Efficient Processing of Monotonic Linear Progressive Queries via Dynamic Materialized Views *

Chao Zhu† Qiang Zhu† Calisto Zuzarte‡

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

‡IBM Canada Software Laboratory, Markham, Ontario, Canada L6G 1C7

Abstract

There is an increasing demand to process emerging types of queries, such as progressive queries (PQs), from numerous contemporary database applications including telematics, e-commerce, business intelligence, and decision support. Unlike a conventional query, a progressive query is formulated in several steps, i.e., consisting of a set of inter-related step-queries (SQ). A user formulates their SQs on the fly based on the results returned by the previous SQs. Processing such queries provides performance improvement opportunities for a database management system. In this paper, we study the efficient processing of a special type of PQ, called a monotonic linear progressive query (MLPQ). We present a technique to process such PQs based on dynamically materialized views. The key idea is to create a superior-relationship graph for step-queries from historical PQs, which can be used to estimate the benefit of materializing a current step-query. The materialized views are then used to improve the performance of future step-queries. Algorithms and strategies to create and maintain a superior-relationship graph, dynamically select materialized views (step-queries), and the search for a materialized view to process a given step-query are discussed. Experimental results demonstrate that our proposed technique is quite promising in efficiently processing this type of progressive query.

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

1 Introduction

In recent years, we have witnessed the emergence of many contemporary database applications such as telematics, e-commerce, business intelligence, and decision support. Such data-intensive applications raise new challenges to process advanced types of queries [6, 14, 15, 18]. A new type of query, called the progressive-query (PQ), was presented in [21]. It was observed that, in many applications, users routinely perform queries step by step. In each step, the query uses the result returned by the previous step. The query result is narrowed down gradually according to the user’s demands. Hence, unlike a conventional query, a PQ is formulated in several steps, i.e., consisting of a set of inter-related step-queries (SQs). A user formulates his/her SQs on the fly based on the result returned by previous SQs.

As an illustration, let us assume that a user wants to search papers from the IEEE digital library. He or she first selects papers that
are in the data mining area and published in 2009. We can imagine how large a set of papers could be returned. The user then adds a condition to narrow down the results to only those papers that are related to mining association rules. The result set is still very large. Thus the user further narrows down the results by adding another condition to search papers authored by Zhu.

The previous example demonstrates two main features of a progressive query. First, a progressive query cannot be known beforehand. Each step-query is formulated dynamically by the user. The user needs to know the result(s) of the previous step-query(ies) to determine the next step query. Second, a progressive query is frequently used to access large data sets, and the intermediate result returned from a step-query may not be held in memory.

These characteristics of progressive queries raise new challenges to process them 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 [5, 16]) that are typically created on base relations may not be directly applicable because a step-query that is not for the first step of a progressive query uses the intermediate result(s) from the previous step-query(ies). To tackle this challenge, an effective collective index technique was introduced in [21]. The main idea of this technique is to construct a special index structure to allow a collection of member indexes on an input relation of a step-query to be efficiently transformed into indexes on the result relation, which can be used to speed up the subsequent step-queries. This work was the first to address query processing issues for progressive queries.

In this paper, we present a novel materialized view technique for efficiently processing progressive queries. The key idea is to dynamically construct a superior-relationship graph for step-queries from the progressive queries that have been executed. The underlying database management system (DBMS) uses the graph to estimate the benefit of materializing the current step-query for a given progressive query. If it is beneficial, the result of the current step-query is materialized as a view. The materialized views are used to optimize the step-queries of future progressive queries. Algorithms/strategies to create and maintain a superior-relationship graph, dynamically select materialized views (step-queries), and search for a materialized view to process a given step-query are presented. Although applying materialized views to speed-up query processing has been well studied [4, 7, 8, 9, 20, 22, 11, 1, 17, 19], adopting a technique based on dynamically materialized views for progressive queries is our novel idea. Dynamically determining the set of materialized views (step-queries) based on continuously available new step-queries for progressive queries is one of the main characteristics of our technique. To our knowledge, no similar work has been reported in the literature.

The other related work includes query processing for adaptive (dynamic) query processing and optimization [10, 12, 13]. The idea in adaptive query optimization is to exploit information that becomes available at query run time 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).

The remainder of this paper is organized as follows. The preliminaries and properties of progressive queries are introduced in Section 2. The main PQ processing procedure and relevant algorithms to construct the superior-relationship graph and dynamically materialize views are presented in Section 3. Experimental results are reported in Section 4. Section 5 summarizes the conclusions and future work.

2 Preliminaries

In this paper, we focus on discussing how to apply a dynamic materialized view technique to process a specific type of progressive query, called the monotonic linear progressive query. In this section, an overview of different types of progressive queries is given. Especially, the monotonic linear progressive query is introduced. A superior-relationship graph that is used in our technique is defined. The main properties of the monotonic linear progressive
query that are useful in our technique are also discussed.

2.1 Types of progressive queries

A progressive query (PQ) is formulated in several steps. Each step, referred to as a step-query (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 step-query can execute on either the result relation returned by the previous step-query or other external base relations. [21] defines three different types of progressive queries: single-input linear PQs, multiple-input linear PQs and non-linear PQs.

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

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

Type 3: non-linear PQs. A non-linear progressive query has the following characteristic: at least one SQ has the results returned by more than two other SQs as inputs. Thus the relationship among step-queries demonstrates a non-linear structure.

In this paper, we consider an extended type of single-input linear PQ that allows the initial step-query to have multiple external relations (i.e., a special type of multiple-input linear PQ where multiple inputs are allowed only for the initial step-query). Since the result size of each step-query is monotonically decreasing as the processing of the query progresses, we call this type of progressive query as the monotonic linear PQ.

2.2 Superior-relationship graph

In our dynamic materialized view technique, we utilize a so-called superior-relationship graph to determine if the result of a step-query under consideration should be materialized as a view. A superior-relationship graph captures the superior (or inferior) relationships among the step-queries for historical progressive queries (i.e., the PQs that have completed their execution).

Let \( s_1 \) and \( s_2 \) be two SQs belonging to one or two historical PQs. The superior relationship from \( s_1 \) to \( s_2 \) is defined as follows. For every \( t_2 \) in Result(\( s_2 \)), if there exists \( t_1 \) in Result(\( s_1 \)) such that \( t_2 \) can be completely derived from \( t_1 \), we say there is a superior relationship from \( s_1 \) to \( s_2 \) where \( s_1 \) is called a superior of \( s_2 \) and \( s_2 \) is called an inferior of \( s_1 \).

Consider the following example. Let Result(\( s_1 \))=\{\( <a_1, a_2, a_3> \), \( <b_1, b_2, b_3> \), \( <c_1, c_2, c_3> \)\}, Result(\( s_2 \))=\{\( <a_1, a_3> \), \( <b_1, b_1> \)\}, and Result(\( s_3 \))=\{\( <a_1, a_4> \)\}. Since any \( t_2 \) in Result(\( s_2 \)) can be derived from a tuple in Result(\( s_1 \)), \( s_1 \) is a superior of \( s_2 \) (i.e., \( s_2 \) is an inferior of \( s_1 \)). However, \( a_4 \) of \( <a_1, a_4> \) in Result(\( s_3 \)) cannot be derived from any tuple in Result(\( s_1 \)). Hence, there is no superior or inferior relationship between \( s_1 \) and \( s_3 \).

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. We define a special graph, called the superior-relationship graph (SRG), to capture 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 \(<sq', sq''>\) representing the superior relationships from step-query \( sq' \) to step-query \( sq'' \) 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 \(< n, id >\) indicating the identifier \( id \) of the PQ that the SQ represented by node \( n \) belongs to. 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: 

\[
\text{PAPER}(\text{Pid}, \text{Pname}, \text{FirstAuthor}, \text{PublishYear}), \\
\text{AUTHOR}(\text{Aid}, \text{Afname}, \text{Alname}, \text{Area}), \\
\text{EDITOR}(\text{Eid}, \text{Efname}, \text{Elname}, \text{Area}), \\
\text{REVIEW}(\text{Eid}, \text{Pid}, \text{Date}).
\]

Let us consider the following three PQs.

Progressive Query 1:

\[
sq_1: \textbf{select} \ Pname, \text{PublishYear}, \text{Alname} \\
\quad \textbf{from} \ \text{PAPER, AUTHOR} \\
\quad \textbf{where} \ \text{FirstAuthor} = \text{Aid};
\]

\[
sq_2: \textbf{select} \ Pname, \text{Alname} \\
\quad \textbf{from} \ \text{Result}(sq_1) \\
\quad \textbf{where} \ \text{PublishYear} = \text{2009};
\]

\[
sq_3: \textbf{select} \ Pname \\
\quad \textbf{from} \ \text{Result}(sq_2) \\
\quad \textbf{where} \ \text{Alname} = \text{'Smith'}. \\
\]

Progressive Query 2:

\[
sq_4: \textbf{select} \ \text{Efname}, \ Pname, \text{PublishYear} \\
\quad \textbf{from} \ \text{PAPER, EDITOR, REVIEW} \\
\quad \textbf{where} \ \text{PAPER.Pid} = \text{REVIEW.Pid} \\
\quad \quad \text{and} \ \text{EDITOR.Eid} = \text{REVIEW.Eid};
\]

\[
sq_5: \textbf{select} \ \text{Efname}, \ Pname, \text{PublishYear} \\
\quad \textbf{from} \ \text{Result}(sq_4) \\
\quad \textbf{where} \ \text{PublishYear} > \text{2008};
\]

\[
sq_6: \textbf{select} \ Pname \\
\quad \textbf{from} \ \text{Result}(sq_5) \\
\quad \textbf{where} \ \text{PublishYear} = \text{2009}. \\
\]

Progressive Query 3:

\[
sq_7: \textbf{select} \ Pname, \text{PublishYear} \\
\quad \textbf{from} \ \text{PAPER} \\
\quad \textbf{where} \ \text{PublishYear} > \text{2008};
\]

\[
sq_8: \textbf{select} \ Pname \\
\quad \textbf{from} \ \text{Result}(sq_7) \\
\quad \textbf{where} \ \text{PublishYear} = \text{2009}. \\
\]

Each PQ consists of two or three SQs. To construct the superior-relationship graph for the SQs in these PQs, we need to identify \( V, E \) and \( B \). \( V = \{sq_1, sq_2, sq_3, sq_4, sq_5, sq_6, sq_7, sq_8\} \). For any pair of nodes in \( V \), if they have a superior relationship and they either do not belong to the same PQ or are consecutive steps of the same PQ, there is an edge between them in the graph. For instance, \(< sq_1, sq_2 >\) and \(< sq_1, sq_8 >\) are two edges in the graph. Note that there is no edge from \( sq_1 \) to \( sq_3 \) although they have a superior relationship. This is because \( sq_1 \) and \( sq_3 \) belong to the same PQ, but they are not two consecutive steps. For each node (or SQ), we also associate the identifier of the corresponding PQ with it. Figure 1 shows the superior-relationship graph for these three PQs. From the figure, we can see that three 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 progressive queries are useful in developing an efficient processing technique.

\textbf{Property 1:} \( \text{Result}(sq_i) \sqsupseteq \text{Result}(sq_j) \) if \( i < j \) and \( sq_i, sq_j \) are two SQs in 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. So if \( sq_j \) is one of the subsequent
SQs of \(sq_i\), any tuple in \(\text{Result}(sq_j)\) must be derivable from \(\text{Result}(sq_i)\).

Property 2: Weight\((sq_i)\) \(\geq\) Weight\((sq_j)\) if \(i < j\) and \(sq_i, sq_j\) are two SQs \(\in\) 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 \(sq_i\).

#### 3 Dynamic materialized-view-based PQ processing

To efficiently process progressive queries, we introduce a dynamic materialized-view based processing procedure for PQs in Section 3.1. Different strategies to create and update a superior-relationship graph are discussed in Section 3.2. Algorithms to dynamically materialize views (step-queries) and maintain or replace existing views are discussed in Section 3.3.

##### 3.1 PQ processing procedure

The view materialization techniques have become very popular in recent years. The decision for view materialization is typically based on statistic information such as access frequency. Such techniques are often used in the data warehouse domain [9, 20].

However, unlike a conventional query, a PQ is formulated as several inter-related step-queries. Each step-query cannot be known beforehand. No one can predict what the next step-query could be. Hence, there is no pre-knowledge about future user (step) queries when deciding view materialization. This situation raises challenges to apply a materialized-view-based technique to efficiently process PQs.

To tackle the challenge, we propose 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 step-query 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 superior-relationship graph (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 if materializing the result of the CSQ is beneficial. If so, the CSQ is materialized as a view to be used for future step-queries. 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 (for the SQ that is not the first in a PQ). The set of the materialized views (SMV) is maintained. Each materialized view \(mv\) is associated with its corresponding step-query \(mv.sq\) as well as its access frequency \(mv.freq\) (assuming \(mv\) itself represents the materialized view, or in other words, the data).

The details of the PQ processing procedure are given in the following algorithm.
Algorithm 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 relation(s) in the FROM clause of csq is (are) a base relation(s) then /* csq is the 1st SQ, i.e., user starts a new PQ */
2. for each step-query sq of cpq from the 2nd to the last (i.e., i ≥ 2) do /* cpq is a completed previous PQ */
3. merge sq and sqi−1, and replace sqi by the merged query;
4. end for;
5. found=false;
6. for each sq in cpq from the 1st to the last (i.e., 1 ≤ i ≤ n) do
7. if sq is found as mv.sq for some view mv in smv then
8. found = true;
9. break;
10. end if;
11. end for;
12. if found = true then add cpq to supq;
13. else AddtoSRG(cpq, srg); end if
14. reset cpq as a new PQ with csq as the 1st SQ;
15. for each materialized view mv in smv do
16. if mv.sq is a superior of csq and size of mv < size of Domain(csq) then
17. evaluate csq on mv;
18. mv.freq++;;
19. break;
20. end if;
21. end for;
22. if csq has not been evaluated on a view then
23. evaluate csq on base relation(s) in the database;
24. end if;
25. let mcsq = csq;
26. else /* csq is not the 1st SQ */
27. add csq to cpq;
28. merge csq with all its previous SQs in cpq and save the merged query in mcsq;
29. for each materialized view mv in smv do
30. if mv.sq is a superior of mcsq and size of mv.sq < size of rpsq then
31. evaluate mcsq on mv;
32. mv.freq++;;
33. break;
34. end if;
35. end for;
36. if mcsq has not been evaluated on a view then
37. evaluate csq on rpsq;
38. end if;
39. end if;
40. if (Checkweight(srg, mcsq)) then
41. create a materialized view entry mv (including the result, query and access frequency) for mcsq;
42. AddtoSMV(mv, smv, srg, supq);
43. end if.

There are two phases in Algorithm 3.1. The first phase (lines 1 - 39) evaluates the current step-query and updates the SRG. The second phase (lines 40 - 43) decides whether the result of the current step-query should be materialized for the future use and updates the set of materialized views.

In the first phase, the algorithm first checks whether the given step-query (csq) is the first (initial) step-query (line 1) of a PQ. If so, the user is actually starting a new PQ and the pre-
vious PQ (i.e., the one saved in $cpq$) is completed. In this case, the previous PQ in $cpq$ needs to be added into either the superior-relationship graph $srg$ or the set $supq$ of used progressive queries (lines 2 - 13). Lines 2 - 4 convert each step-query in $cpq$ into one that is operated directly on the base relation(s) in the database, which can be then compared with the (step-)queries for the materialized views. If one of SQs in $cpq$ is found to have been materialized, $cpq$ is put into $supq$ (lines 6 - 12). Otherwise, $cpq$ is added into $srg$ by algorithm $AddtoSRG()$ (line 13). After having processed the previous PQ in $cpq$, $cpq$ is reset to a new PQ with $csq$ as the first (initial) SQ (line 14). If there exists 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 $Domains(csq)$, we evaluate $csq$ on the materialized view instead of its (base) operand relation(s) (lines 15 - 21). Otherwise, we evaluate $csq$ on its base operand relation(s) in the database directly. If $csq$ is not the first SQ, $cpq$ holds the previous SQs of the current PQ. In this case, $csq$ is added to $cpq$ (line 27). To check if $csq$ can be evaluated on a materialized view, it has to be converted into a step-query, $mcsq$, on the base relation(s) in the database (line 28). 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 $mcsq$, we evaluate $mcsq$ on the materialized view (lines 29 - 35). Otherwise, we evaluate $csq$ on the result of its previous step-query ($rpsq$) (line 37).

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) result of the previous step-query (if not the first SQ). For example, given

$sq_1: \sigma_{year=2009}(\text{Song}),$
$sq_2: \sigma_{country=USA}(\text{Result}(sq_1)),$

the merged second step-query

$mcsq_2: \sigma_{country=USA \land year=2009}(\text{Song})$

is on the base relation Song and obtained by merging $sq_2$ and $sq_1$.

In the second phrase, the algorithm checks to see whether materializing the current step-query $mcsq$ is beneficial by invoking an algorithm $Checkweight()$ (line 40). If so, it creates an entry for the relevant information on the materialized view for $mcsq$ and invokes an algorithm $AddtoSMV()$ to add the entry into $smv$ (lines 41-42).

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

### 3.2 Superior-relationship graph construction

The superior-relationship graph is a key component for our dynamic materialized view 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, we utilize a number of 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 identifiers (one for each SQ in $npq$). To add $npq$ into the SRG, the above nodes, edges and identifiers 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

---

$^1$The Cartesian product is considered if there is more than one relation.
from the new SQs to the old SQs are identified. The edges representing these relationships are added into the SRG. The aforementioned two algorithms utilize heuristic rules in the above two stages to improve the constructing performance.

The generating-based algorithm applies the follow 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) $sq_k$, $sq_j \in$ the same PQ.

The details of the algorithm are specified as follows.

**Algorithm 3.2 : Generating-Based ASRG:**

**AddToSRG1(npq, srg):**

**Input:** (1) new progressive-query (npq); (2) superior-relationship graph (srg).

**Output:** revised superior-relationship graph.

**Method:**

1. if srg is empty then startempty = true;
2. else startempty = false end if;
3. /* Adding an isolated PQ npq into srg */
4. for each step-query nsq of npq do
5. add a node n for nsq into node set V of srg;
6. add $\langle n, npq/s id >$ into identifier set $B$ of srg;
7. if npq has a next SQ nsq then
8. add an edge from npq to nsq into edge set $E$ of srg;
9. end if
10. end for;
11. /* Stage 1: finding external superior relationships */
12. for each progressive query opq in srg do
13. for each SQ osq of opq from the first to the last do
14. if there exists an edge from osq to opq then
15. break;
16. else if there exists a superior relationship from osq to opq then
17. add an edge from osq to opq into edge set $E$ of srg;
18. for each subsequent SQ osq' in opq do
19. if edge from osq to osq' does not exist then
20. add an edge from osq to osq' into edge set $E$ of srg;
21. end if;
22. end for;
23. for each previous SQ nsq' in npq do
24. if edge from nsq' to osq does not exist then
25. add an edge from nsq' to osq into edge set $E$ of srg;
26. for each subsequent SQ osq' in opq do
27. if edge from nsq' to osq' does not exist then;
28. add an edge from nsq' to osq' into edge set $E$ of srg;
29. end if;
30. end for;
31. end if;
32. end for;
33. break;
34. end if;
35. end for;
36. end for;
37. end for;
38. /* Stage 2: finding external inferior relationships */
39. for each progressive query opq in srg do
40. for each SQ osq of opq from the last to the first do
41. for each SQ nsq of npq from the first to the last do
42. if there exists an edge from osq to nsq then
43. break;
44. else if there exists an inferior relationship from nsq to osq then
45. add an edge from osq to nsq into edge set $E$ of srg;
46. for each subsequent SQ nsq' in npq do
47. if edge from osq to nsq' does not exist then
48. add an edge from osq to nsq' into edge set $E$ of srg;
49. end if;
50. end for;
51. for each previous SQ osq' in opq do
52. if edge from osq' to nsq does not exist then
53. add an edge from osq' to nsq into edge set $E$ of srg;
54. for each subsequent SQ nsq' in npq do
55. if edge from osq' to nsq' does not exist then
56. add an edge from osq' to nsq' into edge set $E$ of srg;
57. end if;
58. end if;
59. break;
60. end if;
61. end if;
62. end for;
63. end for;
64. end for;
65. 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 3 - 9 add the nodes, identifiers and internal edges for the SQs from the given PQ into the SRG. The edges between the nodes for the PQ and the external nodes that have already existed in the given SRG are added in two stages. Stage 1 adds the edges for the superior relationships (lines 11 - 37), while stage 2 adds the edges for the inferior relationships (lines 38 - 64).

In stage 1, the algorithm considers one old (existing) PQ in the SRG at a time (line 11). 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 13 - 15). If there exists a superior relationship between the pair, an edge
connecting the corresponding nodes are added into the SRG (lines 16 - 17). The algorithm then automatically generates more superior relationships based on Heuristic Rule 2 (lines 23 - 25) and Heuristic Rule 2 (lines 18 - 22 and 26 - 30). The relevant edges representing these superior relationships are added into the SRG. Because of the above automatic generation, it is possible that the 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 (lines 14, 19, 24 and 27).

In stage 2, the new PQ and the old PQ under consideration play the opposite roles, comparing to stage 1, because an 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 $sq_j$ (where $sq_i$, $sq_j$ are two SQs $\notin$ the same PQ), then there exists no edge from $sq_i$ to any $sq_k$ if $SQ_j$ 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 $sq_j$ (where $sq_i$, $sq_j$ 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.

**Algorithm 3.3 : Pruning-Based ASRG: AddtoSRG2(npg, srg):**

**Input:** (1) new progressive-query (npg); (2) superior-relationship graph (srg).

**Output:** revised superior-relationship graph.

**Method:**
1. if srg is empty then startempty = true;
2. else startempty = false end if;
3. /* Adding an isolated PQ npq into srg */
4. for each step-query nsq of npq do
5. add a node $n$ for nsq into node set $V$ of srg;
6. if npq has a next SQ $nnsq$ then
7. add an edge from nsq to $nnsq$ into edge set $E$ of srg;
8. end if
9. end for;
10. if not startempty then
/* Stage 1: finding external superior relationships */
11. for each progressive query opq in srg do
12. let $m$ = 1;
13. for each SQ nsq of opq from the first to
14. the last do
15. if there exists a superior relationship from nsq to osq then
16. add an edge from nsq to osq into edge set $E$ of srg;
17. else
18. let $m$ = index number of osq in opq $+$ 1;
19. break;
20. end if;
21. end for;
22. end for;
23. /* Stage 2: finding external inferior relationships */
24. for each progressive query opq in srg do
25. let $m$ = 1;
26. for each SQ nsq of opq from the first to
27. the last do
28. /* adding an edge from osq to nsq into edge set $E$ */
29. if there exists an inferior relationship from nsq to osq then
30. add an edge from osq to nsq into edge set $E$ of srg;
31. else
32. let $m$ = index number of nsq in npq $+$ 1;
33. break;
34. end if;
35. end for;
36. end for;
37. end if.
For algorithm 3.2, in the first iteration, we pick up $pq_1$ from the graph and consider its step-queries in the ascending order (from $sq_1$ to $sq_3$) while we consider step-queries 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 pair $[sq_8, sq_2]$. There exists no such a superior relationship either. So we consider pair $[sq_8, sq_3]$. Fortunately, 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 this way, we continue to process remaining nodes pairs: $[sq_7, sq_1]$, $[sq_7, sq_2]$, $[sq_7, sq_3]$, but find no edges. In the second iteration, we pick up $pq_2$ and find an edge from $sq_7$ 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 because of 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 (from $sq_7$ to $sq_8$). The first pair we check is $[sq_7, sq_3]$. Because there is a superior relationship between them, we add an edge from $sq_7$ to $sq_3$ and move to pair $[sq_7, sq_2]$. There is no superior relationship here. According to Heuristic Rule 3, we remove $[sq_7, sq_1]$ from consideration and directly move to consider pair $[sq_8, sq_3]$. We add an edge from $sq_8$ to $sq_3$. In the same way, we find an edge from $sq_7$ to $sq_6$ and terminate. In the inferior stage, we add the incoming edges for $sq_7$ or $sq_8$ into the SRG in a similar way. The details are omitted here because of the space limitation.

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, this algorithm is more efficient. In the sparse graph case, algorithm 3.3 efficiently prunes 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.

### 3.3 View materialization and replacement strategies

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 if 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:

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 is benefits 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:

**Algorithm 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. else break; end if;
11. end for;
12. end for;
13. return false;
14. 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 if any of them satisfies Conditions (1) and (2) discussed above (lines 4 - 13). If so, return true (line 9). Otherwise, return false (line 13). Heuristic Rule 1 is applied to prune impossible cases (line 10).
As mentioned earlier, the materialized views and their relevant information (e.g., associated SQs and access frequencies) are stored in a set of materialized views (SMV). However, the space allocated for the SMV is not unlimited. In addition, when an SMV becomes large, the cost for searching a materialized view also increases. We assume that (1) there is a space limit (SL) for the SMV and (2) the SL is large enough to save the largest materialized view. When the SMV overflows (i.e., its size exceeds the SL), we need to remove some materialized views from it to create enough free space for a new materialized view. When a materialized view \( v \) is removed from the SMV, the corresponding PQ for the SQ associated with \( v \) (i.e., \( v.sq \)) should be removed from the set of used PQs (SUPQ) and added into the SRG.

To decide which materialized views in the SMV should be replaced, we take their access frequencies into consideration. The materialized views in the SMV are stored in the ascending order of their access frequencies. The replacement procedure simply removes one materialized view at a time until enough free space is created for the new materialized view. This replacement strategy is incorporated in the following algorithm to add a materialized view into the SMV.

\[
\text{Algorithm 3.5: AddtoSMV}(mv, smv, srg, supq) \\
\text{Input:} (1) materialized view entry \( mv \) for an SQ; (2) set of materialized views \( \text{smv} \); (3) superior-relationship graph \( \text{srg} \); (4) set of used PQs \( \text{supq} \). \\
\text{Output:} (1) revised \( \text{smv} \) with \( mv \) added; (2) revised \( \text{srg} \); (3) revised \( \text{supq} \). \\
\text{Method:} \\
1. while \( \text{smv} \) does not have enough space to accommodate \( mv \) do \\
2. remove the next \( omv \) from \( \text{smv} \); /* materialized views in \( \text{smv} \) are kept in the ascending order of their access frequencies */ \\
3. remove the PQ \( x \) containing \( omv.sq \) in \( \text{supq} \); \\
4. AddtoSRG(\( x, srg \)); \\
5. end while; \\
6. add \( mv \) into \( \text{smv} \).
\]

Note that the replacement strategy could be extended to take more factors such as the sizes and ages of materialized views in \( \text{smv} \) into consideration, which is beyond the scope of this paper.

4 Experiments

To evaluate the performance of our dynamic materialized-view-based PQ processing (DMVPQ) technique, we conducted simulating experiments. Experiment programs were implemented in Matlab 2007 with Intel® dual core (1.5 GHz) CPU and 1 GB memory running on the Windows® Vista operating system. Specifically, 100 random progressive queries were used in our experiments. Each progressive query was composed of more than one step query, where the step numbers were randomly chosen between 2 and 5. The result sizes for all step queries ranged from 0 to 1000 disk blocks. The superior-relationship graph (SRG) and the set of materialized views (SMV) 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.

Progressive queries were processed one by one. When the execution of a PQ is completed, if no step-query in the PQ was selected as a materialized view, the PQ was added into the SRG. We maintained two parameters \( \text{IPR} \) and \( \text{WPR} \) for each node in the SRG. \( \text{IPR} \) denotes the probability with which a node has an inferior relationship with a step-query under consideration. \( \text{WPR} \) 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 step-query or not. If an SQ under consideration is estimated to be beneficial, it is materialized and added into the set of materialized views. Two parameters \( \text{SPR} \) and \( \text{SIZE} \) are maintained for each materialized view in the set. \( \text{SPR} \) denotes the probability with which the view has a superior relationship with a step-query under consideration. \( \text{SIZE} \) denotes the size of the materialized view. \( \text{IPR}, \text{WPR} \) and \( \text{SPR} \) were randomly chosen between 0 and an upper bound, without violating the definition and properties of a monotonic linear progressive query. \( \text{SIZE} \) was directly acquired from the corresponding PQ. In the experiments, the pruning-based SRG construction algorithm was adopted. Since the objective of our experi-
ments was to evaluate the performance of the DMVPQ technique, the space limitation issue 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 3 shows the performance comparison between the CSSPQ and DMVPQ techniques. The x-axis represents the total number of step-queries in the tests, 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 for small numbers of SQs. The performance of DMVPQ is increasingly better than that of CSSPQ when the number of SQs increases. The reason for this 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 progressive queries 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 $IPR$ to 0.3 and kept the other parameters unchanged. Experimental results are shown in Figure 4. From the figure, we can see that the performance of DMVPQ is dramatically improved. The reason for this is that $IPR$ plays an important role in deciding whether to materialize the result of a step-query. A larger upper bound for $IPR$ implies that each step-query has a higher chance to be 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 Figure 5. A dramatic 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 match a step-query under consideration. A larger upper bound for $SPR$ implies a materialized view has a better chance to match a given step-query. In other words, a step-query has more
available views to utilize to improve its performance.

Our experimental results demonstrate that our DMVPQ technique is quite promising in improving the performance for processing monotonic linear PQs.

5 Conclusion

There is an increasing demand to process progressive queries from various application domains. In this paper, we introduce a novel dynamic materialized-view-based technique to process a special type of progressive query, called the monotonic progressive query. The main contributions of the paper are summarized as follows:

We have presented a progressive query processing procedure to dynamically select executed step-queries as materialized views and apply the materialized views to efficiently process other step-queries.

We have introduced a superior-relationship graph (SRG), which is constructed for a set of historical progressive queries. The SRG is used to estimate the benefit of materializing a step-query. Four heuristic rules are proposed and incorporated into two efficient algorithms to construct an SRG. One is generating-based, while the other is pruning-based. The former automatically generates more edges once one edge is determined. The latter effectively prunes the impossible cases.

We have also presented heuristic-based algorithms to efficiently determine whether a given step-query should be materialized based on the SRG and to replace old materialized views with a newly selected view when the space has exceeded its limit.

We have conducted simulation experiments to evaluate the performance of our proposed technique. The experimental results demonstrate that the proposed technique is quite promising in processing the monotonic linear progressive queries. It outperforms a conventional query processing approach. Especially, its performance improvement is increasingly larger as more queries are processed.

Our future work includes extending the dynamic materialized-view-based technique to process other types of P Qs such as multiple-input linear P Qs and non-linear P Qs and studying the issues to incorporate such techniques into existing database management systems.

About the Authors

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

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

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

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International
Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

References


