OLAP Database Computation with a Splitcube in a Cluster

by

Yongping Zhang

B.Sc., Wuhan University, 1995

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

MASTER OF SCIENCE

In the
School
of
Computing Science

© Yongping Zhang 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:	Yongping Zhang
Degree:	Master of Science
Title of thesis:	OLAP Database Computation with a Splitcube in a Cluster
Examining Committee:	Gábor Tardos Chair
	Dr. Wo-Shun Luk Professor, Computing Science Simon Fraser University Senior Supervisor
	Dr. Ke Wang Professor, Computing Science Simon Fraser University Supervisor
	Dr. Jian Pei Associate Professor, Computing Science Simon Fraser University Examiner
Date Approved:	March 6, 2009

SFU 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

Software development on a cluster for data-intensive applications has always been a challenge. However, the cost advantage over traditional shared memory system has driven the migration of data warehouse to cluster. We propose splitcube - a new approach of OLAP database computation to work on cluster. Splitcube ensures very effective dynamic load balancing and low overhead. We study different ways of splitting the input data for parallel processing in an attempt to heuristically optimize the cost of processing queries for a specific workload at a prescribed level of pre-aggregation. Our results on two real-life datasets reveal great performance improvement in three-fold: 1) Both splitcube building time and query response time experience a near-linear speedup up to 64 processors; 2) The idle time in all but one instance is less than 6% of the total execution time; and 3) Splitcube achieves near-linear or better speedup with much larger datasets.

ACKNOWLEDGEMENT

First and foremost, I want to thank the School of Computing Science, who provided me with this opportunity to further my graduate studies while I'm also working in the School as a full time employee.

My special thanks go to my senior supervisor Wo-Shun Luk, who supplies me with endless inspiration, creative ideas and detail oriented advice. He helped me along every step of the way of my research and helped me finally reach the goal.

Ke Wang, my other supervisor, has been a source of inspiration as well. He always has a special way of looking at things that always leads to new ideas. Jian Pei, the examiner, has an insightful view of the topic and his questions opened my eyes to other research areas. Also, my thanks go to the Chair of my thesis defense, Gabor Tardos, who made the defense well-organized and efficient.

I want to thank all the staff of Computing Science, who support me with encouragement and keep my spirit high. Val Galat and Gerdi Snyder, who guided me through the process, I want to thank them both here.

Especially, sumo Kindersley, who kindly reviewed my thesis for its academic writing, I appreciate her time and energy spent on my thesis. With her help, my thesis looks ever so much better.

I also want to thank Martin Siegert in IT services, who maintains the cluster platform, for his invaluable help in working on the cluster and cleaning up my runaway programs.

Very importantly, I want to thank my wife Yingli Wang. Without her support and encouragement, I couldn’t have done it at all. While I spent hours of my spare time in debugging the program, she has always been there. And that’s where and how I gain strength and confidence to complete my graduate studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

A (computer) cluster is essentially a collection of computers, which are connected via a high-speed local area network, e.g., Ethernet. Cluster systems and symmetric multiprocessor (SMP) are among the most popular hardware platforms for high performance computing (HPC). Clusters are popular because they enjoy a 3-5 times cost/performance advantage over a symmetric multiprocessor (SMP) system, according to a recent industry study ([C08]). Architecturally, clusters scale better than SMP and are more fault tolerant. However, they have their own downsides in comparison to SMP:

- Lots of cooperating components
- Higher latency for sharing
- Software complexity

Traditionally, most applications developed on this HPC platform are devoted to scientific computation, as it is most able to take advantage of this architecture. In contrast, algorithms for data-intensive computation in general are inherently sequential, due to data sharing. When a task is partitioned into sub-tasks to run in different nodes in a cluster, these sub-tasks may need to re-synchronize and exchange data among some/all of the nodes, once in a while, before they can continue their execution. Consequently, it is hard to achieve good speedups.

This research focuses on one important data-intensive application: OLAP database computation, which includes pre-aggregation and query processing on OLAP databases. Our approach is quite unique in the literature on OLAP database computation in a multiprocessor environment. Most published papers on this subject are concerned with either pre-aggregation or query processing. Obviously, if all aggregates are pre-computed, query processing would be quite straightforward. We do believe in pre-aggregation, without which processing complex queries would take a long time. On the other hand, we are doubtful that there exists a cost-effective way of computing all aggregates in a cluster.

Let us begin by examining the literature on pre-aggregation in a single-CPU platform. There are basically two classes of cube building algorithms. Much of the earlier work is about computing cuboids, which can be characterized as materialized views of group-by queries. Much of the work in the literature on OLAP computation on a cluster, e.g., [GS97], [MK99], [NWY2001], [YJA02], [LU03] and [DER06], attempts to parallelize selective cube building algorithms. The more recent articles (e.g., [SRDK02], [LPZ03]) focus on minimizing the storage footprint of the data cube by removing cells in the cube that are identical. Elaborate storage structures are designed so that a network of pointers (real/symbolic) is built to link items which share the same information. These storage structures may not work well in a cluster, because it is hard to partition them and still retain much of the reduction in total storage footprint.

Computation of cuboids can be a sequential process. While a cuboid can be computed from a number of cuboids, it is best computed for one specific cuboid. This child-parent dependency hinders any data parallelism strategy which involves concurrent computation of a cuboid by multiple processors. Not only the parent cuboid needs to be distributed, but also the parts of the cuboid resulting from computation by different processors need to be merged together into a single child cuboid, when it can be used for computing another child cuboid of its own. One can avoid this dependency by adopting task parallelism, i.e., having only one processor computing a cuboid, as shown in [NWY2001]. The disadvantage of this approach is that the workload may be highly unbalanced, because of the uneven distribution of sizes of the cuboids. This problem may be partially alleviated by switching a task to a processor that is idle, but the task may end up in a processor without a matching parent. Finally, one may do away with dynamic load balancing entirely by carefully planning so that the processors will have roughly the same workload. This approach is advocated in [DER06], and produces an algorithm that achieves greater speedup than approaches that involve extensive data sharing. On the other hand, this approach necessitates careful analysis of the input dataset, and cost modelling with high accuracy, in order to achieve load balancing. It is remarked in [DER06] that "cost effective estimation … remains an important open area of research." Finally, task parallelism, as opposed to data parallelism, implies replication of source data in each processor or shared disks. The former results in lengthy data loading time, while the latter requires high shared I/O bandwidth. Moreover, each processor must process the entire dataset in the beginning, while processors in algorithms practicing data partitioning need to process only a fraction of the dataset.

In this research, we take a different approach to OLAP computation, because we stick to a set of assumptions that are different from the norm. We presume that decision making queries are much more complex than just group-by queries, firstly. For example, MDX, a popular industrial query language designed for spreadsheet users ([MS08]), produces matrices (or tables) as the answer to a query. The answer to an MDX query could come from a multitude of cuboids. Computing only and all cuboids does not necessarily lead to fast query processing for any user-defined workload that consists of query matrices. Secondly, we do not insist on pre-computing all aggregates. A user may not opt for pre-computing all aggregates because the huge size of the cube, as a result of data explosion [P2005]. Even if the user is willing to put up with the time required to build the full cube in an attempt to eliminate the need for aggregation during the query time, it may not always pay off due to a large increase in I/O time. OLAP products, such as Microsoft Analysis Services, regularly analyze the OLAP metadata model and heuristically determines the optimum set of aggregations from which all other aggregations can be derived [MS01]. Un-pre-computed aggregates that are required by a query is computed on-the-fly. By not insisting on computing only cuboids, or all aggregates, we demonstrate in this thesis that OLAP database computation can indeed work well with a cluster as a hardware platform.

We propose to compute and query a *splitcube*, which contains only a subset of all aggregate cells. The splitcube consists of a number of *cubelets*, which themselves are lower dimensional cubes. Central to the idea of splitcube is that these cubelets may be computed in isolation once the input dataset has been partitioned. In this sense, data parallelism is practiced. We call the construction of a cubelet a *task.* In the absence of affinity between a proposing node and a task, tasks may be assigned to any processing node during the runtime, in order to achieve load balancing. The advantage of our approach, compared to other existing approaches, is that we practice both data and task parallelism, and manage to avoid their pitfalls.

Given sufficient number of tasks, it is easy for a task dispatcher to keep all the processors busy all the time. On the other hand, excessively large number of tasks leads to proportional large overhead, and more importantly, unacceptably long query response time due to huge on-the-fly aggregation during query processing. The main focus of this research is to develop an algorithm to heuristically optimize the query response time by adjusting the number and size of the tasks. Since query response time in general is dependent on the amount of pre-aggregation, our goal is to find an efficient, low-cost way to partition the input dataset such that an optimal, or near optimal, query response time for a given level of pre-aggregation is achieved.

The rest of the thesis is organized as follows. In Section 2, notations and general concepts about OLAP systems are explained. Section 3 contains an introduction of splitcube. In Section 4, a scheme for building a splitcube and query processing is presented. The procedure to locate an optimal PDS for each multiplier class is explained in detail. In Section 5 we analyze the experimental results on two real-life data sets: the Weather dataset (enhanced), and a real-estate dataset. Section 6 is the concluding section.

2 BASIC CONCEPTS

2.1 Cube and Dimension Hierarchy

The (full) *cube* is defined in this thesis to be a k-dimensional array, where k is a positive integer greater than zero. Each dimension of a cube has D_i members, 1<=i<=k, which are organized as a *hierarchy*. The members at the leaf level are called *primary* members. All other members in a higher level 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.

As an example, consider a 3-dimensional OLAP database. Fig. 1 shows dimension hierarchies of *A*, *B* and *C*. All members at the bottom level are primary members (in shade), and the remaining ones are group members.

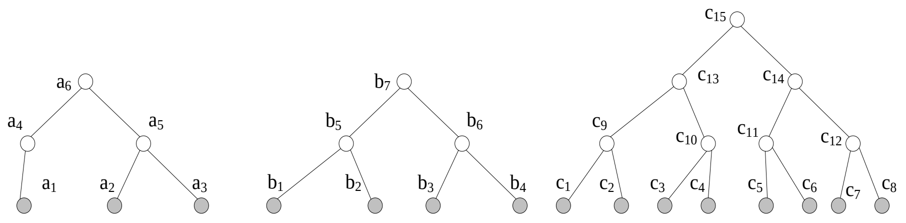

Figure 1: Dimension Hierarchies *A*, *B* and *C*

A cell in the cube has two components: the address in the cube and the measure. It 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 called the *measure*, although the results of this thesis are equally valid for multiple values stored in each cell. Measures of all primary cells are input from a data source(s). The measure of a group cell may be calculated according to the method to be discussed in Section 2.2. If the cube includes all group cells, it is called a *fully pre-aggregated* cube, or otherwise, a *partially pre-aggregated* cube. The set of all primary cells is called a *base cube*. As examples, (a_1, b_2, c_8; 4) and (a_1, b_5, c_{10}; 15) are cells of the 3-dimensional cube shown above. The former cell is a primary cell, and its measure is an input value; while the latter is a group cell with a measure to be derived from the measures of some primary cells. This measure may be pre-computed during the pre-aggregation phase, or computed on-the-fly during the query time.

2.2 Aggregation

We now consider how the measure of a group cell is derived. To this end, we need to elaborate on the relationships between cells in a cube.

A *descendant* of a cell, T, with coordinate (t_1, …, t_k) is another cell, T', with coordinate (t_1', …, t_k') such that each t_i' is a descendant of t_i in the i^th dimension hierarchy. The distance between T and T' 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, T' is an *immediate child* of T, if the distance between them is 1. If t_i is a group member, the immediate children of T *along the dimension i* are those cells which have identical components as T, except for the i^{th} dimension. There are as many sets of immediate children of T as there are group members included in the coordinate of T. The root of the measure tree is one that has no parent. The coordinate of the root consists of roots of all respective hierarchies.

Consider (a_5, b_7, c_{12}) as the address of a group cell, which has in its coordinates a_5, b_7 and c_{12} as group members. Thus, it has a set of immediate children along dimension *A*, i.e., (a_2, b_7, c_{12}) and (a_3, b_7, c_{12}), a set of immediate children along dimension *B*, i.e., (a_5, b_5, c_{12}) and (a_5, b_6, c_{12}), and another set of immediate children along dimension *C*, i.e., (a_5, b_7, c_7) and (a_5, b_7, c_8). It can be shown that the measure of (a_5, b_7, c_{12}) is the sum of measures of the set of immediate children along *any one of the dimensions* [Luk01].

We define the *measure* of a group cell T to be a distributive aggregate function, according to ([GBLP96]), of all measures of all primary cells that are also descendants of T. The definition of a distributive aggregate function is given in [GBLP96]; however, since it applies to only 2-level hierarchies, a more precise definition is required for this thesis. We say an aggregation function *F*() is a distributive one, 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*() functions. *Count*() is also a distributive aggregate function if we choose *G* to be the summation aggregation function. For this thesis, we use only the summation aggregation function, but our results are equally applicable to all other distributive aggregation functions.

2.3 Physical Representation of a Cube

A cube is often considered as a logical view of the OLAP database. A schema of an OLAP database consists of a fact table, and a number of dimension tables. The fact table contains all primary cells, i.e., cells in the base cube, while the dimensional tables store information about each dimension. Our OLAP engine keeps the dimensional tables as relations, and organizes the cells in the cube, including primary and group cells, into a B-tree[1]. In particular, an element in the B-tree is a cell record, which consists of the key and the associated measures. The key of a cell is the coordinate of the cell. In our implementation here, it is compressed into a 64-bit integer. The B-tree is a clustered one, the tuples in the leaf nodes having been sorted according to the descending sequence of the B-keys.

[1] In this thesis, the B-tree is synonymous to the B+-tree where all leaf nodes are of equal distance to the root nodes.

2.4 Aggregation Algorithm

We consider here only the cube building algorithms that run on a single computer. In fact, the kind of cube subject to the algorithm is typically much smaller than the original cube, and with fewer dimensions involved. Thus we need an algorithm that does a superb job when the cube is small enough to fit into the memory, while it does just as well as other published cube building algorithms for large cubes. Besides, the end result of the algorithm is a B-tree, as described in Section 2.3. For this purpose we choose the disk-based version of the algorithm published in [Luk01]. We do not elaborate further on the details of this algorithm which have been included in another manuscript [Luk08], as it is not central to this research.

3 SPLITCUBE – GENERAL CONCEPTS

In this section, we introduce the general concepts of a splitcube, which could be considered as a partially pre-aggregated (PPA) cube. We explain the concepts in the context of a single-computer system. These concepts will then be applied to the OLAP engine on a cluster in Section 5.

3.1 Prefix and Cubelet

Central to the whole idea of splitcube is the partition of the *k* dimensions into two sets: *prefix dimension set*, or PDS, and *cubelet dimension set*, or CDS. For convenience, we consider the first *m* dimensions to form the PDS, i.e., P_1,…, and P_m, where $0 < m < k$. The *prefix* of a cell address is an m-tuple, which is the projection of its coordinates on the PDS.

Given a PDS, a *Splitcube* is defined as a collection of all the cells in the cube, except those cells whose prefixes contains at least one group member of a prefix dimension. Thus, the cells in the splitcube may be partitioned into a number of sets, each of which consists of cells with the same prefix. Each such set is called a *prefix set*, which may be represented by a prefix and a *cubelet*. A cubelet associated with a prefix is itself a (k-m)-dimensional cube. There is a 1-1 correspondence between the cells in prefix set and those in the cubelet. Each cell in a cubelet has the same measure as the corresponding cell in the prefix set, and an address is a (k-m)-tuple which is the projection of a cell address on the CDS. The set of primary cells in this cubelet is the *base cubelet*. Thus the set of base cubelets is a partitioning of the base cube.

The primary purpose of introducing the concept of splitcube is that cubelets can be constructed from their base cubelets, so that constructions of these cubelets can proceed in parallel. This is due to the following proposition, whose proof is omitted, because it follows directly from the discussion of aggregation in Section 2.2.

Proposition 1: The cubelet associated with a prefix set of a splitcube is identical to the (k-m)-dimensional cube constructed from the base cubelet associated with the prefix set.

3.2 Construction of a Cubelet

Computation of a cubelet associated with a specific prefix from the base cube may proceed as follows:

i. Locate the cells in the base cube, i.e., primary cells, with the same prefix.
ii. Project these cells on the CDS and the measure, which form the base cubelet.

iii. Compute the fully aggregated (k-m)-dimensional cube with this base cubelet.

Let us now consider our running example again. Assume that the base cube consists of the following cells, the scalar being the lone measure: (a_1, b_2, c_1; 3), (a_3, b_3, c_3; 4) and (a_3, b_4, c_2; 2).

Consider the splitcube, SC_A, where *A* is the sole prefix dimension in the PDS, and *BC* are the cubelet dimensions in the CDS. The splitcube has only two cubelets, associated with a_1 and a_3 respectively. Projecting the cells in the base cube associated with a_1 on *BC* and the measure, we have only one cell (b_2, c_1; 3) in the base cubelet associated with a_1, while the base cubelet associated with a_3 has two cells, (b_3, c_3; 4) and (b_4, c_2; 2). The cubelet generated from the base cubelet associated with a_1 is shown in Table 1. Note that there are 12 cells in the cubelet, which is the product of the levels of dimension hierarchies *B* and *C*. As a result, this quantity is called the *multiplier* of the cubelet associated with the PDS {*A*}.

Prefix	Cubelet
(a_1)	(b_2, c_1; 3), (b_2, c_9; 3), (b_2, c_{13}; 3), (b_2, c_{15}; 3), (b_5, c_1; 3), (b_5, c_9; 3), (b_5, c_{13}; 3), (b_5, c_{15}; 3), (b_7, c_1; 3), (b_7, c_9; 3), (b_7, c_{13}; 3), (b_7, c_{15}; 3)
(a_3)	… …

Table 1: Prefix & Cubelets with A as PDS

As another example, the prefixes and cubelets for the splitcube, SC_{AB}, are shown in Table 2. The multiplier for the PDS *AB* is 4, which is the number of levels in dimension *C*.

Prefix	Cubelet
(a_1, b_2)	$(c_1; 3), (c_9; 3), (c_{13}; 3), (c_{15}; 3)$
(a_3, b_3)	$(c_3; 4), (c_{10}; 4), (c_{13}; 4), (c_{15}; 4)$
(a_3, b_4)	$(c_2; 2), (c_9; 2), (c_{13}; 2), (c_{15}; 2)$

Table 2: Prefix & Cubelets with AB as PDS

3.3 Query Types – Point Query and Query Matrix

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 Q $(q_1, \ldots, q_k)$ where q_i, $1<=i<=k$, is the coordinate of the cell in the ith dimension. Nonetheless, there are problems if one is concerned with performance of only point queries. First, random point queries rarely retrieve non-empty cells, because most cubes are sparse. Secondly, decision making queries tend to be more complicated, and take a lot longer to execute. Thus good performance in point queries is necessary but not sufficient. Consequently, we consider another type of query which is more representative of the practical queries.

A possible candidate is range query which is sometimes included in performance evaluation in some papers (e.g., [LPZ03]). A random range query does not make much sense in OLAP applications, when the ranges have already been carefully defined by OLAP application designers, in the form of dimension hierarchies. Instead, most commercial OLAP systems implement a query language called MDX ([MS08]). Here, we adopt a simplified form of MDX, i.e., query matrix, which is designed for inter-row/column calculations on a spreadsheet [Wit03].

A *query matrix* displays data contained in the cube 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 Q (q_1, …, q_k) with at least two group members, and two dimensions, say i and j, identified as the row and column dimensions respectively. q_i and q_j must be group members for their respective dimensions, with immediate descendant members $q_{i,1}$, …, $q_{i,r}$, and $q_{j,1}$, … $q_{j,s}$. The answer for the query matrix, i.e., the *answer matrix,* is a table, with $q_{i,1}$, … and $q_{i,r}$ as labels for the row, and $q_{j,1}$, … and $q_{j,s}$ for the column. Assuming i < j, the entry (v,w) of the table, where 1<=v<=i and 1<=j<=s, is the answer for the query (q_1, …, q_{i-1}, $q_{i,v}$, q_{i+1}, …q_{j-1}, $q_{j,w}$, q_{j+1}, …, q_k).

In the example we see in Fig. 1, we assume a query matrix for our 3-dimensional cube has the point query (a_6, b_7, c_{15}), with *A* as the row dimension and *B*, the column dimension, consists of a matrix of point queries as shown below:

Row\Column labels	b_5	b_6
a_4	(a_4, b_5, c_{15})	(a_4, b_6, c_{15})
a_5	(a_5, b_5, c_{15})	(a_5, b_6, c_{15})

Table 3: Query Matrix with A and B as Row and Column Dimensions

3.4 Processing Point Queries

Consider the point query, $(q_1, \ldots, q_k)$, which is *split* into two parts: $(q_1, \ldots, q_m)$ and $(q_{m+1}, \ldots, q_k)$, where each q can be any member in the dimension. Processing this point query on a splitcube may proceed as follows:

1. Decompose $(q_1, \ldots, q_m)$ into a number of prefixes, $(p_1, \ldots, p_m)$, where p_i is a primary descendant of q_i, $1 <= i <= m$.
2. For each of these prefixes, retrieve the measure of the cell, $(q_{m+1}, \ldots, q_k)$ in the associated cubelet.
3. Compute the answer of the query from the measures retrieved.

For example, consider the splitcube SC_A. A query (a_6, b_7, c_{15}), is decomposed into three queries (a_1, b_7, c_{15}), (a_2, b_7, c_{15}), and (a_3, b_7, c_{15}), since a_6 has 3 possible primary members, i.e., a_1, a_2, and a_3. Since the cubelet with prefix a_2 is non-existent, the projected query (b_7, c_{15}) is applied against the two associated cubelets, which retrieves two cells, $(b_7, c_{15}; 7)$ and $(b_7, c_{15}; 6)$. The answer to the query (a_6, b_7, c_{15}) is computed from the measures of these two cells.

3.5 Processing Query Matrices

Generally speaking, processing of a query matrix may proceed by processing point queries individually inside the matrix. To process the query matrix in Table 3, we apply the point queries, one at a time, to the splitcube SC_{AB}(as defined in page 14). Actually, we can do better. Observe that the first coordinate of the query matrix, a_6, is a group member of the dimension *A*, with a_1, a_2, a_3 as its primary descendants. Similarly, the second coordinate of the query matrix, b_7, is a group member of the dimension *B*, with b_1, b_2, b_3, b_4 as its primary descendants. We simply apply the point queries (c_{15}) against the cubelets associated with (a_i, b_j), 1<= i <= 3 and 1<= j <= 4. The results from individual cubelets are then posted to the answer matrix.

This procedure may be a bit more complicated if the query matrix in Table 3 has *C*, instead of *B*, chosen as the column dimension, which is not a dimension included in the PDS. The query matrix will now be:

Row\Column labels	c_{13}	c_{14}
a_4	(a_4, b_7, c_{13})	(a_4, b_7, c_{14})
a_5	(a_5, b_7, c_{13})	(a_5, b_7, c_{14})

Table 4: Query Matrix with A and C as Row and Column Dimensions

In this case, we need to apply two point queries, (c_{13}) and (c_{14}) against the cubelets associated with (a_i, b_j), 1<= i <= 3 and 1<= j <= 4. In other words, the number of point queries applied is doubled.

Consider an arbitrary query matrix, $(q_1, \ldots, q_k)$ with D_i and D_j as the row and column dimensions respectively. Furthermore, q_i and q_j have respectively r and s immediate descendant members. As before, the PDS is $\{D_1, \ldots, D_m\}$. The total number of point queries in the matrix is r*s. But the actual number of point queries posed could be much higher, because some of these point queries must be decomposed into a number of point queries with pre-computed answers. To compute the number of point queries that must be posed to compute the answer of a given query matrix, we need to consider the following cases:

(i) D_i and D_j are both included in PDS: 1 single point query against every cubelet.

(ii) Only D_i (D_j) is included in PDS: r (s) point queries against every cubelet.

(iii) Neither D_i nor D_j is included in PDS: r*s point queries against every cubelet.

Other factors being equal, dimensions that are prone to detailed analysis should be assigned as prefix dimensions.

4 CONSTRUCTION AND QUERYING OF SPLITCUBES IN A CLUSTER

Having described how splitcubes are built and queried, we now consider these processes in a distributed system, i.e., a cluster. A generic architecture of a cluster is shown in the following diagram, where each node is in essence a PC, with its own memory, and local disk drive(s). These PCs are assumed to possess similar processing capability and they communicate with other PCs over an Ethernet network. Where necessary, they may read from a shared storage, but the Ethernet network is the preferred communication media among the nodes because the communications speed over the Ethernet is 3 times faster than that via the shared storage.

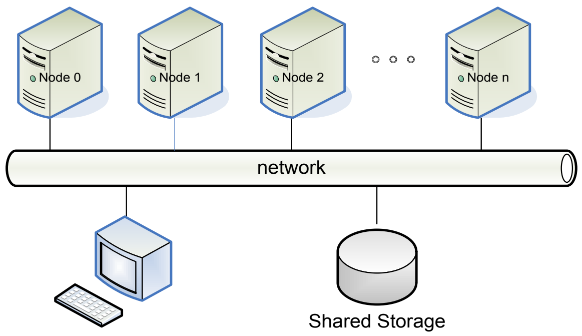

Figure 2: A Cluster of PC's

We begin with a description of how to build a splitcube in a cluster by distributing the work among the nodes, assuming the choice of a PDS, which is followed by a description of how a query matrix may be computed over the cluster. The rest of this section is devoted to the search for an optimal PDS, or PDS's.

4.1 Construction of Splitcubes

The construction of a splitcube is a two-phase process: planning and executing. In the planning stage, one processor node is designated as the Coordinator. The Coordinator has the following responsibilities: parse the input dataset, which is the base cube, selection of a PDS, and partitioning of the base cube into a number of base cubelets associated with PDS. Once the PDS is chosen, the input dataset is partitioned into *n* subsets, where *n* is the number of prefixes associated with the PDS and the input dataset. Each subset is a base cubelet by itself. A *task* is defined as the construction of a cubelet. Associated with a task is a *task control block*, which contains the base cubelet and its associated metadata.

For the executing phase, we adopt the approach of dynamic load balancing for carrying out all the tasks defined in the planning phase. Each task is assigned to a processor node, called *worker*. Here, we assume the number of tasks must be larger than the number of workers. The job of a worker is fairly simple. As it becomes idle, and it is idle initially, it sends a message to the Coordinator. Once a task control block is received, it constructs the cubelet, which is stored on the local disk. The Coordinator allocates task control blocks associated with outstanding tasks to workers on the waiting list on a first-come-first-serve basis, with no attempt to distinguish one worker from the others. On the other hand, we follow a load balancing strategy to determine the order by which the tasks are assigned. This strategy is necessary since the sizes of tasks may vary greatly in accordance with the sizes of the base cubelets.

Some load balancing strategies are inappropriate and should be avoided. For example, if many small tasks are assigned within a very short time, or worse, consecutively, the Coordinator will be kept so busy that it cannot monitor network traffic successfully, resulting in packet losses. At the same time, bursts of network traffic will cause packet collisions, resulting in re-transmission of packets. As another example, large tasks should not be distributed toward the end of the execution phase, because the Coordinator may run out of tasks for distribution while some large tasks are being executed by only a few nodes. In the worst case, all but one Worker could be idle, waiting for one Worker which has just started a very large task. Consequently, the entire execution phase is delayed. One reasonable strategy, which we have adopted, would be to assign large and small tasks in an alternating fashion.

4.2 Querying Splitcubes

Our query processing algorithm is simple, because we depend on the raw power of the workers to deliver the performance. Sophistication in a query processing algorithm may result in certain amount of overhead. Upon receipt of the query, i.e., a query matrix, the Coordinator decomposes the query into a number of point queries, to which the answers are readily available on the fully pre-aggregated cubelets. The query decomposition is done as described in Section 3.4. These point queries are delivered as a bundle to the relevant Workers. Each query, as well as its answer, is accompanied with a tag, indicating the row-column position of the query within the query matrix.

Each worker has a simple query processor to process the point queries that fall within the ranges of the cubelets that are local to the worker. The query processor has two main functions. It ensures all queries assigned to it are posed to the local cubelets according to the sorted sequence of the queries. As the answers of the queries are returned by the local B-tree(s), the worker classifies these answers into groups by the tags of the associated queries, and calculates the sum of answers within the same group. The results are then sent back to the Coordinator for the final tally before returning the answer to the user.

4.3 Optimal Choice of PDS

The quest for an optimal choice of PDS is complicated by the fact that there can be more than one optimal PDS. Since we do not insist on generating all aggregates, there is always the trade-off between the cost to construct the splitcube and the query processing cost for a given query workload. Our strategy here is to partition the PDS's into a number of classes, such that PDS's in each class have similar characteristics, which lead one to believe that they incur similar cost of constructing a splitcube. We then proceed to locate a PDS within each class that produces the lowest processing cost. Note that we are concerned only with the cost of building a splitcube as if there is only one processor. We reason that with an effective load balancing scheme, each worker is kept busy doing useful work. Minimizing the total work to be done translates into minimizing the work to be done by each worker, hence the total elapsed time in a cluster.

Our optimization strategy does not explicitly take into account on the number of workers. As a result, when more workers are introduced, the work will be spread out more, consequently the shorter computation time is. It is important to note that based on workers' computational power, sufficient work should be given to each worker to ensure the speed-up.

The principles for PDS selection are not unlike those for query optimization in a relational DBMS. It must be efficient, and yet avoid very bad choices. Consequently, we can't afford to evaluate each of the 2^k possible PDS's, where k is the number of dimensions. We eliminate sub-optimal PDS's in three phases, so that the end result is a single PDS.

Phase I: We define a quantity called MaxCubeletCount, which is equal to $|D_1|*\ldots*|D_m|$, which is the maximum of the cubelets there can be. All PDS's whose MaxCubeletCount is too large or too small are discarded. We rank all remaining PDS's in each class by their 'maximum' query processing costs (to be defined later) in ascending order, and discard all but the top 3 PDS's. Since the goal of this phase is to quickly rule out a huge chunk of PDS's for consideration in Phase II, no reference is made to the base cube (the input dataset).

Phase II: For each of remaining PDS's, we calculate values of two parameters: the CubeletCount and MaxBaseCubeletSize against the base cube. The former is the actual number of cubelets entailed by the PDS, while the latter is the maximum size of all base cubelets. PDS's with large MaxBaseCubeletSize values are removed, because they cause unbalanced task distributions. With a more realistic estimate of CubeletCount, we re-compute the query processing costs for all remaining PDS. We select PDS with the lowest query processing cost for each class.

Phase III: Eliminate any selected PDS from Phase II, if it has a higher cost for building the splitcube as well as higher query processing cost than another selected PDS. If there are more than one class left after the elimination, the system or the user can choose one of them based some other criteria, e.g., the one with lowest cost in constructing the splitcube (i.e., the PDS with the lowest multiplier) or the one with lowest query processing cost.

An example of how the PDS's are eliminated in a real-life dataset is shown in Section 5.2.2.

4.4 Multiplier Class

Every cell in the base cubelet contributes to the creation of a number of new group cells in the cubelet. This number is called the *Multiplier*, which can be calculated as follows. Let $\{C_1, \ldots, C_j\}$ be the CDS associated with the PDS, and l_i, 1<=i<=j, is the number of levels in the hierarchy associated with the dimension C_i. Thus, the multiplier associated with PDS is the product of $l_1, \ldots, l_j$. If there are n cells in the base cube, the maximum number of (primary and group) cells in the splitcube is n times the multiplier of the PDS. Thus, the multiplier is indicative of the work involved in constructing the splitcube. A multiplier class is the set of all PDS's with the same multiplier, and it is labeled by the value of the multiplier.

In Section 3.2, we have showed that the multiplier value of PDS $\{A\}$ is 12. The following table shows the multiplier classes of all PDS's.

Multiplier Class	CDS
3 (# levels of A, or B)	A [PDS {B, C}], B [PDS {A, C}]
4 (# level of C)	C [PDS {A, B}]
9 (product of # levels of A and B)	AB [PDS {C}]
12 (product of #levels of A/B and C)	BC [PDS {A}], AC [PDS {B}]

Table 5: Multiplier Classes and PDS's

Note that the actual number of cells in the splitcube may not fully reflect this multiplier effect, because some of group cells generated by different cells in the base cubelet may be merged because they have the same coordinates.

4.5 Query Processing Cost

When we consider the query processing performance, it is important to include a discussion of query workload. In practice, the query workload usually consists of a set of query matrices which users have identified to be relevant to them. Therefore, it makes sense to choose a PDS so that the queries in the workload can be processed efficiently. In this thesis, we choose the workload of the most computationally intense query matrices. They have the same point query Q (q_1, …, q_k), where each q is the root of a dimension hierarchy. Thus, there are k*(k+1)/2 queries in our workload. We measure the query processing performance of a PDS in two ways: by the highest possible query processing cost of query matrices and the average cost over all queries within the workload. We choose the former as the primary measure in order to demonstrate the speedup of the query performance as the number of processors increases.

We measure the cost of processing a given query matrix by the number of point queries that must be posed to the cubelets to fetch the answer of the query. This cost is calculated to be the product of the cost per cubelet and the actual number of cubelets. We have derived in Section 3.5 the cost per cubelet. The number of cubelets for the splitcube, which is CubeletCount, is dependent on the input dataset. In the absence of any reference to the input dataset, as is the case in the Phase I of the planning process, one may derive the 'maximum' query cost by estimating MaxCubeletCount, which is the product of $|D_1|$, $|D_2|$, …, and $|D_m|$. Note that while the gap between MaxCubeletCount and the actual one is small for PDS's with small multipliers, it is widened exponentially as the value of m increases due to data skewness. Hence we need to compute a better estimate of the CubeletCount for selective PDS's.

4.6 Computing CubeletCount and MaxBaseCubeletSize

The exact values of CubeletCount and MaxBaseCubeletSize, both of which are required for Phase II, can be derived by scanning the input dataset. If the input dataset is huge, two techniques are employed to speed up the computation of CubeletCount and MaxBaseCubeletSize of a PDS: distributed computation and sampling.

Prior to the planning process of query processing, the input dataset is assumed to be stored in a compressed form in the network drive. In Phase II, the coordinator assigns a number of nodes as *evaluators*; each evaluator is to compute the CubeletCount and MaxBaseCubeletSize of a specific PDS. Once that is done, the estimates of two parameters are sent to the coordinator. Due to limited bandwidth of the network drive (i.e., the shared drive shown in Fig. 4), one should not attempt to employ a large number of evaluators (e.g., > 30) to work in parallel.

The computation by the evaluator is a simple process. It involves reading from the network drive a fraction of the input dataset, and computing the values of the parameters based on the sample data. For our purpose, the actual value of MaxBaseCubeletSize is of importance, because PDS’s with large MaxBaseCubeletSize are eliminated, whereas CubeletCount is used exclusively for *ranking PDS’s*, within the same multiplier class. Consequently, the value of MaxBaseCubeletSize is projected from the sample-based value. If y% of the dataset is actually read, then the sample-based MaxBaseCubeletSize value will be multiplied by a factor of 100/y. On the other hand, no adjustment is made to the sample-based CubeletCount value. While sophisticated statistical sampling techniques may yield more accurate results, we don’t use them for this study because the more sophisticated techniques is almost certainly take more time.

5 EXPERIMENTAL EVALUATION AND ANALYSIS

A number of important issues are addressed in this section on the experimental basis:

- *Scalability*: As the number of processors increases, or as the size of the input dataset increases, how much faster will the cube building and query processing algorithms run?
- *Modeling (query performance)*: On each dataset, our optimization process selects a PDS for each multiplier class, which is supposed to lead to the best query performance, among the PDS's in the multiplier class. How well does our optimization work?
- *Modeling (multiplier class)*: Is the multiplier class a good indicator of the degree of pre-aggregation? More specifically, does the PDS we select for each multiplier class provide a better performance in pre-aggregation the PDS chosen for the class with a smaller multiplier?
- *Sampling accuracy*: Firstly, is sampling necessary, i.e., what would happen if the theoretical values for CubeletCount and MaxBaseCubeletSize are used, without resorting to sampling? Secondly, how well do the estimated values of these parameters serve our optimization strategy?

The organization of the remaining section is as follows. A brief description of the hardware platform can be found in Section 5.1. Then we address the above questions by examining the experimental data related to the Weather and Real Estate datasets (Section 5.2 and 5.3) respectively.

5.1 Experimental Hardware/Software Platform

We have implemented our algorithm on SFU's Beowulf cluster, which largely resembles the configuration shown in Fig. 2. Dedicated exclusively to research computing, especially scientific research computing, it contains 96 nodes, each of which is a PC equipped with dual Athlon MP 2800+ processors (2.133 GHz), 1GB RAM, 15GB hard disk and fast Ethernet interface. The network connection between the nodes is 3-way channel bonded fast Ethernet, providing a 3 times larger bandwidth than the ordinary Ethernet. In addition, each PC connects to the shared storage on the Ethernet. The programs, written in C++, are compiled and run on the local Linux OS. Program-to-program communications are conducted via MPI (Message Passing Interface, Version 2.0), which is a popular language-independent API (application programming interface) with defined semantics and flexible interpretations.

5.2 Weather Dataset (enhanced)

5.2.1 About the dataset

The Weather dataset ([HWL94]) is a very common one for research on cube building algorithm to adopt for experimental analysis. The original dataset has 9 dimensions and one measure; the dimensions' cardinalities are: 2, 8, 10, 30, 101, 152, 179, 352, and 7037. All dimensional hierarchies are 2-level ones, i.e., the only group members are roots of the hierarchies. We bulk up the hierarchies of the last 4 dimensions by adding one more level in each dimension hierarchy. The cardinalities of all dimensions are shown in the following table:

level\dimension	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	2	8	10	30	101	8	8	18	85
3						152	179	352	7037

Table 6: Cardinalities by Levels in the Dimension Hierarchies in the Enhanced Weather Dataset

There are about 1 million tuples in the dataset, i.e., about 1 million primary cells in the base cube. If all group cells were to be pre-computed, the size of cube would swell to 14.7 GB.

5.2.2 Derivation of PDS

In Phase I, we eliminate the PDS's whose MaxCubeletCount is less than 64, the number of workers, or greater than 1 million, the number of cells in the base cube. As a result, we are left with only 9 multiplier classes and 18 PDS's to work with in Phase II. In Phase II, we compute an estimate of each MaxBaseCubeletSize and CubeletCount. Detailed computation is described in Section 5.2.6. (In Table 7, only their actual values are shown.) With the estimate of CubeletCount, we now derive a more accurate estimate of QueryCost. Those PDS's whose MaxBaseCubeletSize is greater 20,000 are removed from our list. At the end of Phase II, only 8 PDS's remain, one for each multiplier class. During Phase III, the PDS#160 is eliminated because both the cost of computing the splitcube and its query cost are greater than those of PDS#67, the choice of Class 216.

PDS #	Multiplier	Max Cubelet Count	Max Query Cost	MaxBase Cubelet Size	Cubelet Count	Query Cost	Phase II	Phase III
17	648	202	515100	464270	202	515100	deleted	
10	648	240	2060400	5633	240	2060400	selected	selected
18	648	808	2060400	100849	787	2006850	deleted	
12	432	300	2575500	32320	300	2575500	deleted	
65	432	374	3210790	17793	259	2223515	selected	selected
13	432	600	5151000	29810	597	5125245	deleted	
7	324	160	1373600	121994	160	1373600	deleted	
11	324	480	4120800	5282	480	4120800	selected	selected
19	324	1616	4120800	81888	1459	3720450	deleted	
160	288	59200	5.08E+08	1826	5204	44676340	selected	deleted
28	256	30300	46359000	21969	7009	10723770	deleted	
35	216	2560	21977600	4706	2206	18938510	deleted	
67	216	2992	25686320	4251	1971	16921035	selected	selected
69	216	3740	32107900	15483	2212	18990020	deleted	
23	162	16160	41208000	77658	4105	10467750	deleted	
15	162	4800	41208000	4680	4192	35988320	deleted	
27	162	48480	74174400	3250	22089	33796170	selected	selected
39	108	25600	2.2E+08	4111	11668	1E+08	deleted	
71	108	29920	2.57E+08	3828	11134	95585390	deleted	
53	108	323200	8.24E+08	14506	19970	50923500	selected	selected
31	81	484800	7.42E+08	3072	37909	58000770	selected	selected

Table 7: PDS Selection - Weather Dataset

5.2.3 Pre-aggregation Performance

We consider here the pre-aggregation performance against the number of processors and the multiplier classes.

Associated with a given PDS, let us define T(i) to be the execution time when the number of workers (i.e., participating processing nodes) is i, i>=1. Traditionally, the speedup of a parallel algorithm when i = n is defined to be the ratio of T(1)/T(n). If the speedup is n for n workers, the speedup is said to be linear. For the sake of brevity, we start with i = 8, in order to consider the speedup of our pre-aggregation scheme at n = 8, 16, 32, and 64.

The chart in Fig. 3 shows the speedups of our scheme for each multiplier class. A speedup line graph for the chosen PDS in each class is the path connecting the 4 points, i.e., (8, T(8)/T(8)), (16, T(8)/T(16)), (32, T(8)/T(32)) and (64, T(8)/T(64)), where T(64) is the execution associated with the PDS. As is shown there, while all of the speedup line graphs for all classes are below the linear speedup graph, they track very closely to it, except for Class 81.

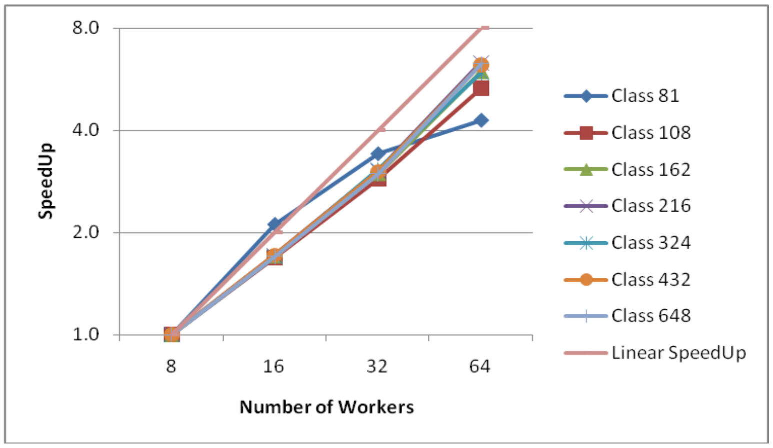

Figure 3: Speedup of Pre-Aggregation Algorithm by Multiplier Class

If one looks into the details about cubelets associated with Class 81, it is not difficult to explain the sub-par performance. With the number of workers being 64, over 40% of the same 37K cubelets have only one cell in their base cubelets. This experimental finding justifies our heuristic that rules out PDS's whose CubeletCounts are likely to be large. On the other hand, one may conclude that our pre-aggregation scheme, coupled with the efficient cluster hardware, performs well even though more than 40% of cubelets are so small that they cannot be further sub-divided.

The comparative performance of our pre-aggregation in terms of the number of workers and the multiplier class is shown in Fig. 4. Let us leave out performance of multiplier class 81, for the same reasons stated above. It is clear from the chart that the pre-aggregation time steadily decreases as the predicted amount of pre-aggregation, signified by the multiple class, decreases. This is true for when the number of workers increases from 8 to 64. In fact, the rate of decrease in pre-aggregation time is so consistent in each case that we could use the multipliers of the classes to predict the relative amounts of pre-aggregation time for the optimal choice of each multiplier class. For example, the ratio of pre-aggregation times between optimal choices of multiplier class X and Y, is roughly equal to X/Y for any of these numbers of workers.

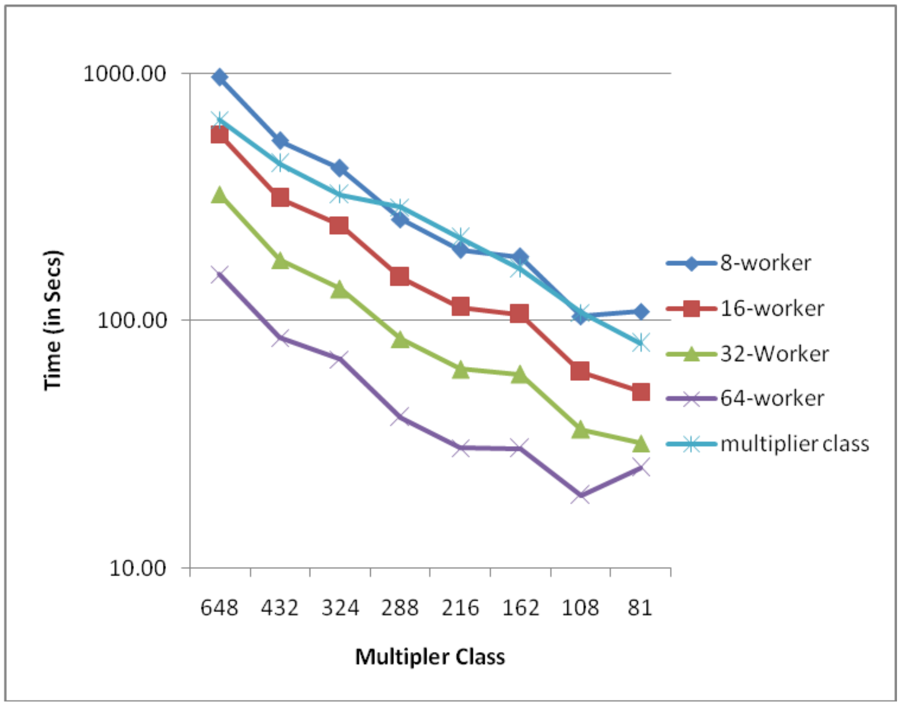

Figure 4: Execution Time of Pre-aggregation Algorithm by Multiplier Class

The chart in Fig. 5 shows the percentage of busy time for all workers, which is the indicator of how well our dynamic load balancing works. Clearly, the workers spend little time sitting idle. This is true even for classes with high multiplier values, i.e., low cubelet counts. It seems that the task scheduling is not an issue here.

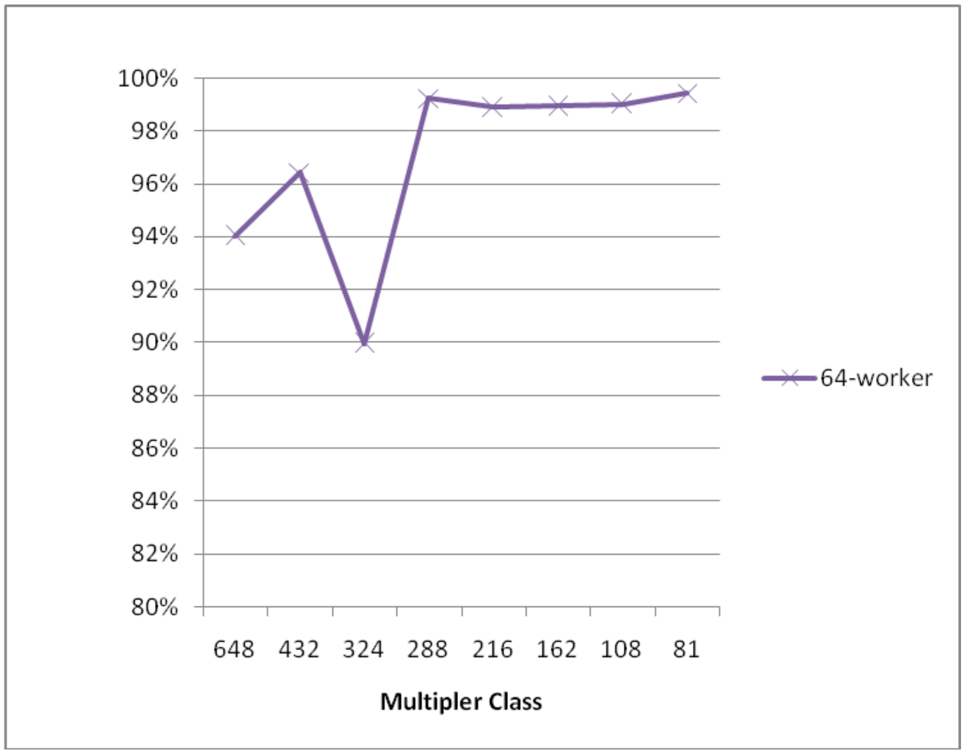

Figure 5: Total Processor Utilization Rate – Pre-Aggregation Process

To conclude our discussion on pre-aggregation performance, the 3 charts above support a convincing argument that the pre-aggregation performance continue to improve as the number of workers increase, provided that the unit of work (cubelet) is not too small.

5.2.4 Querying Performance

Following the example of pre-aggregation, we consider the querying performance against the number of workers and the predicted querying performance as shown Table 7. The query is the worst-case query as defined in Section 4. We also show the average query performance over a class of computationally heavy query matrices.

We first consider the chart in Fig. 6, which is structured similarly to the one in Fig. 3. Some of the line graphs, e.g., those associated with Class 108 and Class 162 exhibit a trend similar to the line graph associated with Linear SpeedUp. On the other hand, the ones associated with Class 432 and Class 648 start to level off as the number of workers increases past 32. In fact, the general trend is that the speedup for the remaining classes drops off as the multiplier increases, because the absolute value of the query response time in each case is so small, i.e., < 1.0 sec., that the querying overhead becomes a significant factor in the query response time.

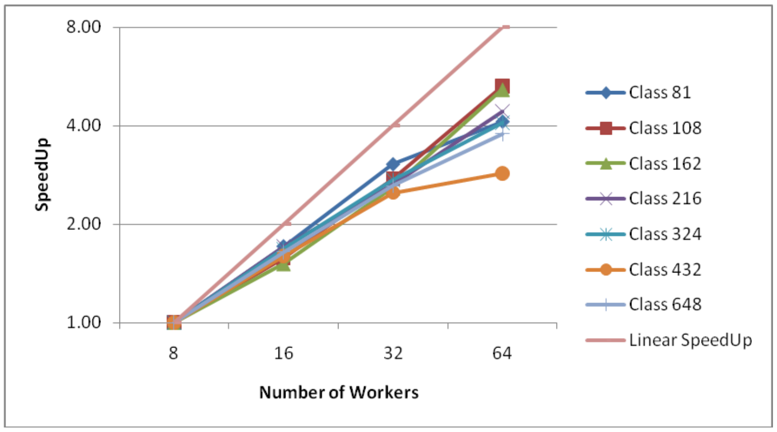

Figure 6: Speedup of Query Processing by Multiplier Class

The response time of the worst-case query is shown in Fig. 7 for each multiplier class. The line graph labelled "predicted" is added to show how nicely it runs in parallel with other line graphs for different worker class. (The response time values of the "predicted" line graph are based on the QueryCount column of Table 6, adjusted proportionally for comparison purposes.) Again, it affirms that our method of predicting the relative query performance of various multiplier classes works well, at least for this dataset.

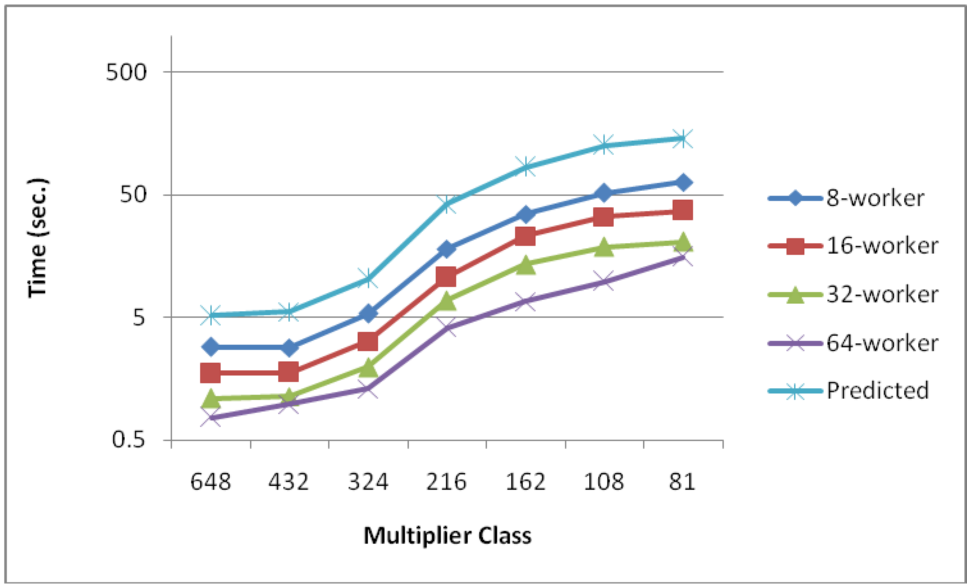

Figure 7: Query Response Time by Number of Workers – Worst Case Query

If the query response time shown in Fig. 7 seems large, it is because we choose to show the performance of the worst-case query. We now briefly look at another measure of query performance over a group of 36 computationally intensive query matrices (see Section 4.5). The average response time of these 36 matrices is roughly 10% of that of the worst-case query matrix (Fig. 8).

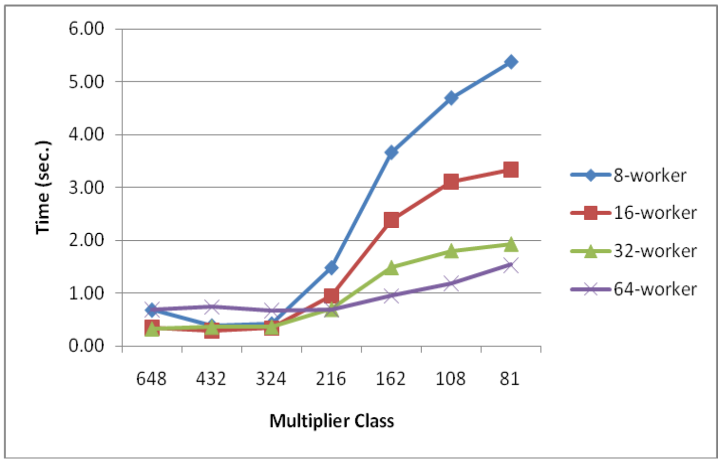

Figure 8: Query Response Time by Number of Workers – Average over Heavy-Duty Queries

In fact, query response times for the four classes with high multipliers are very similar, regardless of the number of workers. Moreover, the average response times for all multiplier classes differ very little if the number of workers is large enough, leading to the conclusion that with this workload, and a relatively small dataset, the query response time 'maxes out' at 64, or even 32 workers.

5.2.5 Scale-up

In order to find out how well our SplitCube approach works for a much larger dataset, we create two randomly generated datasets with exactly the same dimension hierarchies as the modified weather dataset. They have 1 million and 10 million tuples, and are labeled *1M* and *10M* respectively. We focus on only two multiplier classes: 324 and 162. Our optimization scheme produces PDS#11 and PDS#15 for classes 324 and 162 respectively. Note that the choice for class

162 is PDS#27 for the real-life dataset. It is replaced because the number of cubelets entailed by PDS#27 is too large for the randomly generated datasets.

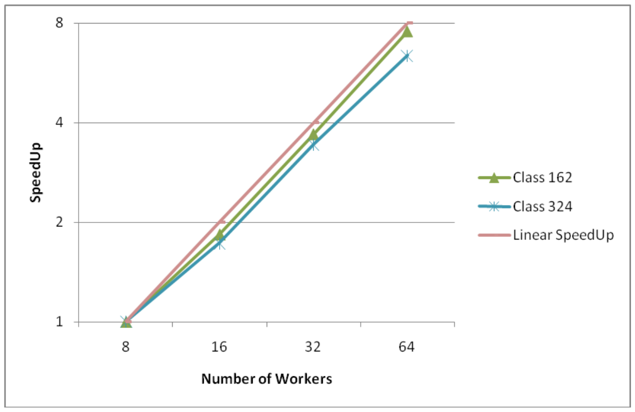

Figure 9: Execution Time of Pre-aggregation Algorithm for Dataset 10M

We are interested in determining the speedups in execution time and query response time for 10M. Comparing the charts shown in Fig. 3 and Fig. 9, one finds that the speedup for 10M is clearly closer to linear speedup than the corresponding speedup for the real-life dataset. This phenomenon is due to the increase in average size of cubelets (> 10), which reduces the overhead, in relative terms. Apparently, the same is true for the speedup in query response time, if one compares the chart in Fig. 6 with the one below in Fig. 10.

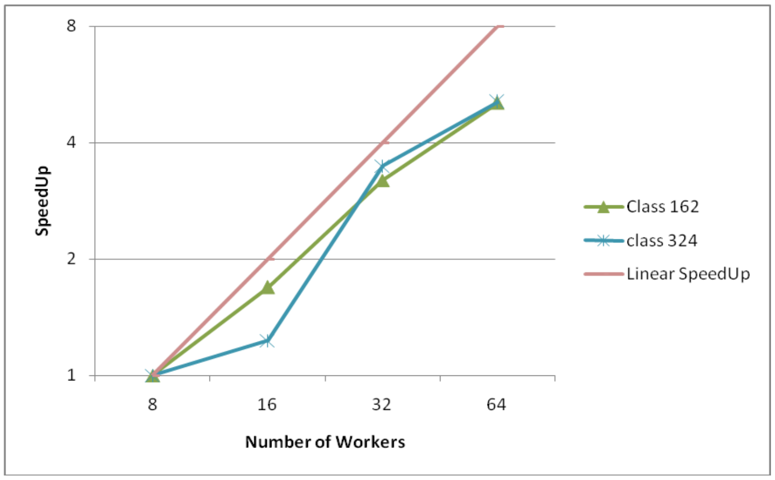

Figure 10: Query Process Time by Multiplier Class

The chart in Fig. 11 shows the scaling up of the execution time for pre-aggregation for three datasets, the real-life, 1M and 10M. We make the following comparisons:

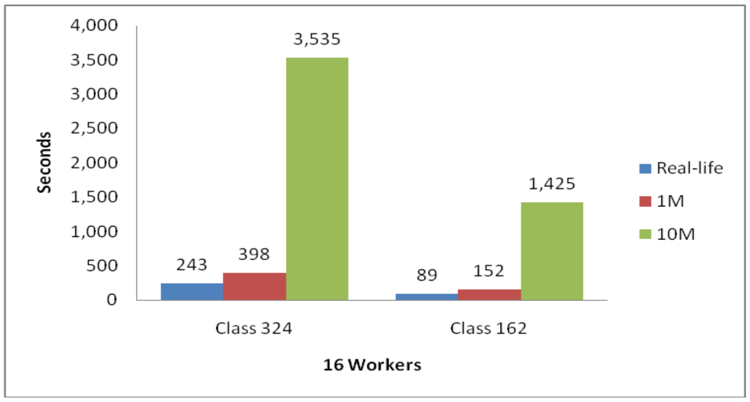

Figure 11: Scale-up in Execution Time for Pre-Aggregation by Multiplier Class

(i) 1M vs. 10M: One finds that our splitcube approach performs even better, in relative terms, when the size of the dataset increases by tenfold, despite substantial increases in disk activity. This improvement is due to the effect of the economy of scale.

(ii) Real-life vs. 1M: The execution time for the real-life dataset is 40% shorter, with roughly the same number of tuples in both dataset. A similar observation is made also in [DER06]. We attribute the difference to the effect of economy of scale as well. For example, in the case of Class 162, the MaxBaseCubeletSize for the real-life dataset is 4,680, while the same for 1M is only 267.

5.2.6 Overhead Analysis

There are two sources of overhead associated with pre-aggregation: the computation of CubeletCount and MaxBaseCubeletSize, and the subsequent partition of the base cube into a number of tasks, after the PDS has been chosen. They are 19 seconds and 20.1 seconds respectively. We are able to reduce the former by roughly 90% by polling only 10% of the Weather base cube (about 100,000 tuples).

We ponder two questions about the overhead here. Firstly, is sampling absolutely necessary, i.e., what would happen if just the theoretical values for these parameters are used? Secondly, how effective is the sampling technique compared to the full scan of the input datasets?

Let us consider first MaxBaseCubeletSize. Unlike a synthetic dataset, most real-life datasets, including the Weather dataset, are skewed in their distribution for values in some set of attributes. Without an accurate measure of this parameter, we may end up with very bad choices. Consider PDS #17 in Table 7, which has more than 45% of the base cube packed into a base cubelet, resulting in a likely speedup of a little more than 2, even when 64 workers are present to share the load. To attest to the effectiveness of sampling, we find that out of 109 PDS's tested, 71% of them whose estimated values of the parameter are within +/- 10% of the actual values. The rest of them are associated with PDS's with large MaxQueryCount values, which are eliminated for consideration even when the sampling begins. The use of the sample-based value of the parameter in lieu of the actual value results in exactly the same outcome in our experiments.

We now turn to CubeletCount. There is a standard probability formula to compute an estimate for this parameter [F57]. Unfortunately, the dimensions in a real-life multi-dimensional database are often correlated, and sometimes heavily correlated. Consequently, the estimate is not reliable. (Come to think of it, there will be no point for dimensional analysis if the dimensions are completely independent of each other.) How effective is the sample-based parameter? Let us consider the PDS's in the multiplier class 108 (Table 6), for example. Three candidate PDS's, #39, #71 and #53 are ranked in the same order according to the MaxQueryCount. The ranking according to the actual QueryCount is different, i.e., #53, #71, and #39. This is largely because of the drastic reduction in CubeletCount for PDS #53 from the predicted one. Without any reference to the input dataset, PDS #39 would be chosen as the choice. Consequently, the query response time for the multiplier class would be 80% longer than otherwise. By probing only 10% of the input data set, this discrepancy is discovered, and the ranking is changed accordingly. Above all, the probing occurs just once to derive estimate for the two parameters simultaneously.

5.3 Real Estate Dataset

The dataset is extracted from a real-life real estate database, which contains attributes of residential property in San Diego, including the geographic location in terms of altitude and longitude. It has 6 dimensions, and 2 measures.

The hierarchy of the last dimension is an R-Tree, which is constructed from the geographic locations of the properties. The details of all six dimensions are shown in Table 8.

level\dimension	1	2	3	4	5	6
1	1	1	1	1	1	1
2	32	4	119	4	5	2
3		106		1103	91	47
4					3138	933
5						18719
6						374950

Table 8: Structure of Dimension Hierarchies - Real Estate Dataset

Following similar steps outlined in Section 4.3, we derive the following table (Table 9), which shows that PDS# 6, 18, and 7 are selected as the choice of the Multiplier classes 72, 48, and 36 respectively (Phase II in Table 9).

PDS#	Multiplier	Max Cubelet Count	Max Query Cost	Max Cubelet Size	Cubelet Count	Query Cost	Phase II	Phase III
5	108	3808	76160	160090	271	5420	Deleted	
12	108	131733	21077280	35151	3716	594560	deleted	
3	72	3510	2094400	16779	830	493850	deleted	
6	72	13090	2094400	12130	1671	267360	selected	deleted
20	72	384370	49199360	545	15165	1941120	deleted	
9	54	35424	21077280	63595	2464	871080	deleted	
18	48	355300	1.35E+09	461	57568	2.19E+08	selected	deleted
7	36	418880	8377600	12089	4031	80620	selected	selected

Table 9: Selection of PDS - Real Estate Dataset

Among these three, PDS#7 is a clear winner since the other two have both higher cube building cost, and higher QueryCount (Phase III in Table 9). Just to demonstrate that our analysis does produce the actual winner, we compare its performance with two other closest competitors, i.e., #3 and #6, as shown in Figs. 12 and 13.

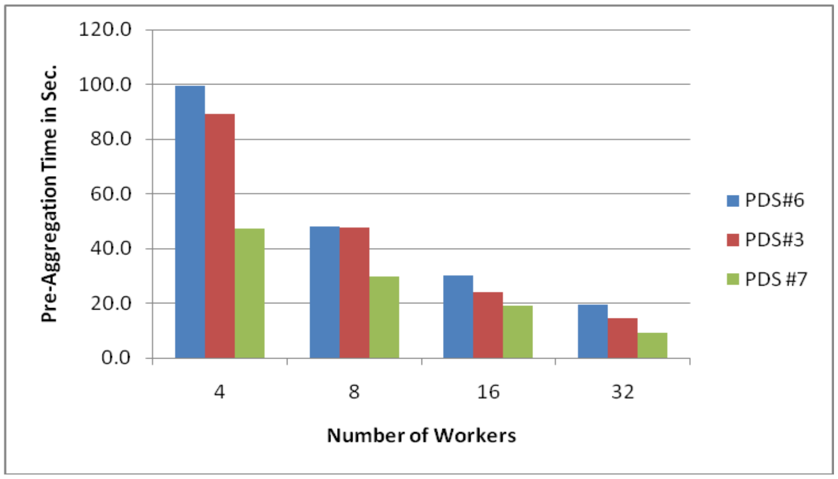

Figure 12: Pre-Aggregation Time by PDS

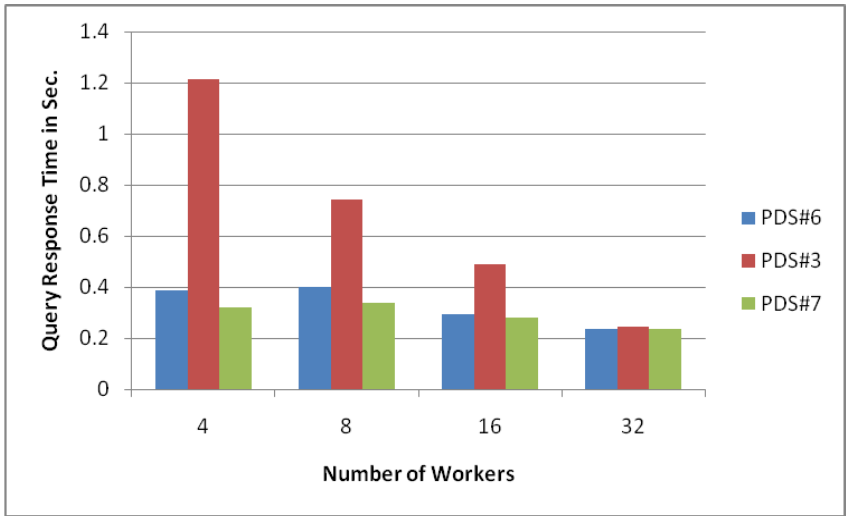

Figure 13: Query Response Time by PDS

We have done some sampling on this data set to estimate the CubeletCount and MaxBaseCubeletSize associated with the 8 PDS's listed in the table 9. The sampling-based estimates on MaxBaseCubeletSize are very good;

the least accurate one is only 9% away from the actual figure. For CubeletCount, the sampling-based estimates associated with PDS#3, PDS#6, and PDS#7 are: 490, 857 and 1,577 respectively. With respect to the actual ranking, these estimates are consistent with the actual values of CubeletCount, which are: 830, 1,671 and 4,031 respectively. These findings confirm the accuracy of our sampling methods on another dataset.

6 CONCLUSION AND FUTURE WORK

In this thesis, we present the splitcube approach for building an OLAP database engine for a cluster system. Our objective is to design an OLAP engine that provides fast response time to a query matrix. Partial pre-aggregation is deemed necessary to achieve this objective, as our experimental results show. This objective sets this research apart from other published research efforts that try to find an efficient way to build all cuboids on a cluster system.

In a broader perspective, what we learn from conducting this research is that the algorithm designer of a data-intensive application to be deployed on a cluster system should try to minimize sharing of data among the processors and create a substantial number of tasks that are free to run on any processor. In other words, we should leverage the advantages of a cluster system, and minimize the negative impact on overall performance due to the inherent limitations of this platform. Of course, if there are too many tasks around, the overhead will eat up any performance gain due to parallel processing. This is why an optimization process is necessary to rule out any schemes that spin out too many tasks. On the other hand, if we cut down the size of a task, so that it becomes an in-memory task, there may be net performance gain.

Clearly, this research is a proof-of-concept work, leaving much room for further refinements. For example, there is an issue about the small cubelets. Combining small cubelets together will improve the economy of scale in cube building and query processing, at the expense of increased overhead. As the MPP technology continues to evolve, our design for our OLAP database engine must be adjusted continually to take advantage of the new hardware. Indeed, many clusters, including ours, have hybrid MPP architecture: a cluster that features a multi-core CPU in each node. As the number of cores in the CPU expands, the amount of computational power vested with each node will multiply. There certainly will be more memory in the node too. New partitioning schemes may be needed to address this shift in computational power.

REFERENCE LIST

[Akin02] M. Akinde, M. Böhlen, T. Johnson, L. Laskhmanan, D. Srivastava, "Efficient OLAP query processing in distributed data warehouses", Information Systems, 2002.

[Alm08] R. Almeida, J. Vieira, M. Vieira, H. Madeira, J. Bernardino, "Efficient Data Distribution for DWS", 10th Int. Conf. DAWAK, Turin, Italy, 2008

[BDH03] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture", IEEE Micro, vol. 23, no. 2, Mar.-Apr. 2003

[C08] C. Cole, "High-Confidence Clustering with Intel Cluster Ready", Open Source Grid & Cluster Conference, 2008

[DER06] F. Dehne, T. Eavis, A. Rau-Chaplin, "The cgmCube Project: Optimizing Parallel Data Cube Generation for ROLAP", Distributed and Parallel Databases, An International Journal, 19 (2006), pp. 29-62.

[F57] W. Feller, "An Introduction to Probability Theory and Its Applications", John Wiley and Sons, 1957.

[Furt05] C. Furtado, A. Lima, E. Pacitti, P. Valduriez, M. Mattoso, "Physical and Virtual Partitioning in OLAP Database Clusters", Proc. 17th Int. Sym. On Computer Architecture and High Performance Computing, 2005

[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

[GS97] S. Goil, A. Choudhary, “High Performance OLAP and Data Mining on Parallel Computers”, Data Mining and Knowledge Discovery, 1 (1997), pp. 391-417

[HWL94] C. Hahn, S. Warren, and J. London. *Cloud reports* http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html

[JVYA05] R. Jin, K. Vaidyanathan, G. Yang, G. Agrawal, “Communication and Memory Optimal Parallel Data Cube Construction”, IEEE Transactions on Parallel and Distributed Systems, Vol. 16, Issue 12, December 2005

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

[LPZ03] L. Lakshamanan, J. Pei, Y. Zhao, “QC-Trees: Efficient Summary Structure for Semantic OLAP”, Proc. of ACM SIGMOD, San Diego, 2003.

[Lu03] H. Lu, J. Yu, L. Feng, Z. Li, “Fully Dynamic Partitioning: Handling Data Skew in Parallel Data Cube Computation”, Distributed and Parallel Databases, An International Journal, 13 (2003), pp. 181-202.

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

[Luk08] W. Luk, “A B-tree Based Scheme for OLAP Aggregation and Query Processing”, submitted for publication.

[MK99] S. Muto and M. Kitsuregawa, “A Dynamic Load Balancing Strategy for Parallel Datacube”, DOLAP 99, Kansas City, Mo.

[MKIK07] K. Morfonois, S. Konakas, Y. Ioannidis, N. Kotsis, “ROLAP Implementations of Data Cubes”, ACM Computing Surveys, Volume 39, No. 4, 2007

[MS01] Microsoft Corp. “MS SQL Server 7.0 OLAP Services”, http://www.microsoft.com/technet/prodtechnol/sql/70/maintain/olap.mspx, July, 2001

[MS08] Microsoft Corp. “Multidimensional Expressions (MDX) Reference”, SQL Server 2008 Books Online, August 2008.

[N07] S. Norall, “Introducing 'data warehouse appliances” http://www.infostor.com/article_display/introducing-data-warehouse-appliances/293088/s-articles/s-infostor/s-top-news/s-1.html, 2007.

[NWY2001] R. Ng, A. Wagner, Y. Yin, “Iceberg-cube computation with PC clusters”, Proc. Of SIGMOD, 2001.

[P2005] Pendse, N., “Database Explosion”, http://www.olapreport.com/DatabaseExplosion.htm, Feb. 2005.

[RS97] K. Ross, S. Srivastava, “Fast Computation of Sparse Datacubes”, Proc. Of VLDB, 1997

[SMR00] T. Stöhr, H. Märtens, E. Rahm, “Multi-Dimensional Database Allocation for Parallel DataWarehouses”, Proc. Of VLDB, Egypt, 2000.

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

[Wit03] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L.

Shen, S. Subramanian, “Spreadsheets in RDBMS for OLAP”, Proc. of ACM SIGMOD, San Diego, 2003.

[YC00] C. Yu, C. Chang, “The state of the Art in Distributed Query Processing”, ACM Computing Surveys, Vol. 32 , No. 4, 2000

[YJA02] G. Yang, R. Jin, G. Agrawal, “Implementing Data Cube Construction Using a Cluster Middleware, Algorithms, Implementation Experience, and Performance Evaluation”, Proc. Of the 2nd International Symposium on cluster Computing and the Grid, 2002.