Inf Syst Front (2016) 18:205–231
DOI 10.1007/s10796-014-9517-2

Optimization of generic progressive queries based on dependency analysis and materialized views

Chao Zhu · Qiang Zhu · Calisto Zuzarte · Wenbin Ma

Published online: 3 September 2014

Abstract Progressive queries (PQ) are a new type of queries emerging from numerous contemporary database applications such as e-commerce, social network, business intelligence, and decision support. Such a PQ is formulated on the fly in several steps via a set of inter-related step-queries (SQ). In our previous work, we presented a framework to process a restricted type of PQs. However, how to process generic PQs remains an open problem. In this paper, we develop a novel technique to efficiently process generic PQs based on materialized views. The SQs of an in-process PQ can utilize the results of previous SQs not only from the same PQ but also from other in-process and completed PQs. The key idea is to create a multiple query dependency graph (MQDG), which captures the data source dependency relationships among SQs from multiple PQs. A mathematic model is developed to estimate the benefit of keeping the result of an SQ as a materialized view (critical SQ/node) based on the MQDG. The kept materialized views are used to improve the performance of the future SQs. Strategies for constructing the MQDG and identifying the critical SQs for materialization by using the MQDG are presented. To manage the storage of the materialized views, we introduce two approaches – one employs a greedy method and the other adopts a dynamic programming (DP) based method. Strategies are also suggested to reduce the input problem size for the DP procedure. Experimental results demonstrate that our technique is quite promising in efficiently processing PQs.

Keywords Database · Progressive query · Materialized view · Query processing · Query optimization

C. Zhu (✉) · Q. Zhu
Department of Computer and Information Science The University of Michigan, Dearborn, MI 48128, USA
e-mail: zhuchaon1@gmail.com

C. Zuzarte · W. Ma
IBM Canada Software Laboratory, Markham, ON L6G 1C7, Canada

1 Introduction

Nowadays, the rapid growth of numerous human-interactive applications (e.g., GIS, literature search, Google) and data-intensive applications (e.g., astronomy, biology) has been changing our life. In many such applications, we observe that users demand to routinely perform their queries step by step. In each step, the result(s) of the previous step(s) is(are) used. Users can progressively complete or modify their demands by examining the result(s) returned by the previous step(s). Such a query, consisting of a set of inter-related and incrementally formulated step-queries (SQ), is called a progressive query (PQ), which was first studied in Zhu et al. (2008).

Planning a sightseeing trip by a traveler is a simple example of the PQ. Assume that the traveler is planning for a trip to see the scenes of some historic sites in China. He/she first issues a search (step-query) on a large traveling database to list all the historic sites along with their relevant information in China. After the result is returned, the traveler realizes that the result set is too large, and he/she does not want to go through each returned entry to determine if he/she is interested in visiting the corresponding site. Thus, in the second step, he/she adds a further condition on the time period (e.g., 1644 - 1912 for the Qing Dynasty) when the historic sites were established. However, the result set is still too large. Therefore, in the third step, the he/she further narrows down the result

by restricting the historic sites to be in/nearby several chosen cities (e.g., Xi'an, Nanjing) in China. This example represents a restricted but commonly used type (monotonic linear) of PQs where a new SQ only uses the result of its immediate preceding SQ. A generic PQ, however, allows its SQs to use the result(s) of any previously executed SQs and/or external table(s) as their inputs, and adopt any qualification condition at will (not being restricted to a pre-determined set of conditions). As an illustration, let us continue the previous example and extend it to a generic PQ. Assume that the traveler does not find enough interesting sites from the chosen cities, he/she may add or change the desired cities and reuse the result from the second SQ. After several SQs are issued, a satisfied list of historic sites is finally determined. The traveler may then want to search for some other information related to the selected cities and some stories/articles about the selected sites, which cannot be found in the results of previous SQs — implying the necessity for a join operation with additional data sources/tables. Other examples of PQs include a drug discovery process in pharmaceutics, a protein identification in bioinformatics, a study on the distribution and U/Pb zircon ages of A-type plutons in geoscience, an analysis on corporate data for decision making, and a search for songs from a world-wide multimedia database (Zhu et al. 2008).

These examples demonstrate a key characteristic of a PQ; namely, a PQ cannot be formulated in advance because the user cannot predict what the next SQ is before the executions of its previous SQs are completed. A user typically needs to analyze the results returned by the previous SQs and make a decision for the next SQ accordingly. How to optimize such queries presents new challenges for a database management system (DBMS).

For conventional queries, we can create indexes on the relevant tables in the database in advance to optimize the query processing. But due to the unpredictability characteristic of PQs, it is difficult to create indexes on results for SQs of PQs beforehand. To tackle this challenge, Zhu et al. introduced an effective index technique, called the collective index method (Zhu et al. 2008). The main idea is to construct a special index structure so that a collection of member indexes on an input table of an SQ can be efficiently transformed into indexes on the result table. The result index can be used to speed up the subsequent SQs. This is the first technique to address the efficient query processing issues for PQs.

It is well known that materialized view techniques have been widely used to optimize the conventional queries. The main idea is to select popular queries from the workload and materialize their results as views. The materialized views are utilized to answer future queries, instead of directly using external tables from the underlying database. In the case of PQs, since the results for SQs are highly desired to be kept in the system for answering future SQs, the materialized view techniques are quite relevant. In Zhu et al. (2010), we introduced a materialized-view based approach to efficiently processing a special type of PQs, called the monotonic linear PQs. The main idea is to construct a so-called superior-relationship graph based on the special containment properties of monotonic linear PQs and use it to dynamically select materialized views to speed up future PQs. But this technique was not designed for handling generic PQs, which may not possess the required containment properties. Hence, a new technique is required to efficiently process generic PQs.

In this paper, we present a novel materialized-view based technique to efficiently process generic PQs. The main idea is as follows: a multiple query dependency graph(MQDG), which captures the data source dependency relationships among external tables, SQs of in-process PQs and critical SQs of completed PQs, is created; a mathematical model is developed to estimate the potential benefit of SQs of completed PQs based on the MQDG; the SQs with significant estimated benefits are selected as critical ones and their results are kept as materialized views in a so-called the critical node view space (CNS); different strategies for effective utilization of the CNS under a space limit are incorporated. Using the MQDG, users can specify new SQs by using not only the results for SQs from the same PQ but also the results for SQs from other in-process PQs since they are all available in the system without additional cost. Furthermore, the results for some popular (critical) SQs can also be utilized by users to optimize their future SQs. Since a user has more options in specifying his/her SQs, with the assistance (e.g., cost estimation) from the system, it is expected that an improved performance of his/her PQ can be achieved. Our experiments demonstrate that this approach is quite promising.

The work that is most related to progressive queries in the literature is discussed as follows. Tiakas et al. proposed an algorithm for processing a top-k dominating query to progressively report k items with the highest domination scores (Tiakas et al. 2011). Raghavan et al. presented a progressive evaluation framework ProgXe to progressively generate query results early and often for mulit-criteria decision support queries (Raghavan and Rundensteiner 2010). Jang et al. designed a methodology of progressive filtering (PF) for multimedia information retrieval, whose applications were called the melody recognition (Jang and Lee 2008). Kache et al. proposed a progressive optimization technique for federated queries, which were regular relational queries accessing data on one or more remote relational or non-relational data sources, possibly combining them with tables stored in the federated DBMS server (Kache et al. 2006). Papadias et al. designed a progressive algorithm for the skyline queries, which was called the BBS

(branch-and-bound skyline). The BBS can quickly return the first skyline points without having to read the entire data file (Papadias et al. 2003). Tan et al. proposed a technique to handle nested queries with aggregates by providing users with (approximate) answers progressively. While the above techniques can be considered as "progressive", queries processed by those techniques are, however, formulated at once (non-progressive).

Other work related to the progressive queries includes the query processing for continuous queries (Agarwal et al. 2006; Babu 2005; Lee et al. 2010; Lim et al. 2006; Mokbel 2006), the adaptive query optimization (Antoshenkov 1993; Babu and Bizarro 2005; Kabra and DeWitt 1998; Liu and Pu 1997; Lu et al. 1995; Markl et al. 2004), and the Extraction-Transformation-Loading (ETL) processing (Jorg and DeBloch 2008; Simitsis et al. 2005; Vassiliadis et al. 2001; Vassiliadis et al. 2005). Continuous queries require a query to repeatedly execute over a continuous stream of data; the main idea of the adaptive query optimization is to adapt the environments of a query for optimizing the query processing during the query execution time. The difference with the progressive queries is that a query in either streaming data or adaptive query optimization is also only formulated once. The ETL processing chains multiple activities/operators together in a workflow. 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 progressive queries.

Many materialized view techniques have been reported in the literature. Different types of databases are considered: the relational database (Zhou et al. 2007; Luo 2007; Goldstein and Larson 2001; Mistry et al. 2001; Gupta et al. 1995; Blakeley et al. 1986; Agrawal et al. 2000; Re and Suciu 2007; Folkert et al. 2005), the data warehousing (Phan and Li 2008; Park et al. 2001; Baralis et al. 1998; Yang et al. 1997), the distributed database (Jiang et al. 2008), and the XML database (Liu and Chen 2008; Tang et al. 2008; Jiang et al. 2008; Balmin et al. 1997; Xu and Ozsoyoglu 2005; Arion et al. 2007). The materialized view techniques are mainly focused on the following three issues: the materialized view selection, the materialized view matching, and the materialized view maintenance. For the materialized view selection problem, which is the most related issue to our problem, has been well studied. Phan et al. presented a dynamic Materialized Query Tables (MQT) management scheme that materialized views and created indexes in an on-demand fashion as a workload executed and managed them with an LRU cache (Phan and Li 2008). Zhou, Luo, et al. presented flexible materialization strategies which selectively materialized only a subset of rows of a table to reduce storage space and view maintenance costs (Zhou et al. 2007; Luo 2007). Baralis et al. developed a technique to select proper materialized views in the multidimensional database by considering only the relevant elements of the multidimensional lattice (Baralis et al. 1998). Yang, Agrawal, Jiang, et al. proposed different approaches to select proper views so as to achieve the best combination of good query performance and low view maintenance (Re and Suciu 2007; Yang et al. 1997; Agrawal et al. 2000; Jiang et al. 2008). However, no technique was designed to select materialized views for processing PQs. The materialized view matching issue is also considered as the problem of answering queries using views. Tang et al. developed different techniques for rewriting XPATH queries using materialized views (Tang et al. 2008; Balmin et al. 1997; Xu and Ozsoyoglu 2005; Arion et al. 2007). Liu et al. presented techniques for answering keyword queries using a minimal number of materialized views (Liu and Chen 2008). Park et al. proposed a method for rewriting a given OLAP query using various kinds of materialized aggregate views (Park et al. 2001). Goldstein et al. designed a fast and scalable algorithm for determining whether part or all of a query can be computed from materialized views and described how it can be incorporated in transformation-based optimizers (Goldstein and Larson 2001). Since the view matching process for SQs of PQs is the same as that for conventional queries, to answer an SQ using views, a commonly used view matching technique can be applied. Therefore, the view matching issue is not our focus in this work. For a database with updates, the materialized view maintenance issue needs to be considered. Blakeley, Folkert, Gupta, et al. presented different materialized view updating methods considering the changes to the external tables in the database (Blakeley et al. 1986; Folkert et al. 2005; Gupta et al. 1995). Mistry et al. introduced a method for reducing the view maintenance cost by determining an optimal set of additional views to materialize (Mistry et al. 2001). For simplicity, we assume read-only databases in this work, which removes the materialized view maintenance issue from consideration. However, the materialized view technique developed in this work could be extended to handle databases with updates in several ways. One way is to identity and remove the affected materialized views (i.e., critical SQs) from the view space (i.e., CNS) based on our MQDG when database changes are detected. Alternatively, we could adapt a conventional materialized view maintenance technique to the task of maintaining materialized views in the view space for PQs. The detailed study on the relevant materialized view maintenance techniques for PQs is our future work. On the other hand, we consider another maintenance issue in this work, that is, how to maintain the view space under a given space limit. Although much work has been done on using the materialized views in the past, only our previous work in Zhu et al. (2010) was focused on applying materialized views to process progressive queries. However, as mentioned earlier, that work was restricted to handle

a special type of PQs. Applying materialized views to efficiently process generic PQs is the new issue addressed in this paper.

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

2 Preliminaries

In this section, some background knowledge of this work is provided. An overview of three types of PQs is given and the dependency graph is defined in Section 2.1. Related concepts for the MQDG are introduced in Section 2.2. A so-called view storage (VS) for saving the materialized views is presented in Section 2.3.

2.1 Progressive query types and multiple query dependency graph

PQs are classified into three types in Zhu et al. (2008). The first type is called the single-linear PQ. Each SQ (except the initial one) in such a PQ can only use the result table returned by its (immediate) preceding SQ as its input. The initial SQ uses an external table from the database as its input. The second type is called the multiple-input linear PQ. Each SQ in a PQ of this type uses not only the result returned by its (immediate) preceding SQ but also some external tables as its inputs. The third type is called the non-linear PQ. In such a PQ, an SQ utilizes the results for more than one previous SQ as its inputs. The domain of an SQ sq_1, denoted by $Domain(SQ)$, is defined as the set of input tables of sq_1.

In our previous work (Zhu et al. 2010), we considered PQs of a special case of the second type (multiple-input linear PQ), called the monotonic linear PQs, where multiple inputs are only allowed for the initial SQ. In this paper, we design a technique to handle generic PQs, namely, all types of PQs mentioned above are allowed.

As mentioned earlier, the main characteristic of a PQ is that the user cannot predict what the next SQ is before the previous SQs are executed. Therefore, the result tables for executed SQs of in-process PQs have to be made available (not discarded) in the system since they may be used by future SQs. In general, multiple PQs are simultaneously processed in a DBMS. We consider the executed SQs of in-process PQs as temporary SQs and keep their result tables in the system. Conceptually, an SQ only utilizes the result tables for previous SQs of the same PQ. In this paper, we also allow users to utilize the result tables for SQs of other in-process PQs rather than the same PQ if a better performance can be achieved. Usually, the result tables for SQs of completed (historical) PQs are no longer kept in the system. However, some historical SQs may be very popular since their results are frequently utilized. Thus, the results for such SQs are still kept in the system even after their corresponding PQs are completed. Such SQs are named critical SQs.

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

Let SSQ be the set of SQs of in-process PQs and critical SQs of completed PQs. Each node in V represents either an external table used by an SQ in SSQ or directly an SQ in SSQ. The former is called a table node, while the latter is called a temporary node (for a temporary SQ) or a critical node (for a critical SQ). If a node v_2 representing an SQ uses as input the external table or the result table associated with node v_1, we say v_2 depends on v_1, which is represented by a directed edge $e =< v_1, v_2 >$ from v_1 to v_2 in E. In this case, we also say that there exists a dependency relationship from v_1 to v_2. The set P in $MQDG(SPQ)$ consists of unique identifiers for all the PQs in SPQ. Function F_P in $MQDG(SPQ)$ maps (labels) each temporary node representing an SQ of a PQ in SPQ to the id in P for the corresponding PQ to which the SQ belongs. Function F_S in $MQDG(SPQ)$ maps each node in V representing an SQ(critical SQs or temporary SQs) or an external table to its result size or table size in S. An MQDG dynamically grows as more SQs of current PQs or new PQs are issued. Figure 1 shows an example of the MQDG.

Several properties of an MQDG can be observed. First of all, there exists no directed circle in the graph. A directed edge from a node v_1 to a node v_2 in the graph represents that v_2 is dependent on v_1, which implies that v_2 is generated later than v_1. On the other hand, all the outgoing paths from v_2 are to connect the nodes which are generated later than v_2. Therefore, it is impossible to form a recursive cycle in the graph. Secondly, isolated sub-graphs may exist in an MQDG. The result for an SQ of a PQ q may never be used by any subsequent SQs of q. Since we allow an SQ to use the result(s) of SQ(s) from different PQs, on the other hand,

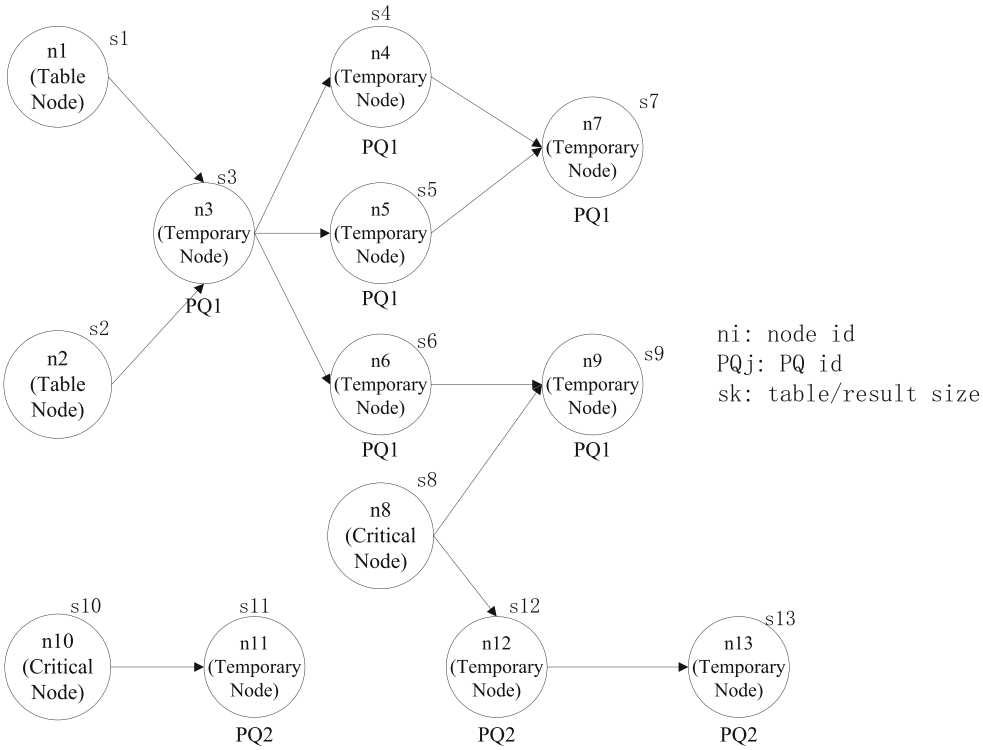

Fig. 1 An example of the multiple query dependency graph (MQDG)

the SQs from different PQs may be connected together in the graph.

Other properties of an MQDG include that each table node has no incoming edge and there is a single sink (final) node for each PQ that returns the final result for the PQ. Note that a dependency graph (DG) for a given PQ was defined in Zhu et al. (2008). There are several differences between a DG and an MQDG. First of all, a DG is for a single PQ, while an MQDG is for multiple PQs. Secondly, a DG is used to illustrate the definition of a (complete) PG, while an MQDG is used to optimize multiple in-process PQs that are incomplete and growing. Finally, a DG does not include nodes for external tables, while an MQDG does.

2.2 Basic concepts in MQDG

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

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

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

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

Note that a node n in an MQDG can be both a direct child node and an indirect child node of node m. For the example in Fig. 4, there exists a direct path from sq_1 to sq_6 which contains only one edge, and there also exists an indirect path from sq_1 to sq_6 which consists two edges $\{< sq_1, sq_2 >, < sq_2, sq_6 >\}$. Consequently, sq_6 is both a direct child node and an indirect child node of sq_1.

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

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

2.3 View storage

As mentioned earlier, the result tables associated with temporary nodes and critical nodes are kept in the system as materialized views. Thus, a space, which is called the view storage (VS), is allocated to save those materialized views. The VS is divided into two subspaces: the temporary node view space (TNS) and the critical node view space (CNS). The TNS is to store the set of result tables for all the temporary nodes (temporary node views), while the CNS is to keep the set of result tables for all the critical nodes (critical node views). Note that the related information of a (materialized) view, e.g., PQ id, the size of the view, and query expression, is also saved in the VS. Figure 2 shows the structure of the view storage.

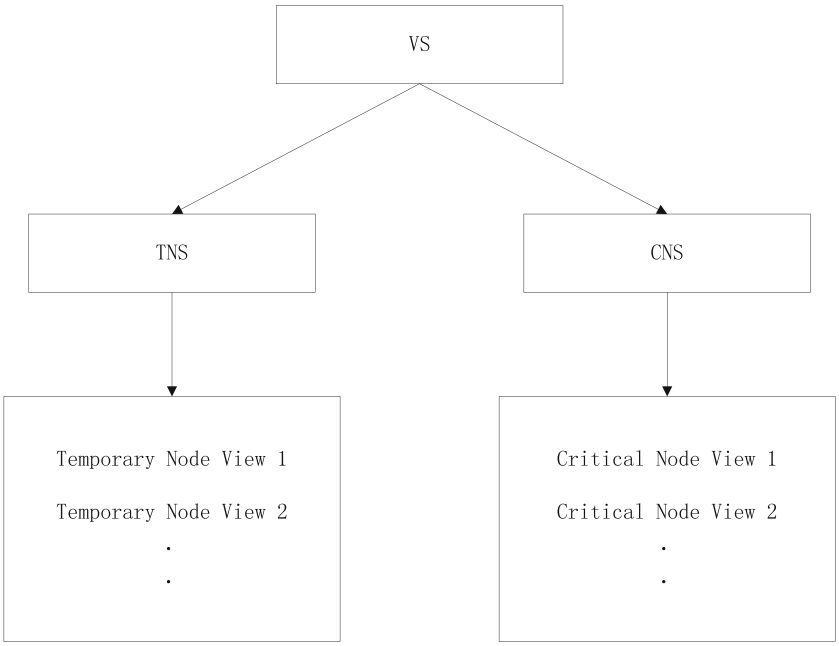

Fig. 2 The structure of the view storage

It is observed that the space limit for the TNS determines how many in-process PQs can be executed simultaneously in the system. This space limit is implied/reflected in the maximum number of in-process PQs allowed in the system, which is usually specified in a control file of the system. Since the result tables of all the temporary nodes (SQs) of an in-process PQ have to be stored in the TNS, there is no (temporary node) view selection problem for this subspace. The TNS may consist of both main memory and disk spaces, with the main memory having a higher priority to be used first. On the other hand, the space limit for the CNS determines how many beneficial critical SQ results can be retained. Since the number of SQs of completed PQs can grow unlimited, it is impossible to keep all their results as critical node views in the CNS. Hence, we face a (critical node) view selection problem for the CNS. Furthermore, when the CNS overflows, we also need to consider how to maintain the view space by replacing some less beneficial existing ones with more beneficial new ones. The view selection and view space maintenance for the CNS are the main issues to be solved by our technique.

3 A materialized-view based technique for efficiently processing PQs

In this section, we present the various components of our materialized view based technique for processing generic PQs. The main processing procedure is introduced in Section 3.1. The estimation model to identify the critical nodes from completed PQs by using the MQDG is discussed in Section 3.2. The policy to remove non-critical nodes from the MQDG is presented in Section 3.3, and the strategies to insert critical nodes into the CNS with a given space limit is discussed in Section 3.4.

3.1 Main processing procedure

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

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

Specifically, our technique address the following four issues: (1) how to construct an MQDG; (2) how to use the MQDG to find the critical nodes from completed PQs; (3) how to remove the non-critical nodes of a completed PQ from the MQDG; and (4) how to address the maintenance issue for the CNS under a certain space limit.

The high-level flowchart of the main procedure is shown in Fig. 3. It starts with a new SQ nsq being issued to the system. nsq can take advantage of all available critical nodes

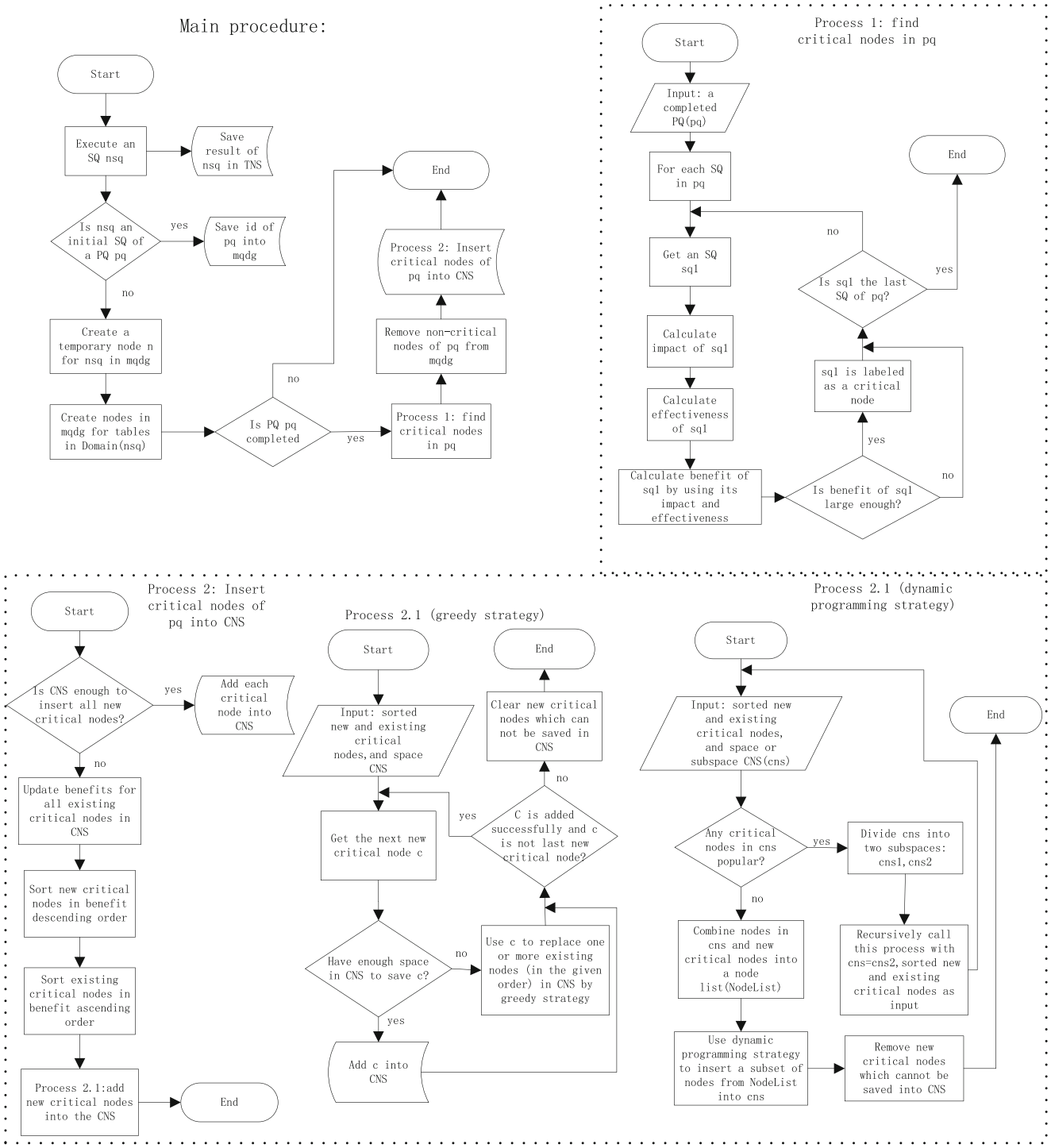

Fig. 3 The flowchart of the main procedure

from the CNS to optimize its query processing. The result table of *nsq* is saved in TNS. If *nsq* is an initial SQ of a new PQ *pq*, the id of *pq* is saved in the MQDG. *nsq* and the domain tables of *nsq* are added into the MQDG. Next, the system checks if PQ *pq* is completed. If so, an algorithm is applied to look for critical nodes from *pq* by using the MQDG (the algorithm is shown as Process 1 in Fig. 3). After that, all non-critical nodes of *pq* are removed. Finally, the newly searched critical nodes are inserted into the CNS. Since the CNS may overflow, we designed two different strategies (greedy strategy and dynamic programming strategy) to address the CNS overflowing problem which are

shown as Process 2 in Fig. 3. The detailed description of the main procedure is given by the following algorithm.

There are two phases in Algorithm 1. The first phase (lines 1 – 18) executes the newly arrived SQ and revises MQDG and VS to include this SQ. The second phase (lines 19 – 25) finds the critical nodes of a completed PQ, removes non-critical nodes of the PQ, updates the MQDG, and inserts the discovered critical nodes into the VS. The second phase is done by invoking several external functions.

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

In the second phase, the algorithm first invokes function FindCriticalNode() to estimate the benefit of each SQ of a completed PQ to identify and return a set of critical nodes (lines 19 – 20). After all critical nodes are identified from a PQ, the non-critical nodes are removed from the MQDG by invoking function RemoveAndTransfer() (lines 21 – 23) and the result tables of all critical nodes are inserted into the CNS by invoking function CriticalNodesInsertion() (line 24).

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

3.2 Estimation model for critical nodes

The main purpose of constructing an MQDG is to estimate the potential benefits for SQs of a completed PQ to identify critical nodes. As mentioned earlier, how to find the critical nodes from a completed PQ is a crucial issue in this work. In this subsection, we discuss this issue and introduce a model for estimating the potential benefits of SQs by using the MQDG.

The main idea to estimate the potential benefit for an SQ sq_1 is to consider that how the result table of sq_1 has been efficiently and effectively used to answer other nodes (SQs) in the MQDG. We developed an estimation model to quantitatively capture the benefit of an SQ (i.e., a temporary node) by using an MQDG. Before introducing the model, two affecting factors are defined first.

(1) *Impact*: if a node n in an MQDG can be directly or indirectly computed from the result table of an SQ sq (i.e., n is a direct or indirect child node of sq), we say sq has an impact on n. We have derived a formula to quantitatively estimate the impact of sq on n. We consider the impact of sq is the accumulated impact of sq on all its direct and indirect child nodes in the MQDG. The larger the value for the impact of sq is, the more nodes in the MQDG can be directly or indi-

Algorithm 1 Selection of materialized views via dependency analysis

Input: (1) newly arrived step-query (nsq); (2) multiple query dependency graph $mqdg = (V, E, P, S, F_S, F_P)$; (3) view storage ($vs$) including temporary node view space (tns) and critical node view space (cns); (4) current maximum impact (cmi); (5) current maximum effectiveness (cme).
Output: (1) revised $mqdg$; (2) revised vs.
Method:
1. execute nsq and save its result table in tr;
2. add tr and relevant information into $vs.tns$;
   /* revise $mqdg$ to include nsq */
3. **if** nsq is an initial SQ for a new PQ **then**
4. add the id of PQ into $mqdg.P$;
5. **end if**
6. create a temporary node tn labeled with the corresponding PQ id for nsq in $mqdg.V$, add the result table size rs of nsq in $mqdg.S$, and map tn to rs in $mqdg.S$;
7. **for** each table r in $Domain(nsq)$ **do**
8. **if** r is an external table **then**
9. **if** r does not have a node in $mqdg.V$ **then**
10. create a table node m for r in $mqdg.V$, add the size of r in $mqdg.S$, and mapping m to the corresponding size in $mqdg.S$;
11. **else**
12. find the table node m representing r in $mqdg.V$;
13. **end if**
14. **else**
15. find the corresponding node m for r in $mqdg.V$;
16. **end if**
17. generate an edge $< m, tn >$ from m to tn in $mqdg.E$;
18. **end for**
    /* find critical nodes in a completed PQ */
19. **if** the corresponding PQ pq in $mqdg$ is completed **then**
20. ($cnset$, $BenefitList$, cmi, cme) = FindCriticalNode($mqdg$, pq, cmi, cme);
    /* remove non-critical nodes for a completed PQ */
21. **for** each non-critical node ncn in pq **do**
22. RemoveAndTransfer($mqdg$, ncn);
23. **end for**
    /* insert critical nodes into the CNS */
24. CriticalNodesInsertion($mqdg$, $cnset$, vs, $BenefitList$, cmi, cme);
25. **end if**.

rectly derived from the result table of sq. Therefore, the value for the impact of sq represents whether the result table of sq is frequently used by other SQs in the MQDG.

(2) *Effectiveness*: it represents the storage effectiveness. We consider that the value v for effectiveness of sq implies that how many tuples on average can be directly computed from a single unit of data in the result table of sq (we assume that the smallest unit in the table is a tuple). Keeping the result table of an SQ sq with a larger v can improve the utilization of the limited CNS. If the size of the result table of sq is fixed, the larger v represents more tuples can be directly derived. Hence, the value for the effectiveness of sq represents that how the result table of sq is effectively used by other SQs in the MQDG.

In the estimation model, the impact and effectiveness of an SQ are combined. The reason for that is as follows: either the impact or the effectiveness of an SQ sq can only partially reflect the potential benefit of sq. Let us consider two different scenarios.

(a) Assume that the impact of an SQ sq is very large, but the result table size of sq is also very large, which leads to a very small effectiveness for sq. In this case, if we only use the impact of sq to represent the potential benefit of sq, the benefit is very large. However, although the result table of sq is frequently used by other nodes (SQs) in the MQDG, the space overhead is very high. In other words, sq is not effectively used by other nodes in the MQDG although it is heavily used.

(b) Assume that the impact of an SQ sq is very small, but the size of sq is also very small, which leads to a very large effectiveness for sq. In this case, if we only use the effectiveness of sq to represent the potential benefit of sq, the benefit is very large. But actually sq is not frequently used to compute other nodes. Therefore, sq is not a good candidate for a critical node.

Now let us introduce the details of our benefit estimation model. Assume that we want to estimate the potential benefit for keeping the result of an SQ sq. Let $Imp(sq)$ be the impact of sq, Imp_{max} be the current maximum impact of all compared SQs in the MQDG, $Eff(sq)$ be the effectiveness of sq, and Eff_{max} be the current maximum effectiveness of all compared SQs in the MQDG. The model is shown as follows:

$$Benefit(sq) = \left(\frac{Imp(sq)}{Imp_{max}}\right)^{\alpha} * \left(\frac{Eff(sq)}{Eff_{max}}\right)^{\beta} \quad (1)$$

where α and β are parameters representing the importance of the impact and the effectiveness of sq in the model, respectively. α and β range from 0 to ∞. For example, typically $\alpha = \beta = 1$ (which will be used in our remaining discussion).

The model computes a value representing how an SQ sq has been efficiently and effectively used by other nodes (SQs) in an MQDG. We consider this value as the potential benefit for keeping the result of sq. The components $Imp(sq)/Imp_{max}$ and $Eff(sq)/Eff_{max}$ represents the normalized impact and effectiveness of sq, respectively.

With the above model, we can quantitatively compute the potential benefit for any temporary SQ sq in the MQDG and decide whether select sq as a critical node. Next, we discuss how to quantitatively calculate the impact and effectiveness of an SQ in detail.

Let us first discuss the details about how to compute the impact of an SQ sq_1 by using the MQDG. The main idea is to compute how much impact sq_1 has already brought to every its direct/indirect child node sq_2 by using the MQDG. The following affecting factors are considered.

(i) *The distance*: it is the number of edges in a path from the node sq_1 to node sq_2. The larger the distance is, the smaller the impact that sq_1 could make to sq_2. As an illustration, we consider the following two scenarios: 1) sq_2 is a direct child node of sq_1. In this case, sq_2 could directly make use of the result of sq_1. 2) sq_2 is an indirect child node of sq_1 and there is an path from sq_1 to sq_2. In this case, sq_2 could not directly take advantage of the result of sq_1. It is clear that sq_1 could make more contribution to executing sq_2 in the first scenario than in the second scenario. In other words, sq_1 has more impact on sq_2 if the distance from sq_1 to sq_2 along the path that is under consideration is shorter. Note that, if there are multiple paths from sq_1 to sq_2, the impact gained by sq_2 through them are accumulated.

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

of sq_1. Therefore, external nodes are more relevant than internal nodes to represent future SQs. In other words, it is reasonable to assign sq_1 a higher impact value if sq_2 is an external node.

(iii) *The number of inputs*: it represents the number of incoming edges of sq_2, assuming sq_2 is a direct child node of sq_1. The reason why this factor matters is that an SQ may only make a partial contribution to the evaluation of its direct child nodes. The larger the number of inputs that sq_2 has, the less the impact that sq_1 could make to sq_2. Consider the following two different scenarios. The first scenario is that sq_2 has only one incoming edge, which is from sq_1. In this case, sq_2 is evaluated totally based on the result table of sq_1. The second scenario is that sq_2 has n ($n > 1$) incoming edges, one of which is from sq_1. In this case, several tables (result tables for SQs or external tables) make contribution on evaluating sq_2. It is obvious that sq_1 has more impact on sq_2 in the first scenario.

Therefore, if an SQ sq has only one input, its input table makes a total contribution in evaluating sq. However, if sq has more than one input, it is required to decide how much contribution each input of sq can make. Assume that an SQ sq_i has three inputs: sq_1, sq_2 and sq_3. $Result(sq_1)$ has three attributes: A, B and C. $Result(sq_2)$ has three attributes: A, D and E. $Result(sq_3)$ has two attribute: A and F. Attribute A is the key attribute. To execute sq_i, three tables are joined and the Cartesian product $T(A, B, C, D, E, F)$ is computed. We observed that each tuple in T is coming partially from $Result(sq_1)$ (A, B, C), partially from $Result(sq_2)$ (D, E), and partially from $Result(sq_3)$ (F). Regardless the filter conditions in the query expression of sq_i, we consider that each input of sq_i (sq_1, sq_2 or sq_3) makes one third contribution to evaluate sq_i.

If sq_2 is an indirect child node of sq_1, the case becomes more complicated since many intermediate SQs on the path from sq_1 to sq_2 also have more than one incoming edge. Thus, sq_1 makes even less contribution to sq_2 and the incoming edges of all the intermediate nodes also need to be considered.

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

$$Imp(sq) = \sum_{c \in ChdS(sq)} \sum_{p \in PthS(sq,c)} \times \frac{(W_d)^{|p|-1} * [W_E * Ex(sq, c) + W_I * In(sq, c)]}{\prod_{x \in NdeS(p)} NE(x)}, \quad (2)$$

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

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

Let us consider the example in Fig. 4. Assume that we want to calculate the impact that sq_1 has brought to sq_6. First, all possible paths from sq_1 to sq_6 are listed:

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

Clearly, $Ex(sq_1, sq_6) = 0$, $In(sq_1, sq_6) = 1$.

For path p_1, $|p_1| = 1$, $NE(sq_6) = 3$. Thus, the impact that sq_1 has brought to sq_6 through p_1 is:

$$Imp(sq6) \text{ on p1} = \frac{(0.5)^0 * 1}{3} = \frac{1}{3}.$$

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

$$Imp(sq6) \text{ on p2} = \frac{(0.5)^1 * 1}{1 * 3} = \frac{1}{6}.$$

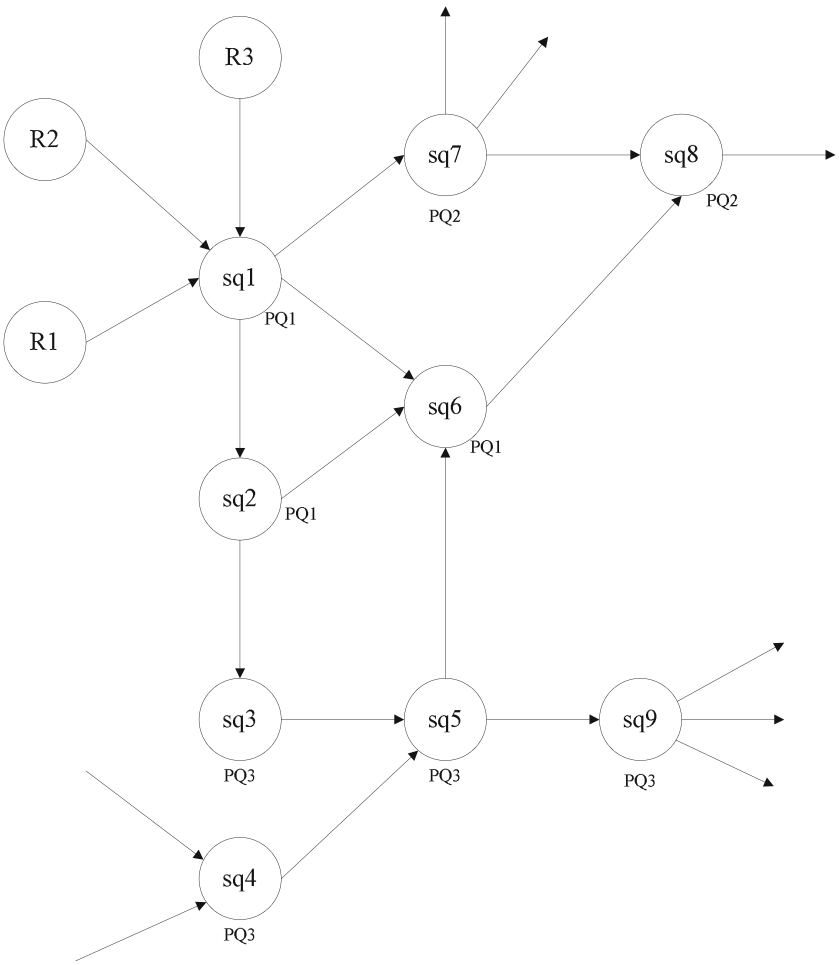

Fig. 4 The MQDG for the example

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

$$Imp(sq6) \text{ on p3 } = \frac{(0.5)^3 * 1}{1 * 1 * 2 * 3} = \frac{1}{48}.$$

Therefore, the total impact that sq_1 has brought to sq_6 is to add the above three impact values together, i.e., $Imp(sq6) \approx 0.52$.

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

In Algorithm 2, i_n denotes the impact that t has brought to its current individual direct child node n along one path. For each direct child node n of t, i_n is calculated differently based on the node type of n (internal or external node of t) (lines 4 – 8). A function RecursiveAcc() is called to recursively calculate the impact that t has brought to the indirect child nodes of t along the current path (line 10). Finally, the total impact that t has brought to all its direct and indirect child nodes is returned (line 12).

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

Using function CalculateImpact(), the value for the impact of an SQ can be easily computed. Next let us consider how to compute the value for the effectiveness of an SQ by using the MQDG. The main idea is to calculate how many tuples on average can be computed from a unit

Algorithm 2 CalculateImpact(*mqdg, t*)

Input: (1) multiple query dependency graph $mqdg = (V, E, P, S, F_S, F_P)$; (2) temporary node ($t$).
Output: impact value of t.
Method:
1. $i_t = 0$;
2. **for** each direct child node n of t **do**
3. N = number of incoming edges of n;
4. **if** n.pqid = t.pqid **then**
   /* n is an internal node */
5. $i_n = W_I$/N;
6. **else**
   /* n is an external node */
7. $i_n = W_E$/N;
8. **end if**
9. $i_t = i_t + i_n$;
10. i_t = RecursiveAcc($mqdg, t, n, i_t, i_n$);
11. **end for**
12. return i_t.

(assume that a unit in the result table is a tuple) of the result table for sq.

For each direct child node n of sq, the average number av of tuples of n, which can be derived from keeping a tuple (unit) from the result of sq, is computed. This number av is considered as the effectiveness of sq for n. To calculate such effectiveness, let us consider the following two cases. Assume that the result sizes of sq and n are s_1 and s_2, respectively. The number of input tables for n is num. In the first case, num equals to 1. It implies that the result tuples for n are totally derived from the result of sq. Thus, the effectiveness of sq for n is computed by s_2/s_1. In the second case, num is greater than 1, say num equals to 2. Assume that the size of another input table of n is s_3. The effectiveness of sq for n is calculated by $s_2/(s_1+s_3)$ since only part of the result for n is derived from the result for sq. After the effectiveness of sq for all its direct child nodes are computed, the accumulated value is considered as the total effectiveness of sq. Note that the effectiveness of a node sq is about the storage utilization for producing the results of the (direct) child nodes of sq. Since the results of the indirect child nodes of sq are produced from their direct parent nodes, the storage of sq has little effect on its indirect child nodes. Hence, the indirect child nodes of sq are not considered in calculating the effectiveness of sq.

Let $DchdS(sq)$ be the set of direct child nodes of sq, $DpadS(c)$ be the set of direct parent nodes of a direct child node c of sq. The following formula is derived to calculate the effectiveness of sq.

$$Eff(sq) = \sum_{c \in DchdS(sq)} \frac{Size(Result(c))}{\sum_{t \in DpadS(c)} Size(Result(t))}. \quad (3)$$

The formula essentially computes the value for the total effectiveness of sq. $Size(Result(c))$ denotes the result size of each direct child node c of sq. $Size(Result(t))$ denotes the result size of each direct parent node t of c. From the formula, we can see that, if c has only one input (direct parent node), the effectiveness of sq for c is $Size(Result(c))/Size(Result(sq))$. Otherwise, the effectiveness of sq for c is $Size(Result(c))/\left(\sum_{t \in DpadS(c)} Size(Result(t))\right)$.

Let us consider the example in Fig. 4. Assume that we want to calculate the effectiveness of $sq1$. $sq1$ has three direct child nodes: $sq2$, $sq6$, and $sq7$. $Size(Result(sq1))$ is 100, $Size(Result(sq2))$ is 50, $Size(Result(sq6))$ is 100, and $Size(Result(sq7))$ is 50.

The effectiveness of sq_1 for its first direct child node sq_2 is:

$$Eff(sq1) \text{ for sq2} = \frac{Size(Result(sq2))}{Size(Result(sq1))} = \frac{50}{100} = \frac{1}{2}.$$

Algorithm 3 RecursiveAcc($mqdg, t, n, i_t, i_n$)

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

The effectiveness of sq_1 for its second direct node sq_6 is:

$$Eff(sq1) \text{ for sq6} = \frac{Size(Result(sq6))}{Size(Result(sq1)) + Size(Result(sq2))} = \frac{100}{100+50} = \frac{2}{3}.$$

Note that sq_1 and sq_2 are input tables of sq_6.

The effectiveness of sq_1 for its third direct node sq_7 is:

$$Eff(sq1) \text{ for sq7} = \frac{Size(Result(sq7))}{Size(Result(sq1))} = \frac{50}{100} = \frac{1}{2}.$$

Therefore, the accumulated effectiveness of $sq1$ is to add the above three effectiveness values together, i.e., $Eff(sq1) \approx 1.67$. In other words, every tuple of $Result(sq1)$ has produced a little more than 1.5 tuples for other SQs in the MQDG.

The following algorithm computes the effectiveness of a temporary node in the MQDG by using the above formula.

Algorithm 4 CalculateEffectiveness($mqdg, t$)

Input: (1) multiple query dependency graph $mqdg = (V, E, P, S, F_S, F_P)$; (2) temporary node ($t$).
Output: effectiveness value of t.
Method:

```
1.  e_t = 0;
2.  for each direct child node n of t do
3.    size_n = the size of the result table of n;
4.    size_pn = 0;
5.    for each direct parent node p of n do
6.      size_p = the size of the result table of p;
7.      size_pn = size_pn + size_p;
8.    end for
9.    e_n = size_n / size_pn;
10.   e_t = e_t+e_n;
11.  end for
12.  return e_t.
```

In Algorithm 4, e_t denotes the total effectiveness that t has brought for all its direct child nodes. e_n represents the effectiveness that t has brought for its current individual direct child node n. e_n is computed differently for a different child based on the number of inputs of n (lines 5 – 9) and e_t is accumulated (line 10).

Now, we go back to discuss our benefit estimation model (1). Let us show an example for estimating the benefit of an SQ in a completed PQ by using the model. Assume that a completed PQ $pq1$ is composed of four SQs: $sq1$, $sq2$, $sq3$, and $sq4$. The impact values for $sq1$, $sq2$, $sq3$, and $sq4$ are: 0.5, 0.9, 0.6, and 0.8, respectively. The effectiveness values for $sq1$, $sq2$, $sq3$, and $sq4$ are: 2, 2.5, 1.5, and 1, respectively. The impact and the effectiveness in the model are of the same importance, i.e., $\alpha = \beta = 1$. The current maximum impact is 1.0 and the current maximum effectiveness is 3. The benefit of $sq1$ is desired. In this example, $Imp(sq1) = 0.5$. $Eff(sq1) = 2$. Hence,

$$Benefit(sq1) = \left(\frac{Imp(sq1)}{Imp_{max}}\right)^{\alpha} * \left(\frac{Eff(sq1)}{Eff_{max}}\right)^{\beta} = \left(\frac{0.5}{1.0}\right)^{1} * \left(\frac{2}{3}\right)^{1} \approx 0.33.$$

Now we apply the model to compute the benefits for all SQs of a completed PQ in the MQDG to identify critical nodes. This process is described as Process 1 in Fig. 3. The detailed algorithm is shown as follow.

Algorithm 5 first calculates the impact and effectiveness values for each SQ fsq of the completed PQ fpq (lines 2 – 7). Next, the current maximum impact and the current maximum effectiveness, which are used for normalization in the model, are updated (lines 8 – 9). After that, the benefit estimation model is applied to calculate the benefit value for each SQ fsq of fpq (lines 10 – 13). If a benefit value is larger than a predefined threshold, the corresponding SQ is considered as a critical node (lines 14 – 15).

3.3 Non-critical node removal

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

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

In Algorithm 6, the given node n is safely removed and all dependency relationships are transferred in four steps. In the first step, the query expressions for all the direct child nodes of n are changed (lines 3 – 4). We know that the result table r for n is used by each of its direct child node. Since n is to be removed, r will no longer exist. Hence, we replace r in the query expression of each direct child node of n by the query expression of n. As a result, r is removed from the domain of each direct child SQ (node). For example, consider sq_1: $\sigma_{c_1=v1}$ (R1); sq_2: $\sigma_{c_2=v2}$ (Result(sq_1)); where σ is the selection operation in the relational algebra. When sq_1 is removed, the query expression of sq_2 is rewritten:

Algorithm 5 FindCriticalNode($mqdg$, fpq, cmi, cme)

Input: (1) multiple query dependency graph $mqdg = (V, E, P, S, F_S, F_P)$; (2) a completed PQ ($fpq$); (3) current maximum impact (cmi); (4) current maximum effectiveness (cme).
Output: a set of critical nodes for fpg, a set of value pairs (id, benefit), current maximum impact, and current maximum effectiveness.
Method:
1. Initialize $cnset$, $BenefitList$, $TempimpList$, $TempeffList$ to empty;
2. **for** each SQ fsq in fpq **do**
3. $impact$ = CalculateImpact($mqdg$, fsq);
4. save $impact$ and id of fsq into $TempimpList$;
5. $effectiveness$ = CalculateEffectiveness($mqdg$, fsq);
6. save $effectiveness$ and id of fsq into $TempeffList$;
7. **end for**
8. imp_{max} = max{cmi, maximum value of impact in $TempimpList$};
9. eff_{max} = max{cme, maximum value of effectiveness in $TempeffList$};
10. **for** each SQ fsq in fpq **do**
11. $impact = impact$ of fsq in $TempimpList$;
12. $effectiveness = effectiveness$ of fsq in $TempeffList$;
13. $benefit = (impact/impact_{max}) * (effectiveness/eff_{max})$;
14. **if** $benefit > THRESHOLD_B$ **then**
15. add fsq into $cnset$;
16. add id and $benefit$ of fsq into $BenefitList$;
17. **end if**
18. **end for**
19. return $cnset$, $BenefitList$, imp_{max}, and eff_{max}.

Algorithm 6 RemoveAndTransfer($mqdg$, n)

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

$\sigma_{c_2=v2}$ ($\sigma_{c_1=v1}$ (R1)). In the second step, new directed edges are generated from each direct parent node t of n to each of its direct child node (lines 5 – 7). Essentially, the tables represented by the direct parent nodes of n are added to the domain of each of its direct child nodes. In the third step, all the edges associated with n are safely removed (line 9). In the last step, n and its relevant information are finally removed (line 10).

Let us use an example to illustrate how to remove a node and transfer all its dependency relationships in an MQDG using Algorithm 6. Assume that we are given an MQDG as shown in Fig. 4. The set of SQs in the figure includes:

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

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

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

Next, directed edges are generated from each direct parent node of sq_6 to each direct child node of sq_6. In this example, the edges are generated from sq_1 to sq_8, sq_2 to sq_8 and sq_5 to sq_8. After that all edges associated with sq_6 are removed and finally, sq_6 is removed. The resulting MQDG is shown in Fig. 5.

3.4 Critical node view space maintenance

The last issue we want to discuss in this section is how to insert identified critical nodes into the critical node view space (CNS). As we mentioned in Section 2.3, the CNS stores all the materialized views for the critical nodes. The size of the CNS is constrained. Therefore, when the CNS overflows, we have to make a decision to remove some views to free space for accommodating new critical nodes (views).

A straight-forward way is to apply a greedy strategy. The main idea is as follows: we first sort all the new critical nodes, which are ready to be inserted into the CNS, according to their benefit values. Next, we add as many critical nodes with the largest benefits as possible until the next critical node cannot be accommodated in the CNS. After that, the benefit of the node c whose benefit is the largest among the remained new critical nodes and the benefit of the node

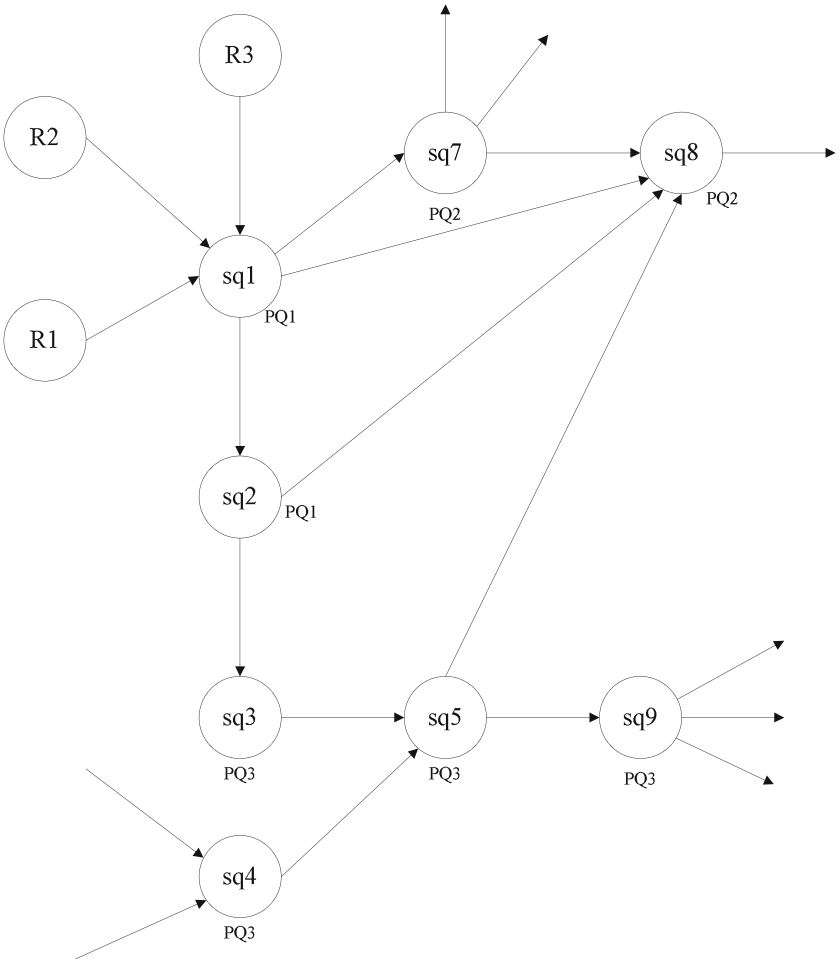

Fig. 5 An example of MQDG after sq_6 is removed

v whose benefit is the smallest in the CNS are compared. If the benefit of c is greater than that of v, v is replaced by c if possible. This process continues until no node in the CNS can be replaced. Let us consider the example in Fig. 6. The candidate critical nodes after sorting are c_1, c_2, c_3, and c_4. c_1 and c_2 are firstly added into the CNS. The remaining space cannot accommodate c_3. Thus, the benefit of c_3 is compared with that of v_4 in the CNS. As a result, c_3 is more beneficial and v_4 is replaced. Next, c_4 is considered. Since the benefit of c_4 is too small to replace any existing node in the CNS, the insertion procedure stops, and c_4 is discarded.

Note that the benefit of a new critical node nc and the benefit of an existing critical node ec in the CNS are estimated at different time and based on different MQDGs. The nodes and edges in an MQDG are updated quite

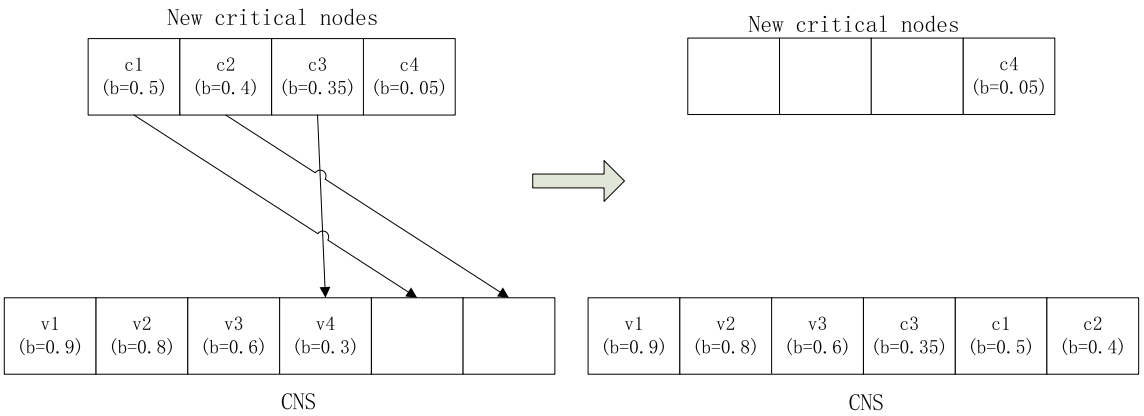

Fig. 6 An example of the greedy strategy

often. Therefore, to adopt the greedy strategy, we have to re-estimate the benefit of each ec based on the current MQDG.

In addition, we have to address a replacement failure problem. Let us consider the following scenario. When the CNS has no space to save the next new critical node nc, the benefit of nc and that of the existing critical node ec_1 whose benefit is the smallest in the CNS are compared. Assume that the benefit of nc is larger. Using the above greedy strategy, ec_1 should be replaced by nc. But the replacement process may fail. The reason for that is as follows: after removing ec_1, the available space in the CNS is still not enough to accommodate nc. In this case, we adopt a revised replacement strategy. Before actually replacing the nodes, we examine whether the replacing process can be done successfully. If so, ec_1 is replaced by nc. Otherwise, the next existing critical node ec_2 whose benefit is the second smallest in the CNS is checked. The benefit of nc and the total benefit of ec_1 and ec_2 are compared. If the latter is not smaller than the former, the insertion process stops. Otherwise, we examine whether replacing both ec_1 and ec_2 by nc can be done successfully. If so, ec_1 and ec_2 are replaced by nc. Otherwise, we check to see if replacing three existing critical nodes by nc can be done, and so on.

The high-level flowchart of the above critical nodes insertion process is described as Process 2 and Process 2.1(greedy strategy) in Fig. 3. The detailed algorithm, which is invoked in Algorithm 1, is shown as follow.

Algorithm 7 inserts a set of new critical nodes into the CNS. It first checks that if the remaining space in the CNS is large enough to accommodate all the new critical nodes (line 1). If so, all the new critical nodes are inserted directed (lines 2 - 4). Otherwise, the remaining work of the algorithm can be done in three phases. The first phase is called the preprocessing phase. The benefit values for all existing critical nodes are updated using the current MQDG by invoking a function UpdateBenefit() (line 6). Next, according to the benefit values, the new critical nodes are sorted in the descending order and existing critical nodes are sorted in the ascending order (line 8). The second phase is called the insertion phase. The algorithm inserts as many new critical nodes as possible until not enough space left (lines 10 – 12). Next, a recursive function RecursiveInsertion() is called to

Algorithm 7 CriticalNodesInsertion ($mqdg$, $cnset$, vs, $BenefitList$, cmi, cme)

Input: (1) multiple query dependency graph $mqdg = (V, E, P, S, F_S, F_P)$; (2) a set of new critical node ($cnset$); (3) view storage (vs) including temporary node view space (tns) and critical node view space (cns); (4) a set of value pairs (id, benefit) for new critical nodes ($BenefitList$); (5) current maximum impact (cmi); (6) current maximum effectiveness (cme).
Output: (1) revised vs; (2) revised $mqdg$.
Method:

```
   /* the remaining space in the CNS is enough and insert all the new critical nodes into the CNS*/
1. if the total size of nodes in cnset ≤ the remaining space of cns then;
2.   for each node n in cnset do
3.     move n (i.e., its result table and related information) from tns to cns;
4.   end for
   /* the remaining space in the CNS is not enough*/
5. else
     /* update the benefits for the existing critical nodes in the CNS*/
6.   ecBenefitList = UpdateBenefit(mqdg, cns, cmi, cme);
7.   ncBenefitList = BenefitList;
8.   sort ncBenefitList in descending order and ecBenefitList in ascending order;
9.   for each node(id) n in ncBenefitList do
     /*the CNS can accommodate the node*/
10.    if size(n) ≤ the remaining space in CNS then
11.      move n (i.e., its result table and related information) from tns to cns;
12.      remove n(including id and benefit) from ncBenefitList;
     /*the CNS cannot accommodate the node*/
13.    else
14.      benefit_n = benefit of n from ncBenefitList;
15.      m = the first node(id) from ecBenefitList;
16.      benefit_m = benefit of m from ecBenefitList;
17.      initialize NodeList and add m into NodeList;
       /*replace one or more existing critical nodes in the CNS by a new critical node. */
18.      (flag, ecBenefitList) = RecursiveInsertion(n, NodeList, vs, benefit_n, benefit_m, ecBenefitList, mqdg);
19.      if flag == true then
20.        remove n(including id and benefit) from ncBenefitList;
21.      else
22.        break;
23.      end if
24.    end if
25.  end for
     /*clear the new critical nodes which are not accommodated in the CNS.*/
26.  for each node(id) m in ncBenefitList do
27.    RemoveAndTransfer(mqdg, m);
28.  end for
29. end if.
```

decide whether to insert a new critical node to replace one or more existing critical node in the CNS (lines 13 – 24). The third phase is called the cleaning phase. For all the remaining new critical nodes which are not inserted into the CNS, the algorithm calls another function RemoveAndTransfer() to safely remove them from the MQDG (lines 26 – 28). The invoked functions UpdateBenefit() and RecursiveInsertion() are given as follow.

Algorithm 8 UpdateBenefit ($mqdg, cns, cmi, cme$)

Input: (1) multiple query dependency graph $mqdg = (V, E, P, S, F_S, F_P)$; (2) critical node view space (cns); (3) current maximum impact (cmi); (4) current maximum effectiveness (cme).
Output: a set of value pairs (id, benefit) for existing critical nodes.
Method:
1. initialize $BenefitList$ to empty;
2. **for** each critical node n in cns **do**
3. $impact$ = CaculateImpact($mqdg, n$);
4. $effectiveness$ = CaculateEffectiveness($mqdg, n$);
5. $benefit = (impact/cmi)^{\alpha} * (effectiveness/cme)^{\beta}$;
6. save (id of n, $benefit$) into $BenefitList$;
7. **end for**
8. return BenefitList.

Algorithm 8 is a function to update the benefits for all the existing critical nodes in the CNS using the given MQDG. It first updates the impact and the effectiveness for each existing critical node n in the CNS (lines 3 – 4). Next, it applies the benefit estimation model to calculate the potential benefit for each n and save (node id, benefit) value pairs into a benefit list (lines 5 – 6). Finally, the benefit list is returned (line 8).

Algorithm 9 is a recursive function to decide whether replacing a set of existing critical node in the CNS by a new critical node. A node list $NodeList$, which contains the identifiers of a list of existing critical nodes, is one of the inputs of this function. If the benefit of the new critical node n is larger than the total benefit of nodes in $NodeList$ (line 5), the size of n and the size of the available space in the CNS (including the size of the remaining space in the CNS and the total size of nodes in $NodeList$) are compared. If the size of n is larger, it implies that the new critical node cannot be accommodated into the CNS even if some existing critical nodes (the nodes in $NodeList$) in the CNS are removed. Therefore, another existing critical node is added into $NodeList$ and the function calls itself to decide whether replacing a new set of existing critical nodes ($NodeList$) in the CNS by n (lines 6 – 11). If the size of the available space in the CNS is larger, it means that n can be inserted successfully. Thus, the insertion process is done as follows: first, the nodes in $NodeList$ are removed from the MQDG and the CNS (lines 14 – 15). Next, n is inserted into the CNS and the insertion success flag is set and returned (lines 18 – 20). Otherwise, the benefit of n is equal or smaller than the total benefit of nodes in $NodeList$, it means that the insertion process cannot be done successfully. Hence, an insertion failure flag is returned (lines 22 – 23).

Note that the greedy strategy we discussed above seeks a locally optimal solution. There may be a solution that is better than the one found. Let us consider the following example. Assume that $size(CNS)$ is 10. There are three views in the CNS: v_1, v_2, and v_3. $size(v_1)$, $size(v_2)$, and $size(v_3)$ are 2, 4, and 3, respectively. $benefit(v_1)$, $benefit(v_2)$, and $benefit(v_3)$ are 4, 3, and 2, respectively. Three new critical nodes c_1, c_2, and c_3 are ready to be inserted. $size(c_1)$, $size(c_2)$, and $size(c_3)$ are 3, 2, 2, respectively. $benefit(c_1)$, $benefit(c_2)$, and $benefit(c_3)$ are 3, 2.5, and 2, respectively. Using the greedy strategy, $c1$ is considered to be inserted first since its benefit is the largest among the three. However, the remaining space in the CNS is not enough to accommodate c_1. Thus, $benefit(c_1)$ is compared with $benefit(v_3)$ (v_3 has the smallest benefit in the CNS) and v_3 is replaced. After that, the insertion process stops. The total benefit of the nodes in the CNS is: 4+3+3=10. However, there exists a better solution, i.e., replacing v_3 by c_2 and c_3 instead of c_1. In this case, the total benefit is: $4 + 3 + 2.5 + 2 = 11.5$.

We notice that the problem of maximizing the total benefit of the critical nodes in the CNS is similar to the classic knapsack problem, i.e., we have a set of items (existing critical nodes and new critical nodes) which are ready to be added into a bag (CNS). Each item has a value (benefit) and a weight (size). The total weight of items (total size) is larger than the weight of the bag (size of the CNS). The target is to find a subset of items which can be accommodated in the bag and the total value is maximized. The most popular solution for the knapsack problem is to apply a dynamic programming (DP) algorithm. Therefore, in our work, we propose another insertion method based on the DP technique.

The only difference between our problem and the knapsack problem is that, in our problem, most items (existing critical nodes) are in the bag (CNS) before the algorithm starts while in the knapsack problem, all the items are not in the bag at the beginning. Hence, to solve the knapsack problem, we find an optimal subset solution based on all the items and only insert the items in the subset into the bag. However, in our problem, we do not need to consider all the items (existing nodes and new nodes). Some popular items which are in the bag (popular existing critical nodes) may never be moved out (replaced by the new critical nodes). In this case, we only consider those unpopular existing critical nodes (c_1) and all the new

Algorithm 9 RecursiveInsertion($n, nodelist, vs, benefit_n, benefit_t, ecBenefitList, mqdg$)

Input: (1) a new critical node (n); (2) an existing critical node (m); (3) a list of existing critical nodes ($NodeList$); (4) view space (vs) including temporary node view space (tns) and critical node view space (cns); (5) the benefit of node n ($benefit_n$); (6) the total benefit of nodes in $NodeList$ ($benefit_t$); (6) a set of value pairs (id, benefit) for the existing critical nodes ($ecBenefitList$); (7) multiple query dependency graph $mqdg = (V, E, P, S, F_S, F_P)$.
Output: a boolean value, a set of value pairs (id, benefit).
Method:
1. $size_n$ = size of n;
2. $size_t$ = total size of all the nodes in $NodeList$;
3. $freespace$ = remaining space in cns;
4. $flag$ = false;
/*the benefit of the new critical node n is larger than that of one or more existing critical node*/
5. **if** $benefit_n > benefit_t$ **then**
/* node n cannot be accommodated in the CNS after removing one or more existing critical node */
6. **if** $size_n > size_t + freespace$
7. let m = last node in $NodeList$;
8. find the next node t following m in $ecBenefitList$;
9. add t into $NodeList$;
10. $benefit_t = benefit_t$ + benefit of t from $ecBenefitList$;
11. return RecursiveInsertion($n, NodeList, vs, benefit_n, benefit_t, ecBenefitList, mqdg$);
/* replace n with one or more existing critical nodes*/
12. **else**
13. **for** each node p in $NodeList$ **do**
14. RemoveAndTransfer($mqdg$, p)
15. remove p (i.e., the result table and related information of p) from cns;
16. remove the value pair (id, benefit) for p in $ecBenefitList$;
17. **end for**
18. move n (i.e., the result table and related information of n) from tns to cns;
19. $flag$ = true;
20. return $flag$ and $ecBenefitList$;
21. **end if**
22. **else**
23. return $flag$ and $ecBenefitList$;
24. **end if**

critical nodes (c_2), and apply the DP based method on c_1 and c_2 to find an optimal subset solution. In this way, the problem size could be reduced, which makes the DP algorithm to exhibit a reasonable performance even in the worst case.

The main idea is to identify the set uec of unpopular existing critical nodes from the CNS and apply the DP based algorithm on uec and all the new critical nodes to find an optimal inserting solution. Assume that the total benefit of the new critical nodes is smaller than that of the existing critical nodes in the CNS. First, all the existing critical nodes in the CNS and all the new critical nodes are sorted in the descending order by their benefit values, respectively. Next, the CNS is divided into two subspaces cns_1 and cns_2 with the same size: s_1 and s_2. s_1 and s_2 can be adjusted slightly to make two subspaces to accommodate an integer number of nodes. Since the nodes in the CNS have an order, after dividing, the nodes with relatively large benefits are in cns_1. Hence, we consider the nodes in cns_1 as the popular existing critical nodes. Next, we determine whether the existing critical nodes in cns_2 are popular. Let us use an illustrative example in Fig. 7 to show how to determine whether the nodes in cns_2 are popular. In the example, the total benefit and the total size of the existing critical nodes in cns_2 are denoted by b_3 and s_3, respectively. We estimate how many new critical nodes can be accommodated in the remaining space of the CNS. Assume that two new critical nodes can be inserted directly, their total size and total benefit are denoted by s_5 and b_5, respectively. After that, for the remaining of the new critical nodes, we consider how many of them can replace the existing nodes in cns_2. Assume that two critical nodes are considered. Their total size and total benefit are s_6 and b_6, respectively. If b_3 is smaller than b_6, it means that replacing the existing nodes in cns_2 by the new critical nodes can bring benefit. In this case, we consider the existing nodes in cns_2 as unpopular nodes. Next, the DP based approach is applied based on the unpopular nodes in the cns_2 and all the new critical nodes. Otherwise, if b_3 is not smaller than b_6, it means some nodes in cns_2 can be considered as popular existing critical nodes. In this case, cns_2 will be divided in two subspaces cns_{21} and cns_{22}, and repeat the same analysis on subspace cns_{22}.

In general, let $v_1, v_2, ..., v_k$ ($k \geq 0$) be the existing critical nodes[1] in the (current) CNS cns, listed in the descending order of their benefit values; let $c_1, c_2, ..., c_t$ ($t \geq 1$) be the new critical nodes, listed in the descending order of their benefit values. Clearly, the free space size for cns is:

$$size(free\ cns) = size(cns) - \sum_{j=1}^{k} size(v_j). \quad (4)$$

[1] If $k = 0$, there is no existing critical node in cns. Assume $\sum_{j=1}^{0}(...) = 0$ in such a case.

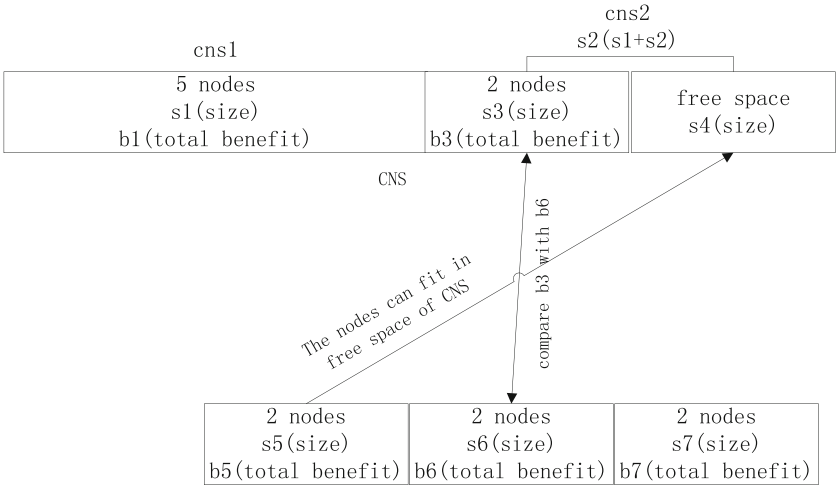

Fig. 7 An example of the DP based insertion method

Let s be the largest integer[2] in $[0, t]$, satisfying:

$$\sum_{i=1}^{s} size(c_i) \leq size(free\ cns). \tag{5}$$

If $s = t$, all the new critical nodes can fit in the free space of cns. In such a case, no DP based algorithm is needed – the problem has been solved. If $s < t$ and the following condition holds:

$$\sum_{c_i \in X} benefit(c_i) < \sum_{j=1}^{k} benefit(n_j), \tag{6}$$

where[3]

$$X = \Bigg\{ c_{i_n} \mid s+1 \leq i_n \leq t \ \ AND \ \ 1 \leq n \leq m \ \ AND \ \ i_u < i_w (for\ any\ u < w) \ \ AND \ \ i_m\ is\ the\ 1st\ integer\ in\ [s+1, t]\ such\ that \sum_{n=1}^{m} size(c_{i_n}) \leq \left[size(cns) - \sum_{i=1}^{s} size(c_i) \right] \ AND \sum_{n}^{m} size(c_{i_n}) + size(c_j) > \left[size(cns) - \sum_{i=1}^{s} size(c_i) \right] \ for\ any\ j \in [i_m+1, t] \Bigg\}, \tag{7}$$

we split cns into two subspaces cns_1 and cns_2 with sizes:

$$size(cns_1) = \sum_{j=1}^{r} size(v_j) \tag{8}$$

where r is an integer in $[1, k]$ such that $\sum_{j=1}^{r-1} size(v_j) < size(cns)/2$ and $\sum_{j=1}^{r} size(v_j) \geq size(cns)/2$; and

$$size(cns_2) = size(cns) - size(cns_1). \tag{9}$$

When Condition (6) is true, we recursively use subspace cns_2 (in place of cns) as the current space to perform the above analysis until Condition (6) does not hold for the current space. When Condition (6) does not hold, we apply the DP based algorithm on all the new critical nodes[4] and the unpopular existing critical nodes in the current space to find an optimal subset of (new and/or existing) critical nodes to be inserted into the current space.

The high-level flowchart of the DP based critical nodes insertion process is described as Process 2 and Process 2.1(dynamic programming strategy) in Fig. 3. The detailed algorithm is shown as follows.

Algorithm 10 inserts a set of new critical nodes into the CNS. If the remaining space of the CNS is large enough to accommodate all the new critical nodes, the new nodes are directly inserted (lines 1 – 4). Otherwise, we sort all existing critical nodes in the CNS and new critical nodes in the descending order by the benefit values, respectively (line 8) and invoke a function DivideAndInsertion() to recursively determine the unpopular critical nodes in the CNS, apply a DP based method to find an optimal subset of insertion nodes, and complete the insertion process (line 9). The invoked function DivideAndInsertion() is given as follows.

Algorithm 11 recursively determines a set of unpopular existing critical nodes in cns and adopts a DP based approach to search an optimal subset of new and

[2] If $s = 0$, the leading new critical node cannot fit in the free space. Condition (5) is trivially true.

[3] Informally, X contains the first m new critical nodes $c_{i_1}, c_{i_2}, ..., c_{i_m}$ from list $c_{s+1}, c_{s+2}, ..., c_s$ that can fit in the remaining space of cns (after removing the space taken by $c_1, ..., c_s$) to the maximum capacity.

[4] Assume that any critical node that cannot fit in the current space has been removed from consideration.

Algorithm 10 CriticalNodesInsertion ($mqdg, cnset, vs, BenefitList, cmi, cme$)

Input: (1) multiple query dependency graph $mqdg = (V, E, P, S, F_S, F_P)$; (2) a set of new critical node ($cnset$); (3) view storage (vs) including temporary node view space (tns) and critical node view space (cns); (4) a set of value pairs (id, benefit) for new critical nodes ($BenefitList$); (5) the current maximum impact (cmi); (6) the current maximum effectiveness (cme).
Output: (1) revised vs; (2) revised $mqdg$.
Method:
/* the remaining space in the CNS is enough and insert all the new critical nodes into the CNS*/
1. **if** the total size of nodes in $cnset \leq$ the remaining space of cns **then**;
2. **for** each node n in $cnset$ **do**
3. move n (i.e., its result table and related information) from tns to cns;
4. **end for**
/* the remaining space in the CNS is not enough*/
5. **else**
/* update the benefits for the existing critical nodes in the CNS*/
6. ecBenefitList = UpdateBenefit($mqdg$, cns, cmi, cme);
7. ncBenefitList = BenefitList;
8. sort $ecBenefitList$ and $ncBenefitList$ in descending order;
/* invoke DP based function to insert new critical nodes.*/
9. DivideAndInsertion ($mqdg, cns, tns, ncBenefitList, ecBenefitList$);
10. **end if**

Algorithm 11 DivideAndInsertion ($mqdg, cns, tns, ncBenefitList, ecBenefitList$)

Input: (1) multiple query dependency graph $mqdg = (V, E, P, S, F_S, F_P)$; (2) the current critical node view space (cns); (3) the temporary node view space (tns); (4) a set of value pairs (id, benefit) for new critical nodes ($ncBenefitList$); (5) a set of value pairs (id, benefit) for existing critical nodes in cns ($ecBenefitList$).
Output: updated cns, tns and $mqdg$.
Method:
1. $ncBenefitList1 = ncBenefitList$, $ecBenefitList1 = ecBenefitList$;
2. $freespace = size_{cns}$ − total size of existing (critical) nodes in cns;
3. b_1 = total benefit of existing critical nodes in cns;
4. s_1 = total size of existing critical nodes in cns;
/*some new critical nodes may fit in the free space of cns (see Condition (5))*/
5. n = the first node in $ncNodeList1$;
6. $FitInSize1 = 0$;
7. **while** $FitInSize1 + size_n \leq freespace$ **do**
8. $FitInSize1 = FitInSize1 + size_n$;
9. remove n from $ncNodeList1$;
10. n = the first node in $ncNodeList$;
11. **end while**;
/*estimate the total benefit (b_2) of nodes which can replace the existing nodes in cns from the remaining new critical nodes.*/
12. $FitInSize2 = 0$, $b_2 = 0$, $s_1 = s_1 + (freespace - FitInSize1)$;
13. **for** each node m in $ncNodeList1$ **do**
14. **if** $FitInSize2 + size_m \leq s_1$ **then**
15. $b_2 = b_2 + benefit_m$;
16. $FitInSize2 = FitInSize2 + size_m$;
17. **end if**;
18. **end while**;
/*compare b_1 and b_2 to determine whether the existing nodes in cns are popular.*/
19. **if** $b_1 \leq b_2$ **then** /*the nodes in cns are unpopular */
/* combine the unpopular critical nodes and new critical nodes together.*/
20. combine $ncBenefitList$ and $ecBenefitList$ into a unified $combinedBenefitList$;
21. remove each node t with $size_t > size_{cns}$ from $combinedBenefitList$;
22. extract all nodes, benefits, and sizes from $combinedBenefitList$ into three lists: $NodeList$, $BenefitList$, and $SizeList$, respectively;
/*invoke DP algorithm with the space limit of $size_{cns}$*/
23. $cnList$ = DPChecking($NodeList, BenefitList, SizeList, size_{cns}$);
/*inserting desirable new critical nodes and removing undesirable existing critical nodes.*/
24. **for** each node t in (cns-$cnList$) **do**
25. remove t (i.e., the result table and related information) from cns;
26. RemoveAndTransfer($mqdg, t$);
27. **end for**
28. **for** each node t in ($cnList$-cns) **do**
29. move t (i.e., the result table and related information) from tns to cns;
30. **end for**
31. **for** each node t in ($ncNodelist$-$cnList$) **do**
32. RemoveAndTransfer($mqdg, t$);
33. **end for**
34. **else** /*some existing nodes in cns are still popular */
35. $actualsize_{cns_1} = 0$;
36. **while** $actualsize_{cns_1} < size_{cns}/2$ **do**
37. n = the first node in $ecNodeList1$;
38. remove n from $ecNodeList1$;
39. $actualsize_{cns_1} = actualsize_{cns_1} + size_n$;
40. **end while**
41. let cns_2 be the subspace of cns that keeps critical nodes in $ecNodeList1$;
42. DivideAndInsertion ($mqdg, cns_2, tns, ncBenefitList, ecBenefitList1$);
43. **end if**

Algorithm 12 DPChecking(*NodeList,BenefitList, SizeList, SpaceLimit*)

Input: (1) a list of canidate critical nodes ($NodeList$); (2) the list of benefits for all nodes in $NodeList$ ($BenefitList$); (3) the list of sizes for all nodes in $NodeList$ ($SizeList$); (4) a space limit ($SpaceSize$).
Output: a set of critical nodes.
Method:

```
1.  Initialize B, TraceBack and cnList;
2.  for m = 0 to SpaceLimit do
3.    B[0, m] = 0;
4.  end for
5.  num = the number of nodes in NodeList;
6.  for i = 0 to num do
7.    for m = 0 to SpaceLimit do
8.      if ((SizeList[i] < m) and (BenefitList[i] + B[i − 1, m − SizeList[i]] > B[i − 1, m])) then
9.        B[i, m] = BenefitList[i] + B[i − 1, m − SizeList[i]];
10.       TraceBack[i, m] = 1;
11.     else
12.       B[i, m] = B[i − 1, m];
13.     end if
14.   end for
15. end for
16. K = SpaceLimit;
17. for i = n downto 1 do
18.   if TraceBack[i, K] ==1 then
19.     n = NodeList[i];
20.     add n to cnList;
21.     K = K - SizeList[i];
22.   end if
23. end for
24. return cnList;
```

unpopular critical nodes to store in the CNS. Lines 1 – 4 initialize some variables and obtain necessary information about cns. Lines 5 – 11 check to see which leading new critical nodes in $ncNodeList$ can fit in the free space of cns (i.e., applying Condition (5)). Lines 12 - 18 check to see which remaining new critical nodes can substitute the existing critical nodes in cns (i.e., using Eq. 7). Lines 19 - 33 handle the case in which Condition (6) does not hold. In this case, the algorithm combines the lists of new and unpopular critical nodes into one (lines 20 – 22) and applies a DP method to find the optimal subset solution (line 23). To reduce the input size for the DP problem, the algorithm removes those critical nodes which cannot fit in the current critical space (line 21). The algorithm then removes the unselected existing critical nodes from the CNS (lines 24 – 27), moves the selected new critical nodes from the TNS to the CNS (lines 28 – 30), and discards the unselected new critical nodes (lines 31 – 33). Lines 34 – 43 handle the case in which Condition (6) holds. In such a case, the algorithm divides the current cns into two subspaces cns_1 and cns_2 based on Equations 8 and 9 (lines 35 – 41). The algorithm then recursively invokes itself for subspace cns_2 (line 42). The invoked function DPChecking() is given as follows.

Algorithm 12 is a DP based function which determines a subset of candidate critical nodes in the given input list to make the total benefit of nodes in the subset to be maximized under the given input space limit. At the beginning, two arrays B and $TraceBack$ are constructed (line 1). B is used to store the maximum benefit (combined) of any subset of critical nodes of different size limits. $TarceBack$ is used to find each critical node after the optimal benefit was reached. First, a nested loop is applied and the optimal benefit under the input space limit is computed (lines 5 – 15). After that, we use $TraceBack$ to find each critical node which was used to reach the optimal benefit (lines 16 – 23). Finally, the optimal subset of critical nodes is returned (line 24).

Note that, given a set of candidate critical nodes and a space limit, the above DP based function can guarantee to find a subset of the given critical nodes that maximizes the total benefit under the given space limit. However, the worst-case complexity of such a DP method is exponential with respect to the input problem size (i.e., the number of given candidate critical nodes in our case). The greedy strategies adopted in Algorithm 11 help reduce the input problem size. Hence, our second method using Algorithms 10-12 is essentially a hybrid DP and greedy approach for maintaining the CNS. For simplicity, we still refer to this method as the DP based approach in this paper to distinguish it from the first purely greedy one using Algorithms 7 and 9.

4 Experiments

To evaluate the performance of our technique, we conducted simulation experiments. The typical experimental results are reported in this section.

4.1 Experiments setup

Experiment programs were implemented in Matlab 2007 on a PC with Intel ® dual core (1.5 GHz) CPU and 4 GB memory running the Windows ® 7 operating system.

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

In addition, each input table of an SQ could be either an external table or a result table for an executed SQ (temporary SQ or critical SQ). The probabilities to choose an external table or a result table for an SQ were not kept the same in our experiments. It was assumed that users had a higher preference to choose the result tables for SQs over external tables for their new SQs since a user tends to utilize their previous results in their new SQs. Hence, the result tables for SQs were assigned a larger probability to be chosen.

To build the relevant multiple query dependency graph (MQDG), we recorded the starting and ending times for each PQ and the execution time for each SQ. The maximum number of PQs allowed to be executed simultaneously in the system was set to 10. The MQDG and the critical node view space (CNS) were initially set to empty. When the processing of a new PQ started, its executed SQs were added into the MQDG gradually. Each SQ not only had a chance to use the results of previous SQs from the same or other in-process PQs in the MQDG but also had a chance to use the results of critical SQs in the CNS. When a PQ pq_i was completed, we applied the model/formula introduced in Section 3.2 to estimate the potential benefit of each SQ in pq_i. After the benefits of all the SQs in pq_i were estimated, we choose those SQs which could bring sufficient potential benefits as critical SQs (nodes) and the critical nodes were saved in the CNS if possible. In the following experiments, the default setting is as follows: W_d was set to 0.5, W_E was set to 2, W_I was set to 1, α and β were set to 1, the space limit of the CNS was set to 25000 disk blocks, and the DP based approach was adopted to address the CNS maintenance issue. We conducted the following experiments according to different purposes.

4.2 Performance of the critical nodes based PQ processing approach

The first experiment was conducted to evaluate the efficiency of our critical node based PQ processing approach. The experiment compared the performance among the no-critical node based PQ (NCPQ) processing approach, the randomly picked critical nodes based PQ (RCPQ) processing approach, and the estimated critical nodes based PQ (ECPQ) processing approach. The NCPQ uses only temporary nodes (results of SQs of in-process PQs) in the MQDG for processing SQs, while the RCPQ and the ECPQ can utilize both temporary nodes and critical nodes in the MQDG

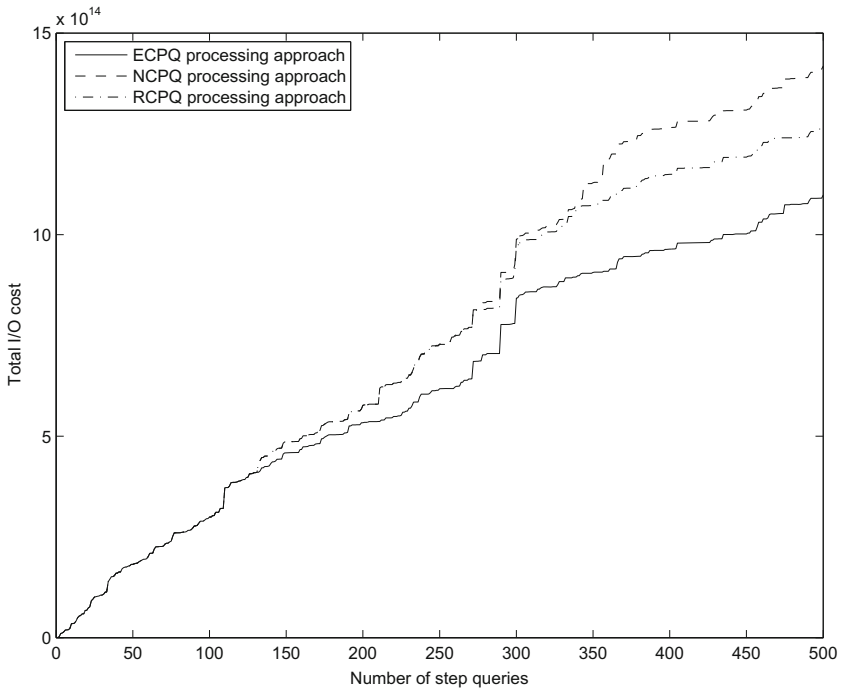

Fig. 8 Performance of different PQ processing approaches

as their inputs. The difference between the RCPQ and the ECPQ is that for the RCPQ, the critical nodes are randomly picked from the completed PQs, while, for the ECPQ, the critical nodes are selected from the completed PQs based on our benefit estimation model. The performance comparison is shown in Fig. 8.

The X-axis represents the total number of SQs executed in the test, and the Y-axis represents the I/O cost (i.e., the number of disk block accesses). The main trend of the figure is that the ECPQ approach is performed better than the RCPQ approach, and the RCPQ approach is performed better than the NCPQ approach. The reason for that is as follows: compared to the ECPQ and the RCPQ, the NCPQ has fewer materialized views (nodes) to utilize from the MQDG. Hence, the NCPQ has less chance to improve the performance of the SQs. As a result, the performance of the NCPQ is the worst among the three. For the ECPQ and the RCPQ, the numbers of views they can utilize to optimize the SQs from the MQDG are the same while the quality of the views are different. The critical nodes (views) discovered by using the ECPQ represent the results for popular SQs in the past, while the critical nodes found by using the RCPQ represent the results for randomly chosen SQs. Therefore, the ECPQ can better optimize the SQs and reach a higher performance. From the figure, we can see that at the beginning, the performance difference among the three curves is not very significant, as more and more PQs were executed, more and more critical nodes were selected to optimize the future SQs. As a result, at the right end of the figure, a significant performance improvement can be observed.

4.3 Performance of benefit estimation model with different parameters

The following two experiments were conducted to evaluate how the factors in the benefit estimation model affect the performance of our ECPQ based processing approach. Recall that, for the benefit estimation model we introduced in Section 3.2, α and β represent the importance of the impact and the effectiveness in the model, respectively, and W_d denotes the impact reducing rate as the distance increases.

In the second experiment, three value pairs (α, β) were set: (0.5, 1), (1, 1), and (1, 0.5), while other parameters remained the same. Figure 9 shows the performance comparisons by using the ECPQ among different value pairs for α and β. As we mentioned before, α and β represent the importance of the impact and the effectiveness in the benefit estimation model, respectively. The smaller the value is, the larger the corresponding importance is. From the figure, we can see that the ECPQ with α of 0.5 and β of 1 has the best performance. It indicates that giving more importance to the impact than the effectiveness in the model can achieve a better performance. From this experiment, we can see that actually, the importance of the impact is higher than that of the effectiveness in the benefit estimation model.

In the third experiment, W_d was set to different values: 0.1, 0.5, and 1, while other parameters remained the same. Figure 10 shows the performance comparisons by using the ECPQ among different values for W_d. From the figure, we can see that the performance of the ECPQ with W_d of 0.5 is

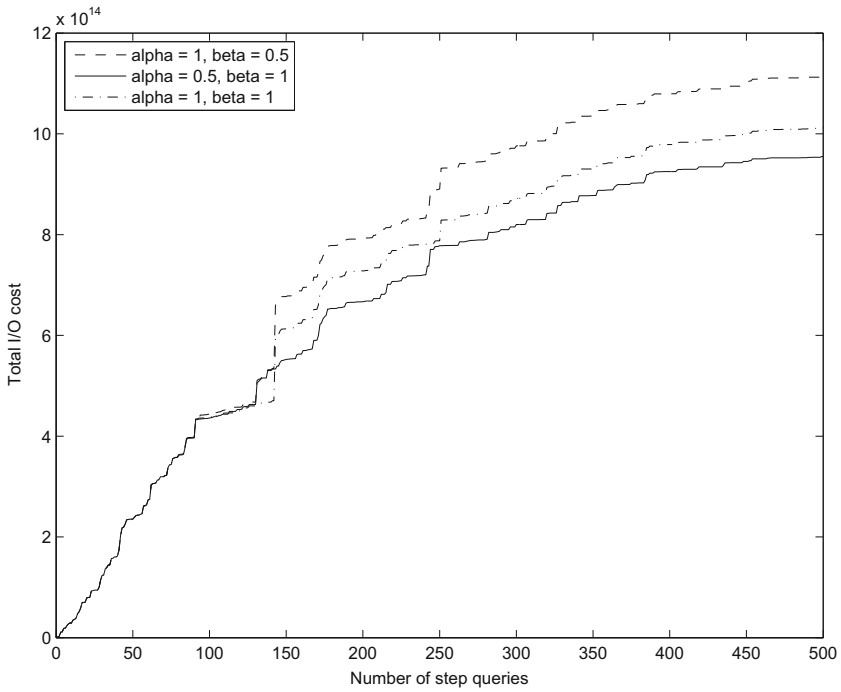

Fig. 9 Performance of $ECPQ$ with different α and β

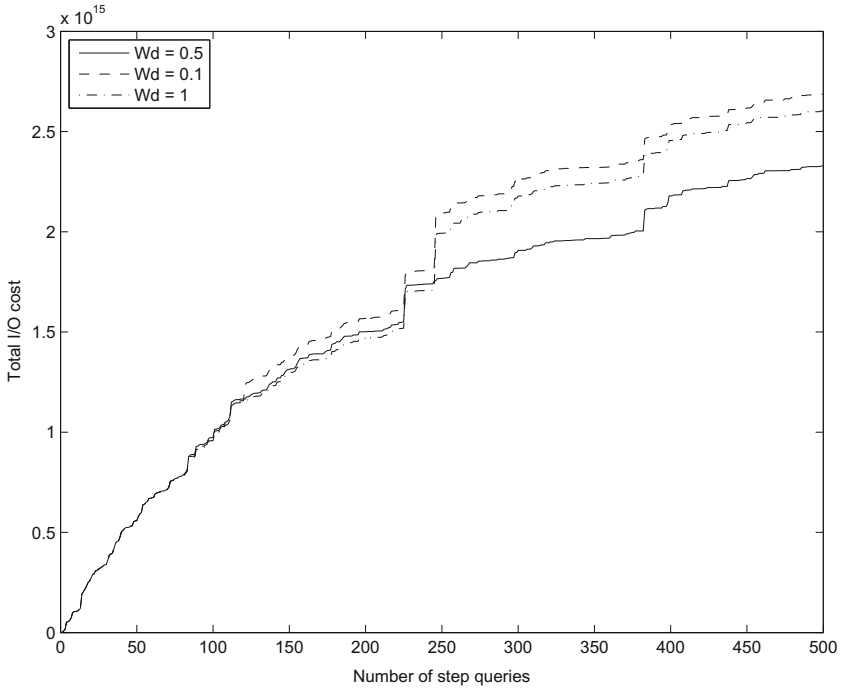

Fig. 10 Performance of $ECPQ$ with different W_d

the best among the three. The reason for that is as follows. W_d affects the impact of an SQ sq_1. It determines that how much impact the indirect child nodes of sq_1 can receive. If W_d is too small, e.g., $W_d = 0.1$, it means the indirect child nodes of sq_1 can contribute little to the impact of sq_1. Therefore, the estimation model has a trend to select nodes which have many direct child nodes but few indirect child nodes as critical nodes. In this case, the model may not be able to differentiate the following two nodes: n_1 and n_2, which have the same number of direct child nodes, but n_1 has many indirect child nodes while n_2 has no indirect child node. It is clear that n_1 is better than n_2. On the other hand, if W_d is too large, e.g., $W_d = 1$, it means that the indirect child nodes of sq_1 can contribute the same as the direct child nodes of sq_1 to the impact of sq_1. Therefore, the estimation model has a trend to select nodes which have many indirect child nodes as critical nodes. In this case, the model cannot differentiate the following two nodes, n_1 and n_2, which have the same number of child nodes (including direct and indirect), but all the child nodes of n_1 are direct child nodes, while nearly all the child nodes of n_2 are indirect child nodes. It is also clear that n_1 is better than n_2. Hence, W_d is an important affecting factor in the model and it needs to be set to a proper value, e.g., 0.5.

4.4 Performance of different CNS maintaining strategies

The fourth experiment was conducted to compare the performance behaviors between the DP based approach and the greedy based approach for maintaining the CNS. The CNS was allocated and assigned with a certain space limit (25000 disk blocks). It was initially set to empty. As more and more PQs were processed, the CNS was expanded larger and larger. Finally, the space limit was reached. Hence, a mechanism is required to decide whether the new critical nodes which were identified from a completed PQ can replace some existing nodes in the CNS. As we mentioned before, two different strategies (greedy based approach and DP based approach) are adopted.

Figure 11 shows the performance comparisons between the DP based approach and the greedy based approach. From the figure, we can see that the DP based approach outperforms the greedy based approach as we thought. At the beginning, two curves are coincided together because the CNS was not full and the space maintaining methods were not applied. After the CNS overflowed, both methods started to work. The DP based approach usually keeps a set of critical nodes with a higher overall quality in the CNS. Hence, the critical nodes kept in the CNS by using the DP based approach have a better chance to improve future SQs. As a result, an improvement can be observed towards the right of the figure.

The next experiment was conducted to compare the computing cost between the DP based approach and the greedy approach. The experiment data was the same as the previous one. The total computing time was 0.041 second by using the DP based approach and 0.0029 second by applying the greedy approach. The total execution time for the 100 tested

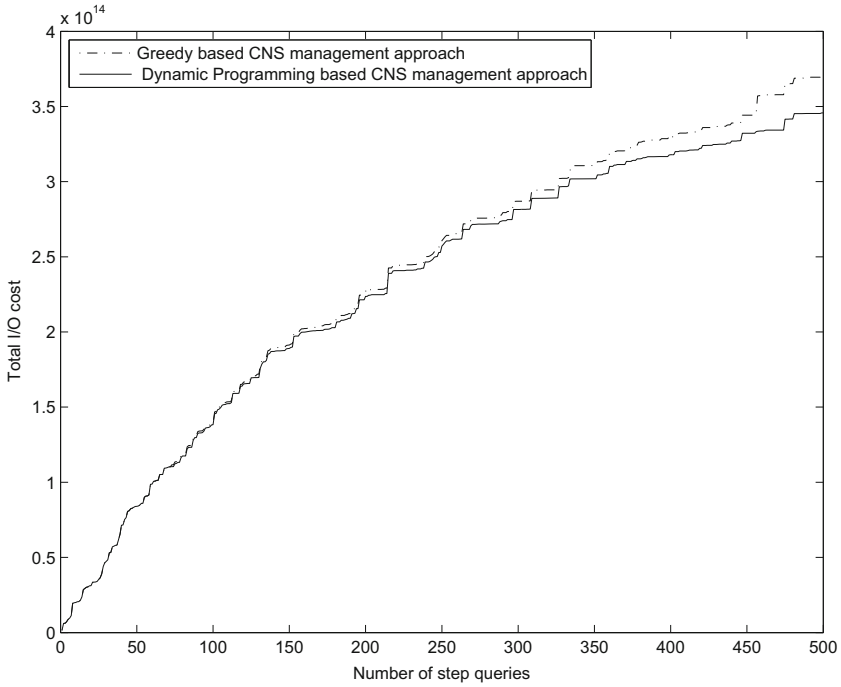

Fig. 11 Performance of CNS maintenance methods

PQs in both cases was 1.8 second (average in 10 runs). From the experiment, we can see that the computing time by using the DP based approach is much higher than that by applying the greedy approach. However, compared to the total query execution time, the cost for the dynamic programming based method is still acceptable.

4.5 Performance of differenct space limits of CNS

We varied the space limit for the CNS in this experiment and wanted to see how it would affect the performance of the ECPQ. For a given space limit, when the CNS overflowed, the DP based approach was used to maintain the

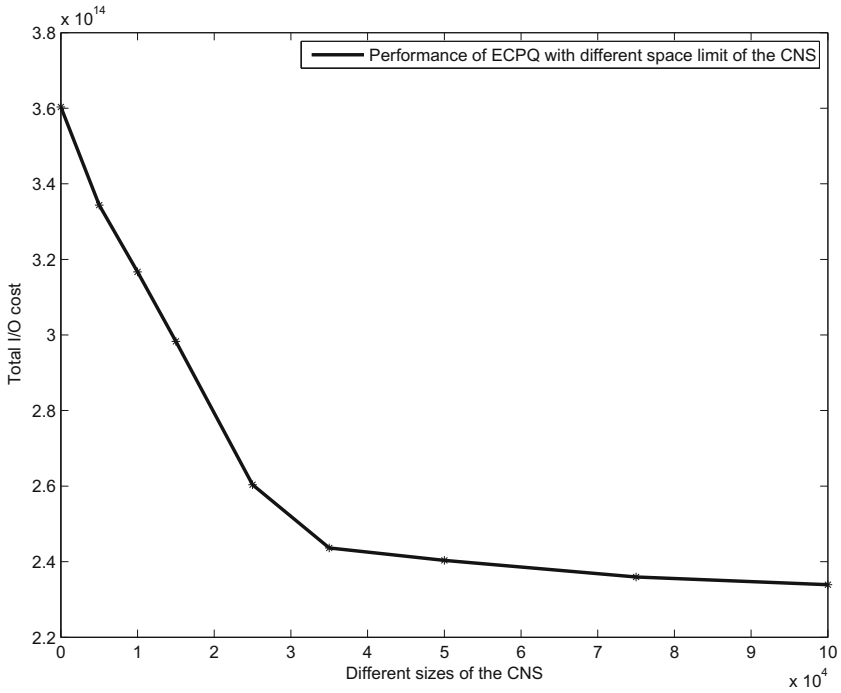

Fig. 12 Performance behavior for different space limit of CNS

CNS. The motivation for doing this experiment was as follows. This study would help us find an appropriate solution to balance the time complexity and the space complexity. In this experiment, the space limit for the CNS was varied between 0 and 100000 disk blocks. The performance behavior was shown in Fig. 12. From the figure, we can see that, the general trend of the performance curve is that, as the space limit for the CNS increases, the performance becomes increasingly better. Furthermore, we noted that, at the beginning, the cost decreases sharply. It means that increasing a small space limit could bring a dynamically improved performance. However, as the space limit continues increasing, the performance curve becomes more and more flat. It is observed that a balanced solution for our experiment case is about 35000 disk blocks.

5 Conclusions

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

We have introduced a multiple query dependence graph (MQDG) which captures the data source dependency relationships among the external tables, SQs of in-process PQs and critical SQs of completed PQs. The MQDG can be used to estimate the benefit of keeping an SQ result as a materialized view. The materialized views are used to improve the performance of the future SQs.

We have developed a mathematical model, which is composed of two key components: the impact and the effectiveness, to estimate the potential benefit of an SQ in a completed PQ using the multiple query dependency graph. A critical SQ is selected if its estimated benefit is sufficiently large.

We have presented a progressive query processing procedure to dynamically construct an MQDG, identify critical SQs by using the model on the MQDG, store their results as materialized views, and apply the views to efficiently process other SQs.

We have suggested a strategy to safely remove nodes from the MQDG. We have also proposed different strategies and algorithms to effectively maintain the critical node view space.

We have studied a direct greedy method and a dynamic programming method to maintain the materialized view space. The former is more efficient, while the latter can guarantee a better solution. To mitigate the high worst-case complexity issue for the dynamic programming method, we have suggested a greedy strategy to reduce the problem input size.

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

Our future work includes further improving the benefit estimation model, extending our technique to handle databases with updates, and applying the technique in a real DBMS environment.

Acknowledgment This work was partially supported by the IBM Canada Software Laboratory and The University of Michigan. The preliminary results of this work were presented at the 2011 Conference of the IBM Centre for Advanced Studies on Collaborative Research – CASCON'11 (Zhu et al. 2011), Toronto, Nov. 7-10, 2011.

References

Babu, S., & Bizarro, P. (2005). Adaptive query processing in the looking glass. In *Proceedings of CIDR conference* (pp. 238–249).

Agrawal, S., Chaudhuri, S., Narasayya, V. (2000). Automated selection of materialized views and indexes in SQL databases. In *Proceedings of VLDB conference* (pp. 391–398).

Agarwal, P.K., Xie, J., Yang, J., Yu, H. (2006). Scalable continuous query processing by tracking hotspots. In *Proceedings of VLDB conference* (pp. 31–42).

Antoshenkov, G. (1993). Dynamic query optimization in RDB/VMS. In *Proceedings of ICDE conference* (pp. 538–547).

Arion, A., Benzaken, V., Manolescu, I., Papakonstantinou, Y. (2007). Structured materialized views for XML queries. In *Proceedings of VLDB conference* (pp. 87–98).

Balmin, A., Beyer, K.-S., Cochrane, R., Pirahesh, H. (1997). A Framework for using materialized Xpath views in xml query processing. In *Proceedings of VLDB conference* (pp. 136–145).

Baralis, E., Paraboschi, S., Teniente, E. (1998). Materialized view selection for multidimensional datasets. In *Proceedings of VLDB conference* (pp. 488–499).

Blakeley, J.A., Larson, P.-Å., Tompa, F.W. (1986). Efficiently updating materialized views. In *Proceedings of SIGMOD conference* (pp. 61–71).

Babu, S. (2005). Adaptive query processing in data stream management systems. Ph.D. Dissert., Stanford Univ.

Folkert, N., Gupta, A., et al. (2005). Optimizing refresh of a set of materialized views. In *Proceedings of VLDB conference*.

Goldstein, J., & Larson, P.-Å. (2001). Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of SIGMOD conference* (pp. 331–342).

Gupta, A., Mumick, I.S., Ross, K.-A. (1995). Adapting materialized views after redefinitions. In *Proceedings of SIGMOD conference* (pp. 211–222).

Lee, J.H., Whang, K.Y., Lim, H.S., Lee, B.S., Heo, J.S. (2010). Progressive processing of continuous range queries in hierarchical wireless sensor networks. *IEICE Transactions*, *93*(7), 1832–1847.

Jang, J.S.R., & Lee, H.R. (2008). A general framework of progressive filtering and its application to query by singing/humming. *IEEE Transactions on Audio, Speech and Language Processing*, *16*(2), 350–358.

Jiang, H., Gao, D., Li, W.S. (2008). Exploiting correlation and parallelism of materialized-view recommendation for distributed data warehouses. In *Proceedings of ICDE conference* (pp. 276–285).

Jorg, T., & DeBloch, S. (2008). Towards generating etl processes for incremental loading. In *Proceedings of IDEAS'08* (pp. 101–110).

Kabra, N., & DeWitt, D.J. (1998). Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of SIGMOD'98* (pp. 106–117).

Kache, H., Han, W.S., Markl, V., Raman, V., Ewen, S. (2006). POP/FED: Progressive query optimization for federated queries in DB2. In *Proceedings of VLDB conference* (pp. 1175–1178).

Lim, H.-S., Lee, L.-G., Lee, M.-J., Whang, K.-Y., Song, I.-Y. (2006). Continuous query processing in data streams using duality of data and queries. In *Proceedings of SIGMOD conference* (pp. 313–324).

Liu, L., & Pu, C. (1997). Dynamic query processing in DIOM. *IEEE Data Engineering Bull*, *20*(3), 30–37.

Liu, Z., & Chen, Y. (2008). Answering keyword queries on xml using materialized views. In *Proceedings of ICDE conference* (pp. 1501–1503).

Lu, H., Tan, K.-L., Dao, S. (1995). The fittest survives: An adaptive approach to query optimization. In *Proceedings of VLDB conference* (pp. 251–262).

Luo, G. (2007). Partial materialized views. In *Proceedings of ICDE conference* (pp. 756–765).

Markl, V., Raman, V., Simmen, D.E., Lohman, G.M., Pirahesh, H. (2004). Robust query processing through progressive optimization. In *Proceedings of SIGMOD conference* (pp. 659–670).

Mistry, H., Roy, P., Sudarshan, S., Ramamritham, K. (2001). Materialized view selection and maintenance using multi-query optimization. In *Proceedings of SIGMOD* (pp. 307–318).

Mokbel, M.F. (2004). Continuous query processing in spatio-temporal databases. In *Proceedings of EDBT workshops* (pp. 100–111).

Papadias, D., Tao, Y., Fu, G., Seeger, B. (2003). An optimal and progressive algorithm for skyline queries. In *Proceedings of SIGMOD conference* (pp. 467–478).

Park, C.-S., Kim, M.-H., Lee, Y.-J. (2001). Rewriting OLAP queries using materialized views and dimension hierarchies in data warehouses. In *Proceedings of ICDE conference* (pp. 515–523).

Phan, T., & Li, W.S. (2008). Dynamic materialization of query views for data warehouse workloads. In *Proceedings of ICDE conference* (pp. 436–445).

Raghavan, V., & Rundensteiner, E.-A. (2010). Progressive result generation for multi-criteria decision support queries. In *Proceedings of ICDE conference* (pp. 733–744).

Re, C., & Suciu, D. (2007). Materialized views in probabilistic databases for information exchange and query optimization. In *Proceedings of VLDB conference* (pp. 51–62).

Simitsis, A.P., Vassiliadis, Sellis, T. (2005). State-space optimization of ETL workflows. *IEEE Transactions on Knowledge and Data Engineering*, *17*(10), 1404–1409.

Tang, N., Yu, J.X., Tamer zsu, M., Choi, B., Wong, K.F. (2008). Multiple materialized view selection for xpath query rewriting. In *Proceedings of ICDE conference* (pp. 873–882).

Tiakas, E., Papadopoulos, A.-N., Manolopoulos, Y. (2011). Progressive processing of subspace dominating queries. *VLDB Journal*, *20*(6), 921–948.

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

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

Xu, W., & Ozsoyoglu, Z.M. (2005). Rewriting XPath queries using materialized views. In *Proceedings of VLDB conference* (pp. 121–132).

Yang, J., Karlapalem, K., Li, Q. (1997). Algorithms for materialized view design in data warehousing environment. In *Proceedings of VLDB conference* (pp. 136–145).

Zhou, J., Larson, P., Goldstein, J., Ding, L. (2007). Dynamic materialized views. In *Proceedings of ICDE conference* (pp. 526–535).

Zhu, C., Zhu, Q., Zuzarte, C. (2010). Efficient processing of monotonic linear progressive queries via dynamic materialized views. In *Proceedings of CASCON conference* (pp. 224–237).

Zhu, C., Zhu, Q., Zuzarte, C., Ma, W. (2011). A materialized-view based technique to optimize progressive queries via dependency analysis. In *Proceedings of CASCON conference* (pp. 60–73).

Zhu, Q., Medjahed, B., Sharma, A., Huang, H. (2008). The collective index: A technique for efficient processing of progressive queries. *The Computer Journal*, *51*(6), 662–676.

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

Qiang Zhu is a Professor in the Department of Computer and Information Science at The University of Michigan, Dearborn, MI, USA. He is also an ACM Distinguished Scientist and an IBM CAS Faculty Fellow. Dr. Zhu's research has been funded by highly competitive funding sources including US National Science Foundation and IBM Corporation. He has numerous research publications in various top journals and conference proceedings in the database field including TODS, TOIS, TKDE, VLDBJ, VLDB and ICDE. 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, spatio-temporal databases, and data mining.

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

Wenbin Ma has a master degree from University of Alberta in Computer Science in 2001. He is a senior software engineer in IBM Toronto Lab. His interest is complex query optimization.