OLAP DATABASE COMPUTATION WITH A SPLITCUBE ON A MULTI-CORE SYSTEM

by

Liang Zhou Jiang
Bachelor of Science, Simon Fraser University, 2006

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In the
School
of
Computing Science

© Liang Zhou Jiang 2009
SIMON FRASER UNIVERSITY
Summer 2009

All rights reserved. However, in accordance with the Copyright Act of Canada, this work may be reproduced, without authorization, under the conditions for Fair Dealing. Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
APPROVAL

Name: Liang Zhou Jiang
Degree: Master of Science
Title of Project: OLAP Database Computation with a Splitcube on a Multi-Core system

Examining Committee:
Chair: Dr. Qianping Gu
Professor, Computing Science
Simon Fraser University

Dr. Woshun Luk
Senior Supervisor
Professor, Computing Science
Simon Fraser University

Dr. Jian Pei
Supervisor
Associate Professor, Computing Science
Simon Fraser University

Dr. Mohamed Hefeeda
Examiner
Assistant Professor, Computing Science
Simon Fraser University

Date Defended/Approved: August 7, 2007
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
ABSTRACT

In On-Line Analytical Processing (OLAP), the computation of a full computed data cube has exponential complexity due to the cube dimensionality. An efficient alternative is splitcube which does not emphasize on a full computed data cube. In this report, we explore the use of newly published Microsoft Parallel Extension to build and query splitcube in parallel on a multi-core system. We provide an implementation of a parallel splitcube building program as well as an implementation of a parallel splitcube query processing program. Experiment data shows we can achieve speedup in parallel splitcube building process which is bounded by the performance of disk I/O operation. Experiment data also shows slowdown in parallel splitcube query process. The slowdown is caused by the limited disk I/O bandwidth and the increased latency of synchronous disk read operations result from multiple non-contiguous accesses to the physical disk.

Keywords: OLAP; Splitcube; SMP; Multi-core; Cluster; Parallel Computing; Parallel Extension; Task Parallelism
ACKNOWLEDGEMENTS

I would like to express my gratitude to Professor Wo Shun Luk who has been more than a supervisor throughout my studies. I won’t go anywhere without his guidance and support. Having the chance to work with him was an honour. I give special acknowledgement to my parents who are of great influence to me throughout my life. I thank them for their unconditional love and support. Last but not least, I owe special thanks to my family, my wife Judy and my lovely daughter Yu Fei. With their continuous support, encouragement, and love, I would not have been able to achieve my goal.
TABLE OF CONTENTS

Approval ................................................................................................................... ii
Abstract .................................................................................................................... iii
Acknowledgements ............................................................................................... iv
Table of Contents ................................................................................................... v
List of Figures ......................................................................................................... vii
List of Tables .......................................................................................................... viii

1: Introduction ....................................................................................................... 1
  1.1 Symmetric Multiprocessing and Cluster ...................................................... 1
  1.2 Multi-core Computing and Microsoft Parallel Extension ......................... 3
  1.3 Motivation and Application ........................................................................ 4
  1.4 Related Works ............................................................................................. 5
  1.5 Thesis Organization .................................................................................... 6

2: Review .............................................................................................................. 8
  2.1 Cube and Dimension Hierarchy .................................................................. 8
  2.2 Building and Querying a Fully Pre-Aggregated Cube ............................... 9
  2.3 Splitcube – General Concepts ..................................................................... 10
  2.4 Building a Splitcube .................................................................................. 11
  2.5 Querying a Splitcube ................................................................................ 12
  2.6 Choice of a PDS ......................................................................................... 13

3: Construction of Splitcube in Parallel .............................................................. 15
  3.1 Analysis and Dissection of Sequential Process of Building Splitcube .......... 15
  3.2 Build Splitcube in Parallel .......................................................................... 16
    3.2.1 Cubelet Building Task ........................................................................... 17
    3.2.2 Execute Cubelet Building Task in Parallel ......................................... 19
      3.2.2.1 Task Generator ............................................................................... 20
      3.2.2.2 Task Executor ............................................................................... 20
      3.2.2.3 Task Scheduler ............................................................................ 21
    3.2.3 Algorithm Improvement ......................................................................... 22
      3.2.3.1 Pipeline Task Generation Process with Double Buffer .................. 22
      3.2.3.2 Duplicate Input ............................................................................ 23

4: Querying Splitcube on a Multi-core System ................................................. 25
  4.1 Analysis and Dissection of Sequential Process of Querying Splitcube ........ 25
  4.2 Query Splitcube in Parallel ........................................................................ 26
    4.2.1 Query Processing Task ......................................................................... 26
    4.2.2 Execute Cubelet Querying Task in Parallel ....................................... 27
      4.2.2.1 Query Decomposition .................................................................. 28
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Commercial grade SMP with separated memory controller and I/O channel</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Entry-level SMP with shared memory and I/O bandwidth</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Dimension hierarchies A, B and C</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Pseudo code of building splitcube sequentially</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Pseudo code of cubelet building task</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>Pseudo code of building splitcube in parallel</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Architectural design of parallel cubelet building model</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>Pseudo code of generating parallel cubelet building tasks</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>Building splitcube in parallel with double buffer and pipeline</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>Building splitcube with duplicate input method</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>Pseudo code of querying splitcube sequentially</td>
<td>26</td>
</tr>
<tr>
<td>12</td>
<td>Pseudo code of querying cubelets in parallel</td>
<td>27</td>
</tr>
<tr>
<td>13</td>
<td>Querying splitcube in parallel</td>
<td>28</td>
</tr>
<tr>
<td>14</td>
<td>Quick sort test – execution time without disk I/O</td>
<td>34</td>
</tr>
<tr>
<td>15</td>
<td>Quick sort test – execution time with disk I/O</td>
<td>35</td>
</tr>
<tr>
<td>16</td>
<td>Quick sort test – CPU time comparison</td>
<td>36</td>
</tr>
<tr>
<td>17</td>
<td>Quick sort test – disk I/O speed statistics</td>
<td>36</td>
</tr>
<tr>
<td>18</td>
<td>Quick sort test – CPU, disk I/O and overall speedup ratio</td>
<td>37</td>
</tr>
<tr>
<td>19</td>
<td>Splitcube building test – speedup ratio</td>
<td>42</td>
</tr>
<tr>
<td>20</td>
<td>Splitcube query process – execution time comparison (a)</td>
<td>46</td>
</tr>
<tr>
<td>21</td>
<td>Splitcube query process – execution time comparison (b)</td>
<td>46</td>
</tr>
<tr>
<td>22</td>
<td>Splitcube query process – CPU time and disk I/O speedup ratio</td>
<td>47</td>
</tr>
<tr>
<td>23</td>
<td>Total number of cubelet files for different PDS</td>
<td>49</td>
</tr>
<tr>
<td>24</td>
<td>Total size for different PDS</td>
<td>50</td>
</tr>
<tr>
<td>25</td>
<td>File size of the maximum cubelet for different PDS</td>
<td>50</td>
</tr>
<tr>
<td>26</td>
<td>Average file size for different PDS</td>
<td>51</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 1  Prefix & cubelets with A as PDS ............................................................... 12
Table 2  Prefix & cubelets with A and B as PDS ..................................................... 12
Table 3  Quick sort test – execution time with and without disk I/O ..................... 34
Table 4  Quick sort test – CPU time as percentage of total time .......................... 35
Table 5  Quick sort test result – disk statistics .................................................... 35
Table 6  JSVM test – CPU and execution time ................................................... 39
Table 7  Cardinalities by levels in the dimension hierarchies in the enhanced weather dataset ........................................................................................ 40
Table 8  PDS configurations of experiments ....................................................... 40
Table 9  Splitcube build test – execution time ..................................................... 41
Table 10 Splitcube building test result – disk performance statistics ....................... 43
Table 11 Query matrix used in experiments ....................................................... 44
Table 12 Point queries processed in query matrix ............................................. 45
Table 13 Splitcube query process – execution time .......................................... 45
Table 14 Splitcube query process – disk I/O speed statistic data ............................ 47
Table 15 Cubelet file size comparison between different PDS ............................ 49
CHAPTER 1

INTRODUCTION

1.1 Symmetric Multiprocessing and Cluster

In computing, symmetric multiprocessing (SMP) refers to a computer architecture which has two or more identical processors that are connected to a single shared main memory. Most multiprocessor systems today use SMP architecture. SMP has many uses in science, industry, and business which often involve multithreaded (multitasked) processing. SMP is one of the most popular hardware platforms for high performance computing (HPC) that employs parallel computing. Traditionally, commercial grade multiprocessing systems contain multiple separated processors, a high bandwidth memory bus, and separated I/O controllers for each processor as shown Figure 1. Recently a popular multi-core variant dominates the entry-level SMP market with multiple CPUs integrated onto a single integrated circuit die (known as a chip multiprocessor or CMP). Such SMP can be found in most new desktop machines and in many laptop machines. The advantages of multi-core SMP are the higher clock rate of cache coherency circuitry and significant improvements on the performance of cache snoop operations. It is largely noticed in improved response time while running CPU-intensive processes. Its disadvantages are sharing same system bus and memory bandwidth among all processors. Limited memory and disk I/O
bandwidth will eventually limits the real world performance advantage. Figure 2 demonstrates a simplified multi-core architecture.

Figure 1 Commercial grade SMP with separated memory controller and I/O channel

Figure 2 Entry-level SMP with shared memory and I/O bandwidth

Another recent architectural trend away from the traditional SMP is the emergence of cluster, or grid of clusters, for high performance computing. A cluster system is a collection of computers connected by a high speed local area network. Cluster is popular because it enjoys a 3-5 times cost/performance advantages over a SMP system, according to a recent industry study ([C08]).
Architecturally, cluster system scales better than SMP and is more fault tolerant. However SMP has its own advantages such as less corporation components required, lower latency for sharing, and lower software complexity. With the most recent popularity of using multi-core CPU on desktop and laptop machines, soon it could be seen that node computer in cluster system is replaced by multi-core machine which can potentially bring in more computation power.

1.2 Multi-core Computing and Microsoft Parallel Extension

The multi-core revolution is a software revolution. Not only does software need to adapt to the new environment by being parallelized, but parallelization makes the software more complicated, error prone, and thus expensive. Among the several programming models that have been proposed for multi-core processors, shared memory model is the most popular one followed by many implementations. In general shared memory model can be further categorized into thread level programming and task level programming [FB08]. In thread level programming, threads typically need to synchronize with each other or ensure mutual exclusive using locks, or condition variables. Its low level model such as kernel threads can achieve the best performance among the different models. Thread level programming is in essence concurrent. This usually increases the complexity and makes programming more difficult. Task level programming differs from thread level programming in that tasks are very light weight, always implemented in user mode. Tasks often run completely independent of each other, e.g. the iteration of a parallel loop. In essence tasks aim at parallel, not computing and typically can be executed in serial manner. A
good example is Microsoft Parallel Extension. In response to the trend of using multi-core processor in PC, recently Microsoft published its Parallel Extension to .Net framework to help software developer transform their programs into parallel fashion to utilize the more computation power brought in by multi-core processors. Parallel Extension to the .Net framework is a managed programming model for data parallelism, task parallelism, and coordination on parallel hardware unified by a common work scheduler. Parallel Extension makes it easier for developers to write programs that scale to take advantage of parallel hardware by providing improved performance as the numbers of cores and processors increase without having to deal with many of the complexities of today’s concurrent programming models.

1.3 Motivation and Application

Broadly speaking, this research project is about development of data-intensive applications on off-the-shelf, non-server class multi-core systems with limited I/O capacity. We study one important data-intensive application: OLAP database computation, which includes pre-aggregation and query processing on OLAP databases.

Recently, Johnny Zhang has completed a research project on OLAP database computation on a cluster [JZ09]. He developed a specific scheme for computing and querying splitcube, which is a partially pre-aggregated cube. A splitcube consists of a number of cubelets, which may be computed and queried in total isolation, and therefore is particularly suited for parallel platforms. It is shown that OLAP database computation based on this scheme can achieve a
near linear speedup on a cluster with 64 nodes, as long as there is sufficient data for meaningful computation. In this research, we implement the same idea and algorithm on a multi-core system with Microsoft Parallel Extension. We will explore the potential benefit we could gain from a multi-core system.

Traditionally, OLAP computation is confined to well-equipped server-class machines. With the multi-core technology and ample supply of low-cost main memory, a desktop or hand-held device may have become a suitable platform for meaningful OLAP applications. This study is important also for future developments in OLAP computation on clusters. Until recently, most applications running on clusters have not considered the impact of multi-core CPU in a single node on the overall strategy of distributing workload among the worker nodes in the cluster. While this study does not directly address this question, it nonetheless will shed some light on this important issue.

1.4 Related Works

Perhaps unsurprisingly, much of the research on database processing in a multi-core system assumes a server platform. Several schemes, e.g., [Q08], have been proposed to provide sharing of data in the L2 cache among concurrent queries submitted by different users. We manage to locate three papers/articles that are relevant to this research. In [SY09], which is most related to this research, the authors describe an implementation of OLAP data cube computation using the map/reduce programming framework on a cluster of multi-core nodes. Essentially, the input dataset is split into multiple subsets, each of which is processed by a core (CPU). According the performance data
on the well-known weather dataset, the algorithm for cube building and query processing on a single CPU (i.e., the serial algorithm) runs much slower than other algorithms on the same dataset. The speedup is about 1.5 for 4 cores, and levels off after 8 CPUs in a cluster with 4 quad-core computers. The other two address the question of limitation in I/O bandwidth of the multi-core systems. A group of researchers associated with Oak Ridge National Lab, which publishes MPI (Message Passing Interface), the de facto standard API for high performance computing, proposes a scheme to ease the I/O bottleneck in a multi-core system [OU06]. One of the cores is appointed as the coordinator for I/O activities for the other cores. On a broader perspective, the author of the article published in the November 2008 issue of IEEE Spectrum ([SM08]) warns that the present design of multi-core may not be helpful for data-intensive applications, as the simulation study shows that more cores will actually slow down the applications, because the data can’t be moved fast enough to the processors.

1.5 Thesis Organization

The rest of this project report is organized as follows. We begin by reviewing some key concepts of splitcube in Chapter 2. This will serve as algorithm foundation to help understand the implementation of splitcube in later sections. In Chapter 3, we discuss the construction of splitcube on a multi-core system by transforming a serial program into a parallel program with task parallelism. Then in Chapter 4, we discuss the parallel implementation of splitcube query processing. We then perform a group of tests using well-known
weather dataset. Experimental results and analysis are presented in Chapter 5.

We will conclude our project and research in Chapter 6.
CHAPTER 2

REVIEW

2.1 Cube and Dimension Hierarchy

The (full) cube is defined in this paper to be a k-dimensional array, where k is a positive integer greater than zero. Each dimension of a cube has Di 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. Figure 3 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.

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 paper 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 (FPA) cube, or otherwise, a partially pre-aggregated cube. The set of all primary cells is called a base cube. As examples, \((a_1, b_2, c_5; 4)\) 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 Building and Querying a Fully Pre-Aggregated Cube

Numerous algorithms have been published on how to build a FPA cube efficiently from the base cube, on serial and parallel platform. The algorithm chosen for this research stores all the cells in the FPA cube into an in-memory B-tree, as they are generated, and then outputs them into the disk. If the memory is not sufficiently large to store all the cells, then the base cube is partitioned in a number of subset, and multiple B-trees are created and stored in files (one per B-tree) on the disk.
A point query is defined as the address of a cell \( Q (q_1, \ldots, q_k) \) where \( q_i, 1 \leq i \leq k, \) is the coordinate of the cell in the \( i^{th} \) dimension. The query processor will look up the cell in the cube and retrieve the measure for the query. In our case, query processing is about retrieving the cell in the appropriate B-tree. In practice, an OLAP query is much more complex than a point query. 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]. For this research, it suffices to describe a query matrix as a collection of point queries. Processing a query matrix is equivalent to find answers for all point queries in the matrix.

2.3 Splitcube – General Concepts

A splitcube is a partially pre-aggregated (PPA) cube. 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., \( P_1, \ldots, P_m \), 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 contains all the cells in the cube, except those cells whose prefixes contain 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 a cubelet. Each cell in 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.

2.4 Building a Splitcube

Building a splitcube consists of building all cubelets associated with the splitcube. Computation of a cubelet associated with a specific prefix from the base cube may proceed as follows:

1. Locate the cells in the base cube, i.e., primary cells, with the same prefix.
2. Project these cells on the CDS and the measure, which form the base cubelet.
3. 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: (a₁, b₂, c₁; 3), (a₃, b₃, c₃; 4) and (a₃, b₄, c₂; 2).

Consider the splitcube, SCₐ, 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 $\{A\}$.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Prefix & Cubelet \\
\hline
$(a_1)$ & $(b_2, c_1; 3), (b_2, c_9; 3), (b_2, c_{13}; 3), (b_2, c_{15}; 3),$
         & \hspace{1em} $(b_5, c_1; 3), (b_5, c_9; 3), (b_5, c_{13}; 3), (b_5, c_{15}; 3),$
         & $(b_7, c_1; 3), (b_7, c_9; 3), (b_7, c_{13}; 3), (b_7, c_{15}; 3)\) \\
\hline
$(a_3)$ & \ldots \ldots \\
\hline
\end{tabular}
\caption{Prefix & cubelets with A as PDS}
\end{table}

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.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Prefix & Cubelet \\
\hline
$(a_1, b_2)$ & $(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)\) \\
\hline
\end{tabular}
\caption{Prefix & cubelets with A and B as PDS}
\end{table}

### 2.5 Querying a Splitcube

Processing a query for a splitcube is equivalent to processing all the point queries inside the matrix. 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 \(q_i\), \(1 \leq i \leq m\).

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.

### 2.6 Choice of a PDS

In Chapter 5, the splitcube building time and query process time will be compared among 8 different PDS's. (There are 2\(k\) splitcubes for a given \(k\)-dimensional cube.) It is therefore instructive to consider how the choice of a PDS will affect the cube building and querying performance, on a single CPU.

In general, the splitcube building time is in a reverse relationship with query processing time. After all, a splitcube is a PPA. More pre-aggregation work are
done, the less on-the-fly computations are required during the query time. However, there are more detailed factors to consider:

The cardinality of the PDS: the more dimensions included in the PDS, the less dimensions in a cubelet, which results in shorter splitcube building time.

The number of cubelets: the more cubelets there are, the smaller on average is each cubelet, which results in less building time for each cubelet. However, the number of cubelets will increase the overhead in cubelet building and query processing, because there will be more files to open and close.

The size of the splitcube: generally speaking, the smaller the splitcube, the shorter is the splitcube building time, and the longer is the query processing time. However, if the footprint of the splitcube is very large, the query processing may take longer time due to heavier I/O time.
CHAPTER 3

CONSTRUCTION OF SPLITCUBE IN PARALLEL

3.1 Analysis and Dissection of Sequential Process of Building Splitcube

The process of building splitcube starts with parsing and reading input data and the associated metadata, e.g., dimension hierarchy tables. The metadata file stores not only the total number of dimensions, the name of each dimension, also the hierarchy structure of each dimension including individual value and their aggregations. Contents of this file are used to define Prefix dimension set (PDS) and cubelet dimension set (CDS). Data dictionaries of the splitcube including PDS and CDS are created and stored into an archive file. Later they will be retrieved and used in query processing. Database tuples will be loaded from files and stored in memory or a temporary file. These tuples are primary cells that are part of base cubelets. For each non-zero size cubelet (cubelet has zero number of records), building process will first determine the partitions (i.e., B-trees) needed for the cubelet. The number of partitions and the index of cubelet will be used by file manager to set up file streams. After all primary cells are loaded from database tuples, the GenerateAggregate routine will be called to build B-tree(s) and store the result into partition data file. Metadata of the B-tree will also be archived so that it can be retrieved to re-build B-tree from partition data file for query processing. The exact same procedure to build a single cubelet will be
performed on all cubelets throughout iterations. The following is the pseudo code of the sequential process to construct a splitcube.

\[
\text{CreateSplitcube}
\]

\[
\{ \\
\quad \text{Parse the dimension hierarchy table file} \\
\quad \text{Build the data dictionary of the whole cube} \\
\quad \text{Get aggregation level} \\
\quad \text{Define prefix dimension set (PDS) and cubelet dimension set (CDS)} \\
\quad \text{Parse database tuples and load them as primary cells} \\
\quad \text{Sort all primary cells based on PDS} \\
\quad \text{Write data dictionary into archive} \\
\quad \text{For each non-zero size cubelet do} \\
\quad \quad \text{Determine the partitions needed for the cubelet} \\
\quad \quad \text{Set up file manager for partition data files} \\
\quad \quad \text{Load primary cells that are part of base cubelet} \\
\quad \quad \text{Build B-trees for each partition} \\
\quad \quad \text{Write metadata of B-trees into archive} \\
\quad \text{End For} \\
\}\]

Figure 4   Pseudo code of building splitcube sequentially

The discussion above shows that in order to parallelize the process of building splitcube we can parallelize the procedure of building each cubelet (code lines shown in grey section).

3.2 Build Splitcube in Parallel

Figure 4 shows every iteration of the FOR loop will perform the exact same procedure and construct a cubelet. If all primary cells that are part of the relevant base cubelet are available, building a single cubelet is a total
independent task. We first define the parallel task in a parallel process of building splitcube.

A parallel task is defined as a single execution unit which consists of several execution steps. Execution steps in a parallel task are executed sequentially. A parallel task is a self-contained component which includes all required data in order to execute the task. There is no particular order, partial or total, among all parallel tasks. With required data is available, multiple parallel tasks can be executed in any order. In parallel process of splitcube, a parallel task can be a cubelet building task, or a query processing task, or even a task that generates parallel tasks (parallel task generator).

3.2.1 Cubelet Building Task

We briefly introduce the computation of a cubelet we discuss how to build a splitcube in section 2.4. Here we give the detail on how to build a cubelet in our splitcube implementation. We can make up a cubelet building task from the FOR loop (the grey part) as mentioned above. Once the file streams of partition files are ready, a cubelet building task starts load primary cells that are part of the base cubelets. A fully aggregated (k-m)-dimensional cube will be computed from the base cubelet and the results will be inserted into B-tree based on the cell address. Result B-tree then will be flushed onto disk. The following is the pseudo code of the cubelet building task.
BuildingCubeletTask
{
    Set up file manager to open partition data files
    For each partition do
        Load primary cells that are part of base cubelet
        Build B-tree from base cubelet by aggregation
        Record metadata of B-tree
        Write B-tree into partition data file
    End for
}

Figure 5  Pseudo code of cubelet building task

With parallel cubelet building tasks ready and in place, now we can change
the algorithm and build splitcube in parallel. The following is the pseudo code of
building splitcube in parallel fashion

CreateSplitcubeInParallel
{
    Parse the dimension hierarchy table file
    Build the data dictionary of the whole cube
    Get aggregation level
    Define PDS and CDS
    Build data dictionary of the cubelets
    Compute the total number of cubelets
    Parse input data and load database tuples
    Sort loaded database tuples based on PDS
    Write the data dictionary into archive
    Generate parallel cubelet building tasks
    Execute all cubelet building tasks in parallel
}

Figure 6  Pseudo code of building splitcube in parallel
3.2.2 Execute Cubelet Building Task in Parallel

The architectural design of our parallel cubelet building model is demonstrated in Figure 7. Key components include task generator, task scheduler, and task executor. All cubelet building tasks are first generated by the task generator. When the task generator creates cubelet building tasks, it assigns weight to each task based on some metrics e.g. the size of prefix set. Later such weight will be used by task scheduler in task distribution for load balancing. When all tasks are generated and ready, task scheduler will start parallel execution by requesting computational resources from operation system and dispatching multiple task executors to build cubelets. In this research, we practise task parallelism. Thus task scheduler will submit requests to OS kernel through Microsoft Parallel Extension library without knowing and managing any multithreading issues. This greatly eases code programming and lets us focus on cube building algorithm. Nonetheless, the application developer may through the task scheduler dictate how and in what order the tasks in the task pool are executed.

Figure 7 Architectural design of parallel cubelet building model
3.2.2.1 Task Generator

A key step in parallel process of building splitcube is the generation of parallel cubelet building tasks. The result of this step is a number of independent cubelet building tasks that can be executed in any order. We call this step and its related function task generator. The following is the pseudo code of how task generator works.

ParallelCubeletBuildingTaskGenerator
{
    For each non-zero size cubelet do
        Create a new parallel cubelet building task
        Pass the reference to the data dictionary of the whole cube to task
        Pass the reference to the data dictionary of the splitcube to task
        Determine the partitions needed for the cubelet and store it in task
        Set up file manager for partition data files
        Locate and load corresponding database tuples
        Assign a weight based on some policy
        Insert task into task pool
    End For
}

Figure 8 Pseudo code of generating parallel cubelet building tasks

For each new task, task generator will assign a weight to the task. Newly created task will be inserted into task pool based on some policy set by task scheduler and prior knowledge on the specific dataset.

3.2.2.2 Task Executor

All cubelets are built by task executor. When running, a task executor gets CPU resource and calls functions to execute all steps in a cubelet building task as described in section 3.2.1. The task executor in our model is similar to
the worker node in a cluster model. The difference is, unlike worker in a cluster, all task executors share memory and disk I/O bandwidth. From the viewpoint of an operating system, a task executor is actually a thread created by Microsoft Parallel Extension library when it is dispatched by task scheduler. While running, a task executor may give up its CPU share and is put in sleep if it is waiting for I/O to load data from or flush data to disk. Multiple tasks are executed simultaneously in parallel.

### 3.2.2.3 Task Scheduler

The task scheduler exercises its control on task execution in three ways. It may choose the time to initiate the execution. For example, it may initiate the execution without all tasks have been created. This is what we call pipelining, to be described in Section 3.2.3.1. It may specify how many threads (or task executors) are to be created. The number of threads need not be the same as the number of cores. For our experiments, we choose 4 or 8 threads. The most important function of the task scheduler is to dictate the choice of the task to be chosen for execution for load balancing purposes.

After tasks have been created, the task scheduler is called to start task execution. Task scheduler will submit the request of running task executors to the runtime system via Microsoft Parallel Extension library. The total number of task executors dispatched by the task scheduler can be equal to, greater than, or less than the total number of available processors or CPU cores. Once a task executor gets its CPU share, it will first consult task scheduler. Task scheduler will instruct task executor to get a proper task from task pool based on some
policy set for loading balance and the runtime data provided by the task executor. During its execution a task executor may give up CPU for IO operation which may potentially leave the CPU idle if no other task executor is waiting for CPU share. It is task scheduler’s job to find out the best number of task executors to be run. Such number cannot be too small which may leave CPU in idle due to insufficient jobs. Neither can it be too large which may deteriorate performance due to the overhead of switching between threads. Our goal is to keep CPU as busy as possible and achieve maximum CPU usage and speedup. Later in our experiments, we will test our algorithm with both 4 (equal to the number of cores) and 8 (twice as many as available cores).

3.2.3 Algorithm Improvement

3.2.3.1 Pipeline Task Generation Process with Double Buffer

Task generation step usually keeps one CPU busy while leaving all other available CPU resources idle. For a large splitcube it may take very long time to get all cubelet building tasks ready. Or even it may not be possible to load all database tuples in memory though we concern more to keep all CPU resources as busy as possible. From the logic viewpoint, we cannot parallelize the task generation step and the task execution step as a task must be generated before it can be executed. But it is only valid on the same single task. When there are bunch of tasks, we could utilize the idle CPU resource to start task execution immediately once a task is ready without waiting for all tasks are created. This is called pipelining. In order to use pipeline, we also need to synchronize the access to the task pool. This actual serializes the access to the task pool and
introduces overhead to obtain exclusive mutex lock. To solve such problem, we introduce the double buffer schema which intends to cut down the overhead and smooth the access to the shared resource. Figure 9 shows the major components in double buffer and pipeline implementation.

![Building splitcube in parallel with double buffer and pipeline](image)

**Figure 9** Building splitcube in parallel with double buffer and pipeline

### 3.2.3.2 Duplicate Input

Double buffer implementation assumes cubelet building tasks will need to access to the shared resource to load database tuples and extract primary cells from it. Duplicate input method tries to create total isolated environment for each execution unit by duplicating the input before parallel execution starts such that when a cubelet building task gets started it will read input from its own environment without worrying about accessing to the shared resource. Figure 10 demonstrates how duplicate input method works.
Figure 10  Building splitcube with duplicate input method
CHAPTER 4

QUERYING SPLITCUBE ON A MULTI-CORE SYSTEM

4.1 Analysis and Dissection of Sequential Process of Querying Splitcube

Query process of splitcube starts with recovering data cube from files obtained through splitcube building process. After a query is entered, it will be first fed into a process called query decomposition. A query may be decomposed into a number of sub-queries with each query has primary members only on dimensions that are in PDS. Sub-queries then will be sorted and grouped by their PDS value. If the decomposition results in a single sub-query set with a specific PDS value, it means such query can be answered by a single cubelet.

On the other hand, if a query is decomposed into a group of sub-query sets, then several cubelets are involved in computation in order to get the answer. The purpose of query decomposition is to find out from which cubelets answers to the query can be retrieved. It then accesses cubelet files one by one in serial to retrieve answer. The following is the pseudo code of sequential query process on splitcube
QuerySplitcube
{
  Build the data dictionary of the whole cube from archive file
  Build the data dictionary of cubelets from archive file
  Input query
  Decompose query into sub-queries based on PDS and CDS
  Sort sub-queries and group them by PDS value
  For each sub-query set do
    Determine the partition file from which B-tree can be rebuilt
    Read from file and rebuild B-tree
    Search key values that is the answer to the query
    Send the answer to the query matrix for consolidation
  End For
}

Figure 11 Pseudo code of querying splitcube sequentially

4.2 Query Splitcube in Parallel

4.2.1 Query Processing Task

A query processing task is also a parallel task. A query processing task needs to know from which file data can be read to rebuild the cubelet and how many files it needs if multiple partitions are used when building the cubelet. Thus a query processing task has a pointer pointing to the data dictionary of cubelet as well as prefix index. Prefix index indicates from which file B-tree can be read as well as the number of partitions used to build the cubelet. Data dictionary contains the metadata indicating how B-tree can be rebuilt such as the root node and the number of tree levels. When processing query on a single CPU, there will be only one query being processed at any one time, and one answer will be returned after retrieval and computation. The aggregation to the final result takes place as the answer of each query returns. Unlike the serial
version, while querying cubelets in parallel, multiple answers would be ready at the same time. The aggregation process may proceed in two ways. The result of a single query may be aggregated immediately after it is returned. Alternatively, the result of all queries may be saved, and the aggregation process begins until all results are available. We opt for the latter in order to avoid the overhead arising from synchronization of concurrent accesses to a result matrix. A query processing task has a list of sub-queries (all have the identical PDS prefix), each has a tag attached. A tag is just the coordinates that indicate the appropriate position of a sub-query in the query matrix.

The following is the pseudo code of how a parallel query processing task computes answers to a query matrix.

```plaintext
QueryCubeletInParallel
{
    Build the data dictionary of the whole cube from archive file
    Build the data dictionary of cubelets from archive file
    Input query
    Decompose query into sub-queries based on PDS and CDS
    Sort sub-queries
    Generate parallel query processing tasks
    Execute query processing task in parallel
    Consolidate all the answers
}
```

Figure 12 Pseudo code of querying cubelets in parallel

4.2.2 Execute Cubelet Querying Task in Parallel

Similar as building splitcube in parallel, Figure 13 demonstrates the architectural design of parallel query processing model. Key components include query decomposition, task generator, task executor, and task scheduler.
Except for query decomposition, they are all similar to the key components as in parallel cubelet building process.

![Diagram of Querying splitcube in parallel](image)

**Figure 13** Querying splitcube in parallel

### 4.2.2.1 Query Decomposition

A crucial step in processing queries is query decomposition. It tells query processor from which cubelets answer can be retrieved and then computed. For a single point query, the decomposition may result in a group of sub-query sets, each set with a unique PDS value. That means, in order to answer the query, partial answer will be retrieved from several cubelets. This is where parallel query process becomes handy because it is possible to compute partial answer
from different cubelets simultaneously. This is how single point query is processed. It is similar to process query matrix. Instead of decomposing a single point query, each query in query matrix will be decomposed one after another. Decomposition usually produces a list of cubelets that will need to be accessed to retrieve answers. After query decomposition, there will be no single sub-query requires to access more than one cubelets in order to compute the answer. This is the characteristic of query decomposition.

4.2.2.2 Task Generator

As in cubelet building process, all parallel query tasks are prepared and generated by task generator. Task generator will create a single query task for all sub-queries that can be answered by the same cubelet. Task generator can also assign a weight to a task such that task scheduler can dispatch tasks based on some load balancing policy.

4.2.2.3 Task Executor

The job of task executor is to locate and load cubelet file, rebuild B-tree(s), and retrieve keys from the B-tree(s). It is the worker in our model. For each single query in a query task, task executor will first strip off its coordinates and then retrieve key from B-tree(s). Once the key has been found and returned, task executor will re-attach the coordinates to the query for consolidation. As in cubelet building process, when running, a task executor may give up its CPU share for I/O operation.
4.2.2.4 Task Scheduler

The task scheduler in query processing plays the same role as it does in cubelet building process. After all tasks are ready, task scheduler is called to start the execution. Task scheduler will submit the request of running all task executors to the OS via Microsoft Parallel Extension library. Once a task executor gets its CPU share it will first get the next available task from task pool and execute it. As in parallel cubelet building procedure, all task executors alternate CPU shares with other task executors or other processes to achieve maximum CPU usage. Multiple queries are computed simultaneously in parallel. Once a query task is finished, a task executor will look for a new task from task pool unless there are no more tasks available.
CHAPTER 5

EXPERIMENTAL EVALUATION AND ANALYSIS

In this chapter, we compare the result of various splitcubes. The objective is to show which splitcube(s) does better than others in our experiments, and investigate why it is better by analyzing the test data collected during the experiments. We begin this chapter with a general description of the software and hardware platforms. To demonstrate the capability of these platforms, we run two simple applications, i.e., the quick sort and video encoding. The former is a very I/O intensive application, and the latter is just the opposite. The test results will set the stage for analyzing the performance of various splitcubes, with respect to cube building and querying splitcube.

5.1 Experiment Configuration

5.1.1 Experimental Hardware Platform

All experiments are performed on a compatible PC equipped with Intel Core 2 Quad processor (Q8200). Q8200 processor has 4 cores and 4MB Level 2 cache. Its front side bus bandwidth is 1333MHz and CPU clock speed is 2.33GHz. Total memory (RAM) installed on the system is 4GB. The hard disk is connected to the motherboard with Serial ATA interface.
5.1.2 Experimental Software Platform

All experiments are conducted on the platform running Microsoft Windows Vista with Service Pack 2. Source code are developed with C++ and compiled with Microsoft Visual Studio 2008 and Parallel Extension to .Net Framework 3.5 CTP. Statistic data of CPU and disk I/O operation performance are captured by Windows Resource and Performance Monitor.

5.2 Preliminary Tests

5.2.1 Experiment Purpose

The purpose of preliminary tests is to show how to adopt parallel programming provided by Microsoft Parallel Extension and quickly change our program from serial to parallel computing. Through the preliminary tests, we will set up the framework to help transform our serial program into parallel fashion and discover potential problems of the implicit use of multi-threading by Parallel Extension in program transformation. We also try to find out how Parallel Extension performs on different types of programs: computational complex application with little disk I/O, disk I/O intensive application without heavy computation, and both computational complex and disk I/O intensive application. Results of the preliminary tests will help us predict the performance of our splitcube program and are set as benchmark to justify the outcome of experiments on cube building and querying splitcube.
5.2.2 Quick Sort Experiment

In quick sort experiment, each test will sort 120 integer arrays. Experiments are performed with 3 different distributions: equal length data, evenly spread data, and highly skewed data. This is to simulate the situation later in splitcube experiment that there are various cubelet building tasks with different size. Each test will be performed twice, both with and without disk operation. This is to discover how disk I/O will impact on total performance. This will give us hints and clues on how disk I/O affects in later splitcube experiments. Data distributions are set as following:

- Equal length – Each of total 120 arrays has 1,000,000 integers
- Evenly spread – 120 arrays have the repeated sequence of 500 integers, 1,000 integers, 5,000 integers, 10,000 integers, 50,000 integers, 100,000 integers, 500,000 integers and 1,000,000 integers
- Highly skewed – one eighth of arrays in total 120 arrays will have 1,000,000 integers. The rest arrays will have 10 integers only for each.

5.2.2.1 Quick Sort Experiment Result and Analysis

Table 3 lists the experiment result of quick sort test in execution time both with and without disk I/O operations. Results show the parallel execution without disk I/O achieves a linear speedup while the parallel execution with disk I/O obtains a non-linear speedup. Results are also visually presented in Figure 14 and 15. It is clear that including disk I/O the total execution time is heavily
increased. Without disk I/O operation, all 3 tests are finished within a minute. But with disk I/O, the experiment on highly skewed data takes 7.73 minutes to finish while on equal length data it ridiculously takes almost an hour to finish. Such results suggest that disk I/O operation will have impacts on overall performance. But in what degree? We try to answer such question by further looking at the statistic data of CPU time and disk I/O captured by Windows performance monitor in quick sort test with disk I/O.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Without Disk I/O (in sec.)</th>
<th>With Disk I/O (in min.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>Parallel</td>
</tr>
<tr>
<td>Highly skewed</td>
<td>8.00</td>
<td>2.00</td>
</tr>
<tr>
<td>Evenly spread</td>
<td>16.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Equal Length</td>
<td>40.00</td>
<td>10.00</td>
</tr>
</tbody>
</table>

Table 3 Quick sort test – execution time with and without disk I/O

Figure 14 Quick sort test – execution time without disk I/O
CPU statistic data are listed in table 4 and disk I/O statistic data are listed in table 5. Comparisons are made between serial and parallel execution and the results are visually presented in Figure 16 and 17.

**Table 4**  Quick sort test – CPU time as percentage of total time

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Processor Time As Percentage of Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial (%)</td>
</tr>
<tr>
<td>Highly Skewed</td>
<td>25.514</td>
</tr>
<tr>
<td>Evenly spread</td>
<td>25.756</td>
</tr>
<tr>
<td>Equal Length</td>
<td>25.780</td>
</tr>
</tbody>
</table>

**Table 5**  Quick sort test result – disk statistics

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Disk Write Bytes/Sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
</tr>
<tr>
<td>Highly skewed</td>
<td>312,532,241</td>
</tr>
<tr>
<td>Evenly spread</td>
<td>257,535,138</td>
</tr>
<tr>
<td>Equal Length</td>
<td>250,001,606</td>
</tr>
</tbody>
</table>
Figure 16  Quick sort test – CPU time comparison

Figure 17  Quick sort test – disk I/O speed statistics

CPU statistic data show an increased use of CPU time for all 3 types of date in parallel execution at the ratio around 2.5. But such increased ratio
doesn't match the overall speedup ratio listed in Table 3. Disk I/O statistic data also shows an increased disk write speed in parallel execution. The disk I/O speedup ratio matches the overall speedup ratio. It suggests in quick sort test with disk I/O operation the overall speedup in parallel execution is bounded by the performance of disk I/O. This is visually presented in Figure 18.

![Quick Sort - Speed-up Ratio](image)

**Figure 18** Quick sort test – CPU, disk I/O and overall speedup ratio

To conclude quick sort test, we obtain the following results.

Without disk I/O, parallel execution achieves linear speedup

Disk I/O will have impacts on overall performance. This suggests that the overall performance of cube building and querying splitcube in later experiments could be also affected by disk I/O.

When disk I/O is heavy, the overall performance is bounded by the performance of disk I/O.
5.2.3 JSVM Video Coding Experiment

Quick sort is not a computational complex algorithm. Sorting hundreds of arrays with million integers is just a computational rich operation but is not a computational complex operation. This is different from our splitcube building program. In order to explore the performance of Microsoft Parallel Extension on computational heavy application, we also try with JSVM video coding as an example of applications with little disk I/O.

In JSVM video coding test, we will execute 8 parallel video coding tasks using JSVM software. Each task will produce a H.264/Scalable Video Coding (H.264/SVC) stream with two layers. Encoding configurations are stored in three files: general configuration file, layer 1 and layer 2 configuration file. Source videos are stored in two YUV files, one for QCIF7.5 and one for QCIF15. JSVM encoding program will load encoding instructions from configuration files and produces three files, two re-constructed YUV files for verification and evaluation and one final H.264 SVC video stream.

5.2.3.1 JSVM Experiment Result and Analysis

As in quick sort test, we also captured the CPU statistic data. Experiment results are summarized in Table 6.

Results show a linear speedup on overall execution time and CPU usage. JSVM video coding test indicates that with a computation-intensive application, we could achieve linear speedup if program is executed in parallel.
<table>
<thead>
<tr>
<th>Execution Time in min.</th>
<th>CPU Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% Processor Time</td>
</tr>
<tr>
<td>Serial</td>
<td>Parallel</td>
</tr>
<tr>
<td>Serial (%)</td>
<td>Parallel (%)</td>
</tr>
<tr>
<td>Serial (%)</td>
<td>25.407</td>
</tr>
<tr>
<td>Parallel (%)</td>
<td>98.477</td>
</tr>
<tr>
<td>Serial (%)</td>
<td>25.299</td>
</tr>
<tr>
<td>Parallel (%)</td>
<td>3.88</td>
</tr>
<tr>
<td>Ratio (P/S)</td>
<td></td>
</tr>
</tbody>
</table>

Table 6  JSVM test – CPU and execution time

5.3 Experiment of Splitcube Construction

5.3.1 Experiment Purpose

The purpose of experiment is to show with the use of Microsoft Parallel Extension and minor program changes whether or not we will be able to speed up the cubelet building process. If we can get the speedup can we achieve the linear speedup? Through experiments we also try to find out with different number of cubelets (result of different PDS settings), whether or not we will be able to speed up the cubelet building process in all cases. And if so, how much we can get? Will we get the same speedup with different size of cubelets?

5.3.2 Experiment Dataset

All experiments use the enhanced weather dataset ([HWL94]) as the data source. The weather dataset is a very common one for research on cube building algorithm to adopt for experimental analysis. The original dataset has 9 dimensions, whose cardinalities are: 2, 8, 10, 30, 101, 152, 179, 352, and 7037. Being different from the original data set, the enhanced weather dataset has one more level in the dimension hierarchy for the last 4 dimensions while the original dataset has 2-levels dimension hierarchies only for all dimensions i.e., the only group members are roots of the hierarchies. The cardinalities of all dimensions are shown in Table 7.
Cardinalities by levels in the dimension hierarchies in the enhanced weather dataset

There are about 1 million (1,015,638) tuples in weather dataset, i.e., about 1 million primary cells in base cube. If all group cells were to be pre-computed, the size of cube would be around 14.7 GB.

### 5.3.3 Experiment Settings

Experiment will be performed with the PDS length equal to 2, 3, and 4. Different PDS result in different total number of cubelets, ranging from 16 to 11668. Table 8 summarizes the selections of PDS for each experiment.

B-trees are built first as an in-memory data structure when building cubelets. If a B-tree is too large, it will be partitioned into multiple smaller ones. Thus there may be more than one file associated with one cubelet.

<table>
<thead>
<tr>
<th>PDS Configuration</th>
<th>Number of Cubelets</th>
<th>Number of B-tree Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDS_3 2 Dimension 1 and 2</td>
<td>16</td>
<td>751</td>
</tr>
<tr>
<td>PDS_5 2 Dimension 1 and 3</td>
<td>20</td>
<td>475</td>
</tr>
<tr>
<td>PDS_6 2 Dimension 2 and 3</td>
<td>80</td>
<td>672</td>
</tr>
<tr>
<td>PDS_7 3 Dimension 1, 2 and 3</td>
<td>160</td>
<td>785</td>
</tr>
<tr>
<td>PDS_11 3 Dimension 1, 2 and 4</td>
<td>480</td>
<td>480</td>
</tr>
<tr>
<td>PDS_35 3 Dimension 1, 2, and 6</td>
<td>2206</td>
<td>2202</td>
</tr>
<tr>
<td>PDS_15 4 Dimension 1, 2, 3 and 4</td>
<td>4192</td>
<td>4192</td>
</tr>
<tr>
<td>PDS_39 4 Dimension 1, 2, 3 and 6</td>
<td>11668</td>
<td>11668</td>
</tr>
</tbody>
</table>

Table 8 PDS configurations of experiments
Each test set will include 3 tests. One test is conducted with serial version of the program. The others are executed with the parallel version. Two parallel algorithms discussed in section 3.2.3 are used in the parallel part of experiments: pipeline execution with double buffer and duplicate input method. Both are using priority queue to implement task pool. Details of the algorithm can be found in section 3.2.3.

### 5.3.4 Experiment Results and Analysis

Table 9 lists the execution time in minutes and speedup ratio of each test for splitcube building experiments. Results show for all tests parallel execution achieves non-linear speedup. Speedup ratios vary on different PDS selections, ranging from 1.55 to 2.44.

<table>
<thead>
<tr>
<th>Test Set</th>
<th>Serial</th>
<th>Parallel</th>
<th>Double buffer</th>
<th>Duplicate input</th>
<th>Overall Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDS_3</td>
<td>20.75</td>
<td>12.9</td>
<td>1.61</td>
<td>12.67</td>
<td>1.62</td>
</tr>
<tr>
<td>PDS_5</td>
<td>28.97</td>
<td>19.12</td>
<td>1.52</td>
<td>18.3</td>
<td>1.58</td>
</tr>
<tr>
<td>PDS_6</td>
<td>32.15</td>
<td>13.42</td>
<td>2.40</td>
<td>15.3</td>
<td>2.10</td>
</tr>
<tr>
<td>PDS_7</td>
<td>23.90</td>
<td>10.13</td>
<td>2.36</td>
<td>9.47</td>
<td>2.52</td>
</tr>
<tr>
<td>PDS_11</td>
<td>38.82</td>
<td>24.03</td>
<td>1.62</td>
<td>22.55</td>
<td>1.72</td>
</tr>
<tr>
<td>PDS_35</td>
<td>22.10</td>
<td>12.83</td>
<td>1.72</td>
<td>12.07</td>
<td>1.83</td>
</tr>
<tr>
<td>PDS_15</td>
<td>24.52</td>
<td>12.05</td>
<td>2.03</td>
<td>11.88</td>
<td>2.06</td>
</tr>
<tr>
<td>PDS_39</td>
<td>16.55</td>
<td>7.43</td>
<td>2.23</td>
<td>6.7</td>
<td>2.47</td>
</tr>
</tbody>
</table>

Table 9 Splitcube build test – execution time

As we predicted in preliminary tests, building splitcube in parallel achieves non-linear speedup. But the question we would like to answer is why the parallel execution doesn’t achieve linear speedup. Is it because the computation of B-trees is not computationally heavy enough? Or is it because of the disk I/O?
What are the factors that decide the overall performance? We are trying to find the answer through the CPU and disk statistic data captured during experiments. Speedup ratio of total performance in execution time, ratio of increased CPU time and speedup ratio of disk I/O are visually presented in Figure 19.

![Speed-up Ratio](image)

**Figure 19  Splitcube building test – speedup ratio**

Preliminary tests with JSVM video coding indicates for computational heavy application executing program in parallel it is possible to achieve linear speedup on CPU usage. Although building B-tree is not as complex and heavy as the computation in video coding, such computation is still heavy enough to keep all CPU cores busy when running in parallel. Figure 19 shows close to linear speedup on CPU usage in splitcube building process. In most cases, parallel execution increases the CPU usage from 25% to above 95%. But the linear increasing usage of CPU doesn’t result in a linear speedup on overall performance. Figure 19 also shows the overall performance seems to be
bounded by the disk I/O performance. Unlike JSVM video coding test, building splitcube is also a disk I/O intensive task as it produces B-trees and store them in disk files. Like the experiment of quick sort with disk I/O, Figure 19 shows the total performance is heavily impacted by the intensive disk I/O.

<table>
<thead>
<tr>
<th>Test Set</th>
<th>Processor Time</th>
<th>User Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>Parallel</td>
</tr>
<tr>
<td>PDS_3</td>
<td>25.286</td>
<td>92.538</td>
</tr>
<tr>
<td>PDS_5</td>
<td>24.769</td>
<td>50.009</td>
</tr>
<tr>
<td>PDS_6</td>
<td>25.201</td>
<td>80.272</td>
</tr>
<tr>
<td>PDS_7</td>
<td>25.236</td>
<td>95.547</td>
</tr>
<tr>
<td>PDS_35</td>
<td>25.371</td>
<td>98.814</td>
</tr>
<tr>
<td>PDS_15</td>
<td>25.446</td>
<td>98.771</td>
</tr>
<tr>
<td>PDS_39</td>
<td>26.421</td>
<td>95.266</td>
</tr>
</tbody>
</table>

Table 10 Splitcube building test result – disk performance statistics

As one may see in Figure 19, the I/O bandwidth is heavily correlated with the speedup that is achieved by each splitcube, with the exception of PDS_5 which we will discuss in Section 5 of this chapter. Building a splitcube does speed up by parallelizing cubelet building tasks. But there are losses in each individual cubelet task. This is why parallelization doesn’t achieve linear speedup on overall performance. In fact experiment data show each cubelet building task takes more time to finish in parallel execution than it does in serial execution. But the loss has been compensated by running more tasks at the same time. That is why it still achieves speedup though the speedup is non-linear. Each cubelet building task has 2 types of operation that are using different resources: B-tree computation which uses CPU and loading/flushing data which involves disk I/O. Each task still uses single CPU for B-tree
computation thus it will take the same amount time to finish in both parallel and serial executions. But disk I/O speed of each task in parallel execution is lower than the disk I/O speed in serial execution. It is revealed by CPU and disk I/O statistics data captured during experiments, which show linear increasing of CPU usage and non-linear increasing of disk I/O speed. It is understandable as in parallel execution a cubelet building task will share disk I/O bandwidth with other tasks while in serial execution a single task exclusively has the entire I/O bandwidth. In most cases, B-tree computation of a single task takes more time than disk I/O does, thus there is a linear increasing of CPU usage. But in some case such as PDS_5, the disk I/O is so heavy which takes more time than B-tree computation does. Then there are times CPU is put in idle.

5.4 Experiment of Splitcube Query Processing

5.4.1 Experiment Settings

Experiment will be performed on the enhanced Weather dataset with the cubelet files we produced in splitcube building experiments. Three sets of query matrices, as listed in Table 11, will be produced using cubelet files generated during experiments of building splitcube. Each experiment with parallel execution will be performed twice, one with 4 parallel tasks and one with 8 parallel tasks.

<table>
<thead>
<tr>
<th>No</th>
<th>Column Dimension</th>
<th>Row Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>QM1</td>
<td>Present-weather</td>
<td>Station-id</td>
</tr>
<tr>
<td>QM2</td>
<td>Present-weather</td>
<td>Longitude</td>
</tr>
<tr>
<td>QM3</td>
<td>Weather-change-code</td>
<td>Hour</td>
</tr>
</tbody>
</table>

Table 11 Query matrix used in experiments
5.4.2 Experiment Results and Analysis

Experiment results on execution time are listed in table 12. Performance data show splitcubes that have more cubelet files take longer time to process queries. This is reasonable because a splitcube with fewer cubelets conduct more pre-aggregations than those splitcubes with more (see Chapter 2.6.). Result data on all 3 query matrices consistently show a slowdown when processing queries in parallel. More importantly result data show parallel execution takes much more time to finish processing queries than serial execution. Such results are visually presented in Figure 18 and Figure 19.

<table>
<thead>
<tr>
<th>Test Set</th>
<th># of Cubelets</th>
<th>Serial</th>
<th>Parallel</th>
<th>Ratio(S/P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDS_3</td>
<td>16</td>
<td>2,948</td>
<td>3,479</td>
<td>0.8474</td>
</tr>
<tr>
<td>PDS_5</td>
<td>20</td>
<td>1,701</td>
<td>2,294</td>
<td>0.7415</td>
</tr>
<tr>
<td>PDS_6</td>
<td>80</td>
<td>4,274</td>
<td>8,065</td>
<td>0.5299</td>
</tr>
<tr>
<td>PDS_7</td>
<td>160</td>
<td>7,769</td>
<td>14,695</td>
<td>0.5287</td>
</tr>
<tr>
<td>PDS_11</td>
<td>480</td>
<td>24,273</td>
<td>49,873</td>
<td>0.4867</td>
</tr>
<tr>
<td>PDS_35</td>
<td>2,342</td>
<td>25,616</td>
<td>34,617</td>
<td>0.7400</td>
</tr>
<tr>
<td>PDS_15</td>
<td>4,800</td>
<td>45,895</td>
<td>73,242</td>
<td>0.6266</td>
</tr>
<tr>
<td>PDS_39</td>
<td>23,420</td>
<td>65,208</td>
<td>69,810</td>
<td>0.9341</td>
</tr>
</tbody>
</table>

Table 13 Splitcube query process – execution time
We identify two factors that may explain the non-existent speedup in parallel query processing, compared with parallel cube building and quick sort. First of all, the CPU computation accounts for a very small percentage of the total
execution of a querying task. More importantly, read is the only I/O operation for query processing, while in other two applications, disk write operation dominates. Read must be done synchronously because the program can’t continue until the data are in the buffer. In cube building, the write operation associated with a cubelet can proceed in parallel to another task building another cubelet. The end result is the I/O bandwidth suffers a huge decrease from the serial version to the parallel version, as shown in Table 13. Disk performance statistic data captured in experiment show the overall performance is bounded by disk I/O performance. This is visually presented in Figure 20

<table>
<thead>
<tr>
<th>Test Set</th>
<th># of Cubelets</th>
<th>Disk Read Bytes/Sec</th>
<th>Ratio (P/S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Serial</td>
<td>Parallel</td>
</tr>
<tr>
<td>PDS_6</td>
<td>80</td>
<td>6,255,078.000</td>
<td>3,486,367.000</td>
</tr>
<tr>
<td>PDS_7</td>
<td>160</td>
<td>6,363,616.000</td>
<td>3,399,436.000</td>
</tr>
<tr>
<td>PDS_11</td>
<td>480</td>
<td>10,576,586.000</td>
<td>4,748,552.000</td>
</tr>
<tr>
<td>PDS_35</td>
<td>2,342</td>
<td>4,733,726.000</td>
<td>3,303,670.000</td>
</tr>
<tr>
<td>PDS_15</td>
<td>4,800</td>
<td>5,603,451.000</td>
<td>3,271,861.000</td>
</tr>
<tr>
<td>PDS_39</td>
<td>23,420</td>
<td>3,824,983.000</td>
<td>3,101,987.000</td>
</tr>
</tbody>
</table>

Table 14 Splitcube query process – disk I/O speed statistic data

![Figure 22](image)

Figure 22 Splitcube query process – CPU time and disk I/O speedup ratio
Result data in Figure 22 also show, with the increasing number of cubelet files, the slowdown in parallel execution alleviates. In another word, performance of parallel execution improves with the increasing number of cubelet files. In some cases, parallel execution even achieves tiny speedup. We have the similar yet not identical symptom in splitcube building experiments, parallel executes achieve different speedup ratio. The related question is what factors actually dominate the performance of disk I/O. This is the question we will discuss and try to answer in next section.

5.5 Guideline for Choosing an Appropriate Splitcube

Development of a general strategy to identify a good PDS for both cube building and query processing on a multi-core system is beyond the scope of this research study. Nonetheless, based on the experiments we have performed on a variety of splitcubes based on a real-life dataset, we can rank the splitcubes on the basis of their cube building and query processing performance. Hopefully, our findings will help developing an optimization strategy to select a good PDS.

Until a better query processing strategy is found, it appears that we are better off sticking to the serial version. Hence, the splitcube that includes more aggregates in general do better in query processing. Thus the splitcubes with a large number of splitcube will not fare well. From the cube building performance data, the splitcubes that include most aggregates suffer in a multi-core system due to the limited bandwidth of the system. The splitcube PDS_7 achieves the best speedup because it has a moderate number of cubelets, which is also a factor contributing to good query processing performance (for the serial version).
We notice some similarities between two studies of splitcube computation on the multi-core and cluster platform. The number of cubelets of splitcube is a crucial factor for securing good performance in both studies. Another factor is the maximum size of a cubelet. In the case of PDS_5, the maximum size of the cubelet accounts for approximately 5% of the total size of the splitcube, causing load imbalance.

<table>
<thead>
<tr>
<th>Test Set</th>
<th>Number of Cubelets</th>
<th>Number of files</th>
<th>Total Size (GB)</th>
<th>Max. File Size (KB)</th>
<th>Min. File Size (KB)</th>
<th>Avg. File Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDS_3</td>
<td>16</td>
<td>751</td>
<td>4.10</td>
<td>87103</td>
<td>8</td>
<td>5724.58</td>
</tr>
<tr>
<td>PDS_5</td>
<td>20</td>
<td>475</td>
<td>4.25</td>
<td>200992</td>
<td>28</td>
<td>9382.00</td>
</tr>
<tr>
<td>PDS_6</td>
<td>80</td>
<td>672</td>
<td>4.37</td>
<td>69138</td>
<td>28</td>
<td>6818.86</td>
</tr>
<tr>
<td>PDS_7</td>
<td>160</td>
<td>785</td>
<td>2.85</td>
<td>64218</td>
<td>8</td>
<td>3806.93</td>
</tr>
<tr>
<td>PDS_11</td>
<td>480</td>
<td>480</td>
<td>3.21</td>
<td>17216</td>
<td>16</td>
<td>7012.35</td>
</tr>
<tr>
<td>PDS_35</td>
<td>2432</td>
<td>2206</td>
<td>1.74</td>
<td>7412</td>
<td>8</td>
<td>827.07</td>
</tr>
<tr>
<td>PDS_15</td>
<td>4800</td>
<td>4192</td>
<td>1.66</td>
<td>7560</td>
<td>8</td>
<td>415.23</td>
</tr>
<tr>
<td>PDS_39</td>
<td>24320</td>
<td>11668</td>
<td>0.96</td>
<td>3156</td>
<td>8</td>
<td>86.44</td>
</tr>
</tbody>
</table>

Table 15  Cubelet file size comparison between different PDS

![Total Number of Files](image_url)  

Figure 23  Total number of cubelet files for different PDS
Figure 24  Total size for different PDS

Figure 25  File size of the maximum cubelet for different PDS
Figure 26  Average file size for different PDS
CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this paper, we demonstrate a parallel implementation of splitcube for building an OLAP database engine on an entry-level, no-server class multi-core system. Our objective is to investigate if, with partial pre-aggregation and parallel computing, it is achievable to speed up cube building process and query matrix processing. Our experiments show running splitcube building algorithm in parallel achieves non-linear speedup on multi-core system, and the speedup is bounded by the performance of disk I/O. Our experiments also show querying splitcube in parallel on multi-core system doesn’t achieve speedup, more importantly it suffers slowdown in parallel execution. Thus it is recommended not to run query processing in parallel until a better query processing strategy is found on multi-core system.

In a broader perspective, what we learn from conducting this research is that the data-intensive application without excessive demand on disk I/O could utilize the extra computation power brought in by extra CPU cores to speed up its overall performance if it can be parallelized. With the minimum change to the algorithm by practicing task parallelism, such application could achieve non-linear speedup on multi-core system. On the other hand, improvements to the overall performance and the speedup of parallelization on data-intensive application is
still limited and is heavily impacted by the limited memory bandwidth, memory-management schema, and disk I/O that are out-paced by the development of single-chip multi-core on the entry-level SMP system. In our experiments, a noticeable fact is that each parallelized task takes longer time to execute than its serial counterpart. On a cluster system the performance gain of parallelization will be level off by the overhead of managing too many tasks. Our experiments on multi-core system reveal the same symptom, emphasizing the importance of the selection of goods.

Clearly, this research is a proof-of-concept work, leaving much room for further refinements. For example, legacy c style coding could be changed and improved to leverage the advantages of .Net framework and Parallel Extension. Algorithm can be further improved to use more memory that is cheap and largely available on today’s multi-core system. New strategies that promote higher disk I/O efficiency such as the scheme suggested in [OU06] should be tried out. As cluster systems are getting more popular and nodes in cluster are likely to be replaced by cheap multi-core systems the design must be adjusted continually to take advantages of both systems and make the efficient use of new hardware.
REFERENCES


[DE09] D. Eadline, "Scaling Bandwidth, SMPs may have (one), but clusters have bandwidth", http://www.linux-mag.com/id/739


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


