OPTIMIZATION OF PROGRESSIVE QUERIES VIA MATERIALIZED VIEWS FOR LARGE DATABASES

by

Chao Zhu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information Systems Engineering)
in the University of Michigan-Dearborn
2014

Dissertation Committee:

Professor Qiang Zhu (Chair/Advisor)
Professor William Grosky
Associate Professor Yi Maggie Guo
Associate Professor Brahim Medjahed

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Professor Qiang Zhu for his continuous support of my Ph.D. study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this dissertation. I could not have imagined having a better advisor and mentor for my Ph.D. study.

Besides my advisor, I would like to thank the rest of my dissertation committee: Professor William Grosky, Professor Brahim Medjahed, and Professor Yi Guo, for their encouragement, advice, and insightful comments.

My sincere thanks also go to IBM Canada Software Laboratory for giving me three years fellowship support for my Ph.D. research work, and Calisto Zuzarte (Senior Technical Staff Member at IBM) and Wenbin Ma (Software Engineer at IBM) for offering me the summer internship opportunities in their groups and supervising me to work on diverse exciting projects.

A special thanks to my family. Words cannot express how grateful I am to my father, Like Zhu, and my mother, Jianyuan Yu, for all of the sacrifices that they have made on my behalf. I would also like to thank all of my friends who supported me in writing, and incented me to strive towards my goal. At the end I would like express appreciation to my wife, Chen Luo, who spent sleepless nights with and was always my support in the moments when there was no one to answer my queries.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

There is an increasing demand to efficiently process emerging types of queries, such as progressive queries (PQ), on large scale databases from numerous contemporary applications including telematics, e-commerce, and social media. Unlike a conventional query, a PQ consists of a set of interrelated step-queries (SQ). A user formulates a new SQ on the fly based on the result(s) from the previously executed SQ(s). Processing PQs raises a number of new challenges. Existing database management systems were not designed to efficiently process such queries. In this dissertation, we propose a suite of novel materialized-view based techniques to efficiently process PQs. First, we propose a dynamic materialized-view based approach to efficiently processing a special type of PQs, called monotonic linear PQs. We introduce a so-called superior relationship graph to capture superior relationships among SQs of such a PQ and suggest a method to estimate the benefit of keeping the result of an SQ as a materialized view using the graph. To efficiently construct the superior relationship graph, we propose two algorithms: generating-based and pruning-based. To improve the view searching efficiency and quality, we design an algorithm with a special storage structure to store and manage the materialized views. Second, to handle generic PQs, we define a so-called multiple query dependency graph to capture the data source dependency relationships that exist among SQs and external tables of a generic PQ. Using the graph, a mathematical benefit estimation model, which takes both the impact and the effectiveness of materialization into consideration, is derived. A greedy method and a dynamic programming method to solve the view maintenance problem are proposed. Third, to efficiently find usable materialized views from the view space/set for answering a given SQ, we suggest a dynamic materialized view index method. A special index tree structure with nodes ordered by a two-level priority rule that facilitates ef-

ficient locating of different types of nodes is designed. Bitmaps encoded with special methods are also used to refine the pruning of unusable views during a search. Fourth, to support PQs in a big data environment like Hadoop, we propose an index based technique for performing a new column family join operation on Hbase tables. To efficiently process such a join operation, we suggest a multiple freedom family index. A parallel MapReduce algorithm to construct the index is developed. To perform a column family join on two Hbase tables using the indexes, we present two partitioning methods to balance the workload among map nodes in a MapReduce algorithm. The introduced column family join operation and its relevant processing technique can ensure the closure property that is essential to the processing of PQs. To examine the performance of the proposed techniques, we performed extensive empirical and theoretical analyses. Our studies show that the proposed techniques are quite promising in efficiently processing PQs. To our knowledge, our work is the first to apply the materialized-view based approach to efficiently processing progressive queries on large databases.

CHAPTER 1

Introduction

In this dissertation, we investigate the issues and methods for optimizing a new type of queries, called progressive queries, via materialized views for large scale databases.

1.1 New query type emerged in data intensive applications

In recent years, we have witnessed the emergence of many contemporary database applications, such as telematics, e-commerce, social networks, bioinformatics, business intelligence, and decision support, that have a high demand on processing/analyzing a huge amount of data. Many new challenges for managing and processing large databases for such data intensive applications have been raised. One of these challenges is how to efficiently process new types of queries on large databases [36, 84, 86, 107, 134]. The progressive queries studied recently by Zhu *et al.* in [134] represent one type of such new queries. Zhu *et al.* observed that a user often formulated a query progressively in a sequence of steps in many data intensive applications such as those mentioned above.

Let us consider an example. Assume that a traveler wants to select a set of songs from a worldwide song database containing millions of songs and lyrics to burn CDs to be played on her next trip. She first issues a query on the database to list all the songs released in the last three years. She finds that there are too many such songs in the database. She then narrows down the list by adding a condition on the genre. However, she finds that the list is still too long. Thus she further narrows down the list by adding another condition to restrict songs to those sung by several

her favorite singers and with a length less than 4 minutes. Finally, she finds a reasonable (not too large) set of songs she liked to enjoy for her trip.

Planning a sightseeing trip by a traveler is another example. Assume that the traveler is planning for a trip to see the scenes of some historic sites in China. In the first step, she first issues a search on a large travelling 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 she does not want to go through each returned entry to determine if she is interested in visiting the corresponding site. Thus, in the second step, 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, 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. Assume that the traveler does not find enough interesting sites from the chosen cities, she may add or change the desired cities and reuse the result from the second step. After several steps, 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 in the following step(s), which cannot be found not in the results of previous steps — implying the necessity to bring additional data sources into the searches.

A product searching at the Amazon web site represents a similar example. Assume that a user desires to buy a suitable laptop. In the first step, she issues a search to list all the laptop sales at the web site (database). After the result is returned, the user may realize that the result set is too large, and she does not want to scan all the returned sales by following several screens. Thus, in the second step, the user adds a further condition on the brand name to restrict the laptop sales, say from Dell. However, the result set may still be too large. Therefore, in the third step, the user further narrows down the result set by adding another condition for the price limit. Assume

that the user does not find any laptops under the given price limit condition, she may change the desired brand name of a laptop to HP or Lenovo and reuse the result for the first step. After several steps, a satisfied laptop is found. Then the user may search for some other information related to the selected laptop, e.g., the central processor frequency, the hard disk capacity, the production location, the users' comments and rankings, and so on.

Other examples of this type of queries include a biologist identifies an unknown DNA sequence via a sequence of tasks (e.g., alignment, validation, and comparison), a geo-scientist accesses massive volumes of earth science data via a number of complex multi-step queries, a decision maker explores and analyzes the relevant information from multiple data sets and in multiple steps, and a homeland security officer identifies a potential terrorist via analyzing and linking various pieces of information in many stages. It is not difficult to see that there are numerous application scenarios in which progressive queries play an important role in fulfilling users' application requirements.

1.2 Research problem

Unlike a conventional query, a progressive query (PQ) is defined as a query that is formulated in more than one step (progressively), where each step is called a step-query (SQ) [134]. To be more specific, let us assume that the underlying database is a relational one. A user submits her first SQ sq_1 on one or more external (existing) tables/relations in the database. Based on the result (table) R_1 of sq_1, the user submits a second SQ sq_2 using R_1 and/or additional external tables as its input. In general, an SQ may use the result(s) of its previous SQ(s) and/or external tables as its input. After the last SQ is submitted, a PQ is completed. Hence, the user gradually approaches her desired result by issuing a number of related SQs to the database. The following characteristics of progressive queries can be observed:

Characteristic 1: PQs are typically executed on large databases. This is due to the fact that the

applications like bioinformatics, social media networks, and e-commerce in which PQs are applied typically demand to perform analysis on a huge amount of data. For example, as of April 2014, the size of the well-known GenBank sequence database has reached about 159 GB bases [85]; the popular social media Facebook has over 3 billion pieces of content being generated on every day [24]; and FICO's falcon credit card fraud detection system manages over 2.1 billion valid accounts around the world [24].

Characteristic 2: SQs of a PQ cannot be known beforehand. Each SQ, which can be a full-fledged query on its own, is formulated on the fly by the user. The user needs to know the result(s) of the previous SQs to determine the next SQ. Thus, a PQ can never be fully known until its last SQ is completed.

Characteristic 3: SQs of a PQ are interrelated. In a PQ, each SQ is formulated based on the results of the previous SQs. Thus, the SQs in a PQ are not independent. Some relationships among the SQs exist. For example, the fifth SQ may use the results of the first and third SQs in a particular PQ.

These characteristics raise a number of challenges for processing/optimizing PQs efficiently in a database management system (DBMS). Characteristic 1 implies that the results of SQs of a PQ usually cannot fit in main memory and are typically saved as temporary tables on the disk. However, no access structures such as indexes exist for such temporary tables. Creating indexes from scratch for the results of SQs would incur much overhead. How to establish effective access methods for the results of SQs to facilitate the efficient processing of subsequent SQs presents a new challenge for query processing, which is called the *access inefficiency* challenge in our discussion. Characteristic 2 indicates that a PQ cannot be fully determined before its processing starts. This is totally different from the traditional query processing in which a query has to be given first so that its optimization/processing strategies can be decided by examining its requirements. This

phenomenon leads to a so-called the *query unpredictability* challenge in our discussion. Characteristic 3 together with Characteristic 2 entail that the results of previous SQs of a PQ have to be kept in the system so that a subsequent SQ can utilize them for its evaluation without re-executing the referenced SQs. On the other hand, Characteristic 1 imposes a requirement on managing the results of SQs effectively in the system to mitigate the high space demand. These requirements present another so-called the *result management* challenge in our discussion. A traditional DBMS was not designed to efficiently process PQs. To properly tackle these challenges, a new techniques are required.

To address the access inefficiency challenge, Zhu *et al.* proposed a so-called collective index method in [134], which can efficiently transform the indexes on the input table(s) of an SQ into the ones on the result table of the SQ without building them from scratch. The transformed indexes can be used to improve the processing efficiency of the new SQs on the result table. However, no work has been reported in the literature to tackle the query unpredictability and the result management challenges for PQs, which will be addressed in this work.

It is well known that utilizing materialized views to efficiently process queries is one of the important optimization techniques for conventional queries. The main idea of such a technique is to precompute (i.e., materialize) some user-defined views in advance so that queries using these views can be processed more efficiently without computing the views on the fly during query processing. We notice that PQs exhibit some properties that match well the working principle of a materialized-view based query optimization technique. For example, since a subsequent SQ of a PQ uses the result(s) of the previous SQ(s), the results of previous SQs can be naturally considered as materialized views. However, PQs also possess some properties that do not fall in the scope of a traditional materialized-view based approach. First, the results of all SQs of an in-process PQ have to be materialized, while, for a conventional materialized-view based method,

only a small subset of the views are usually selected to be materialized. Hence, materialized view management and searching become crucial for a materialized view method for PQs. Second, many results of SQs of a PQ are typically no longer needed after the PQ is completed. This implies that the lifetimes of many materialized views for PQs are shorter than those of traditional materialized views. A mechanism for effectively selecting useful materialized views/SQs and efficiently removing useless materialized views/SQs for completed PQs is desired. Third, the SQs of PQs are unpredictable, while the definitions of traditional views are typically given in advance for a traditional materialized-view based method. This implies that a dynamic feature is expected for a materialized-view based method for PQs. Fourth, the SQs of a PQ are related to each other, while traditional views do not necessarily possess any inter-relationships. Such (inter-) relationships should be taken into consideration for a materialized-view based method for PQs.

In this work, we explore a number of new materialized-view based query optimization techniques for PQs that utilize the special properties of such queries to tackle the query unpredictability and result management challenges. Most of our techniques are developed for a relational database environment, while some of our techniques are targeted for a non-relational big data environment.

1.3 Our approaches

First of all, we notice that a special type of PQs, called monotonic linear PQs, are frequently used in many database applications. In such a monotonic linear PQ, each SQ sq only uses the result of its (immediate) preceding SQ unless sq is the initial SQ of the PQ, which uses one or more external tables in the latter case. The first goal of our work is to develop a materialized-view based technique to efficiently process this type of PQs.

As mentioned earlier, due to the query unpredictability, the result of every executed SQ of an in-process PQ has to be kept (materialized) in the system until the PQ is completed. We can call

this type of materialization a trivial materialization. We notice that some SQs of a completed (historical) PQ may be useful for evaluating future SQs of other PQs. We need to identify such beneficial SQs and keep (materialize) their results in the system to facilitate efficient processing of other PQs. We can call this type of materialization a non-trivial one since not every SQ is selected for materialization. To determine which SQs are beneficial to other SQs, we introduce a so-called superior relationship graph to capture the superior (or inferior) relationships among the SQs of historical (monotonic linear) PQs. Using this graph, we can estimate the benefit of materializing an SQ in the graph. To efficiently construct such a graph, we adopt some heuristic rules to develop a generating-based algorithm and a pruning-based algorithm. The former is more efficient for a dense graph, while the latter is more efficient for a sparse graph. A method using this graph to dynamically select SQs for materialization is suggested.

As one can imagine, the number of such materialized views in the system can be very large. Hence, we design a special storage structure, called the relationship linked structure, to store and manage the materialized views to improve the view searching efficiency and quality. Specifically, we divide the set of materialized views into four groups. We then utilize the transitive property of the superior/inferior relationships among views to intelligently construct the storage structure and use it to efficiently find a high quality materialized view when answering a given SQ.

The materialized view set maintenance issue is solved by applying a replacement strategy based on the sizes and ages of materialized views when the space for keeping the materialized views overflows. A DBMS architecture incorporating the above technique to process PQs is suggested. Extensive experiments were carefully designed and conducted to evaluate the performance of adopted strategies and the impact of various parameters.

Although the above technique can efficiently process monotonic linear PQs, it cannot be applied to process a generic PQ since the required superior/inferior relationships among SQs may not

exist for a generic PQ. However, we notice that there exists a type of data source dependency relationships among SQs and external tables. We introduce a so-called multiple query dependency graph to capture such relationships. This graph allows us to estimate the impact of materializing an SQ on its child nodes as well as the effectiveness of such a materialization. A mathematical benefit estimation model taking both the impact and the effectiveness into consideration is derived. A number of impacting factors including the distance, the node type and the number of inputs are incorporated in the model.

When a PQ is completed, we use the mathematical model to estimate the benefit of keeping the result of each SQ of the PQ as a materialized view. The results of those SQs of the PQ with significant estimated benefits are selected as so-called critical (materialized) views and kept in a critical view space. The processing of a new SQ of a PQ can utilize the results of previous SQs not only from the same PQ but also from other in-process PQs as well as the critical views from completed (historical) PQs in the critical view space. To maintain the multiple query dependency graph, an algorithm is developed to remove non-critical SQs/nodes of a completed PQ from the graph and transfer the corresponding dependency relationships to the relevant surviving nodes in the graph.

To maintain the critical view space, we first introduce a greedy algorithm to replace some old critical views with the new ones when the space limit is reached. Although the greedy algorithm is efficient, it seeks a locally optimal solution. We notice that the problem of maximizing the total benefit of the critical views in the critical view space is similar to the classic knapsack problem. To improve the quality of critical views in the critical view space, we develop a dynamic programming based approach to solving the replacement problem for the critical view space by converting it to a knapsack problem. To mitigate the high worst-case complexity issue for a dynamic programming procedure, we adopt a greedy strategy to reduce the size of the input set of candidate critical views.

After promising materialized views are selected and saved in the view space (set), how to efficiently find usable views from the view space for answering an SQ is another important issue. To overcome the high overhead problem with the straightforward sequential search method, we develop a dynamic materialized view index method to efficiently find the views that are potentially usable for answering a given SQ. This index allows a user to search for usable critical views as well as usable temporary views (i.e., the results of SQs of in-process PQs) in the view space when answering a given SQ. It uses the input tables of an SQ as its search key. It also adopts specially encoded bitmaps to further prune unusable views during a search. Since the materialized views in the view space are dynamically generated during of the processing of PQs, our index is dynamically updated to incorporate new views. On the other hand, since temporary views are frequently expired (i.e., when the corresponding PQs are completed), our index can efficiently transform selected temporary views into critical views and remove unselected ones. A two-level priority rule is adopted to order the nodes (input tables) of the index tree in such a way that views for search/insertion/deletion can be quickly located. Efficient algorithms for index construction, maintenance, and search are developed.

The above techniques are developed for a relational database environment. We also conduct a preliminary study on supporting PQs in a popular big data environment, i.e., the Hadoop environment. A Hadoop supported database is called Hbase, which may contain one or more Hbase tables. An Hbase table consists of a set of rows with values for one or more column families. A column family can have a dynamic number of columns. In a relational environment, each SQ of a PQ operates on one or more relational tables and return a new relational table as the result. This closure property is a basic requirement to perform a PQ since the result of a previous SQ can be used as an input table for a subsequent SQ. However, most existing query techniques for Hbase tables lack of this closure property; namely, a query result is typically exported in the form of a

relational table and returned to the requesting application, rather than saved as a new Hbase table in the system for further querying. On the other hand, existing techniques perform query operations (e.g., join) at the column level rather than the column family level. We notice that many applications demand operations for Hbase tables at the column family level. We, hence, introduce tree column family level operations and focus on discussing efficient processing of the most difficult operation, i.e., the column family join. In fact, we define four types of column family joins according to different matching freedoms in the join condition. To efficiently process a column family join, we introduce a multiple freedom family index, which itself is realized via an Hbase table so that some Hbase table features such as the built-in ordering of row ids can be utilized. A parallel MapReduce algorithm is developed for constructing the index. To join two Hbase tables via the indexes on the joining column families, we examine two partitioning methods in order to achieve a balanced work load among map nodes. A MapReduce algorithm for a column family join using the indexes is suggested.

For each technique proposed in this dissertation, we conduct extensive experiments to evaluate its efficiency and/or effectiveness in various cases. Our experimental results demonstrate that our techniques are promising in efficiently processing PQs.

1.4 Dissertation organization

The remainder of this dissertation is organized as follows. Chapter 2 discusses the related work. We classify the related work into four categories in our discussion; namely, the work related to PQs, the work related to materialized views, the work related to indexes, and the work related to big data environment. Chapter 3 presents a dynamic materialized-view based approach to efficiently process a special type of PQs, called monotonic linear PQs. We first discuss various types of PQs. We then introduce a superior relationship graph for a set of monotonic linear PQs, discuss

the main properties of monotonic linear PQs and suggest a framework to process such PQs. Various algorithms for constructing a superior relationship graph, selecting promising views for materialization, and managing materialized views are presented. Chapter 4 proposes a materialized-view based approach to efficiently process generic PQs. A special graph to capture the data source dependency relationships among SQs and external tables is defined, and a mathematical model to select critical materialized views is presented. Different strategies for maintaining the materialized view set are also discussed. Chapter 5 suggests an approach to efficiently select usable materialized views from a view set for answering SQs. A new index technique for indexing materialized views, including the index structure, construction algorithm, and maintenance strategies, is described. A search algorithm using the index is presented. Chapter 6 presents our preliminary study on supporting PQs in a big data environment. Four column family join operations based on different matching freedoms for Hbase tables are defined, and a multiple freedom family index to support efficient processing of a column family join for Hbase tables in the Hadoop environment is introduced. MapReduce algorithms for constructing the index and using it to efficiently process a column family join are discussed. Chapter 7 summarizes the dissertation and discusses the future work.

CHAPTER 2

Related Work

In this chapter, we discuss the work related to our study in this dissertation. We review some query processing techniques which are kind of progressive in Section 2.1. We discuss the work related to materialized views in three categories, i.e., view selection, view matching, and view maintenance, in Section 2.2. We overview the work related to indexes in Section 2.3 and discuss the work related to the big data environment in Section 2.4.

2.1 Work related to PQs

The work that is most related to progressive queries in the literature includes query processing for continuous queries [3, 11, 70, 83], adaptive (dynamic) query optimization [7, 12, 55, 75, 76], and ETL (Extraction-Transformation-Loading) processing [53, 103, 113, 114]. Continuous queries require the repeated execution of a query over a continuous stream of data [11]. The main difference between continuous queries and progressive queries is that a continuous query is formulated at once (although data is dynamic) while a PQ is formulated in several steps. The main idea of adaptive query optimization is to exploit information that becomes available at query runtime and adapt the query plan to changing environments during execution. While the adaptive query optimization problem may be seen as "progressive" (performed at compile-time and run-time), queries are however formulated at once ("non-progressive"). Extraction-Transformation-Loading (ETL) processes are used to extract data from multiple sources, cleanse them, integrate them, and propagate them to a data warehouse incrementally. In an ETL workflow, activities/operators are chained

together. One operator uses the results of previous operators. However, all the activities/operators in an ETL workflow are programmed in advance, which is different from a PQ, although new data are incrementally added to a data warehouse. In addition, an operator in an ETL workflow tends to be much simpler than an SQ in a PQ. The latter can be a full-fledged query on its own.

Other work 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 [112]. Raghavan *et al.* presented a progressive evaluation framework ProgXe to progressively generate query results early and often for mulit-criteria decision support queries [95]. Jang *et al.* designed a methodology of progressive filtering (PF) for multimedia information retrieval, whose applications are called the melody recognition [49]. Kache *et al.* proposed a progressive optimization technique for federated queries, which are 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 [56]. 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 [90]. 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 of those techniques are however formulated at once (non-progressive).

The only previous work that directly studied efficient processing of progressive queries is the collective index technique proposed by Zhu, et al. in [134]. The main idea is to construct a special index structure that allows a collection of member indexes on an input relation of an SQ to be efficiently transformed into indexes on the result relation, which can be used to speed up the subsequent SQs. In this work, we explore another approach to efficiently processing PQs, i.e., studying the materialized view based techniques and the relevant issues.

2.2 Work related to materialized views

Applying materialized views to speed-up query processing has been well studied in the literature [4,14,20,30,33,34,38,43,66,80,82,96,127]. Different types of database systems were considered, including relational databases [43,66,135], object-oriented databases [5], data warehouses [16,91, 93, 121], distributed database [51], XML databases [10, 15, 51, 74, 109, 118], and others [54, 96]. Various issues were studied, including materialized view selection, materialized view matching [25,67–69,82], materialized view maintenance [39,40,126], materialized view concurrency control [78, 79], and materialized view indexing [18, 99]. Since the view concurrency control is rarely studied and the index related work is discussed in the next subsection, we are mainly focus on discussing three issues, i.e., the materialized view selection, the materialized view matching, and the materialized view maintenance in this subsection.

Most of the view selection techniques follow the paradigm of static view selection introduced by Theodoratos and Sellis [111], which selects views from a given input view set under storage or maintenance constraints. This line of work is good for cases where the materialized views are not changed over time.

Baralis *et al.* developed a technique to select proper materialized views for the multidimensional datasets by considering only the relevant elements of the multidimensional lattice [16]. Agrawal *et al.* described an industry-strength tool for automated selection of materialized views for SQL workloads [4]. Liang *et al.* introduced heuristic-based algorithms to solve the view selection problem under the maintenance time constraint for data warehouses [69]. Lee *et al.* suggested a genetic algorithm to compute a near-optimal set of views to minimize the total query response time over all queries [68]. Ezeife proposed a method for selecting and materializing views, which selects and horizontally fragments a view, recomputes the size of the stored partitioned view while deciding further views to select [29]. Gupta *et al.* presented polynomial-time heuristics for selection

of views using an AND view graph, an OR view graph or an AND-OR view graph for different scenarios [37]. Chirkova *et al.* presented techniques for finding a minimum-size view set for a single query without self-joins by using the shape of the query and its constraints [25]. Aouiche *et al.* proposed a framework to exploit a clustering technique to solve the materialized view selection problem [8]. Hung *et al.* derived a cost model and efficient view selection algorithms that effectively exploit the gain and loss metrics [47]. Tang *at al.* developed a heuristic method to identify a minimal view set for a given XPath query [109]. Yang, 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 [51, 96, 121].

Another line of work is influenced by multiple query optimization techniques, with the aim of finding reusable sub-queries. Theodoratos and Sellis modeled the problem as a state space search problem. Each state is a multiple-query graph specifying Select-Join queries [111]. Mistry *et al.* presented algorithms that can be used to efficiently select materialized views to speed up workloads by exploiting common subexpressions and indices [82].

Although much work on materialized view selection has been done in the past, no technique designed to select materialized views for processing PQs, as we report in this study, has been found in the literature.

The materialized view matching issue is also considered as part of the problem of answering queries using views. In query optimization, rewriting a query using a set of materialized views may yield a more efficient query execution plan. Yang *et al.* presented a query transformation approach for answering PSJ-queries by using derived tables (materialized views) in [119], which was one of the earliest work we found in the literature for materialized view matching. Srivastava *et al.* extended the work to address aggregation queries in [106]. Park *et al.* proposed a method for rewriting a given OLAP query using various kinds of materialized aggregate views [91]. 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 could be incorporated in transformation-based optimizers [33]. Tang, Balmin, Xu, *et al.* developed different techniques for rewriting XPATH queries using materialized views [10, 15, 109, 118]. Liu *et al.* presented techniques for answering keyword queries using a minimal number of materialized views [74]. Since the view matching process for SQs of PQs is the same as that for conventional queries, to answer an SQ using views, any commonly used view matching technique can be applied.

The materialized view maintenance issue is important because of the changes to the database. Blakeley, Folkert, Gupta, *et al.* presented different materialized view updating methods considering the changes to the external tables in the database [14, 30, 38]. Zhou, Luo, *et al.* presented flexible materialization strategies which selectively materialize only a subset of rows of a table to reduce storage space and view maintenance costs [80, 127]. A conventional materialized view maintenance technique can be utilized for PQs. We will not consider the materialized view maintenance in this work. However, we consider another maintenance issue, i.e., how to maintain a set of materialized views in a view space under a given space limit, which is not considered before in the literature.

2.3 Work related to indexs

We also adopt several index techniques in our work. Many index techniques, which are used to efficiently access data objects in a database, are reported in the literature. The most well-known index structure is the B-tree, introduced by Bayer *et al.* in [17]. Its extended versions such as the B^+-tree and the B^*-tree are also widely used in many different areas.

Robinson presented a dynamic index, called the K-D-B tree, to retrieve multi-key records via range queries [98]. Guttman *et al.* described a dynamic index structure, called the R-tree, to

efficiently handle multidimensional spatial data [42]. Berchtold, Katayama, Chakrabarti, Sakurai, *et al.* proposed index structures to access high dimensional data sets [19, 21, 57, 100]. Kuo *et al.* proposed methodologies to control the access of B-tree-indexed data in a batch and real-time fashion [63]. Chan *et al.* presented the RE-tree, which is an index structure for large databases of Regular Expressions (RE) specifications [22]. Wang, Jiang, *et al.* proposed index structures for searching XML documents [52, 115, 116]. He *et al.* introduced an index structure, called Closure-tree, to support subgraph queries and similarity queries [44]. Zhang *et al.* proposed the Bed tree, a B+-tree based index structure, for string similarity searches [125]. Although how to efficiently access data objects in different types of databases was well studied in the literature, no index was designed to find usable materialized views for answering SQs as we consider in this study.

In our work, a bitmap index is utilized. A bitmap index is a special type of index that uses bitmaps and answers queries by performing bitwise logical operations on these bitmaps. A traditional table index associates with each index key value a list of row identifiers or primary keys for rows that have that value. It is well known that the list of rows associated with a given index key value can be represented by a bitmap or bit vector. In a bitmap representation, each row in a table is associated with a bit in a long string, an N-bit string if there are N rows in the table, and the bit is set to 1 in the bitmap if the associated row is contained in the list represented; otherwise the bit is set to 0. A bitmap technique works well if the number of possible key values in the index is small, while there are a large number of rows. When a large number of values exist in an index, each of the bitmaps is likely to be rather sparse, i.e., very few bits will be 1 in the bitmaps, resulting in heavy storage requirements for storing a lot of zeros. To tackle this challenge, Patrick *et al.* presented a bitmap compression approach by changing representation from bitmap to row identifier list and back [92].

Many different bitmap indexes based techniques have been proposed in the literature [23, 31,

45, 87, 104, 105, 117, 122]. Chan *et al.* presented a general framework to study the design space of bitmap indexes for selection queries and examine the disk-space and time characteristics that the various alternative index choices offer [23]. Nitsos *et al.* reported a hybrid-indexing scheme (Bitmap-Tree) that integrates the advantages of bitmap indexing and file inversion to improve the query processing efficiency and reduce the storage overhead [87]. Yoon *et al.* proposed a bitmap-indexing scheme for speeding up the access control to the XML documents [122]. Sinha *et al.* introduced adaptive bitmap indexes, which conform to space limits while dynamically adapting to the query load and offering excellent performance [104]. Sinha *et al.* proposed a multi-resolution, parallelizable bitmap index, which supports a fine-grained trade-off between storage requirements and query performance [105]. He *et al.* developed a bitmap pruning strategy for processing the iceberg query, which is a special type of aggregation query that computes aggregate values above a user-provided threshold [45]. Fusco *et al.* proposed a compressed bitmap index approach that significantly reduces both CPU load and disk consumption [31]. The difference between the conventional bitmap indexes and our bitmap based technique is that a conventional bitmap index is applied to answer queries, while our bitmap based approach is used to filtering undesirable views.

There is a substantial body of work exploring the index techniques and the view materialization techniques together. Gupta *et al.* extended the framework [111] we mentioned in Section 2.2 to accommodate index selection while the workload allows both aggregation and selection queries [111]. An index is selected only after the view it is defined over is selected. A greedy algorithm is applied to choose either a view or an index at each step to maximize the benefit per unit space. Roussopoulos presented a method to select a set of views and maintain an index for each of them to support efficient query processing [99]. The index of each view contains pointers to the tuples of the base tables used to construct the view. Kimura *et al.* adopted a form of Integer Linear Programming (ILP) to select the best set of materialized views and indexes for a given workload

under given database size constraints, taking into consideration of the effect of correlated attributes [60]. Bellatreche *et al.* introduced a technique to select optimal or near optimal join indexes for a given set of OLAP queries, where the indexes can be built on materialized views as well as dimension and fact base tables [18]. Talebi *et al.* examined the exact and inexact methods for selecting materialized views and indexes to efficiently process OLAP queries [108]. Aouiche and Darmont applied a data mining process to select candidate materialized views and indexes in data warehouse environments [9]. Graefe and Zwilling studied techniques for transaction support for indexed summary views [35]. Kuno and Graefe proposed a deferred technique to maintain indexes and materialized views [62]. 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 [93].

However, all the above work considered indexes that were built on base tables and/or materialized views to accelerate the processing of queries on the database in conjunction with materialized views. In contrast, the index technique we introduce in this work directly uses materialized views, instead of the underlying data, as indexed objects, with a goal of removing as many undesirable views as possible from consideration for view matching during query processing based on materialized views.

2.4 Work related to big data environment

Since we develop methods for performing an operation, i.e., the family join, on big data using Hadoop and MapReduce to achieve the data parallelization, we also discuss the related work on Hadoop and MapReduce here. The MapReduce framework has been well studied [26, 101]. Shim *et al.* introduced the MapReduce framework based on Hadoop, and discussed how to design efficient MapReduce algorithms [101]. Condie *et al.* described a modified MapReduce architecture

that allows data to be pipelined between operators and demonstrated a modified version of the Hadoop MapReduce framework that supported online aggregation [26].

In the literature, MapReduce related techniques can be divided into three categories. The first category includes all the techniques for completing the functionalities and/or improving the performance of MapReduce and/or Hadoop [27, 28, 65, 71, 110, 120]. Yang *et al.* added a Merge phase to MapReduce that could efficiently merge data already partitioned and sorted by map and reduce modules [120]. Tao *et al.* presented the notion of minimal algorithm, i.e., an algorithm that guaranteed the best parallelization in multiple aspects at the same time [110]. Elghandour *et al.* demonstrated a technique which managed storage and reused the intermediate results of the MapReduce workflows executed in the Pig data analysis system [28]. Lim em et al. introduced an execution plan space for MapReduce workflows generated by popular workflow generators [71]. Laptev *et al.* proposed a non-parametric extension of Hadoop which allowed the incremental computation of early results for arbitrary work-flows [65]. Dittrich *et al.* discussed different data management techniques used in MapReduce [27].

In the second category, the performance analysis issues of MapReduce are of interest [48, 50, 58, 94]. Quiane-Ruiz *et al.* proposed a family of Recovery Algorithms for Fast-Tracking (RAFT) MapReduce [94]. Jahani *et al.* proposed a technique, which automatically analyzed MapReduce programs and applied appropriate data aware optimizations [48]. Khoussainova *et al.* presented a system that enabled users to ask questions about the relative performances of pairs of MapReduce jobs [58]. Jiang *et al.* conducted a performance study of MapReduce (Hadoop) on a 100-node cluster of Amazon EC2 with various levels of parallelism [50].

The third category contains all the techniques for applying MapReduce to accomplish new tasks or solve new problems on Hadoop. Our proposed technique for PQs belongs to this category. Liu *et al.* demonstrated a new dimensional Extract-Transform-Load (ETL) programming framework

that used MapReduce to achieve scalability [73]. Aly *et al.* developed a prototype implementation of the MapReduce framework for answering continuous queries over streams of data [6]. Ghoting *et al.* proposed SystemML in which machine learning algorithms were expressed in a higher-level language and were compiled and executed in a MapReduce environment [32]. Pansare *et al.* reported the methods of including online aggregation into a MapReduce system for large-scale data processing [89]. Lam *et al.* described a framework like MapReduce, but specifically developed for fast data [64]. He *et al.* presented a big data placement structure called RCFile (Record Columnar File) and its implementation in the Hadoop system [46]. Bahmani *et al.* designed a fast MapReduce algorithm for Monte Carlo approximation of personalized PageRank vectors of all the nodes in a graph [13]. Kolb *et al.* proposed and evaluated approaches for skew handling and load balancing in MapReduce [61].

In addition, in the third category, strategies for processing different types of joins using MapReduce were also well studied [1, 2, 41, 59, 77, 81, 88, 97, 102, 123, 124]. Afrati *et al.* examined strategies for joining several relations in the map-reduce environment [2]. Gupta *et al.* suggested an approach for processing multi-way spatial joins on map-reduce platform [41]. Okcan *et al.* proposed a join model for mapping arbitrary join conditions to Map and Reduce functions [88]. Vernica *et al.* proposed an approach for end-to-end set-similarity joins [97]. Silva *et al.* presented a multi-round MapReduce based algorithm, which was called MRSimJoin, to efficiently solve the Similarity Join problem [102]. Afrati *et al.* proposed several algorithms for finding all pairs of elements from an input set that meet a similarity threshold [1]. Kim *et al.* designed algorithms for top-k similarity join using the MapReduce framework [59]. Metwally *et al.* proposed a so-called V-SMART-Join, a scalable MapReduce based framework for discovering all pairs of similar entities [81]. Lu, Zhang, *et al.* introduced techniques for performing kNN join on large data using MapReduce [77, 123]. Zhang *et al.* studied how to process multi-way Theta-join queries using

MapReduce from a cost-effective perspective [124]. However, none of the above techniques were designed for family joins in Hbase tables that are introduced in this work. Furthermore, our work is in the context of PQs, where the closure property ensures that the join result is also an Hbase table. To our knowledge, our work is the first to efficiently process family joins on Hbase tables using MapRedue.

CHAPTER 3

Dynamic Materialized-view Based Approach For Monotonic Linear PQs

In this chapter, we introduce a dynamic materialized-view based approach to efficiently process monotonic linear PQs. Some background knowledge is discussed first. Different types of progressive queries is introduced in Section 3.1. A superior relationship graph that is used in our technique for processing monotonic linear PQs is defined in Section 3.2. The main properties of the monotonic linear progressive query are discussed in Section 3.3. After that, the main processing procedure is shown in Section 3.4. Efficient strategies to create and update a superior relationship graph are discussed in Section 3.5. The algorithm to decide whether to materialize a view is discussed in Section 3.6. The storage structure and algorithms to manage the set of materialized views (SMV) are given in Section 3.7. The view search algorithms are described in Section 3.8, and the experimental results are discussed in Section 3.9

3.1 Types of progressive queries

A progressive query (PQ) is formulated in several steps. Each step, referred to as a step-query (SQ), is executed over one or more tables/relations and returns one table as a result. Result(SQ) and Domain(SQ) represent the result table of the SQ and the set of tables on which the SQ is executed, respectively. A SQ can be executed on either the result table(s) returned by the previous SQs and/or other external base table(s). In [134], Zhu *et al.* classified the progressive queries into

the following three types:

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

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

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

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

3.2 Superior/inferior relationship and Superior relationship graph

In our dynamic materialized view technique for monotonic linear PQ, we utilize a so-called superior relationship graph (SRG) to determine if the result of a SQ under consideration should be materialized as a view. A superior relationship graph captures the superior (or inferior) relationships among the SQs for historical progressive queries.

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

a superior relationship from sq_1 to sq_2, where sq_1 is called a superior of sq_2 and sq_2 is called an inferior of sq_1 .

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

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

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

Example 1. Given the following four tables:

PAPER(P#, Title, FirstAuthor, PublishYear),

AUTHOR(A#, A_Fname, A_Lname, Area),

EDITOR(E#, E_Fname, E_Lname, Area),

REVIEW(E#, P#, Date),

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

Progressive Query 1 (pq_1):

sq_1: $\pi_{Title,\ PublishYear,\ A\_Lname}(PAPER \overset{\bowtie}{}_{FirstAuthor=Aid}\ AUTHOR)$,

sq_2: $\pi_{Title,\ A\_Lname}(\sigma_{PublishYear=2009}(Result(sq_1)))$,

sq_3: $\pi_{Title}(\sigma_{A\_Lname='Smith'}(Result(sq_2)))$.

Progressive Query 2 (pq_2):

sq_4: $\pi_{E\_Lname,\ Title,\ PublishYear}(PAPER \overset{\bowtie}{}_{PAPER.P\#=REVIEW.P\#}$

$(REVIEW \overset{\bowtie}{}_{REVIEW.E\#=EDITOR.E\#}\ EDITOR))$,

sq_5: $\sigma_{PublishYear>2008}((Result(sq_4))$,

sq_6: $\pi_{Title}(\sigma_{PublishYear=2009}(Result(sq_5)))$.

Progressive Query 3 (pq_3):

sq_7: $\pi_{Title,\ PublishYear}(\sigma_{PublishYear>2008}(PAPER))$,

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

Fig. 3.1 shows the superior relationship graph for these three PQs. From the figure, we can see that four SQs would benefit from materializing the result of sq_1. The number of out-going edges for a node v is the weight of v, which is not shown in the figure. Clearly, the weights of the nodes in an SRG can be calculated once the graph is given.

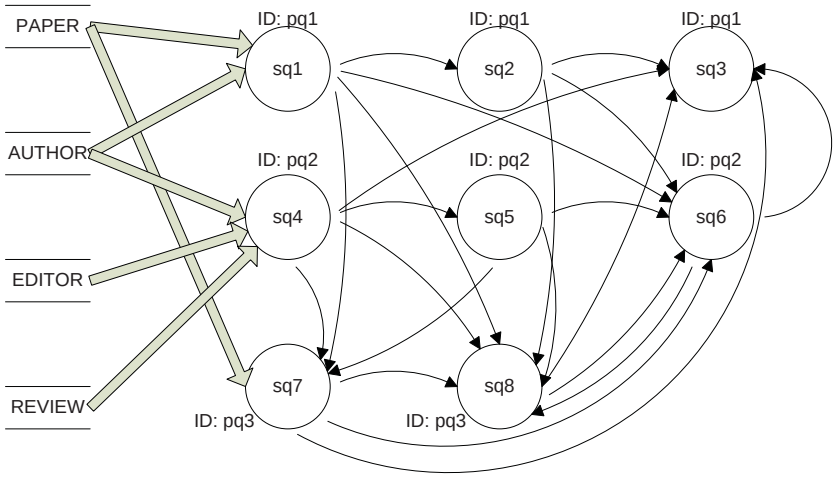

Figure 3.1: Superior relationship graph of Example 1

3.3 Main properties of monotonic linear PQs

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

Property 1: Result(sq_i) $\sqsupseteq$ Result(sq_j) if i<j and sq_i, sq_j are two SQs belonging to the same PQ, where $\sqsupseteq$ indicates that the right operand can be completely derived from the left one.

According to the definition, the current SQ only uses the result table returned by the previous SQ. Hence, if sq_j is one of the subsequent SQs of sq_i, every tuple in Result(sq_j) must be derivable from Result(sq_i).

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

As defined earlier, the weight of an SQ is the number of out-going edges in the SRG, which represents the benefit of materializing the result of the SQ. Based on Property 1, sq_i must be a superior of sq_j. As mentioned before, we do not consider the superior relationships between

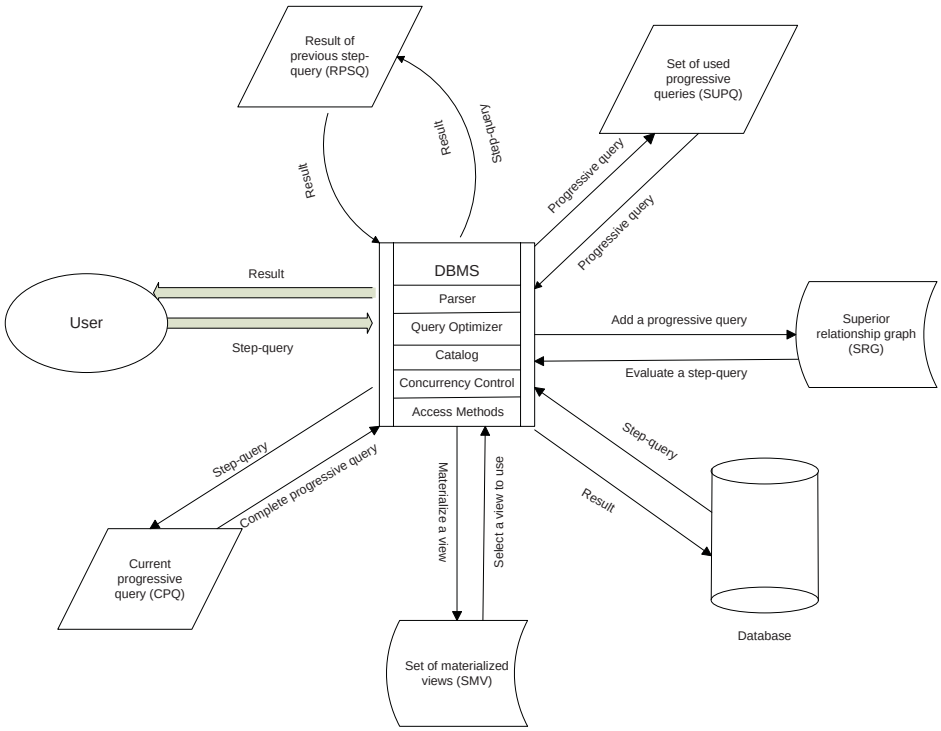

Figure 3.2: PQ processing procedure based on dynamic materialized views

two non-consecutive SQs within the same PQ when we construct the SRG. All the other superior relationships (out-going edges) for sq_j must also be valid for sq_i.

3.4 PQ processing procedure

The view materialization techniques have been popular in query optimization, as mentioned in Chapters 1 and 2. The decision for view materialization is typically based on statistic information such as access frequency. However, unlike a conventional query, a PQ is formulated as a number of inter-related SQs. Each SQ cannot be known beforehand. No one can predict what the next SQ could be. Hence, there is no prior knowledge about future user (step) queries when deciding view materialization. This situation raises a challenge (query unpredictability) to apply a materialized view based technique to efficiently process PQs.

To tackle this challenge, we introduce a dynamic materialized-view based approach for processing monotonic linear PQs. Fig.3.2 depicts the processing procedure (system architecture). There are several components involved in the procedure. The user submits one SQ at each step for the current PQ (CPQ). The current SQ (CSQ) is the one that is currently being processed in the system. The underlying database management system (DBMS) coordinates the PQ processing based on the dynamic materialized-view approach. This DBMS has all the typical modules such as the parser, catalog, query optimizer and concurrency control that a conventional DBMS has. However, these modules are enhanced to handle a PQ based on dynamically materialized views as follows. A superior relationship graph (SRG) is dynamically constructed by the system. Initially, the SRG is empty. When more and more completed PQs are dynamically added to it, it grows larger and larger. This graph is used to determine if materializing the result of the CSQ is beneficial. If so, the CSQ is materialized as a view to be used for future SQs. If an SQ of the CPQ is chosen to be materialized, the CPQ is put into a set of used PQs (SUPQ) rather than added into the SRG when it is completed. The reason for this is that, if one of the SQs of a PQ has been materialized, the SQs of this PQ should not be used in the SRG to estimate the benefits of materializing another SQ. Otherwise, the benefits of a materialized SQ may be double counted. A PQ in the SUPQ can be added to the SRG later on when its materialized SQ is removed from the set of the materialized views because of the space limitation. The result of the previous SQ (RPSQ) is always saved for the possible use of evaluating the CSQ. The CSQ is evaluated either on a materialized view (if beneficial) or on the base table(s) in the database (for the first SQ) or on the RPSQ. The set of the materialized views (SMV) is managed. Each materialized view mv is associated with its corresponding SQ $mv.sq$ as well as its access frequency $mv.freq$ (note that mv itself represents the materialized view).

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

ALGORITHM 3.4.1 : **Dynamic materialized-view based PQ processing procedure (DMVPQ)**
Input: (1) current SQ (csq); (2) current PQ (cpq); (3) set of materialized views (smv); (4) result of previous SQ ($rpsq$); (5) set of used progressive queries ($supq$); (6) superior relationship graph (srg).
Output: (1) the result of csq; (2) a revised srg; (3) a revised cpq; (4) a revised smv; (5) a revised $supq$.
Method:

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

There are two phases in Algorithm 3.4.1. The first phase (lines 1 - 29) evaluates the current SQ and updates the SRG. The second phase (lines 30 - 33) decides whether the result of the current SQ should be materialized for the future use and updates the set of materialized views.

In the first phase, the algorithm first checks whether the given SQ (csq) is the first (initial) SQ (line 1) of a new PQ. If so, the user is actually starting a new PQ and the previous PQ (i.e., the one saved in cpq if any) is completed. In this case, the previous PQ in cpq needs to be added into either the superior relationship graph srg or the set $supq$ of used progressive queries (lines 2 - 9). Lines 3 - 5 convert each SQ in cpq into one that is operated directly on the base table(s) in the database, which can be then compared with the (step-)queries for the materialized views. If one of SQs in cpq

is found to have been materialized, cpq is put into $supq$ (lines 6 - 7). Otherwise, cpq is added into srg by algorithm $AddtoSRG()$ (line 8). After having processed the previous PQ in cpq, cpq is reset to a new PQ with csq as the first (initial) SQ (line 10). If a materialized view whose associated SQ is a superior of csq and whose size is smaller than the size of the table(s)[1] in Domain(csq) is found from the materialized view set smv by algorithm $SearchView()$, we evaluate csq on the found materialized view instead of its (base) operand table(s) (lines 11 - 14). Otherwise, we evaluate csq on its base operand table(s) in the database directly (lines 15 - 16). If csq is not the first SQ, cpq holds the previous SQs of the current/ongoing PQ. In this case, csq is added to cpq (line 20). To check if csq can be evaluated on a materialized view, it needs to be converted into a SQ, $mcsq$, on the base table(s) in the database (line 21). If there exists a materialized view whose associated SQ is a superior of $mcsq$ and whose size is smaller than the size of the result of the SQ directly preceding csq, we evaluate csq on the materialized view (lines 22 - 25). Otherwise, we evaluate csq on the result of its previous SQ ($rpsq$) (lines 26 - 28).

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

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

on base tables $PAPER$ and $ATHOUR$, which has the same result as sq_2.

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

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

frequency) on the materialized view for $mcsq$ and invokes an algorithm $AddtoSMV()$ to add the entry into smv (lines 31 - 33).

The invoked algorithms: $AddtoSRG()$, $SearchView()$, $CheckWeight()$ and $Addto\ SMV()$ are to be discussed in the following sections.

3.5 Superior relationship graph construction

The superior relationship graph is a key component for our dynamic materialized-view based monotonic linear PQ processing technique. It allows us to dynamically accumulate information about executed PQs and effectively use it to select materialized views for efficient execution of future PQs. To efficiently construct such a graph, we apply several heuristic rules derived from the properties of the monotonic linear PQs that were discussed in Section 3.3.

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

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

stage, all the inferior relationships from the new SQs to the old SQs are identified. The edges representing these relationships are added into the SRG. The aforementioned two algorithms apply heuristic rules in the above two stages to improve the constructing performance.

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

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

Proof: we know that sq_i is superior to sq_j. Assume that there exists an SQ sq_k satisfying the following conditions: k is larger than j; sq_j and sq_k are in the same monotonic linear PQ; sq_i is not superior to sq_k. Since sq_j and sq_k are in the same PQ pq_1 and k is larger than j, according to the Property 1 of monotonic linear PQs we mentioned derived from Section 3.3, sq_j is superior to sq_k, which means the result of sq_k is contained in the result of sq_j. In addition, sq_i is superior to sq_j, which means the result of sq_j is contained in the result of sq_i. In this case, we can easily see that the result of sq_k can be derived from the result of sq_i. In other words, sq_i is superior to sq_k, which is contradictory to our assumption. Hence, such sq_k does not exist.

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

Proof: we known that sq_i is superior to sq_j. Assume that there exists an SQ sq_k satisfying the following conditions: k is smaller than i; sq_i and sq_k are in the same monotonic linear PQ; sq_k is not superior to sq_j. Since sq_i and sq_k are in the same PQ pq_1 and k is smaller than j, according to Property 1 of monotonic linear PQs, sq_k is superior to sq_i, which means the result of sq_i can be derived from the result of sq_k. In addition, sq_i is superior to sq_j, which means the result of sq_j can be derived from the result of sq_i. In this case, we can find that the result of sq_j can also be

derived from the result of sq_k. In other words, sq_k is superior to sq_j, which is contradictory to our assumption. Hence, such sq_k does not exist.

The details of the algorithm are specified as follows.

ALGORITHM 3.5.1 : **Generating-Based AddtoSRG1(npg, srg)**
Input: (1) new PQ (npg); (2) superior relationship graph (srg).
Output: revised superior relationship graph (srg).
Method:
1. **if** srg is empty **then** startempty = true;
2. **else** startempty = false **end if**
   /* Adding an isolated PQ npq into srg */
3. add the node and the node-identifier pair for each SQ of npq into sets V and B of srg, respectively;
4. add an edge from each SQ of npq to its immediate subsequent SQ (if any) of npq into edge set E of srg;
5. **if** not startempty **then**
   /* Stage 1: finding external superior relationships */
6. **for** each PQ opq (other than npq) in srg **do**
7. **for** each SQ nsq of npq from the last to the first **do**
8. **for** each SQ osq of opq from the first to the last **do**
9. **if** there exists an edge from nsq to osq **then**
10. break;
11. **else if** there exists a superior relationship from nsq to osq **then**
12. add an edge from nsq to osq into edge set E of srg;
13. **for** each subsequent SQ osq' in opq **do**
14. **if** edge from nsq to osq' does not exist **then**;
15. add an edge from nsq to osq' into edge set E of srg;
16. **end if**
17. **end for**
18. **for** each previous SQ nsq' in npq **do**
19. **if** edge from nsq' to osq does not exist **then**;
20. add an edge from nsq' to osq into edge set E of srg;
21. **for** each subsequent SQ osq' in opq **do**
22. **if** edge from nsq' to osq' does not exist **then**;
23. add an edge from nsq' to osq' into edge set E of srg;
24. **end if**
25. **end for**
26. **end if**
27. **end for**
28. break;
29. **end if**
30. **end for**
31. **end for**
32. **end for**
    /* Stage 2: finding external inferior relationships */
33. **for** each PQ opq (other than npq) in srg **do**
34. exchange the roles of opq and npq in lines 7 - 31 to find the superior relationships from an SQ in opq to an SQ in npq;
    /* i.e., finding the inferior relationships from an SQ in npq to an SQ in opq */
35. **end for**
36. **end if**.

In this algorithm, lines 1 and 2 set a flag to indicate whether the given SRG is empty or not. If it is empty, neither stage 1 nor stage 2 needs to be considered. Lines 4 - 5 add the nodes, node-identifier pairs and internal edges for the SQs from the given PQ into the SRG. The edges between

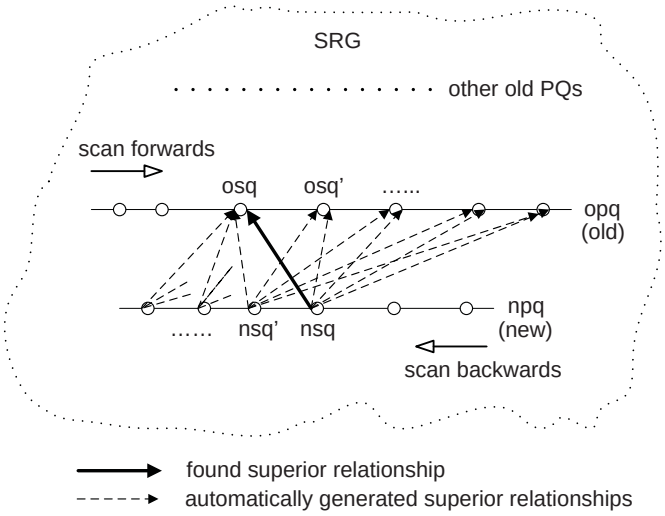

Figure 3.3: Superior relationships automatically generated in Stage 1 of AddtoSRG1()

a node for the PQ and an external node that has already existed in the given SRG are added in two stages. Stage 1 adds the edges for the superior relationships (lines 6 - 32), while stage 2 adds the edges for the inferior relationships (lines 33 - 35).

In stage 1, the algorithm considers one old (existing) PQ in the SRG at a time (line 6). It then scans the SQs of the new PQ backwards and the SQs of the old PQ under consideration forwards and examines each pair of SQs from the two PQs (lines 7 - 8). If there exists a superior relationship between the pair, an edge connecting the corresponding nodes are added into the SRG (lines 11 - 12). The algorithm then automatically generates more superior relationships based on Heuristic Rule 1 (lines 13 - 17 and 21 - 25) and Heuristic Rule 2 (lines 18 - 20). The relevant edges representing these superior relationships are added into the SRG (see Fig. 3.3). Because of the above automatic generation, it is possible that a relevant edge has already been added when a pair of SQs from the two PQs under consideration is examined. Such situations are considered by the algorithm to avoid duplicate additions (lines 9, 14, 19 and 22).

In stage 2, the new PQ and the old PQ under consideration play the opposite roles, comparing to stage 1, because an inferior relationship is opposite to its superior counterpart. With this observation in mind, the algorithm behaves in a similar way.

In contrast to Algorithm 3.5.1, the pruning-based SRG construction algorithm applies the following two heuristic rules to eliminate the pairs of SQs that cannot have superior or inferior relationships, i.e., considering impossible cases rather than possible cases.

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

Proof: Assume that there exists an SQ sq_k satisfies the following conditions: k is smaller than j; sq_k and sq_j are in the same monotonic linear PQ; sq_i is superior to sq_k. Since k is smaller than j, and sq_k and sq_j are in the same monotonic linear PQ, based on the Property 1 of monotonic linear PQs, sq_k is superior to sq_j, which means that the result of sq_j can be derived from the result of sq_k. On the other hand, since we assume that sq_i is superior to sq_k, the result of sq_k can be derived from the result of sq_i. Hence, the result of sq_j can be derived from sq_i, which is contractor to the given condition that there exists no edge from sq_i to sq_j. Therefore, such sq_k in the assumption does not exist.

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

Proof: Assume that there exists an SQ sq_k satisfies the following conditions: k is larger than i; sq_k and sq_i are in the same monotonic linear PQ; sq_k is superior to sq_j. Since k is larger than i, and sq_k and sq_i are in the same monotonic linear PQ, based on the Property 1 of monotonic linear PQs, sq_i is superior to sq_k. In the conditions, sq_i is not superior to sq_j, which means that the result of

sq_k can be derived from the result of sq_i. On the other hand, since we assume that sq_k is superior to sq_j, the result of sq_j can be derived from the result of sq_k. Hence, the result of sq_j can be derived from the result of sq_i, which is contradictory to the given condition that there exists no edge from sq_i to sq_j. Hence, such sq_k in the assumption does not exist.

The details of the algorithm are given below.

ALGORITHM 3.5.2 : **Pruning-Based AddtoSRG2(**npg**,** srg**)**
Input: (1) new PQ (npg); (2) superior relationship graph (srg).
Output: revised superior relationship graph (srg).
Method:
1. **if** srg is empty **then** startempty = true;
2. **else** startempty = false **end if**;
   /* Adding an isolated PQ npq into srg */
3. add the node and the node-identifier pair for each SQ of npq into sets V and B of srg, respectively;
4. add an edge from each SQ of npq to its immediate subsequent SQ (if any) of npq into edge set E of srg;
5. **if** not startempty **then**
   /* Stage 1: finding external superior relationships */
6. **for** each PQ opq in srg **do**
7. let $m = 1$;
8. **for** each SQ nsq of npq from the first to the last **do**
9. **for** each SQ osq of opq from the last to the m-th **do**
10. **if** there exists a superior relationship from nsq to osq **then**
11. add an edge from nsq to osq into edge set E of srg;
12. **else**
13. let m = index number of osq in opq + 1;
14. break;
15. **end if**
16. **end for**
17. **end for**
18. **end for**
   /* Stage 2: finding external inferior relationships */
19. **for** each PQ opq in srg **do**
20. exchange the roles of opq and npq in lines 7 - 17 to find the superior relationships from an SQ in opq to an SQ in npq;
    /* i.e., finding the inferior relationships from an SQ in npq to an SQ in opq */
21. **end for**
22. **end if**.

Lines 1 - 4 are the same as those in Algorithm 3.5.1. There are also two stages in this algorithm. In stage 1, the algorithm considers one old (existing) PQ in the SRG at a time (line 6). It then scans the SQs of the new PQ forwards and the SQs of the old PQ under consideration backwards and examines each pair of SQs from the two PQs (lines 8 - 9). If there exists a superior relationship between the pair, an edge connecting the corresponding nodes are added into the SRG (lines 10 - 11). Otherwise, the algorithm prunes the remaining SQs of opq (Heuristic Rule 3) and resets the

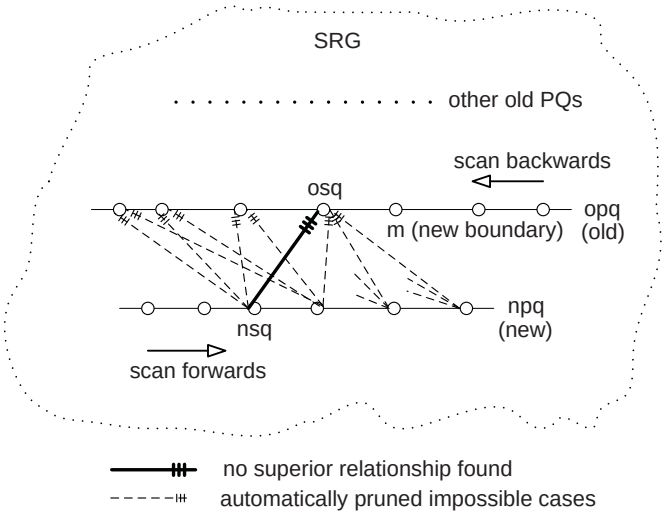

Figure 3.4: Impossible superior relationships automatically pruned in Stage 1 of AddtoSRG2()

scan boundary of the SQs in the old PQ under consideration (Heuristic Rule 4). Fig. 3.4 illustrates the ideas of pruning in this stage. In stage 2, the algorithm behaves similarly except that the new PQ and the old PQ under consideration play the opposite roles.

As an illustration, let us consider the example in Fig. 3.1. Assume that we already have pq_1 (containing sq_1, sq_2 and sq_3) and pq_2 (containing sq_4, sq_5 and sq_6) in the SRG. Our goal is to add pq_3 (containing sq_7 and sq_8) into the graph. Both algorithms first add the nodes,node-identifier pairs and internal edges for pq_3 into the graph. In the superior stage, the algorithms find all the out-going edges (representing superior relationships) from sq_7 or sq_8 to other nodes. In the inferior stage, the algorithms find all the incoming edges (representing inferior relationships) from other nodes to sq_7 or sq_8.

For Algorithm 3.5.1, in the first iteration, we pick up pq_1 from the graph and consider its SQs in the ascending order (from sq_1 to sq_3) while we consider SQs from pq_3 in the descending order. For the first pair [sq_8, sq_1], we find that there is no superior relationship from sq_8 to sq_1. We then

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

For Algorithm 3.5.2, in the first iteration, we pick up pq_1 from the graph and consider its SQs in the descending order (from sq_3 to sq_1) while we consider SQs from pq_3 in the ascending order. For the first pair [sq_7, sq_3], there is a superior relationship from sq_7 to sq_3. So we add an edge from sq_7 to sq_3 and move to consider pair [sq_7, sq_2]. There is no superior relationship in this case. According to Heuristic Rule 3, we remove [sq_7, sq_1] from consideration, and according to Heuristic Rule 4, we remove [sq_8, sq_2] and [sq_8, sq_1] from consideration. We then directly move to consider pair [sq_8, sq_3] and add an edge from sq_8 to sq_3 since such a superior relationship exists. In the second iteration, we handle pq_2 in the same way. We find edges from sq_7 to sq_6 and sq_8 to sq_6. In the inferior stage, we add the incoming edges for sq_7 or sq_8 into the SRG. The details are omitted here.

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

To compare the two algorithms, let us consider two different situations, i.e., the given SRG is a dense graph or a sparse graph. In the dense graph case, Algorithm 3.5.1 could automatically generate many edges by applying Heuristic Rules 1 and 2. In this case, Algorithm 3.5.1 is more efficient. In the sparse graph case, Algorithm 3.5.2 efficiently prunes many useless pairs without checking them individually. In this case, Algorithm 3.5.2 is better. As a result, two algorithms can be used in different situations. This observation is validated through experiments reported in Section 3.9.

3.6 Weight checking

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

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

(1) The query represented by node y is an inferior of x.

(2) Node y has a sufficient weight (i.e., greater than a given threshold).

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

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

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

ALGORITHM 3.6.1 : **Checkweight(**srg**,** csq**)**
Input: (1) superior relationship graph srg; (2) current SQ csq.
Output: true or false.
Method:

```
 1. if srg is empty then
 2.  return false;
 3. else
 4.  for each PQ pq in srg do
 5.   for each SQ sq of pq from the last to the first do
 6.    if sq is an inferior of csq then
 7.     weight = number of out-going edges of sq;
 8.     if weight exceeds a given threshold then
 9.      return true;
10.     end if
11.    else break end if
12.   end for
13.  end for
14.  return false
15. end if.
```

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

3.7 Storage structure and management of materialized view set

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

3.7.1 Storage structure

A straightforward way to implement the SMV is to store materialized views in a linear queue. A new materialized view is always added to the end of the queue. Thus, when an SQ to be evaluated arrives, the system has to scan the view set sequentially to search an appropriate view to use for the SQ. Clearly, if the number of views in the SMV is large, the process to find a usable view can be slow, yielding a low system performance. On the other hand, the views in the SMV may

have superior/inferior relationships among themselves, the linear structure cannot guarantee that the first usable view found is the best one for the given SQ. For example, assume that A and B are materialized views in the SMV, A's associated query is superior to B's associated query, and B's query is superior to the given SQ. If A contains 10000 tuples and B contains 100 tuples, B is clearly a better view to use for the SQ than A. However, in the linear structure, if A is placed before B, the sequential scanning method may return A as a chosen view unless the entire queue is examined.

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

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

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

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

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

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

As a result, four view sets (i.e., the top-view, middle-view, bottom-view and independent-view

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

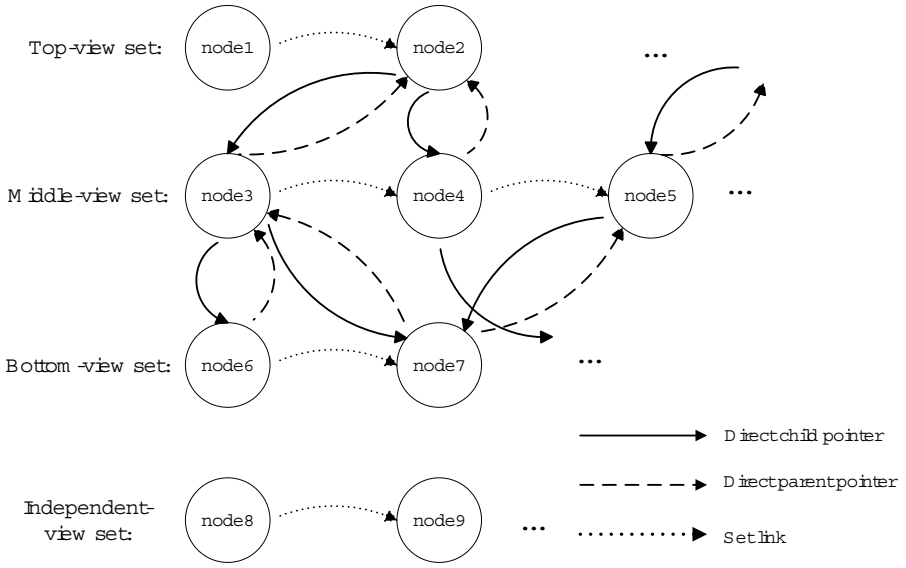

Figure 3.5: An example of the storage structure RLS of the SMV

sets) are maintained within the SMV. Each view set is represented by a linked list.

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

Fig. 3.5 shows an example of the storage structure RLS of the SMV. In the figure, each node

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

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

In the RLS of the SMV, each node (view) has a special data structure (see Fig. 3.6) to keep the relevant information, which includes the node id (ID) to identify the node, the associated SQ expression (SQ) for the represented view, the next view pointer (NV) to point to the next node in the same (top, middle, bottom, or independent) set, the direct parent pointer set (PPS) to store the addresses/pointers of all the direct parent nodes of this node, the direct child pointer set (CPS) to store the addresses/pointers of all the direct child nodes of this node, a frequency counter (FC) to indicate the use frequency of the represented view, a superior/inferior relationship testing record (STR) to keep the previously discovered relationships, and the address/pointer of the view (DataP) to point to the materialized view data.

When a new view/node N (corresponding to the current SQ) is to be added to the SMV, it is compared with existing views/nodes in the SMV to discover its superior/inferior relationships with them. To improve the processing performance, as done before, we apply heuristic rules to automatically derive new relationships with more nodes in the SMV once a relationship with one node is discovered. We also try to avoid a duplicate comparison if the relationship of N with an existing node has already been discovered or derived previously.

STR is a temporary storage for an existing node N' to record the previously discovered or heuristic-derived superior/inferior relationships with a new node being inserted. STR consists of a node id (ID) and an indicator (REL). The node id identifies the node M (i.e., N or a previously inserted node) with which the relationship(s) has been discovered/derived previously. The indicator

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

Table 3.1: Discovered Superior/Inferior Relationship Indicator

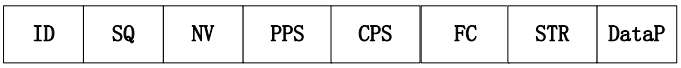

Figure 3.6: The data structure of each node in the SMV

is a two-bit binary value, where the lower bit indicates the existence of an inferior relationship from N' to M and the higher bit indicates the existence of a superior relationship from N' to M. The possible values of REL and their meanings are summarized in Table 3.1.

3.7.2 RLS storage structure construction

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

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

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

Heuristic Rule 5 is similar to Heuristic Rules 1 and 3, while Heuristic Rule 6 is similar to

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

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

During the above process, we use a status flag ($status\_flag$) to indicate the status of determining the view set to which the new node N belongs to. The flag is initially set to -1. The values of this status flag and their meanings are summarized in Table 3.2. Values 0 - 3 indicate that the view set to which N belongs to has been determined; while values 4 - 5 indicate that only partial information, which is insufficient to determine the view set membership of N, is obtained. In fact, when the flag value is 4, there are three cases. First, N has a direct child node M in the independent-view set. In this case, N belongs to the top-view set. Note that N cannot have a direct parent node in any view set in this case. Otherwise, M could not belong to the independent-view set in the first place due to the relationship transitivity. Second, N has a direct parent node in a (any) view set. In this case, N belongs to the bottom-view set. Note that N cannot have a direct child M in the independent-view set in this case. Otherwise, it violates the fact that M belongs to the independent-view set due to the transitivity. Third, N has no relationship with any view in the current SMV. In this case, N belongs to the independent-view set. When the flag value equals to

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

Table 3.2: Status Flag Values and Their Indicated Status

5, there are two cases. First, N has a direct parent node in a (any) view set. In this case, N is a middle-view node since it also has a direct child. Second, N has no direct parent node in any view set. In this case, N belongs to the top-view set. When the flag value equals to 6, there is no need to insert N into the SMV since it has already been represented by an existing node in the SMV.

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

ALGORITHM 3.7.1 : **InsertViewIntoSMV** (N, smv)
Input: (1) new materialized view node N; (2) set of materialized views (smv) with the RLS structure.
Output: updated smv.
Method:
1. initialize $status\_flag$ to -1 and the fields of N to NULL or $\emptyset$;
   /*Stage 1: find direct child nodes of N in the bottom-view, middle-view and top-view sets */
2. **if** the bottom-view set is not empty **then**
3. **for** each node S in the bottom-view set **do**
   /* find direct child nodes of N that lie on each upward path of S and try to
   determine the view set membership of N from bottom up */
4. $status\_flag$=AddFromBottom($N, S, smv, status\_flag$);
5. **if** $status\_flag$ == 6 **then** /* N already has an equivalent in smv */
6. return; /* no need to insert N */
7. **end if**
8. **end for**
9. **if** $status\_flag$ == -1 **then**
   /* no direct child node was found for N in the bottom-view, middle-view or top-view set */
10. set $status\_flag$=4; /* record partial information */
11. **end if**
12. **end if**
   /*Stage 2: find direct parent nodes of N in the bottom-view, middle-view and top-view sets */
13. **if** $status\_flag \neq 3$ **then**
   /* N has not been determined to be in the top-view set */
14. **if** the top-view set is not empty **then**
15. **for** each node T in the top-view set **do**

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

In Algorithm 3.7.1, before the first stage, the relevant fields for new node N are initialized to be ready for the node data structure in smv, and flag $status\_flag$ is initialized to -1 (line 1).

In the first stage, if the bottom-view set B is not empty, this algorithm invokes a recursive

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

At the beginning of the second stage, the algorithm first checks if N has been determined to be a top-view (line 13). If it is true (i.e., $status\_flag$ = 3), the second stage is skipped since N has no direct parent node in such a case. Otherwise, if the top-view set D is not empty (line 14), the algorithm invokes a recursive function AddFromTop() to discover all the direct parent nodes of N in smv by following the descendant (downward) paths of each node in D. The goal is to find the smallest (lowest) direct parent of N along each downward path. Depending on how low the algorithm can go down along the paths, the information about the view set membership of N may be obtained. For example, if the lowest node of a path is found to be a direct parent of N, then N is determined to be in the bottom-view set. In conjunction with some partial information obtained from the first stage, there are more cases in which the view set membership of N can be determined. The details of AddFromTop() will be discussed later on. It is possible that N is found to be equivalent to a node in smv during the procedure (line 17). In such a case, there is no need to add N into smv and the algorithm returns (line 18). If N is found not to be inferior to any node in the top-view set and known to have at least one direct child (from the first stage), N

is determined to be in the top-view set (lines 21 - 25). At the end of the second stage, the possible values of $status\_flag$ are 1, 2, 3, 4, 6 (program exits) and -1 (the bottom-view set — hence, the middle-view and top-view sets as well are empty).

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

the end of stage 3, N is inserted into a proper view set in smv according to the value of determined $status\_flag$.

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

ALGORITHM 3.7.2 : **AddFromBottom(**N**,** S**,** smv**,** $status\_flag$**)**
Input: (1) new materialized view node (N); (2) a compared node (S); (3) set of materialized views (smv) with the RLS structure; (4) a status flag for the view set membership determination ($status\_flag$).
Output: (1) updated $status\_flag$; (2) updated smv.
Method:

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

Algorithm 3.7.2 is used to find the direct child nodes of N in the bottom-view set, the middle-

view set and the top-view set that lie on the upward paths from S and determine the membership of a view set for N from bottom up if possible. It traverses up from S in smv by recursively following the parent links of S (lines 25 - 26). The algorithm first identifies the relationship between N and S, which could be found previously (lines 2 - 5) or is discovered in the current invocation (lines 6 - 18). If N and S is found to be equivalent, there is no need to insert N into smv (lines 8 - 11 and 27 - 29). Note that, when such an equivalence is found (line 8), the algorithm has to clean up the possible direct child/parent links added for N from its direct child nodes discovered so far before it returns. If N is found to be superior or inferior to S for the first time, such a relationship needs to be propagated to the descendants or ancestors of S based on Heuristic Rule 5 or 6, respectively (lines 12 - 18). If N is superior to S, there are two cases in which S becomes a direct child of N. The first case is when S was in the top-view set before N is added (lines 20 - 23), i.e., S had at least one child but no parent. After N is added, N becomes the only (direct) parent of S. In this case, it is determined that N belongs to the top-view set (line 21), and S has to be moved to the middle-view set (line 22). The second case is when N is found to be not superior to any direct parent of S (lines 31 - 36). Since it is unknown if N has its own direct parent in this case, the view set membership for N cannot be determined (line 33). When the algorithm returns, $status\_flag$ has one of the following values: 3, 5, 6 and -1 (no direct child so far).

ALGORITHM 3.7.3 : **AddFromTop(**N**,** T**,** smv**,** $status\_flag$**)**
Input: (1) new materialized view node (N); (2) a compared node (T); (3) set of materialized views (smv) with the RLS structure; (4) a status flag for the view set membership determination ($status\_flag$).
Output: (1) updated $status\_flag$; (2) updated smv.
Method:
1. $inferior\_relationship$ = false;
2. **if** relationship between N and T was found before **then** /* i.e., $T.STR.ID == N.ID$ */
3. **if** N is inferior to T **then** /* i.e., $T.STR.REL$==01 */
4. $inferior\_relationship$ = true;
5. **end if**
6. **else** /* relationship between N and T has never been explored before */
7. find the relationship between N and T and record the information in $T.STR$;
   /* i.e., set $T.STR.REL$ to 00, 01, or 11 accordingly and $T.STR.ID$ = $N.ID$ */
8. **if** N and T are equivalent **then** /* i.e., $T.STR.REL = 11$ */
9. set $status\_flag$ = 6;
10. clean N from smv if N was linked in smv via a direct child/parent relationship previously;
11. return $status\_flag$; /* no need to insert N */

12. **else if** N is inferior to T **then** /* $T.STR.REL$ == 01 */
13. propagate the inferior relationship to each ancestor of T;
/* i.e., set $X.STR.ID = N.ID$ and $X.STR.REL$ = 01 for each (unset) ancestor X of T;
14. **end if**
15. **end if**
16. **if** $inferior\_relationship$ = true **then**
17. **if** T is a bottom-view **then**
18. **if** $status\_flag$!= 1 **then**
19. $status\_flag$ = 1; /* N is determined to be in the bottom-view set */
20. **end if**
21. move T from the bottom-view set to the middle-view set;
22. link T and N together with a direct parent/child relationship;
/* i.e., update $T.CPS$ and $N.PPS$ to indicate T is a direct parent of N */
23. **else** /* T is a middle-view or top-view */
24. **for** each direct child node K in $T.CPS$ **do**
/* find direct parent nodes of N on each downward path from T recursively and
determine the view set membership for N from top down */
25. $status\_flag$=AddFromTop($N, K, smv, status\_flag$);
26. **if** $status\_flag$ == 6 **then** /* N already has an equivalent in smv */
27. return $status\_flag$; /* no need to insert N */
28. **end if**
29. **end for**
30. **if** N is not inferior to any direct child node of T **then**
31. **if** $status\_flag$ ==4 **then** /* N is known to have no direct child */
32. set $status\_flag$ = 1; /* N is determined to be in the bottom-view set */
33. **else** /* $status\_flag$ = 5; i.e., N has at least one direct child */
34. set $status\_flag$ = 2; /* N is determined to be in the middle-view set */
35. **end if**
36. link T and N together with a direct parent/child relationship;
/* i.e., update $T.CPS$ and $N.PPS$ to indicate T is a direct parent of N */
37. **end if**
38. **end if**
39. **end if**
40. return $status\_flag$.

Algorithm 3.7.3 is used to find the direct parent nodes of N in the bottom-view set, the middle-view set and the top-view set that lie on the downward paths from T and determine the membership of a view set for N from top down if possible. It is similar to Algorithm 3.7.2, except that it traverses down (instead of up) from T in smv by recursively following the direct child links of T (lines 24 - 25). The algorithm first identifies the relationship between N and T, which could be found previously (lines 2 - 5) or is discovered in the current invocation (lines 6 - 15). If N and T is found to be equivalent, there is no need to add N into smv (lines 8 - 11 and 26 - 28). If N is found to be inferior to T for the first time, such a relationship needs to be propagated to the ancestors of T based on Heuristic Rule 6 (lines 12 - 14). Note that it is impossible for a new superior relationship from N to T (i.e., $T.STR.REL$ = 10) to be discovered at this time since all

possible superior relationships from N to an existing node in smv have been discovered in the first stage. If N is inferior to T, there are two cases in which T becomes a direct parent of N. The first case is when T was in the bottom-view set before N is added (lines 17 - 22), i.e., T had at least one parent but no child. After N is added, N becomes the only (direct) child of T. In this case, it is determined that N belongs to the bottom-view set (lines 18 - 20), and T has to be moved to the middle-view set (line 21). The second case is when N is found to be not inferior to any direct child of T (lines 30 - 37). In this case, N is determined to be in the bottom-view set (lines 31 - 32) or the middle-view set (lines 33 - 34), depending on whether a direct child of N has been found in the first stage or not. Note that $status\_flag$ cannot be -1 at line 33 since, when this algorithm is invoked at line 16 in Algorithm 3.7.1, the top-view set must not be empty, which implies that the bottom-view set cannot be empty. As mentioned earlier, $status\_flag$ = -1 at the end of the first stage only if the bottom-view set is empty. When the algorithm returns, $status\_flag$ may have one of the following values: 1, 2, 4, 5 and 6.

3.7.3 Examples

Now let us use some insertion examples to illustrate how the SMV construction algorithm works in different scenarios. Assume that we have a partially constructed SMV as shown in Fig. 3.7.

In Fig. 3.7, the nodes from $n1$ to $n5$ are the top-views, the nodes from $n6$ to $n11$ are the middle-views, the nodes from $n12$ to $n16$ are the bottom-views, and the nodes from $n17$ to $n20$ are the independent views. We use a pair (N, M) to denote that node N is superior to node M. All the superior relationships in Fig. 3.7 are shown as follows: ($n1$, $n9$), ($n9$, $n12$), ($n2$, $n13$), ($n3$, $n6$), ($n6$, $n10$), ($n10$, $n14$), ($n3$, $n7$), ($n7$, $n11$), ($n11$, $n14$), ($n4$, $n7$), ($n5$, $n8$), ($n8$, $n11$), ($n11$, $n16$).

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

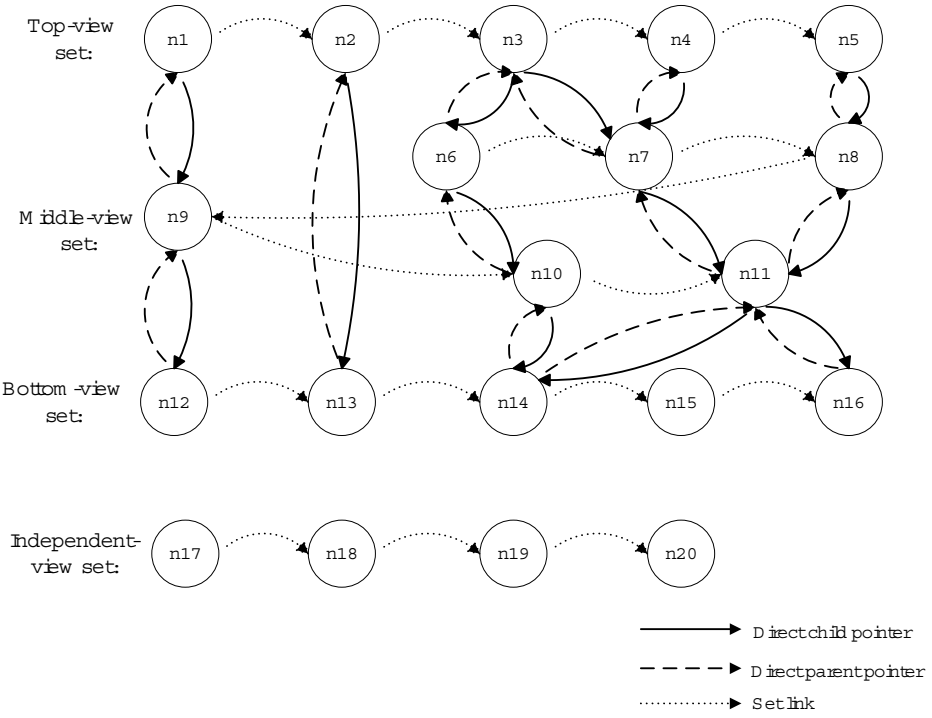

Figure 3.7: A partially constructed SMV

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

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

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

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

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

To add $sq1$ into the SMV, the bottom-view set is checked first. The algorithm wants to find those nodes to which $sq1$ is superior in the SMV. Nodes $n12$ and $n13$ are passed because $sq1$ is superior to neither of them. When the algorithm finds that $sq1$ is superior to $n14$, it traverses up through the direct parent node links of $n14$ to visit $n10$ and $n11$. Again, the algorithm finds that $sq1$ is superior to $n10$. Therefore, the algorithm continues to traverse up through the direct parent node link of $n10$

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

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

To add $sq3$ into the SMV, the algorithm checks the bottom-view set first as before. It is found that $sq3$ is not superior to any bottom-view node. The top-view set is then checked. It is found that

$n1$ is superior to $sq3$. Hence, the algorithm traverses down through the direct child node link of $n1$ to visit $n9$. It is found that $n9$ is also superior to $sq3$. Thus, the algorithm continues to traverse down through the direct child node link of $n9$ to visit $n12$. Node $n12$ is still superior to $sq3$. Since $n12$ is a bottom-view node, the algorithm determines that $sq3$ is a bottom-view node. Therefore, it moves $n12$ to the middle-view set and links $sq3$ directly below $n12$ ($status\_flag = 1$; lines 17 - 22 in AddFromTop()). The algorithm then goes back to check the rest of top-view nodes $n2$, ..., $n5$ and finds no superiors. After that, the independent-view set is also checked to see if there exists any independent-view node which is superior to $sq3$ and none is found. Finally, $sq3$ is put into the bottom-view set and the insertion process ends.

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

To add $sq5$ into the SMV, the bottom-view set is checked first as before. The algorithm finds that $sq5$ is superior to $n14$. It then traverses up through the direct parent node links of $n14$ to visit $n10$ and $n11$. Node $n10$ is passed, but $n11$ is found to be equivalent to $sq5$ ($status\_flag$=6; lines 8 - 11 in AddFromBottom()). In this case, no need to add $sq5$ into the SMV. Therefore, the algorithm stops the insertion process and makes no change for the SMV. Fig. 3.8 shows the SMV after inserting the nodes $sq1$, $sq2$, $sq3$ and $sq4$ ($sq5$ is not added).

Now let us consider another insertion example to illustrate how the information in the superior/inferior relationship testing record (STR) helps the algorithm improve its efficiency. As mentioned before, STR is a temporary storage for an existing node N in the SMV to record the

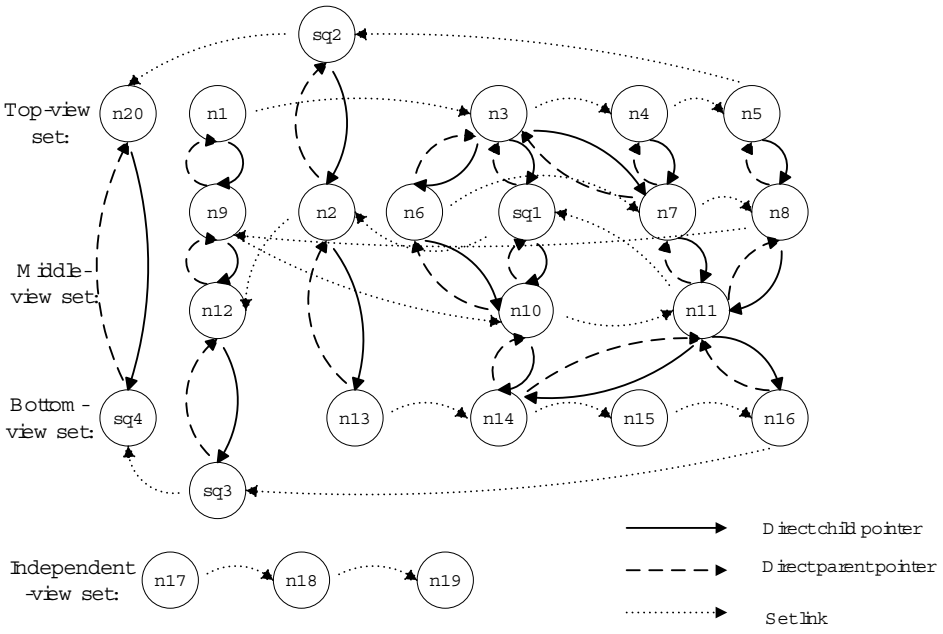

Figure 3.8: The modified SMV after inserting the nodes from $sq1$ to $sq5$

previously-discovered or heuristic-derived superior/inferior relationships with a new node being inserted. The algorithm can make use of the STRs of the existing nodes to avoid some duplicate or unnecessary comparison work.

In this example, we still consider the SMV shown in Fig. 3.7. Assume that the new node $sq6$ has the following superior relationships with the existing nodes in the SMV: ($sq6$, $n14$), ($n10$, $sq6$), ($n6$, $sq6$), ($n3$, $sq6$). To add $sq6$ into the SMV, in the first stage, the bottom-view set is checked. Nodes $n12$ and $n13$ are passed since they have no relationship with $sq6$. After it is found that $sq6$ is superior to $n14$, the algorithm traverses up to check $n10$ and $n11$. After two pair-wise comparisons, it is found that $sq6$ is superior to neither $n10$ nor $n11$. However, another relationship (i.e., $n10$ is superior to $sq6$) which is supposed to be found in the second stage is discovered (line 15 in AddFromBottom()). This information is saved in the STR of $n10$. Based on Heuristic 6, we also can derive (without pair-wise comparisons) the superior relationship from each ancestor

node of $n10$ to $sq6$, i.e.,the STRs of $n6$ and $n3$ are also updated (line 16 in AddFromBottom()). After that, the algorithm links $sq6$ directly above (superior to) $n14$ ($status\_flag$ = 5; lines 31 - 35 in AddFromBottom()) and goes back to check the other bottom-view nodes. In the second stage, the top-view set is checked. Nodes $n1$ and $n2$ are passed since they have no relationship with $sq6$. From the STR of $n3$, it is found that $n3$ is superior to $sq6$ (the actual pair-wise comparison is avoided) and the algorithm directly traverses down to visit $n6$ and $n7$. Again, from the STR of $n6$, it is found that $n6$ is superior to $sq6$. Hence, the algorithm keeps traversing down through $n10$ until $n14$ is reached (lines 2 - 5 and 24 - 25 in AddFromTop()), then $sq6$ is linked directly below (inferior to) $n10$ ($status\_flag$ = 2; lines 33 - 36 in AddFromTop()). The third stage is skipped since $sq6$ has already been determined to be in the middle-view set and no relationship with an independent view is possible (otherwise, the independent view could not be independent since it would have relationships with existing top-view(s) and bottom-view(s)). Finally, the algorithm inserts $sq6$ into the SMV. From this example, we can see that, using the STR, many duplicate (previously compared) and/or unnecessary (derived) pair-wise comparisons can be avoided.

3.7.4 Materialized View Set Maintenance

Algorithm 3.7.1 can be used to insert a view into the SMV. However, the number of views that can be saved in the SMV is not unlimited. There is a space constraint for the SMV. We assume that (1) there is a space limit (SL) for the SMV and (2) the SL is large enough to hold the largest materialized view. When the SMV overflows (i.e., its size exceeds the SL), we need to delete some materialized views from it to create enough free space for accommodating a new materialized view.

The algorithm RemoveViewFromSMV(M, smv) to delete a given materialized view (node) M from the SMV smv is relatively straightforward. The main idea is to remove the relevant child/parent links for M from its direct child/parent nodes, adjust the view set memberships (after deletion) for the direct child/ parent nodes when necessary, transfer the relationships of M with its

direct child/parent nodes to other relevant nodes in smv when necessary, and remove M from the corresponding top/bottom/independent-view set in smv. The details of this algorithm are omitted due to the space limitation.

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

ALGORITHM 3.7.4 : **InsertViewWithMaintenance**(N, smv, srg, $supq$)
Input: (1) new materialized view node (N) to be inserted; (2) set of materialized views (smv) with the RLS structure; (3) superior relationship graph (srg); (4) set of used PQs ($supq$).
Output: (1) updated smv with N added; (2) revised srg; (3) revised $supq$.
Method:
1. **while** smv does not have enough space to accommodate N **do**
2. find a view M to be removed from smv according to the access frequencies;
3. RemoveViewFromSMV(M, smv);
4. **if** the corresponding PQ x containing $M.sq$ has no SQ represented in smv **then**
5. remove x from $supq$;
6. AddtoSRG(x, srg);
7. **end if**
8. **end while**;
9. InsertViewIntoSMV(N, smv).

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

3.8 View search

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

3.9 Experiments

To evaluate the performance of the dynamic materialize view based monotonic linear PQ processing approach, we conducted extensive simulation experiments. Experiment programs were implemented in Matlab 2007 on a PC with Intel® dual core (1.5 GHz) CPU and 2 GB memory running on the Windows® Vista operating system. The experimental data set consisted of 10 external tables of randomly generated data with sizes ranging from 0 to 1000 disk blocks. 100 random progressive queries were used for each experiment. Each PQ was composed of two or more SQs, where the number of steps was randomly chosen between 2 and 5. The result size of each SQ also ranged from 0 to 1000 disk blocks. The experiments were grouped into three sets. Their typical experimental results are reported in the following sections, respectively.

3.9.1 Performance of Dynamic Materialized-View Based Approach

The first set of experiments was conducted to evaluate the efficiency of our dynamic materialized-view based monotonic linear PQ processing approach (DMVPQ). The superior relationship graph (SRG) and the set of materialized views (SMV) using the RLS structure were initially set to empty.

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

In the first experiment, the upper bounds for IPR, WPR and SPR were set to 0.1, 0.5 and 0.1, respectively. Fig. 3.9 shows the performance comparisons between the CSSPQ and the DMVPQ techniques.The x-axis represents the total number of SQs executed in the system, and the y-axis represents the I/O cost (i.e., the number of disk block accesses). From the figure, we can see that the two performance curves are very close to each other when the number of SQs processed is small. The performance of DMVPQ is increasingly better than that of CSSPQ when the number of SQs increases. The reason for this is as follows. At the beginning, both SRG and SMV are

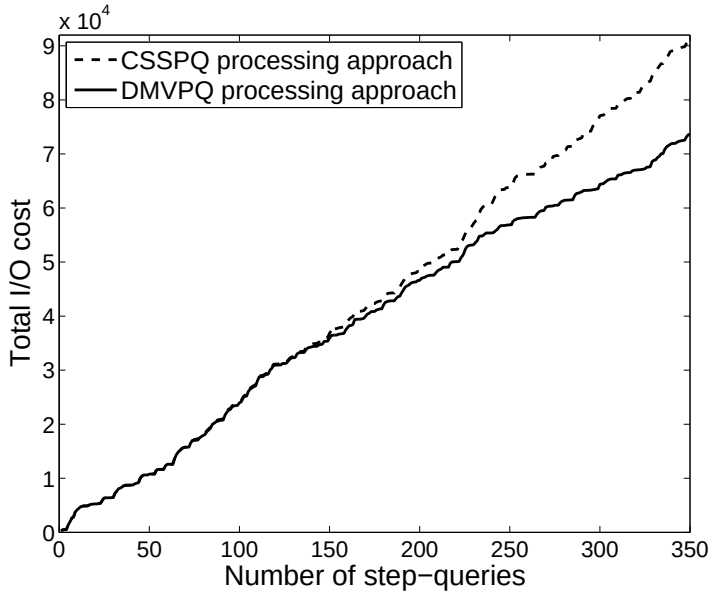

Figure 3.9: Performance comparison between DMVPQ and CSSPQ

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

In the second experiment, we increased the upper bound for parameter IPR to 0.3 and kept the other parameters unchanged. The experimental results are shown in Fig. 3.10. From the figure, we can see that the performance of DMVPQ is significantly improved. The reason for this is that IPR plays an important role in deciding whether to materialize the result of an SQ. A larger upper bound for IPR implies that an SQ has a higher chance to be materialized. Hence, the SMV grows faster, and the subsequent queries have more views to utilize to improve their performance.

Another crucial factor to affect the query performance is parameter SPR. In the third experiment, we changed the upper bound for SPR to 0.3 and kept the other parameters unchanged. Experimental results are shown in Fig. 3.11. A significant performance increase for DMVPQ is

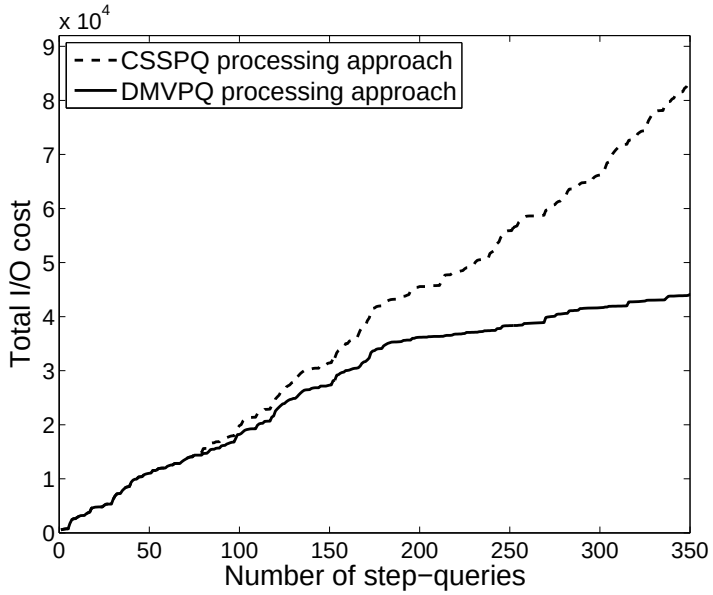

Figure 3.10: Performance comparison between DMVPQ and CSSPQ with IPR being changed to 0.3

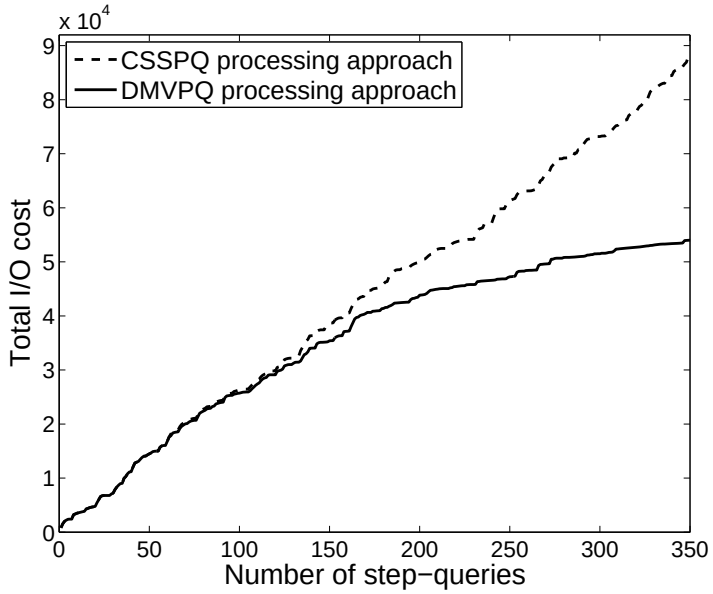

Figure 3.11: Performance comparison between DMVPQ and CSSPQ with SPR being changed to 0.3

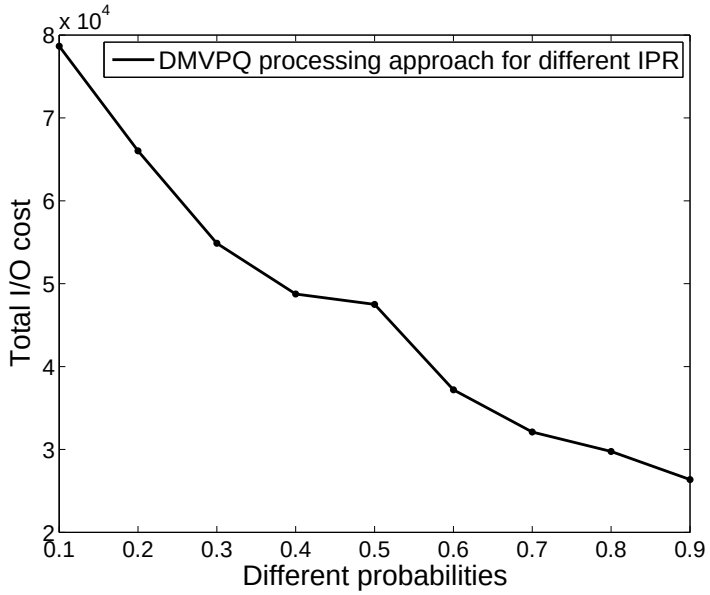

Figure 3.12: Performance change with different IPRs for DMVPQ

also observed. The reason for this improvement is that SPR is the factor to determine whether a materialized view would be usable for an SQ under consideration. A larger upper bound for SPR implies a materialized view has a better chance to be usable for a given SQ. In other words, an SQ has more available views to utilize to improve its performance.

In the next experiment, we considered various upper bounds for IPR ranging from 0.1 to 0.9 and kept other parameters unchanged. The performance curve is shown in Fig. 3.12. From the figure, we can clearly see that the performance is improved as IPR increases.

We also conducted another experiment for various upper bounds for SPR ranging from 0.1 to 0.9 and kept other parameters unchanged. The experimental results are shown in Fig. 3.13. A similar performance pattern is also observed.

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

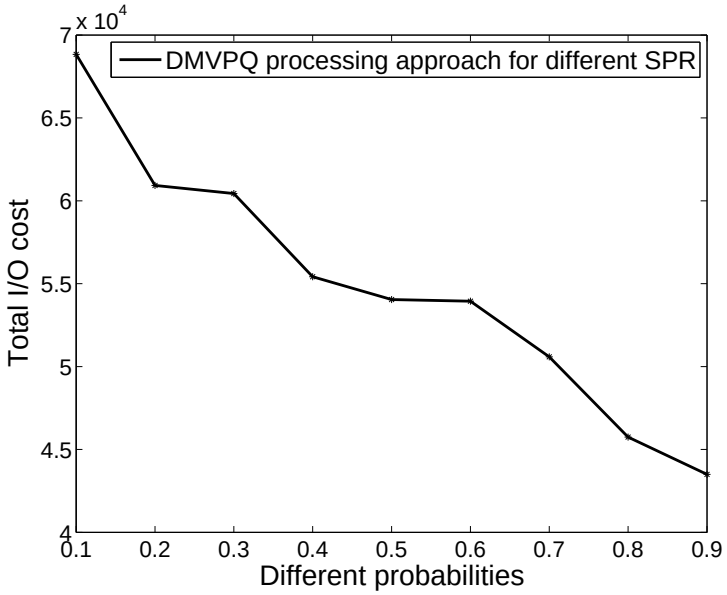

Figure 3.13: Performance change with different SPRs for DMVPQ

3.9.2 Performance of SRG Construction Methods

The second set of experiments was conducted to compare the performance behaviors of the generating-based method and the pruning-based method for constructing a superior relationship graph (SRG). The SRG was initially set to empty. The progressive queries were processed one by one. When a new SQ was added into the SRG, we needed to find all the superior or inferior relationships between the new SQ and the SQs in the SRG. As mentioned before, a straightforward way to construct an SGR is to perform the pair-wise comparisons between the new SQ to be added and each existing SQ in the SRG. But the cost of this way is usually very high, which led us to develop the generating-based method and the pruning-based method to avoid some unnecessary comparisons. In the experiments, we wanted to compare the performance of these two methods so as to identify the scenarios where one method could be better than the other. The performance was measured in terms of the cost (pair-wise comparisons) saved over the straightforward pair-wise comparison

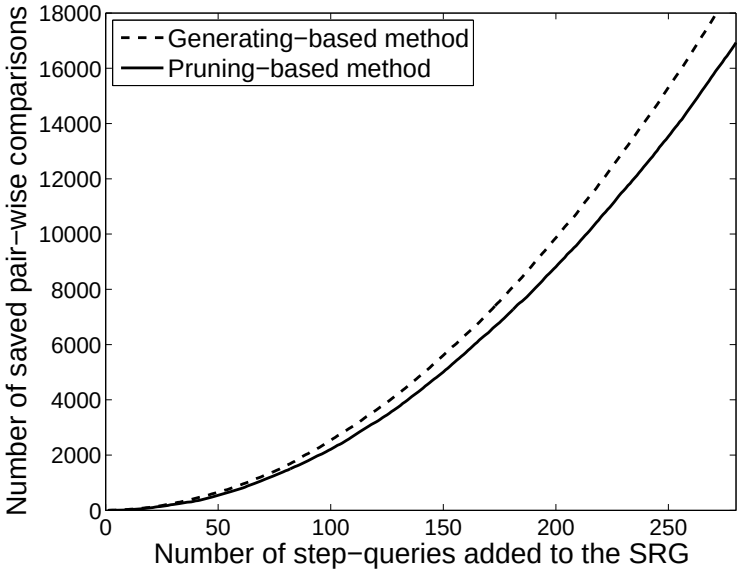

Figure 3.14: Comparison of saved costs between the generating-based method and the pruning-based method with $IPR = 0.7$

method. We also maintained the IPR for each node in the SRG, which represents the probability with which this node is inferior to a new SQ to be added.

In the first experiment, we set the upper bound of IPR to a relatively high value 0.7, which led to a high chance for the existing nodes in the SRG to have a (inferior) relationship with a new SQ to be added. Hence the resulting SRG was a dense (in terms of edges) graph. Fig. 3.14 shows the comparison of the saved costs between the generating-based method and the pruning-based method. From the figure, we can observe that the former method outperforms the latter method to construct the SRG in such a case. This is because the generating-based method has a better chance to automatically derive/generate more relationships (edges) in a dense graph to save many (pair-wise) comparisons. The larger the SRG, the more savings the generating-based method could achieve.

In the second experiment, we set the upper bound of IPR to a relatively low value 0.3. The

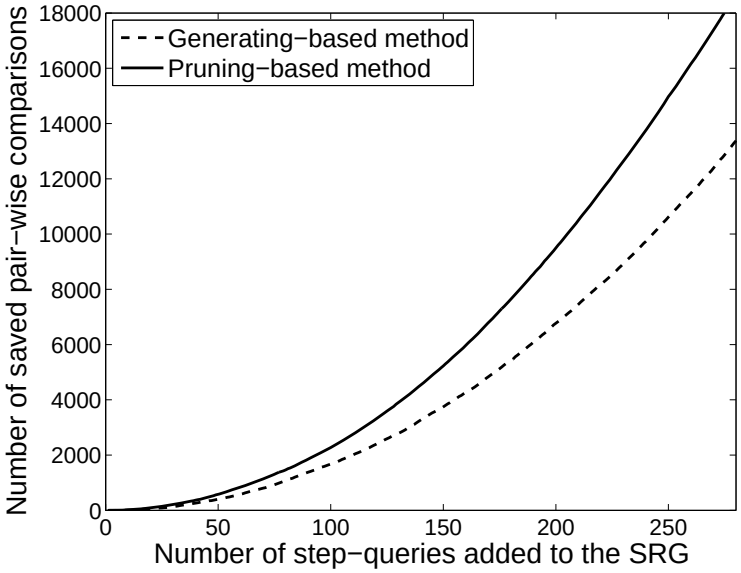

Figure 3.15: Comparison of saved costs between the generating-based method and the pruning-based method with $IPR = 0.3$

resulting SRG was a sparse graph. Fig. 3.15 shows the comparison of the saved costs between the generating-based method and the pruning-based method. We can see that the pruning based method performed better than the generating based method in this case. This is because the pruning-based method has a better chance to automatically eliminate/prune impossible relationships (edges) in a sparse graph to save many comparisons. The larger the SRG, the more savings the pruning-based method could obtain.

The previous two experiments demonstrate that both the generating-based method and the pruning-based method can save an increasing amount of cost as the SRG grows. The generating-based method is better for a dense graph, while the pruning-based method is better for a sparse graph, as we predicted in Section 3.5.

3.9.3 Performance of View Search Using New SMV Storage Structure

The third set of experiments was conducted to examine the view search performance for the set of materialized views (SMV) using our new storage structure RLS. The SMV was initially set to empty. Four different materialized view sets (top-view set, middle-view set, bottom-view set and independent-view set) were maintained. Progressive queries were processed one by one. For each SQ of the current PQ, the SMV was searched to find a usable view that could be used to answer the SQ. After an SQ was executed, its result had a chance to be kept as a (materialized) view and stored in the SMV. The new storage structure RLS for the MVS was built by using Algorithm 3.7.1. We maintained three parameters UPR, $SCPR$ and WPR for each SQ. UPR denotes the probability for a view in the top-view set to be a usable view for the given SQ; $SCPR$ denotes the probability for a direct child node to be a usable view replacing its direct parent node for the given SQ; WPR denotes the probability for an SQ result to be kept as a materialized view. Two additional parameters $SUPR$ and $INPR$ were also maintained in our experiments. $SUPR$ denotes the probability for a view to be superior to another view in the SMV, and $INPR$ denotes the probability for a view to be inferior to another view in the SMV. UPR, $SCPR$, WPR, $SUPR$ and $INPR$ were randomly chosen between 0 and their respective upper bound. In the experiments, the upper bounds for UPR, $SCPR$, WPR, $SUPR$ and $INPR$ were set to 0.3, 0.9, 0.3, 0.3 and 0.3, respectively.

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

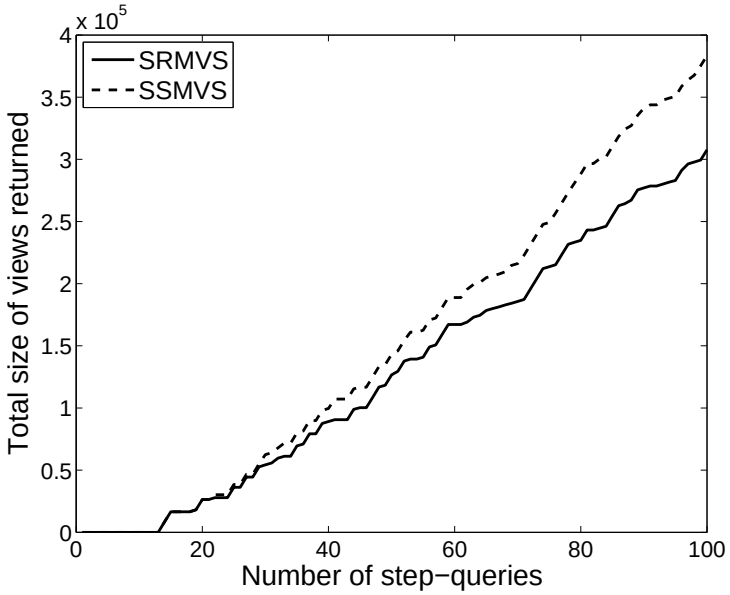

Figure 3.16: Comparison of view quality between SSMVS and SRMVS based on FTS

(SRMVS). Various scenarios were considered.

In the first experiment, we used the fastest time strategy for both the SSMVS and the SRMVS. For the SSMVS, the search returned the first usable view (if any) in the linear queue. For the SRMVS, the search first found the first usable view (if any) in the top-view set and then recursively followed the direct child (inferior) link of the found view to see if a better (smaller) usable view could be obtained. Hence the SRMVS returned an improved usable view (if possible) over the first usable view found in the top-view set. Note that neither the SRMVS nor the SSMVS based on the FTS can guarantee that the best usable view in the SMV is found. Fig. 3.16 shows the comparison of view quality (in terms of view size) between the SSMVS and the SRMVS based on the FTS. The x-axis represents the total number of SQs processed in the system. For each SQ, a usable view may be returned from the SMV to answer the query. The y-axis represents the total size of the returned views. The smaller the total size is, the higher the view quality is achieved. From the figure, we can see that two curves are very close at the beginning. The view quality obtained from the SRMVS

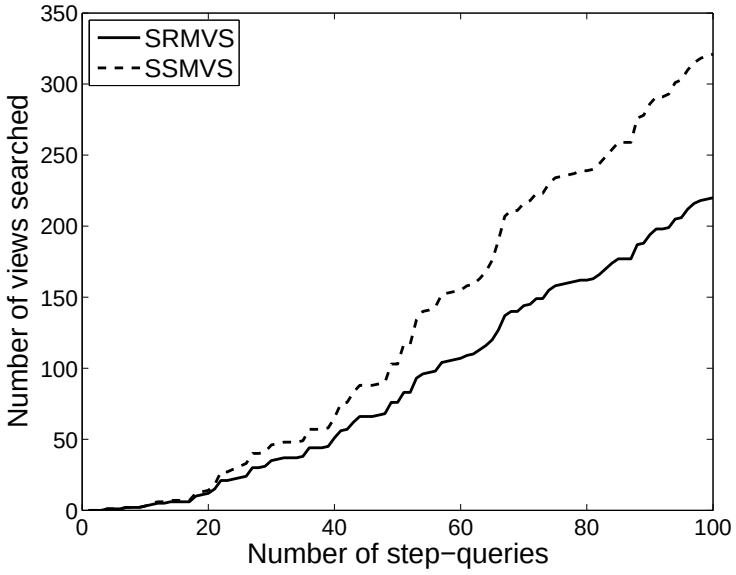

Figure 3.17: Comparison of view searching costs between SSMVS and SRMVS based on FTS

is increasingly better than that from the SSMVS as the number of SQs increases. The reason for this is as follows. At the beginning, the SMVs were empty for both the SSMVS and the SRMVS. Thus, for the first several SQs, no view could be used to answer them and the total view size was 0 for both the SSMVS and the SRMVS. As the number of SQs increased, more and more SQ results were saved as materialized views in the SMVs, which could be used to answer the following SQs and the total view size started to increase. As the SMV grew, the SRMVS had a better chance to return an improved usable view (via the maintained supper/inferior relationships) over the first usable view found in the top-view set. Therefore, the view quality curve of the SRMVS becomes better and better, compared to that of the SSMVS.

Fig. 3.17 shows the comparison of view searching costs between the SSMVS and the SRMVS based on the FTS. The x-axis still represents the number of SQs processed. The y-axis represents the total number of views searched. From the figure, we can see that the searching cost of the SRMVS is always smaller than the one of the SSMVS. The reason for this is as follows. The

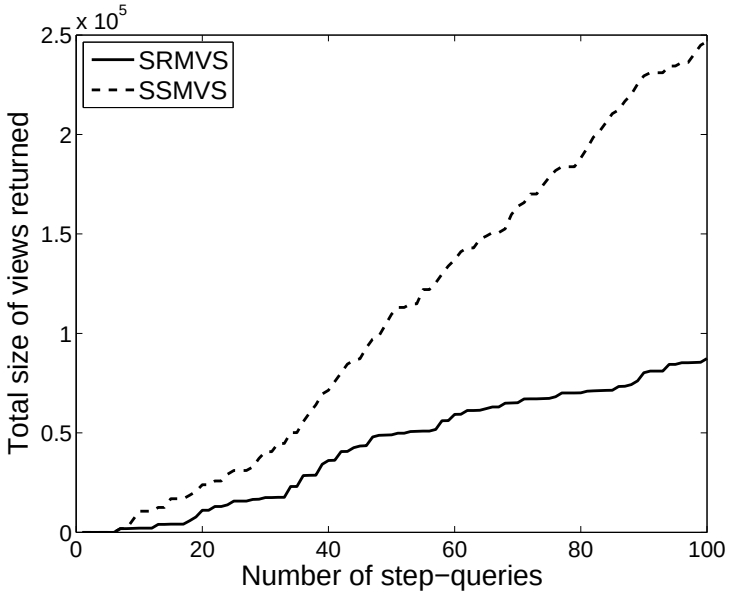

Figure 3.18: Comparison of view quality between SSMVS based on FTS and SRMVS based on BRS

SRMVS based on the FTS typically can save some searching cost by pruning the middle-views and bottom-views that are descendants of a non-usable top-view, while the SSMVS based on the FTS has to search all the views in the SMV in the worst case.

In the second experiment, we applied the FTS for the SSMVS and the BRS for the SRMVS. We still compared both the view quality and the searching costs between the SSMVS and the SRMVS. Fig. 3.18 shows the comparison of view quality between the SSMVS based on the FTS and the SRMVS based on the BRS. From the figure, we can see that the view quality from the SRMVS is dramatically improved by using the BRS instead of the FTS. The reason for this is as follows. Using the SSMVS based on the FTS, once a usable view is found in the SMV, the view is returned, which does not guarantee the quality. On the other hand, the SRMVS based on the BRS examines every usable top-view and its descendants as well as every usable independent-view until the best (smallest) usable view is found. Hence, it guarantees that the best usable view in the SMV is

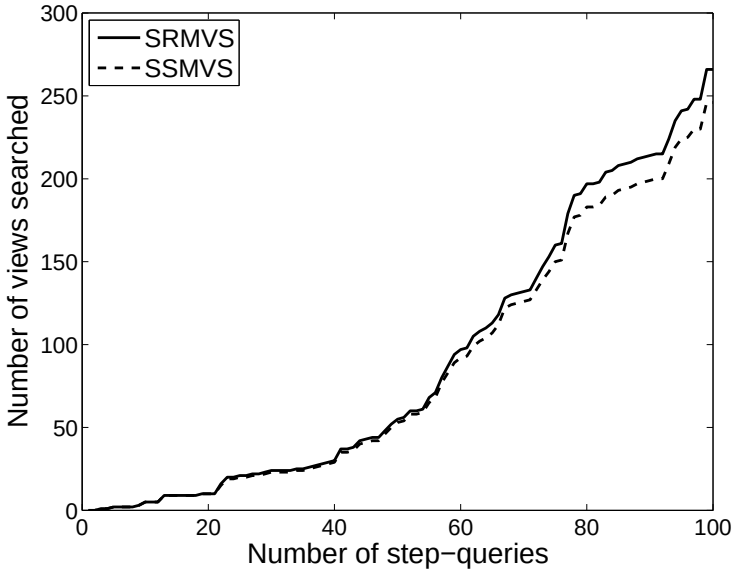

Figure 3.19: Comparison of view searching costs between SSMVS based on FTS and SRMVS based on BRS

returned for the given SQs.

Fig. 3.19 shows the comparison of view searching costs between the SSMVS based on the FTS and the SRMVS based on the BRS. From the figure, we can find that the searching cost for the SRMVS based on the BRS is a little bit higher than that of the SSMVS based on the FTS. This is because the SRMVS based on the BRS has to check all the usable views in the top-view set and their descendants as well as the usable views in the independent-view set. On the other hand, the SSMVS based on the FTS only needs to return the first usable view found in the linear queue of the SMV. If there are many usable views in the SMV, the SSMVS does not incur much cost. Hence, the searching cost of the SRMVS based on the BRS is usually higher than that of the SSMVS based on the FTS. However, due to the capability of the SRMVS for pruning the descendants of non-usable views, the cost difference between the two methods is small.

In the third experiment, both the SRMVS and the SSMVS adopted the BRS. In this case, both

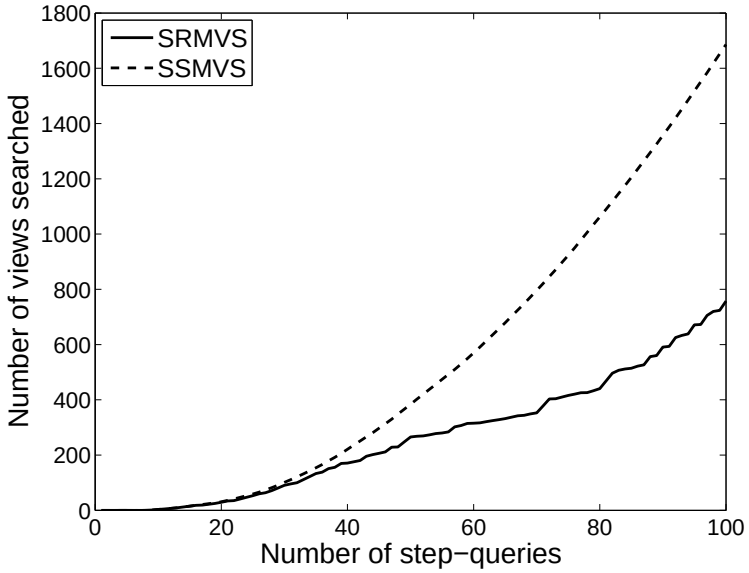

Figure 3.20: Comparison of view searching costs between SSMVS and SRMVS based on BRS

methods had the same view quality since they both guaranteed that the best usable view for a given SQ was returned. Hence, we compared only their view searching costs. Fig. 3.20 shows the comparison of searching costs between the SSMVS and the SRMVS based on the BRS. We observed that the searching cost for the SSMVS based on the BRS was much higher than the one of the SRMVS based on the BRS. This is because the former method has to check all the views to find the best usable view for a given SQ. On the other hand, the descendants of non-usable views are removed (pruned) from consideration by the SRMVS based on the BRS.

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

The results of our research in this chapter were reported in [128, 130].

CHAPTER 4
A Materialized-view Based Approach for Efficiently Processing Generic PQs

The technique presented in the last chapter was designed to process the special monotonic linear PQs. It cannot be used to handle generic PQs since the required superior relationships may not exist among the SQs of generic PQs. We notice that another type of relationships, called the data source dependency relationships, always exist among SQs and external tables of generic PQs. Utilizing such relationships, we develop another materialized-view based approach for optimizing generic PQs in this chapter. A multiple query dependency graph (MQDG) that is used in our technique for processing generic PQs is defined in Section 4.1. Some basic terms of the MQDG are described in Section 4.2. A view storage (VS), which stores result tables associated with two types of nodes in the MQDG, is discussed in Section 4.3. After that, the main processing procedure is introduced in Section 4.4. A mathematical benefit estimation model using the MQDG to identify the critical nodes from completed PQs for view materialization is discussed in Section 4.5. The policy to remove non-critical nodes from the MQDG is presented in Section 4.6. The strategies to insert critical nodes into the critical node view space considering the space limit is discussed in Section 4.7.

4.1 Multiple Query Dependency Graph

In our dynamic materialized view technique for generic PQs, the domain of an SQ sq_1, denoted by $Domain(SQ)$, is defined as the set of input tables of sq_1. As mentioned earlier, due to the query unpredictability property, the result tables for executed SQs of in-process PQs have to be made available (not discarded) 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 (as temporary materialized views) in the system. Conceptually, an SQ only utilizes the result tables for previous SQs of the same PQ. In this work, 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 are very popular and 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.

The main idea of our dynamic materialized view technique for generic PQs 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 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 better 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.

In the remaining of this section, let us define the multiple query dependency graph (MQDG). 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 (a temporary SQ) or a critical node (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. Fig.4.1 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 depended on v_1. In other words, v_2 is generated later than v_1. On the other hand, all the outgoing paths from v_2 are to connect the nodes which are generated later than v_2. Therefore, it is impossible to form a recursive cycle in the graph. Secondly, 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. Similarly, the SQs from different PQs may be connected together in the graph.

Other properties of an MQDG include that each table node has no incoming edge and there is a single sink (final) node for each PQ that returns the final result for the PQ. Note that a dependency

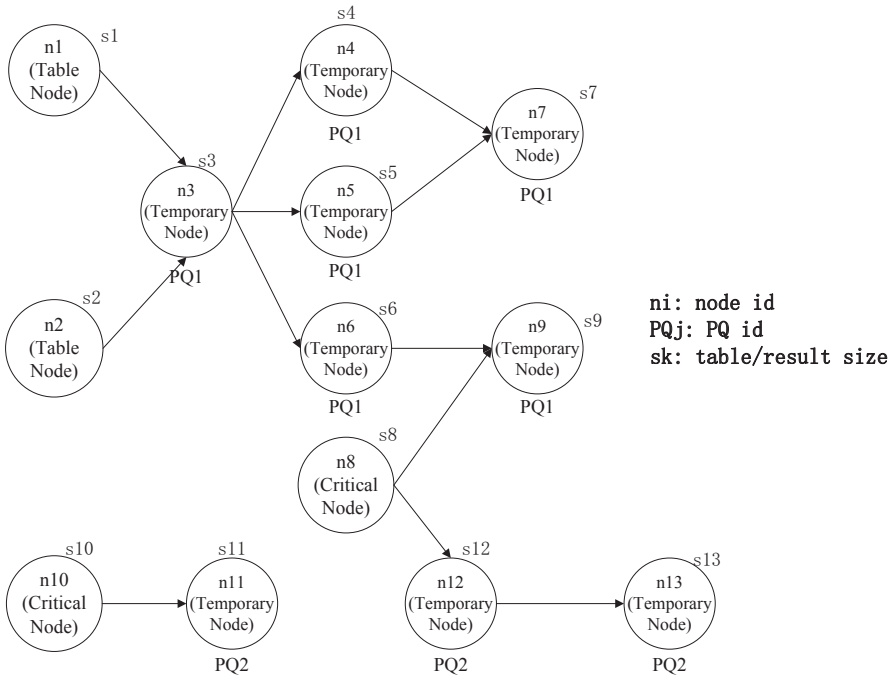

Figure 4.1: An example of the multiple query dependency graph (MQDG)

graph (DG) for a given PQ was defined in [134]. 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.

4.2 Basic terms for MQDG

Next, let us introduce some basic terms 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. See the example in Figure 4.4, 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.

4.3 View Storage

As mentioned earlier, result tables associated with temporary nodes and critical nodes are kept in the system as materialized views. Thus, an empty 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 a set of result tables (view) for temporary nodes, while the CNS is to keep a set of result tables (view) for critical nodes. Note that the related information of a view, e.g., PQ id, the size of the view, query expression, is also saved in the VS. Fig. 4.2 shows the structure of the view storage. The result tables for temporary nodes are also called temporary materialized views(TMVs) while the result tables for critical nodes are also called critical materialized views(CMVs).

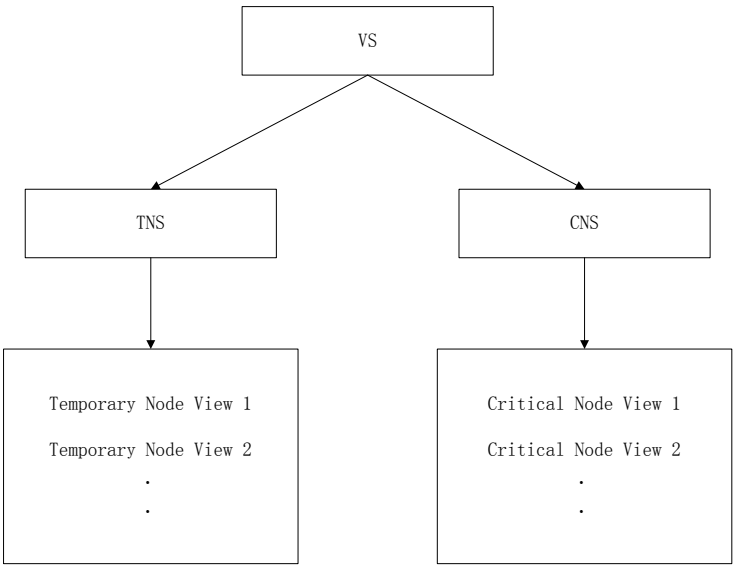

Figure 4.2: The structure of the view storage

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

4.4 Main processing procedure

Since we have finished the approach for processing monotonic linear PQs, next, we consider that what we can do if the generic PQs are issued. It is totally the different situation since the query is more complicated and no monotonic properties can be used. To process a generic PQ, as we

mentioned in Section 4.1, 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 next 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 are presented in this Chapter.

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 Figure 4.3. It starts with a new SQ nsq being issued to the system. nsq can take advantage of all available critical nodes 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 Figure 4.3). 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 Figure 4.3. The detailed description of the main procedure is given by the following algorithm.

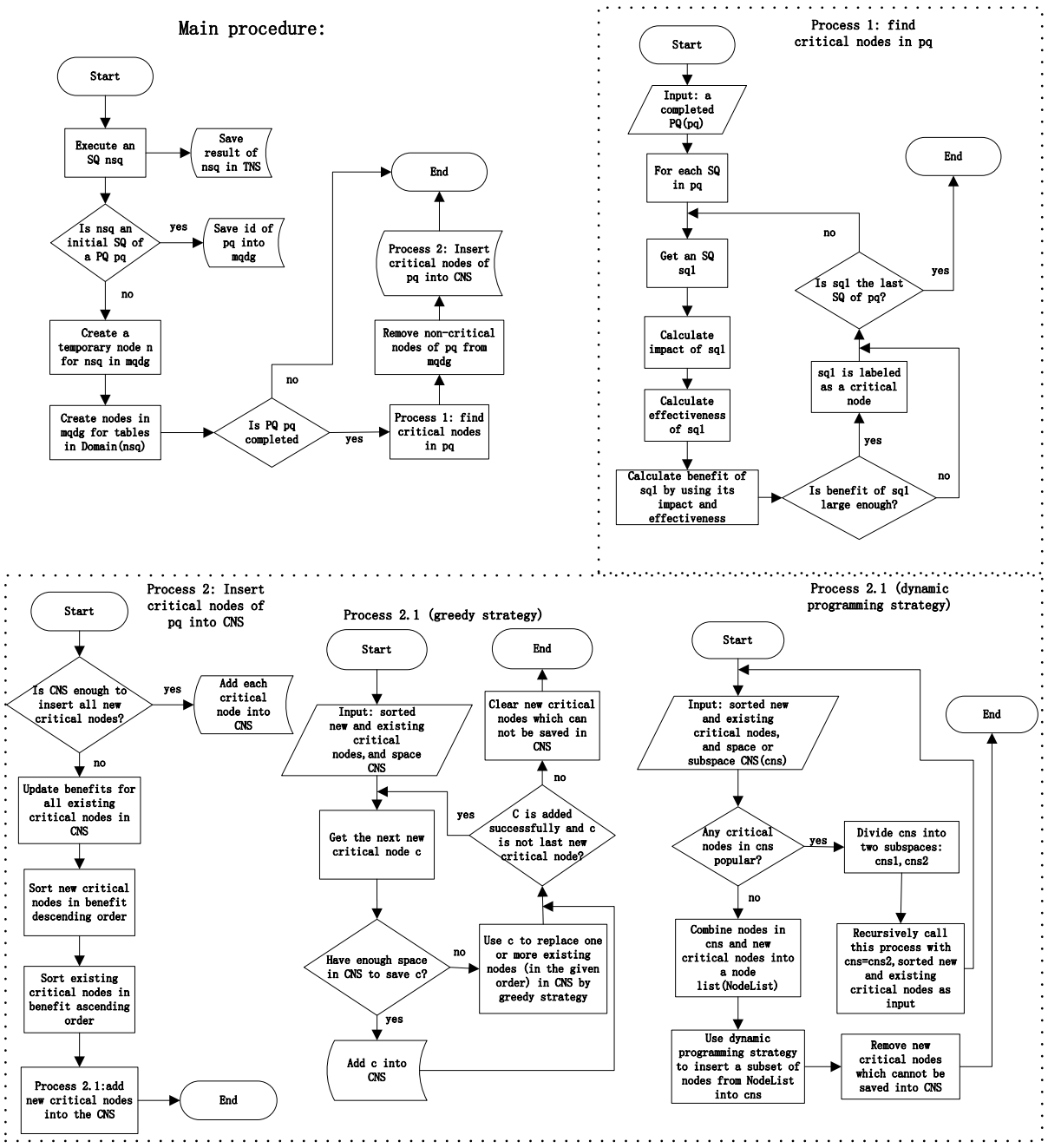

Figure 4.3: the flowchart of the main procedure

ALGORITHM 4.4.1 : **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**.

There are two phases in Algorithm 4.4.1. 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.

4.5 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 indirectly 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 effec-

tiveness 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:

$$Benefit(sq) = (\frac{Imp(sq)}{Imp_{max}})^{\alpha} * (\frac{Eff(sq)}{Eff_{max}})^{\beta} \tag{4.1}$$

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:

$$Imp(sq) = \sum_{c \in ChdS(sq)} \sum_{p \in PthS(sq,c)} \frac{(W_d)^{|p|-1} * [W_E * Ex(sq, c) + W_I * In(sq, c)]}{\prod_{x \in NdeS(p)} NE(x)}, \tag{4.2}$$

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 discus-

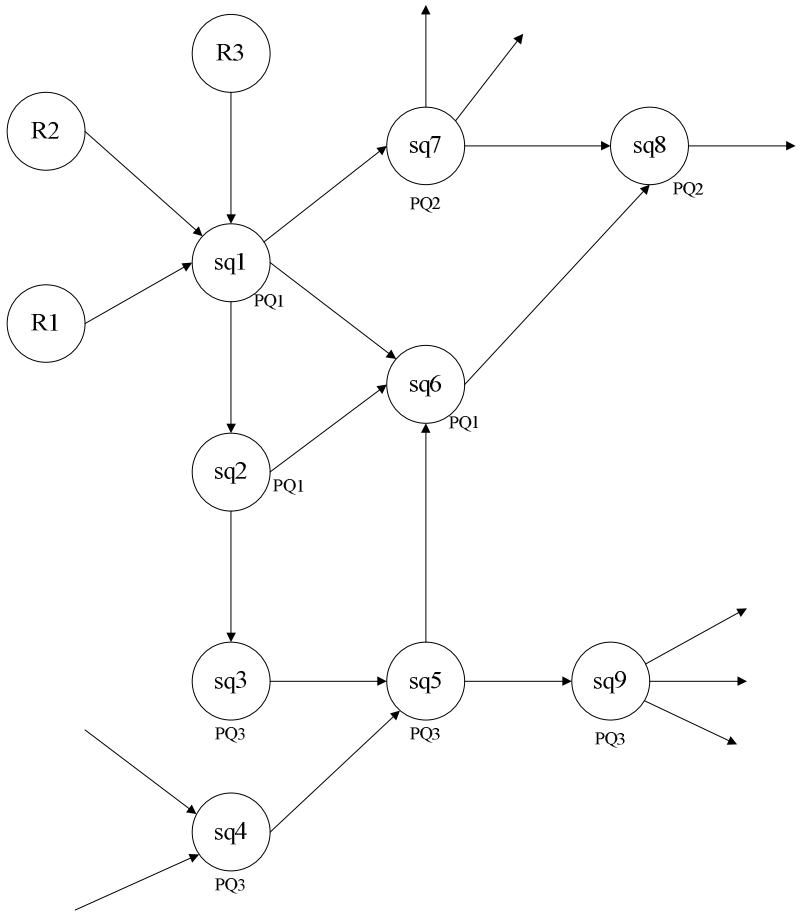

Figure 4.4: The MQDG for the example

sion), 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 Figure 4.4. 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:

$Imp(sq6)$ 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:

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

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

$Imp(sq6)$ 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 deep-first fashion to accumulate the impact values that the node has brought to each of its direct and indirect child nodes.

ALGORITHM 4.5.1 : **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.

In Algorithm 4.5.1, 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 4.5.2 : **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.

Algorithm 4.5.2 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 (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.

$$Eff(sq) = \sum_{c \in DchdS(sq)} \frac{Size(Result(c))}{\sum_{t \in DpadS(c)} Size(Result(t))} \tag{4.3}$$

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 $Size(Result(c))/(\sum_{t \in DpadS(c)} Size(Result(t)))$.

Let us consider the example in Figure 4.4. 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:

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

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

$Eff(sq1)$ 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:

$Eff(sq1)$ 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 4.5.3 : **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 4.5.3, 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., α = β =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,

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

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 Figure 4.3. The detailed algorithm is shown as follows:

ALGORITHM 4.5.4 : **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 4.5.4 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).

4.6 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.

ALGORITHM 4.6.1 : **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 $mqdg$.

In Algorithm 4.6.1, 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: $\sigma_{c_2=v2}$ ($\sigma_{c_1=v1}$ (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 4.6.1. Assume that we are given an MQDG as shown in Figure 4.4. 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 Figure 4.5.

4.7 Critical node view space maintenance

The next issue we want to discuss in this Chapter is how to insert identified critical nodes into the critical node view space (CNS). As we mentioned in Section 4.3, 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

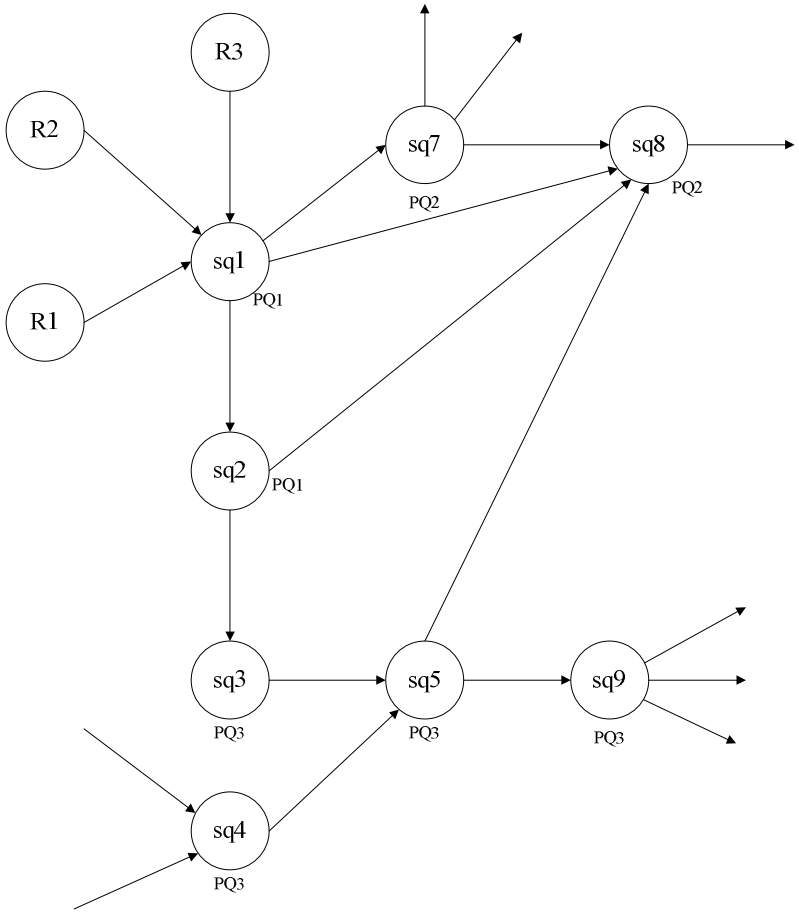

Figure 4.5: An example of MQDG after sq_6 is removed

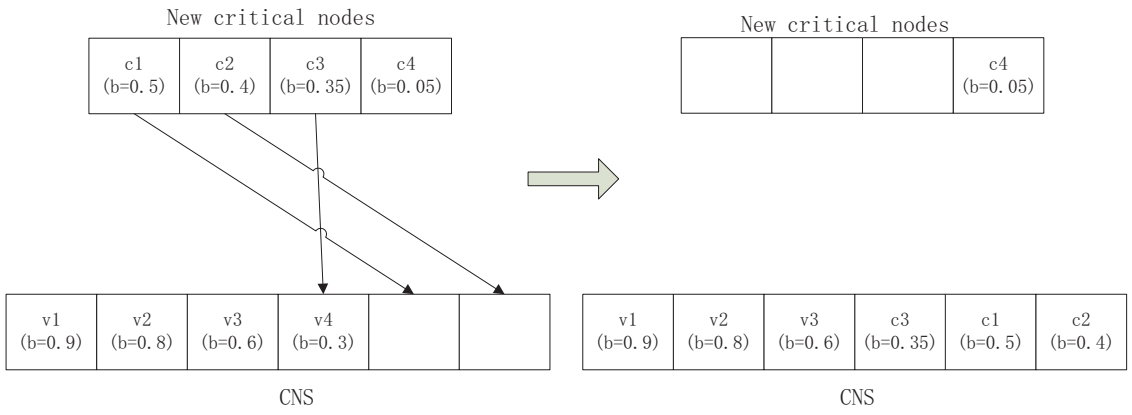

Figure 4.6: An example of the greedy strategy

critical node cannot be accommodated in the CNS. After that, the benefit of the node c whose benefit is the largest among the remaining new critical nodes and the benefit of the node 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 Figure 4.6. 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 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 Figure 4.3. The detailed algorithm, which is invoked in Algorithm 4.4.1, is shown as follows:

ALGORITHM 4.7.1 : **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 $cnset \leq$ 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 $ncBenefitList$ 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** $size(n) \leq$ 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**.

Algorithm 4.7.1 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 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 follows:

ALGORITHM 4.7.2 : **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 4.7.2 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 4.7.3 : **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**

Algorithm 4.7.3 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 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

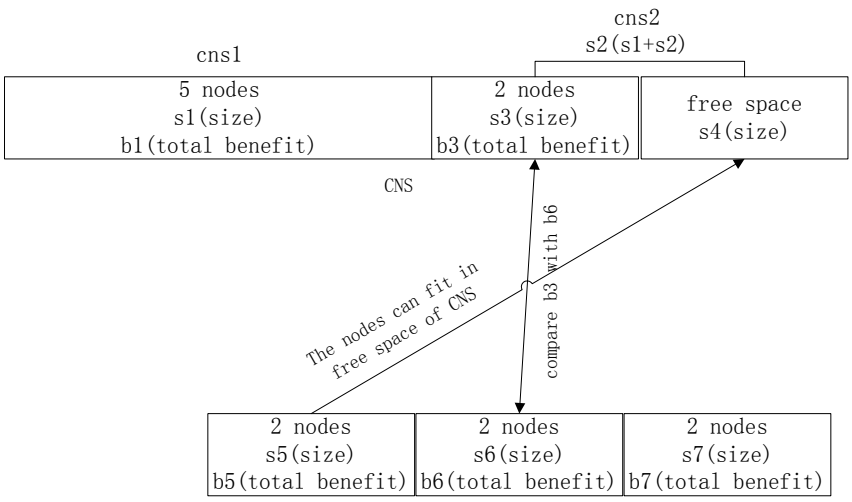

Figure 4.7: An example of the DP based insertion method

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 Figure 4.7 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:

$$size(free\ cns) \ = \ size(cns) - \sum_{j=1}^{k} size(v_j). \tag{4.4}$$

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

$$\sum_{i=1}^{s} size(c_i) \ \leq \ size(free\ cns). \tag{4.5}$$

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:

$$\sum_{c_i \in X} benefit(c_i) \ < \ \sum_{j=1}^{k} benefit(v_j), \tag{4.6}$$

where[3]

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

[2] If $s = 0$, the leading new critical node cannot fit in the free space. Condition (4.5) 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.

$$X = \{c_{i_n} \mid s+1 \le i_n \le t \ AND \ 1 \le n \le m \ AND \ i_u < i_w (for\ any\ u < w) \ AND \ i_m\ is\ the\ 1st\ integer\ in\ [s+1,t]\ such\ that\ \sum_{n=1}^{m} size(c_{i_n}) \le [size(cns) - \sum_{i=1}^{s} size(c_i)] \ AND \ \sum_{n}^{m} size(c_{i_n}) + size(c_j) > [size(cns) - \sum_{i=1}^{s} size(c_i)] \ for\ any\ j \in [i_m+1, t] \} \quad (4.7)$$

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

$$size(cns_1) = \sum_{j=1}^{r} size(v_j) \quad (4.8)$$

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

$$size(cns_2) = size(cns) - size(cns_1). \quad (4.9)$$

When Condition (4.6) is true, we recursively use subspace cns_2 (in place of cns) as the current space to perform the above analysis until Condition 4.6 does not hold for the current space. When Condition (4.6) 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 Figure 4.3. The detailed algorithm is shown as follows:

ALGORITHM 4.7.4 : **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

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

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 4.7.4 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 4.7.5 : **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 (4.5))*/
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 4.7.5 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 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 (4.5)). Lines 12 - 18 check to see which remaining new critical nodes can substitute the existing critical nodes in cns (i.e., using Equation (4.7)). Lines 19 - 33 handle the case in which Condition (4.6) 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 (4.6) holds. In such a case, the algorithm divides the current cns into two subspaces cns_1 and cns_2 based on Equations (4.8) and (4.9) (lines 35 - 41). The algorithm then recursively invokes itself for subspace cns_2 (line 42).

The invoked function DPChecking() is given as follows.

ALGORITHM 4.7.6 : **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 $cnList$;

Algorithm 4.7.6 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 4.7.5 help reduce the input problem size. Hence, our second method using Algorithms 4.7.4 - 4.7.6 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 4.7.1 and 4.7.3.

4.8 Experiments

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

4.8.1 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.

100 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 4.5 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.

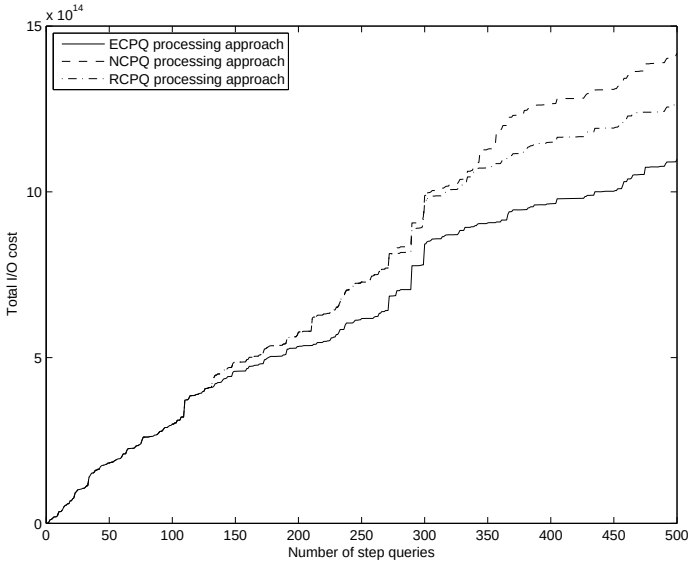

Figure 4.8: Performance of different PQ processing approaches

4.8.2 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 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 Figure 4.8.

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.

4.8.3 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 4.5, 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 4.9 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.

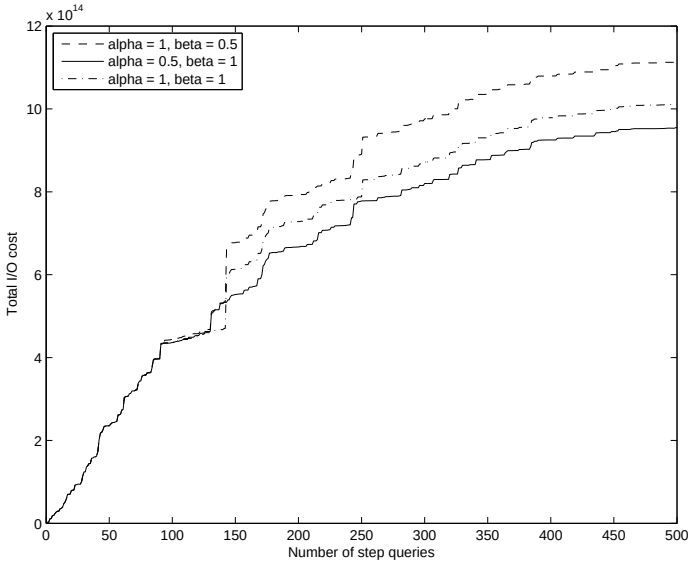

Figure 4.9: Performance of $ECPQ$ with different α and β

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 4.10 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 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

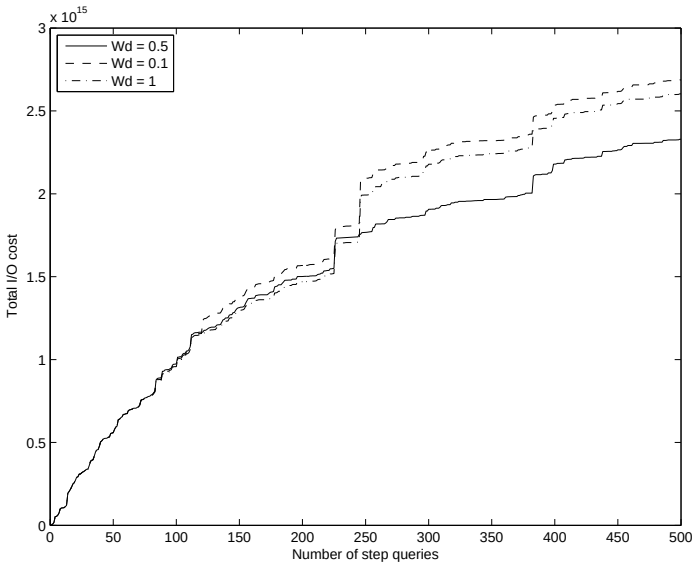

Figure 4.10: Performance of $ECPQ$ with different W_d

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.

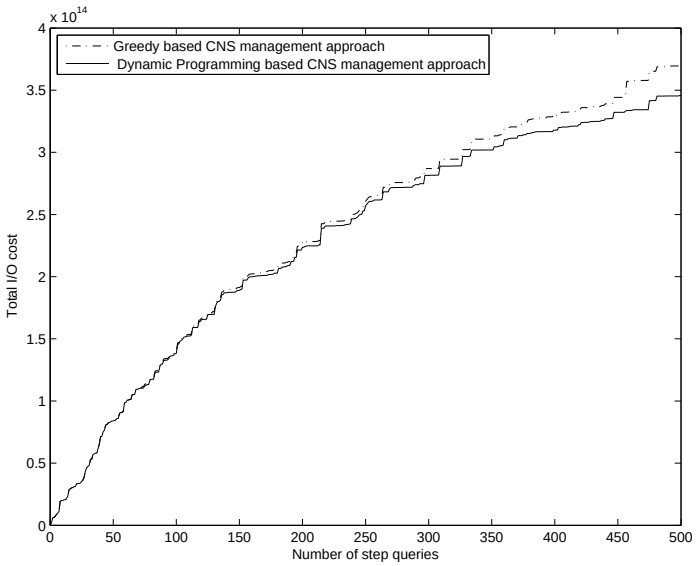

Figure 4.11: Performance of CNS Maintenance Methods

4.8.4 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 4.11 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 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.

4.8.5 Performance of different 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 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 Figure 4.12. 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

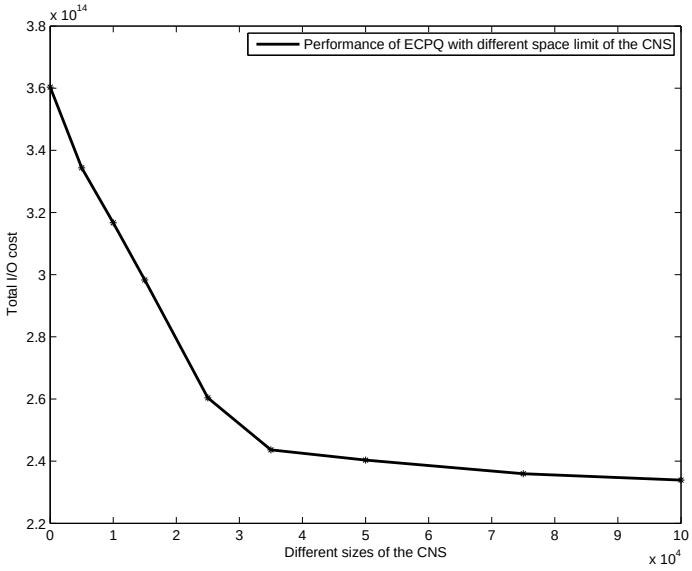

Figure 4.12: Performance behavior for different space limit of CNS

disk blocks.

The results of our research in this chapter were reported in [129, 133].

CHAPTER 5
A Dynamic Materialized View Index

When we discussed a materialized view based approach for processing generic PQs in Chapter 4, our focus was on how to select promising critical nodes. After we have got critical nodes, how to efficiently use them is also an important issue, which has not been discussed. In this chapter, we study how to efficiently find usable views from the VS (view space) for answering SQs. We present a so-called dynamic materialized view index (DMVI). The structure of the DMVI is introduced in Section 5.1. A bitmap matching technique, which is considered as part of the DMVI technique, is presented in Section 5.2. The DMVI construction issue is discussed in Section 5.3. The view search by using the DMVI is described in Section 5.4, and the view maintenance issue for the DMVI is discussed in Section 5.5.

5.1 Index structure

We want to develop an efficient method to search for potential usable views (TMVs or CMVs) from the VS to answer given SQs. A straightforward way to do this is to apply a sequential scan on the VS. Each view in the VS is checked to see if it is a usable view for answering the given SQ. However, the overhead of this approach is usually high, especially when the number of views is large. Note that, in general, matching a view with a given query (i.e., checking if the former can be used to answer the latter) is computationally expensive. Hence, developing an efficient view access method to rapidly identify usable views for answering SQs is crucial in achieving efficient optimization for PQs.

In this chapter, we develop a dynamic materialized view index (DMVI) to efficiently find the views that are potentially usable for answering the SQs. However, the special characteristics of

the materialized views for PQs raise some new challenges. The first challenge is that all the materialized views are dynamically generated while PQs are processed. Therefore, the DMVI has to be dynamically updated to access new views. The second challenge is the high complexity of maintaining the VS since the TMVs are created and removed with a high frequency. Furthermore, all of the CMVs in the VS are transformed/selected from the TMVs. Therefore, the DMVI has to be efficiently maintained in accordance with the VS.

The main idea of the DMVI is to dynamically build an index for all the materialized views in the VS. For each view v in the VS, its corresponding query expression contains the input tables of v. Unlike a conventional index on a table, which uses the attribute values as search keys to find the satisfied rows of the table, the DMVI uses the identifiers of the view input tables as the search keys to find the usable views. We call the set of all the input tables of an SQ (view) the domain of the SQ (view). Hence, we also call an input table a domain table. The criterion used to search the DMVI is that the domain of a usable view is the same as that of the given SQ. Note that, although a view may also be usable if its domain is a superset of the domain of the given SQ, such a view usually does not match the given SQ as closely as a view whose domain equals to that of the SQ. To reduce the number of views returned from the index search, we do not consider the superset criterion. On the other hand, it is not guaranteed that the views returned from the equal-domain criterion are always usable for answering the given SQ. Hence, a refined checking (bitmap based method) on the usability of the returned views is required. Therefore, the DMVI is an approximate index with an objective to return a set S of materialized views for a given SQ such that (1) the views in S match the given SQ as closely as possible; (2) the size of S is as small as possible. The views in S are then examined to see if they can be used to efficiently process the given SQ.

The data structure of the DMVI is an ordered tree in which there is an order among the children of a node. Each leaf node of the tree represents one or multiple materialized views which share the

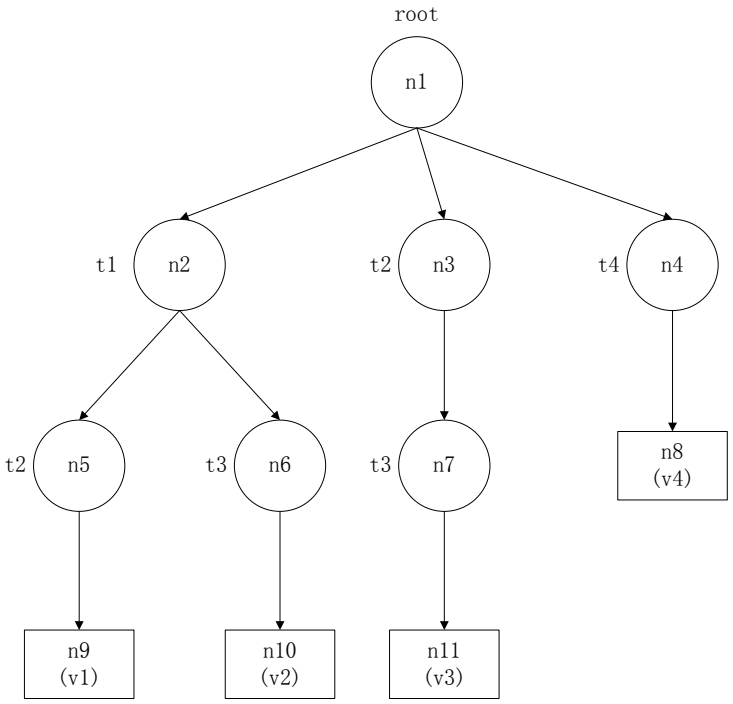

Figure 5.1: An example of the DMVI

same search path in the DMVI. Each internal node n (except the root) represents a domain table t. In other words, n is associated with the identifier of the domain table t. For any view v in a leaf node whose search path contains n, its domain must contain t. The root node of the tree is the starting point for a search. The domain tables labeled on the path between the root and a leaf node for a materialized view v are all the domain (input) tables for v. Note that, for simplicity, we will use a domain table and a domain table identifier interchangeably in our discussion. Fig. 5.1 shows an example of the dynamic materialized view index, where four materialized views v_1, v_2, v_3 and v_4 are indexed and four domain tables t_1, t_2, t_3 and t_4 are used by the views. The domains of v_1 and v_2, for example, are $\{t_1, t_2\}$ and $\{t_1, t_3\}$, respectively. t_1 and t_2 are used as a search key for v_1 in the DMVI.

As mentioned earlier, the first challenge in creating an index for the views in PQ processing

is that all the materialized views are dynamically generated. To tackle this challenge, the DMVI must support how to dynamically incorporate new views. In the previous example, assume that we have another materialized view v_5 whose domain tables are: t_1 and t_4. v_5 can be indexed in the tree in two alternative ways: (1) create an internal node n_{12} labeled with t_4 and connect n_{12} to n_2 as a child, then create a leaf node n_{13} for v_5 and connect n_{13} to n_{12} as a child; (2) create an internal node n_{14} labeled with t_1 and connect n_{14} to n_4 as a child, then create a leaf node n_{15} for v_5 and connect n_{15} to n_{14} as a child. To avoid ambiguity, we need a priority order for the nodes for insertions in the DMVI.

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

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

Case 1: n_1 is chosen to be the next node on the search path for v if the domain of v contains table t_1.

Case 2: n_2 is chosen to be the next node on the search path of v if the domain of v contains t_2 but not t_1.

.....

Case m: n_m is chosen to be the next node on the search path of v if the domain of v contains

t_m but not t_1, t_2, ..., or t_{m-1}.

If none of node n_j ($1 \leq j \leq m$) has its labeled table contained in the domain of v and the domain of v still has tables that have not been labeled on the search path of v, one of unlabeled domain table(s) t is selected and a new internal node representing t is created as a child node of ni. If all domain tables of v have been labeled on the search path of v, a leaf node is created/chosen (if already exists) at the end of the path.

In the previous example, two candidate nodes considered are n_2 and n_4. n_2 is on the left to n_4. Hence, n_2 rather than n_4 is chosen to build the search path of v_5. n_3 is excluded because table t_2 represented by n_3 is not in the domain of v_5.

From the rule, we can observe two properties of the DMVI.

(1) At the first level of the tree (i.e., the level just below the root), if the leftmost internal node n is labeled with a domain table t, then all the indexed views whose domains contain t can be found in the subtree rooted at n.

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

In general, given an internal node $n[m]$ labeled with a domain table t at the m-th level of the tree, if an indexed view v is under the subtree rooted at $n[m]$, t must be the m-th domain table of v; if v is under a subtree rooted at a node on the left to $n[m]$, the node with t as a label can only be found at a level higher ($>$) than m; if v is under a subtree rooted at a node on the right to $n[m]$, the node with t can only be found at a level lower ($<$) than m.

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

accordance with the VS.

On one hand, the DMVI should support a logical level transformation from a TMV to a CMV (not physically move the view). As mentioned earlier, each leaf node of the DMVI stores information of one or more views. The information of a view includes the view name, a view type indicator to differentiate the view types (TMV or CMV), etc. We will discuss the details of the view information structure in Section 5.3. When a view transformation occurs, the view itself and all its related information are kept unchanged except the view type indicator.

On the other hand, it is required to safely and efficiently remove the search paths in the DMVI which are associated with invalid TMVs or CMVs. Furthermore, since in general, an SQ can be formulated using as inputs the external (base) tables, the result tables of the previous SQs of in-process PQs (i.e., TMVs) and the result tables of the SQs of historical PQs (i.e., CMVs), the TMVs and CMVs can also be the domain tables of an SQ besides the external tables. This implies that CMVs and TMVs can appear in the search keys for the views indexed in the DMVI. Therefore, how to adapt search paths in the DMVI that contain invalid TMVs or CMVs is also an important issue. We will discuss the details of the DMVI maintenance in Section 5.5.

5.2 View bitmap-based matching in the DMVI

The main purpose of introducing the DMVI is to efficiently find usable views for answering the SQs. Using the structure of the DMVI introduced in Section 5.1, the system filters out unusable views in the VS and only returns the views that share the same domain as the SQ to be processed. However, as mentioned earlier, the returned views are not guaranteed to be usable for answering the SQ. To reduce the number of cases in which we have to directly examine a returned view for its usability, which is computationally expensive, we adopt an efficient refined filtering mechanism, which is called the bitmap-based matching.

The query expression of a view is encoded as several bitmaps in a special way. The bitmaps are saved in the DMVI. As mentioned before, each leaf node of the DMVI stores the information of a view. The information includes: the view name/identifier, the view type (indicator), the query expression of the view, the view bitmaps, and the view location. Hence, the view bitmaps can be accessed in the leaf nodes of the tree. As we will see, the bitmap encoding method depends on the domain of a query expression (for a view or an SQ). In other words, the bitmap encoding method is the same for those query expressions that share the same domain. When an SQ sq arrives, the system creates the bitmaps for the query expression of sq using a certain bitmap encoding method. Next, the index discussed in Section 5.1 is searched to find all the leaf nodes whose associated views share the same domain with sq. For each view in the returned set, its bitmaps are compared with those for sq. If the bitmaps for a view do not match with that for sq, the view is filtered out. Note that our bitmap matching is different from a conventional view matching. As we will see, even if a view passes the bitmap matching, it still may not be usable for answering sq. A final direct view matching examination is needed. However, using the bitmap matching technique, the number of candidate views for the direct view matching examination is further reduced.

Like most related work in the literature, we consider the common select-project-join query expressions (for SQs and views) and assume that the (qualification) conditions for the select and join operations are in the conjunctive normal form (CNF) in the following discussion.

To encode a query expression (for an SQ or view), a bitmap encoding method is required. As mentioned above, our encoding method depends on the domain of the query expression. Specifically, the bitmap encoding method creates three bitmaps: one for each operation (i.e., project, select and join) of the query expression. Given a domain T (consisting of input tables), its encoding method is described as follows:

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

For each domain table t in T, the project bitmap assigns a bit for each attribute a of t. If a appears in the target attribute list of the project operation, the bit for a in the bitmap is set to 1. Otherwise, the bit for a is set to 0. Let us consider a simple example. Assume that T contains only one domain table t. t has four attributes: a_1, a_2, a_3, and a_4. The encoding method allocates a bit for each of a_1, a_2, a_3 and a_4. If the given query expression is: $\pi_{a_1,a_2}(t)$, then the bits for a_1 and a_2 are set to 1 and the bits for a_3 and a_4 are set to 0. Hence, the project bitmap for this query is 1100.

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

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

If the range of a is restricted by one or more clauses in the CNF of the condition of the select operation, the bits in the bit segment for a are set accordingly. Let us consider a simple example. Assume that attribute a_1 represents the age of a person and its range is from 0 to 150. The bitmap encoding method divides the range of a_1 into 5 subranges: (0, 30], (30, 60], (61, 90], (91, 120] and (120, 150]. Then the encoding method assigns a bit segment which contains 5 bits for a_1. Assume that the given query expression is: $\sigma_{a_1>70}(t)$. In the bit segment for a_1, the bits for the subranges with at least one value satisfying the condition are set to 1, and the other bits are set to 0. Note that, although only some (not all) values in the subrange [61, 90] satisfy the query condition, its corresponding bit in the bit segment is set to 1. Hence, the bit segment for a_1 in this example is 00111.

If the range of a is not restricted by any clause in the CNF of the condition of the select op-

eration, all the bits of the bit segment of a are set to 1. In other words, we take a conservative approach by keeping all the subranges. The select bitmap consists of all the bit segments for the attributes of the domain tables in T.

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

To generate a bitmap for a join operation, all the possible attribute pairs that can be used for a join operation are discovered from the domain tables in T first. Such an attribute pair is called a join pair. For each join pair, it is assigned with a bit in the bitmap. If a join pair appears in the join condition of the query expression (with any comparison such as $=$, $<$, $>$), then its bit in the bitmap is set to 1. Otherwise, the bit is set to 0.

Now let us discuss how to use the bitmaps to compare a given SQ sq with a view v. As we mentioned earlier, the main purpose of the bitmap matching is to filter out unusable views that are returned by the searching on the DMVI tree. The key idea is to prune some views which have no containment relationship with the SQ, namely, the views do not contain the result of the SQ.

Assume that the query expressions of v and sq are encoded using the same bitmap encoding method (i.e., v and sq have the same domain). The process of the bitmap matching can be done in three stages.

In the first stage, the project bitmap pbm_1 for v and the project bitmap pbm_2 for sq are compared. The bit value of 1 represents that its corresponding attribute appears in the result of the relevant query. Therefore, if the bit for an attribute a in pbm_1 is 0 but in $bpbm_2$ is 1, it means that a is in the result of sq but not in v. Hence, the system can conclude that v cannot contain the result of sq. To compare pbm_1 and pbm_2, the system performs a bitwise complement on pbm_2 first and then a bitwise OR on pbm_1 and pbm_2. If the result contains 0, it means v cannot contain the result of sq. For example, if pbm_1 is 00111, pbm_2 is 10011. First, a bitwise complement is performed on

pbm_2 to get 01100. Then, a bitwise OR is applied on pbm_1 and pbm_2, resulting in 01111. Since the result contains 0, v cannot contain the result of sq. Hence, v is filtered out.

In the second stage, the select bitmap sbm_1 for v that has passed the first stage test and the select bitmap sbm_2 for sq are compared. If the bit segment for an attribute a in sbm_1 indicates a narrower range (i.e., missing some 1's) than the bit segment for a in sbm_2, it implies that v restricts the range of a in its select operation more, i.e., the select operation filters out more rows than sq. Hence, there may exist some rows in the result of sq but not in v. In other words, v cannot contain the result of sq. In this case, v should be filtered out. Similar to the first stage, a bitwise complement is performed on sbm_2 first, then a bitwise OR is applied on sbm_1 and sbm_2. If the result contains 0, it implies v cannot contain the result of sq. For example, assume that each of the two bitmaps for v and sq consists of only one bit segment, say, sbm_1 is 00011 and sbm_2 is 00001. First, a bitwise complement is performed on sbm_2, resulting in 11110. Then, a bitwise OR is applied on sbm_1 and sbm_2, resulting in 11111. Thus, the containment relationship between v and the result of sq is still unknown. v needs to be further examined.

In the third stage, the join bitmap jbm_1 for v that has passed the second stage test and the join bitmap jbm_2 for sq are compared. Each bit in the join bitmap indicates the occurrence of a pair of join attributes in the condition of the join operation for a given query. If the join bitmaps of v and sq are different, it is very difficult to determine if the containment relationship between v and sq holds, which makes the view matching examination difficult. To reduce the view matching cost, we exclude such views from consideration. Hence, we require jbm_1 and jbm_2 to be the exactly same. Otherwise, v is filtered out.

From the above discussion, we can see that our complete DMVI consists of the tree structure discussed in Section 5.1 and the bitmaps presented in this section. The objective of the DMVI is to efficiently filter out those views that are clearly unusable for answering the given SQ or very

difficult to perform view matching, resulting in a small set of candidate views. The candidate views are then further examined (i.e., perform view comparison) to see if they are indeed usable for answering the given SQ.

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

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

sq: $\pi_{name}(\sigma_{age>60\ and\ annual\_salary>5000}(T1 \overset{\bowtie}{_{t1\_id=t2\_id}} T2))$;

v: $\pi_{name,\ age}(\sigma_{age>30}(T1 \overset{\bowtie}{_{t1\_id=t2\_id}} T2))$;

First of all, the query expressions of both sq and v are encoded and six bitmaps are generated: $ProjectBitmap_{sq}$ is: 01000; $SelectBitmap_{sq}$ is: 0011101111111; $JoinBitmap_{sq}$ is:1; $ProjectBitmap_{v}$ is: 01100; $SelectBitmap_{v}$ is: 0111111111111; $JoinBitmap_{v}$ is:1.

Next, the system performs a bitwise complement on $ProjectBitmap_{sq}$ (10111) first and then a bitwise OR on $ProjectBitmap_{sq}$ and $ProjectBitmap_{v}$. The result is 11111. No 0 is contained. After that, the system applies a bitwise complement again on $SelectBitmap_{sq}$ (1100010000000) and a bitwise OR on $SelectBitmap_{sq}$ and $SelectBitmap_{v}$. The result is also straight 1. Finally, $JoinBitmap_{sq}$ and $JoinBitmap_{v}$ are compared and they are exactly the same. Therefore, v is

considered to match sq and is returned.

5.3 The DMVI construction

In this section, we discuss the details of the DMVI construction. The DMVI is dynamically created. If no view is indexed, the DMVI contains only a root node. When the result table of an SQ becomes a materialized view v, it is added into the VS and indexed in the DMVI. The main idea is to build a search path p for v in the DMVI using the domain tables of v as the elements of the search key. Each internal node in p represents a domain table (i.e., a search key element) of v. The leaf node which is at the end of p stores the information of v.

In Section 5.1, we defined a priority order for the existing internal nodes of the index tree to determine the unique search paths of new views. In this section, we discuss how to construct the DMVI as an ordered tree. We need two orderings for the domain tables, i.e., the order of the domain tables of new view v and the order of the domain tables for the entire workload. The former determines which domain table (internal node) of v is inserted (created) first. The latter determines where to insert a domain table of v in the tree in relation to other domain tables in the DMVI. Let us consider the following example. Assume that a domain table t is selected and the internal node in representing t is to be inserted into the DMVI as a child node of n. n has one existing child node cn. How to insert in is ambiguous because in can appear on the left or the right to cn. In this case, the order of the domain tables for the entire workload is required. The policy we use is that the domain table with a higher priority appears on the left.

To solve the above two ordering issues, we assign different priorities to different domain tables. A two-level priority rule is used to order the domain tables. At the first level, the domain tables are recognized only by their types (TMVs, CMVs, or external tables). The priority order for these three domain table types from high to low are: TMVs, CMVs, and the external tables. At the

second level, within each table type, an older (i.e., created earlier) domain table is given a higher priority. With the two-level priority rule, the order of the domain tables of v and the order of the domain tables in the entire workload can be determined.

Let us consider an example: given a set of domain tables: T = {cmv_1, et_2, et_1, tmv_2, tmv_1}, where cmv_1 is a CMV; tmv_1 and tmv_2 are TMVs; et_1 and et_2 are external tables. Assume that an older table has a smaller subscript index. To determine the order of the tables in T, the tables are first sorted by the table types: tmv_2, tmv_1, cmv_1, et_2, et_1. After that, the tables are further sorted by their time order within each type. The ordered list is: tmv_1, tmv_2, cmv_1, et_1, et_2.

Now let us discuss how to index a new view v in the DMVI. The basic process is described as follows. All the domain tables of v are sorted by the two-level priority rule. The domain tables in the entire workload are also sorted by the rule and saved in a workload list. The domain tables of v are picked up one at a time in the given order. For the first picked domain table t_1, each child node of the root at the first level of the tree is checked. If there exists an internal node n_1 labeled with t_1, then n_1 is picked as the first node element on the search path for v. Otherwise, a new internal node n_2 representing t_1 is created. In this case, we need to decide where to insert n_2 in relation to other existing first level nodes. Clearly, n_2 must be a child node of the root. In the DMVI, if a node has multiple child nodes, the order (from the left to the right) of these child nodes is determined by the order of their labeled domain tables in the workload list. Thus, the labeled domain table of each child node of the root is compared with that of n_2 one by one from the left to the right according to the order in the workload list. In this way, the system can find a unique place to insert n_2 in relation to other first level nodes. Next, the insertion process is recursively applied to incorporate other domain tables of v into the search path of v. After all the domain tables of v are labeled on the search path, a leaf node is created/chosen (if already exists) to save the information of v.

The following recursive algorithm describes the procedure for inserting (indexing) a new view

(v) into the DMVI ($dmvi$). At the beginning, the algorithm is invoked using the root node of the DMVI (for $cnode$) and the complete list of domain tables of the view (for $cdomainlist$). It assumes that the input lists of the domain tables for both the view ($cdomainlist$) and the workload ($workloadlist$) have been sorted using the two-level priority rule.

ALGORITHM 5.3.1 : **ViewInsertion(**v**,** $dmvi$**,** $cnode$**,** $cdomainlist$**,** $workloadlist$**)**
Input: (1) the new view v for indexing; (2) the DMVI $dmvi$; (3) the node $cnode$ in the DMVI that leads the remaining search path of the view; (4) the list $cdomainlist$ of current (unprocessed) domain tables of the view; (5) the list $workloadlist$ of domain tables in the workload.
Output: the revised DMVI.
Method:

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

The algorithm recursively builds the search path for a new view v in the DMVI. If all domain tables of v have been picked out to build the search path of v (line 1), a leaf node is used (if exist) or created (if not exist) at the end of the path to save the information of v (lines 2 - 7). Otherwise, a domain table $ftable$ from the domain table list of v is picked up, an internal node $inode$ that labels $ftable$ is used (if exist) or created (if not exist), and $inode$ is added into the search path of v (lines

12, 14 - 17, 19 - 21). After that, the function calls itself to insert remaining domain tables of v into the search path (lines 13, 18, 22). Using the above algorithm, we can build the DMVI dynamically by inserting every new view when it becomes available.

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

5.4 View searching using the DMVI

Let us describe how to apply the DMVI to find the usable views in the VS for the view matching. When an SQ sq is issued, the set T of domain tables of sq is extracted and sorted using the two-level priority rule. According to T, a proper bitmap encoding method is applied to generate the bitmaps for sq. The ordered tables in T are used as the search key to find the leaf node ln. For each view v in ln, the bitmaps are extracted and compared with those of sq. If the view is not filtered out by the three-stage bitmap matching, the view is returned. Using the DMVI, as we will see in Section 5.6, the number of the candidate views that are used to perform the final direct view comparison for an SQ is significantly reduced. The details of the view searching algorithm are specified as follows:

ALGORITHM 5.4.1 : **SeachViews(**sq**,**$dmvi$**)**
Input: (1) a new SQ sq; (2) the DMVI $dmvi$.
Output: a set of matched views.
Method:
1. $domain_{sq}$ = Domain of sq;
2. sort domain tables in $domain_{sq}$ using the two level priority rule;
3. encode the query expression of sq using the bitmap encoding method for $domain_{sq}$;
4. $ProjectBitmap_{sq}$ = the project bitmap of sq;
5. $SelectBitmap_{sq}$ = the select bitmap of sq;
6. $JoinBitmap_{sq}$ = the join bitmap of sq;
7. search $dmvi$ using $domain_{sq}$ as the search key;
8. **if** no leaf node found **then**
9. return $\emptyset$;/*return empty*/
/*some views which share the same domain with sq are found.*/
10. **else**
11. n = the reached leaf node;

```
12.  for each view v in n do
13.    ProjectBitmap_v = the project bitmap of v;
14.    SelectBitmap_v = the select bitmap of v;
15.    JoinBitmap_v = the join bitmap of v;
16.    if (-ProjectBitmap_sq) | (ProjectBitmap_v) contains 0 then
     /*'|' represents the bitwise OR; '-' represents the bitwise complement */
17.      continue;
18.    else if (-SelectBitmap_sq) | (SelectBitmap_v) contains 0 then;
19.      continue;
20.    else if JoinBitmap_sq == JoinBitmap_v then
21.      continue;
22.    else
23.      add v into ViewList;
24.    end if
25.  end for
26.  return ViewList;
27. end if.
```

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

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

5.5 The DMVI maintenance issues

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

ALGORITHM 5.5.1 : **PathRemove(**v**,**$dmvi$**)**
Input: (1) the view v to be removed; (2) the DMVI $dmvi$.
Output: the revised DMVI.
Method:
1. $domain_v$ = Domain of v;
2. sort domain tables in $domain_v$ using the two-level priority rule;
3. search $dmvi$ using $domain_v$ as the search key;
4. **if** no leaf node found **then**
5. return;
/*The leaf node which contains v is found.*/
6. **else**
7. n = the reached leaf node;
/*The search path of v is shared by other views in the DMVI.*/
8. **if** n contains multiple views **then**
9. remove the information of v in n and return;
/*The search path of v becomes invalid.*/
10. **else**
/*Remove the search path of v in the DMVI.*/
11. RecursiveRemove(n,$dmvi$);
12. **end if**
13. **end if**.

In this algorithm, lines 1 and 2 extract the domain of the removed view v and sort that domain using the two-level priority rule. Line 3 searches the DMVI by using the domain of v as the search key. If no leaf node is found, it means that the view is not indexed in the DMVI, thus, no further work needs to be done (lines 4 - 5). Otherwise, the discovered leaf node is checked. If the leaf

node contains other views except v, then only the information of v is n is removed (lines 8 - 9). If the leaf node contains only v, then function $RecursiveRemove()$ is called to recursively move the search path of v in the DMVI (lines 11).

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

ALGORITHM 5.5.2 : **RecursiveRemove(**n**,**$dmvi$**)**
Input: (1) a node n in the DMVI; (2) the DMVI $dmvi$.
Output: the revised DMVI.
Method:
1. m = the direct parent node of n in $dmvi$;
2. remove n and its associated links;
3. **if** m is the root or has other direct child nodes **then**
4. return;
5. **else**
6. RecursiveRemove(m,$dmvi$);
7. **end if.**

However, only updating the search path of v in the DMVI is not sufficient. Let us consider an example. After an SQ of a PQ is executed, its result table is saved as a TMV v_1 and indexed in the DMVI. Assume that v_1 is used by some other SQs. When the PQ is completed, the SQ needs to be discarded or transformed into a CMV. In the former case, view v_1 should be removed from the VS. As a result, the search keys (i.e., the search paths) of all the views whose domains include v_1 become invalid. Therefore, in the second stage, for all the views whose domains contain v, their search paths need to be rebuilt.

First, we have to find all the views whose domains contain v. A straightforward way to do this is to traverse the DMVI. However, we can make use of the properties of the DMVI to improve such

a search. According to the first property of the DMVI mentioned in Section 4.2, at the first level of the tree, if an internal node n is the leftmost child node of the root and its labeled domain table t becomes invalid, all the views whose domains contain t can be found in the subtree rooted at n. Furthermore, according to the second property of the DMVI, at the first level of the tree, no matter where node n whose domain table t becomes invalid is, all the views whose domains contain t can be found in the subtree rooted at n or the subtrees rooted at the nodes on the left to n. Therefore, there is no need to search the nodes on the right to n.

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

Let us consider the following example. Assume that, in a given DMVI, four TMVs $tmv_1 \sim tmv_4$ and one CMV cmv_1 are indexed; four external tables $et_1 \sim et_4$ are used as domain tables; tmv_1, tmv_2, and cmv_1 are also used as domain tables. The DMVI is shown in Figure 5.2.

In the figure, we can see that tmv_1 and tmv_2 are labeled by the first level nodes n_2 and n_3 in the DMVI. If tmv_1 becomes invalid, to find all the views whose domains contain tmv_1, only the subtree rooted at n_2 needs to be searched. If tmv_2 becomes invalid, to find all the views whose domains contain tmv_2, only the subtrees rooted at n_2 and n_3 need to be searched.

After all the views v's whose domains contain the invalid view v are discovered, the search path

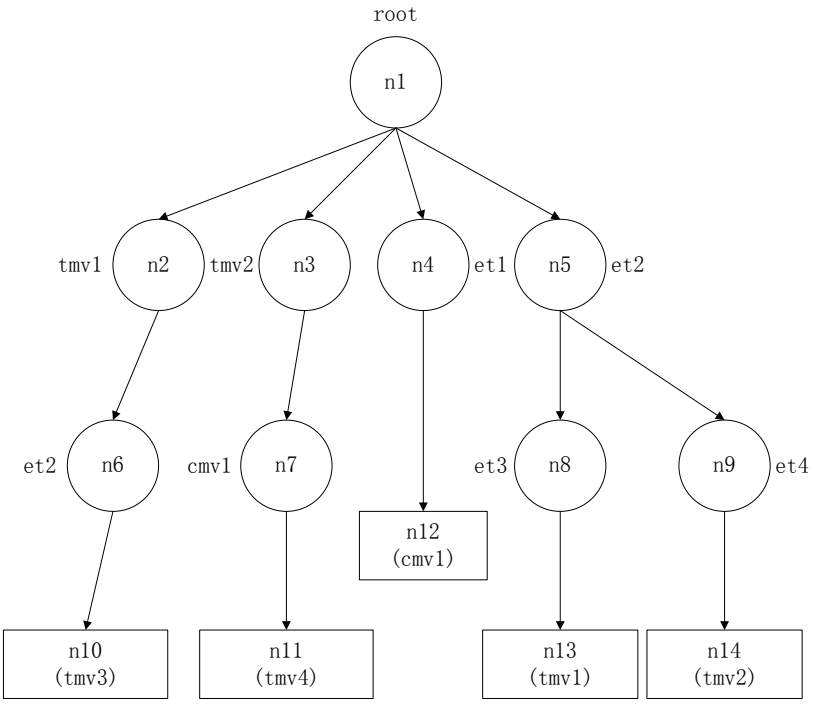

Figure 5.2: An example of the DMVI with views as domain tables

for each v' is rebuilt. The main idea to rebuild the search path for v' is shown as follows. The old search path for v' is removed first. The removing process is similar to that of deleting the search path of a removed view in the DMVI, as described before. Next, the query expression qe of v' is rewritten by merging it with the query expression of v after the occurrence(s) of v is removed, and the domain T of v' is updated by replacing v in T with the domain tables of v. After that, the domain tables in T are sorted using the two-level priority rule, and v' with the updated T is inserted back to the DMVI.

Let us consider the following example. Assume that the domain T_1 of a view v_1 is: $\{ec_1, ec_2\}$; the query expression of v_1 is: $\pi_{ec_1.a_1}(ec_1 \overset{}{\bowtie}_{ec_1.a_1=ec_2.a_2} ec_2)$; the domain T_2 of a view v_2 is: $\{ec_3, v_1\}$; the query expression of v_2 is: $\pi_{ec_3.a_3}(v_1 \overset{}{\bowtie}_{v_1.a_1=ec_3.a_3} ec_3)$. In this example, the domain of v_2 contains v_1. Hence, if v_1 is removed, the search path for v_2 in the DMVI becomes invalid

and has to be rebuilt. The old search path for v_2 is removed first. Based on T_1 and the query expression of v_1, T_2 is changed to $\{ec_1, ec_2, ec_3\}$ and the query expression of v_2 is rewritten as: $\pi_{ec_3.a_3}((\pi_{ec_1.a_1}(ec_1 \overset{}{\underset{ec_1.a_1=ec_2.a_2}{\bowtie}} ec_2)) \overset{}{\underset{ec_1.a_1=ec_3.a_3}{\bowtie}} ec_3)$. After that, the new query expression is saved and the domain tables in T_2 are sorted and used to build a new search path for v_2.

The algorithm to rebuild search paths for all the views whose domains include the invalid views is summarized as follows:

ALGORITHM 5.5.3 : **RebuildPath**(v,$dmvi$,$workloadlist$)
Input: (1) a removed view v; (2) the DMVI $dmvi$; (3) the list $workloadlist$ of domain tables in the workload.
Output: the revised DMVI.
Method:

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

```
31. for each node n in leaf_node do
32.   if n exists in dmvi then
33.     m = direct parent node of n;
34.     while m is labeled by v do
35.       m = direct parent node of m;
36.     end while
37.     remove the subtree rooted at m;
38.     RecursiveRemove(m,dmvi);
39.   end if
40. end for
```

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

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

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

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

5.6 Experiments

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

5.6.1 Experiments setup

The experiment programs were implemented using Matlab 2010 on a PC with Intel® dual core (1.5 GHz) CPU and 4 GB memory running the Windows ® 7 operating system. The underlying DBMS used to run the SQs of a PQ was MySQL.

In our experiments, 100 random progressive queries and 50 random external tables with uniformly distributed data were generated. The number of SQs in each PQ was randomly chosen between 2 and 20. The sizes for external tables ranged from 1 to 1000 disk blocks with each disk block containing 4096 bytes. The experiments were begun by running (the SQs of) the first PQ and ended after having completed all (100) PQs on MySQL. The timestamps were used to record the starting and ending times for SQs of PQs. Multiple PQs were executed simultaneously. The maximum number of PQs that could be run at the same time was set to 10. A DMVI was dynamically constructed to index the materialized views which were generated by the system. The DMVI was also used to efficiently search usable views for answering the SQs.

Each SQ sq was generated in two steps. First, the domain of sq was determined. The domain of sq contains one or more domain tables, where the domain size is randomly chosen between 1 and 5. Each domain table of sq could be either an external table or a materialized view (TMV or

CMV). The probabilities for choosing an external table and a materialized view were set differently in our experiments. We assumed that users preferred to choose previous SQ results (i.e., TMVs and CMVs) over external tables for their new SQs if possible. Hence, CMVs and TMVs were assigned a larger probability (i.e., 0.75) of being chosen than that for external tables (i.e., 0.25). Second, the query expression of sq was built. According to the domain T of sq, attributes were randomly chosen from the domain tables in T to determine the project operations (target attributes), select operations (attributes whose ranges were restricted), and join operations (pairs of joining attributes).

In addition, to construct the VS and the DMVI, some parameters were set. The VS had two subspaces: CNS (for CMVs) and TVS (for TMVs). Since the number of simultaneously executing PQs was constrained ($\leq$ 10), which implies the maximum number of TMVs in the TVS was restricted (only the results of SQs of in-process PQs were saved in the TVS as TMVs), we did not set a space limit for TVS. But, for the CNS, the space limit was set to 25000 disk blocks. The main idea for the CNS maintenance is that when the CNS overflows, its CMVs are re-estimated and sorted by using their potential benefits. The CMVs with the smallest benefit is removed first. This process continues until the CNS can accommodate the new CMV. The DMVI construction started with a single root node. When a materialized view v (TMV or CMV) was saved in the VS, its domain tables were sorted by the two-level priority rule, and a search path was created for v in the DMVI. The bitmaps of v were generated and saved at the end of the path, i.e., in a leaf node. Before each SQ sq was executed, the DMVI was searched and the usable views were returned. When a view (CMV or TMV) v was removed, its corresponding search path in the DMVI was also removed and the search paths of all the views whose domains included v were rebuilt.

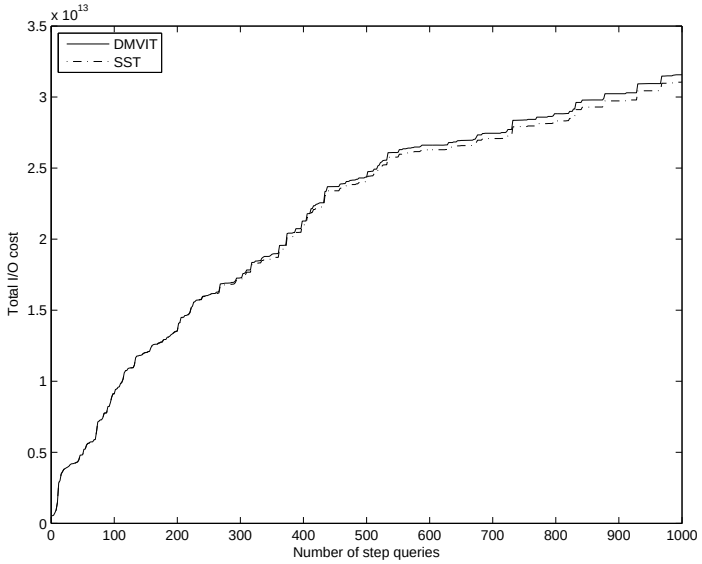

Figure 5.3: The I/O cost comparison of running 100 progressive queries between the SST and the DMVIT

5.6.2 Performance of the DMVI based view searching technique

The first set of experiments were conducted to evaluate the efficiency of our DMVI based view searching technique (DMVIT). To make a comparison, we used the straightforward technique, called the sequential scan based view searching technique (SST), was used. The main idea of the SST is as follows. To find a usable view from the VS for answering the given SQ sq, views are checked one by one from the VS sequentially. If the view v contains the result of sq, then v is considered to be a candidate view. After examining all the views in the VS, the best (smallest) view is chosen from the candidate views to answer sq. In contrast to the SST, our technique first uses the DMVI to filter out the views that do not share the same domain with the given SQ sq. Next, it prunes the views whose bitmaps do not match with those for sq. After that, the discovered views are processed in the same way as the SST, i.e., examining each view against sq to find the best view for answering sq.

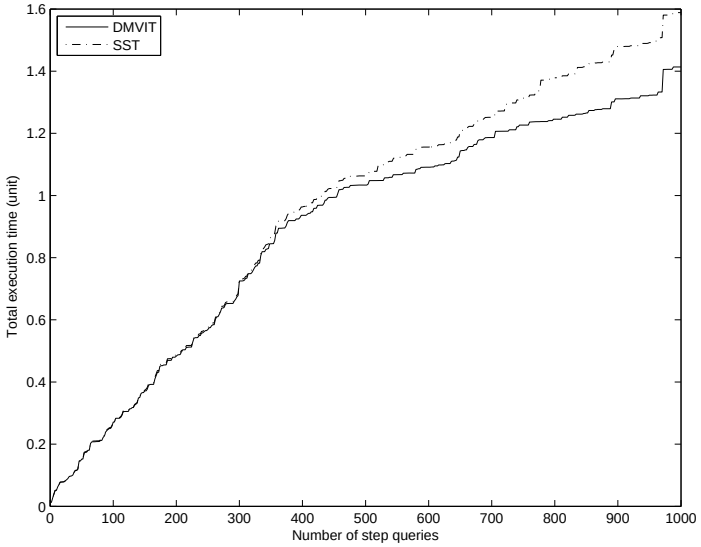

Figure 5.4: The time cost comparison of running 100 progressive queries between the SST and the DMVIT

In this set of experiments, the I/O costs for processing PQs using the two view searching techniques were first compared. Fig. 5.3 shows the comparison of the total I/O cost of running 100 progressive queries between using the SST and the DMVIT. From the figure, we can see that two performance curves are very close, which indicates that two view searching techniques have little different effect on the total I/O cost. Since the I/O cost reflects the quality of views used in the PQ processing, the two techniques are comparable in term of the quality of views found.

To capture both I/O and view matching costs, Fig. 5.4 shows the comparison of execution time of running 100 progressive queries between using the SST and the DMVIT. During the processing of PQs, the materialized views were dynamically generated and indexed into the DMVI. Hence, the DMVI was constructed in parallel with the processing of PQs. The execution time for the DMVI construction was also included in the cost of executing PQs. From the figure, we can see that two curves are very close at the beginning. However, as the total number of SQs being

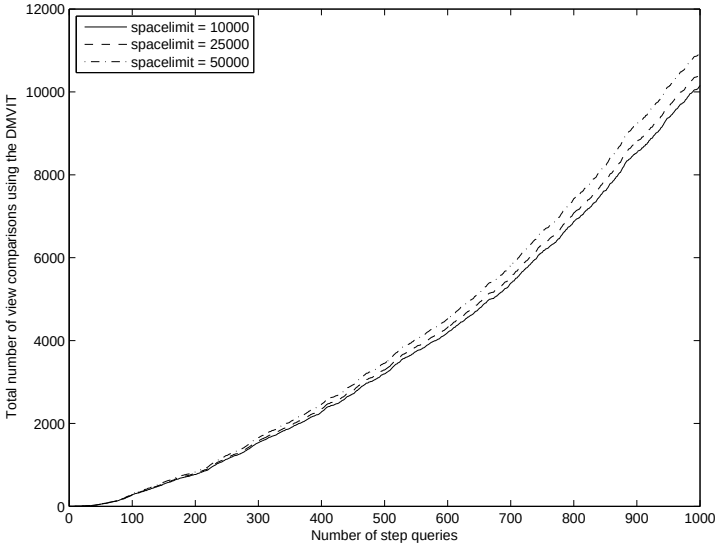

Figure 5.5: The performance of the DMVIT with different CNS space limits

executed increases, the performance of the DMVIT becomes increasingly better. The reason for this is explained as follows. As more and more SQs being executed, more and more SQ results are materialized and kept in the VS. The cost for view searching by the SST (i.e., examining each view in the VS) increases sharply, but the cost for view searching by the DMVIT (i.e., examining only the views discovered by the DMVI) is relatively stable. As a result, the total PQ execution time of the DMVIT is significantly better than that of the SST. Note that, compared to the SST, the DMVIT utilizes an index to significantly reduce the view matching cost..

5.6.3 Scalability of the DMVI based view searching technique

The second set of experiments were conducted to examine the scalability of the DMVIT. The performances of using the DMVIT and the SST with different CNS space limits were compared. Since the maximum number of simultaneously executing PQs was fixed, the size of the TVS was controlled within a certain range. Thus, the size of the VS is dominated by the size of the CNS. The

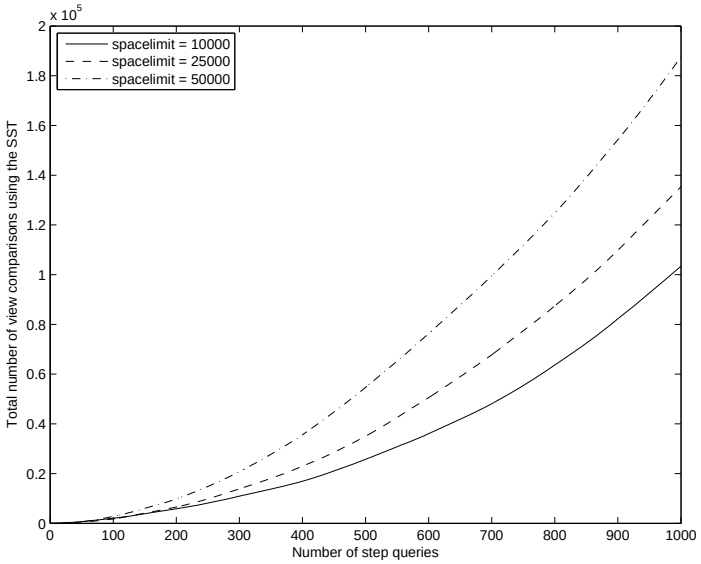

Figure 5.6: The performance of the SST with different CNS space limits

numbers of view comparisons (examining cost) for the DMVIT with different CNS space limits are shown in Fig. 5.5, and the numbers of view comparisons for the SSTs are shown in Fig. 5.6. From Fig. 5.5, we observe that the three performance curves are quite close to each other, which implies that the view matching costs by using the DMVIT with various CNS space limits (thus VS sizes) are relatively stable. However, from Fig. 5.6, we can observe significant differences among the three performance curves. On the other hand, from Section 5.6.2, we know that the PQ performance improves as more materialized views are available. In other words, as more materialized views are available, the DMVIT gains more performance and incurs less searching overhead, comparing to the SST. Hence, the scalability of the DMVIT is better than that of the SST.

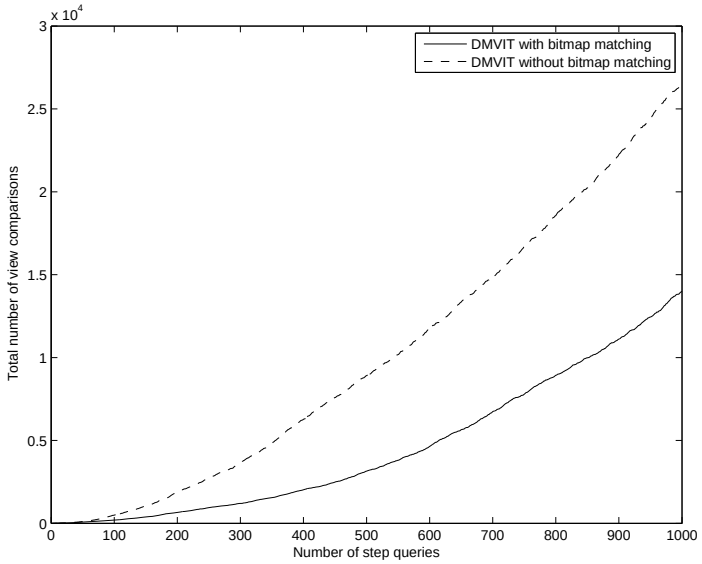

Figure 5.7: The performance of the DMVIT with or without the bitmap matching

5.6.4 Performance of the bitmap matching

The third set of experiments were to examine the importance of the bitmap matching in the DMVI. Fig. 6.5.5 shows the performance of the DMVIT with or without the bitmap matching. It is obvious that much unnecessary direct view comparison cost is saved by using the bitmap matching.

5.6.5 Performance of maintaining the DMVI with different DMVI construction techniques

In this set of experiments, the DMVI maintenance cost for the two-level priority ordering based DMVI constructing technique (TPDMVI) and the DMVI maintenance cost for the non-priority ordering DMVI constructing technique (NPDMVI) were compared. The main idea of the TPDMVI is as follows. When a view v is ready to index in the DMVI, the domain tables of v are sorted by using the two-level priority rule. Next, each domain table of v is picked in the order and inserted into the DMVI. Furthermore, if a node has multiple child nodes, the order of its child

nodes is also determined by the two-level priority order of the domain tables in the workload. The NPDMVI, on the other hand, assigns no priority to any domain table, which causes the DMVI to employ a random order. The purpose of employing the two-level priority order rule is to improve the efficiency of discovering invalid search paths related to a deleted view, which is the major component of work for deleting a view from the DMVI (i.e., maintaining the DMVI). When a view v is to be deleted, the system has to discover all the other views whose domains contain v (i.e., having invalid search paths). If an internal node n representing v appears at the first level of the DMVI (child of the root), only the subtree of n and the subtrees of the left siblings of n (if any) are searched by using the TPDMVI, while the whole tree has to be traversed by using the NPDMVI. The performance of searching the views with invalid search paths by using the TPDMVI and the NPDMVI is compared in Fig. 5.8, where X-axis represents the total number of SQs in the test and Y-axis represents the total number of nodes visited in the DMVI during the search. From the figure, we can see that compared to the NPDMVI, the TPDMVI can save much cost for discovering views with invalid search paths during the DMVI maintenance.

5.6.6 Effectiveness of the DMVI based view searching technique

In the last set of experiments, the effectiveness of the DMVIT was compared with that of the SST. As mentioned earlier, the DMVIT filters out the views whose domains are different from that of the given SQ. However, some views that are filtered out by the DMVIT may be usable for answering the given SQ. For example, a view v_1 that has a different domain from a given SQ sq is filtered out by the DMVIT when searching usable views for sq. Assume that a view v_2 is returned by the DMVIT. However, it is possible that v_1 can be used for answering sq and the size of v_1 is smaller than v_2. In other words, v_1 is more suitable than v_2 for answering sq. In this case, we consider that the most usable view is missed by the DMVIT. We define a hitting rate for the DMVIT as the

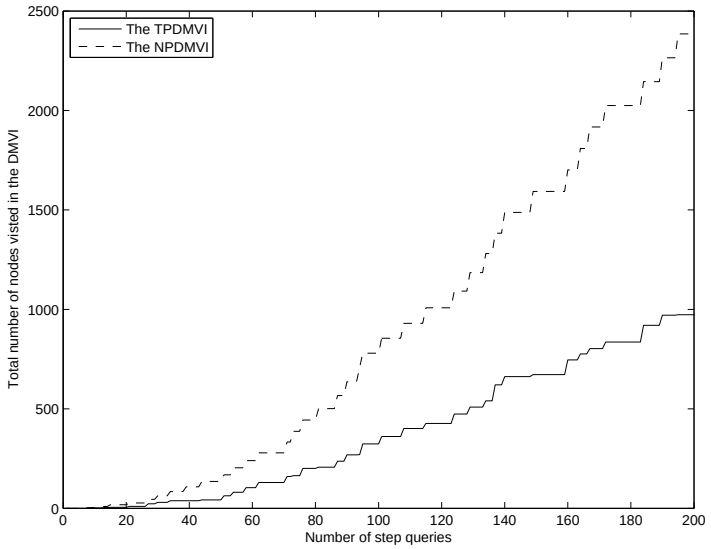

Figure 5.8: The performance of the view searching with invalid search paths between the TPDMVI and the NPDMVI

percentage of the most usable views that can be discovered by it. In this set of experiments, we utilized a commonly used view matching mechanism in a commercial DBMS and calculated the hitting rates of the most usable views discovered by the DMVIT and the SST for the tested cases, respectively. Our results are described as follows. The hitting rates of the SST and the DMVIT were 100% and 83%, respectively; while the numbers of views checked/compared by the SST and the DMVIT during the search were 163546 and 14648, respectively. From the experiments, we can see that, comparing to the SST, the DMVIT can dramatically reduce the number of checked views (by 91%) while keep a high hitting rate (at 83%) when discovering usable views. These results are consistent with the conclusion observed from Figures 5.3 and 5.4. Hence, our proposed DMVI technique is quite effective.

The results of our research in this chapter were reported in [131, 132].

CHAPTER 6
Supporting PQs in Big Data Environment

In the previous chapters, we have presented several materialized view based techniques for optimizing PQs in a relational database environment. Now let us consider how we can handle PQs in a popular big data environment like Hadoop. As mentioned, the SQs of a PQ are issued based on the results of their previous SQs, and some SQs may join the results of the previous SQs with external tables. Therefore, the results of SQs and the external tables must be interoperable in a PQ. This closure property requirement is guaranteed for a relational database since the result of a query on relational tables is still a relational table. However, for a big dataset (database), e.g., an Hbase database on Hadoop, no technique provides such support. In this chapter, as an initial work to support PQs in a big data environment, we define some operations on one or more Hbase tables, which return a new Hbase table as the result. We will focus on discussing the most challenging but very useful operation, i.e., the column family join, in this chapter. Three column family level operations for Hbase tables are introduced in Section 6.1. A direct approach for processing the (column) family join is presented in Section 6.2. A new index structure, called the multiple freedom family index (MFFI), to support efficient family joins is presented in Section 6.3, and an MFFI based family join processing method is discussed in Section 6.4.

6.1 Column family level operations

In a big data environment, the MapReduce programming model and its default implementation Hadoop are frequently used for parallelizing the big data processing. A Hadoop supported database is called Hbase, which is a scalable, distributed database consisting of one or more Hbase tables for storing big data. The query languages for Hbase such as Hive and Pig Latin provide some abilities

for retrieving data from Hbase tables. However, the data results returned by such queries are rarely kept in Hbase tables. In other words, the closure property is not preserved.

The specific reason is as follows. The structure of an Hbase table is quite different from that of a relational table. An Hbase table (instance) consists of one or more ordered rows (vertical expansion) which are identified by their row ids, and its schema consists of one or more so called column families (horizontal expansion). Each column family may contain one or more columns, which are not specified in the table schema. Furthermore, each column may contain a list of values with different versions (e.g., time-stamps). Fig. 6.1 shows the structure of an example Hbase table. Let us see another concrete example of an Hbase table $EmpRelative$ about employed relative names which contains four column families: $spouse$, $parents$, $children$, $others$. Each column family consists of one or more columns, e.g., the column family $parents$ has two columns: $father$ and $mother$, and the column family $children$ has two columns: son and $daughter$. Each column may contain multiple values. For example, the column son may contain multiple values if the employee has more than one son.

Most Hbase query languages map columns from an Hbase table to relational table columns and perform desired operations on them. As a result, the results of those query operations can easily fit into relational tables (i.e., a set of columns without column families) rather than Hbase tables. We observe that, if a query operation could be performed at the column family level (e.g., a projection on a column family of an Hbase table), the results can be easily saved as an Hbase table. Therefore, some new column family level (query) operations on one or more Hbase tables are desired.

In this work, we present three column family level operations: the column family selection, the column family projection, and the column family join, corresponding to three basic relational algebra operations: selection, projection, and join. The difference between a column family level operation and a basic (relational algebra) operation is as follows: a basic operation is performed

Row key	Column-family1	Column-family2		Column-family3		
	Column1	Column1	Column2	Column1	Column2	Column3
key1	t1:val1 t2:val2	t2:val18 t3:val19	t1:val7	t2:val11 t3:val12		
key2	t1:val3 t3:val4	t1:val8 t2:val9	t1:val10		t1:val13	
key3		t2:val5 t3:val6			t2:val14 t3:val15	t2:val16 t3:val17
.....	.....	.....	.....	.....	.....	.....

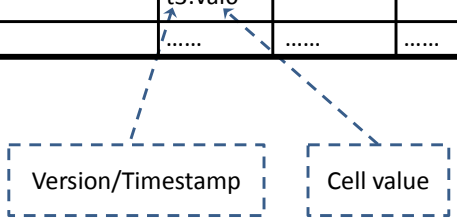

Figure 6.1: An example of an Hbase table

on columns (one value per row) of one or more relational tables while a column family level operation is performed on column families (multiple values per row) of one or more Hbase tables. For example, the set of values for column family $EmpRelative.\ children$ in a row corresponding employee "$James$" may be {son:1:John, son:2:Steven, daughter:1:Jenifer}.

Let us consider the column family join first. The column family join, which can be simply called the family join, is used to integrate matched rows from two Hbase tables into combined rows. Assume that the schemas of two Hbase table are: $T(trid, tcf_1, tcf_2, \ldots, tcf_n)$, $S(srid, scf_1, scf_2, \ldots, scf_m)$, where $trid$ and $srid$ represent the row ids of T and S, respectively; $tcf_i(1{\leq}i{\leq}\text{n})$ is a column family of T and $scf_i(1{\leq}i{\leq}\text{m})$ is a column family of S. In general, the family join is denoted by: $T \bowtie f_{JC}\ S$, where $\bowtie f$ denotes the family join operator, and JC is a Boolean expression which consists of one or more join predicates. Each join predicate is of the form $< T.tcf_i > \theta < S.scf_j >$, where $\theta \in \{ \subset, \subseteq, \supset, \supseteq, =, \doteq \}$. Note that we define a new comparison operator $\doteq$ as: return true if any value from the left ($T.tcf_i$) has a matched value from the right

($S.scf_j$). Join predicates are connected by the Boolean operators and, or, and not to form a JC.

The column family selection and the column family projection can be defined in a similar way by keeping in mind that a column family may contain a set of values. Since these two operations are performed on only one Hbase table and can be easily processed, we will not discuss them in detail in this work. On the other hand, the column family join is performed on two Hbase tables. If it is not properly processed, the join cost may be unacceptable. Thus, in the remaining of this chapter, we will focus on discussing how to properly perform and optimize the column family join.

Let us consider an example of the family join. Assume that we have two Hbase tables T_1 and T_2. T_1 is a table about users and T_2 is a table about employees. Both T_1 and T_2 contain two column families: the ids and the phone numbers. The phone number column family contains three columns: the home phone number, the cell phone number, and the optional phone number. Each column may also contain a set of phone numbers with different versions (timestamps). We want to know the relationships among users and employees. Therefore, a family join operation is desired between T_1 and T_2, with the column families about phone numbers from two tables being used as the join keys. If a value (e.g., a home phone number) of a user is equal to a value (e.g., a cell phone number) of an employee, the user is considered to have a close relationship with the employee, e.g., they may be in the same family, or they may be close friends.

In the above example, we observe that the new operator $\doteq$ is used in the join condition to find any-to-any matched phone numbers. Actually, this type of family joins (using $\doteq$ in the join condition) has a wide range of applications in real life. Thus, as the first work, we chose this type of family joins to consider, and develop some methods to efficiently process them. In the following sections, the term family join will represent a special type of the family join where the only join condition allowed is of the form $T_1.cf_i \doteq T_2.cf_j$.

In fact, such a family join can be done with different types of freedoms. Assume that a family

join is performed on an Hbase table T_1 and an Hbase table T_2 with a join condition: $T_1.cf_1$=$T_2.cf_2$. As mentioned earlier, a column family consists of a set of columns, and each column is composed of a set of values with possibly different versions (timestamps). Thus, each value v_1 in a column family can be represented as: column quantifier (id): version number: v_1. In other words, each value has two prefixes: the column id and the version number. Given a value in cf_1, to find a matched value in cf_2, first, we need to know the matching criterion. Two bits representing two prefixes of values (column id and version number) are used for recognizing the matching criterion. 00 indicates that we only care about the value itself, i.e., given value v_1 in cf_1, the matching criterion is to find the same value in cf_2 without considering its two prefixes. We call the family join using this matching criterion the free family join (FFJ), denoted by $\bowtie f^{(00)}$. 10 indicates that we are also interested in the column (column id) to which the value belongs, i.e., the matching criterion is to find a matched value with the matched column id in cf_2. We call the family join using this matching criterion the column-oriented family join(COFJ), denoted by $\bowtie f^{(10)}$. Similarly, 01 indicates that we are concerned about the version of the value instead of its column id, i.e., the matching criterion is to find a matched value with the matched version number in cf_2. We call the family join using this matching criterion the version-oriented family join(VOFJ), denoted by $\bowtie f^{(01)}$. Finally, 11 indicates that both the column id and the version number are our interest, i.e., the matching criterion is to find a matched value with both matched prefixes. We call the family join using this matching criterion the strict family join(SFJ), denoted by $\bowtie f^{(11)}$. We observe that the freedom of the FFJ is the highest among others since only the value itself is considered while the freedom of the SFJ is the lowest since the value and its two prefixes are all considered.

Let us go back to consider the previous example. If the user only cares about the phone numbers, i.e., do not care about the type of the phone numbers (e.g., a home phone number or a cell phone number) or whether if the phone number is current or not, the FFJ is useful. If the user wants to

find the home phone number matches, the COFJ is suitable. If the user wants to match the current (i.e., the newest version) phone numbers, the VOFJ is favorable. If the user requires to match the newest cell phone numbers, the SFJ should be used.

From the previous example, we can see that the four types of family joins (i.e., FFJ, COFJ, VOFJ, and SFJ) are all useful. Hence, we develop techniques to process a family join with different freedoms.

6.2 A direct family join processing method

How to efficiently process a family join requires a further study, especially when the parallelization is considered. In the rest of the chapter, we discuss two methods to process such a join.

A straightforward method for processing a family join is called the direct family join method (DFJM) in our discussion. Assume that two Hbase tables T_1 and T_2 are family joined on $T_1.cf_1$ and $T_2.cf_2$. The DFJM adopts a two nested loop comparison procedure. At the first level, cf_1 of each row r_1 of T_1 and cf_2 of each row r_2 of T_2 are paired. This is realized by adopting a nested-loop at this level. At the second level, for each pair (cf_1, cf_2), each value v_1 of cf_1 is compared with every value v_2 of cf_2. This is realized by adopting another nested loop at this level. If v_1 matches v_2, r_1 and r_2 are extracted from original tables and joined together. Note that different family join types adopt different value matching criteria. An example of the DFJM for $\bowtie f^{(11)}$ is shown in Fig. 6.2.

The DFJM can also be implemented with parallelization using MapReduce. The main process is described as follows. All rows from T_1 and T_2 are processed and a list of key/value pairs (($rowid$, $tableid$), cf) are produced where ($rowid$, $tableid$) is the key, cf is the value, $tableid$ is the original table id (i.e., T_1 or T_2). cf represents the join key (family column cf_1 or cf_2) of the row with an id of $rowid$. The key/value pairs are assigned to various map nodes to achieve the parallelization. For each map node, the input is a list of key/value pairs. The pairs with the same $tableid$ are

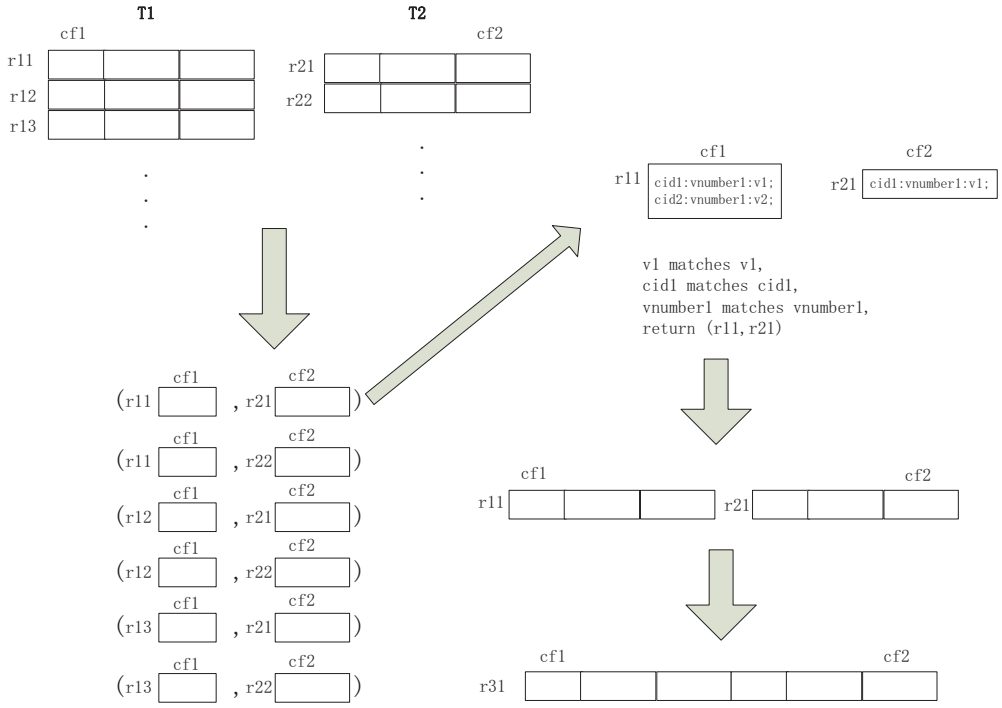

Figure 6.2: An example of the DFJM for $\bowtie f^{(11)}$

grouped (to get two groups) and their $tableid$s are removed, e.g., from((rid_1, t_1), cf_1)) to (rid_1, cf_1). Next, each key/value pair of a group is paired with each key/value pair of the other group, e.g., ((rid_1,cf_1),(rid_2,cf_2)).

After that, a nest-loop comparison is applied on each value v_1 of cf_1 and each value v_2 of cf_2. If v_1 matches v_2 (the matching criteria are different according to various family join types), a key/value pair (rid_1, rid_2) is generated as an output of the map function. The reduce function uses a list of key/value pairs ($rowid$, $rowid$) produced from the map function as its input. Actually, no summarizing job is needed. Hence, the only task for the reduce function is to extract matched rows from original tables and join them together. For each key/value pair (rid_1, rid_2), rid_1 and rid_2 are used as key values to search their corresponding rows (r_1 and r_2) from T_1 and T_2, respectively. After that, r_1 and r_2 are extracted and joined together. It is worth to mention that a new row id has

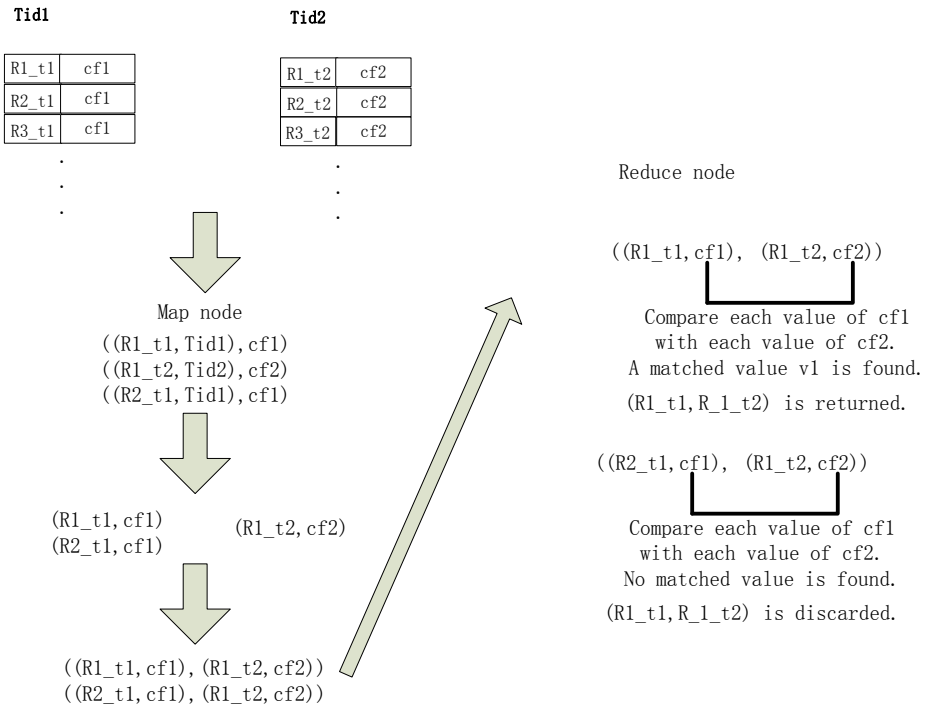

Figure 6.3: An example of the DFJM using MapReduce

to be created for the join result. It can be automatically generated by the system or created by the user. One simple approach is to simply concatenate rid_1 and rid_2 together as the new id. Finally, the join result is produced and returned by the reduce function. An example of the DFJM using MapReduce is shown in Fig. 6.3.

6.3 A multiple freedom family index (MFFI)

Since the DFJM is simple and straightforward, the strategy of using the two nested-loop comparison procedure leads to a high computational cost even if the parallelism is employed. Therefore, we propose an index based method for efficiently processing the family joins. In this section, we introduce a new index first.

6.3.1 Index Structure

The idea of the index is to keep the information on a possible join key (column family) cf_1 of an Hbase table T_1 in a special index structure. Each value v_1 in cf_1 is exacted and used as a key value of the index. Since v_1 can be represented as: rid(row id):cid(column id):$vnumber$(version number):v_1, three prefixes of v_1 (row id, column id, version number) in cf_1 are saved as the content associated with v_1. Since v_1 may appear multiple times in cf_1 (in the same row or different rows) and v_1 should not be duplicated when it is used as a key value in the index, we collect the prefixes of all the occurrences of v_1 from cf_1, organize them in a nested structure, and associate them with v_1 (v_1 as the key value). We name such an index the Multiple Freedom Family Index (MFFI).

Let us illustrate the data structure of the MFFI. The MFFI consists of a number of records(in vertical view), one for each distinct value v_1, where v_1 is a value in a join key cf_1 of an Hbase table T_1. v_1 has three prefixes: row id (to which row v_1 belongs in T_1), column id (to which column v_1 belongs in T_1), and version number (how recent v_1 is). As we mentioned, v_1 may appear multiple times. Thus the prefixes of different occurrences of v_1 have to be kept and organized in a reasonable way. Assume that r_1 is the record in the MFFI for v_1. v_1 is saved as the key value. The remaining of r_1 has a three-level nested structure. At the first level, the space is divided into several subspaces, one for each distinct version number. At the second level, each version subspace is divided into one or more column subspaces, one for each distinct column id. At the third level, each column subspace contains one or more row ids, one for each row containing an occurrence of v_1. Using this structure, we can see that row ids of different occurrences of v_1 with the same column id and version id are placed in one column subspace. Similarly, the row ids of different v_1s with the same version id are gathered in one version subspace. The data structure of the MFFI is shown in Fig. 6.4.

The purpose of creating an MFFI is to efficiently perform family joins. In our approach, one

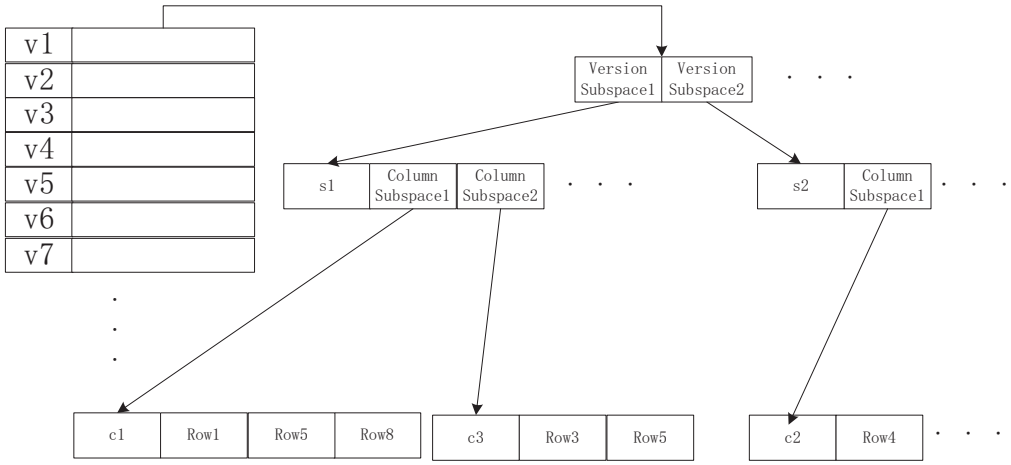

Figure 6.4: The structure of the MFFI

of the important steps is to search row ids of different occurrences of a given value under some search condition. From Fig. 6.4, we can see that different occurrences of a value are organized in the three-level nested structure. Let us illustrate how to search row ids of different occurrences of a value v_1. If the search condition is to satisfy a given version number and a given column id, i.e., 11, at the first level, version subspaces are sequentially searched. The version subspace VS which contains the given version number is accessed. At the second level, the column subspaces contained in VS are searched similarly. After that, the column subspace which contains the given column number is accessed and its contained row ids are returned. If the search condition is to satisfy a given version number, i.e., 01, the second level search is not needed and all the row ids contained in the satisfied version subspace are returned. If the search condition is to satisfy a given column id, i.e., 10, at the first level, all version subspaces are accessed. At the second level, the column subspaces contained in each version subspace are searched and those column subspaces which contain the given column id are accessed and their contained row ids are returned. If no

search condition is issued, i.e., 00, all row ids contained in the record of v_1 are returned.

We can see that with our nested structure, most searches can be efficiently processed, especially for version number specialized searches (only sequential search at the first level). However, our structure is not very good at dealing with column id specialized searches. Thus, if we know that many column id specialized searches will be performed, we can adjust the structure of an MFFI when we create it. We can change the structure to make the column subspaces at the first level and the version subspace at the second level. In this way, the column id specialized searches can be efficiently processed. By default, we choose the version subspaces at the first level. This is because the number of versions is typically less than the number of columns. If one version does not satisfy a search condition, a larger (column) subspace can be pruned.

One of the benefits of such an MFFI is that we can easily save the MFFI itself into an Hbase table without creating a special storage structure. Let us show an example to save a record r_1 of an MFFI into a row r_2 in the Hbase table implementing the MFFI. First, the key value v_1 of r_1 is also saved as the row id of r_2. Next, each version subspace of r_1 is considered as a column family of r_2, the corresponding version id is used as the column family id. Similarly, each column subspace of a version subspace is considered as a column of a column family in r_2 and a set of row ids in a column subspace is considered as a set of different version numbers in a column of r_2.

6.3.2 Creating an MFFI

The next issue is how to create an MFFI $mffi_1$ for a column family cf_1 of an Hbase table T_1. As we mentioned, an MFFI itself is implemented as an Hbase table. Thus, the process of creating $mffi_1$ is also a process to create an Hbase table. We will also use $mffi_1$ to represent its implementing Hbase table in the following discussion. First, $mffi_1$ (Hbase table) is initialized. The schema of $mffi_1$ includes a row key and several column families, corresponding to possible

versions of values, which is expected to be not large. For each row of T_1, the corresponding cf_1 is scanned. For each value v_1 in cf_1, v_1 and its three prefixes (i.e., the version number, the column id and the row id) are extracted. Assume that the version number of v_1 is $vnumber_1$, the column id of v_1 is cid_1, and the row id of v_1 is rid_1. If v_1 is not indexed in $mffi_1$, a new row r_1 is created in $mffi_1$ with v_1 as its search key value (row id). The column family mcf_1 of $mffi_1$ which corresponds to $vnumber_1$ is accessed. A column is created in mcf_1 with cid_1 as its column id and rid_1 as its value. Otherwise, if there exists a row r_1 with v_1 as its row id, $vnumber_1$, cid_1, and rid_1 are directly inserted, namely, inserting cid_1 and rid_1 into the column family mcf_1 which corresponds to $vnumber_1$. More specifically, if there exists a column with the column id of cid_1 in mcf_1, rid_1 is directly inserted. Otherwise, a column is created in mcf_1 with cid_1 as its column id and rid_1 as its value.

Let us consider a simple example. Assume that we want to create an MFFI $mffi_1$ for a column family cf_1 of an Hbase table T_1. T_1 contains only two rows. In the first row, cf_1 has two columns c_1 and c_2; c_1 has two values v_1 and v_2, with the version numbers of $vnumber_1$ and $vnumber_2$, respectively; c_2 has one value v_1, with the version number of $vnumber_2$. In the second row, cf_1 contains one column c_2; c_2 has two values v_1 and v_3, with the same version number of $vnumber_1$. Using the creating procedure we just described, $mffi_1$ is created and shown in Fig. 6.5.

Since the parallelization of the join processing is desirable for large-scale datasets, in this work, we also provide an approach to creating $mffi_1$ using MapReduce. The main idea is as follows. The MFFI creation can be done in four stages. The first stage is called the preprocessing stage. For each row of T_1, its row id and cf_1 are extracted and a key/value pair (rid, cf_1) is generated. The second stage is called the map stage. The key/value pairs produced from the first stage are assigned to multiple map nodes to achieve the parallelization. For each map node, the input is a list of key/value pairs (rid, cf_1). The map function is to extract the values and their prefixes (version

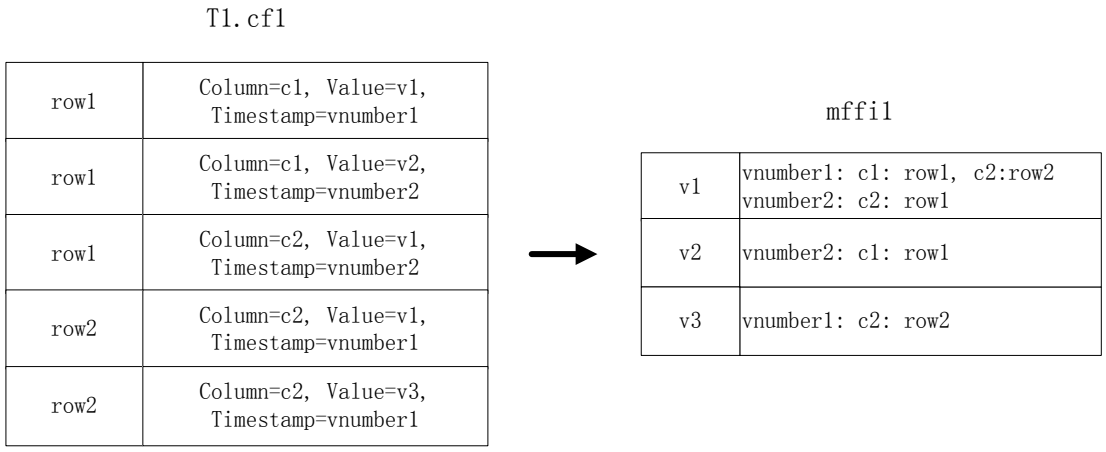

Figure 6.5: An example of the MFFI creation

number, column id) from cf_1. For each value v_1 in cf_1 of row rid, its version number $vnumber_1$ and column id cid_1 are extracted, and a key/value pair (v_1, ($vnumber_1$, cid_1, rid_1)) is returned as an output of the function. The third stage is called the aggregate stage. The key/value pairs produced from the map stage are aggregated together. The key/value pairs with the same key value are gathered and combined. For example, if there are two key/value pairs (v_1, ($vnumber_1$, cid_1, rid_1)) and (v_1, ($vnumber_1$, cid_2, rid_3)), the result of the aggregation is: (v_1, ($vnumber_1$, cid_1, rid_1), ($vnumber_1$, cid_2, rid_3)). The last stage is called the reduce stage. The $mffi_1$ is created in this stage. The input of the reduce function is the result produced from the aggregate stage. Let us use the previous example to illustrate how the reduce function works. The input is: (v_1, ($vnumber_1$, cid_1, rid_1), ($vnumber_1$, cid_2, rid_3)). Assume that $mffi_1$ has already been initialized. A row r_1 is created in $mffi_1$ with the row key value of v_1. Next, each prefix record (version number, column id, row id) of v_1 is sequentially scanned and inserted into r_1. The insertion process has already been discussed previously. As a result, $mffi_1$ can be created in parallel and the creation performance is improved significantly when T_1 is very large. An example of the MFFI creation using MapReduce

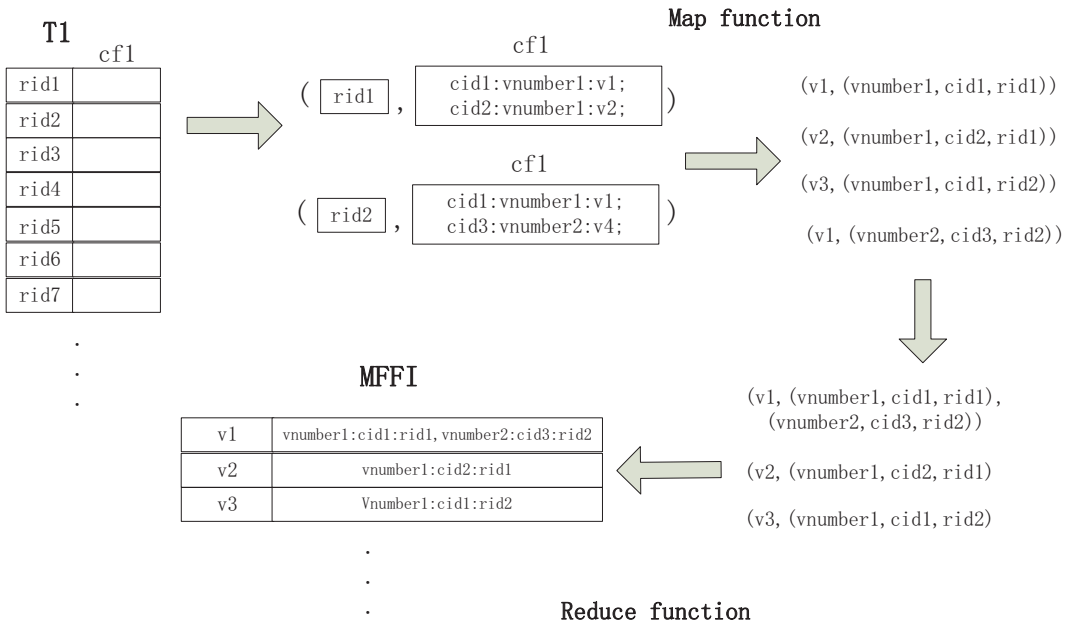

Figure 6.6: An example of the MFFI creation using MapReduce

is shown in Fig. 6.6.

6.4 An MFFI based family join approach

After introducing the MFFI, we now discuss how to use the MFFI to efficiently perform a family join. We propose a so-called MFFI based family join approach (MFJA) for processing the family joins. For the parallelization purpose, the MFJA is developed using MapReduce. In the following discussion, we use SFJ as our default family join type. Let us use the same assumption as before, i.e., T_1 and T_2 are family joined on $T_1.cf_1$ and $T_2.cf_2$. Two MFFIs $mffi_1$ and $mffi_2$ are created for cf_1 and cf_2, respectively. Since the family join is to compare cf_1 of a row r_1 in T_1 with cf_2 of a row r_2 in T_2. If there exists a value v_1 (including its two prefixes: column id and version number) in both cf_1 and cf_2, r_1 and r_2 are considered to be matched and r_1 and r_2 are joined together. Thus,

the main work of the family join is to find matched row pairs (e.g., r_1 and r_2) from T_1 and T_2. As mentioned, an MFFI extracts all the useful information of a column family cf of an Hbase table, where distinct values (e.g., v_1) of cf are saved as the key values of the MFFI, and the row identifiers (row id) which represent rows containing different occurrences of value v_1 are saved as values of key value v_1. Each row id can be represented as: $vnumber$ (version number): cid (column id): rid (row id). Thus, each row id has two prefixes: version number and column id. Having $mffi_1$ and $mffi_2$, we can use them to easily find matched row pairs from T_1 and T_2. The main idea of the MFJA is to perform the join directly on the key values of $mffi_1$ and $mffi_2$. Note that an MFFI is an Hbase table and all the rows are automatically ordered by their key values. Hence, a merge join is performed on $mffi_1$ and $mffi_2$. If a key value of a row mr_1 in $mffi_1$ matches a key value of a row mr_2 in $mffi_2$, we consider a value r_1 in mr_1 to be matched with a value r_2 in mr_2 if the two prefixes of r_1 and the two prefixes of r_2 are matched, respectively. In this case, a row in T_1 with id of r_1 and a row in T_2 with id of r_2 are considered as a row pair and joined together.

More specifically, the MFJA performs a join in five stages. The first stage is called the partitioning stage. In this stage, $mffi_1$ and $mffi_2$ are divided into the same number of partitions. Each partition of $mffi_1$ has a matched partition in $mffi_2$. Two matched partitions are assigned to the same map node. The reason for this is as follows. As we mentioned, $mffi_1$ and $mffi_2$ are merge joined on their keys. To perform the merge join on multiple nodes, $mffi_1$ and $mffi_2$ have to be partitioned in a special way to provide support for the parallelized merge join.

We have considered two strategies for partitioning the MFFIs. A direct method is to partition $mffi_1$ first. $mffi_1$ is equally divided into N partitions (i.e., each partition contains the same number of rows). For each partition of $mffi_1$, the key value of its last row is kept. Next, $mffi_2$ is divided based on each partition of $mffi_1$. For example, assume that p_1 is the first partition of $mffi_1$ and the key value of its last row is v_1 (i.e., the maximum row key value). Based on v_1,

the key values of $mffi_2$ are checked starting from the first row. All the rows with key values not larger than v_1 are included in the first partition p_2 of $mffi_2$. p_2 is considered to be the matched partition for p_1 and they are assigned to the same node. The remaining part of $mffi_2$ is similarly partitioned. Note that if p_2 contains no rows, it means p_1 has no matched partition in $mffi_2$. In other words, no row in $mffi_2$ can be merge joined with a row in p_1. In this case, p_1 and p_2 are not assigned to any node.

However, using the direct partitioning method, the work load for each map node may not be balanced. For instance, partition p_2 in the previous example may be so large that it contains the whole $mffi_2$. Actually, as we will see, the main work of a map node is to process row ids of original tables. Hence, we use the number of row ids being processed to represent the workload of a map node. To balance the workload, an improved partitioning method is suggested below.

The improved partitioning method is called the Counting Index Based Partitioning Method (CIPM). First, a new auxiliary index structure, called the Counting Index (CI), is introduced. A CI ci_1 is created for a specific MFFI $mffi_1$ and saved in another Hbase table. The schema of ci_1 only contains two column families (with a sole column in each), one for keeping every key value v_1 of $mffi_1$, and the other for keeping the accumulated counting number of original table row ids from the first row of $mffi_1$ to the row the key value v_1 in $mffi_1$, e.g., (v_1, 2000) may be an example of a row in ci_1. ci can be easily created while $mffi_1$ is being constructed. When a row r_1 is inserted into $mffi_1$, a row r_2 is also inserted into ci_1, accordingly. The key value of r_1 is saved into the first column of r_2 and the number n_1 of original table row ids in r_1 is counted. If r_1 is the first row of $mffi_1$, it means that the accumulated counting number of row ids is n_1 and n_1 is saved in the second column of r_2 in this case, i.e., the resulting row r_2=(v_1, n_1). Otherwise, the accumulated counting number n_2 of the previous row of ci_1 is extracted, and the new accumulated counting number n_1+n_2 is saved in the second column of r_2, i.e., the resulting row r_2=(v_1, n_1+n_2).

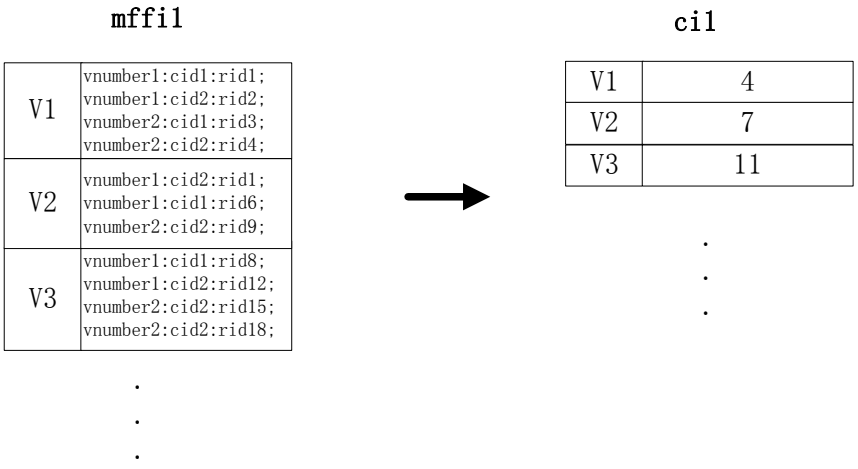

Figure 6.7: An example of the CI

An example of CI is shown in Fig. 6.7.

The CIPM partitions $mffi_1$ and $mffi_2$ in three steps. In the first step, $mffi_1$ and $mffi_2$ are preprocessed and some rows which cannot contribute to the join result are removed from consideration. The main idea is to eliminate unusable rows at the beginning and end of each of $mffi_1$ and $mffi_2$. We know that the key values of $mffi_1$ and $mffi_2$ are in the ascending order since they are used as row ids of the corresponding Hbase tables. First, the key values of $mffi_1$ and $mffi_2$ are scanned from the top to the bottom. Assume that v_1 and v_2 are the key values of the first rows of $mffi_1$ and $mffi_2$, respectively. If v_1 is smaller than v_2, it means that v_1 has no matched (exactly the same) key value in $mffi_2$ and the first row of $mffi_1$ is considered as an unusable row. Next, the key values of $mffi_1$ are checked starting from the second row, this process continues until a row r_1 with the key value which is not smaller than v_2 is found in $mffi_1$. After that, all rows before r_1 in $mffi_1$ are marked as unusable rows and will be removed from consideration in the following partitioning work. Similarly, if v_1 is larger than v_2, the unusable rows at the beginning of $mffi_2$ are removed from consideration in the same way. Second, the key values of $mffi_1$ and

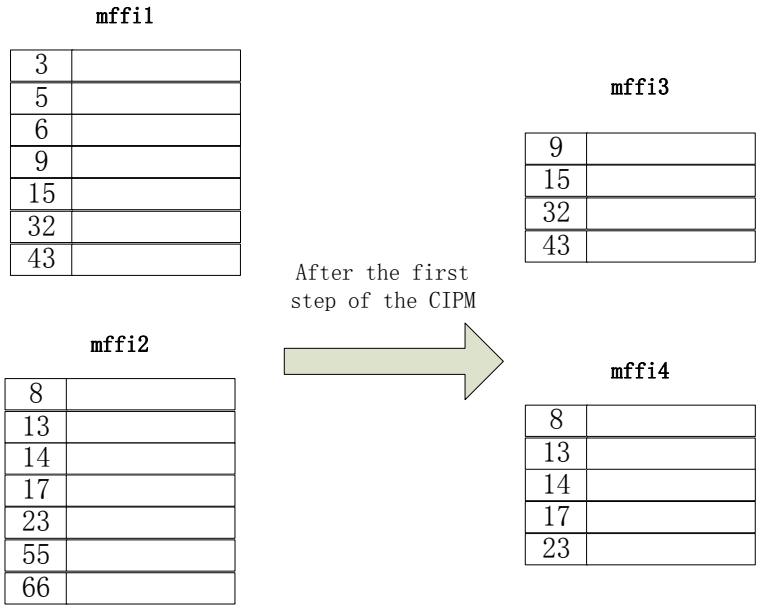

Figure 6.8: An example of the first step of the CIPM

$mffi_2$ are checked from the bottom to the top. Assume that v_3 and v_4 are the key values of the last rows of $mffi_1$ and $mffi_2$, respectively. If v_3 is larger than v_4, it indicates that v_3 cannot match any key value of $mffi_2$ (Reason: v_4 is the largest key value in $mffi_2$, v_3 is larger than v_4. Thus, there exists no key value in $mffi_2$ which is equal to v_3). In this case, the key values of $mffi_1$ are scanned in the reverse order starting from the penultimate row. Once a row r_2 with the key value which is not larger than v_4 is found, the rows after r_2 in $mffi_1$ are marked as unusable rows and removed from consideration. Similarly, if v_3 is smaller than v_4, the unusable rows at the end of $mffi_2$ are removed from consideration in the same way. An example of the first step of the CIPM is shown in Fig. 6.8.

The second step of the CIPM is similar to the direct method. After the preprocessing work, two MFFIs without unusable starting and ending rows are available: $mffi_3$ (revised from $mffi_1$) and

$mffi_4$ (revised from $mffi_2$). $mffi_3$ and $mffi_4$ are divided into the same number of partitions using the direct partitioning strategy, i.e., $mffi_3$ is partitioned first, based on each partition of $mffi_3$, $mffi_4$ is partitioned accordingly. However, the number of partitions here is set to a proper number m which is much larger than the number of partitions for the direct partitioning method. In other words, $mffi_3$ and $mffi_4$ are divided into finer partitions here.

As mentioned earlier, each partition p in $mffi_3$ has a matched partition q in $mffi_4$. p and q are called a pair of matched partitions. Since the second step may yield too many pairs of small partitions (i.e., using too many map nodes), the third step is to combine some pairs of small partitions into a pair of large partitions. Each pair of large partitions will be assigned to a map node for the merge join. Our goal is to control the total size of each pair of large partitions to balance the workload among the map nodes. The procedure starts from the first pair of small partitions: p_1 (from $mffi_3$) and p_2 (from $mffi_4$). Assume that the maximum capacity of a map node is N (i.e., the maximum number of original table row ids that can be processed). Two CIs ci_1 and ci_2 are created for $mffi_3$ (from $mffi_1$) and $mffi_4$ (from $mffi_2$), respectively. Assume that v_1 and v_2 are the key values of the last rows of p_1 and p_2, respectively. ci_1 and ci_2 are searched by using v_1 and v_2, respectively, and the accumulated counting numbers of row ids n_1 (for v_1) and n_2 (for v_2) are found.

If n_1+$n_2 <$ N, which means that the total number of row ids in p_1 and p_2 is smaller than the capacity of a map node can process, the second pair of small partitions p_3 (from $mffi_3$) and p_4 (from $mffi_4$) are checked. The accumulated counting numbers of row ids n_3 (for p_3) and n_4 (for p_4) are found by searching ci_1 and ci_2. Note that the accumulated counting number is computed from the first row of an MFFI, the actual number of row ids in p_3 and p_4 are n_3-n_1 and n_4-n_2, respectively. If n_1+n_2+(n_3-n_1)+(n_4-n_2) $>$ N, i.e., n_3+$n_4 >$ N, it means that, if the second pair of partitions is combined with the first pair, the maximum capacity of a map node is

exceeded. Thus, the first pair of partitions cannot be combined with other pairs. The first pair of partitions is assigned to a map node in this case. Otherwise, if n_3+n_4 < N, which means a map node can handle at least two pairs of small partitions, we form a new pair of partitions by combining p_1 and p_3 and combining p_2 and p_4, and check to see if the next pair of small partitions can be combined. This process continues until the maximum capacity of a map node is reached or exceeded. The combined pair of large partitions is assigned to a map node. For example, two pairs of small partitions are: (p_1, p_2) and (p_3, p_4). After the combination, a new (combined) pair of large partitions is p_5 (contains p_1 and p_3) and p_6 (contains p_2 and p_4). In this way, remaining pairs of small partitions are also combined into pairs of large partitions.

In case n_1+n_2 > N (larger than the maximum capacity of a map node), since $mffi_3$ is equally divided into m partitions and each partition is relatively small (i.e., n_1 is small), it means that n_2 is too large. In this case, p_2 is equally divided into two partitions: p_{21} and p_{22} with sizes n_{21}=n_{22}=(1/2)*n_2. Based on p_{21} and p_{22}, p_1 is also divided into two partitions: p_{11} and p_{12} with sizes n_{11}+n_{12}=n_1. If n_{11}+n_{21}<N, which means that p_{11} and p_{12} can be processed in a map node, no further action is needed. Otherwise, p_{11} and p_{12} are repeatedly divided into smaller partitions until the new created two small partitions can be fit into a map node. The pair of partitions p_{12} and p_{22} is processed in the same way. Clearly, if n_1+n_2=N, no further combination or division is needed. The main procedure of the CIPM is summarized in the following algorithm.

ALGORITHM 6.4.1 : **The counting index based partitioning method (CIPM).**
Input: (1) two Hbase tables (T_1 and T_2); (2) two column families (cf_1 in T_1 and cf_2 in T_2); (3) two multiple freedom family indexes for cf_1 ($mffi_1$) and cf_2 ($mffi_2$), respectively; (4) the number of partitions (m); (5) the maximum workload(processing capability) of a node (N);
Output: partitioned $mffi_1$ and $mffi_2$;
Method:
/* The first step: mark unusable rows at the beginning and end of $mffi_1$ and $mffi_2$ */
1. v_1 = first key value of $mffi_1$; v_2 = first key value of $mffi_2$;
2. v_3 = last key value of $mffi_1$; v_4 = last key value of $mffi_2$;
3. $flag$ = true;
4. **if** $v_1 < v_2$ **then**
5. find the nth row in $mffi_1$ whose the key value is no longer smaller than v_2;
6. mark the first $n-1$ rows starting from the first row in $mffi_1$ as unusable;
7. **else if** $v_1 > v_2$ **then**

8. find the nth row in $mffi_2$ whose the key value is no longer smaller than v_1
9. mark the first $n-1$ rows starting from the first row in $mffi_2$ as unusable;
10. **end if**
11. **if** $v_3 > v_4$ **then**
12. find the nth countdown row in $mffi_1$ whose the key value is no longer larger than v_4;
13. mark the first $n-1$ countdown rows starting from the last row in $mffi_1$ as unusable;
14. **else if** $v_3 < v_4$ **then**
15. find the nth countdown row in $mffi_2$ which the key value is no larger than v_3;
16. mark the $n-1$ countdown rows start from the last row in $mffi_2$ as unusable;
17. **end if**
18. **if** all the rows of $mffi_1$ or $mffi_2$ are marked as unusable **then**
19. return;
20. **end if**
/*The second step: divide $mffi_1$ and $mffi_2$ into small partitions*/
21. remove unusable rows from $mffi_1$ and $mffi_2$
22. create counting indexes ci_1 for $mffi_1$ and ci_2 for $mffi_2$, respectively;
23. initialize four partition lists $partition_1$, $partition_2$, $partitionresult_1$, $partitionresult_2$;
24. evenly partition $mffi_1$ into m partitions;
25. save the boundary of each partition in $partitionlist_1$;
26. partition $mffi_2$ based on each partition of $mffi_1$;
27. save the boundary of each partition in $partitionlist_2$;
/*The third step: combine small partitions to large partitions for $mffi_1$ and $mffi_2$ */
28. **for** each small pair of partitions p_1 and p_2 **do**
29. n_1 = the accumulated number of original row ids in $mffi_1$ before p_1;
30. n_2 = the accumulated number of original row ids in $mffi_2$ before p_2;
31. **if** p_1 and p_2 cannot be processed in one map node **then**
32. m = m-1;
33. p_1 and p_2 are removed from $partitionlist_1$ and $partitionlist_2$, respectively;
34. ($partitionresult_1$, $partitionresult_2$, ci_1, ci_2) = DividePartition($partitionresult_1$, $partitionresult_2$, ci_1, ci_2, p_1, p_2, n_1, n_2);
35. **end if**
36. **end for**
37. n = 0; c_1 = c_2 = 0;
38. **while** $n \leq m$-1 **do**
39. c_3 = c_4 =0;
40. initialize $templist_1$ and $templist_2$;
/* call a function to recursively combine small pairs of partitions */
41. (p_1, p_2, n, c_1, c_2, $flag$) = CombinePartition($partitionlist_1$, $partitionlist_2$, ci_1, ci_2, n, N, c_1, c_2, c_3, c_4, $templist_1$, $templist_2$);
/* the workload of a node cannot accommodate a pair of small partitions */
42. add p_1 to $partitionresult_1$ and add p_2 to $partitionresult_2$;
43. **end while**
44. return $partitionresult_1$ and $partitionresult_2$;

In this algorithm, lines 1 to 20 shows the first step of the CIPM, i.e., preprocessing the two MFFIs and marking unusable rows. The second step of the CIPM is described in lines 21 - 27. Marked unusable rows of $mffi_1$ and $mffi_2$ are removed from the partitioning work (line 21). Two counting index are created for $mffi_1$ and $mffi_2$, respectively (line 22). $mffi_1$ is evenly divided into small partitions and the boundary of each partition is saved in a list for the later use (lines 24 - 25). According to each partition of $mffi_1$, $mffi_2$ is partitioned accordingly and the

boundary of each partition is also saved in a list (lines 26 - 27).

The last step of the CIPM is done in lines 28 - 43. As mentioned earlier, a partition in $mffi_1$ and its matched partition in $mffi_2$ are called a pair of matched partitions. In fact, some small pairs of partitions may not actually be small, This happens when a small partition in $mffi_1$ has a very large corresponding partition in $mffi_2$. In this case, such a pair (p_1, p_2) of partitions cannot fit into a map node. These pairs are processed first. In Algorithm 6.4.2, a recursive function DividePartition() is called to divide p_1 and p_2 into smaller partitions until the new created pairs of partitions can fit into a map node (line 34). All other pairs of small partitions are sequentially processed (lines 38 - 43). A recursive function CombinePartition() is called to combine some pairs of small partitions into a pair of large partitions (line 41). For each pair (p_1, p_2) of large partitions produced from the function, p_1 (from $mffi_1$) and p_2 (from $mffi_2$) are saved into two result partition lists, respectively (line 42). Finally, two result partition lists are returned (line 44). The recursive functions DividePartition() and CombinePartition() are shown as follows:

ALGORITHM 6.4.2 : **DividePartition**
Input: (1) two lists of partitions ($partitionresult_1$ and $partitionresult_2$); (2) two counting indexes for $mffi_1$ (ci_1) and $mffi_2$ (ci_2), respectively; (3) two partitions p_1 and p_2 from $mffi_1$ and $mffi_2$, respectively; (4) two counting variables n_1 and n_2 for the accumulated numbers of rows before p_1 and p_2 in $mffi_1$ and $mffi_2$, respectively;
Output: two updated lists of partitions ($partitionresult_1$ and $partitionresult_2$); two updated counting indexes (ci_1 and ci_2)
Method:
1. p_2 is equally divided into two partitions p_{21} and p_{22}; p_1 is divided into p_{11} and p_{12}, accordingly ;
2. update ci_1 and ci_2 accordingly;
3. v_{kj} = the key value of the last row of p_{kj} (k=1,2; j=1,2);
4. n_{kj} = the accumulated number of original row ids for v_{kj} from ci_k (k=1,2; j=1,2);
5. **if** (n_{11}-n_1)+(n_{21}-n_2) $> N$ **then**
6. ($partitionresult_1$, $partitionresult_2$, ci_1, ci_2) = DividePartition($partitionresult_1$, $partitionresult_2$, ci_1, ci_2, p_{11}, p_{21}, n_1, n_2);
7. **else if** (n_{11}-n_1)+(n_{21}-n_2) $\leq N$ **then**
8. add p_{11} into $partitionresult_1$ and add p_{21} into $partitionresult_2$;
9. **end if**
10. **if**(n_{12}-n_{11}-n_1)+(n_{22}-n_{21}-n_2) $> N$ **then**
11. ($partitionresult_1$, $partitionresult_2$, ci_1, ci_2) = DividePartition($partitionresult_1$, $partitionresult_2$, ci_1, ci_2, p_{21}, p_{22}, n_{11}+n_1, n_{21}+n_2);
12. **else if** (n_{12}-n_{11}-n_1)+(n_{22}-n_{21}-n_2) $\leq N$ **then**
13. add p_{11} into $partitionresult_1$ and add p_{21} into $partitionresult_2$;
14. **end if**
15. return $partitionresult_1$, $partitionresult_2$, ci_1 and ci_2;

This function divides a pair of partitions (p_1, p_2) into smaller pairs of partitions which can be fit

into a map node. p_2 is equally divided into two partitions: p_{21} and p_{22}. Based on p_{21} and p_{22}, p_1 is divided into p_{11} and p_{12}, accordingly (line 1). Since p_1 and p_2 are divided into smaller partitions, ci_1 and ci_2 are updated accordingly (line 2). The key value of the last row of each new partition (i.e., p_{11}, p_{12}, p_{21}, and p_{22}) is extracted (line 3). Next, the accumulated number of original row ids for each new partition is searched on updated ci_1 or ci_2 by using the key value of its last row (line 4). If p_{11} plus p_{21} exceed the maximum capacity of a map node, the function recursively call itself to divide (p_{11}, p_{21}) into two smaller pairs of partitions (lines 5, 6). Otherwise, if p_{11} and p_{21} can be fit into a map node (line 7), p_{11} and p_{21} are directly added into two result partition lists, respectively (line 8). Similarly, p_{12} and p_{22} are processed in the same way (lines 10 - 14). Finally, two result partition lists are returned (line 15).

ALGORITHM 6.4.3 : **CombinePartition**
Input: (1) two lists of partitions ($partitionlist_1$ and $partitionlist_2$); (2) two counting indexes for $mffi_1$ (ci_1) and $mffi_2$ (ci_2), respectively; (3) the maximum workload capacity of a node (N); (4) the number of processed pairs of partitions (n); (5) four counting variables (c_1 and c_2 for the numbers of rows which have been successfully partitioned and combined in $mffi_1$ and $mffi_2$, respectively; c_3 and c_4 for the numbers of rows for partitions in $templist_1$ and $templist_2$, respectively); (6) two temporary lists of partitions ($templist_1$ and $templist_2$).
Output: two combined larger partitions of $mffi_1$ and $mffi_2$, three updated counting variables (n for the number of processed pair of partitions, c_1 and c_2 for the numbers of rows which have been successfully partitioned and combined in $mffi_1$ and $mffi_2$, respectively), a flag to show the combining work is successful or not.
Method:
/* get a pair of small partitions: p_1 and p_2 */
1. p_1 = the (n+1)-th partition in $partitionlist_1$; p_2 = the (n+1)-th partition in $partitionlist_2$;
2. v_1 = the key value of the last row of p_1; v_2 = the key value of the last row of p_2;
/* compute the number of ids in p_1 and p_2 */
3. c_5 = the accumulated number of original row ids for the key value v_1 from ci_1;
4. c_6 = the accumulated number of original row ids for the key value v_2 from ci_2;
5. c_5 = c_5 - c_1, c_6 = c_6 - c_2;
6. $flag$ = true;
/* the maximum workload of a node is exceeded */
7. **if** $c_5 + c_6 + c_3 + c_4 > N$ **then**
8. return (null, null, n, c_1, c_2, $false$);
/* the maximum workload of a node is not exceeded */
9. **else**
10. n = n +1;
11. c_3 = c_3 + c_5, c_4 = c_4 + c_6;
12. c_1 = c_5+c_1, c_2 = c_6+c_2;
13. add p_1 into $templist_1$ and p_2 into $templist_2$;
/*call itself to accommodating another pair of small partitions (if any) */
14. **if** there are still unprocessed partitions in $partitionlist_1$ and $partitionlist_2$ **then**
15. (p_3, p_4, n, c_1, c_2, $flag$) = CombinePartition($partitionlist_1$, $partitionlist_2$, ci_1, ci_2, m, $workload$, c_1, c_2, c_3, c_4, $templist_1$, $templist_2$);
16. **end if**
17. **if** flag == false **then**
18. p_3 = combine partitions in $templist_1$;
19. p_4 = combine partitions in $templist_2$;

20. $flag$ = true;
21. return (p_3, p_4, n, c_1, c_2, $flag$);
22. **else**
23. return (p_3, p_4, n, c_1, c_2, $flag$);
24. **end if**
25. **end if**

The function finds the next pair of small partitions (p_1, p_2) (lines 1 - 2). The actual numbers c_5 and c_6 of original row ids in p_1 and p_2 are computed, respectively (lines 3 - 5). If the summation of the numbers of original row ids in p_1 and p_2 and the numbers of row ids in all the partitions in the two temporary partition lists is smaller than or equal to the maximum capacity of a map node (line 9), it means that a map node can process p_1 and p_2 together with all the partitions in temporary partition lists. Thus, p_1 is added into the temporary partition list $templist_1$ for $mffi_1$ and p_2 is added into the partition list $templist_2$ for $mffi_2$ (line 13). After that, the function recursively calls itself to check another pair of small partitions (if any) to see if it can be included in the same node (lines 14 - 16). If the newly checked pair of partitions cannot be accommodated, the partitions in current $templist_1$ and $templist_2$ are combined, respectively (lines 18 - 19) and the result partitions are returned (line 21). Otherwise, if the summation of the numbers of row ids in p_1 and p_2 and the numbers of row ids in all the partitions in two temporary partition lists is larger than the maximum capacity of a map node, p_1 and p_2 cannot be added into the temporary partition lists. No combined partition is returned in this case (lines 7 - 8).

Let us consider an illustrative example in Fig. 6.9. Two MFFIs $mffi_1$ and $mffi_2$ are partitioned using the CIPM. In the first step, the first and second rows of $mffi_1$ are marked as unusable rows and do not participate in the remaining partitioning work. In the second step, $mffi_1$ is equally divided into four partitions: p_1, p_3, p_5, and p_7. Each partition contains two rows. Next, $mffi_2$ is partitioned accordingly. Since the key value of the last row of p_1 is 6, all the rows with the key values no larger than 6 in $mffi_2$, i.e., the first four rows of $mffi_2$, constitute the first partition p_2 of $mffi_2$. Similarly, the remaining part of $mffi_2$ are divided into three partitions: p_4,

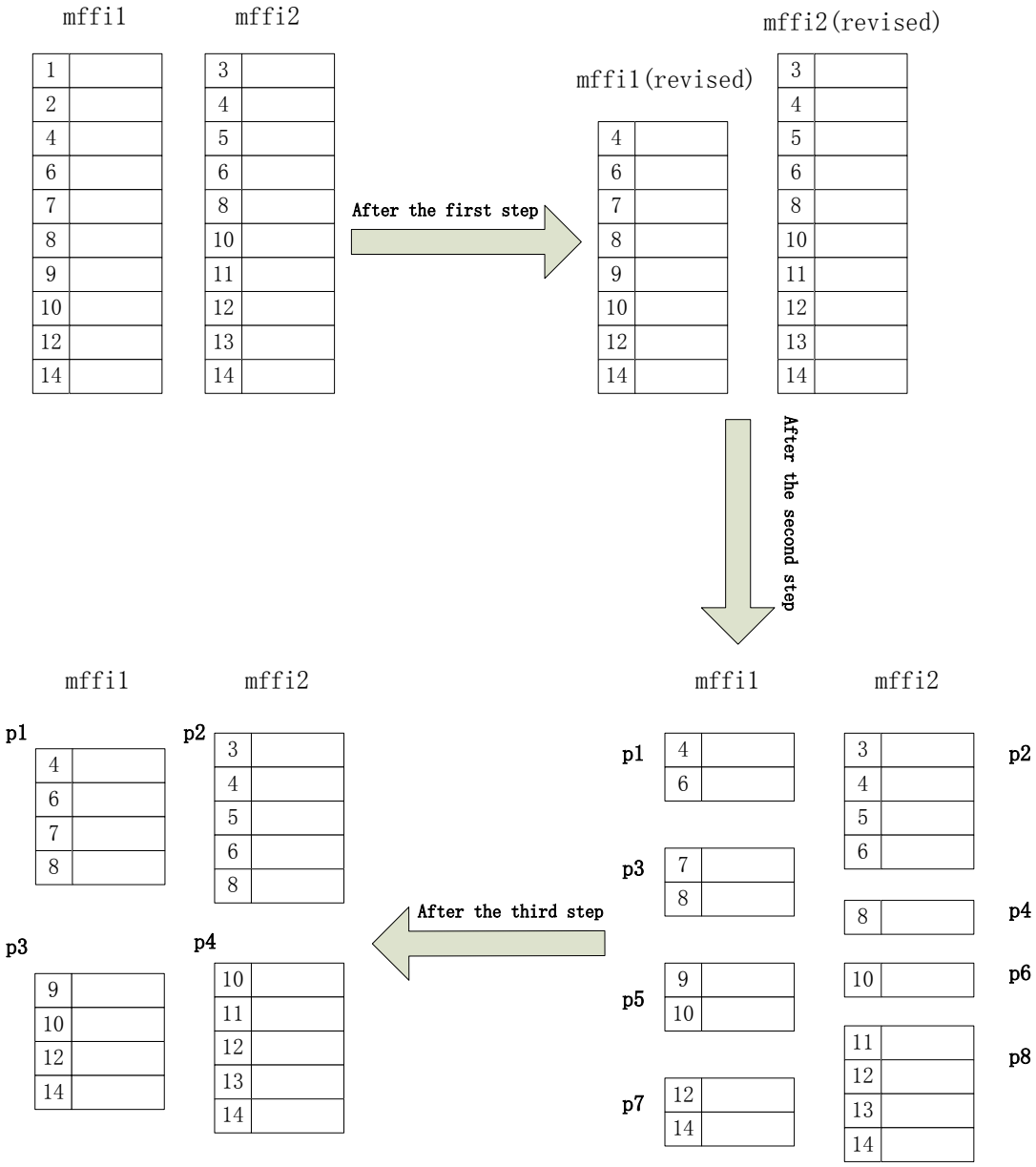

Figure 6.9: An example of the CIPM

p_6, and p_8. In the third step, assume that the accumulated (counting) numbers of ids n_1 (for p_1), n_2 (for p_2), n_3 (for p_3), n_4 (for p_4), n_5 (for p_5), n_6 (for p_6), n_7 (for p_7), n_8 (for p_8) from the ci_1 and ci_2 are: 5, 10, 20, 15, 25, 20, 35, 30, respectively. The maximum workload of a node is 40. First, the first pair of small partitions (p_1, p_2) is checked. The total number of row ids is 5+10 =15. The maximum workload is not exceeded. Thus, the second pair (p_3, p_4) is checked. The total number of row ids for the two pairs of partitions is: 5+10+(20-5)+(15-10)=35 (still smaller than 40). Next, the third pair (p_3, p_4) is checked. The total id numbers is: 5+10+(20-5)+(15-10)+(25-20)+(20-15)=45 (larger than 40). Hence, the third pair of partitions is not included and the first two pairs of partitions are combined into a pair of large partitions. The remaining pairs of small partitions are combined similarly. The final partitioning result is also shown in Fig. 6.9. Note that this example does not show a case in which a pair of partitions from the second step is too large. In such a case, we can split the partitions as described in the algorithm.

The second stage of the MFJA is called the preprocessing stage. Since $mffi_2$ is partitioned based on the given partitions of $mffi_1$ (the second step of the CIPM), some partitions of $mffi_2$ may not contain any row. After combining small partitions to large partitions (the third step of the CIPM), some partitions of $mffi_2$ may still be empty (empty partitions are combined together). In this case, we need to avoid sending those pairs of partitions, which contain an empty partition, to a map node. In addition, before actually sending a pair of partitions to a map node, some necessary preprocessing work is needed. For example, the MFFI identifiers are added in each row of two partitions and a list of key/value pairs ((row id, MFFI identifier), $\cdots$) are generated.

The third stage of the MFJA is the map stage. The input is a list of key/value pairs that are produced from the previous stage. In this stage, a merge join is applied. First, according to the MFFI identifiers, the list of key/value pairs are regrouped according to original partitions p_1 and p_2. Next, a merge join is applied on the key values of p_1 and p_2. If the key value v_1 of a row r_1 in p_1

is equal to the key value v_2 of a row r_2 in p_2, rows r_1 and r_2 are matched and joined together. After that, we utilize the structure of the MFFI to efficiently discover matched original table row ids from r_1 and r_2. We use the SFJ as an example to illustrate the processing ideas. The other types of family join operations can be processed similarly. To process the SFJ, r_1 and r_2 are extracted from the two respective MFFIs, each of which has a three-level structure. At the first level, the version number $vnumber_1$ of each version subspace in r_1 is compared with the version number $vnumber_2$ of each version subspace in r_2. If $vnumber_1$ is not equal to $vnumber_2$, no further process is needed for the two corresponding version subspaces. Otherwise, the version subspaces for $vnumber_1$ and $vnumber_2$ are accessed. At the second level, the column id cid_1 of each column subspace in the version subspace for $vnumber_1$ is compared with the column id cid_2 of each column subspace in the version subspace for $vnumber_2$. If cid_1 is not equal to cid_2, the two corresponding column subspaces are pruned. Otherwise, the column subspaces for cid_1 and cid_2 are accessed. At the third level, each original table row id rid_1 in the column subspace for cid_1 is paired with each original table row id rid_2 in the column subspace for cid_2 and returned as a matched key pair (rid_1, rid_2) in the map output. This output is used as the input key/value pair list for the next reduce stage. We can see that, using MFFIs, much unnecessary comparison work could be pruned in the map stage. The map function for the SFJ is given as follows.

ALGORITHM 6.4.4 : **The map function**
Input: (1) a list of key/value pairs (($rowkey$, $mffiid$), $\cdots$); (2) given partition pair (p_1, p_2);
Output: a list of matched key pairs (($rowid$, $rowid$), $\cdots$);
Method:
1. regroup the input key/value pairs according to partitions p_1 and p_2;
2. a = b = 1; $resultlist$ = $\emptyset$;
   /* a merge join is applied. */
3. **while** $a \leq$ row number in p_1 and $b \leq$ row number in p_2 **do**
4. r_1 = ath row in $p1$; r_2 = bth row in $p2$;
5. v_1 = key value of r_1; v_2 = key value of r_2;
6. **if** v_1 == v_2 **then**
7. **for** version number $vnumber_1$ of each version subspace in r_1 **do**
8. **for** version number $vnumber_2$ of each version subspace in r_2 **do**
9. **if** $vnumber_1$ == $vnumber_2$ **then**
10. **for** column id cid_1 of each column subspace in version subspace for $vnumber_1$ **do**
11. **for** column id cid_2 of each column subspace in version subspace for $vnumber_2$ **do**
12. **if** cid_1 == cid_2 **then**

```
13.              for each original table row id rid_1 in column subspace for cid_1 do
14.                for each original table row id rid_2 in column subspace for cid_2 do
15.                  add a key pair (rid_1, rid_2)into resultlist;
16.                end for
17.              end for
18.            end if
19.          end for
20.        end for
21.      end if
22.    end for
23.  end for
24.  a = a + 1; b = b + 1;
25. else if v_1 < v_2 then
26.    a = a + 1;
27. else
28.    b = b + 1;
29. end if
30. end while
31. return resultlist;
```

In the algorithm, according to the table (MFFI) identifiers, the input key/value pairs are re-grouped according to two partitions p_1 and p_2 (line 1). Next, a merge join is applied on the key values of rows in p_1 and p_2 (lines 3 - 30). More specifically, a key value v_1 of a row r_1 in p_1 is compared with a key value v_2 of a row r_2 in p_2 (lines 4 - 6). If v_1 is equal to v_2, three nested-loop comparisons are applied to discover matched original table row id pairs from r_1 and r_2 (lines 7 - 23). After that, the key values of the next pair of rows r_1 and r_2 are compared. If $v_1 < v_2$, the key value of the next row of r_1 is extracted and compared with v_2 (lines 25 - 26). Otherwise (i.e., $v_1 > v_2$), the key value of the next row of r_2 is extracted and compared with v_1 (lines 27 - 28).

The fourth stage of the MFJA is called the duplication removal stage. All (MapRedure) key/value pairs (i.e., matched key pairs) produced from the third stage are assembled and sorted. The duplicated key/value pairs are removed. After that, the distinct key/value pairs are assigned to multiple reduce nodes.

The last stage of the MFJA is the reduce stage. The similar reduce function that is used in the DFJM is applied. The input is a list of (MapReduce) key/value pairs that are generated from the previous stage. For each key/value pair (rid_1, rid_2), rid_1 and rid_2 are used as key values to search the corresponding rows row_1 and row_2 in original tables, and row_1 and row_2 are extracted and

joined together.

Assume that the total numbers of values of cf_1 in T_1, and cf_2 in T_2 are N_1 and N_2, respectively. The total distinct numbers of values of cf_1 in T_1, and cf_2 in T_2 are D_1 and D_2, respectively. The unit comparison cost is C (C is different for various family join types). The worst-case time complexity to apply the DFJM is: O (N_1*N_2*C) (each value of cf_1 is compared with each value of cf_2). Assume that we create two MFFIs $mffi_1$ and $mffi_2$ for cf_1 and cf_2, respectively. D_1 and D_2 represent the numbers of rows in $mffi_1$ and $mffi_2$, respectively. N_1 / D_1 indicates the average number of original row ids in a row of $mffi_1$ while N_2 / D_2 indicates the average number of original row ids in a row of $mffi_2$. Thus, (N_1*N_2*C) / (D_1*D_2) represents the cost to compare each original row id in a row of $mffi_1$ with each original row id in a row of $mffi_2$. We denote it as $RowComparisonCost$. Since $mffi_1$ and $mffi_2$ are merge joined on their row keys, the total merge join cost is: $Minimum(D_1, D_2)$* $RowComparisonCost$. Hence, the worst-case time complexity of the MFJA is: O ((N_1*N_2*C)/D_1) or O ((N_1*N_2*C)/D_2). Since D_1 and D_2 are usually larger than one, the MFJA is more efficient than the DFJM for processing the family joins.

6.5 Experiments

In this section, we report the results of our experiments to show the performance of our techniques.

6.5.1 Experiments setup

To evaluate the performance of our techniques, we conducted experiments on a six-node cluster with one node served as the master node. The cluster runs Hadoop 0.20.2. Every node in the cluster has an Intel(R) dual Core CPU with 4GB DDR-3 memory and 2TB HD attached.

We tested the join performance using two datasets, where the first one is a relatively small set of real world data, and the second one is a relatively large set of synthetic data. The real world dataset we used contains approximately 63 million distinct IP addresses and more than 100 million

different cookies. An IP address may have multiple cookies, while a cookie may be shared by multiple IP addresses. Each row of a file (Hbase table) contains an IP address as the row key and several cookies which constitute a column family (one cookie for each column).

The second dataset consists of synthetic data which was created using an Hbase table generator. The data contains the information about users and their phone numbers. The schema of the Hbase table contains a row key (user id) and a column family. The column family consists three columns, representing a home phone number, a cell phone number, and an optional phone number. A column may contain several versions of phone numbers. The dataset contains more than 500 million phone numbers from 200 million of different users.

We performed family joins on both real data and synthetic data. The real data is saved in several Hbase tables (with the same schema). These Hbase tables are grouped. Each group contains two Hbase tables with the same size (e.g., 1G). For each group, two Hbase tables are family joined on their cookie column families. Similarly, the synthetic data is preprocessed in a similar way. Two synthetic Hbase tables are family joined on their phone number column families.

We implemented the DFJM and the MFJA using MapReduce on Hadoop. The MFFIs were created for the column families of Hbase tables on which the family join was performed. While an MFFI was being generated, a corresponding CI was also created. The DFJM directly processed the family joins on the original tables. The task was accomplished in parallel using MapReduce. Finally, the join result was assembled and saved. On the another hand, the MFJA processed the family joins using two MFFIs instead. The MFFIs were partitioned by using the CIPM (the counting index base partitioning method) and sent to different map nodes for computation. After that, the reduce nodes fetched rows from the original tables and finished the join task. In the experiments, each family join was performed 10 times and the average performance was measured.

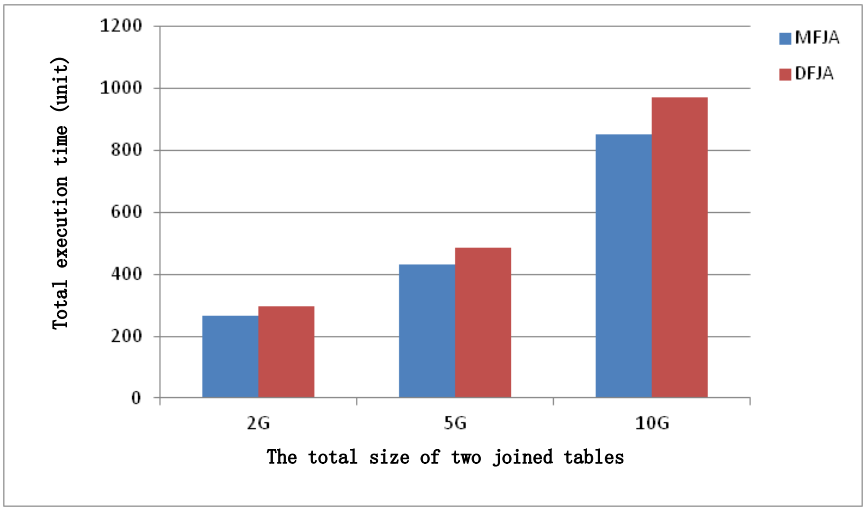

Figure 6.10: Performance comparisons between DFJA and MFJA using SFJ

6.5.2 Performance of the DFJM and the MFJA

The first experiment was conducted to evaluate the efficiency of our MFFI based family join approach. As mentioned earlier, we had several groups (different sizes) of Hbase tables which contained real data. For each group, two Hbase tables were joined on their cookie column families by using both the DFJM and the MFJA. Fig.6.10 shows the performance comparisons between the DFJM and the MFJA on the real data. Note that we use the SFJ as our default family join type. The x-axis represents the total size of two joined Hbase tables, and y-axis represents the total execution time of the join work. From the figure, we observe that the MFJA always outperforms the DFJM for different Hbase table sizes.

6.5.3 Performance of the MFJA for different family join types

The second experiment was conducted to evaluate the performance of the MFJA for different types of family joins. We mentioned that the family joins could be done with different freedoms.

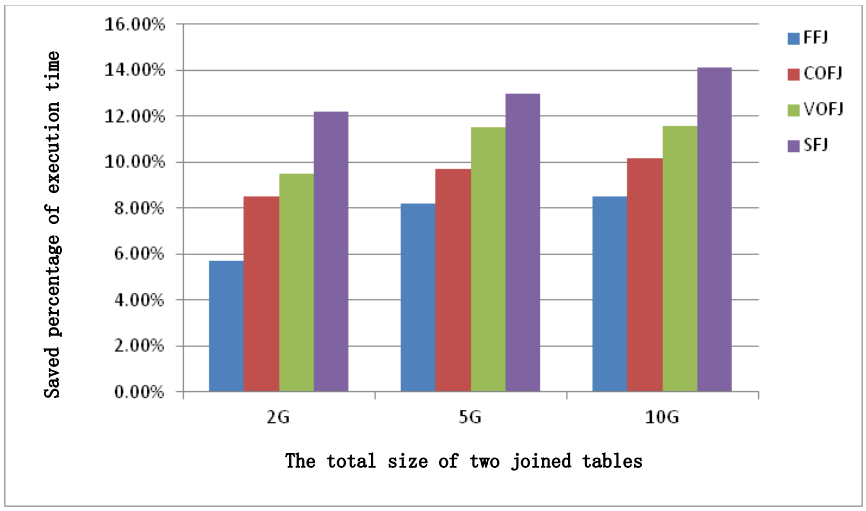

Figure 6.11: Saved percentage of execution time by using the MFJA on the real world data for different types of family joins

According to different matching criterions, there were four types of family joins: the free family join (FFJ), the column-oriented family join (COFJ), the version-oriented family join (VOFJ), and the strict family join (SFJ). Since the MFJA performs always better than the DFJA, we want to know that for different types of family joins, using the MFJA, how much percentage of execution time can be saved compared to that by using the DFJA. Fig. 6.11 shows the saved percentage of execution time by using the MFJA on the real world data for different types of family joins and different sizes of Hbase tables. From the figure, we can see that the SFJ always saved much more percentage of execution time than others. The VOFJ performs better than the COFJ and the FFJ saved least percentage of execution time. The reason for that is as follows. As we mentioned earlier, the MFFI has a three-level structure in each row. The first level is the version level and the second level is the column level. For the SFJ, the version level and the column level work together to maximally prune unnecessary comparisons. For the VOFJ, only the version level help to prune

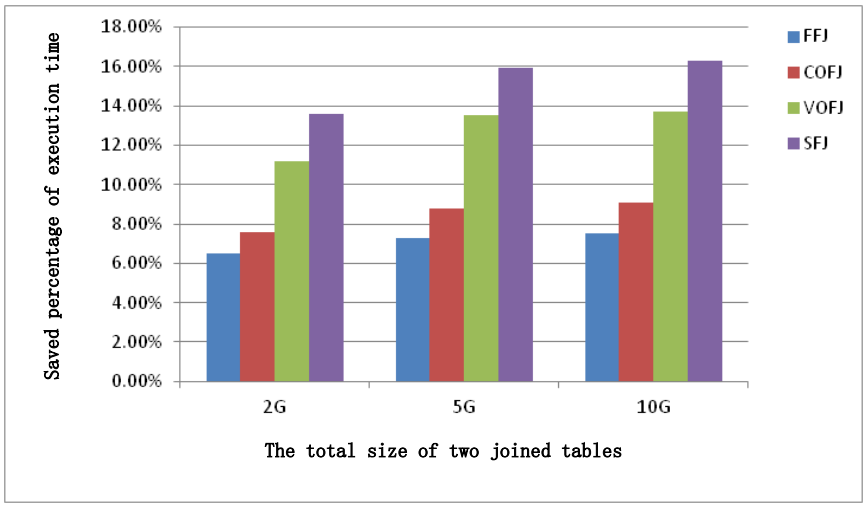

Figure 6.12: Saved percentage of execution time by using the MFJA on the synthetic data for different types of family joins

unusable comparisons while for the COFJ, only the column level contribute to prune unnecessary comparisons. Thus, the VOFJ and the COFJ performs less than the SFJ. Since in our design, the column level is below the version level and the version numbers is usually smaller than the column numbers, which means that there is more chance for using the version level to prune more unusable comparisons. As a result, the VOFJ saved more percentage of execution time than that saved by the COFJ. For the FFJ, the three-level structure is not used to prune comparisons. Hence, the saved percentage of execution time by the FFJ is less than others. Fig. 6.12 shows the saved percentage of execution time by using the MFJA on the synthetic data for different types of family joins and different sizes of Hbase tables. The similar results can be observed.

6.5.4 Performance of the MFJA with different partitioning methods

The third experiment was conducted to examine the performance of the MFJA with different partitioning methods. As we mentioned, the MFFIs have to be partitioned in a reasonable way for the parallelization purpose. A straightforward partitioning method (SPM) and a counting index based partitioning method (CIPM) were discussed in section 6.4. The actual workload (i.e., the number of rows) assigned for each map node is compared between the CIPM and the SPM in Fig. 6.13. We observe that a more balanced workload distribution is achieved by the CIPM. Fig. 6.14 shows the performance between the MFJA with the CIPM and the one with the SPM. Note that the SFJ is used as our default family join type. From the figure, we can see that, as the total size of two joined Hbase tables increases, the CIPM performs increasingly better. The reason for that is as follows. If the total size of two joined Hbase tables is small, although the CIPM can better balance the workload for each map node, the performance improvement is not obvious. But if total size of two joined tables is larger, using the SPM, a map node may be assigned with all of the work while the work is evenly divided among all map nodes by using the CIPM. As a result, the performance of the CIPM is increasingly improved.

6.5.5 Effect of the duplication rates

The last experiment was to evaluate the effect of a parameter of the dataset, which is called the duplication rate (dr), on both the DFJM and the MFJA. The duplication rate is defined as: the number of distinct values / the number of all values. In the synthetic dataset, we can control the distinct values of the dataset by setting the dr before generating the data. A larger dr implies a larger number of distinct values in the dataset. Fig. 6.15 shows the performance of the DFJM and the MFJA on the same maximum workload capacity in the system (20G) with different duplication rates. From the figure, we observe that, as the duplication rate increases, the performance of the

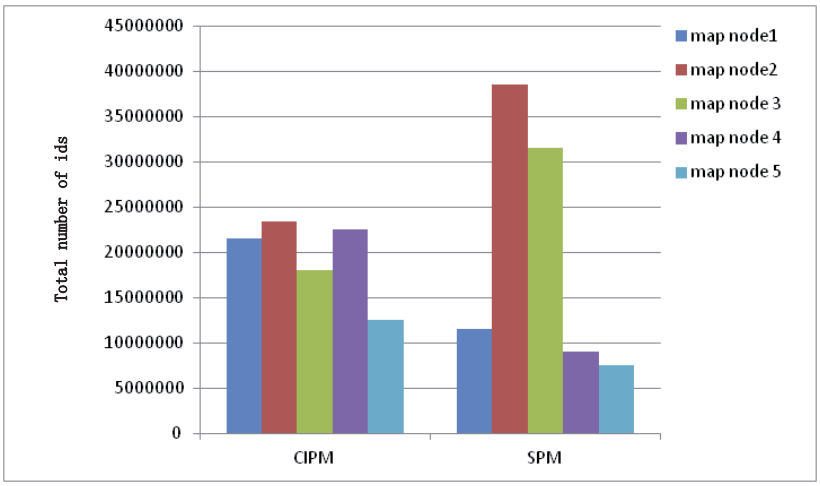

Figure 6.13: Performance comparisons of the CIPM and the SPM

Figure 6.14: Node balance comparisons between the CIPM and the SPM

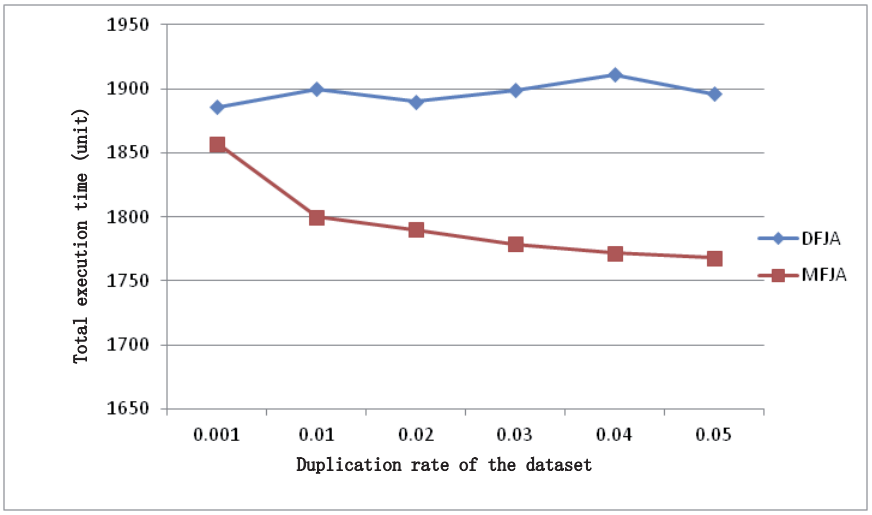

Figure 6.15: Performance comparisons between the MFJA and the DFJM with different drs

MFJA becomes increasingly better, while the performance of the DFJM is relatively stable. The reason for that is as follows: as we mentioned in section 6.4, the worst-case time complexity for the MFJA is O $((N_1{*}N_2{*}C)/D_1)$ or O $((N_1{*}N_2{*}C)/D_2)$. Thus, as the dr increases, D_1 and D_2 become larger, and the worst-case time complexity of the MFJA is reduced. As a result, the performance of the MFJA is improved.

Our experiments demonstrate that implementing various types of family joins in a big data environment is feasible and utilizing proper strategies such as indexing and intelligent partitioning can improve the performance of a family join.

CHAPTER 7
Conclusion

There is an increasing demand for progressive queries (PQ) from numerous application domains. How to efficiently process PQs raises new challenges, such as the access inefficiency, the query unpredictability, and the result management challenge. Existing DBMSs were not designed to efficiently process such queries. To tackle some of the challenges, we have investigated the issues and methods for efficiently processing PQs via materialized views in this work. A suite of new techniques have been proposed after carefully analyzing the relevant problems.

The main contributions of this dissertation are summarized as follows:

- We have developed a novel dynamic materialized-view based approach for efficiently processing the monotonic linear PQs. A so-call superior relationship graph to capture the superior relationships among SQs of historical (monotonic linear) PQs was defined. A method for using the graph to estimate the benefit of keeping the result of an SQ as a materialized view to improve the processing efficiency of future SQs was suggested. Based on the properties of monotonic linear PQs, a generating-based algorithm (which is more efficient for a dense graph) and a pruning-based algorithm (which is more suitable for a sparse graph) were proposed to efficiently construct a superior relationship graph. A special structure and its relevant algorithms to effectively manage the materialized views so as to improve the view searching efficiency and quality were also presented. A system architecture to incorporate the proposed technique to process PQs was discussed. Extensive experiments were conducted to evaluate the performance of the adopted strategies with various parameters. Our theoretical and empirical studies have demonstrated that the proposed technique is quite promising in efficiently processing monotonic linear PQs.

- We have proposed a new materialized-view based approach for efficiently processing generic PQs. A so-called multiple query dependency graph (MQDG) was constructed to capture the data source dependency relationships among the SQs and external tables of generic PQs. A mathematical model using this graph to estimate the potential benefit of selecting an SQ of a completed PQ for view materialization was derived. Various factors affecting the impact and effectiveness of view materialization were considered. A method to select materialized views via dependency analysis was suggested. An algorithm for dynamically removing useless nodes and transferring dependency relationships for the MQDG during the query processing was introduced. To maintain the materialized view space, we studied both a direct greedy method and a dynamic programming method. The former is more efficient, while the latter guarantees a better solution. To mitigate the high worst-case complexity issue for the dynamic programming procedure, we suggested a greedy strategy to reduce the problem input size. Our extensive experimental results demonstrated that the proposed technique significantly improves the processing efficiency for generic PQs.

- We have introduced a new dynamic materialized view index method to efficiently discover usable materialized views from a given view space/set for answering SQs of generic PQs. A special index tree structure was designed to cope with the highly dynamic nature of the materialized view space for PQs. In particular, a two-level priority rule was adopted to order the tree nodes so that different types of views can be quickly located in the tree. Furthermore, a bitmap encoding and matching mechanism was applied to refine the pruning power of unusable views during a search. The algorithms and strategies for the index tree construction, search, and maintenance were presented. Our experimental results showed that the proposed index method and relevant strategies are effective in searching usable views for answering SQs.

- We have developed an index based technique for performing a new column family join operation on Hbase tables in a popular big data environment. The useful column family join operation with four freedom variations was introduced. Two family join approaches (a direct method and an index based method) were discussed. To efficiently process a column family join, we suggested using a so-called multiple freedom family index (MFFI). An MFFI uses the cell values from the indexed column family in the underlying Hbase table as the main search key values, and the space for each key value was further divided into a number of subspaces based on the version numbers and the column ids of the underlying Hbase table. An MFFI itself was implemented as an Hbase table in order take advantages of the supported features for Hbase in the Hadoop. A parallel MapReduce algorithm for building the MFFI was presented. An MFFI based parallel MapReduce algorithm for processing a column family join was proposed. In particular, an auxiliary counting index was used in the algorithm to achieve a balanced workload among the map nodes. Our experiments demonstrated that the suggested technique for processing column family joins was efficient. This technique enable the big data environment to possess the closure property for query processing, which is an essential requirement for supporting PQs.

The work reported in this dissertation only represents the beginning of research effort in addressing the relevant issues. In particular, our research on supporting PQs in a big data environment is quite preliminary. More research needs to be done in order to completely solve this problem. Our future work includes: extend our techniques to handle more types of SQs (e.g., those involving aggregate functions); study how to process approximate PQs; investigate methods to deal with spatio-temporal PQs; develop more efficient access methods (e.g., hash based) to support PQs; explore issues for incorporating our techniques in a real DBMS; build a materialized view mechanism for supporting PQs in the Hadoop environment; develop techniques to efficiently process

PQs via materialized views in the Hadoop environment; and study methods to handle PQs in other types of databases such as XML.

In summary, the problem of efficiently processing PQs is challenging; the materialized-view based techniques proposed in this dissertation are promising; and many interesting research issues remain to be resolved in the future.

Bibliography

[1] Afrati, Foto N., Sarma, Anish Das., *et al.*(2012) Fuzzy Joins Using MapReduce. *Proc. of ICDE Conf.*, pp. 498-509.

[2] Afrati, Foto N. and Ullman, Jeffrey D.(2010) Optimizing joins in a map-reduce environment. *Proc. of EDBT Conf.*, pp. 99-110.

[3] Agarwal, P. K., Xie, J., Yang, J. and Yu, H. (2006) Scalable continuous query processing by tracking hotspots. *Proc. of VLDB Conf.*, pp. 31-42, ACM.

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

[5] Ali, Akhtar., M., *et al.* (2003) MOVIE: An incremental maintenance system for materialized object views. *Data Knowl. Eng. (DKE)*, 47(2): 131-166.

[6] Aly, Ahmed M., Sallam, Asmaa., *et al.* (2012) M3: Stream Processing on Main-Memory MapReduce. *Proc. of ICDE Conf.*, pp. 1253-1256.

[7] Antoshenkov, G. (1993) Dynamic query optimization in RDB/VMS. *Proc. of ICDE Conf.*, pp. 538-547, IEEE Computer Society.

[8] Aouiche, K., Jouve, P.-E., *et al.* (2006) Clustering-Based Materialized View Selection in Data Warehouses. *Proc. of ADBIS Conf.*, pp. 81-95, Springer.

[9] Aouiche, K. and Darmont, J. (2009) Data mining-based materialized view and index selection in data warehouses. *J. Intell. Inf. Syst.*, 33(1): 65-93.

[10] Arion, A., Benzaken, V., *et al.* (2007) Structured Materialized Views for XML Queries. *Proc. of VLDB Conf.*, pp. 87-98, ACM.

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

[12] Babu, S. and Bizarro, P. (2005) Adaptive query processing in the looking glass. *Proc. of CIDR Conf.*, Asilomar, CA, 4-7 January, pp. 238-249, online proceedings, http: //www-db.cs.wisc.edu/cidr/.

[13] Bahmani, Bahman., Chakrabarti, Kaushik., *et al.* (2011) Fast personalized PageRank on MapReduce. *Proc. of SIGMOD Conf.*, pp. 973-984.

[14] Blakeley, J. A., Larson, P.-Å., *et al.* (1986) Efficiently updating materialized views. *Proc. of SIGMOD Conf.*, pp. 61-71.

[15] Balmin, A., Beyer, K.-S., *et al.* (1997) A Framework for Using Materialized XPath Views in XML Query Processing. *Proc. of VLDB Conf.*, pp. 136-145.

[16] Baralis, E., Paraboschi, S., *et al.* (1998) Materialized View Selection for Multidimensional Datasets. *Proc. of VLDB Conf.*, pp. 488-499.

[17] Bayer, R. and McCreight, E. (1972) Organization and maintenance of large ordered indexes. *Acta Informatica 7, 3*, pp. 173-189.

[18] Bellatreche, L., Karlapalem, K., *et al.* (2000) Evaluation of Materialized View Indexing in Data Warehousing Environments. *Proc. of DaWaK*, pp. 57-66, Springer.

[19] Berchtold, S., Keim, D. A., *et al.* (1996) The X-tree: An Index Structure for High-Dimensional Data. *Proc.of VLDB Conf.*, pp. 28-39.

[20] Chaudhuri, S., Krishnamurthy, R., Potamianos, S., *et al.* (1995) Optimizing queries with materialized views. *Proc. of ICDE Conf.*, pp. 190-200, IEEE Computer Society.

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

[22] Chan, C. Y., Garofalakis, M. N., *et al.* (2002) RE-Tree: An Efficient Index Structure for Regular Expressions. *Proc.of VLDB Conf.*, pp. 263-274.

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

[24] Chen, C. L. P. and C.-Y. Zhang. (2014) Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences*, 275: 314-347.

[25] Chirkova, R., Li, C. and Li, J. (2006) Answering queries using materialized views with minimum size. *VLDB Journal*, 15(3): 191-210.

[26] Condie, Tyson., Conway, Neil., *et al.* (2010) Online aggregation and continuous query support in MapReduce. *Proc. of ICDE Conf.*, pp. 1115-1118.

[27] Dittrich, Jens. and Quiane-Ruiz, Jorge-Arnulfo. (2012) Efficient Big Data Processing in Hadoop MapReduce. *Proc. of VLDB Conf.*, 5(12): 2014-2015.

[28] Elghandour, Iman. and Aboulnaga, Ashraf. (2012) ReStore: reusing results of MapReduce jobs in pig. *Proc. of SIGMOD Conf.*, pp. 701-704.

[29] Ezeife, C.I. (2001) Selecting and materializing horizontally partitioned warehouse views. *Data and Knowledge Engineering*, 36(2): 185-210.

[30] Folkert, N., Gupta, A., *et al.* (2005) Optimizing Refresh of a Set of Materialized Views. *Proc. of VLDB Conf.*, pp. 1043-1054.

[31] Fusco, F., Vlachos, M., *et al.* (2012) Real-time creation of bitmap indexes on streaming network data. *The VLDB Journal*, 21(3): 287-307.

[32] Ghoting, Amol., Krishnamurthy, Rajasekar., et al.(2011) SystemML: Declarative machine learning on MapReduce. *Proc. of ICDE Conf.*, pp. 231-242.

[33] Goldstein, J. and Larson, P-Å. (2001) Optimizing Queries Using Materialized Views: A practical, scalable solution. *Proc. of SIGMOD Conf.*, pp. 331-342, ACM.

[34] Gou, G., Kormilitsin, M., *et al.* (2006) Query evaluation using overlapping views: completeness and efficiency. *Proc. of SIGMOD Conf.*, pp. 37-48, ACM.

[35] Graefe, G. and Zwilling, M. J. (2004) Transaction support for indexed summary views. *Proc.of SIGMOD Conf.*, pp. 323-334.

[36] Gray, J. and Szalay, A. S. (2004) Where the rubber meets the sky: bridging the gap between databases and science. *IEEE Data Eng. Bull.*, 27(4): 3-11.

[37] Gupta, A. and Mumick, I. S. (2005) Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 17(1): 24-43.

[38] Gupta, A., Mumick, I.S., *et al.* (1995) Adapting Materialized Views after Redefinitions. *Proc. of SIGMOD Conf.*, pp. 211-222.

[39] Gupta, A. and Mumick, I. S. (1995) Maintenance of materialized views: problems, techniques, and applications. *Data Engineering Bulletin*,18(2): 3-18.

[40] Gupta, A., Mumick, I. S., *et al.* (1993) Maintaining views incrementally. *Proc. of SIGMOD Conf.*, Washington, D.C., 26-28 May, pp. 157-166, ACM.

[41] Gupta, Himanshu., Chawda, Bhupesh., *et al.* (2013) Processing multi-way spatial joins on map-reduce. *Proc. of EDBT Conf.*, pp. 113-124.

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

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

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

[45] He, B., Hsiao, H., *et al*. (2012) Efficient Iceberg Query Evaluation Using Compressed Bitmap Index. *IEEE Trans. Knowl. Data Eng.*, 24(9): 1570-1583.

[46] He, Yongqiang., Lee, Rubao., *et al.* (2011) RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. *Proc. of ICDE Conf.*, pp. 1199-1208.

[47] Hung, M.-C., Huang, M.-L., *et al.* (2007) Efficient approaches for materialized views selection in a data warehouse. *Inf. Sci. (ISCI)*, 177(6): 1333-1348.

[48] Jahani, Eaman., Cafarella, Michael J., et al.(2011) Automatic Optimization for MapReduce Programs. *Proc. of VLDB Conf.*, 4(6): 385-396.

[49] Jang, J.S.R. and 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.

[50] Jiang, Dawei., Ooi, Beng. Chin., et al.(2010) The Performance of MapReduce: An In-depth Study. *Proc. of VLDB Conf.*, 3(1): 472-483.

[51] Jiang, H., Gao, D., *et al.* (2008) Exploiting Correlation and Parallelism of Materialized-View Recommendation for Distributed Data Warehouses. *Proc. of ICDE Conf.*, pp. 276-285.

[52] Jiang, H., Lu, H., *et al.* (2003) XR-Tree: Indexing XML Data for Efficient Structural Joins. *Proc.of ICDE Conf.*, pp. 253-264.

[53] Jorg, T. and DeBloch, S. (2008) Towards Generating ETL Processes for Incremental Loading. *Proc. of IDEAS'08*, pp. 101-110.

[54] Joshi, S. and Jermaine, C. M. (2008) Materialized Sample Views for Database Approximation. *IEEE Trans. Knowl. Data Eng. (TKDE)*, pp. 151.

[55] Kabra, N. and DeWitt, D. J. (1998) Efficient mid-query re-optimization of sub-optimal query execution plans. *Proc. of SIGMOD*, pp. 106-117, ACM.

[56] Kache, H., Han, W.S., *et al.* (2006) POP/FED: Progressive Query Optimization for Federated Queries in DB2. *Proc. of VLDB Conf.*, pp. 1175-1178.

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

[58] Khoussainova, Nodira., Balazinska, Magdalena., *et al.* (2012) PerfXplain: Debugging MapReduce Job Performance. *Proc. of VLDB Conf.*, 5(7): 598-609.

[59] Kim, Younghoon. and Shim, Kyuseok. (2012) Parallel Top-K Similarity Join Algorithms Using MapReduce. *Proc. of ICDE Conf.*, pp. 510-521.

[60] Kimura, H., Guo, G., *et al.* (2010) CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. *PVLDB*, 3(1): 1103-1113.

[61] Kolb, Lars., Thor, Andreas. *et al.* (2012) Load Balancing for MapReduce-based Entity Resolution. *Proc. of ICDE Conf.*, pp. 618-629.

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

[63] Kuo, T. W., Wei, C. H., *et al.* (1999) Real-Time Data Access Control on B-Tree Index Structures. *Proc.of ICDE Conf.*, pp. 458-467.

[64] Lam, Wang., Liu, Lu., *et al.* (2012) Muppet: MapReduce-Style Processing of Fast Data. *Proc. of VLDB Conf.*, 5(12): 1814-1825.

[65] Laptev, Nikolay., Zeng, Kai., *et al.* (2012) Early Accurate Results for Advanced Analytics on MapReduce. *Proc. of VLDB Conf.*, 5(10): 1028-1039.

[66] Larson, P.-Å. and Yang, H. Z.(1985) Computing queries from derived relations. *Proc. of VLDB Conf.*, pp. 259-269, ACM.

[67] Larson, P.-Å. and Zhou, J. (2007) View matching for outer-join views. *VLDB Journal*, 16(1): 29-53.

[68] Lee, M. and Hammer, J. (2001) Speeding Up Materialized View Selection in Data Warehouses Using a Randomized Algorithm. *Int. J. Coop. Inf. Syst. (IJCIS)*, 10(3): 327-353.

[69] Liang, W., Wang, H. *et al.* (2001) Materialized view selection under the maintenance time constraint. *Data Knowl. Eng. (DKE)*, 37(2): 203-216.

[70] Lim, H.-S., Lee, L.-G., *et al.* (2006) Continuous query processing in data streams using duality of data and queries. *Proc. of SIGMOD Conf.*, pp. 313-324, ACM.

[71] Lim, Harold., Herodotou, Herodotos., *et al.* (2012) Stubby: A Transformation-based Optimizer for MapReduce Workflows. *Proc. of VLDB Conf.*, 5(11): 1196-1207.

[72] Lin, Yuting., Agrawal, Divyakant., *et al.* (2011) Llama: leveraging columnar storage for scalable join processing in the MapReduce framework. *Proc. of SIGMOD Conf.*, pp. 961-972.

[73] Liu, Xiufeng., Thomsen, Christian., *et al.* (2012) MapReduce-based Dimensional ETL Made Easy. *Proc. of VLDB Conf.*, 5(12): 1882-1885.

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

[75] Liu, L. and Pu, C. (1997) Dynamic query processing in DIOM *IEEE Data Eng. Bull*, 20(3): 30-37.

[76] Lu, H., Tan, K.-L. *et al.* (1995) The Fittest Survives: An Adaptive Approach to Query Optimization. *Proc. of VLDB Conf.*, pp. 251-262, ACM.

[77] Lu, Wei., Shen, Yanyan., *et al.* (2012) Efficient Processing of k Nearest Neighbor Joins using MapReduce. *Proc. of VLDB Conf.*, 5(10): 1016-1027.

[78] Lu, M.-C. and Wu, F. (2004) A Structure for Materialized Views of Data Warehouse with Concurrency Control. *Proc. of IKE Conf.*, pp. 385-391.

[79] Luo, G., Naughton, J. F., *et al.* (2005) Locking Protocols for Materialized Aggregate Join Views. *IEEE Trans. Knowl. Data Eng. (TKDE)*, 17(6): 796-807.

[80] Luo, G. (2007) Partial Materialized Views. *Proc. of ICDE Conf.*, pp. 756-765.

[81] Metwally, Ahmed. and Faloutsos, Christos.(2012) V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *Proc. of VLDB Conf.*, 5(8): 704-715.

[82] Mistry, H., Roy, P., *et al.* (2001) Materialized View Selection and Maintenance Using Multi-Query Optimization. *Proc. of SIGMOD*, pp. 307-318, ACM.

[83] Mokbel, M. F. Continuous query processing in spatio-temporal databases. *Proc. of EDBT Workshops*, pp. 100-111, Springer.

[84] Nambiar, U., Lud, B., *et al.* (2006) The GEON portal: accelerating knowledge discovery in the geosciences. *Proc.of ACM International Workshop on Web Information and Data Management (WIDM)*, pp. 83-90, ACM.

[85] NCBI. (2014) GenBank and WGS statistics. *http://www.ncbi.nlm.nih.gov/genbank/statistics*, U.S. National Library of Medicine.

[86] Nieto-Santisteban, M. A., Gray, J., *et al.* (2005) When database systems meet the grid. *Proc.of CIDR Conf.*, pp. 154-161, ACM.

[87] Nitsos, I., Evangelidis, G., *et al.* (2004) Bitmap-Tree Indexing for Set Operations on Free Text. *Proc.of ICDE Conf.*, pp. 837.

[88] Okcan, Alper. and Riedewald, Mirek.(2011) Processing theta-joins using MapReduce. *Proc. of SIGMOD Conf.*, pp. 949-960.

[89] Pansare, Niketan., Borkar, Vinayak R., *et al.* (2011) Online Aggregation for Large MapReduce Jobs. *Proc. of VLDB Conf.*, 4(11): 1135-1145.

[90] Papadias, D., Tao, Y., *et al.* (2003) An optimal and progressive algorithm for skyline queries *Proc. of SIGMOD Conf.*, pp. 467-478.

[91] Park, C.-S., Kim, M.-H., *et al.* (2001) Rewriting OLAP Queries Using Materialized Views and Dimension Hierarchies in Data Warehouses. *Proc. of ICDE Conf.*, pp. 515-523, IEEE Computer Society.

[92] Patrick, O'Neil. (1987) Model 204 Architecture end Performance. *Springer-Verlag Lecture Notes in Computer Science 359, 2nd International Workshop on High Performance Transactions Systems*.

[93] Phan, T. and Li, W. S. (2008) Dynamic Materialization of Query Views for Data Warehouse Workloads. *Proc. of ICDE Conf.*, pp. 436-445.

[94] Quiane-Ruiz, Jorge-Arnulfo., Pinkel, Christoph. *et al.* (2011) RAFTing MapReduce: Fast recovery on the RAFT. *Proc. of ICDE Conf.*, pp. 589-600.

[95] Raghavan, V. and Rundensteiner, E.A. (2010) Progressive result generation for multi-criteria decision support queries. *Proc. of ICDE Conf.*, pp. 733-744.

[96] Re, C. and Suciu, D. (2007) Materialized Views in Probabilistic Databases for Information Exchange and Query Optimization. *Proc. of VLDB Conf.*, pp. 51-62, ACM.

[97] Vernica, Rares., Carey, Michael J., *et al.* (2010) Efficient parallel set-similarity joins using MapReduce. *Proc. of SIGMOD Conf.*, pp. 495-506.

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

[99] Roussopolous, N. (1982) View indexing in relational databases. *ACM Trans. on Database Systems*, 7(2): 258-290.

[100] Sakurai, Y., Yoshikawa, M., *et al.* (2000) The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. *Proc.of VLDB Conf.*, pp. 516-526.

[101] Shim, Kyuseok. (2012) MapReduce Algorithms for Big Data Analysis. *Proc. of VLDB Conf.*, 5(12): 2016-2017.

[102] Silva, Yasin N. and Reed, Jason M. (2012) Exploiting MapReduce-based similarity joins. *Proc. of SIGMOD Conf.*, pp. 693-696.

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

[104] Sinha, R. R., Winslett, M., *et al.* (2008) Adaptive Bitmap Indexes for Space-Constrained Systems. *Proc.of ICDE Conf.*, pp. 1418-1420.

[105] Sinha, H. H. and Winslett, M. (2010) Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.*, 32(3): 16.

[106] Srivastava, D., Dar, S., *et al.* (1996) Answering queries with aggregation. *Proc.of VLDB Conf.*, pp. 318-329.

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

[108] Talebi, Z. A., Chirkova, R., *et al.* (2008) Exact and inexact methods for selecting views and indexes for OLAP performance improvement. *Proc. of EDBT Conf.*, pp. 311-322.

[109] Tang, N., Yu, J. X., *et al.* Multiple Materialized View Selection for XPath Query Rewriting. *Proc.of ICDE Conf.*, pp. 873-882, IEEE.

[110] Tao, Yufei., Lin, Wenqing., *et al.* (2013) Minimal MapReduce algorithms. *Proc. of SIGMOD Conf.*, pp. 529-540.

[111] Theodoratos, D. and Sellis, T. (1997) Data warehouse configuration. *Proc.of VLDB Conf.*, pp. 126-135.

[112] Tiakas, E., Papadopoulos, A.-N. *et al.* (2011) Progressive processing of subspace dominating queries *VLDB Journal*, 20(6): 921-948.

[113] Vassiliadis, P., Vagena, Z., *et al.* (2001) ARKTOS: towards the modeling, design, control and execution of ETL processes. *Info. Sys*, 26(8): 537-561.

[114] Vassiliadis, P., Simitsis, A., *et al.* (2005) A generic and customizable framework for the design of ETL scenarios. *Info. Sys.*, 30(7): 492-525.

[115] Wang, H., Park, S., *et al.* (2003) ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. *Proc.of SIGMOND Conf.*, pp. 110-121.

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

[117] Wu, K., Shoshani, A., *et al.* (2010) Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst.*, 35(1): 2.

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

[119] Yang, H. Z. and Larson, P.-A. (1987) Query transformation for PSJ-queries. *Proc.of VLDB Conf.*, pp. 245-254.

[120] Yang, Hung chih., Dasdan, Ali., *et al*. (2007) Map-reduce-merge: simplified relational data processing on large clusters. *Proc. of SIGNOD Conf.*, pp. 1029-1040.

[121] Yang, J., Karlapalem, K., *et al.* (1997) Algorithms for Materialized View Design in Data Warehousing Environment. *Proc. of VLDB Conf.*, pp. 136-145.

[122] Yoon, Jong P. (2006) Presto Authorization: A Bitmap Indexing Scheme for High-Speed Access Control to XML Documents. *IEEE Trans. Knowl. Data Eng.*, 18(7): 971-987.

[123] Zhang, Chi., Li, Feifei., et al.(2012) Efficient parallel kNN joins for large data in MapReduce. *Proc. of EDBT Conf.*, pp. 38-49, 2012.

[124] Zhang, Xiaofei., Chen, Lei., *et al.* (2012) Efficient Multi-way Theta-Join Processing Using MapReduce. *Proc. of VLDB Conf.*, 5(11): 1184-1195.

[125] Zhang, Z., Hadjieleftheriou, M., *et al.* (2010) Bed-tree: an all-purpose index structure for string similarity search based on edit distance. *Proc.of SIGMOND Conf.*, pp. 915-926.

[126] Zhou, J., Larson, P.-Å., *et al.* (2007) Lazy Maintenance of Materialized Views. *Proc. of VLDB Conf.*, pp. 231-242, ACM.

[127] Zhou, J., Larson, P., *et al.* (2007) Dynamic Materialized Views. *Proc. of ICDE Conf.*, pp. 526-535.

[128] Zhu, C., Zhu, Q., *et al.* (2010) Efficient processing of monotonic linear progressive queries via dynamic materialized views. *Proc. of CASCON Conf.*, pp. 224-237, ACM.

[129] Zhu, C., Zhu, Q., *et al.* (2011) A Materialized-View Based Technique to Optimize Progressive Queries via Dependency Analysis. *Proc. of CASCON Conf.*, pp. 60-73, ACM.

[130] Zhu, C., Zhu, Q., *et al.* (2014) Optimization of Monotonic Linear Progressive Queries Based on Dynamic Materialized Views. *The Computer Journal*, 57(5): 708-730.

[131] Zhu, C., Zhu, Q., *et al.* (2012) DMVI: A Dynamic Materialized View Index for Efficiently Discovering Usable Views for Progressive Queries. *Proc.of CASCON Conf.*, pp. 42-56, ACM.

[132] Zhu, C., Zhu, Q., *et al.* (2013) Developing a Dynamic Materialized View Index for Efficiently Discovering Usable Views for Progressive Queries. *Journal of Information Processing Systems*, 9(4): 511-537.

[133] Zhu, C., Zhu, Q., *et al.* (2014) Optimization of generic progressive queries based on dependency analysis and materialized views. *Information Systems Frontiers*, DOI:10.1007/s10796-014-9517-2, to appear.

[134] Zhu, Q., Medjahed, B., *et al.* (2008) The Collective index: A Technique for Efficient Processing of Progressive Queries. *The Computer Journal*, 51(6): 662-676.

[135] Zilio, D., Zuzarte, C., *et al.* (2004) Recommending materialized views and indexes with IBM DB2 design advisor. *Proc.of ICAC*, pp. 180-188, IEEE Computer Society.