR.ID =
N.ID */
8. if N and T are equivalent then   /* i.e., T.STR.REL = 11 */
9.  set status_flag = 6;
10.  clean N from smv if N was linked in smv via a direct child/parent
relationship previously;
11.  return status_flag;   /* no need to insert N */
12. else if N is inferior to T then   /* T.STR.REL == 01 */
13.  propagate the inferior relationship to each ancestor of T;
      /* i.e., set X.STR.ID = N.ID and X.STR.REL = 01 for each (unset)
ancestor X of T;
14. end if
15. end if
16. if inferior_relationship = true then
17. if T is a bottom-view then
18.  if status_flag != 1 then
19.  status_flag = 1;   /* N is determined to be in the bottom-view set */
20.  end if
21.  move T from the bottom-view set to the middle-view set;
22.  link T and N together with a direct parent/child relationship;
      /* i.e., update T.CPS and N.PPS to indicate T is a direct parent of N */
23. else /* T is a middle-view or top-view */
24.  for each direct child node K in T.CPS do
     /* find direct parent nodes of N on each downward path from T recursively
and determine the view set membership for N from top down */
25.  status_flag=AddFromTop(N, K, smv, status_flag);
26.  if status_flag == 6 then   /* N already has an equivalent in smv */
27.  return status_flag;   /* no need to insert N */
28.  end if
29.  end for
30.  if N is not inferior to any direct child node of T then
31.  if status_flag ==4 then   /* N is known to have no direct child */
32.  set status_flag = 1;
      /* N is determined to be in the bottom-view set */
33.  else /* status_flag = 5; i.e., N has at least one direct child */
34.  set status_flag = 2;
      /* N is determined to be in the middle-view set */
35.  end if
36.  link T and N together with a direct parent/child relationship;
      /* i.e., update T.CPS and N.PPS to indicate T is a direct parent of N */
37.  end if
38.  end if
39. end if
40. return status_flag.
```

Algorithm 3.7 is used to find the direct parent nodes of N in the bottom-view set, the middle-view set and the top-view set that lie on the downward paths from T and determine the membership of a view set for N from top down if possible. It is similar to Algorithm 3.6, except that it traverses down (instead of up) from T in smv by recursively following the direct child links of T (lines 24–25). The algorithm first identifies the relationship between N and T, which could be found previously (lines 2–5) or is discovered in the current invocation (lines 6–15). If N and T are found to be equivalent, there is no need to add N into smv (lines 8–11 and 26–28). If N is found to be inferior to T for the first time, such a relationship needs to be propagated to the ancestors of T based on Heuristic Rule 6 (lines 12–14). Note that it is impossible for a new superior relationship from N to T (i.e. $T.STR.REL = 10$) to be discovered at this time, since all possible superior relationships from N to an existing node in smv have been discovered in the first stage. If N is inferior to T, there are two cases in which T becomes a direct parent of N. The first case is when T was in the bottom-view set before N is added (lines 17–22), i.e. T had at least one parent but no child. After N is added, N becomes the only (direct) child of T. In this case, it is determined that N belongs to the bottom-view set (lines 18–20), and T has to be moved to the middle-view set (line 21). The second case is when N is found to be not inferior to any direct child of T (lines 30–37). In this case, N is determined to be in the bottom-view set (lines 31–32) or the middle-view set (lines 33–34), depending on whether a direct child of N has been found in the first stage or not. Note that *status_flag* cannot be −1 at line 33 since when this algorithm is invoked at line 16 in Algorithm 3.5, the top-view set must not be empty, which implies that the bottom-view set cannot be empty. As mentioned earlier, $status\_flag = -1$ at the end of the first stage only if the bottom-view set is empty. When the algorithm returns, $status\_flag$ may have one of the following values: 1, 2, 4, 5 and 6.

3.4.3. *Examples*

Now let us use some insertion examples to illustrate how the SMV construction algorithm works in different scenarios. Assume that we have a partially constructed SMV as shown in Fig. 7.

In Fig. 7, the nodes from $n1$ to $n5$ are the top-views, the nodes from $n6$ to $n11$ are the middle-views, the nodes from $n12$ to $n16$ are the bottom-views, and the nodes from $n17$ to $n20$ are the independent views. We use a pair (N, M) to denote that node N is superior to node M. All the superior relationships in Fig. 7 are shown as follows: ($n1$, $n9$), ($n9$, $n12$), ($n2$, $n13$), ($n3$, $n6$),

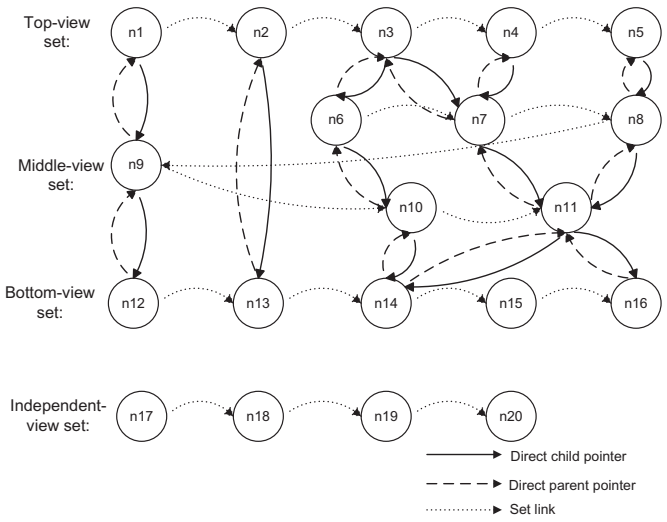

FIGURE 7. A partially constructed SMV.

$(n6, n10)$, $(n10, n14)$, $(n3, n7)$, $(n7, n11)$, $(n11, n14)$, $(n4, n7)$, $(n5, n8)$, $(n8, n11)$, $(n11, n16)$.

Suppose we want to add the results of five SQs $sq1$, $sq2$, $sq3$, $sq4$ and $sq5$ as new (materialized) views into the SMV. Assume that the superior (inferior) or equivalent relationships between the new views and the existing nodes in the SMV are as follows:

$sq1$: $(n3, sq1)$, $(sq1, n10)$, $(sq1, n14)$;

$sq2$: $(sq2, n2)$, $(sq2, n13)$;

$sq3$: $(n12, sq3)$, $(n9, sq3)$, $(n1, sq3)$;

$sq4$: $(n20, sq4)$;

$sq5$: $(sq5, n14)$, $sq5$ is equivalent to $n11$ (i.e. $(s5, n11)$ and $(n11, s5)$).

To add $sq1$ into the SMV, the bottom-view set is checked first. The algorithm wants to find those nodes to which $sq1$ is superior in the SMV. Nodes $n12$ and $n13$ are passed because $sq1$ is superior to neither of them. When the algorithm finds that $sq1$ is superior to $n14$, it traverses up through the direct parent node links of $n14$ to visit $n10$ and $n11$. Again, the algorithm finds that $sq1$ is superior to $n10$. Therefore, the algorithm continues to traverse up through the direct parent node link of $n10$ to visit $n6$. Since node $sq1$ is not superior to $n6$, the algorithm stops traversing up and links $sq1$ directly above (superior to) $n10$ ($status\_flag = 5$; lines 31–35 in AddFromBottom()). Since $sq1$ has no relationship with $n11$, the algorithm does not pursue further along that path. The algorithm then goes back to check the rest of bottom-view nodes $n15$ and $n16$. No superior relationship is found. After the bottom-view nodes have been checked, all the top-view nodes are examined one by one. The algorithm wants to find those nodes that are superior to $sq1$ in the SMV. Nodes $n1$ and $n2$ are passed because they are not superior to $sq1$. Since $n3$ is superior to $sq1$, the algorithm traverses down though the direct child node links of $n3$ to visit $n6$ and $n7$. However, neither $n6$ nor $n7$ is superior to $sq1$. Thus, the algorithm stops traversing down and links $sq1$ directly below (inferior to) $n3$ ($status\_flag = 2$; lines 33–35 in AddFromTop()). The algorithm also goes back to check the rest of top-view nodes $n4$ and $n5$, but no superior relationship is found. Finally, $sq1$ is added into the middle-view set and the insertion process ends.

To add $sq2$ into the SMV, the same algorithm is applied. First, the bottom-view set is checked and $n13$ to which $s2$ is superior is found. The algorithm then traverses up through the direct parent link of $n13$ to visit $n2$ and finds that $sq2$ is also superior to $n2$. Since $n2$ is a top-view node, $sq2$ must be a top-view node. Hence, the algorithm moves $n2$ to the middle-view set and links $sq2$ directly above (superior to) $n2$ ($status\_flag = 3$; lines 20–23 in AddFrombottom()). The algorithm then goes back to check the rest of bottom-view nodes $n14$, $n15$ and $n16$ and finds that $sq2$ is not superior to any of them. Hence, the algorithm determines that $sq2$ is a top-view node since no node in the SMV is superior to it. As a result, the second stage is skipped. The algorithm directly checks the independent-view nodes to see whether $sq2$ is superior to any of them and finds none. Finally, $sq2$ is inserted into the top-view set and the insertion process ends.

To add $sq3$ into the SMV, the algorithm checks the bottom-view set first as before. It is found that $sq3$ is not superior to any bottom-view node. The top-view set is then checked. It is found that $n1$ is superior to $sq3$. Hence, the algorithm traverses down through the direct child node link of $n1$ to visit $n9$. It is found that $n9$ is also superior to $sq3$. Thus, the algorithm continues to traverse down through the direct child node link of $n9$ to visit $n12$. Node $n12$ is still superior to $sq3$. Since $n12$ is a bottom-view node, the algorithm determines that $sq3$ is a bottom-view node. Therefore, it moves $n12$ to the middle-view set and links $sq3$ directly below $n12$ ($status\_flag = 1$; lines 17–22 in AddFromTop()). The algorithm then goes back to check the rest of top-view nodes $n2, \ldots, n5$ and finds no superiors. After that, the independent-view set is also checked to see if there exists any independent-view node which is superior to $sq3$ and none is found. Finally, $sq3$ is put into the bottom-view set and the insertion process ends.

To add $sq4$ into the SMV, a similar work is done. First, the bottom-view set is checked. But $sq4$ is not superior to any bottom-view node. Second, the top-view set is checked. However, no top-view node is superior to $sq4$. Third, the independent-view set is checked. It is found that $n20$ is superior to $sq4$. Hence, $n20$ is moved to the top-view set and $sq4$ is linked directly below $n20$ ($status\_flag = 1$; lines 38–41 in InsertViewIntoSMV()). Finally, the algorithm puts $sq4$ into the bottom-view set and the insertion process ends.

To add $sq5$ into the SMV, the bottom-view set is checked first as before. The algorithm finds that $sq5$ is superior to $n14$. It then traverses up through the direct parent node links of $n14$ to visit $n10$ and $n11$. Node $n10$ is passed, but $n11$ is found to be equivalent to $sq5$ ($status\_flag = 6$; lines 8–11 in

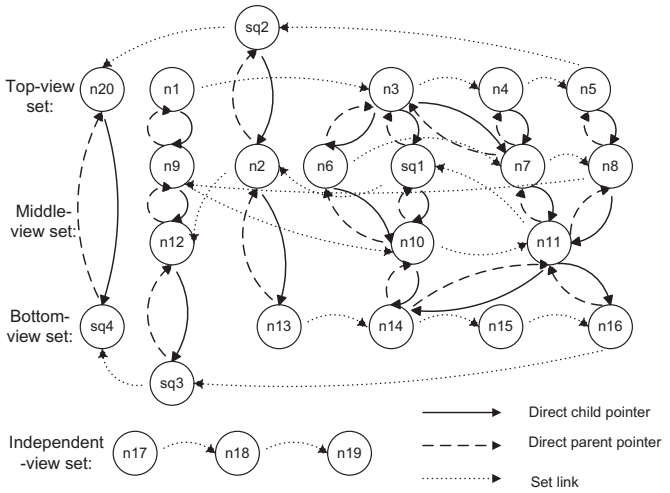

FIGURE 8. The modified SMV after inserting the nodes from $sq1$ to $sq5$.

AddFromBottom()). In this case, no need to add $sq5$ into the SMV. Therefore, the algorithm stops the insertion process and makes no change for the SMV. Figure 8 shows the SMV after inserting the nodes $sq1$, $sq2$, $sq3$ and $sq4$ ($sq5$ is not added).

Now let us consider another insertion example to illustrate how the information in the superior–inferior relationship testing record (STR) helps the algorithm improve its efficiency. As mentioned before, STR is a temporary storage for an existing node N in the SMV to record the previously discovered or heuristic-derived superior–inferior relationships with a new node being inserted. The algorithm can make use of the STRs of the existing nodes to avoid some duplicate or unnecessary comparison work.

In this example, we still consider the SMV shown in Fig. 7. Assume that the new node $sq6$ has the following superior relationships with the existing nodes in the SMV: ($sq6$, $n14$), ($n10$, $sq6$), ($n6$, $sq6$), ($n3$, $sq6$). To add $sq6$ into the SMV, in the first stage, the bottom-view set is checked. Nodes $n12$ and $n13$ are passed since they have no relationship with $sq6$. After it is found that $sq6$ is superior to $n14$, the algorithm traverses up to check $n10$ and $n11$. After two pair-wise comparisons, it is found that $sq6$ is superior to neither $n10$ nor $n11$. However, another relationship (i.e. $n10$ is superior to $sq6$) that is supposed to be found in the second stage is discovered (line 15 in AddFromBottom()). This information is saved in the STR of $n10$. Based on Heuristic 6, we also can derive (without pair-wise comparisons) the superior relationship from each ancestor node of $n10$ to $sq6$, i.e. the STRs of $n6$ and $n3$ are also updated (line 16 in AddFromBottom()). After that, the algorithm links $sq6$ directly above (superior to) $n14$ ($status\_flag = 5$; lines 31–35 in AddFromBottom()) and goes back to check the other bottom-view nodes. In the second stage, the top-view set is checked. Nodes $n1$ and $n2$ are passed since they have no relationship with $sq6$. From the STR of $n3$, it is found that $n3$ is superior to $sq6$ (the actual pair-wise comparison is avoided) and the algorithm directly traverses down to visit $n6$ and $n7$. Again, from the STR of $n6$, it is found that $n6$ is superior to $sq6$. Hence, the algorithm keeps traversing down through $n10$ until $n14$ is reached (lines 2–5 and 24–25 in AddFromTop()), then $sq6$ is linked directly below (inferior to) $n10$ ($status\_flag = 2$; lines 33–36 in AddFromTop()). The third stage is skipped since $sq6$ has already been determined to be in the middle-view set and no relationship with an independent view is possible (otherwise, the independent view could not be independent, since it would have relationships with existing top-view(s) and bottom-view(s)). Finally, the algorithm inserts $sq6$ into the SMV. From this example, we can see that, using the STR, many duplicate (previously compared) and/or unnecessary (derived) pair-wise comparisons can be avoided.

3.4.4. Materialized view set maintenance

Algorithm 3.5 can be used to insert a view into the SMV. However, the number of views that can be saved in the SMV is not unlimited. There is a space constraint for the SMV. We assume that (1) there is a space limit (SL) for the SMV and (2) the SL is large enough to hold the largest materialized view. When the SMV overflows (i.e. its size exceeds the SL), we need to delete some materialized views from it to create enough free space for accommodating a new materialized view.

The algorithm RemoveViewFromSMV(M, smv) to delete a given materialized view (node) M from the SMV smv is relatively straightforward. The main idea is to remove the relevant child–parent links for M from its direct child–parent nodes, adjust the view set memberships (after deletion) for the direct child–parent nodes when necessary, transfer the relationships of M with its direct child–parent nodes to other relevant nodes in smv when necessary and remove M from the corresponding top/bottom/independent-view set in smv. The details of this algorithm are omitted due to space limitation.

To decide which materialized views in the SMV should be replaced when space is not enough to accommodate a new materialized view, we utilize the access frequencies of materialized views in the SMV. The replacement policy is to simply remove the materialized view with the least access frequency one at a time until sufficient free space becomes available for the new materialized view. One way to efficiently find the materialized view with the least access frequency is to employ an auxiliary sorted list of the nodes in the SMV in the ascending order of their access frequencies. Note that, when a materialized view v is removed from the SMV, the corresponding PQ for the SQ associated with v (i.e. $v.sq$) needs to be checked to see whether none of its SQs is used for materialized views. If so, this PQ is removed from the SUPQs and added into the SRG. The following algorithm integrates all the previous algorithms to maintain the SMV, the SUPQ and the SRG while inserting a new materialized view into the SMV.

REVISED ALGORITHM SEGMENT 3.8. **InsertViewWithMaintenance**(N, smv, srg, $supq$)
Input: (1) new materialized view node (N) to be inserted; (2) set of materialized views (smv) with the RLS structure; (3) superior relationship graph (srg); (4) set of used PQs ($supq$).
Output: (1) updated smv with N added; (2) revised srg; (3) revised $supq$.
Method:
1. **while** smv does not have enough space to accommodate N **do**
2. find a view M to be removed from smv according to the access frequencies;
3. RemoveViewFromSMV(M, smv);
4. **if** the corresponding PQ x containing $M.sq$ has no SQ represented in smv **then**
5. remove x from $supq$;
6. AddtoSRG(x, srg);
7. **end if**
8. **end while**;
9. InsertViewIntoSMV(N, smv).

Note that the above replacement strategy could be extended to take more factors such as the sizes and ages of materialized views in smv into consideration. Such a discussion is beyond the scope of this paper.

3.5. View search

With our RLS structure for the SMV, when a new SQ sq arrives, the process to search for a materialized view that can be used to evaluate sq is efficient and effective. This is because only a small part of the SMV is usually examined and some optimization (i.e. minimizing the materialized view size) for improving the searched result is performed. For example, once a top-view v is found to be superior to sq, i.e. v is usable, an improved (smaller) usable view may be found by recursively following its direct child links until a descendant node is no longer superior to sq. On the other hand, if a top-view is found not superior to sq, all its descendants can be pruned.

4. EXPERIMENTS

To evaluate the performance of the proposed technique, we conducted extensive simulation experiments. Experiment programs were implemented in Matlab 2007 on a PC with Intel® dual core (1.5 GHz) CPU and 2 GB memory running on the Windows® Vista operating system. The experimental data set consisted of 10 external tables of randomly generated data with sizes ranging from 0 to 1000 disk blocks. Hundred random PQs were used for each experiment. Each PQ was composed of two or more SQs, where the number of steps was randomly chosen between 2 and 5. The result size of each SQ also ranged from 0 to 1000 disk blocks. The experiments were grouped into three sets. Their typical experimental results are reported in the following subsections, respectively.

4.1. Performance of dynamic materialized-view-based PQ processing approach

The first set of experiments was conducted to evaluate the efficiency of our dynamic materialized-view-based PQ processing approach (DMVPQ). The SRG and the SMVs using the RLS structure were initially set to empty. In experiments, we compared the performance between the (conventional) consecutive sequential scan-based PQ processing technique (CSSPQ) and our DMVPQ technique. PQs were processed one by one. When the execution of a PQ is completed, if no SQ in the PQ was selected as a materialized view, the PQ was added into the SRG. We maintained two parameters IPR and WPR for each node in the SRG. IPR denotes the probability with which a node has an inferior relationship with a SQ under consideration. WPR denotes the probability with which a node satisfies a weight threshold for the result of an SQ to be selected as a materialized view. Both parameters were considered together to decide whether to materialize a SQ or not. If an SQ under consideration is estimated to be beneficial, it is materialized and added into the SMVs. Two parameters SPR and SIZE are maintained for each materialized view in the set. SPR denotes the probability with which the view has a superior relationship with a SQ under consideration. SIZE denotes the size of the materialized view. Each of IPR, WPR and SPR was randomly chosen between 0 and an upper bound, without violating the definition and properties of a monotonic linear PQ. SIZE was directly acquired from the corresponding PQ. In the experiments, the pruning-based SRG construction algorithm was adopted. Since the objective of our experiments was to evaluate the performance of the DMVPQ technique, the space constraint was not considered.

In the first experiment, the upper bounds for IPR, WPR and SPR were set to 0.1, 0.5 and 0.1, respectively. Figure 9 shows the performance comparisons between the CSSPQ and the DMVPQ techniques.The x-axis represents the total number of SQs executed in the system, and the y-axis represents the I/O cost (i.e. the number of disk block accesses). From the figure, we can see that the two performance curves are very close to each other when the number of SQs processed is small. The performance of DMVPQ is increasingly better than that of CSSPQ when the number of SQs increases. The reason for this

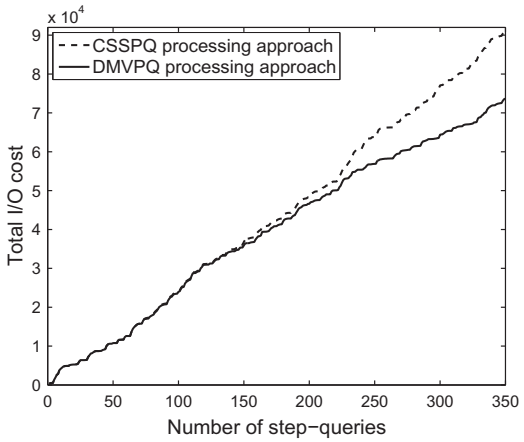

FIGURE 9. Performance comparison between DMVPQ and CSSPQ.

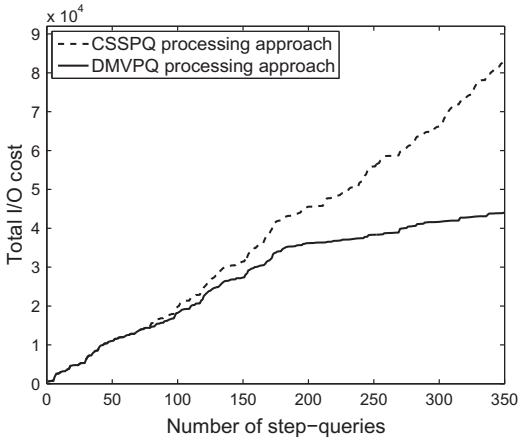

FIGURE 10. Performance comparison between DMVPQ and CSSPQ with IPR being changed to 0.3.

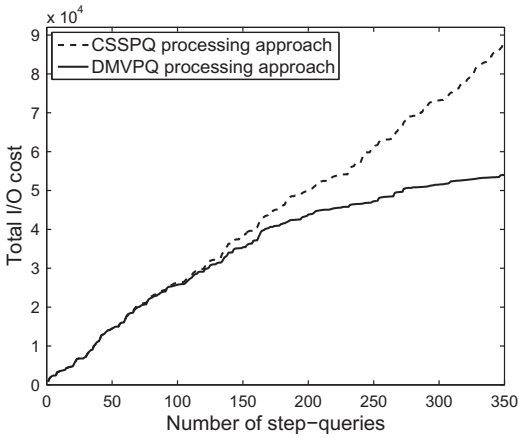

FIGURE 11. Performance comparison between DMVPQ and CSSPQ with SPR being changed to 0.3.

is as follows. At the beginning, both SRG and SMV are empty—no view could be utilized to improve the query performance. As more and more PQs are executed, the SRG and MVC grow larger and larger. In other words, more and more materialized views become available for improving the query performance. As a result, the performance of DMVPQ is significantly improved.

In the second experiment, we increased the upper bound for parameter IPR to 0.3 and kept the other parameters unchanged. The experimental results are shown in Fig. 10. From the figure, we can see that the performance of DMVPQ is significantly improved. The reason for this is that IPR plays an important role in deciding whether to materialize the result of a SQ. A larger upper bound for IPR implies that a SQ has a higher chance of being materialized. Hence, the SMV grows faster, and the subsequent queries have more views to utilize to improve their performance.

Another crucial factor to affect the query performance is parameter SPR. In the third experiment, we changed the upper bound for SPR to 0.3 and kept the other parameters unchanged. Experimental results are shown in Fig. 11. A significant performance increase for DMVPQ is also observed. The reason for this improvement is that SPR is the factor to determine whether a materialized view would be usable for a SQ under consideration. A larger upper bound for SPR implies that a materialized view has a better chance of being usable for a given SQ. In other words, a SQ has more available views to utilize to improve its performance.

In the next experiment, we considered various upper bounds for IPR ranging from 0.1 to 0.9 and kept other parameters unchanged. The performance curve is shown in Fig. 12. From the figure, we can clearly see that the performance is improved as IPR increases. We also conducted another experiment for various upper bounds for SPR ranging from 0.1 to 0.9 and kept other parameters unchanged. The experimental results are shown in Fig. 13. A similar performance pattern is also observed.

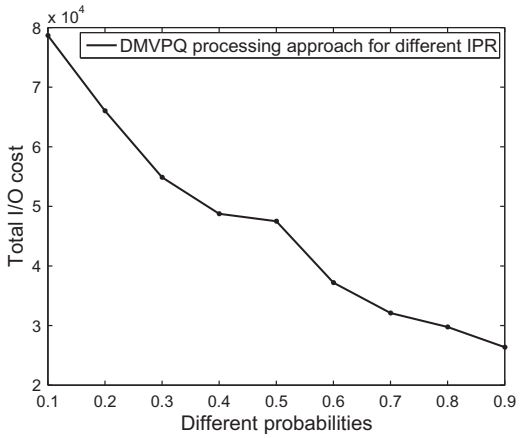

FIGURE 12. Performance change with different IPRs for DMVPQ.

The results of the first set of the experiments demonstrate that our DMVPQ technique is quite promising in improving the performance for processing monotonic linear PQs.

4.2. Performance of SRG construction methods

The second set of experiments was conducted to compare the performance behaviors of the generating-based method and the pruning-based method for constructing a SRG. The SRG was initially set to empty. The PQs were processed one by one. When a new SQ was added into the SRG, we needed to find all the superior or inferior relationships between the new SQ and the SQs in the SRG. As mentioned before, a straightforward way to construct an SGR is to perform the pair-wise comparisons between the new SQ to be added and each existing SQ in the SRG. But the cost of this way is usually very high, which led us to develop the generating-based method and the pruning-based method to avoid some unnecessary comparisons. In the experiments, we wanted to compare the performance of

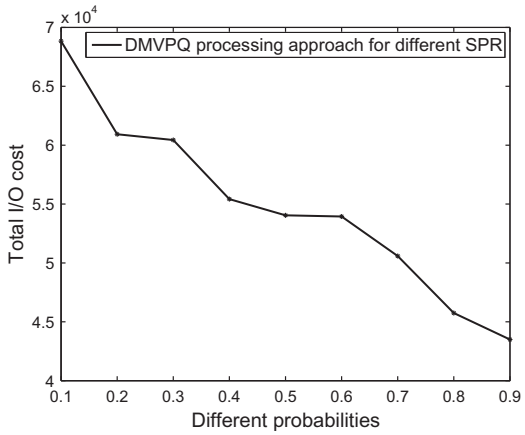

FIGURE 13. Performance change with different SPRs for DMVPQ.

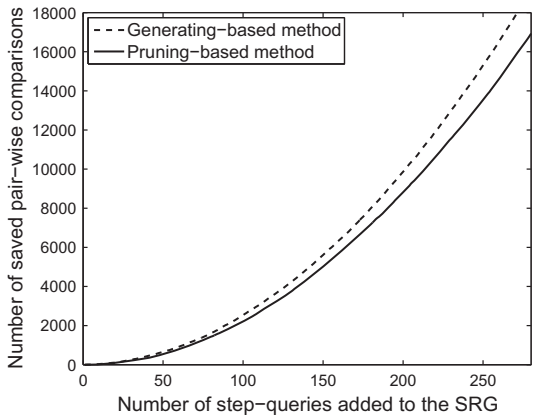

FIGURE 14. Comparison of saved costs between the generating-based method and the pruning-based method with $IPR = 0.7$.

these two methods so as to identify the scenarios where one method could be better than the other. The performance was measured in terms of the cost (pair-wise comparisons) saved over the straightforward pair-wise comparison method. We also maintained the IPR for each node in the SRG, which represents the probability of this node being inferior to a new SQ to be added.

In the first experiment, we set the upper bound of IPR to a relatively high value 0.7, which led to a high chance of the existing nodes in the SRG having a (inferior) relationship with a new SQ to be added. Hence, the resulting SRG was a dense (in terms of edges) graph. Figure 14 shows the comparison of the saved costs between the generating-based method and the pruning-based method. From the figure, we can observe that the former method outperforms the latter method to construct the SRG in such a case. This is because the generating-based method has a better chance of automatically deriving/generating more relationships (edges) in a dense graph to save many (pair-wise) comparisons. The larger the SRG, the more savings the generating-based method could achieve.

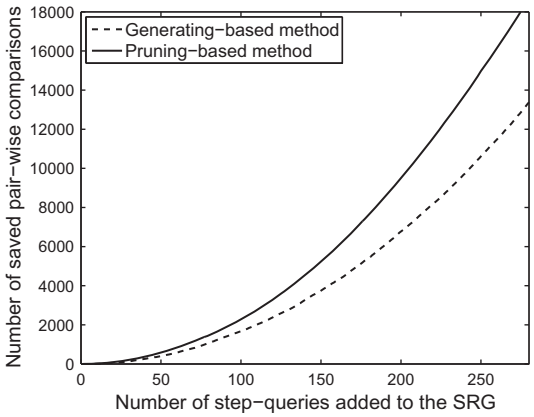

FIGURE 15. Comparison of saved costs between the generating-based method and the pruning-based method with $IPR = 0.3$.

In the second experiment, we set the upper bound of IPR to a relatively low value of 0.3. The resulting SRG was a sparse graph. Figure 15 shows the comparison of the saved costs between the generating-based method and the pruning-based method. We can see that the pruning-based method performed better than the generating-based method in this case. This is because the pruning-based method has a better chance to automatically eliminate/prune impossible relationships (edges) in a sparse graph to save many comparisons. The larger the SRG, the more savings the pruning-based method could obtain.

The previous two experiments demonstrate that both the generating-based method and the pruning-based method can save an increasing amount of cost as the SRG grows. The generating-based method is better for a dense graph, while the pruning-based method is better for a sparse graph, as we predicted in Section 3.2.

4.3. Performance of view search using new SMV storage structure

The third set of experiments was conducted to examine the view-searching performance for the SMVs using our new storage structure RLS. The SMV was initially set to empty. Four different materialized view sets (top-view set, middle-view set, bottom-view set and independent-view set) were maintained. PQs were processed one by one. For each SQ of the current PQ, the SMV was searched to find a usable view that could be used to answer the SQ. After a SQ was executed, its result had a chance to be kept as a (materialized) view and stored in the SMV. The new storage structure RLS for the MVS was built by using Algorithm 3.5. We maintained three parameters UPR, $SCPR$ and WPR for each SQ. UPR denotes the probability of a view in the top-view set being a usable view for the given SQ; $SCPR$ denotes the probability for a direct child node to be a usable view replacing its direct parent node for the given SQ; WPR denotes the probability of a SQ result being kept as a materialized view. Two additional parameters $SUPR$

and $INPR$ were also maintained in our experiments. $SUPR$ denotes the probability of a view being superior to another view in the SMV, and $INPR$ denotes the probability of a view to be inferior to another view in the SMV. UPR, $SCPR$, WPR, $SUPR$ and $INPR$ were randomly chosen between 0 and their respective upper bound. In the experiments, the upper bounds for UPR, $SCPR$, WPR, $SUPR$ and $INPR$ were set to 0.3, 0.9, 0.3, 0.3 and 0.3, respectively.

To search for a usable view in the SMV for a given SQ, we examined two searching strategies: the fastest time strategy (FTS) and the best result strategy (BRS). The FTS returns the first usable view found in the SMV for the given SQ, while the BRS returns the best usable view (i.e. the smallest one) in the SMV for the given SQ. We compared the performance between the (conventional) sequential search method with the MVS organized as a linear queue (SSMVS) and the superior (inferior) relationship based search method with the SMV organized using our new RLS structure (SRMVS). Various scenarios were considered.

In the first experiment, we used the fastest time strategy for both the SSMVS and the SRMVS. For the SSMVS, the search returned the first usable view (if any) in the linear queue. For the SRMVS, the search first found the first usable view (if any) in the top-view set and then recursively followed the direct child (inferior) link of the found view to see whether a better (smaller) usable view could be obtained. Hence, the SRMVS returned an improved usable view (if possible) over the first usable view found in the top-view set. Note that neither the SRMVS nor the SSMVS based on the FTS can guarantee that the best usable view in the SMV is found. Figure 16 shows the comparison of view quality (in terms of view size) between the SSMVS and the SRMVS based on the FTS. The x-axis represents the total number of SQs processed in the system. For each SQ, a usable view may be returned from the SMV to answer the query. The y-axis represents the total size of the returned views. The smaller the total size, the higher the view quality achieved. From the figure, we can see that two curves are very close at the beginning. The view quality obtained from the SRMVS is increasingly better than that from the SSMVS as the number of SQs increases. The reason for this is as follows. At the beginning, the SMVs were empty for both the SSMVS and the SRMVS. Thus, for the first several SQs, no view could be used to answer them and the total view size was 0 for both the SSMVS and the SRMVS. As the number of SQs increased, more and more SQ results were saved as materialized views in the SMVs, which could be used to answer the following SQs and the total view size started to increase. As the SMV grew, the SRMVS had a better chance of returning an improved usable view (via the maintained superior–inferior relationships) over the first usable view found in the top-view set. Therefore, the view quality curve of the SRMVS becomes better and better compared with that of the SSMVS.

Figure 17 shows the comparison of view-searching costs between the SSMVS and the SRMVS based on the FTS. The x-axis still represents the number of SQs processed. The y-axis represents the total number of views searched. From the figure, we can see that the searching cost of the SRMVS is always smaller than the that one of the SSMVS. The reason for this is as follows. The SRMVS based on the FTS typically can save some searching cost by pruning the middle-views and bottom-views that are descendants of a non-usable top-view, while the SSMVS based on the FTS has to search for all the views in the SMV in the worst case.

In the second experiment, we applied the FTS for the SSMVS and the BRS for the SRMVS. We still compared both the view quality and the searching costs between the SSMVS and the SRMVS. Figure 18 shows the comparison of view quality between the SSMVS based on the FTS and the SRMVS based on the BRS. From the figure, we can see that the view quality from the SRMVS is dramatically improved by using the BRS instead of the FTS. The reason for this is as follows. Using the SSMVS based on the FTS, once a usable view is found

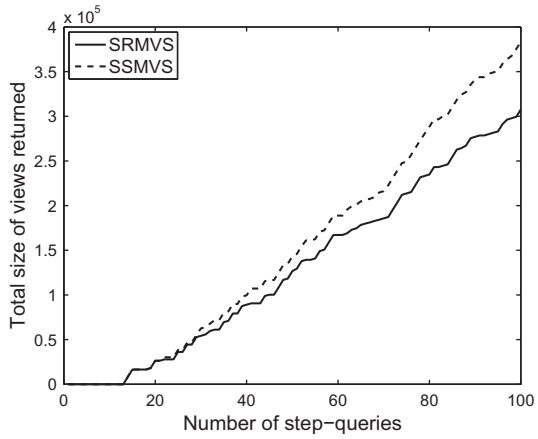

FIGURE 16. Comparison of view quality between SSMVS and SRMVS based on FTS.

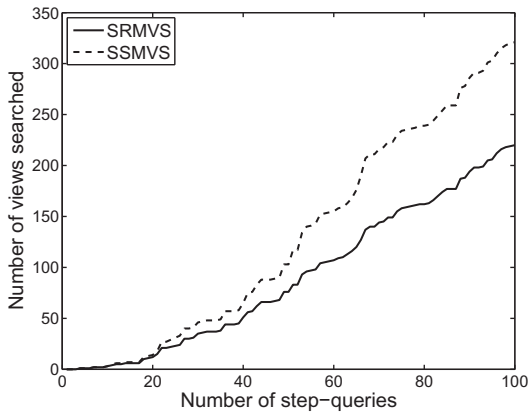

FIGURE 17. Comparison of view-searching costs between SSMVS and SRMVS based on FTS.

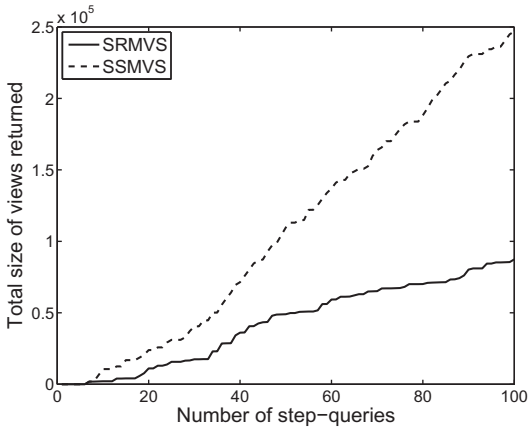

FIGURE 18. Comparison of view quality between SSMVS based on FTS and SRMVS based on BRS.

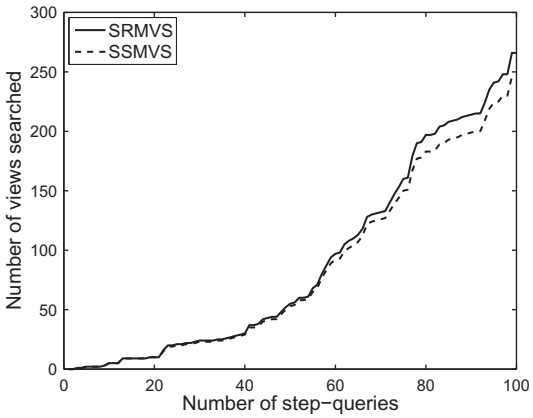

FIGURE 19. Comparison of view-searching costs between SSMVS based on FTS and SRMVS based on BRS.

in the SMV, the view is returned, which does not guarantee the quality. On the other hand, the SRMVS based on the BRS examines every usable top view and its descendants as well as every usable independent-view until the best (smallest) usable view is found. Hence, it guarantees that the best usable view in the SMV is returned for the given SQs.

Figure 19 shows the comparison of view-searching costs between the SSMVS based on the FTS and the SRMVS based on the BRS. From the figure, we can find that the searching cost of the SRMVS based on the BRS is a little bit higher than that of the SSMVS based on the FTS. This is because the SRMVS based on the BRS has to check all the usable views in the top-view set and their descendants as well as the usable views in the independent-view set. On the other hand, the SSMVS based on the FTS only needs to return the first usable view found in the linear queue of the SMV. If there are many usable views in the SMV, the SSMVS does not incur much cost. Hence, the searching cost of the SRMVS based on the BRS is usually higher than that of the SSMVS based on the FTS. However, due to

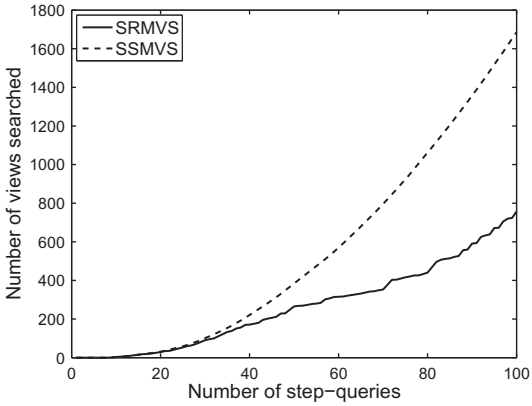

FIGURE 20. Comparison of view-searching costs between SSMVS and SRMVS based on BRS.

the capability of the SRMVS for pruning the descendants of non-usable views, the cost difference between the two methods is small.

In the third experiment, both the SRMVS and the SSMVS adopted the BRS. In this case, both methods had the same view quality, since they both guaranteed that the best usable view for a given SQ was returned. Hence, we compared only their view-searching costs. Figure 20 shows the comparison of searching costs between the SSMVS and the SRMVS based on the BRS. We observed that the searching cost of the SSMVS based on the BRS was much higher than the one of the SRMVS based on the BRS. This is because the former method has to check all the views to find the best usable view for a given SQ. On the other hand, the descendants of non-usable views are removed (pruned) from consideration by the SRMVS based on the BRS.

Our experiments demonstrate that the SRMVS based on either the FTS or the BRS is quite promising in efficiently searching for quality usable views for given SQs, compared with the SSMVS.

5. CONCLUSIONS

There is an increasing demand for processing PQs from various application domains. Existing DBMSs were not designed to process such queries efficiently. In this paper, we have proposed a novel dynamic materialized-view-based technique to process a special type of PQ called the monotonic PQ. The main contributions of the paper are summarized as follows:

(i) We have presented a PQ processing procedure to dynamically select the results of executed SQs as materialized views and apply the materialized views to efficiently process other SQs. A framework incorporating the procedure is described.
(ii) We have introduced a SRG, which is constructed for a set of historical PQs. The SRG captures the

superior–inferior relationships among SQs in these historical PQs and is used to estimate the benefit of keeping the result of a SQ as a materialized view. Two algorithms, i.e. generating-based and pruning-based, to efficiently construct an SRG are proposed. The former automatically generates more edges once one edge is found, while the latter effectively prunes the impossible cases once an edge is not found. Both algorithms apply heuristics that are derived from the properties of monotonic linear PQs. The algorithm that uses the SRG to determine whether the result of a given SQ should be kept as a materialized view is also suggested.

(iii) We have also suggested a storage structure, called the RLS, for managing the SMVs. The RLS maintains the superior–inferior relationships among the materialized views and classifies the materialized views into four groups. It supports efficient search for a good usable view to evaluate a given SQ. The efficient algorithms for managing and searching for the SMV with the RLS structure are discussed. A strategy to handle the space constraint and several heuristics to improve efficiency are incorporated.

(iv) We have conducted extensive simulation experiments to evaluate the performance of our proposed techniques for various issues. The experimental results demonstrate that the dynamic materialized-view-based approach is quite promising in processing the monotonic linear PQs. Its performance improvement over the conventional consecutive sequential scan-based query processing approach is increasingly larger as the number of processed queries increases. The empirical study of the performance of the two SRG construction algorithms shows that both algorithms outperform the straightforward construction method. The generating-based algorithm is more suitable for a dense SRG, while the pruning-based algorithm is more suitable for a sparse SRG. The performance evaluation of the SMV using our new storage structure RLS demonstrates that this new structure can support efficient search for a usable view with a good quality for a given SQ.

Our work is just the beginning of the research on optimizing PQs. Further study is required to completely solve the relevant issues. Our ongoing and future work includes investigating materialized-view-based techniques to process other types of PQs such as multiple-input linear PQs and non-linear PQs, developing a multi-layered materialized view technique that utilizes storage hierarchies to efficiently process PQs, studying the issues to incorporate our developed techniques into existing DBMS and extending the PQ processing to NoSQL database settings. Some preliminary results on developing a materialized-view-based technique to process generic PQs have been reported in Zhu *et al*. [51].

ACKNOWLEDGEMENTS

The preliminary results of this work were presented at the 2010 Conference of the IBM Centre for Advanced Studies on Collaborative Research—CASCON'10 [52] (paper available at: http:// www.engin.umd.umich.edu / ∼qzhu / papers / cascon10zhu.pdf), Toronto, 1–4 November 2010.

FUNDING

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

REFERENCES

[1] Gray, J. and Szalay, A.S. (2004) Where the rubber meets the sky: bridging the gap between databases and science. *IEEE Data Eng. Bull.*, **27**, 3–11.

[2] Nambiar, U., Lud, B., Lin, K. and Baru, C. (2006) The GEON Portal: Accelerating Knowledge Discovery in the Geosciences. *Proc. ACM Int. Workshop on Web Information and Data Management (WIDM)*, Arlington, VA, USA, November 10, pp. 83–90. ACM.

[3] Nieto-Santisteban, M.A., Gray, J., Szalay, A.S., Annis, J., Thakar, A.R. and O'Mullane, W. (2005) When Database Systems Meet the Grid. *Proc. CIDR Conf.*, Asilomar, CA, USA January 4–7, pp. 154–161. ACM.

[4] Stevens, R. *et al.* (2004) myGrid and the drug discovery process. *Drug Discov. Today: BIOSILICO*, **2**, 140–148.

[5] Zhu, Q., Medjahed, B., Sharma, A. and Huang, H. (2008) The collective index: a technique for efficient processing of progressive queries. *Comp. J.*, **51**, 662–676.

[6] Comer, D. (1979) The ubiquitous B-tree. *ACM Comput. Surv.*, **11**, 121–137.

[7] Agarwal, P.K., Xie, J., Yang, J. and Yu, H. (2006) Scalable Continuous Query Processing by Tracking Hotspots. *Proc. VLDB Conf.*, Coex, Seoul, Korea, September 12–15, pp. 31–42. ACM.

[8] Babu, S. (2005) Adaptive query processing in data stream management systems. Ph.D. Dissert., Stanford University.

[9] Lim, H.-S., Lee, L.-G., Lee, M.-J., Whang, K.-Y. and Song, I.-Y. (2006) Continuous Query Processing in Data Streams Using Duality of Data and Queries. *Proc. SIGMOD Conf.*, Chicago, IL, USA, June 27–29, pp. 313–324. ACM.

[10] Mokbel, M.F. Continuous Query Processing in Spatio-Temporal Databases. *Proc. EDBT Workshops*, Heraklion, Crete, Greece, March 14–18, pp. 100–111. Springer.

[11] Antoshenkov, G. (1993) Dynamic Query Optimization in RDB/VMS. *Proc. ICDE Conf.*, Vienna, Austria, April 19–23, pp. 538–547. IEEE Computer Society.

[12] Babu, S. and Bizarro, P. (2005) Adaptive Query Processing in the Looking Glass. *Proc. CIDR Conf.*, Asilomar, CA, USA, January 4–7, pp. 238–249. online proceedings, http://www.db.cs.wisc.edu/cidr/.

[13] Kabra, N. and DeWitt, D.J. (1998) Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. *Proc. SIGMOD*, Seattle, WA, USA, June 2–4, pp. 106–117. ACM.

[14] Liu, L. and Pu, C. (1997) Dynamic query processing in DIOM. *IEEE Data Eng. Bull*, **20**, 30–37.

[15] Lu, H., Tan, K.-L. and Dao, S. (1995) The Fittest Survives: An Adaptive Approach to Query Optimization. *Proc. VLDB Conf.*, Zurich, Switzerland, September 11–15, pp. 251–262. ACM.

[16] Jorg, T. and DeBloch, S. (2008) Towards Generating ETL Processes for Incremental Loading. *Proc. IDEAS'08*, Coimbra, Portugal, September 10–12, pp. 101–110.

[17] Simitsis, A.P., Vassiliadis, P. and Sellis, T. (2005) State-space optimization of ETL workflows. *IEEE Trans. Knowl. Data Eng.*, **17**, 1404–1409.

[18] Vassiliadis, P., Vagena, Z., Skiadopoulos, S., Karayannidis, N. and Sellis, T. (2001) ARKTOS: towards the modeling, design, control and execution of ETL processes. *Info. Syst.*, **26**, 537–561.

[19] Vassiliadis, P., Simitsis, A., Georgantas, P., Terrovitis, M. and Skiadopoulos, S. (2005) A generic and customizable framework for the design of ETL scenarios. *Info. Syst.*, **30**, 492–525.

[20] Chaudhuri, S., Krishnamurthy, R., Potamianos, S. and Shim, K. (1995) Optimizing Queries with Materialized Views. *Proc. ICDE Conf.*, Taipei, Taiwan, March 6–10, pp. 190–200. IEEE Computer Society.

[21] Goldstein, J. and Larson, P.-Å. (2001) Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. *Proc. SIGMOD Conf.*, Santa Barbara, CA, USA, May 21–24, pp. 331–342. ACM.

[22] Gou, G., Kormilitsin, M. and Chirkova, R. (2006) Query Evaluation Using Overlapping Views: Completeness and Efficiency. *Proc. SIGMOD Conf.*, Chicago, IL, USA, June 27–29, pp. 37–48. ACM.

[23] Halevy, A.Y. (2001) Answering queries using views: a survey. *VLDB J.*, **10**, 270–294.

[24] Larson, P.-Å. and Yang, H.Z. (1985) Computing Queries from Derived Relations. *Proc. VLDB Conf.*, Stockholm, Sweden, August 21–23, pp. 259–269. ACM.

[25] Zilio, D. *et al.* (2004) Recommending materialized views and indexes with IBM DB2 design advisor. *Proc. ICAC*, New York, NY, USA, May 17–19, pp. 180–188. IEEE Computer Society.

[26] Akhtar Ali, M. *et al.* (2003) MOVIE: an incremental maintenance system for materialized object views. *Data Knowl. Eng. (DKE)*, **47**, 131–166.

[27] Jiang, H., Gao, D. and Li, W.-S. (2007) Exploiting Correlation and Parallelism of Materialized-View Recommendation for Distributed Data Warehouses. *Proc. ICDE Conf.*, Istanbul, Turkey, April 15–20, pp. 276–285. IEEE Computer Society.

[28] Park, C.-S., Kim, M.-H. and Lee, Y.-J. (2001) Rewriting OLAP Queries Using Materialized Views and Dimension Hierarchies in Data Warehouses. *Proc. ICDE Conf.*, Heidelberg, Germany, April 2–6, pp. 515–523. IEEE Computer Society.

[29] Arion, A., Benzaken, V., Manolescu, I. and Papakonstantinou, Y. (2007) Structured Materialized Views for XML Queries. *Proc. VLDB Conf.*, Vienna, Austria, September 23–27, pp. 87–98. ACM.

[30] Xu, W. and Ozsoyoglu, Z.M. (2005) Rewriting XPath Queries Using Materialized Views. *Proc. VLDB Conf.*, Trondheim, Norway, 30 August–2 September, pp. 121–132. ACM.

[31] Joshi, S. and Jermaine, C.M. (2008) Materialized sample views for database approximation. *IEEE Trans. Knowl. Data Eng. (TKDE)*, **20**, 337–351.

[32] Re, C. and Suciu, D. (2007) Materialized Views in Probabilistic Databases for Information Exchange and Query Optimization. *Proc. VLDB Conf.*, Vienna, Austria, September 23–27, pp. 51–62. ACM.

[33] Chirkova, R., Li, C. and Li, J. (2006) Answering queries using materialized views with minimum size. *VLDB J.*, **15**, 191–210.

[34] Lee, M. and Hammer, J. (2001) Speeding up materialized view selection in data warehouses using a randomized algorithm. *Int. J. Coop. Inf. Syst. (IJCIS)*, **10**, 327–353.

[35] Liang, W., Wang, H. and Orlowska, M.E. (2001) Materialized view selection under the maintenance time constraint. *Data Knowl. Eng. (DKE)*, **37**, 203–216.

[36] Mistry, H., Roy, P., Sudarshan, S. and Ramamritham, K. (2001) Materialized View Selection and Maintenance Using Multi-Query Optimization. *Proc. SIGMOD*, Santa Barbara, CA, USA, May 21–24, pp. 307–318. ACM.

[37] Gupta, A. and Mumick, I.S. (1995) Maintenance of materialized views: problems, techniques, and applications. *Data Eng. Bull.*, **18**, 3–18.

[38] Gupta, A., Mumick, I.S. and Subrahmanian, V.S. (1993) Maintaining Views Incrementally. *Proc. SIGMOD Conf.*, Washington, DC, USA May 26–28, pp. 157–166. ACM.

[39] Zhou, J., Larson, P.-Å. and Elmongui, H.G. (2007) Lazy Maintenance of Materialized Views. *Proc. VLDB Conf.*, Vienna, Austria, September 23–27, pp. 231–242. ACM.

[40] Larson, P.-Å. and Zhou, J. (2007) View matching for outer-join views. *VLDB J.*, **16**, 29–53.

[41] Lu, M.-C. and Wu, F. (2004) A Structure for Materialized Views of Data Warehouse with Concurrency Control. *Proc. IKE Conf.*, Las Vegas, NV, USA, June 21–24, pp. 385–391.

[42] Luo, G., Naughton, J.F., Ellmann, C.J. and Watzke, M. (2005) Locking protocols for materialized aggregate join views. *IEEE Trans. Knowl. Data Eng. (TKDE)*, **17**, 796–807.

[43] Bellatreche, L., Karlapalem, K. and Li, Q. (2000) Evaluation of Materialized View Indexing in Data Warehousing Environments. *Proc. DaWaK*, London, UK, September 4–6 , pp. 57–66. Springer.

[44] Roussopolous, N. (1982) View indexing in relational databases. *ACM Trans. Database Syst.*, **7**, 258–290.

[45] Ezeife, C.I. (2001) Selecting and materializing horizontally partitioned warehouse views. *Data Knowl. Eng.*, **36**, 185–210.

[46] Hung, M.-C., Huang, M.-L., Yang, D.-L. and Hsueh, N.-L. (2007) Efficient approaches for materialized views selection in a data warehouse. *Inf. Sci. (ISCI)*, **177**, 1333–1348.

[47] Agrawal, S., Chaudhuri, S. and Narasayya, V. (2000) Automated Selection of Materialized Views and Indexes in SQL Databases. *Proc. VLDB Conf.*, Cairo, Egypt, September 10–14, pp. 391–398. ACM.

[48] Tang, N., Yu, J.X., Ozsu, M.T., Choi, B. and Wong, K.-F. (2008) Multiple Materialized View Selection for XPath Query Rewriting. *Proc. ICDE Conf.*, Cancun, Mexico, April 7–12, pp. 873–882. IEEE.

[49] Aouiche, K., Jouve, P.-E. and Darmont, J. (2006) Clustering-Based Materialized View Selection in Data Warehouses. *Proc. ADBIS Conf.*, Thessaloniki, Hellas, September 3–7, pp. 81–95. Springer.

[50] Gupta, A. and Mumick, I.S. (2005) Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, **17**, 24–43.

[51] Zhu, C., Zhu, Q., Zuzarte, C. and Ma, W. (2011) A Materialized-view Based Technique to Optimize Progressive Queries via Dependency Analysis. *Proc. CASCON*, Toronto, Ontario, Canada, 7–10 November, pp. 60–73. ACM.

[52] Zhu, C., Zhu, Q. and Zuzarte, C. (2010) Efficient Processing of Monotonic Linear Progressive Queries via Dynamic Materialized Views. *Proc. CASCON*, Toronto, Ontario, Canada, November 1–4, pp. 224–237. ACM.