PARTIAL AGGREGATION AND QUERY PROCESSING OF OLAP CUBES

by

Zhenshan Guo
B.E., Changchun Institute of Post and Telecommunications, 1995

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In the
School
of
Computing Science

© Zhenshan Guo 2009

SIMON FRASER UNIVERSITY

Spring 2009

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.

APPROVAL

Name: **Zhenshan Guo**

Degree: **Master of Science**

Title of Thesis: **Partial Aggregation and Query Processing of OLAP Cubes**

Examining Committee:

Chair: **Dr. Jiangchuan Liu**
Assistant Professor of Computing Science

____

Dr. Wo-shun Luk
Senior Supervisor
Professor of Computing Science

____

Dr. Ke Wang
Supervisor
Professor of Computing Science

____

Dr. Jian Pei
Internal Examiner
Associate Professor of Computing Science

Date Defended/Approved: October 17, 2008

SIMON FRASER UNIVERSITY
LIBRARY

Declaration of Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website <www.lib.sfu.ca> at: <http://ir.lib.sfu.ca/handle/1892/112>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada

Last revision: Spring 09

ABSTRACT

This work presents a novel and flexible PPA (Partial Pre-Aggregation) construction and query processing technique in OLAP (On-Line Analytical Processing) applications - SplitCube, which greatly reduces the cube size, shortens the cube building time and maintains an acceptable query performance at the time. Furthermore, we devise two enhanced query processing techniques. They can further improve the query performance or reduce cube building time further and keep query response time at an acceptable level. The result analysis shows more insights in cube construction and query processing procedure and illustrates the advantages and disadvantages of each algorithm. Finally, we give guidelines in how to choose the right algorithm in different user cases.

Keywords: PPA, OLAP, SplitCube, MQPSB, QPRQ

ACKNOWLEDGEMENTS

I am deeply indebted to my advisor and supervisor, Professor Wo-shun Luk, for his invaluable assistance, support and guidance. Without his help, this work would not be possible. My deep gratitude also goes to my supervisor, Professor Ke Wang, for his encouragement and support. Special thanks to the members of my committee to attend my defence: Professor Jiangchuan Liu and Professor Jian Pei. Their support and patience are appreciated.

Lastly, I would like to thank my family for their understanding and endless support, through the duration of my studies.

TABLE OF CONTENTS

Approval ii
Abstract iii
Acknowledgements iv
Table of Contents v
List of Figures vii
List of Tables viii
Chapter 1: Introduction 1
1.1 OLAP and Pre-Aggregation 1
1.2 Motivation and Objectives 2
1.3 Main Features 4
1.3.1 SplitCube 4
1.3.2 Matrix Query 5
1.3.3 B-Tree and its Application in Query Processing 5
1.4 Thesis Organization 7
Chapter 2: Related work 8
2.1 Partial Pre-Aggregation 8
2.2 Query Processing 10
Chapter 3: Basic concepts and notations 12
3.1 OLAP Cube and Dimension Hierarchy 12
3.1.1 Sub-Cube 13
3.1.2 SplitCube 14
3.2 Aggregation 15
3.3 Address Mapping Scheme 17
3.4 Cell Record and B-tree Structure 17
3.5 Query Performance and Types of Query 19
Chapter 4: Splitcube 21
4.1 Cube Construction 21
4.2 Query Processing 26
4.3 PDS Selection 28

Chapter 5: Merging of Query Point Set and B-tree 29

Chapter 6: Query Processing with Range Queries.. 34

Chapter 7: Experimental Results 39

7.1 Heavy-Duty Queries 39
7.2 QPRQ vs. STDSC 43
7.3 MQPSB vs. STDSC.. 45
7.4 QPRQ vs. MQPSB.. 48

Chapter 8: Conclusions and future work 50

Reference List.. 52

LIST OF FIGURES

Figure 1: Dimension Hierarchies of a Dataset and the OLAP Base Cube 13
Figure 2: FPA Cube, PPA Cube, and Base Cube.. 14
Figure 3: Conversions between Coordinates and Keys 18
Figure 4: Example of B-tree Structure 18
Figure 5: A Sample Dataset and its Dimension Hierarchies 23
Figure 6: Merging of Query Point Set and B-tree.. 30
Figure 7: Range Query Processing in QPRQ and STDSC 35
Figure 8: Comparison between STDSC and QPRQ in Cube Construction...... 44
Figure 9: Comparison between STDSC and QPRQ in Query Processing 45

LIST OF TABLES

Table 1: Example of a Matrix Query 20
Table 2: Dimension Cardinalities in Weather Dataset...................................... 39
Table 3: Performance Data of the Full Cube 40
Table 4: Performance Data of STDSC... 41
Table 5: Performance Data of QPRQ 41
Table 6: Performance Data of MQPSB.. 41
Table 7: Comparison between STDSC and MQPSB in Query Processing...... 46

CHAPTER 1: INTRODUCTION

1.1 OLAP and Pre-Aggregation

OLAP (On-Line Analytical Processing) has become one of the standard services in most commercial database systems as the analysis of huge amount of data plays a crucial role in an organization's decision-making process. OLAP systems provide both construction of and access to aggregate data, which is usually stored in a multi-dimensional database, or better known as cube.

OLAP cubes can be thought of as extension to a two-dimensional array of a spreadsheet, and are more suited for analysis and display of large amount of data. The power of OLAP comes from the tremendous amount of calculation on the input dataset. For online ad-hoc requests, this means a long response time. To reduce query response time and improve user experience, it is intuitive to calculate the data in advance, which is called Pre-Aggregation.

There are two types of pre-aggregation in OLAP, full pre-aggregation (FPA) and partial pre-aggregation (PPA). As indicated by their names, FPA pre-calculates all possible data that could be requested; whereas PPA pre-creates only part of the full cube. Obviously FPA can provide the best query performance and user experience. However, FPA can expand the cube to hundreds of times of the size of the original input dataset, which makes the construction and update of the OLAP cube much longer, especially for those systems that require daily

update on their input datasets. Therefore, PPA becomes the first choice of most OLAP systems.

1.2 Motivation and Objectives

During the last decade, a big collection of research articles has focused on cube construction, and many fast cube-building algorithms have been proposed. However, only a small portion of them focused on PPA. This small portion of articles is more concerned with the optimization in cube construction under a preset storage limitation. Under such a constraint, the selection of which cells to aggregate is the key question while construction time is usually ignored. Real industry presents a different picture from what is assumed in literature. To begin with, the storage is not a big issue in OLAP systems as it was before. In addition, cube construction time becomes more important as the input datasets are getting bigger and updates are more frequent (daily, usually). Conclusions from those papers have become out-of-date in today's environment.

Query processing is the most important part of an OLAP system as it contributes most to the success of the system, but research on query processing in OLAP system is even rarer than literature on PPA. Moreover, in past works response time of the query for a single cell or a range of cells, which is the basic units of a cube, is often used to measure query performance. However, such queries are not common in real OLAP applications. Queries for a group of cells in the form of spreadsheets, called *matrix queries*, are widely used in OLAP systems and are more suitable for query performance measurement.

Investigations of query processing techniques can help improve the response of an OLAP system and deserve more attention from the literature. Furthermore, a flexible control on the balance between cube construction performance and query performance is needed to meet different demands on either query response time or cube construction time. However, previous research results are far from enough for such needs. Recent works ([LAKS03], [SRDK02], [SRDK03], [MI06]) on OLAP focus on storage reduction of the data cube, which will improve the cube building efficiency and query processing efficiency.

In this thesis, we opt for a different approach, which we shall call the SplitCube approach. By relying on novel query processing techniques, we need not generate a whole data cube. By reducing the storage of the data cube, we can achieve faster query processing, which in turn makes it possible to further reduce the amount of pre-aggregation. Our goal is to devise a fast PPA construction algorithm to provide a flexible control on PPA cube construction performance and significantly improve query processing on a PPA, so that we can minimize our PPA construction time while keeping query performance at an acceptable level in an on-line environment. In this research, we describe the designs of two PPA algorithms. Furthermore, we present three query-processing techniques: Standard SplitCube (STDSC), Merging of Query Point Set and B-tree (MQPSB), and Query Processing with Range Queries (QPRQ). Through a series of comparison and analysis of the algorithms, we give an insight to the

advantages and disadvantages of each algorithm and provide the quasi-optimal solutions for different user cases.

1.3 Main Features

In our research, we first devise a fast and flexible PPA algorithm, called SplitCube, and then provide a query-processing algorithm on SplitCube. B-tree is chosen as our fundamental structure in storing and accessing pre-computed data. B-tree is used as a synonym to B+ tree in this thesis. Based on the nice features of the B-tree storage structure, we devise two enhanced query-processing algorithms, MQPSB and QPRQ.

1.3.1 SplitCube

The basic concept of *SplitCube* is to divide the input dataset into a set of smaller datasets and build each of them into an independent cube, which is called a *cubelet* in this thesis. We use one or more than one dimension from the dimension set of the input dataset to do the division. These dimensions are called *prefix dimensions*. Cells with identical values on prefix dimensions are put into the same group. Groups are then built into cubelets one by one. As all cells in the same group have the same values on prefix dimensions, we can use those values to build a unique identifier for each group and safely remove these dimensions from groups. This divide-and-conquer method brings two advantages. Firstly, building a cubelet becomes faster as each group contains fewer dimensions and cells. This improvement distinguishes SplitCube from previous PPA algorithms. It significantly reduces scale and requirements on system

resources. Secondly, the algorithm can easily adapt to multi-core systems, where processors may compute cubelets in parallel, without any collaboration with other processors.

1.3.2 Matrix Query

As mentioned in section 1.2, matrix queries are very common in real OLAP applications. Essentially a matrix query is a group of queries for single cells in the form of spreadsheet [Wit03]. In this thesis, we use the performance on some matrix queries to measure performance of query-processing algorithms. This measuring method can make our experiments and conclusions more accurate and practical for real applications.

1.3.3 B-Tree and its Application in Query Processing

In creating each cubelet, we build a B-tree with the aggregation algorithm described in detail in [LUK01]. The B-tree is indexed on *B-key*, which is the unique key generated from the values on all dimensions of a cell. Conversely, we can get the values on all dimensions of a cell from its B-key. Cells are stored at leaf level with its B-key and *measure*, which is a numerical value of interest to the user. The B-tree structure offers us a sequential access to cells on their B-keys. For efficiency reasons, cells with all measures being zero are not stored in a B-tree.

Employing B-tree in both aggregate computing and retrieving introduces some overheads on CPU and disk consumption, but the benefits it brings to

query processing outweighs its cost much more. It speeds up the construction of aggregate data and makes it possible for efficient query processing algorithms.

1.3.3.1 Merging of Query Point Set and B-tree (MQPSB)

One of our observations is that the cells between any two consecutive cells in a B-tree should be *zero-measure* cells because cells with zero-measure(s) are not stored in the B-tree. This is one place that we can improve our query-processing algorithm. The cells involved in a query, especially in a matrix query, are ordered by their B-key increasingly before being processed. After we access one cell, all cells with their B-keys less than the B-key of the *'next'* cell in the B-tree can be safely skipped because if we cannot find them in the B-tree, their measures are zeros. This query technique greatly reduces query response time, as most cubes are very sparse (i.e. quite a portion of cells in those cubes contains zeros as their measures.)

1.3.3.2 Query Processing with Range Queries (QPRQ)

For a matrix query, the cells it contains can be ordered by their B-key to form a query range. For each cell in the range, we need to do a search on the B-tree to locate the cell first, and then retrieve the cell's measure. Our observation is that all or part of the cells involved in a matrix query are consecutive by their B-keys, therefore we do not need to do the search for each of them. After accessing the first non-zero-measure cell in the B-tree, we check the next cell to see if it is the next one that we want to access in the matrix query. If it is, we save

from skipping the search for the next cell. This observation can significantly improve query performance in most queries.

1.4 Thesis Organization

In the following, we will review previous work in the literature in Chapter 2, and define the problem formally in Chapter 3. In Chapter 4, 5, and 6, we discuss the Standard SplitCube, Merging of Query Point Set and B-tree, and Query Processing with Range Queries, respectively. Finally, we analyze our experimental results in Chapter 7, and conclude in Chapter 8.

CHAPTER 2: RELATED WORK

2.1 Partial Pre-Aggregation

From the beginning of OLAP research, the necessity of partial pre-aggregation has been recognized. One of the first papers on partial pre-aggregation was published in 1996 [HRU96]. A view is a suitably modified query with a group-by clause, in which a level of each dimensional hierarchy is represented. The optimization objective is to select a set of views that will minimize the query processing cost while satisfying the constraint that total size of these views must not exceed a pre-set storage allocation. Two important assumptions are made with respect to the query processing cost: one is that a query is identical to some view, and the other is that the cost of processing such a query is equal to the size of view. With these assumptions, a model is built to define an optimal solution to this optimization problem. Remarkably, a greedy algorithm is presented which can produce a solution that is very “close” to the optimal one. A follow-up paper [SDNR96] presents an improved greedy algorithm, which runs much faster. Interestingly, it introduces a pre-aggregation algorithm that can achieve more cost saving in query processing than the “optimal” solution, for a specialized class of cubes. The idea is to partition the cube into smaller k-dimensional sub-cubes, a.k.a. chunks, each about a size of a page. Instead of selecting views for materialization, the system pre-computes selective chunks. In query processing, chunks that overlap with the query are first

identified. Among these chunks, those that have not been pre-computed need to be computed on the fly. With its smaller cube size, chunks may fit into a storage space while a materialized view may not. This chunk-based scheme is often referred to as a MOLAP (multidimensional) system.

Other papers on partial pre-aggregation are similar in the way that they can be compared within an optimization framework that includes things such as query workload, the objective function and constraints. A query workload is a mix of relational queries, or views, which are typically associated with the OLAP application(s) in question. In [GM99], the constraint is the maintenance cost of the pre-aggregated views. In [HCKL00], the query workload is any set of views chosen by the user. The problem of optimizing storage size for the pre-computed views is studied, subject to the constraint of a pre-determined query response time. In [KR99], a scheme whereby materialized views are dynamically computed, i.e. computed on demand, is described.

In [KMP02], it is shown that there could be potentially billions of views, in which case many published algorithms on view selection are slow in practice. Randomized search is proposed as a solution under the space constraint as well as under the maintenance cost constraint.

The chunk-based pre-aggregation scheme described earlier is one of the few schemes that are not based on view materialization. Others include [SDKR02], [SDKR03], and [LL03]. Partial Pre-Aggregation (PPA) accounts for a small portion of the work. Their schemes are to stop producing more group aggregate cells when the number of the potential aggregate cells is deemed

small. Interestingly the experimental results in [SDKR03] show that their pre-aggregation scheme optimizes both pre-aggregation time and query processing time, i.e., queries run faster when less pre-aggregation is done. There are few detail on how this remarkable feat is possible, nor is there any explanation on the discrepancy between apparently contradictory statements made in these two papers. In [LL03], a scheme, which can speed up query processing time in theory and practice, is proposed for pre-aggregation for MOLAP systems, despite the assertions that every aggregate in MOLAP database must be pre-computed (e.g. [PS99], [H98]). Our SplitCube approach is built on this scheme; however, we use a way to select the subset of aggregates for pre-computation. The difference will be discussed in detail in Section 3.1.

2.2 Query Processing

A detailed description for query processing using chunks, particularly, the cache management thereof, can be found in [DRNS98]. It states that chunks may be better utilized for caching purposes and more ‘fine-grained’ than views because of its uniform size. Unfortunately, the chunk-based storage organization is prone to poor performance when k, the number of dimensions, is large and the cube is sparse [BR99].

A detailed query optimization scheme for ROLAP systems is described in [ZDN98], in which selective views are materialized as relations. Queries for processing are MDX queries, which are called matrix queries. An MDX query is decomposed into a number of group-by sub-queries. A number of new relational join queries are introduced that allow multiple related group-by sub-queries to

share common subtasks. Algorithms are developed to choose which materialized views (relations) are used to evaluate the set of sub-queries. Further research results on this topic can be found in [Liang00], and [KP03].

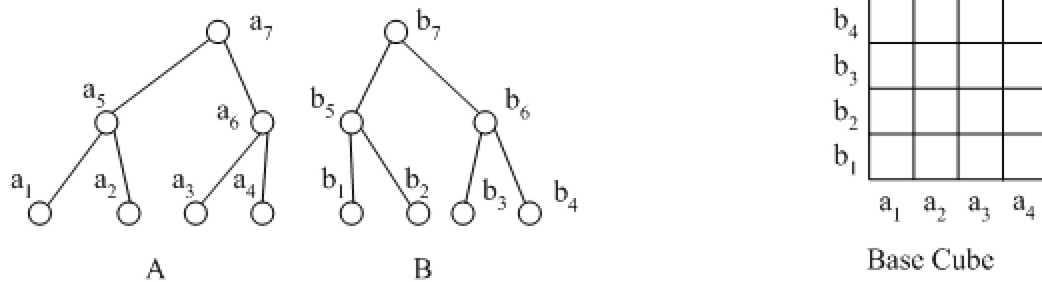
CHAPTER 3: BASIC CONCEPTS AND NOTATIONS

3.1 OLAP Cube and Dimension Hierarchy

A full OLAP cube is defined to be a k-dimensional array, where k is a positive integer. Each dimension of a cube has D_i members, $1 <= i <= k$, which are possible values on the dimension and are organized as a hierarchy. The members at the leaf level are called primary members. All other members in higher levels of the dimension hierarchy are called group members. The hierarchy is a tree hierarchy, where a member is assumed to have exactly one parent except for the root, which has no parent. In particular, there is exactly one path between a group member and any of its descendants. Primary descendants of a group member are descendants that are primary members in the dimension hierarchy.

For example, consider a 2-dimensional OLAP database with dimensions A and B. Figure 1 shows the dimension hierarchies of A and B. All members that are numbered within the interval [1, 4] are primary members, and the remaining ones are group members. The cube contains all 7 members on each dimension. The base cube contains all 4 primary members, but no group members, on each dimension. The base cube of this OLAP database is also shown in Figure 1.

Figure 1 Dimension Hierarchies of A and B of a Dataset and the OLAP Base Cube



A cell in the cube is identified uniquely by a *k*-tuple, which is composed of its coordinates along the *k* dimensions. A cell is a group cell if at least one coordinate of the cell is a group member of some dimension; otherwise, it is a primary cell. A cell stores a single numeric value, which is the measure. It is also valid for a cell to store multiple values as its measure. Measures of all primary cells are input from a data source. The measure of a group cell can be calculated according to the method to be discussed later. The calculation can take place on demand, usually when its value is required for computation of the answer to a query, or it can be pre-computed, i.e., before the query time. In the latter case, the cell is called a pre-aggregated cell. A cube is fully aggregated if all group cells in the full cube are pre-aggregated.

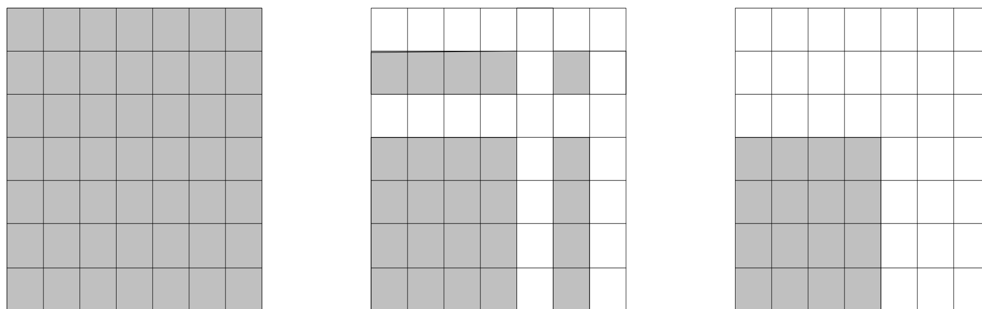
An alternative view of a cube is a relation, which is called the fact table. A fact table contains all primary and group cells. The B-tree for a cube is the physical representation of the fact table.

3.1.1 Sub-Cube

A sub-cube is a *k*-dimensional array, which has all primary cells and a subset of group cells of the full cube. The base cube, as a sub-cube of the cube,

is the *k*-dimensional array, each dimension of which has only the primary members. This is the smallest sub-cube. Figure 2 shows the fully aggregated cube, a PA cube, and a base cube, with dimensional hierarchies shown in Figure 1. The measures of all shaded cells in the Figure 2 are known before query time, because either they have been pre-computed or they are raw data. In case of the PA, as is shown in Figure 2(ii), a_6 and b_6 are pre-aggregated members. In other words, this PA cube is defined by two sets of pre-aggregated members on the two dimensions: $\{a_6\}$ and $\{b_6\}$.

Figure 2 (i) FPA Cube, (ii) PPA Cube, and (iii) Base Cube



Clearly, each PA cube contains enough information to generate a full cube with additional processing, because the base cube is always included.

3.1.2 SplitCube

A *Cubelet* is a specialized sub-cube, which is identifiable by a set of dimensions, called *prefix dimension set*, or *PDS*. The compliment of a PDS is called *cubelet dimension set*, or *CDS*. A cubelet has only primary members for the PDS and all members in the CDS. For convenience, we consider the first *m*

dimensions be included into in the PDS, where $0 < m < k$. Let $P_1, \ldots, and\ P_m$ be the set of the primary members in the prefix dimensions in the PDS. Each tuple in the Cartesian product $P_1 \times \ldots \times P_m$ is called a *prefix*. A cubelet associated with a prefix consists of all cells in the sub cube whose coordinates have the same members as the prefix for all dimensions in the PDS. Since all cells in this cubelet have the same prefix, we may simply truncate prefix from the coordinates of these cells, so that the cubelet will now be viewed as a cube with CDS as the dimension set. For the rest of the thesis, we view a cubelet as a *(k-m)-dimensional* cube.

3.2 Aggregation

We now consider how the measure of a group cell is derived. To achieve this we need to elaborate the relationships among cells in a cube.

A descendant of a cell, C, with coordinate $(t_1, \ldots, t_k)$ is the cell, C', with coordinate $(t_1', \ldots, t_k')$ such that each t_i' is either equal to t_i or is a descendant of t_i in the ith dimension hierarchy, and at least one of them is a descendant of its counterpart in the coordinate of C. The distance between C and C' is defined to be the hamming distance between them, i.e., the sum of all distances between two corresponding members in each dimension hierarchy. In particular, C' is an immediate child of C if the distance between them is 1. If t_i is a group member, the immediate children of C along the dimension i are those cells who have identical components as C except for the ith dimension. For any group member in the coordinate of C there will be a set of immediate children of C along that dimension.

A primary descendant of a cell C on the ith dimension is one of the set of cells whose coordinates are exactly the same as C's coordinate except that their coordinates on the ith dimension are primary descendants of C's coordinate on the ith dimension.

We define the measure of a group cell C as a distributive aggregate function, according to ([GBLP96]), of all measures of all primary cells that are also descendants of C. Since the definition given in [GBLP96] applies to only 2-level hierarchies a more precise definition is required. We say an aggregation function F is distributive if there exists another function G, such that $F(\{S\}) = G(F\{S_1\}, \ldots, F\{S_n\})$ where S is a set of scalar values and $\{S_1, \ldots, S_n\}$ is a partition of S. If F is the summation aggregation function, i.e. sum(), then this equation holds if G is also the summation aggregation. In fact, $F = G$ if F is the maximum() or the minimum(). Count() is also a distributive aggregate function if we choose G to be the summation aggregation function.

This definition of the measure of a group cell opens up an efficient way to compute the aggregate of a cell without reference to all of its descendant primary cells. The following theorem, Theorem 1, from [LL03] demonstrates the relation of measures between a group cell and its descendant cells.

Theorem 1: The measure of a group cell C ($t_1, \ldots, t_k$) is an aggregate function of measures of all its immediate children, along any single dimension i, as long as t_i is a group member for that dimension.

For example, the measure of the cell C (a_6, b_7) is, by definition, the aggregation function of measures of the following cells: (a_3, b_1), (a_3, b_2), (a_3, b_3),

(a_3, b_4), (a_4, b_1), (a_4, b_2), (a_4, b_3), and (a_4, b_4). According to Theorem 1, this value can be derived from the measures of the cells (a_3, b_7) and (a_4, b_7) as its immediate children along the A dimension, or from the measures of the cells (a_6, b_5) and (a_6, b_6) as its immediate children along the B dimension.

3.3 Address Mapping Scheme

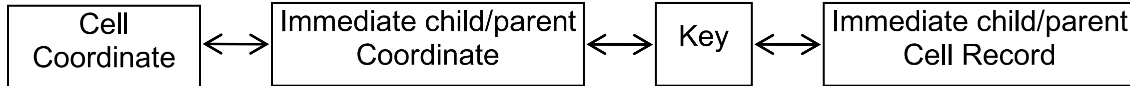
All members of each dimension, say dimension i, are mapped on the 1-to-1 basis into a range of integers $\{0, 1, \ldots, D_{i-1}\}$. The mapping, f, is designed such that the integer representation of a group member is greater than the integer representation of any of its descendant, i.e. $f(m) > f(m')$ if m' is a descendant of m. Let D_i be the total number of primary and group members for dimension i. The dimensions are so named such that $D_i <= D_{i+1}$ for $0<= i <= k-1$. Thus, there are all together $D_0*\ldots*D_{k-1}$ cells in the cube. The key of a k-dimensional cell with the numeric coordinate ($v_0, \ldots, v_{k-1}$) is given by the formula $v_0*D_1*\ldots*D_{k-1}$ + $v_1*D_2*\ldots*D_{k-1} + \ldots + v_{k-1}$, which is referred to as B-key in this thesis. Note that this key can be converted back uniquely to the numeric coordinate. If we consider the cube as a linear array, the address of a cell is then the offset from the top of the linear array.

3.4 Cell Record and B-tree Structure

The key, together with the non-zero measure(s) of the cell, makes a cell record. Such format saves a lot of space in both main-memory and disk. All cell records with non-zero measure(s) are stored in a B-tree structure to facilitate random and sequential accesses to the records, i.e. cells. The coordinate of a

cell is associated with the coordinate of its immediate child/parent along any dimension, and vice versa, by referring to its dimension hierarchy. The following figure illustrates the sequence of mappings involved:

Figure 3 Conversions between Coordinates and Keys

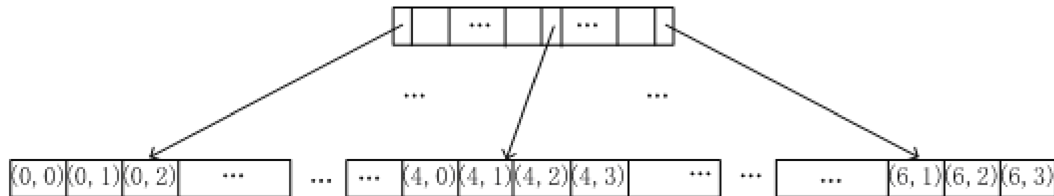


The conversion between a cell's coordinate and its key is unique. Given either the coordinate or the key of a cell, we can easily compute the coordinate/cell record of its immediate parent/children.

Lemma 1: The key of a group cell is always greater than the keys of its immediate children cells.

In Figure 4, a possible B-tree is built from a dataset with dimension hierarchies in Figure 1. A directed edge in the tree represents the association between the cell and one of its immediate children. The measure of a cell can be computed from its immediate children.

Figure 4 Example of B-tree Structure



Cell records are actually stored in a small space unit, page, in main memory. There are pointers (memory addresses) among pages to maintain such

a tree structure. When the cell records are written to disk, it is done page by page. Pointers among pages are converted into file position pointers so that the connections among pages are not lost.

3.5 Query Performance and Types of Query

Testing for performance of random point query is common among most papers on cube building. A point query is defined as the address of a cell T ($q_1, \ldots, q_k$) where q_i, $1<=i<=k$, is the coordinate of the cell in the ith dimension. Since there is a 1-to-1 correspondence between point queries and the cell addresses, the terms of point query and cell are used interchangeably in this thesis.

A range query is essentially a sub-cube in our context. A random range query however does not make much sense in OLAP applications when the ranges have already been carefully defined in the form of dimension hierarchies. Instead, most OLAP systems implement a query language called MDX [MS03]. Here, we adopt a simplified form of MDX, the matrix query.

A matrix query is to display data contained in the fact table in the form of a pivot table, which is made popular by many data visualization tools such as spreadsheet packages. It consists of three components: a (point) query with at least two group members, and two dimensions, say i and j, identified as the row and column dimensions respectively, and q_i and q_j must be group members for their respective dimensions, with immediate descendant members $q_{i,1}, \ldots, q_{i,r}$, and $q_{j,1}, \ldots q_{j,c}$. The answer to the matrix query will be a table, with $q_{i,1}, \ldots$ and $q_{i,r}$ as labels for the row, and $q_{j,1}, \ldots$ and $q_{j,c}$ for the column. Assuming $i < j$, the entry

(v,w) of the table, where $1<=v<=i$ and $1<=w<=j$, is the answer for the query ($q1, \ldots, q_{i-1}, q_{i,v}, q_{i+1}, \ldots q_{j-1}, q_{j,w}, q_{j+1}, \ldots, q_k$).

For example, the query matrix for the 2D database, (a_7, b_7), with A as the row dimension and B as the column dimension, consists of a matrix of point queries (a_5, b_5), (a_5, b_6), (a_6, b_5) and (a_6, b_6).

Table 1 Example of a Matrix Query

Row\Column labels	b_5	b_6
a_5	(a_5,b_5)	(a_5,b_6)
a_6	(a_6,b_5)	(a_6,b_6)

Processing of matrix query is accomplished by processing point queries individually inside the matrix. To process the matrix query (a_7, b_7) in the above example, we apply the point queries (a_5,b_5), (a_5,b_6), (a_6,b_5), and (a_6,b_6) of cubelet SCA. For this example, we may do better. Observing that the first coordinate of the matrix query, a_7, is a group member of the dimension A with a_1, a_2, a_3, and a_4 as its primary descendants, we may simply apply the point queries (b_5) and (b_6) against the cubelets associated with a_1, a_2, a_3, and a_4.

CHAPTER 4: SPLITCUBE

SplitCube, the aforementioned PPA algorithm, contains two phases in building a partial OLAP cube: dataset division and cubelet construction.

4.1 Cube Construction

In order to break the input down into a set of groups, we first choose prefix dimensions from the dimension set of the input, and then we project the original input on prefix dimensions. Tuples with the identical values on prefix dimensions are put into the same group. Since all tuples within one group has the same values on prefix dimensions, we will ignore them and only consider cubelet dimensions. To identify each group, we use the values on prefix dimensions to generate a unique number as the identifier of the cubelet generated from that group.

In this thesis, we have two algorithms to build cubelets. The first one is for constructing full cubelets, and the second one is for creating partial cubelets. We use the disk-based aggregation algorithm in [LUK01] to build full cubelets. The second algorithm is actually a variant of the first one. The difference lies in that the partial cubelet construction skips the pre-aggregation on the last dimension. Here, we present the two algorithms and give examples to show how each algorithm works.

Aggregation data for one cubelet is stored in a standard B tree. For performance reasons, we take a page of size 4096 bytes to store a node. In non-leaf node, the record size is the size of measure(s) plus the size of a pointer, which is 4 bytes on a 32-bit platform. The size of a record in leaf node is the size of B-key (8 bytes) plus the size of measure(s). In our experiment, the measure size is 8 bytes. We choose 0.5 as our density rate for non-leaf nodes. The fan-out is determined by formula (nodesize – nodeheadersize) * densityrate / noderecordesize. In our case, it is (4096-8) * 0.5 / 12 = 170. Such a big fan-out makes the number of levels less than 3 for most of B-trees (cubelets). A B-tree with fewer levels makes the searching faster, which is desirable in our query processing.

Full Cubelet Building Algorithm

INPUT: Input dataset for the cubelet to be built, *DS*; Dimension Hierarchies, *DH*; *PDS*

OUTPUT: The B-tree for that cubelet

BEGIN

  Create an empty B-tree, *BT* based on the *DH* and *PDS*

  For each tuple, *t,* in *DS*

    Use theorem 1 to update all ancestor cells of *t* in *BT*

  End For

  Skip cells with zero measure and write them into the cube file

END

As we can see in the following partial cubelet building algorithm, the only difference is that there is a checking on the value of the last dimension of each cell to be updated.

Partial Cubelet Building Algorithm

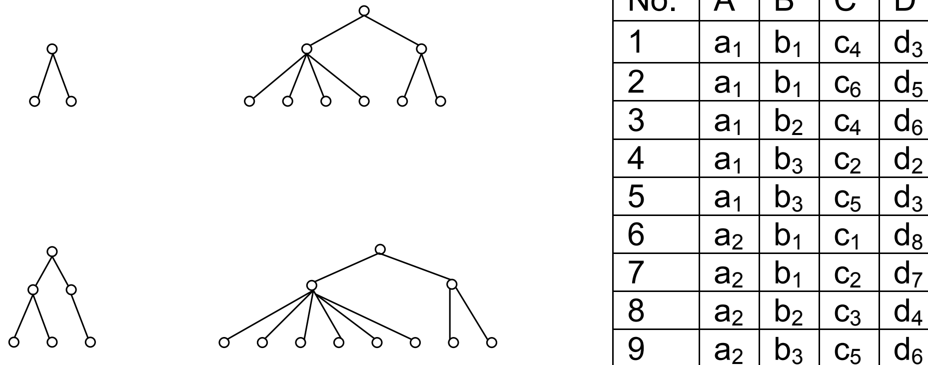
INPUT: Input dataset for the cubelet to be built, *DS*; Dimension Hierarchies, *DH*; *PDS*

OUTPUT: The B-tree for that cubelet

```
BEGIN
    Create an empty B-tree, BT based on the DH and PDS
    For each tuple, t, in DS
        For each ancestor cell, c in the cell list to be updated (determined by theorem 1)
            If c's coordinate on the last dimension is a group member then
                Continue
            Else
                Update c's measure with t
            End If
    End For
    Skip cells with zero measure and write them into the cube file
END
```

We will use the following sample dataset and dimension hierarchies to show how each algorithm works.

Figure 5 A Sample Dataset and its Dimension Hierarchies



No.	A	B	C	D
1	a_1	b_1	c_4	d_3
2	a_1	b_1	c_6	d_5
3	a_1	b_2	c_4	d_6
4	a_1	b_3	c_2	d_2
5	a_1	b_3	c_5	d_3
6	a_2	b_1	c_1	d_8
7	a_2	b_1	c_2	d_7
8	a_2	b_2	c_3	d_4
9	a_2	b_3	c_5	d_6

Suppose we choose dimension A and dimension B to be the prefix dimensions, and we will get 6 groups from the sample dataset: $a_1b_1\{1,2\}$, $a_1b_2\{3\}$, $a_1b_3\{4,5\}$, $a_2b_1\{6,7\}$, $a_2b_2\{8\}$, and $a_2b_3\{9\}$. In each group, we use the combination of values on prefix dimensions as the group's ID and use record IDs to represent records in the dataset. In real dataset, it is very likely that some groups contain

no records. After the splitting, we check if there is any group too big for our memory capacity. If so, big groups need to be further divided into sub-groups. Each group/sub-group can then be loaded into memory and built into a cubelet/sub-cubelet.

We take the first group as an example to illustrate how to build a cubelet in the two algorithms described. An empty B-tree for dimension C and dimension D will contain the following cells:

$\{c_1d_1, \ldots, c_1d_{11}, c_2d_1, \ldots, c_2d_{11}, c_3d_1, \ldots, c_3d_{11}, c_4d_1, \ldots, c_4d_{11}, c_5d_1, \ldots, c_5d_{11}, c_6d_1, \ldots, c_6d_{11}, c_7d_1, \ldots, c_7d_{11}, c_8d_1, \ldots, c_8d_{11}, c_9d_1, \ldots, c_9d_{11}\}$

In group a_1b_1, we have only two tuples (primary cells), c_4d_3 and c_6d_5. So only the following cells in the B-tree will be updated according to theorem 1, and these cells are to be stored in the cube file.

$\{c_4d_3,c_4d_9,c_4d_{11},c_5d_6,c_5d_9,c_5d_{11},c_7d_3,c_7d_9,c_7d_{11},c_8d_6,c_8d_9,c_8d_{11},c_9d_3,c_9d_6,c_9d_9, c_9d_{11}\}$

This is how STDSC builds a full cubelet. In the partial cubelet building algorithm, the same empty B-tree is created and we take group a_1b_1 as an example to show the difference. The same set of cells as above is selected and considered to be updated. Since partial cubelet construction does not do pre-aggregation on the last dimension, all cells with the value of last dimension being a group member will be ignored. Thus only the following cells are updated and stored into the cube file.

$\{c_4d_3, c_5d_6, c_7d_3, c_8d_6, c_9d_3, c_9d_6\}$

The size of the cubelet built from partial cubelet building algorithm is much smaller than the one built by the full cubelet algorithm. We list out the cells of each cubelet built from the two algorithms as follows:

Full cubelets:

a_1b_1:{c_4d_3,c_4d_9,c_4d_{11},c_5d_6,c_5d_9,c_5d_{11},c_7d_3,c_7d_9,c_7d_{11},c_8d_6,c_8d_9,c_8d_{11},c_9d_3,c_9d_6,c_9d_9,c_9d_{11}}

a_1b_2:{c_4d_6,c_4d_9,c_4d_{11},c_7d_6,c_7d_9,c_7d_{11},c_9d_6,c_9d_9,c_9d_{11}}

a_1b_3:{c_2d_2,c_2d_9,c_2d_{11},c_5d_3,c_5d_9,c_5d_{11},c_7d_2,c_7d_9,c_7d_{11},c_8d_3,c_8d_9,c_8d_{11},c_9d_2,c_9d_3,c_9d_9,c_9d_{11}}

a_2b_1:{c_1d_8,c_1d_{10},c_1d_{11},c_2d_7,c_2d_{10},c_2d_{11},c_7d_7,c_7d_8,c_7d_{10},c_7d_{11},c_9d_7,c_9d_8,c_9d_{10},c_9d_{11}}

a_2b_2:{c_3d_4,c_3d_9,c_3d_{11},c_7d_4,c_7d_9,c_7d_{11},c_9d_4,c_9d_9,c_9d_{11}}

a_2b_3:{c_5d_6,c_5d_9,c_5d_{11},c_8d_6,c_8d_9,c_8d_{11},c_9d_6,c_9d_9,c_9d_{11}}

Partial cubelets:

a_1b_1: {c_4d_3, c_5d_6, c_7d_3, c_8d_6, c_9d_3, c_9d_6}

a_1b_2: {c_4d_6, c_7d_6, c_9d_6}

a_1b_3: {c_2d_2, c_5d_3, c_7d_2, c_8d_3, c_9d_2, c_9d_3}

a_2b_1: {c_1d_8, c_2d_7, c_7d_7, c_7d_8, c_9d_7, c_9d_8}

a_2b_2: {c_3d_4, c_7d_4, c_9d_4}

a_2b_3: {c_5d_6, c_8d_6, c_9d_6}

After all cubelets are created, we write the configuration information about the cube into a separate file, including the number of prefix dimensions and which dimensions they are, and the entry point of each cubelet in the cube file. Before a query can be made on a cube, the configuration file of the cube must be loaded into memory first.

4.2 Query Processing

Generally, query processing in SplitCube and its variants are composed of three steps due to the spread of aggregates into multiple individual cubelets. The first step is to determine which cubelets are involved in the query processing. The involved cubelets can be determined from the query, PDS, and the dimension hierarchies. For values on prefix dimensions in the query, each of them will be replaced by the set of its primary descendants. The elements of Cartesian Product of all these sets form a new set, from which we generate the identifiers of cubelets that are involved in the query using the same algorithm mentioned in cube construction. In the second step, we take the values on cubelet dimensions of the query to form a new query for each cubelet involved. We use the method in previous chapter to calculate the B-key of the new query. With the B-key, we can retrieve the measure(s) for the new query in the B-tree of each cubelet. From the result retrieved from each cubelet, we can compute the result for the original query in the last step.

For matrix queries, the first step is same as a single-point query. However, in the second step, we will use the values in the query and the specified row dimension and column dimension to form a set of point queries, called QPS, which will be issued to each cubelet. The last step is similar to single-point query processing. The following is the algorithm of standard SplitCube (STDSC) for matrix queries.

The Algorithm for Matrix Queries in STDSC

INPUT: the query, *orgQ*; row dimension index, *rowIndex*; column dimension index, *columnIndex*; *PDS*; dimension hierarchies, *DH*

OUTPUT: The answer to the matrix query

BEGIN

Let *QPS* be the query point set for each cubelet, which is derived from *orgQ*, *rowIndex*, *columnIndex*, *PDS*, and *DH*

Let *CS* be the set of cubelets involved in the matrix query, which is determined by *orgQ*, *PDS*, and *DH*

Let *AS* be the set holding answers of point queries from each cubelet

For each cubelet, *c,* in *CS*

For each point query, *q* in *QPS*

Let *a* be the answer of *q* on *c*

Put *a* into *AS*

End For

End For

Calculate the answer to *orgQ* from *AS*

END

We will use the same sample dataset and dimension hierarchies to illustrate how a matrix query is processed. Suppose our matrix query is $\{a_3, b_4, c_9, d_{11}\}$, row dimension is C, and column dimension is D. The query is divided into two parts: $\{a_3, b_4\}$ as the PDS part, and $\{c_9, d_{11}\}$ as the CDS part. The primary descendant set for a_3 and b_4 are $\{a_1, a_2\}$, and $\{b_1, b_2\}$, respectively. There are four elements in their Cartesian product, namely, a_1b_1, a_1b_2, a_2b_1, and a_2b_2, which represent the four cubelets involved in this query. The CDS part, $\{c_9, d_{11}\}$, together with the row dimension and column dimension, will form a query point set for this matrix query, $\{(c_7,d_9), (c_7,d_{10}), (c_8,d_9), (c_8,d_{10})\}$. This query point set, as a set of individual queries, is issued to the four cubelets respectively. Results from the four cubelets are put together to compute the answer to the query $\{a_3, b_4, c_9, d_{11}\}$.

4.3 PDS Selection

PDS, as one of the most important parameters, affects the performance in both cube construction and query processing. In generating a cube, it determines the number of cubelets and the cube size, which are both critical factors in generating a cube. Generally, the smaller a cube is, the faster we can build it. From the discussion in Section 4.2, we can see that a point query is issued to each cubelet involved. Generally, the more cubelets a cube contains, the more cubelets are involved for the same matrix query. To improve query performance, we need a small number of cubelets; however, we also need to pick as many prefix dimensions as possible to reduce the cube size. Choosing dimensions with a small number of primary members can reduce cube size and limit the number of cubelets as well. In our experiments, this strategy succeeds in achieving good performance in both cube construction and query processing.

CHAPTER 5: MERGING OF QUERY POINT SET AND B-TREE

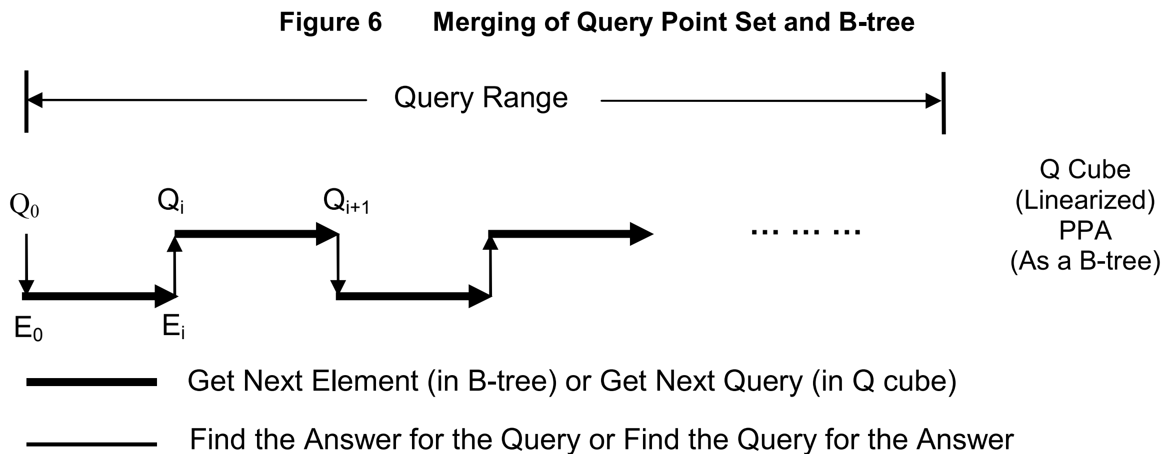
From Section 3.5, we know a matrix query is indeed a set of query points, which is called QPS, i.e. Query Point Set. In each cubelet, only cells with non-zero measures will be stored in the B-tree. Obviously, the overlap between QPS and B-tree is the points whose values we need to derive the answer to the matrix query. Unfortunately, despite our efforts to minimize the QPS, it is still a very large set, which could contain hundreds or thousands of point queries. Only a tiny fraction of them is stored in the B-tree. The standard algorithm to compute the intersection of these two large sets of points has linear running time in terms of the size of the QPS and the number of elements in the PA cube within the same range as the QPS. It is linear because both the PA cube and the QPS are sorted according to their B-key. The problem is that a linear algorithm is not good enough, because the QPS can be very large if the PA cube contains only a few pre-computed aggregates. It takes a long time just to generate millions of query points.

Our solution is to avoid scanning every query contained in the QPS within the B-tree, thus we need not generate all query points. We observe that if a query point, as a key, cannot be found in the PA, it can be ignored. One may, as an alternative, scan only the elements within the query range in the PA. For each element within the range, its key may be converted into a cell address. Only one lookup is needed to find out whether the cell address matches any query point in

the QPS. This approach, by matching the potential answers with the queries, will work well if the potential answers are fewer than the queries. However, the condition cannot be ascertained in advanced.

We propose an effective solution for this problem, which makes use of the random access feature to the PA. The process is illustrated in Figure 6:

Figure 6 Merging of Query Point Set and B-tree



The matching process begins with finding the answer for the first query, say Q_0 in the QPS. If the answer to Q_0 is found in E_0, an element in the PA cube, the measure in E_0 will be retrieved. Then next element of E_0, say E_1 is located. E_1 will then be the element that contains the key, which is the smallest one among all keys in the PA cube that is greater than Q_0. In case that the answer to Q_0 cannot be located, E_1 is located as well. In either case, a query for E_1, say Q_i, is created, and an attempt is made to locate Q_i in the QPS. If the attempt is successful, the measure of E_1 will be retrieved as the answer for Q_i. Regardless of whether Q_i is found, the next query in key sequence, Q_{i+1}, will be located and

the search for the answer in the PA cube for a query, which is Q_{i+1} this time, will begin again. This process will be repeated until the end of the range is reached.

The Algorithm for Matrix Queries in MQPSB

INPUT: the query, *orgQ*; row dimension index, *rowIndex*; column dimension index, *columnIndex*; *PDS*; dimension hierarchies, *DH*

OUTPUT: The answer to the matrix query

BEGIN

    Let *QPS* be the query point set for each cubelet, which is derived from *orgQ*, *rowIndex*, *columnIndex*, *PDS*, and *DH*

    Let *CS* be the set of cubelets involved in the matrix query, which is determined by *orgQ*, *PDS*, and *DH*

    Let *AS* be the set holding answers of point queries from each cubelet

    For each cubelet, *c,* in *CS*

        Let *q* be the first point query in *QPS*

        Let *done* = FALSE

        While (*done* is FALSE)

            If an element *e* is found in *c*, with the B-key of *e* equals to the B-key of *q* Then

                Retrieve the measure of *e* and put it into *AS*

                Let *S* be the set of all elements in *c* with B-keys > *q*

                If *S* is empty Then

                    done = TRUE

                Else

                    Let *e* = the element with the smallest key in *S*

                End If

                If *q* is found such that *q* = *e* Then

                    q = *e*

                    Let *S* be the set of all query points in *QPS* with keys > *e*

                    If *S* is empty Then

                        done = TURE

                    Else

                        Let *q* = the query point in *QPS* with the smallest B-key

                    End If

                End If

            Else

                Let *q* = the query point in *QPS* with the smallest B-key

```
            End If
        End While
    End For
    Calculate the answer to orgQ from AS
END
```

Note that the algorithm does not do exactly what is portrayed in Figure 5. If a query is found for a given answer (as Q_i is matched with E_i), the next element to E_i is located instead of the next query Q_{i+1}, as is the case in Figure 6. The next query is located only in the case when no query can be found for a given answer. This modification simplifies implementation because the retrieval of answer to a query is part of the B-tree code. It does not hurt performance.

We use the same example as for STDSC in figure 5. Suppose our query is $\{a_3, b_4, c_9, d_{11}\}$, C is the row dimension and D is the column dimension. A and B are prefix dimensions. From example 1, we know four cubelets will be involved in this query, (a_1,b_1), (a_1,b_2), (a_2,b_1), and (a_2,b_2). There will be 4 point queries, $\{(c_7,d_9), (c_7,d_{10}), (c_8,d_9), (c_8,d_{10})\}$, for each of the four cubelets. In STDSC, we will search 4 times for 4 query points. We take the first cubelet, (a_1, b_1), to show how MQPSB works. The following is how non-zero cells are stored in the cubelet (a_1,b_1).

a_1b_1:$\{c_4d_3,c_4d_9,c_4d_{11},c_5d_6,c_5d_9,c_5d_{11},c_7d_3,c_7d_9,c_7d_{11},c_8d_6,c_8d_9,c_8d_{11},c_9d_3,c_9d_6,c_9d_9,c_9d_{11}\}$

According to the algorithm, we locate the first point query (c_7, d_9) in cubelet a_1b_1. Cell (c_7, d_9) is found, so we move to the next cell (c_7, d_{11}), which is not found in $\{(c_7, d_9), (c_7, d_{10}), (c_8, d_9), (c_8, d_{10})\}$. Then we move to the next point

query with B-key bigger than (c_7, d_{11}), which is (c_8, d_9). Now we do a search for (c_8, d_9) in a_1b_1. Since it is found, then we check the next cell to (c_8, d_9), which is (c_8, d_{11}). As (c_8, d_{11}) cannot be found in the query point set and there is no other point queries having their B-keys larger than (c_8, d_{11}), the query processing is done. From the description above, we can see that we only did two searches in cubelet a_1b_1 rather than four searches in STDSC.

The change in the processing of a set of point queries in MQPSB substantially improves the algorithm's query performance because it sharply reduces the number of searches in cubelets.

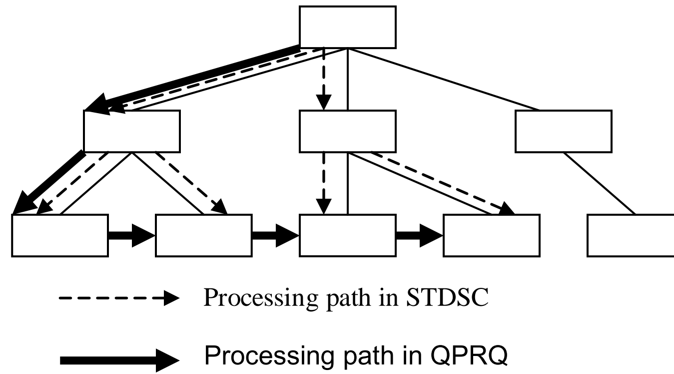
CHAPTER 6: QUERY PROCESSING WITH RANGE QUERIES

In addition to MQPSB, we now devise another improved query-processing algorithm, Query Processing with Range Queries (QPRQ).

QPRQ takes the advantage of the nice structure of B-tree. The B-tree employed in our research is clustered on the B-keys of cells, which are lexicographically ordered on dimensions. If we have a set of point queries and the B-keys of all these point queries are consecutively stored in the B-tree, then we only need to locate the first query point in the B-tree and then keep retrieving the measures of cells behind the first cell until we reach the last query point in the B-tree. This would save us a lot from searching in B-tree, especially when the set is big.

However, the query points derived from a matrix query are not always consecutively stored in the B-tree. To take the advantage of the beautiful feature of B-tree, we artificially create such query point set, referred to as Range Query in this thesis. A range query is defined as a cell address $(q_1, \ldots, q_k)$, together with a range of primary members (in consecutive order) in the kth dimension. If q_k is a primary member, then it is in essence a point query with a range of one member, i.e. itself. Otherwise, the associated range includes the primary members that are descendants of q_k. The following figure shows the difference between STDSC and QPRQ in processing a range query.

Figure 7   Range Query Processing in QPRQ and STDSC



Our method is to adopt the partial cubelet-creating algorithm described in chapter 4 and skip the aggregation on the last dimension in building each cubelet, so that only cells with values on the last dimension being primary members are stored in cubelets. Whenever there is a point query with the value on the last dimension being a group member, we construct a range query for that point query and calculate the answer by issuing the range query to the cubelet.

Suppose we have a point query $(q_1, \ldots, q_k)$, and q_k is a group member on the last dimension k. Based on this point query, we build a range query $\{(q_1, \ldots, q_{km}), \ldots, (q_1, \ldots, q_{kn})\}$, where $\{q_{km}, \ldots, q_{kn}\}$ are the primary descendants of q_k in the hierarchy of dimension k. From the hierarchy structure, we can see that $q_{km}, \ldots, q_{kn}$ are consecutive in the hierarchy tree, so cells $(q_1, \ldots, q_{km}), \ldots, (q_1, \ldots, q_{kn})$ should all be consecutively stored in the cubelet. When we calculate the answer to $(q_1, \ldots, q_k)$ we only need to locate $(q_1, \ldots, q_{km})$ first, and then we retrieve the measures of cells whose B-keys are bigger than that of $(q_1, \ldots, q_{km})$ in the B-tree, until we reach the cell with its B-key larger than that of $(q_1, \ldots, q_{kn})$. Finally, we calculate the answer from those measures.

The Algorithm for Matrix Queries in QPRQ

INPUT: the query, *orgQ*; row dimension index, *rowIndex*; column dimension index, *columnIndex*; *PDS*; dimension hierarchies, *DH*

OUTPUT: The answer to the matrix query

BEGIN

Let *QPS* be the query point set for each cubelet, which is derived from *orgQ*, *rowIndex*, *columnIndex*, *PDS*, and *DH*

Let *CS* be the set of cubelets involved in the matrix query, which is determined by *orgQ*, *PDS*, and *DH*

Let *AS* be the set holding answers to point queries from each cubelet

For each cubelet, *c,* in *CS*

For each point query, *q* in *QPS*

Construct the range query, *RQ* for *q*

Let *RAS* be the set to hold answers to query points in *RQ*

Locate the cell c_f in *c* with the B-key of c_f not less than the B-key of the first query point in *RQ*

While (the B-key of c_f not bigger than the last query point in *RQ*)

Put the measure of c_f into *RAS*

Let c_f be the next cell of c_f in cubelet *c*

End While

Calculate the answer, *a* for *q* on *c* from *RAS*

Put *a* into *AS*

End For

End For

Calculate the answer to *orgQ* from *AS*

END

You may ask why we do not pre-aggregate on the last dimension, so that we do not have to construct the range query, which is supposedly faster than QPRQ. That is true. However, most of the performance improvement is gained from cublet construction and disk storage by skipping the aggregation on the last dimension. The last dimension contains the most members among all dimensions. Without pre-computation on it, we can build the cube with a shorter PDS within a similar amount of time that STDSC takes to build the cube with a

longer PDS. A shorter PDS means fewer cubelets involved in the same query, which certainly will improve the performance of query processing.

To show the strength of QPRQ, we still use the same example dataset as in Figure 5, but we will use a shorter PDS to group the dataset and build cubelets. In this example, we only use dimension A to do the grouping. The primary cells (input records) contained in the two groups are shown as follows:

Group a_1: {1, 2, 3, 4, 5}
Group a_2: {6, 7, 8, 9}

The cells created in each cubelet are presented below:

a_1:{$b_1c_4d_3$,$b_1c_6d_5$,$b_1c_7d_3$,$b_1c_8d_5$,$b_1c_9d_3$,$b_1c_9d_5$,$b_2c_4d_6$,$b_2c_7d_6$,$b_2c_9d_6$,$b_3c_2d_2$,$b_3c_5d_3$,$b_3c_7d_2$,$b_3c_8d_3$,$b_3c_9d_2$,$b_3c_9d_3$,$b_4c_4d_3$,$b_4c_4d_6$,$b_4c_6d_5$,$b_4c_7d_3$,$b_4c_7d_6$,$b_4c_8d_5$,$b_4c_9d_3$,$b_4c_9d_5$,$b_4c_9d_6$,$b_5c_2d_2$,$b_5c_5d_3$,$b_5c_7d_2$,$b_5c_8d_3$,$b_5c_9d_2$,$b_5c_9d_3$,$b_6c_2d_2$,$b_6c_4d_3$,$b_6c_4d_6$,$b_6c_5d_3$,$b_6c_6d_5$,$b_6c_7d_2$,$b_6c_7d_3$,$b_6c_7d_6$,$b_6c_8d_3$,$b_6c_8d_5$,$b_6c_9d_2$,$b_6c_9d_3$,$b_6c_9d_5$,$b_6c_9d_6$}

a_2:{$b_1c_1d_8$,$b_1c_2d_7$,$b_1c_7d_7$,$b_1c_7d_8$,$b_1c_9d_7$,$b_1c_9d_8$,$b_2c_3d_4$,$b_2c_7d_4$,$b_2c_9d_4$,$b_3c_5d_6$,$b_3c_8d_6$,$b_3c_9d_6$,$b_4c_1d_8$,$b_4c_2d_7$,$b_4c_3d_4$,$b_4c_7d_4$,$b_4c_7d_7$,$b_4c_7d_8$,$b_4c_9d_4$,$b_4c_9d_7$,$b_4c_9d_8$,$b_5c_5d_6$,$b_5c_8d_6$,$b_5c_9d_6$,$b_6c_1d_8$,$b_6c_2d_7$,$b_6c_3d_4$,$b_6c_5d_6$,$b_6c_7d_4$,$b_6c_7d_7$,$b_6c_7d_8$,$b_6c_8d_6$,$b_6c_9d_4$,$b_6c_9d_6$,$b_6c_9d_7$,$b_6c_9d_8$}

There are totally 80 non-zero cells in the cube. Recall that in the STDSC, with PDS being {A, B}, there are totally 74 non-zero cells. Clearly, the cube construction time and cube size should be similar. Now, let us see how QPRQ works. We use the same matrix query as in STDSC, (a_3, b_4, c_9, d_{11}), and C as the

row dimension and D the column dimension. Only two cubelets are involved, a_1 and a_2. For each of them, there will be four point queries:

$\{(b_4,c_7,d_9),(b_4,c_7,d_{10}),(b_4,c_8,d_9),(b_4,c_8,d_{10})\}$

Here, the advantage of QPRQ is obvious. We can see that the 4 point queries are issued to 2 cubelets, so there are totally 8 point queries in QPRQ. Whereas in STDSC, there are 4 point queries for 4 cubelets, 16 piont queries in total.

We take the first point query on cubelet a_1 as an example to show how QPRQ works. First of all, a range query, $\{(b_4, c_7, d_1), \ldots, (b_4,c_7,d_6)\}$, is derived from (b_4,c_7,d_9). Since (b_4,c_7,d_1) and (b_4,c_7,d_2) can not be found in a_1, we locate (b_4,c_7,d_3) in a_1 and record its measure. Then we take the measure of the next cell (b_4,c_7,d_6) because its B-key is not bigger than the last point query in the range query, (b_4,c_7,d_6). After that, the B-key of the next cell, (b_4, c_8, d_5), is checked and we find out that it is bigger than (b_4, c_7, d_6). Now, the range query processing for point query (b_4, c_7, d_9) is done. The same processing is repeated for all the point queries in each cubelet involved.

Even for each individual point query, the range query processing is a little slower than STDSC, which does not need extra processing. The benefit we obtain from fewer point queries and advanced range query processing technique outweighs the performance loss in each individual point query. We will see how the significant the performance improvement is in QPRQ in the experimental result analysis in the next chapter.

CHAPTER 7: EXPERIMENTAL RESULTS

All the experiments presented in this thesis are conducted on a Compaq laptop equipped with a 2.4GHz processor and 512-Mb memory. To get measurement that is more meaningful on performance for each algorithm, we test algorithms with heavy-duty matrix queries as discussed later.

In our experiment, most tests are done on the weather dataset, which originally has 9 dimensions, whose cardinalities are: 2, 8, 10, 30, 101, 152, 179, 352, and 7037. All dimensional hierarchies are 2-level, i.e. the only group members are roots of the hierarchies. We bulk the hierarchies of the last 4 dimensions by adding one more level to each dimension hierarchy. The cardinalities of all dimensions are shown in the following table:

Table 2 Dimension Cardinalities in Weather Dataset

level\dimension	0	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1	1
2	2	8	10	30	101	8	8	18	98
3						152	179	352	7037

7.1 Heavy-Duty Queries

Heavy-duty queries are defined by the heavy computation needed to deliver the answers to them. Generally, a decision-making query retrieves a large number of data from the cube. For this thesis, heavy-duty queries make it easier to compare performance of different query processing strategies. In weather dataset choosing the 5th dimension and 9th dimension as the row and column

dimensions, respectively, generates the most sub-queries. However, we choose the last two dimensions as the row dimension and column dimension in our experiments. The reason is that the number of sub-queries generated from this combination is around the average number of sub-queries among the 36 possible combinations, so we use it as the representative of heavy-duty matrix queries. To eliminate the impact on query performance caused by previous queries, we write a large file to clear system cache before a query is issued to the cube.

Before we test our new algorithms, we first build a full cube with the Weather dataset, against which results from new algorithms can be compared. The cube size, cube construction time, and the query performance on the full cube are used as a baseline to show how the novel algorithms improve in both cube construction and query processing. In the test, STDSC and MQPSB use the full cubelet construction algorithm and QPRQ uses the partial cubelet building algorithm to build the cube. The following tables show the testing results on the full cube and our new algorithms, respectively.

Table 3 Performance Data of the Full Cube

PDS	Construction Time (s)	Size (GB)	Number of Non-empty Cubelets	Response Time(s)
N/A	10041	9.03	1	157.602

Table 4 Performance Data of STDSC

PDS	Construction Time (s)	Size (GB)	Number of Non-empty Cubelets	Response Time(s)
0,1,2,5	337	0.985	11668	182.893
0,1,2,4	407	1.47	4105	87.366
0,1,2,3	475	1.66	4192	80.506
0,1,5	466	1.74	2206	72.935
0,1,2	709	2.32	160	11.256
1,2,3	1017	3.36	2364	77.171
1,8	2383	4.99	45055	606.39

Table 5 Performance Data of QPRQ

PDS	Construction Time (s)	Size (GB)	Number of Non-empty Cubelets	Response Time(s)
0,1,2,(8)	448	1.58	160	22.863
0,1,(8)	630	2.45	16	1.332
1,2,(8)	627	2.48	80	6.610
1,(8)	1339	4.37	8	1.051

Table 6 Performance Data of MQPSB

PDS	Construction Time (s)	Size (GB)	Number of Non-empty Cubelets	Response Time(s)
0,1,2,5	337	0.985	11668	34.239
0,1,2,4	407	1.47	4105	23.884
0,1,2,3	475	1.66	4192	28.681
0,1,5	466	1.74	2206	25.55
0,1,2	709	2.32	160	7.98
1,2,3	1017	3.36	2364	25.14
1,8	2383	4.99	45055	254.82

From Table 3 and Table 4, we can see the substantial improvement obtained from STDSC in cube storage, cube building time, and query processing. The PPA cube is only 10% - 50% of the size of the full cube, and STDSC takes only 3% - 25% of the full cube's building time. Compared with the full cube algorithm, STDSC also shows big advantage. For the same matrix query, STDSC is 1 - 14 times faster than the full cube algorithm. Obviously, the other two enhanced algorithms, MQPSB and QPRQ, should perform even better than STDSC. In the rest of this section we will focus on the advantage and disadvantage of each algorithm, and try to discover some useful patterns in comparing any two of them, which are helpful in choosing the right algorithm for different user cases.

To make the comparison more accurate, one of the basic principles in our analysis is that all the comparisons should be based on the same set of aggregates. For example, when algorithm A and algorithm B are compared, they should create the same set of aggregates and the same set is used for the query processing as well.

MQPSB and STDSC both generate full cubelets, so the comparison between them should be simply on the same PDS. They will create the same group of cubelets and process queries on the same set of aggregates.

However, we cannot compare QPRQ with STDSC the same way as we compare MQPSB with STDSC because QPRQ only creates partial cubelets. One way of comparing QPRQ with STDSC would be to compare the performance data of QPRQ with {A} as the PDS against the data of STDSC with {A, 8}. The

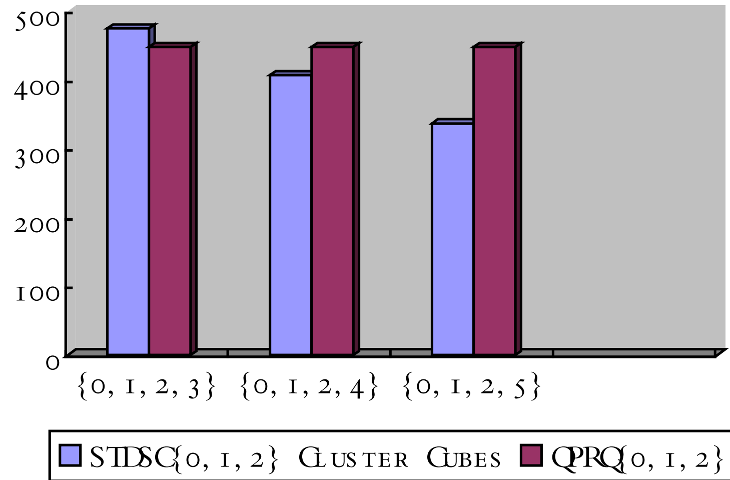
problem with this comparison is that there are far more cubelets associated with STDSC (i.e., 45055, when A is {1}) than that with QPRQ, (i.e, 8). Since a big number of cubelets greatly lowers the performance in query processing, the last dimension is usually not picked as a prefix dimension in STDSC.

7.2 QPRQ vs. STDSC

For the above reasons we decide to compare QPRQ with {A} as PDS, with STDSC on its cluster cubes. For a cube created by QPRQ with PDS {A}, its cluster cubes generated by STDSC are those that are created with PDS(s) {A, 0}, {A, 1}… {A, n}, where n is the last dimension. Comparing the two algorithms in this way may not be fair to them, but it will present us more meaningful guideline in choosing which algorithm to use in real environment.

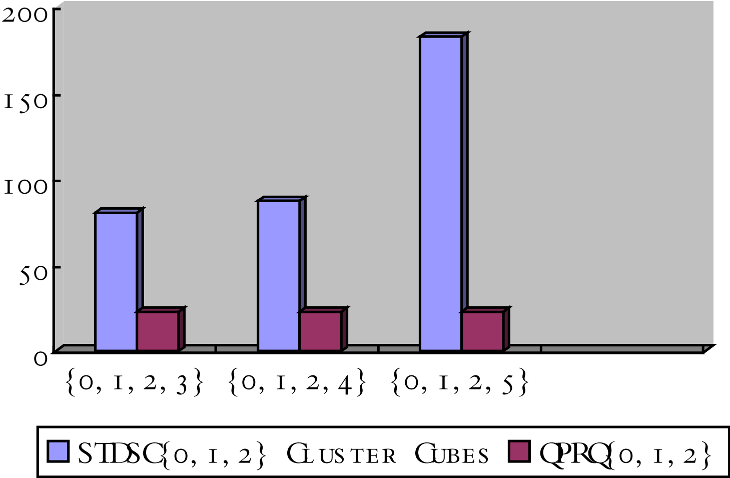
We first compare QPRQ with STDSC in cube construction. From Figure 8, we can see that QPRQ does not show consistent advantage over STDSC in comparing with its cluster cubes. For example, QPRQ with PDS {0, 1, 2, (8)} builds the cube faster than STDSC with PDS {0, 1, 2, 3}, but QPRQ with PDS {0, 1, 2, (8)} takes longer time than STDSC with PDS {0, 1, 2, 5}. Looking at the data in Table 4 and Table 5 carefully, we find out that in both of the algorithms the time used in building the cube really depends on the final cube size, which is consistent with our analysis in previous sections. Thus, we cannot say which one is better than the other in cube construction, but the difference in their cube construction performance is not big.

Figure 8   Comparison between STDSC and QPRQ in Cube Construction



In query processing, QPRQ performs much better on each cube than STDSC on its cluster cubes as shown in Figure 9. The performance difference is so big that we can even ignore their performance difference in cube construction. We can draw a conclusion here that, for every PDS we choose with STDSC we can always replace it by QPRQ with the corresponding PDS and get a much better query performance. For example, it is usually preferable to replace STDSC {0, 1, 2, 3}, {0, 1, 2, 4,} or {0, 1, 2, 5} with QPRQ {0, 1, 2}. If one finds QPRQ {0, 1, 2} too slow in cube building, instead of opting for STDSC {0, 1, 2}, one should drop the last dimension from the PDS, say QPRQ {0, 1}, which is again preferable to either STDSC {0, 1, 2}, or {0, 1, 5}.

Figure 9 Comparison between STDSC and QPRQ in Query Processing



7.3 MQPSB vs. STDSC

The comparison between MQPSB and STDSC is more straightforward than the comparison between QPRQ and STDSC. In the testing result, we can see that MQPSB is always outperforming STDSC in query processing.

In theory, in order to fully reveal the power of MQPSB, both the query point set and the B-tree should be large. A bigger query point set makes it possible to skip more query points, and a bigger B-tree means a bigger save on a skipped query point. In this thesis, a big B-tree is equivalent to a cubelet containing a big number of points/cells. We compare the two algorithms over 4 cubes, the results are presented in Table 7:

Table 7 Comparison between STDSC and MQPSB in Query Processing

PDS	MQPSB		STDSC	
	Time (s)	# of queried points	Time (s)	# of queried points
0,1,5	25.546	35890	50.413	3375180
0,1,2	7.981	13233	9.914	244800
1,2,3	25.136	116752	51.785	3616920
1,8	254.824	83471	257.330	810990

You may be surprised not to see any significant difference in query performance with such a huge difference in the number of queries presented above. However, let us take a second and look at the number and it will be more understandable.

First, query processing includes at least two types of costs: I/O cost and calculation cost. MQPSB focuses on the saving on calculation cost, not I/O cost. STDSC and MQPSB have the same I/O cost on the same cube because they read in the same set of cubelets for the same matrix query. The I/O cost takes a big portion of the whole cost in query processing, which makes the advantage of MQPSB much less obvious. In the case of {1, 8}, the file size is lot larger than other PDS(s). Therefore, the file I/O dominates the query processing time. The MQPSB does essentially the same amount of I/O as the STDSC. When the cubelet is small (meaning the B-tree is small) the MQPSB may skip some query points, but it can rarely skip a page, because most of cubelets span only one or two pages.

Second, the same point query costs differently on different cubelets. Searching a number within 10,000 numbers takes much longer than search the

number within 4 numbers, which means a query on a big cubelet/B-tree costs more than the query on a small cubelet. The numbers in the table above are counted without distinguishing between big cubelets and small cubelets. Let us take the cube {1 8} as example, around 13,000 cubelets in the cube have only one primary point/cell, and all 45,054 non-empty cubelets have less than 30 primary points/cells. Searching within such small cubelets/B-tree is trivial even between searching 100,000 times and searching 1000,000 times. On the other hand, we do not have compression on cubelets. The minimal size of a cubelet is the size of one page in memory, which is 4 KB in our case. Therefore, a cubelet with only one or two cells will take at least 4 KB on disk. Considering the huge number of cubelets in cube {1, 8} the total size of the cube is over 5 GB. Our heavy-duty query needs to access all cubelets, so the I/O cost dominates the query processing time. That explains why the performance of MQPSB and STDSC is so close even the number of query points issued in STDSC is almost 10 times of the number of query points in MQPSB.

One pattern we notice in the experiment is that picking the row/column dimension of a matrix query from the PDS of the cube will effectively weaken the advantage of MQPSB because it significantly reduces the size of query point set issued to each cubelet. For example, if we pick dimension 4, present-weather, as row dimension, and dimension 8, station-id, as column dimension in the heavy-duty matrix query, the size of query point set for each cubelet in cube {0, 1, 5} will be 101*85. However, if we choose the same row dimension and column dimension in cube {1, 4}, the number of query points for each cubelet will be only

85, which is not big enough to show the power of MQPSB. As we can expect, the two algorithms perform almost same with the matrix query on cube {1, 4}.

In conclusion, we should use MQPSB instead of STDSC with the same PDS and try to choose the PDS to make both query point set and cubelets/B-trees big enough to take the advantage of merging between query point and B-tree.

7.4 QPRQ vs. MQPSB

From the discussion above, we can see that both QPRQ and MQPSB perform better than STDSC. We want to see how they compare with each other. We will compare QPRQ with MQPSB the same way as we compare QPRQ with STDSC for the similar reasons.

From the testing results, we see a similar pattern as what we see in comparing QPRQ with STDSC. In addition, we find out, as the number of cubelets increases, MQPSB's query performance gets close to QPRQ very quickly. For example, when the number of cubelets is small, like 160 from PDS {0, 1, 2, (8)}, QPRQ is around 8 times faster than MQPSB. However, when the number is bigger than 4000, like PDS {0, 1, 2, 4}, the performance difference is already very small. Because of the limitation of the testing environment, we cannot do some testing on cubes with the number of cubelets bigger than 60,000. However, the pattern we observe from the analysis of experimental result is that QPRQ is a better choice when the number of cubelets is relatively small;

and MQPSB will outperform QPRQ when the number of cubelets is large enough.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

In this thesis, we propose a SplitCube approach for partial pre-aggregation. Compared to the full pre-aggregation, the storage footprint is reduced by 50% - 90% with the same query performance. Based on the framework of SplitCube, we then derive two enhanced query-processing techniques. To test and compare the performance of each technique in both cube construction and query processing, we design our experiment in such a way that the comparison is based on the same standard and the same set of aggregates. To make the results more meaningful for comparing between QPRQ and the other two algorithms, STDSC and QPRQ, we extend the comparison from on the same set of aggregates to within the same cluster cubes.

From the analysis of our testing results, we observe the following patterns that are useful in choosing the right algorithm in real applications:

1. QPRQ and MQPSB should always be picked as a better choice in cube construction and query processing.
2. In choosing between QPRQ and MQPSB, we need to take the number of cubelets into account. If the size of the cube is huge and a big number of cubelets is inevitable, we should consider MQPSB; otherwise, QPRQ should perform better than MQPSB in query processing.

The focus of our research in this thesis is on the query processing techniques. We analyze the factors in processing a query and building a cube as well. To improve the query performance we can either refine our query processing algorithms themselves or shorten cube construction time to make more room for query processing like what we do in QPRQ.

In our future research, we may design some more efficient query processing algorithms based on our findings in this thesis, i.e. combining MQPSB together with QPRQ. Furthermore, we can carry out our experiment on an online computation environment to get more accurate results than what is done in this thesis in a single-user development environment. Finally yet equally important, distributive aggregation algorithms are also a new horizon. As the multi-core CPU is becoming more common even among desktops and laptops, algorithms that may be easily distributed will perform much better than those that are not. According to the SplitCube approach, cubelets in a cube are independent from each other during cubelet construction and query processing. Once the PDS is determined, the processing on multiple cubelets can proceed in parallel.

REFERENCE LIST

[BPT97] E. Baralis, S. Parabolschi, E. Teniente, “Materialized Views Selection in a Multidimensional Database”, in Proc. of VLDB, 1997

[BR99] K. Beyer, and R. Ramakrishnan, “Bottom-Up Computation of Sparse and Iceberg CUBEs”, in Proc. of SIGMOD, 1999.

[DRNS98] P.M. Deshpande, K. Ramasamy, J. Naughton, A. Shukia, “Caching Multidimensional Queries Using Chunks”, In Proc. of ACM SIGMOD, 1998.

[DRT99] A. Datta, K. Ramamritham, H. Thomas, “Curio: A Novel Solution for Efficient Storage and Indexing in Data Warehouses”, Proc. of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[GBLP96] J. Gray, A. Bosworth, A. Layman, H. Prahesh, “Data Cube: A Relational Aggregation Operator Generalizing group-BY, Cross-Tabs, and Sub-Totals”, Proc. of ICDE ’96, New Orleans, February, 1996

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, J. Ullman, “Index Selection for OLAP”, In Proc. 13th ICDE, Manchester, UK, 1997

[GM99] H. Gupta, I.S. Mumick, Selection of Views to Materialize Under a Maintenance-Time Constraint, Proc. International Conf. on Database Theory, 1999.

[H98] J. Hellerstein, “Data Warehousing, Decision Support & OLAP”, http://redbook.cs.berkeley.edu/lec28.html

[HCKL00] E. Hung, D. Cheung, B. Kao, Y. Liang (2000) “An optimization problem in data cube system design”. Knowledge Discovery and Data Mining, Lecture Notes in Artificial Intelligence 1805, 74-85.

[HRU96] V. Harinarayan, A. Rajaraman, J. Ullmna, “Implementing Data Cubes Efficiently”, Proc. of ACM SIGMOD, 1996.

[HWL94] C. Hahn, S Warren, J. London, “Edited Synoptic Cloud Reports from Ships and Stations over the Globe”, 1982-1991

[IBM02] IBM Product Announcement: “DB2 OLAP Server V8.1 Delivers Relational Integration and Scalability”, April 2002

[KC02] H.-G. Kang, C.-W. Chung, “Exploting Versions for On-Line Data Warehouse Maintenance in MOLAP Servers”, Proc. of VLDB, Hong Kong, 2002

[KMP02] P. Kalnis, N. Mamoulis, D. Papadias, "View Selection Using Randomized Search", Data & Knowledge Engineering (DKE), 42(1), 89-111, 2002

[KP03] P. Kalnis, D. Papadias, "Multi-query Optimization for On-Line Analytical Processing", Information Systems, 28(5), 457-473, 2003.

[KR99] Y. Kotidies, N. Roussopoulos, “DynaMat: A Dynamic View Management System for Data Warehouses”, Proc. of ACM SIGMOD, Philadelphia, PA., 1999.

[Laks03] L. Lakshamanan, J. Pei, Y. Zhao, “QC-Trees: Efficient Summary Structure for Semantic OLAP”, Proc. of ACM SIGMOD, San Diego, 2003.

[Lee02] Y. Lee, K. Whang, Y. Moon, I. Song, “A One-Pass Aggregation Algorithm with the Optimal Buffer Size in Multidimensional OLAP”, Proc. of VLDB, 2002

[Liang00] W. Liang, M. E. Orlowska, J. X. Yu, “Optimizing multiple dimensional queries simultaneously in multidimensional databases”, The VLDB Journal, 2000

[Li03] C. Li, “A Partial Pre-Computation of Aggregates for OLAP Databases”, M.Sc. thesis, Simon Fraser University, July, 2003.

[Luk01] W. Luk, “ADODA: A Desktop Online Data Analyzer”, Proc. of DASFAA 2001, Hong Kong, 2002.

[LL03] W. Luk, C. Li, “A Partial Pre-Aggregation Scheme for HOLAP Engines”, In Proc. of DAWAK, Spain, 2003

[MI06] K. Morfonois, Y. Ioannidis, “CURE for Cubes: Cubing Using a ROLAP Engine”, Proc. of VLDB, Seoul, Korea, 2006

[MS03] Microsoft SQL Server 2000 Books Online, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/startsql/portal_7ap1.asp

[Pendse03] N. Pendse, “Datadase Explosion”, OLAP Reports, http://www.olapreport.com/DatabaseExplosion.htm

[PS99] P. Vassiliadis, T. Sellis, “A Survey of Logical Models for OLAP Databases”, SIGMOD Record, Vol. 28, No. 4, 1999

[Sara97] S. Srarwagi, “Indexing OLAP Data”, IEEE Bulletin of the Technical Committee on Data Engeering, Vol. 20, No. 1, 1997.

[SDN98] A. Shukla, P. Deshpande, J. Naughton, “Materialized View Selection for Multidimensional Datasets”, Proc. of the 24th VLDB Conf., New York, 1998

[SDNR96] A. Shukla, P. Deshpande, J. Naughton, K. Ramasamy, "Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies", Proc. of the 22th VLDB Conf., Bombay, 1996

[SRDK02] Y. Sismanis, N. Roussopoulos, A. Deligiannakis, Y. Kotidis, "Dwarf: Shrinking the Petacube", Proc. of ACM SIGMOD, Madison, 2002.

[SRDK03] Y. Sismanis, N. Roussopoulos, A. Deligiannakis, Y. Kotidis, "Hierarchical Dwarfs for the Rollup Cube", Proc. of DOLAP, November, 2003.

[Wit03] A. Witkowski et al, Spreadsheets in RDBMS for OLAP, Proc. of ACM SIGMOD, San Diego, 2003

[ZDN97] Y. Zhao, P.M. Deshpande, J.F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates", In Proc. of ACM SIGMOD, Tucson, Arizona, 1997

[ZDNS98] Y. Zhao, P.M. Deshpande, J.F. Naughton, A. Shukia, "Simultaneous Optimization and Evaluation of Multiple Dimensional Queries", In Proc. of ACM SIGMOD, 1998