

# Hash Based Caching Mechanism for Web-OLAP

by

Bai Yu

B.Eng, Zhejiang University, 2011

Thesis Submitted in Partial Fulfillment  
of the Requirements for the Degree of

Master of Science

in the  
School of Computing Science  
Faculty of Applied Sciences

© Bai Yu 2014

SIMON FRASER UNIVERSITY

Fall 2014

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:** Bai Yu  
**Degree:** Master of Science  
**Title of Thesis:** Hash Based Caching Mechanism for Web-OLAP

**Examining Committee:** Dr. Arrvindh Shriraman, Assistant Professor  
Computing Science, Simon Fraser University  
Chair

---

Dr. Wo-shun Luk, Professor  
Computing Science, Simon Fraser University  
Senior Supervisor

---

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

---

Dr. Ke Wang, Professor  
Computing Science, Simon Fraser University  
External Examiner

**Date Approved:** September 22nd, 2014

## Partial Copyright Licence



The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the non-exclusive, royalty-free right to include a digital copy of this thesis, project or extended essay[s] and associated supplemental files ("Work") (title[s] below) in Summit, the Institutional Research Repository at SFU. SFU may also make copies of the Work for purposes of a scholarly or research nature; for users of the SFU Library; or in response to a request from another library, or educational institution, on SFU's own behalf or for one of its users. Distribution may be in any form.

The author has further agreed that SFU may keep more than one copy of the Work for purposes of back-up and security; and that SFU may, without changing the content, translate, if technically possible, the Work to any medium or format for the purpose of preserving the Work and facilitating the exercise of SFU's rights under this licence.

It is understood that copying, publication, or public performance of the Work for commercial purposes shall not be allowed without the author's written permission.

While granting the above uses to SFU, the author retains copyright ownership and moral rights in the Work, and may deal with the copyright in the Work in any way consistent with the terms of this licence, including the right to change the Work for subsequent purposes, including editing and publishing the Work in whole or in part, and licensing the content to other parties as the author may desire.

The author represents and warrants that he/she has the right to grant the rights contained in this licence and that the Work does not, to the best of the author's knowledge, infringe upon anyone's copyright. The author has obtained written copyright permission, where required, for the use of any third-party copyrighted material contained in the Work. The author represents and warrants that the Work is his/her own original work and that he/she has not previously assigned or relinquished the rights conferred in this licence.

Simon Fraser University Library  
Burnaby, British Columbia, Canada

revised Fall 2013

# Abstract

In recent years, the advent of client-side data caching techniques has brought significant performance improvement in highly interactive web applications.

In this thesis, we focus on the client-side data caching mechanism in the web-based OLAP system. Previous work or methods on this area lack the universality of dealing arbitrary *shapes* of the cached data. For instance, in 2 dimensional data, it can only calculate the overlaps between two rectangles. While in reality, in most cases, the cached data can be any sizes and *shapes*. Therefore, we propose a new client-side hash based caching mechanism that can deal with arbitrary *shapes* of the dataset. Also, we come up with the idea of downloading the inflated sub-cubes in background and serve the users first. By combining this idea with our hash based caching mechanism, we can not only serve the users at the very first time, but also provide high performance for the subsequent cross-tab or drill-down operations by utilizing the cached inflated sub-cubes.

Our result shows that this new method has great performance improvement compared with the previous method and when the size of the inflated sub-cubes is not so large, our mechanism also outperforms the no-cached one.

**Keywords:** Web-OLAP; MDX query; client-side caching; Query processing

*To whomever whoever reads this!*

*“Don’t worry, Gromit. Everything’s under control!”*  
— *The Wrong Trousers*, AARDMAN ANIMATIONS, 1993

# Acknowledgments

I would like to thank Dr. Arrvindh Shriraman for taking time out of his schedule to serve as the chair of the committee. I am also thankful for Dr. Jian Pei for accepting to be my supervisor and Dr. Ke Wang for being my external examiner and dedicating their valuable time to review my thesis.

I also want to thank Elahe Kamaliha, the former labmate, for her generous help and guidance to help me start with my thesis.

My special gratitude goes to my senior supervisor, Dr. Wo-Shun Luk, who has offered invaluable guidance and support to me through out my thesis from the beginning to the end with his knowledge and patience. Whenever I needed help, he made his time available and helped me with his valuable guidance and advice to tackle any problems I met. It was not possible for me to finish this journey without his warm encouragement and nice help.

# Contents

<b>Approval</b>	<b>ii</b>
<b>Partial Copyright License</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>Dedication</b>	<b>v</b>
<b>Quotation</b>	<b>vi</b>
<b>Acknowledgments</b>	<b>vii</b>
<b>Contents</b>	<b>viii</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Figures</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Client-side data caching . . . . .	1
1.2 Motivation and Objective . . . . .	1
1.3 Related Work . . . . .	3
1.4 Thesis Organization . . . . .	4
<b>2 Overview of the Client-Server System</b>	<b>6</b>
2.1 Web-OLAP Architecture . . . . .	6
2.2 Web-OLAP Server . . . . .	7
2.2.1 OLAP Server . . . . .	7
2.2.2 Web Service . . . . .	7
2.3 Web-OLAP Client . . . . .	8
2.3.1 UI . . . . .	8



2.4	Client-Server communications . . . . .	8
2.4.1	Issue a Request . . . . .	9
2.4.2	Generate MDX Query . . . . .	9
2.4.3	Send MDX Query to Server-side . . . . .	10
2.4.4	Processing the XMLA Message . . . . .	10
2.4.5	Receive the Cell Table . . . . .	11
<b>3</b>	<b>Hash Based Caching Mechanism for Web-Based OLAP</b>	<b>12</b>
3.1	Terminologies and Concepts . . . . .	13
3.1.1	Cell . . . . .	13
3.1.2	Query Cell . . . . .	14
3.1.3	Answer Sub-cube . . . . .	14
3.1.4	Inflated Sub-cube . . . . .	14
3.2	Hash Based Caching Architecture . . . . .	15
3.3	Aggregate Cells . . . . .	16
3.3.1	Problem Formalization . . . . .	16
3.3.2	Sort-Merge Algorithm . . . . .	16
3.4	Generate Queries . . . . .	19
3.5	Data Split . . . . .	20
3.5.1	CellOrdinal Rewrite . . . . .	20
3.5.2	Evaluation . . . . .	23
3.6	Why Sort-Merge algorithm? . . . . .	25
3.6.1	Brute-force Search . . . . .	25
3.6.2	No Aggregations . . . . .	28
3.6.3	Performance Evaluation . . . . .	28
3.6.4	Effectiveness . . . . .	29
3.7	Cache vs. No-Cache Comparison . . . . .	31
<b>4</b>	<b>Query Processing</b>	<b>33</b>
4.1	Query Categorization . . . . .	33
4.2	Subset Query . . . . .	33
4.2.1	Identify the Parent Cell . . . . .	34
4.2.2	Extract the Subset Data . . . . .	35
4.3	Overlap Query . . . . .	36
4.4	Drill-down Query . . . . .	37
4.4.1	Finding the Touched Cells . . . . .	38
4.4.2	Grouping the Sub-cells . . . . .	39
4.4.3	Extracting the Data . . . . .	39

4.4.4	Evaluation . . . . .	40
<b>5</b>	<b>Downloading the Inflated Sub-cube in Background</b>	<b>42</b>
5.1	User Access Pattern – Serve the Users First . . . . .	42
5.2	Implementation . . . . .	43
<b>6</b>	<b>User Workflow</b>	<b>45</b>
6.1	Performance Factors . . . . .	45
6.2	Workflow Scenario . . . . .	46
6.3	Evaluation . . . . .	47
<b>7</b>	<b>Conclusion and Future Work</b>	<b>49</b>
7.1	Conclusion . . . . .	49
7.2	Future Work . . . . .	50
	<b>Bibliography</b>	<b>51</b>
	<b>Appendix A Partial XMLA Response</b>	<b>53</b>
	<b>Appendix B Tables for Data Split Experiment</b>	<b>55</b>
	<b>Appendix C User Workflow for Drill-downs</b>	<b>58</b>

# List of Tables

3.1	Evaluation Environment . . . . .	23
3.2	Split Time with different Cells & Dimensions . . . . .	24
3.3	Sort-Merge Effectiveness for 2-dimensional Data . . . . .	30
3.4	Sort-Merge Effectiveness for 3-dimensional Data . . . . .	30
5.1	Query Processing Comparison . . . . .	44
6.1	Sample Case for User Workflow . . . . .	46
6.2	Workflow Summary . . . . .	47
B.1	Data Split Experiment for 2-Dimensional Data . . . . .	55
B.2	Data Split Experiment for 3-Dimensional Data . . . . .	56
B.3	Data Split Experiment for 4-Dimensional Data . . . . .	57
C.1	User Workflow for Drill-downs . . . . .	58

# List of Figures

2.1	System Components of a Client-Centric OLAP System . . . . .	6
2.2	Web-OLAP UI . . . . .	9
2.3	Left: A complete fact table with 3 dimensions and a measure. Right: Corresponding cell table for this fact table. . . . .	11
3.1	2-Dimensional Example . . . . .	13
3.2	Terminology: Cell . . . . .	13
3.3	Hash Table of the Caching Mechanism . . . . .	15
3.4	The Procedure of Our Caching Mechanism . . . . .	16
3.5	Caching Mechanism Example . . . . .	17
3.6	Sort-Merge Algorithm . . . . .	17
3.7	Example for Sort-Merge algorithm fails to find the optimal solution . . . . .	19
3.8	Aggregated Result for Each Cell . . . . .	21
3.9	Underlying Level Measures and the Corresponding <i>CellOrdinal</i> Values . . . . .	21
3.10	Measures and the Corresponding <i>CellOrdinal</i> Value After <i>CellOrdinal Rewrite</i> Phase . . . . .	21
3.11	Example used to compare three different strategies . . . . .	25
3.12	Brute-Force Algorithm: Exponential Time . . . . .	26
3.13	Pre-process Time Comparison . . . . .	28
3.14	Data Download Time Comparison . . . . .	29
3.15	Example for different cached percentage . . . . .	31
3.16	Data Download Size Comparison . . . . .	32
3.17	Time Comparison . . . . .	32
4.1	Drill Down: Row . . . . .	37
4.2	Drill Down: Cell . . . . .	37
4.3	Drill Down Example . . . . .	38
4.4	Drill Down Time Comparison . . . . .	40

# Chapter 1

## Introduction

### 1.1 Client-side data caching

With the advent of AJAX technology, client-side data caching has emerged as a popular technique in modern highly interactive web applications to improve performance, because the network communication cost is always the bottleneck for most of the web systems. By storing and utilizing the cached data in the client-side, these web applications can save a lot of time by eliminating the round trip time of retrieving data from the server side.

In this thesis, we focus on the client-side caching technique for web-based OLAP (OnLine Analytic Processing). This system is typically a client-centric client-server OLAP which includes a light-weight OLAP processor (in JavaScript) to perform OLAP functions, e.g., cross-tab on data downloaded from the server. This type of OLAP system was first proposed in [17].

### 1.2 Motivation and Objective

The main objective of this caching mechanism is to provide an environment where the user can explore (i.e., to operate on) a data cube on the client side, without any reliance on the server, for sake of enhancing user experience. A user of this system would first submit a summary query, e.g., "what is the total sales of all bikes in 2007?". Upon the return of the answer from the OLAP server, the user may submit a more detailed query, e.g., "what is the sales mountain bikes for the first quarter?". This time around, this drill-down query, or any other subsequent drill-down queries, would be answered from the data in the cache, without a round trip to the server.

A straightforward design for this caching mechanism is to download not just the answer to the summary query, but also other data that includes all leaf nodes of the dimension hierarchies of all dimensions included in the summary query. For example, the example summary query includes two

dimensions: Time and Product. The server is to return sales information about all kinds of bikes (including mountain bikes) in all months and all days of 2007. This (inflated) answer would be stored in the cache. Then, the summary query and all subsequent drill-down queries would be answered with data from only the local cache.

This scheme has been implemented and shown to work (Elahe's thesis, [9]), but there are downsides to it too. First and foremost, the caching system can only hold the data for just one query. Any subsequent data downloaded from the server will be discarded and cannot be cached for future use. Besides, the user must accept the trade-off between longer processing time for the summary query and the benefits of having all drill-down data in the local cache. Besides, the cache is built primarily for the purpose of supporting the drill-down operation, and consequently it can not support other query processing functions.

In this thesis, we propose a hash based caching mechanism that can store the data for multiple queries. Basically, we treat the intersection of every hierarchy in each dimension as a separate unit. Each unit has its own `key` (the concatenation of every hierarchy id from each dimension) and `value` (the inflated data). We split the query result to one or more units and store all of them into our hash table. So, the next time, if the user issues a query that includes a unit already in our cache, we can easily identify and exclude it. Then, the main task is to aggregate the un-cached units to fewer number of *larger units* (the query cell in section 3.1.2). Next, our system will generate the queries for each of these large units and send them to the server side. These queries can be called as remainder queries [7]. A remainder query is required when the user's query cannot be answered with only the data residing on the client side. After the inflated sub-cubes are downloaded, they would be split so that each unit points to its own portion of data. Finally, each unit will be inserted into our hash table for future use. Our hash based caching mechanism can handle any type of queries, see chapter 4 and has better drill-down performance compared to the previous method, see section 4.4.

The research reported in this thesis is also motivated by the realization that the user need not accept the trade-off as described above. In a typical drill-down situation, we figure out that the user will have to take some time to issue a drill down query after reading result of the summary query. Let's call this time gap between presentation of the summary result to the user and the submission of the first drill-down query by the user as the *think time*, which incidentally includes the time for the user to decide which row, column or unit to expand on the result table. It is not unreasonable to assume the *think time* to be 3 seconds on average. Within this interval, the client is capable of downloading from the OLAP server quite a few megabytes of data in the background. Accordingly, the client should send to the server two queries to the server: once we get the result for the user's original query and the query retrieving the inflated answer will be sent to the server immediately. This process can go on until the entire answer of the latter query is stored in the local cache. From that point on, any drill-down queries can be processed with the data in the local cache. Thus, the

user need not suffer any delay due to the task of download the inflated sub-cubes.

### 1.3 Related Work

This research is very closely with the research reported in [9]. The differences between the two will be elaborated in chapter 2. For the rest of this chapter, we will describe how this research is related with other research reported elsewhere on data cache and data caching.

The concept of the data caching has evolved over years. In the beginning, a relational database system relied on data caching to reduce the traffic between the memory and the disk. A database consists of a number of data pages, which are moved between the buffer pools and disk storage. With the increase in the size of a buffer pool, the cache hit ratio has gradually increased to well over 90%. This is how the data caching of most commercial relational database and OLAP systems by and large work. The idea of data caching on the client side emerged in late 80's as a new technique in an attempt to bolster the performance of a relational database system. With the shift of database architecture from server-based to client-server architecture, it was only natural to look for ways to utilize the considerable computational power in the client to achieve higher overall system performance. Instead of having queries processed in the server, the data are shipped to the client for processing. In other words, after a query has been submitted at the client, the query is processed locally, using the data inside the cache on the client side. If necessary, a new query, which is called remainder query, is sent to the server to retrieve the required data that is missing from the local cache. The data cache on the client side does not always contain results of an SQL query. Instead it may contain rows that satisfy a certain predicate in the where-clause of a query. This is called semantic caching [5] [10].

During the early days of OLAP systems, many (e.g., [11] [14] [13]) have proposed a data caching for OLAP data on the server side, where the basic object in the data cache is a materialized view, i.e., a data cube, or some subset thereof. In this sense, they are considered to be some kind of semantic caching. They are mostly concerned with cache admission and eviction policies. Some authors have considered caching of data cubes in a distributed setting [8]. To our best knowledge, there is not any research on data caching of data cubes on a single-user client.

In [15], the authors consider the challenges of semantic caching on the server side of a relational or OLAP database systems. Unlike a cache of data pages, a cache of data objects is harder to manage, because they may be related to each other. This factor complicates the admission and eviction policies for general-purpose data caching. On performance of semantic caching on the client side, it is reported in [7] that when the number of attributes of an SQL query is 4 or greater, the query response time is about the same as in the non-cached architecture, as the cache management overhead on the client side eats up all the savings due to semantic caching.

Over the last few years, the platform of a client in a client-server database system has shifted

to a mobile device, which communicates with the OLAP server over Internet. The client runs inside a generic browser, or as a dedicated application on the operating system of the mobile device. The primary objective of a data cache for this type of system is enhancement of user experience [6], although performance is part and parcel of this objective. For this research, we don't aim at promoting overall system performance. The server indeed has to do more work, just to ensure that the user will not see any deterioration in query response time. In many ways, our data caching works similarly as semantic caching on the client side in other systems described above. However, we have chosen to organize our cache data as cells, which are connected to other related cells by hashing. Overall, we maintain a low cache management overhead.

## 1.4 Thesis Organization

The organization of the rest of this thesis is as follows: in chapter 2, we talk about the system overview of the Web-OLAP system. Each component in both the client and server side is explained in details and the procedures of query from the time it is issued from the web UI to the moment the client receives the processed data are also discussed.

In chapter 3, the overall system architecture of our hash based caching mechanism will be illustrated in details. In this chapter, we introduce some terminologies and concepts first and then describe how the cells are stored in our cache and how we can quickly index one if given a specific cell id (or the *key*) to exclude the cached ones. Besides, we talk about all the subsequent phases: aggregating cells, generating queries, splitting data and cells integration to illustrate how a query is processed and optimized in the client side. Also, we propose an algorithm called *Sort-Merge* to aggregate the un-cached cells to minimize the number of queries sent to the server side. We also show that this algorithm is a trade-off considering the pre-process time and the data download time plus the split time by comparing with the other two methods.

In chapter 4, we categorize the users' queries to 3 different types and discuss the strategies of how to process each of them. In addition, we explain how to utilize the global caching hash table to speed up the subsequent queries in a user workflow. We propose a method that can quickly locate the touched cells for the drill-down operations to narrow down search domain. The result indicates that this method based on our new hash based caching mechanism outperforms the previous one.

In chapter 5, we come up with a new idea of downloading the answer sub-cubes for the initial query and then downloading the inflated sub-cubes in background. By applying this idea, our system can not only serve the users at the very first time, but also provide better performance for the subsequent cross-tab or drill-down operations by utilizing the inflated sub-cubes.

In chapter 6, we talk about the user workflow and perform experiments comparing the three different methods. Our result shows that our hash based mechanism can provide faster response time for the drill-down requests and therefore better user experience.



In the last chapter, we discuss the conclusion of this thesis. We also present some points and ideas need to improve or deserve continuing working on in the future work section.

## Chapter 2

# Overview of the Client-Server System

### 2.1 Web-OLAP Architecture

The architecture of the system under investigation in this research is shown in Figure 2.1.

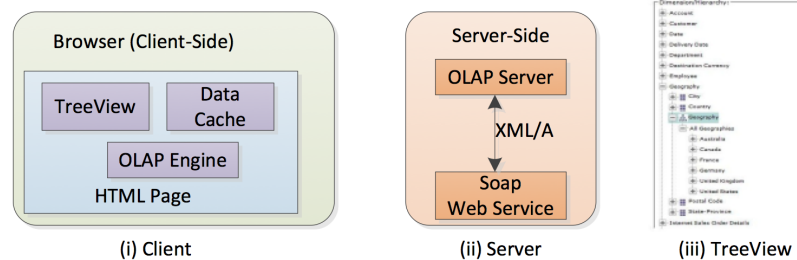


Figure 2.1: System Components of a Client-Centric OLAP System

On the server side, there are two major components. The OLAP Server is a regular, off-the-shelf product. In our prototype system, it is the SQL Server Analytic Services 2008R2. (In this paper, we use the sample data cube AdventureWorks that comes with the server for experimental purposes). The Soap Web Service is a custom-built software module, which acts as in-between the OLAP Server and the client, and provides some value-added functions to the client. In our case, the data released from the OLAP server will be further processed before being forwarded to the client. Compression/de-compression is another example function. XMLA (XML for Analysis) [16] is the de facto communication protocol between OLAP Server and the outside world. Metadata and data stored inside the OLAP Server can be queried via XMLA.

Unlike components on the server-side, the components on the client-side of our system are very different from those on a traditional client, which are just a HTML page, plus a UI component. Even the UI in our client is different from the traditional (server-centric) client-server system. It incorporates a component called TreeView [16] (see Fig. (iii) in Figure 2.1), developed by Yahoo, which serves two purposes: First, the TreeView may be programmed to extract the metadata from the OLAP server, and populate it on the client side. Second, it provides a convenient way for the user to enter the query, which is captured by the client. Together with the metadata stored by TreeView, the client will be able to figure out exactly the structure of the answer to come from the OLAP server. If the client finds out that the answer can be derived entirely from data already in cache, the user's query will not be forwarded to the server. On some other occasions, the client may substitute the user's query with another one that fetches more, or less, data than necessary.

The client communicates with the Soap Web Services using the AJAX API.

## **2.2 Web-OLAP Server**

### **2.2.1 OLAP Server**

In the past decades, the online analytical processing (OLAP) has emerged as an approach to answering multi-dimensional analytical (MDA) queries swiftly [3]. Different from the traditional relational database, OLAP is specifically for data analyzing. Typical applications of OLAP include business reporting for sales, marketing, management reporting, business process management and etc [1]. And for OLAP databases, as they serve as data analysis, they do not support write operations most of time [18]. Basically, OLAP consists of three basic analytical operations: roll-up, drill-down, and slicing and dicing [12]. Our Web-OLAP uses the SQL Server Analytic Services 2008R2 as the OLAP data provider which is responsible for answering the requests from the web service layer in the server side.

### **2.2.2 Web Service**

The Soap Web Service is a middleware between the OLAP Server and the client. The web service layer provides a set of functions that can be invoked from the remote distance client. It listens to the clients' requests, receives the data and then translate the requests by using the appropriate XMLA messages and send them to the analytic data provider to retrieve the result.

#### **XMLA**

XMLA stands for XML for Analysis, which is an industry standard for data access in analytical systems, such as OLAP and data mining. XMLA is based on other industry standards such as

XML, SOAP and HTTP. Currently, XMLA is maintained by XMLA Council with Microsoft, Hyperion and SAS being the official XMLA Council founder members [2].

XMLA consists of only two methods, which are *Discover* and *Execute* respectively [4].

*Discover* method allows the applications to request the metadata or other restrictions and properties from the OLAP server. In our Web-OLAP system, metadata includes information like dimensions, hierarchies and measures.

*Execute* method allows the applications to execute an MDX query on the OLAP server. One example of the result of the XMLA *execute* for an MDX query shown is included in appendix A.

After the web service layer receive the XMLA result from the OLAP server, it will do some process to the XMLA messages to reduce massive networking communication cost. Detailed procedures are described in section 2.4.4.

## 2.3 Web-OLAP Client

### 2.3.1 UI

Figure 2.2 shows a screen-shot of the Web-OLAP UI in the web browser. Users can issue a MDX query by selecting the hierarchies in the *Tree View* structure on the left hand side of the UI, which is an open source TreeView from Yahoo! UI 2 [16]. Any selected hierarchies will be displayed in the *Hierarchy Selection* box. Users can also select the measures they are interested in. And the results will be displayed in the *Query Result* box. The *Log Message* box can output some logs from our client-side query engine, which is written with JavaScript.

The *Tree View* only holds and displays the dimension names at the very top level when the *Server*, *Catalog* and *Cube* are selected. This is because the complete hierarchies in every dimension is huge and there's no need for our Web-OLAP to store these additional information if the users never dive deep into parts of the dimensions. Instead, the *Tree View* can download the sub-hierarchies on the fly using the JSOC whenever the users try to expand the lower levels by clicking the + symbol on the left side of each internal label.

## 2.4 Client-Server communications

In this section, we go through a query issued from the Web-OLAP UI client to the server application. We describe the process of back and forth communications between the client and server until the results are displayed to the user by tracking the lifetime of a query.

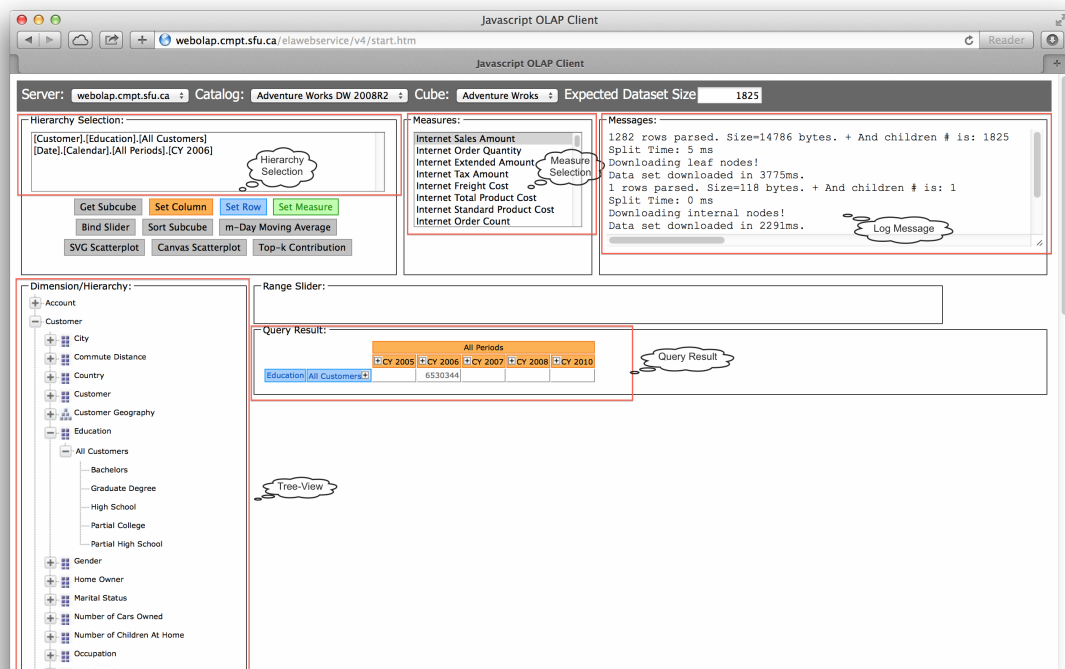


Figure 2.2: Web-OLAP UI

### 2.4.1 Issue a Request

As we stated in section 2.3, a user can select the hierarchies and measures in some dimensions in the *Tree View* and *Measures Selection* box. The selection can be treated as requesting a cube from the server.

### 2.4.2 Generate MDX Query

The Multidimensional Expressions (MDX) is the query language for OLAP databases that allows the users to retrieve multidimensional objects, such as cubes, and return multidimensional cell sets that contain the cube's data. For example, if the users choose the measure and hierarchies below:

Measure value: Internet Sale Amount

Dimension 1 (ID: 193): [Customer].[Education].[All Customers]

Dimension 2 (ID: 199): [Product].[Color].[All Products]

The corresponding MDX query would be:

```
select {[Measures].[Internet Sales Amount]} on axis(0),
{[Customer].[Education].[All Customers]} on axis(1),
{[Product].[Color].[All Products]} on axis(2) from [Adventure Works]}
```

### 2.4.3 Send MDX Query to Server-side

The MDX query will be sent to the server-side and been enveloped in a XMLA execute message by the web service and then been sent to the OLAP server. The OLAP server will execute the MDX query and return the multidimensional dataset with the form of an XMLA message to the web service. The XMLA message contains lots of detailed information, see Appendix A as a reference. This example is taken from Elahe's thesis [9].

### 2.4.4 Processing the XMLA Message

Elahe in her thesis [9] proposed a new framework to process the XMLA message differs from the original work by Hsiao's system [17]. In the original system, the fact table (the table to be displayed to the user) is generated from the dataset prior to sending it to the client. While, in Elahe's new system, it converts the XMLA message to a cell table.

#### CellOrdinal

In the XMLA message returned from the OLAP server to the web service (appendix A), every entry (cell) is addressed in a special format: each cell is uniquely identified by a numeric value named *CellOrdinal*. See the example below:

```
<CellData>
  <Cell CellOrdinal="0">
    <Value xsi:type="xsd:decimal">7267018.3655</Value>
    <FmtValue>$7,267,018.37</FmtValue>
  </Cell>
  .....
</CellData>
```

This number specifies the exact position of an entry in a sub-cube as if the multidimensional data is a  $d$ -dimensional array and the array is traversed in a row-major. Each entry in the XMLA message corresponds to a row in the fact table.


In Elahe's framework, the XMLA message will be parsed to a cell table. Figure 2.3 illustrates an example of how a 3-dimensional dataset to converted to cell table.<sup>1</sup>

Compared to the XMLA message or the fact table, the cell table avoids massive additional networking communication. And on average, the size of the cell table is 30% to the size of the fact table used in [17].

---

<sup>1</sup>This figure is taken from [9]

Geography	Date	Product	Sales Amount
Australia	2005	Bike	M0
Canada	2005	Bike	M1
France	2005	Bike	M2
Australia	2006	Bike	M3
Canada	2006	Bike	M4
France	2006	Bike	M5
Australia	2007	Bike	M6
Canada	2007	Bike	M7
France	2007	Bike	M8
Australia	2005	Clothing	M9
Canada	2005	Clothing	M10



CellOrdinal	Sales Amount
0	M0
1	M1
2	M2
3	M3
4	M4
5	M5
6	M6
7	M7
8	M8
9	M9
10	M10

Figure 2.3: Left: A complete fact table with 3 dimensions and a measure. Right: Corresponding cell table for this fact table.

### 2.4.5 Receive the Cell Table

In our implementation, we also use the cell table as the format for the server-client communication. And when the cell table transmitted to the client side, the query engine should be able to process it and perform aggregation on it. The detailed procedure is described in chapter 3.

## Chapter 3

# Hash Based Caching Mechanism for Web-Based OLAP

As described in [9], when the user requests a new inflated answer sub-cube, it is possible that a fraction of the cube already exists at the client's memory within an already cached sub-cube. And the basic benefit of introducing a client-side caching mechanism for web-based OLAP is to help the users to get rid of the phase of executing the query on the OLAP server again if the data is already cached. Generally, the network communication is the bottleneck for most of the highly interactive web applications. Utilizing the cached data and retrieve them locally from the client side could significantly reduce the time of communicating to the server and therefore highly increase the performance as well as providing better user experience.

Elahe, in her thesis [9], proposed an algorithm to calculate the overlaps between two different cubes. This algorithm has  $O(mn)$  time complexity, where  $n$  is the number of dimensions and  $m$  is the number of hierarchies in each dimension. This algorithm runs fast. However, it can only calculate the overlaps between two regular *shapes*. For instance, in 2-dimensional table, it can only compare two rectangles. See Figure 3.1 for the cube 1 and cube 2.

But when the cached data is not a regular *shape*, for instance, in 2-dimensional data, not a rectangle, the algorithm is not applicable. And in reality, the cached data is most likely not a regular *shape* when the user keeps retrieving new part of data in the same dimensions.

So, we come up with a new mechanism, which is our hash based mechanism, that can deal with any *shape* of cached data and meanwhile has higher performance for the subsequent drill-downs.

In this chapter, we will talk about the detailed architectures of our hash based caching mechanism based on the cell table we received from the web service. We will demonstrate the procedure of how a query is processed in the client-side and how we build/update our hash table after we get



		Colors		
		200 Black	201 Blue	205 Red
Education	194 Bachelor	3099617	756587	
	195 Graduate	1589316	492745	Sub-Cube 2
	196 High School	1353089	348869	
	197 College		Sub-Cube 1	

Cube 1 
                 
 Cube 2

Figure 3.1: 2-Dimensional Example

the result from the server. Before introducing the details, let's firstly take a look at some terminologies and concepts that would be used in the following sections.

### 3.1 Terminologies and Concepts

Before we start describing the design and implementation details, we firstly clarify some terminologies and concepts that would be used in future.

#### 3.1.1 Cell

A cell is a sub-cube with aggregated dataset and its dimensions are usually in internal levels. The underlying datas are the ones of the intersections of every leaf nodes in each dimension. For example, if the user request one query with X1, X2 in the first dimension and Y1, Y2 in the second dimension, then we get 4 cells which are cell{X1, Y1}, cell{X2, Y1}, cell{X1, Y2} and cell{X2, Y2} respectively. See figure 3.2.

			Dimension 2				
			Y1		Y2		
			Y1.1	Y1.2	Y2.1	Y2.2	Y2.3
Dimension 1	X1	X1.1	(1, 0)	(1, 5)	(1, 10)	(1, 15)	(1, 20)
		X1.2	Cell {X1, Y1}		Cell {X1, Y2}		
		X1.3	(1, 1)	(1, 6)	(1, 11)	(1, 16)	(1, 21)
	X2	X2.1	(1, 2)	(1, 7)	(1, 12)	(1, 17)	(1, 22)
		X2.2	Cell {X2, Y1}		Cell {X2, Y2}		
		X2.2	(10, 3)	(10, 8)	(10, 13)	(10, 18)	(10, 23)
		(10, 4)	(10, 9)	(10, 14)	(10, 19)	(10, 24)	

Figure 3.2: Terminology: Cell

Usually, each cell contains multiple rows of data. And each entry of these rows is the measure

value (the number with black color in each bracket, the red one the *cellOrdinal* assigned with each measure value) with the intersection of the leaf nodes. For instance,  $cell\{X2, Y1\}$  contains the data of  $\{X2.1, Y1.1\}$ ,  $\{X2.2, Y1.1\}$ ,  $\{X2.1, Y1.2\}$  and  $\{X2.2, Y1.2\}$ .

### 3.1.2 Query Cell

A query cell is an aggregated cell with at least one cell inside. Besides, it has to be a regular *shape*. For instance, in 2-dimensional data, a query cell is a rectangle that covers one or more cells. The query cell can be formalized as:

$$\{D_{11}, D_{12}, \dots, D_{1n_1}\} \times \{D_{21}, D_{22}, \dots, D_{2n_2}\} \times \dots \times \{D_{n1}, D_{n2}, \dots, D_{nn_n}\} \quad (3.1)$$

where  $D_{ij}$  is label  $j$  in dimension in  $i$  and any:

$$D_{1m_1} \times D_{2m_2} \times \dots \times D_{nm_n}, 1 \leq m_1 \leq n_1, \dots, 1 \leq m_n \leq n_n \quad (3.2)$$

is a cell. The reason we introduce the query cell concept is that we can use one MDX query to represent one query cell. And one of our goals is to find the minimum number of query cells from these un-cached cells.

### 3.1.3 Answer Sub-cube

The answer sub-cube is the data just needed for serving the user's request. Take the same example in figure 3.2, the answer sub-cube is  $[6, 0]$ ,  $[40, 1]$ ,  $[9, 2]$ ,  $[60, 3]$ . Each value equals to the summarized result of each cell. Generally, the most straightforward way to get what the users want is to retrieve the answer sub-cube from the server side.

### 3.1.4 Inflated Sub-cube

In contrast, the inflated sub-cube is the data with the intersection of all the leaf level nodes in each dimension. In figure 3.2, the inflated sub-cubes are all the datas in every cell. In this thesis, our algorithm is based on downloading the inflated sub-cubes, because we do believe that the users will benefit a lot for the subsequent cross-tab or drill-down operations by utilizing the inflated sub-cubes instead of downloading the answer sub-cube from the server every time. In most scenarios, the network communication is the bottleneck of the system performance. Downloading the inflated sub-cube may cost much more time for the initial query, but it can significantly decrease the number of communications between the client and the server.

## 3.2 Hash Based Caching Architecture

The basic idea of dealing with arbitrary shapes of cached data is to treat each cell as a separate unit. And each cell is assigned a unique id. The id basically is the combination of all the ids in each dimension.

We keep a hash table globally to store the `key-value` pair, where the `key` is the id of each cell and the `value` is the inflated sub-cube data for such a cell. See figure 3.3 as the hash table build for figure 3.2.

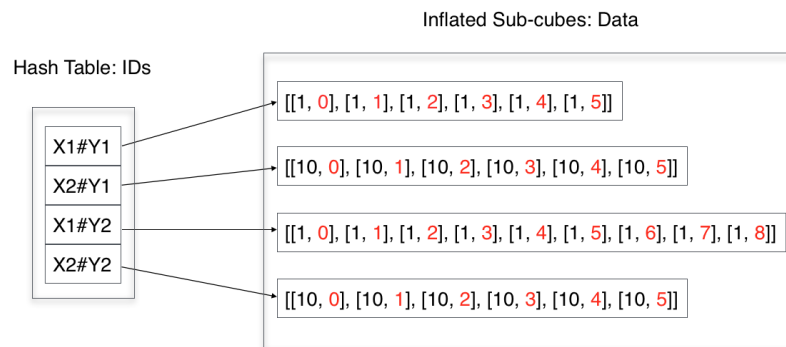


Figure 3.3: Hash Table of the Caching Mechanism

After building such a hash table, we can easily identify that whether a cell is cached or not by searching in the hash table which costs only  $O(1)$  time in theory. And more importantly, our hash table can hold any *shape* of cached dataset, since each cell is a basic unit rather than the whole cube.

After excluding the cached cells, the next step is to aggregate the un-cached cells to minimal number of query cells as each query cell maps to one MDX query. If we can reduce the number of query cells as much as possible, we can not only reduce the number of MDX queries sent to the server, but also improve the performance of the server side by reducing the cache misses, since the server will return faster if we request a chunk of dataset using one MDX query instead of multiple sub-queries each time. After the cells are aggregated to larger query cells, we generate the queries for each of them respectively and then send these MDX queries to the server side. The server will execute these queries and return the results back. After all the data is downloaded, our system will split the aggregated datasets so that each un-cached cell can map to its own portion of data and then be inserted into the hash table.

Figure 3.4 demonstrates the procedure of how we aggregate the cells, send the corresponding queries to the server and split the downloaded data to each cell after the cell excluding phase.

Now, we will talk about implementation details of each phase.

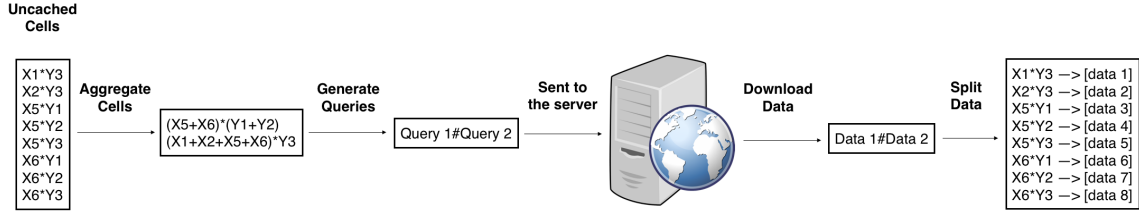


Figure 3.4: The Procedure of Our Caching Mechanism

### 3.3 Aggregate Cells

#### 3.3.1 Problem Formalization

The goal to aggregate the un-cached cells is to produce minimal number of query cells. We can formalize the problem as follows:

Given a set of formulas with the form of:

$$D_{1n_1} \times D_{2n_2} \times \dots \times D_{nn_n} \tag{3.3}$$

and two of them can be merged to a new one if and only if at least  $n - 1$  factors with the same index are the same. For example, if we have a formula:

$$D_{1n_1} \times D_{2n_2} \times \dots \times D_{im_i} \times \dots \times D_{nn_n}, \text{ where } i \leq n \tag{3.4}$$

Then, it can be merged with formula 3.3 to:

$$D_{1n_1} \times D_{2n_2} \times \dots \times (D_{in_i} + D_{im_i}) \times \dots \times D_{nn_n}, \text{ where } i \leq n \tag{3.5}$$

And the problem is to find the optimal combinations of any formulas that can be merged together so that the final number of formulas is minimal.

#### 3.3.2 Sort-Merge Algorithm

However, there's no known algorithm runs in polynomial time that can generate the optimal solution. We proposed an approximate method called the Sort-Merge algorithm to aggregate the un-cached cells. We first sort the cells according to the ids. Then, we check whether two continuous cells can be merged together from the beginning to the end. The two cells can only be merged if only one of their ids differ in the same dimension. We apply the checking recursively until no two cells can be merged. The goal is to aggregate these cells to generate minimal number of query cells. Each query cell can be represented as a MDX query. By reducing the number of MDX queries sent to the server, we can not only recude the networking communication cost, but also reduce the server execution time, which is illustrated in details in section 3.6.

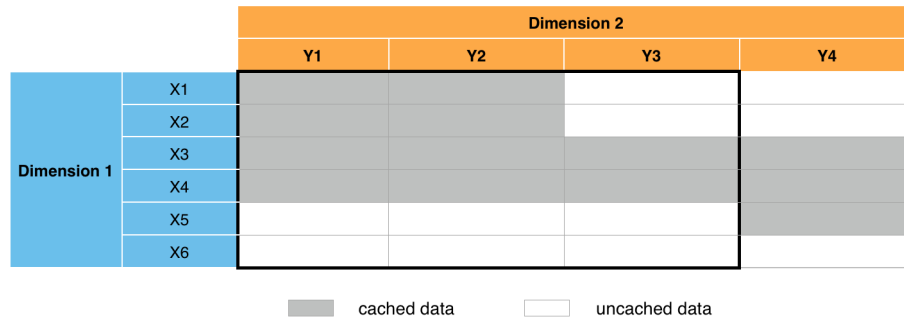


Figure 3.5: Caching Mechanism Example

Figure 3.5 is an example that shows how the Sort-Merge algorithm works.

Assume that we have some cached data cells and these cells are stored in a hash table individually. Now, the user is going to issue a request for the results in the box with black boarder. Our algorithm firstly check all the cells to see whether they are cached or not. After the checking phase, our system finds out that only 8 cells are not cached. Then, our system sort these cells according to their ids. At last, a recursive merge will be performed for every two continuous cells. For instance,  $X1*Y3$  and  $X2*Y3$  can be merged to  $(X1+X2)*Y3$ . The procedure goes as figure 3.6:

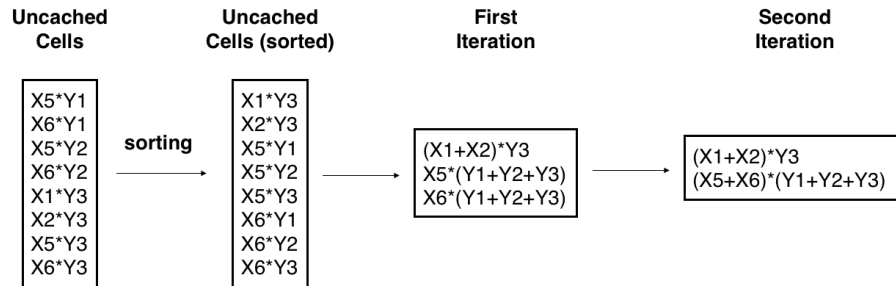


Figure 3.6: Sort-Merge Algorithm

Finally, all the un-cached cells are aggregated to two query cells, which are represented as:  $X1, X2\#Y3$  and  $X5, X6\#Y1, Y2, Y3$ . And each query cell can be represented as a single MDX query so that we can retrieve the whole chunk of data at the same time.

Algorithm 1 is the pseudo-code for this sort and merge algorithm.

**Algorithm 1** SortMerge(cells)

---

```

1: Initialize FLAG = 0
2: for i=0; i ≤ cells.length-1; ++i do
3:   index = ValidateCells(cells[i], cells[i+1])
4:   if index ≠ -1 then
5:     CombineCells(cells[i], cells[i+1])
6:     FLAG = 1
7:   end if
8: end for
9: if FLAG == 1 then
10:  return SortMerge(cells)
11: end if

```

---

The `ValidateCells` function is a helper function that determines whether two cells are combinable. It returns the dimension index (should be only one) that the ids of the two cells are different, otherwise returns -1. While the `CombineCells` function just simply combines these two cells.

**Analysis**

Now, we will analyze time complexity of the `Sort-Merge` algorithm. Each time, we check all the cells to see if two of them can be merged together. In each iteration, the time complexity is  $O(nd)$ , where  $n$  is the number of cells and  $d$  is the number of dimensions. As for every two cells, we'll check their ids in every dimension. Then, the condition to perform the next iteration is that we find at least two cells are combinable. So, the time complexity for the next iteration is at most  $O((n-1)d)$ . At the last iteration, we'll at least check two compressed cells in  $O(2d)$ . So, in worst case, during each iteration, we only merge two cells, the time complexity is:

$$T(n) = O(nd) + O((n-1)d) + O((n-2)d) + \dots + O(2d) = O\left(\frac{(n+2)(n-1)}{2} \times d\right) = O(n^2d) \quad (3.6)$$

Meanwhile, essentially, the `Sort-Merge` algorithm is a greedy algorithm with local optimization. The `Sort` phase sorts the cells according to their ids and indeed gives the former dimensions the higher priorities. Thus, if the optimal query cells should be based on the later dimensions, the `Sort-Merge` algorithm is not guaranteed to find the optimal solution. Figure 3.7(a) and figure 3.7(b) show one scenario that the `Sort-Merge` algorithm fail to find the optimal solution:

The final solution by applying the `Sort-Merge` algorithm is 5 query cells. While the optimal solution is:  $(X1+X5+X6+X7+X8)*Y1$ ,  $(X1+X7+X8)*Y2$  and  $(X1+X2+X3+X7+X8)*Y3$ , which has only 3 query cells.

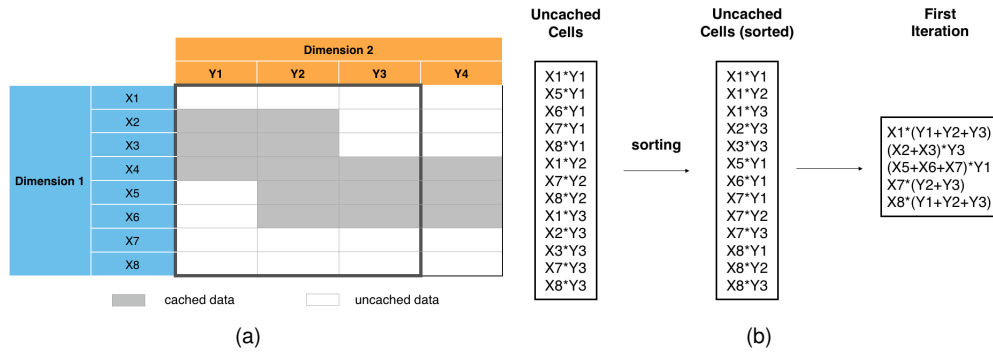


Figure 3.7: Example for Sort-Merge algorithm fails to find the optimal solution

There's no known algorithm runs in polynomial time that can generate the optimal solution. The reason we adopt the Sort-Merge algorithm is a trade-off between the algorithm execution time, data download time and the data split time. Section 3.6 conducts some comparisons among different strategies to support this conclusion.

### 3.4 Generate Queries

After aggregating the un-cached cells, the next step is to generate the queries for each of these query cells. Algorithm 2 shows the detailed steps of generating a query from a query cell.

---

**Algorithm 2** GenerateQuery(queryCell)

---

```

1: Initialize hierarchies = []
2: subCells = queryCell.split("#")
3: for i=0; i < subCells.length; ++i do
4:   initialize members = []
5:   subSubCells = subCells[i].split(",")
6:   for i=j; j < subSubCells.length; ++j do
7:     var node = tree.getNodeByIndex(subSubCells[j])
8:     members.push(node.text)
9:   end for
10:  hierarchies.push(members.join("#"))
11:  return GenerateQueryFromHierarchis(hierarchies);
12: end for

```

---

This algorithm checks the merged cells in every dimension and generate the relative hierarchies and them merged them together. So, it's like the reversed order of the Sort-Merge function.

## 3.5 Data Split

After we receive the data downloaded from the server, we'll split the data so that each cell points to its own portion of data. Note that the data downloaded from the server is an one dimensional array with each entry is the combination of a measure value and a *cellOrdinal*. If the user's query contains multiple cells, we need to extract the part of data for each of them so that for an individual cell, it only points to its own portion. The basic idea is to decompose the aggregated cells to the original atomic cells first and then for each atomic cell, we scan the whole part of the aggregated data and extract its own portion.

Algorithm 3 shows the procesure of decomposing the aggregated query cells. It extracts the portion of data from each aggregated dataset and insert the `key-value` pair into the global hash map, which is also the global cache, for furture use.

---

### Algorithm 3 BuildHashMap(compressedCells)

---

```

1: for var i = 0; i < compressedCells.length; ++i do
2:   initialize result = [];
3:   var atomicCells = generateSubCells(compressedCells[i]);
4:   var children = generateChildren(compressedCells[i]);
5:   for var j = 0; j < atomicCells.length; ++j do
6:     splitMeasures(atomicCells[j], children, cube[i].celMeasure);
7:   end for
8: end for

```

---

The `generateSubCells` function is a helper function that split the compressed cell into several atomic cells. For instance,  $X_1, X_2\#Y_1, Y_2$  will be split into  $X_1\#Y_1, X_1\#Y_2, X_2\#Y_1$  and  $X_2\#Y_2$ . And the `generateChildren` function will generate all the children nodes at the leaf level in each dimension. After collecting the data we need, next, we'll extract the data for each of the individual cell, which the `splitMeasures` function will do.

### 3.5.1 CellOrdinal Rewrite

Before we describe the `splitMeasures` function, we realized that it's not only the data we need to split, but also we need to take care of the *CellOrdinal* values. As we can see, in the original aggregated dataset, the *CellOrdinals* associated with the measure values is continuous in global. And for a single cell, it may contain multiple measure values while the *CellOrdinals* of these values is not continuous. For example, we selected  $X_1, X_2$  and  $X_3$  as the first dimension and  $Y_1, Y_2$  as the second dimension. Figure 3.8 shows the aggregated result for each cell.

While in the underlying or leaf level, the measures are scattered as follows, see figure 3.9.



		Dimension 2	
		Y1	Y2
Dimension 1	X1	6	9
	X2	40	60
	X3	200	300

Figure 3.8: Aggregated Result for Each Cell

			Dimension 2				
			Y1		Y2		
			Y1.1	Y1.2	Y2.1	Y2.2	Y2.3
Dimension 1	X1	X1.1	(1, 0)	(1, 6)	(1, 12)	(1, 18)	(1, 24)
		X1.2	(1, 1)	(1, 7)	(1, 13)	(1, 19)	(1, 25)
		X1.3	(1, 2)	(1, 8)	(1, 14)	(1, 20)	(1, 26)
	X2	X2.1	(10, 3)	(10, 9)	(10, 15)	(10, 21)	(10, 27)
		X2.2	(10, 4)	(10, 10)	(10, 16)	(10, 22)	(10, 28)
	X3	X3.1	(100, 5)	(100, 11)	(100, 17)	(100, 23)	(100, 29)

end

Figure 3.9: Underlying Level Measures and the Corresponding *CellOrdinal* Values

In the example, the *CellOrdinal* for each measure in cell X1#Y1 is not continuous. After we extract the portion of measures from the aggregated measures for this cell, we need to rewrite the *CellOrdinal* values so that they're continuous. We achieve this goal by recording a *index* value in the *SplitMeasures* function. It self increases whenever we find a measure for our current cell or the related measure is empty. So after *CellOrdinal Rewrite*, the measures and their corresponding *CellOrdinal* value would look like below, see figure 3.5.1:

		Dimension 2	
		Y1	
		Y1.1	Y1.2
Dimension 1	X1	X1.1	(1, 0)    (1, 6)→(1, 3)
		X1.2	(1, 1)    (1, 7)→(1, 4)
		X1.3	(1, 2)    (1, 8)→(1, 5)

Figure 3.10: Measures and the Corresponding *CellOrdinal* Value After *CellOrdinal Rewrite* Phase

Algorithm 4 shows the details for splitting the measures values for a cell.

---

**Algorithm 4** SplitMeasuresRec(subCell, children, aggregatedMeasures, measures, dimensionLength, current, start, cubeIndex, index)

---

```

1: if start < 0 then
2:   // reach the end of the dimensions
3:   var cellOrdinal = calculateCellOrdinal(dimensionLength, current);
4:   for var i = cubeIndex; i > -1; -i do
5:     if aggregatedMeasures[i][1] == cellOrdinal then
6:       cubeIndex = i-1;
7:       var measure = aggregatedMeasures[i].slice(0);
8:       measure[1] = index;
9:       measures.push(measure);
10:      index++;
11:      break;
12:     end if
13:     if aggregatedMeasures[i][1] > cellOrdinal then
14:       cubeIndex = i;
15:       index++;
16:       break;
17:     end if
18:   end for
19: else
20:   for var i = 0; i < children[start].length; ++i do
21:     var axisNode = tree.getNodeByIndex(subCells[start]);
22:     var memberNode = children[start][i];
23:     if Query.isAncestor(axisNode, memberNode) then
24:       current.unshift(i);
25:       splitMeasuresRec(subKeys, children, aggregatedMeasures, measures, dim, dimension-
           Length, start-1, cubeIndex, index);
26:       current.shift();
27:     end if
28:   end for
29: end if

```

---

The depth first search is the core idea of this algorithm and it can handle arbitrary dimension data. And this algorithm not only splits the aggregated data cube, but also includes the *CellOrdinal Rewrite* phase.

Each time, the function checks the combinations of all the children in each dimension and also

scan the whole aggregated dataset. If the number of the cells equals to 1, the size of the dataset, or more precisely, the number of rows in the dataset is always less or equal than the number of the children, since it's a 1-to-1 mapping from every child to every row in the dataset and some rows may not exist in the dataset. Meanwhile if the dataset contains multiple cells, then the number of rows in the dataset usually is greater than the number of children of each individual cell. Let's assume that the number of the children for cell  $i$  is  $c_i$  and the number of rows in the dataset is  $r$ . And also, we assume number of cells is  $n$ . Then, the total time complexity of the `SplitMeasuresRec` function is:

$$T(n) = O\left(\sum_{i=1}^n \max(c_i, r)\right) \quad (3.7)$$

### 3.5.2 Evaluation

Now, we will evaluate the performance of our `SplitMeasuresRec` function. Table 3.1 demonstrates the test environment and the client-server specifications we have used for evaluation throughout all the work in this thesis.

	Test Machine Specification
Front-End Machine	Core i7 @2.6GHz 16GB RAM Mac OSX 64-bit
Back-end Server	quad-core Intel Xeon E5420 4 GB RAM Windows Server 2008 R2 64-bit
OLAP Server	Microsoft Analysis Services 2008 R2
Web Service	ASP.Net
Browser Platform	Safari 7.0.5
Dataset	Adventure Works DW

Table 3.1: Evaluation Environment

As we stated in section 3.5, the data split performance is related to the number of children and the number of the cells it contains. So our experiments will test on different number of rows of the dataset and different number of the children for several different dimensions. Table 3.2 shows test result. And this result supports our analysis on the time complexity for the data split. Also, we can see from this table that even when the data size is huge, our split time is still reasonable and acceptable.

# Cells	# Children	# Rows	Size(KB)	Split Time (ms)
2-dimensional				
1	1400	880	10.7	4
1	9125	8468	114.1	9
1	47000	43616	627	36
2	3080	1936	24.1	6
2	11000	10208	138.8	12
2	56125	52084	750.2	59
4	9379	4746	60.2	11
4	18080	15594	216.8	20
4	71840	61962	890.7	74
8	18758	9492	125.3	26
8	46646	23604	319.9	63
8	89920	77556	1115.1	138
3-dimensional				
1	7000	4400	53.0	8
1	25000	22840	304.3	19
1	45625	41683	563.6	31
1	235000	214696	3068.3	159
2	15400	9680	120.3	18
2	78575	49390	644.2	73
2	280625	256379	3684.4	292
4	46895	23730	302.8	39
4	90400	76953	1042.8	84
4	359200	305769	4396.1	352
8	93790	47460	611.7	112
8	233230	118020	1595.6	280
8	449600	382722	5522.3	623
4-dimensional				
1	42000	26400	340.7	27
1	150000	137040	1923.7	113
1	273750	250098	3590.5	193
1	1410000	1288176	19291.2	1119
2	92400	58080	757.6	61
2	471450	296340	4091.9	447
2	1683750	1538274	23258.9	1872
4	281370	142380	1929.0	230
4	542400	461718	6678.3	550
4	2155200	1834614	27802.0	2316
8	562740	284760	3935.1	801
8	1399380	708120	10068.3	2098
8	2697600	2296332	35035.7	4328

Table 3.2: Split Time with different Cells &amp; Dimensions

### 3.6 Why Sort-Merge algorithm?

Now, we will compare the time among different strategies and see that why we choose the Sort-Merge algorithm. Let's consider the other two extremes that go for different directions: one algorithm is a brute-force algorithm that runs in exponential time but it can come up with the optimal solution, while the other one we do not even try to aggregate the cells to generate the query cells at all.

#### 3.6.1 Brute-force Search

Firstly, let's consider the brute-force search algorithm. This algorithm simply searches the combinations of every two cells. We can formalize the problem as a search problem in a graph, in which the start vertex corresponds to the original cells (or the set of formulas discussed in section 3.3.1) and from each vertex  $v$  there emanates arcs to its child vertices, which each corresponds to the result of performing one merge operations on  $v$ .  $v$  will have a child vertex for each possible merge operations that could be performed on it; if there are  $n$  cells in  $v$ , then this means it could have up to  $n(n-1)/2$  children in the worst case, because it could be that all  $n$  cells (or formulas) share a full complement of  $d-1$  factors, meaning that any pair of them could be combined.

If a vertex has no pair of cells that can be combined then it is a *leaf* – it has no children, and can not be processed any further. What we want to find is a leaf vertex that has the fewest number of query cells (or aggregated formulas). Since every merge operation, corresponding to an arc in the graph, reduces the number of cells (or formulas) by 1, this is equivalent to searching for a deepest-possible vertex. This can be done using DFS or BFS.

Note however that the same vertex (or formula) can be generated multiple times over using this approach, so it is beneficial that we do an optimization to reduce the the search domain. We can maintain a hash table seen that records all expressions that have already been processed; then if we visit a vertex (or formula), try to generate a child for it, and see that this child is already in seen, we avoid visiting this child a second time.

Figure 3.11 is the example we use to show how the brute-force search algorithm works.

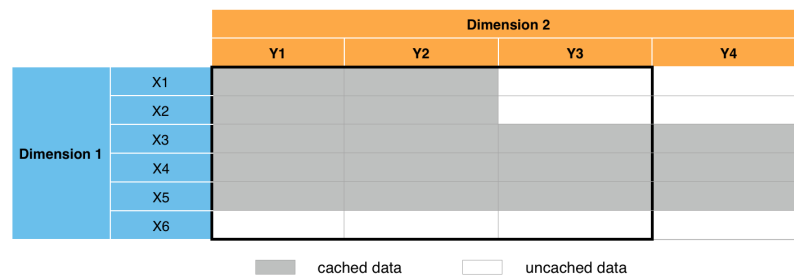


Figure 3.11: Example used to compare three different strategies

Figure 3.12 illustrates the search procedures with the cutting off additional search domain by using the hash table based on figure 3.11.

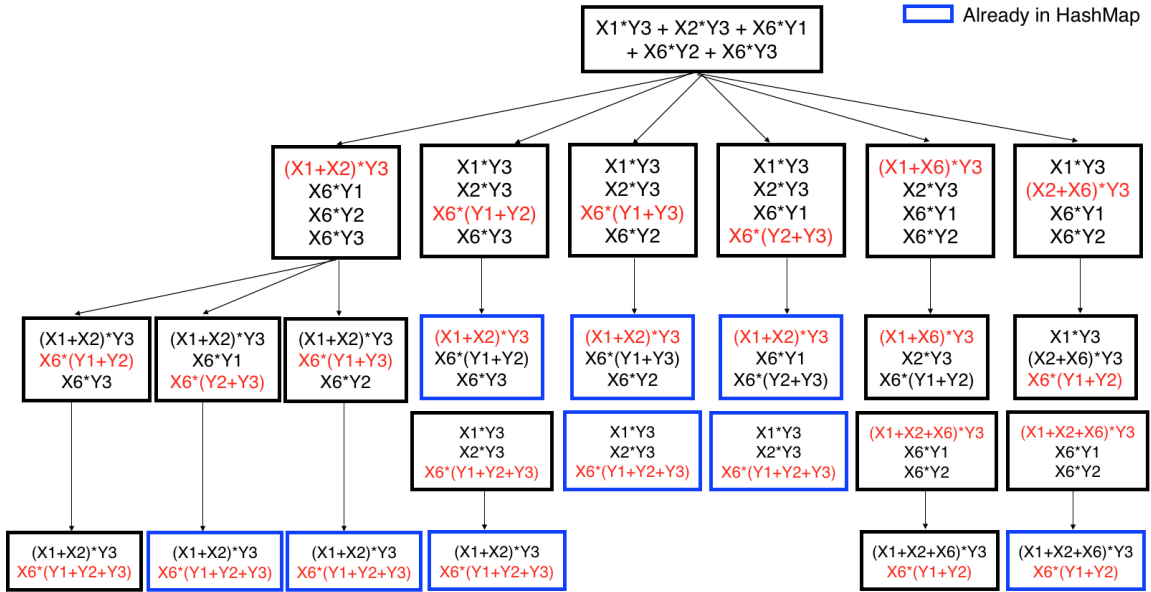


Figure 3.12: Brute-Force Algorithm: Exponential Time

Every time, we check the combinations of every two vertex starting from the beginning to the end. And algorithm 5 is the pseudo-code for this brute-force algorithm.

**Algorithm 5** BruteForceSearch(originalCells)

---

```

1: Initialize var queue = new Queue();
2: Initialize var map = new HashMap();
3: Initialize var cells = [];
4: queue.enqueue(originalCells);
5: var size = queue.size;
6: while queue not empty do
7:   for i = 0; i ≤ size; ++i do
8:     keys = queue.dequeue();
9:     children = MergeCells(cells);
10:    for j = 0; j ≤ children.length; ++j do
11:      if (!map.contains(children[j])) then
12:        map.put(children[j], true);
13:        queue.enqueue(children[j]);
14:      end if
15:    end for
16:  end for
17: end while

```

---

The `mergeCells` function checks the combinations of every two cells and return an array of the children. Algorithm 6 presents the details of this method.

**Algorithm 6** MergeCells(cells)

---

```

1: Initialize var solutions = [];
2: for i = 0; i ≤ cells.length-1; ++i do
3:   for j = i+1; j ≤ cells.length; ++j do
4:     if cells[i] and cells[j] are combinable then
5:       Initialize var solution = cells;
6:       Merge cells[i] and cells[j] in solution;
7:       solutions.push(solution);
8:     end if
9:   end for
10: end for
11: return solutions;

```

---

As we stated above, each vertex in the search graph could have up to  $n(n-1)/2$  children, so the function calculating its time complexity would be:

$$T(n) = \frac{n(n-1)}{2} + T(n-1) \quad (3.8)$$

By applying the Master Theorem, we know that:

$$T(n) = n^n \quad (3.9)$$

which means this brute-force search algorithm runs in exponential time.

### 3.6.2 No Aggregations

We also consider the algorithm that does not even aggregate any cells at all. So, each cell would be a query cell and the processing time would be the least. Besides, another benefit of applying the no aggregation method is that it does not need to split data downloaded from the server, because for each of all the query cells contains only one cell as well as the corresponding data. However, we have to create a query for every cell. And this will increase the number and size of the queries sent to the server. Meanwhile, the server will have to search the result for each of these queries individually. This is an interesting comparison with the search for chunks of data when we aggregate some cells together as it might cause more cache misses and bring up more networking communication.

### 3.6.3 Performance Evaluation

And figure 3.13 shows the pre-process time among these three different strategies.

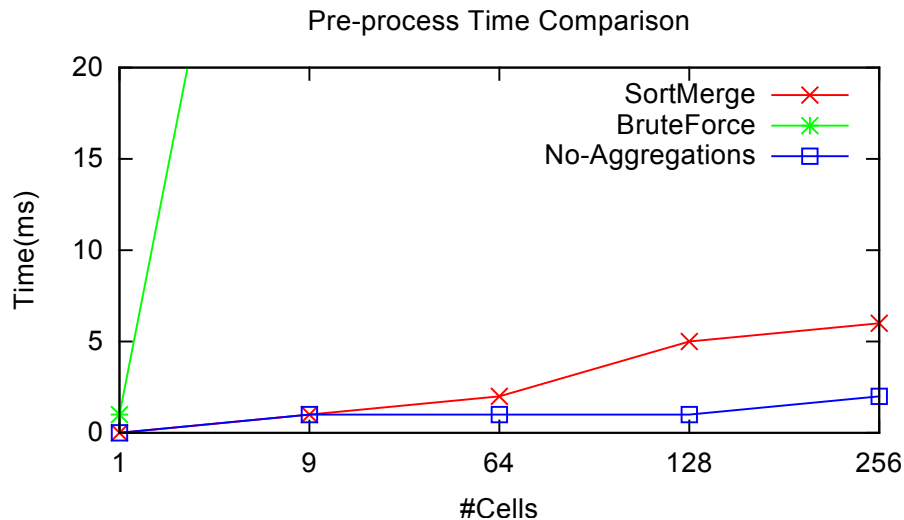


Figure 3.13: Pre-process Time Comparison



We can easily see from the figure that both the `Sort-Merge` and the `No-aggregations` algorithm run fast and there is not big difference between them. However, the brute-force search algorithm which runs in exponential time is really slow when the number of cells goes up. Indeed, it is too slow to be eligible to be considered as a practical approach even when the number of cells is not big. Therefore, we will leave out the brute-force search algorithm for future comparisons.

Figure 3.14 shows the data download time + data split time comparison between the `Sort-Merge` algorithm and the `No-aggregations` algorithm.

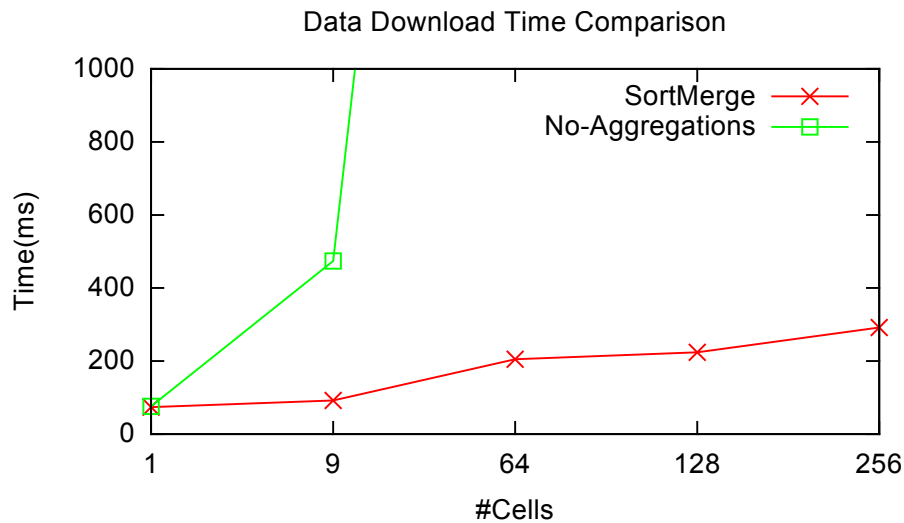


Figure 3.14: Data Download Time Comparison

As we can see, as the number of cells increases, the no-aggregations algorithm will take much more time to download the data from the server side. And even when the number of the un-cached cells is not large, the cost time of the no-aggregations algorithm is also unacceptable. This is because it needs to send much more queries to the remote server if we do not do the cell aggregation. Besides, the server needs to spend more time to process each query individually than processing chunks of data together.

### 3.6.4 Effectiveness

Besides, in reality, in most scenarios, the `Sort-Merge` algorithm can produce the optimal solutions. Table 3.3 shows some test cases with different number of cells with each includes different number of un-cached cells inside in random positions for 2-dimensional data. For instance, for case 7, we have the  $4 \times 4$  rectangle which contains 16 cells in total. Then, we choose 3 of the cells in random positions as the cached cells to mock the cached data of the user's previous queries. Next, we'll run

both our *Sort-Merge* algorithm and the brute-force search algorithm to generate the query cells. The *#Query Cells* column shows the result our *Sort-Merge* algorithm produces and the *#Optimal Query Cell* shows the result our brute-force search algorithm produces, which is also the optimal solution. In this case, both of them output the optimal query cells, which are 4 query cells totally.

Case	#Cells(#Row×#Col)	#Un-cached Cells	#Query Cells	#Optimal Query Cell	Optimal?
1	1×1	1	1	1	Yes
2	1×2	1	1	1	Yes
3	2×2	2	1	1	Yes
4	2×4	4	2	2	Yes
5	2×4	6	2	2	Yes
6	4×4	5	4	3	<b>No</b>
7	4×4	13	4	4	Yes
8	4×8	10	4	4	Yes
9	4×8	20	4	4	Yes
10	4×8	28	4	4	Yes

Table 3.3: Sort-Merge Effectiveness for 2-dimensional Data

Table 3.4 presents some test cases with 3-dimensional data. Again, all the cached cells are randomly picked.

#Cells(#X×#Y×#Z)	#Un-cached Cells	#Query Cells	#Optimal Query Cell	Optimal?
1×1×1	1	1	1	Yes
1×1×2	1	1	1	Yes
1×2×2	2	1	1	Yes
2×2×2	4	3	2	<b>No</b>
2×2×2	6	4	3	<b>No</b>
2×2×4	5	3	3	Yes
2×2×4	8	4	4	Yes
2×2×4	13	4	4	Yes
2×4×4	10	5	5	Yes
2×4×4	20	7	7	Yes
2×4×4	28	7	7	Yes

Table 3.4: Sort-Merge Effectiveness for 3-dimensional Data

We can see that, in most cases, for both the 2-dimensional or 3-dimensional data, our *Sort-Merge* algorithm outputs the optimal solutions. And this shows that the algorithm works effectively in reality.

In conclusion, the Sort-Merge algorithm is a trade-off between the pre-process time, data download time and data split time. In practical, it works fast and always comes up with optimal solutions in most circumstances.

### 3.7 Cache vs. No-Cache Comparison

Now, we will compare our hash based mechanism to the one without using the cache to show that our caching system can reduce the size and the time for the download data from the server. And the performance of our caching mechanism really depends on the percentage of the cached data in the newly issued query. We compare the five scenarios with different percentage of cached cells. Figure 3.15(a) to figure 3.15(e) illustrate the examples we use:

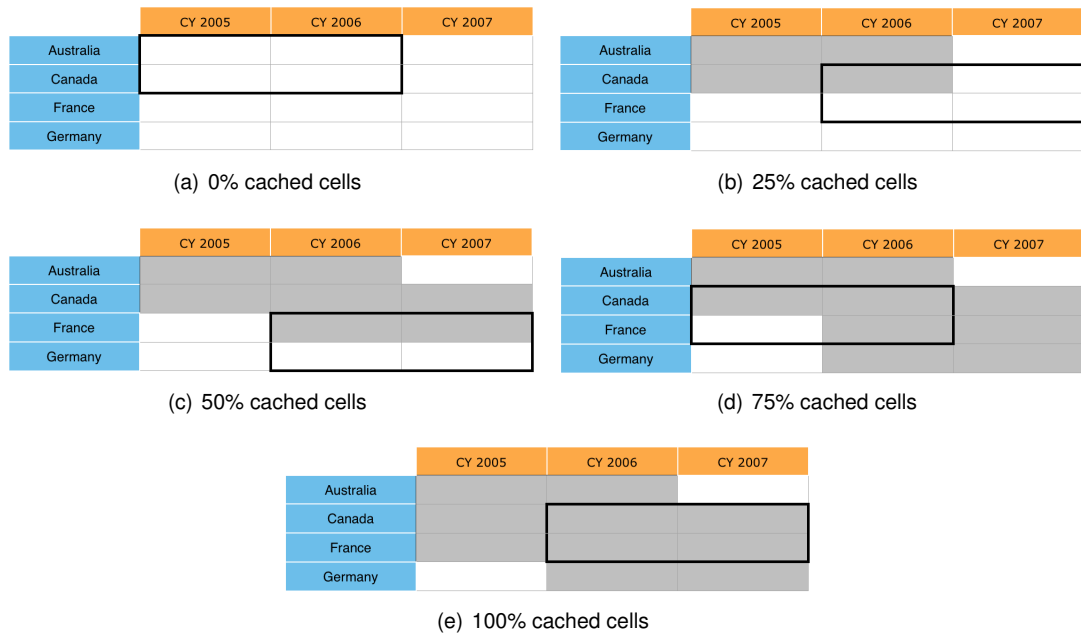


Figure 3.15: Example for different cached percentage

And figure 3.16 shows the data size to be downloaded comparison between the cached and non-cached mechanisms.

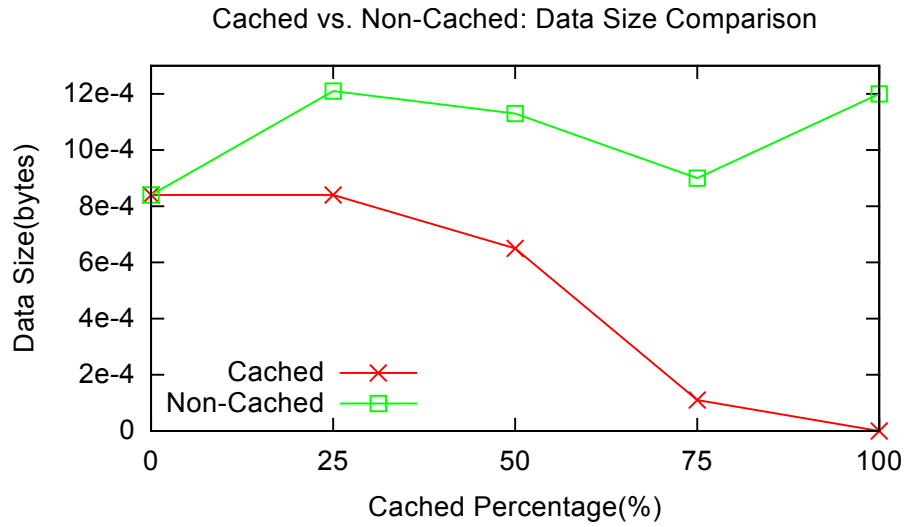


Figure 3.16: Data Download Size Comparison

Figure 3.17 shows the performance between the cached and non-cached ones.

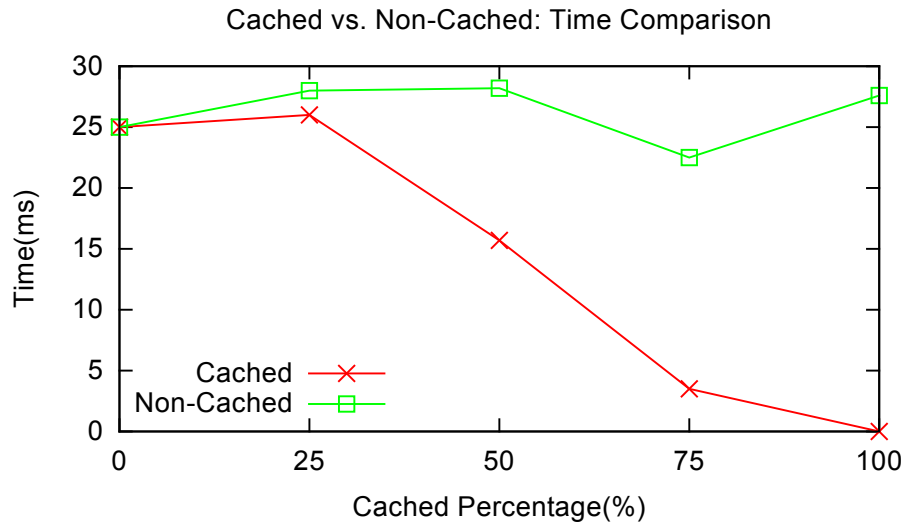


Figure 3.17: Time Comparison

It's obvious that by applying our hash based mechanism, we download much less data from the server side and gain a huge performance improvement.

## Chapter 4

# Query Processing

We discussed the architecture of the hash based caching mechanism in chapter 3. Now, we will talk about how the user's queries are processed based on it and how we utilize the hash based caching data structure to serve the user's subsequent drill-down or roll-up requests to reduce the network communication workflow.

### 4.1 Query Categorization

Given a query submitted by the user, we can identify it to one of the 3 different categories:

1. the query result has no overlaps with any cells in the cache
2. the query result is a subset of the cells in the cache
3. the query result has some overlaps with some cells in the cache

And depending on different categorizes, we have different ways to process them. For case 1, we simply submit a new query to the server, download the result and merge the corresponding cells into our cache. For case 2, as the request cube data is contained in our hash table, we call it as subset query and the corresponding processing way is discussed in section 4.2. And case 3 is discussed in section 4.3.

### 4.2 Subset Query

As the subset query result is contained in our hash table, the key point is to identify which cell it belongs to. Or equivalently, we need to find out which cell (the so called parent cell) in the cache that includes the data for the subset query. For instance, if the cache includes cells `Canada#2005`

and France#2005. If the user requests Burnaby#2005 (subset query). We need to identify that Canada#2005 is the parent cell which contains all the data for Burnaby#2005.

### 4.2.1 Identify the Parent Cell

Let's firstly formalize this problem as follows. Assume the number of dimensions is  $n$  and the depth of the current sub cell in dimension  $i$  is  $d_i$ . Now, we should find out a value  $S_i$  ( $0 \leq S_i \leq d_i$ ) for each dimension so that we can find a cached parent cell by moving up  $S_i$  steps for dimension  $i$ . Apparently, in worst case, the time complexity is:

$$T(n) = d_1 \times d_2 \times \dots \times d_n \quad (4.1)$$

for each specific cell. The basic idea is to check the combinations of moving up  $S_i$  steps for every dimension. We can use DFS to generate all the combinations recursively. Algorithm 7 shows the procedure of finding the cached parent cell.

---

#### Algorithm 7 FindParentCell(dimensions, start)

---

```

1: Initialize var dimension = dimensions[start];
2: while dimensions[start] != null do
3:   if start == dimensions.length - 1 then
4:     // reach the last dimension
5:     key = GenerateKey(dimensions);
6:     if CachedKeys contains key then
7:       return key
8:     end if
9:   else if start < dimensions.length - 1 then
10:    key = FindParentCell(dimensions, start+1);
11:    if key != null then
12:      return key;
13:    end if
14:  end if
15:  dimensions[start] = dimensions[start].parent;
16: end while
17: dimensions[start] = dimension;
18: return null;

```

---

This may be very time consuming when the number of dimensions and the value of  $d_i$  is high. However, fortunately, in most circumstances, the users only request 2 or 3 dimensional data. And

meanwhile, the depth in each dimension is usually not high due to the natural properties of the hierarchies. So, this algorithm works fast in reality.

### **4.2.2 Extract the Subset Data**

After we locate the parent cell, the next step is to extract the portion of data from the parent cell, because the parent cell contains other parts of data as well. This can be achieved by scanning the inflated sub-cube of the parent cell and whenever we find a row (or node) that belongs to our subset query, its value will be added. The time complexity is  $O(n)$  where  $n$  is the number of rows in the individual parent cell. Algorithm 8 shows the procedure of extracting the data for the requested subset query.

---

**Algorithm 8** ExtractData(parentCell, dimensions, rowAxisNode, rowLevelToFetch, columnAxisNode, columnLevelToFetch, answerInGetMatrix)

---

```

1: var index = 0;
2: for var j = 0; j < parentCell.children[1].length; ++j do
3:   if this.isAncestorCol(dimensions, children[1][j]) then
4:     for var i = 0; i < parentCell.children[0].length; ++i do
5:       if this.isAncestorRow(parentCell.dimensions, parentCell.children[0][i]) then
6:         var current = [i, j];
7:         var dim = [parentCell.children[0].length, parentCell.children[1].length];
8:         var cellOrdinal = this.calculateCellOrdinal(dim, current);
9:         for var k = index; k < parentCell.measures.length; ++k do
10:          if parentCell.measures[k][1] == cellOrdinal then
11:            index = k + 1;
12:            break;
13:          end if
14:          if parentCell.measures[k][1] > cellOrdinal then
15:            index = k;
16:            k = -1;
17:            break;
18:          end if
19:        end for
20:        assign var row = the specified row we drilled-down to
21:        assign var col = the specified column we drilled-down to
22:        if k > -1 and k < parentCell.measures.length then
23:          answerInGetMatrix[row][col] += parentCell.measures[k][0];
24:        end if
25:      end if
26:    end for
27:  end if
28: end for

```

---

### 4.3 Overlap Query

For the overlap query, we know that some cells in the same level in our cache are included in this query. So, the first job is to exclude the overlap parts, or the cached cells. Then, we aggregate the un-cached cells to some query cells and generate the queries for them. Next, we send these queries to the server side and split the data downloaded from the server. Finally, we extract the



specific portion of data for each cell and insert them to our hash map cache for further use. The detailed procedures are discussed in chapter 3 already.

### 4.4 Drill-down Query

The drill-down query is a type of query issued from the table UI in the Web-OLAP system. Users can drill-down a specific row or column, see figure 4.1 as the example to drill-down the Canada row.

		All Products								
		+ Accessories	+ Bikes	+ Clothing	+ Components	+ Accessories	+ Bikes	+ Clothing	+ Components	
All Customers	Australia+	138571.00	0.00	0.00	0.00	Alberta+	414.00	0.00	0.00	0.00
	Canada+	103370.00	0.00	0.00	0.00	British Columbia+	102919.00	0.00	0.00	0.00
	France+	63315.00	0.00	0.00	0.00	Brunswick+	0.00	0.00	0.00	0.00
	Germany+	62133.00	0.00	0.00	0.00	Manitoba+	0.00	0.00	0.00	0.00
	United Kingdom+	76534.00	0.00	0.00	0.00	Ontario+	37.00	0.00	0.00	0.00
	United States+	256234.00	0.00	0.00	0.00	Quebec+	0.00	0.00	0.00	0.00

Figure 4.1: Drill Down: Row

Also they can drill-down a cell, see figure 4.2(a) and figure 4.2(b) to drill-down the Canada#Bikes cell.

		All Products							
		+ Accessories	+ Bikes	+ Clothing	+ Components	+ Accessories	+ Bikes	+ Clothing	+ Components
All Customers	Australia+	138691	8852050						
	Canada+	103378	1821302						
	France+	63407	2533576						
	Germany+	62233	2808514						
	United Kingdom+	76630	3282843						
	United States+	256422	8999860						

(a)

		All Products							
		+ Accessories	+ Bikes	+ Clothing	+ Components	+ Accessories	+ Bikes	+ Clothing	+ Components
All Customers	Australia+	138691					8852050		
	Canada+	103378							
	France+	63407						2533576	
	Germany+	62233						2808514	
	United Kingdom+	76630						3282843	
	United States+	256422						8999860	

(b)

Figure 4.2: Drill Down: Cell

This drill-down request indeed is a special case of the subset query discussed in section 4.2 with two minor differences:

1. Each time, the drill-down is to drill 1 level down.
2. The result set of the drill-down query usually covers multiple cells. For example, if the user drill-down the Canada row in figure 4.2(a), it touches both the Canada#Accessories and Canada#Bikes cells.

Though there are 3 different types of drill-downs (row, column, cell), all of them can be treated as expanding a *slice* of the table. Like the drill-down on row is expanding the cells in the specified row.

In Elahe’s thesis [9], there is only one global huge cell and all the measures are gather together and stored in a one dimensional array. When drill-down request is invoked, it needs to scan the whole aggregated dataset to find out what portions of data belong to the subset. While in our hash based caching mechanism, as we treat each cell as a separate unit and their measures are extracted from the original global aggregated dataset so that each cell points to its own chunk of data. Therefore, we can easily locate the cells of each slice touched when drill-down is issued. This can significantly save the time especially when the cached data is large and we only interested in drill-down to a small slice of the cube. Figure 4.3 gives one example of how we quickly find the touched cells in the specific row by adopting our approach.

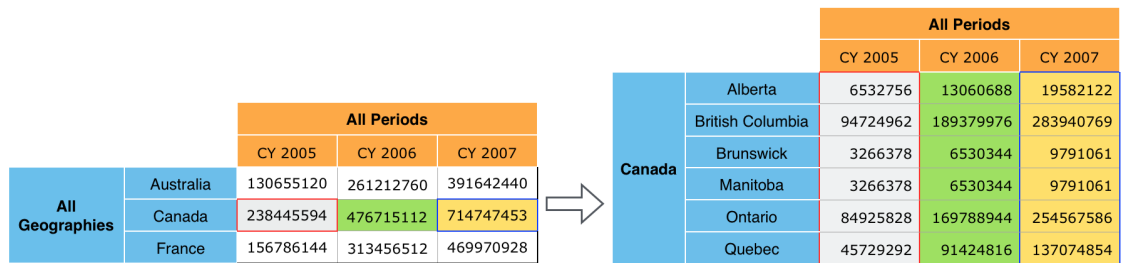


Figure 4.3: Drill Down Example

In this example, we choose to expand the Canada row. We can easily identify that the cells we need to search is the 3 in the Canada row, thus Canada#CY 2005, Canada#CY 2006 and Canada#CY 2007. Then, for each sub-cell, for instance, Alberta#CY 2005, we only need to search in its parent cell, which is Canada#CY 2005.

#### 4.4.1 Finding the Touched Cells

Finding the touched cells for all the sub-cells in the slice is similar to locating the parent cell for the subset query. The difference is there might be multiple sub cells. Therefore, we should find the parent cells for each of the sub cells. And obviously, locating the parent cell in cache efficiently will definitely improve our dill-down performance since we can narrow down the search domain by finding the right parent cells. And again, the FindParentCell works fast in the drill-down queries. This is because most graphic UI cannot display a table or cube with more than 2 dimensions. Therefore, our algorithm would be efficient for the drill down operations.

### 4.4.2 Grouping the Sub-cells

Based on the `FindParentCell` method, now we can decide the cached parent cell for every sub-cell when drill-down to an arbitrary depth. Next, for the purpose of efficiency, it is beneficial to group the sub-cells together if their parent cell is the same. The reason is that we want to scan the dataset in their parent cell once, and we can extract the measure values for every sub-cell at the same time. In most cases, the parent cell contains large amount of data, and therefore, reducing the scan times can significantly reduce the cache misses and therefore save the total drill-down time.

Algorithm 9 shows how we group the sub-cells and bind them together to the same parent cell.

---

**Algorithm 9** GroupSubCells(rowGroup, columnGroup)
 

---

```

1: Initialize var matchedCells = new HashMap();
2: for var i = 0; i < rowGroup.length; i++ do
3:   for var j = 0; j < columnGroup.length; j++ do
4:     var dimensions = [];
5:     dimensions = [rowGroup[i], columnGroup[j]];
6:     var parentCell = FindParentCell(dimensions, 0);
7:     if parentCell != undefined and parentCell != null then
8:       if parentCell.contains(parentCell) then
9:         var subCells = matchedKeys.get(parentCell);
10:        subCells.push(dimensions);
11:      else
12:        var subCells = [];
13:        subCells.push(dimensions);
14:        matchedKeys.put(parentCell, subCells);
15:      end if
16:    end if
17:  end for
18: end for
19: return matchedCells

```

---

Here, we use a hash map, `matchedCells`, to group the sub-cells under their parent cell.

### 4.4.3 Extracting the Data

After we finish grouping the sub-cells, the next step is to extract the portions of data for each sub-cells in the parent cell.

Extracting the data for the sub-cells is similar to the one in section 4.2.2. And our algorithm outperforms the previous drill-down algorithm in Elahe's system [9], because the previous one needs

to scan the whole dataset while our algorithm only scans the slice of touched cells.

#### 4.4.4 Evaluation

As stated in the previous sections, drill-down operations can be viewed as expanding a slice of a cube. And compared to the previous version, our algorithm does not need to scan the whole dataset. Our drill-down only needs to search the cells inside the slice. Now, we will test our drill-down on different number of cells in total and fixed sub-cubes. We use the example in figure 4.1 and every time, we download one more cell and choose to expand the *Canada* row only. Below is the workflow we use to evaluate the performance of our drill-down operations:

1. Download Cell *Canada#Accessories*, expand *Canada* row.
2. Download Cell *Australia, Canada#Accessories*, expand *Canada* row.
3. Download Cell *Australia, Canada, France#Accessories*, expand *Canada* row.
4. ....

In the previous implementation, every time, we expand the *Canada* row, it needs to scan the whole dataset. For example, it needs to scan all the entries in the (*Australia, Canada, France*) *#Accessories* aggregated cell. While in our mechanism, we only need to scan the entries in cell *Canada#Accessories*.

Figure 4.4 shows the performance between our hash based drill-down and the scanning whole dataset drill-down operations.

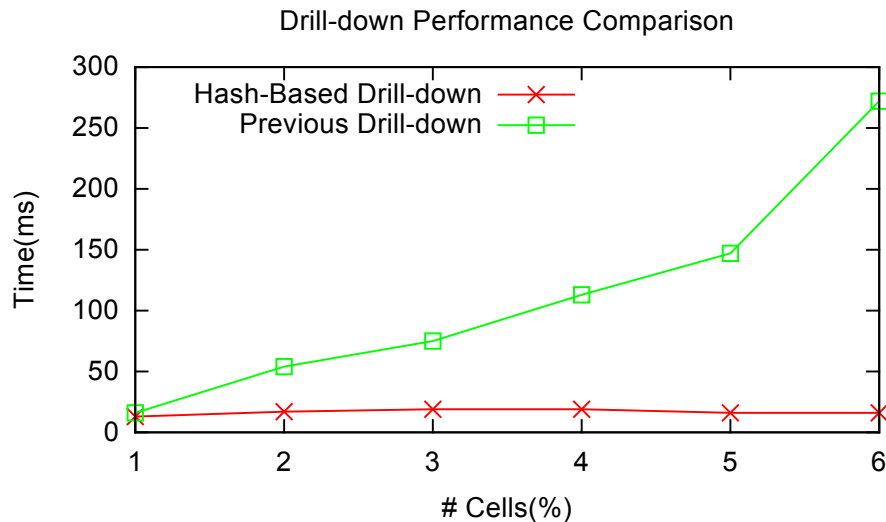


Figure 4.4: Drill Down Time Comparison

As illustrated in fig 4.4, the performance of our proposed algorithm is significantly better than previous one in terms of running time. Particularly, our drill-down time is only related to the number of cells touched. While in the previous drill-down method, no matter how many cells are in the slice, it still needs to scan the whole dataset.

Usually, users will issue multiple drill-down requests after they see the result table for the initial query. And the main purpose of our hash based caching mechanism is to let the users benefit the subsequent drill-down or roll-up queries by downloading the inflated sub-cubes. Section 6.3 will present the performance evaluation for some reasonable user workflows. And we will do the comparisons between the previous method and the one combining our downloading inflated sub-cubes in background idea.

## Chapter 5

# Downloading the Inflated Sub-cube in Background

As illustrated earlier, our system downloads the inflated answer sub-cubes to speed up the future queries, because the user may explore the downloaded result table by requesting different cross-tab or drill-down operations. And based on the statistics in Elahe's thesis [9], this mechanism works well when the size of the inflated sub-cube is not so large. While if the size of the inflated sub-cube is large, the cost of the load time may not overcome several drill-down requests or even more than sending the queries to the server and getting the results back.

### 5.1 User Access Pattern – Serve the Users First

Meanwhile, as for users, they tend to see the specified (aggregated) result immediately, do some analysis and then decide to choose which slice of cells to drill-down to. Also, it would usually take several seconds for the users to think while they're viewing the specified result data.

Based on this observation, we realize that it is a better idea to download the result at the specific level for their initial query without diving deep into the leaf level and then utilize the time quantum of the users' *think time* to download the inflated answer sub-cubes in background and store them in the cache. By applying this idea, we can not only meet the users' need at the very first time, but also have the inflated answer sub-cubes ready in cache in background. The users would feel no difference compared to downloading only the specified sub-cube for their first query, but they will gain much benefit for the future cross-tab or drill-down operations since the inflated sub-cube is cached so that all the subsequent requests could be served locally.

## 5.2 Implementation

The implementation for the idea of downloading inflated sub-cube in background is simple. Every time, when the users request some sub-cubes at the internal level, we will generate two queries: one for the answer (current level) sub-cubes and the other for the inflated (leaf level) sub-cubes. For example, if the user wants to see the Internet Sales Amount on the intersection with [Customer].[Geography].[Country].[Canada] and [Products].[Categories].[Accessories]. The query for downloading the answer sub-cubes is:

```
select [Measures].[Internet Sales Amount] on axis(0),
[Customer].[Customer Geography].[All Customers].[Canada] on axis(1) ,
[Product].[Product Categories].[All Products].[Accessories] on axis(2)
from [Adventure Works]
```

while the query for downloading the inflated sub-cubes for the leaf level is:

```
select [Measures].[Internet Sales Amount] on axis(0),
DESCENDANTS([Customer].[Customer Geography].[All Customers].[Canada], 4,
LEAVES) on axis(1),
DESCENDANTS([Product].[Product Categories].[All Products].[Accessories],
2, LEAVES) on axis(2)
from [Adventure Works]
```

Once we get the data for the specified level and present the result to the user, we'll download the inflated answer sub-cube in background immediately. Table 5.1 presents the query processing strategies between the previous method and our hash based mechanism combining with the downloading inflated sub-cubes in background idea.

We can see that, under the previous method, the system will try to download the inflated sub-cubes for the user's initial query and therefore, the user may need to wait a long time before they can see the result table. While in our implementation, our system downloads the answer sub-cubes first and download the inflated sub-cubes in background when the user is looking at the result table and deciding which row, column or cell to expand (drill-down). So the user will see the fact table at the very first time. And our system will have the inflated sub-cubes ready before the users issue their subsequent drill-down queries. Therefore, the users will also gain the benefit for their subsequent requests. Besides, our drill-down operations has faster response time compared to the previous method.

Our evaluation result shows that this new strategy could significantly save the time for the initial downloading time and even when size of the inflated sub-cubes is large, it still outperforms both the downloading inflated sub-cubes directly and the one that just sends queries to retrieve the answer sub-cubes to the server every time if enough quantum of *thinking time* is given.

Table 5.1: Query Processing Comparison

(a) Previous Mechanism

Steps	Users	Previous Mechanism
Step 1	Submit a Query	Download Inflated Sub-cubes
Step 2	Read the Answer	
Step 3	Issue Drill-down Queries	Search in the Aggregated Dataset – Slow Response Time

(b) Hash Based Mechanism

Steps	Users	Downloading Inflated Sub-cubes In Background
Step 1	Submit a Query	Download Initial Answer Sub-cubes
Step 2	Read the Answer	Download Inflated Sub-cubes
Step 3	Issue Drill-down Queries	Search in the Touched Cells Only – Faster Response Time



# Chapter 6

## User Workflow

In the previous chapters, we talked about the whole implementation of our hash based caching mechanism. As we stated earlier that we choose to download the inflated sub-cubes for the user's initial query. And compared to the method of downloading the answer sub-cubes from the server side for every request, downloading the inflated sub-cubes at one time and sever the user by utilizing the local cached data is beneficial if the user is going to perform enough number of subsequent cross-tab or drill-down operations from the first result table since we can save the round-trip networking communication time by aggregating the local data.

### 6.1 Performance Factors

Our system performance depends on how much the user want to explore the first result table. In other words, it depends on how many number of cross-tab or drill-down operations the user will issue under the first query, since almost for every subsequent drill-down query, aggregating the local inflated sub-cubes data is always faster than the issuing a new request to the server to download the answer sub-cubes. The more subsequent drill-down requests is issued, the more benefit the user will gain. However, meanwhile, we should pay the time to download the inflated sub-cubes. And if the size of the inflated sub-cubes is large, the user may need to wait a long time before the inflated sub-cubes are ready in client-side. Besides, if the size of the inflated sub-cubes is large, our data aggregation time slows down since we have to scan more data to extact the needed rows.

In summary, our system performance is related to:

1. the size of the inflated sub-cubes
2. the number of subsequent drill-downs issued by the user
3. the data aggregation time (or script time)

## 6.2 Workflow Scenario

Thus, we decide to compare our system performance between the downloading the inflated sub-cubes and downloading answer sub-cubes given reasonable number of roll-up and drill-down operations for different size of inflated sub-cubes. And table 6.1 presents a sample test case in our workflows, see appendix C for details.

	Previous Method - Download inflated sub-cubes	Download inflated sub-cubes in background	Answer sub-cube
<b>Workflow</b>	<b>Script time</b>	<b>Script time</b>	<b>Load time</b>
Download [Customer].[Education], [Date].[2005]&[Date].[2006]&[Date].[2007] &[Date].[2008]&[Date].[2010]	Load time (size)	Load time (size)	
	366 (53KB)	74 (53KB)	78
Drill down on [Customer].[Education].[Bachelors]	7	4	88
Roll up on [Customer].[Education]	1	1	84
Drill down on [Date].[2005].[H2]	3	2	71
Drill down on [Date].[2008].[H2]	4	2	65
Drill down on [Date].[2008].[H2].[Q3]	3	2	75
Drill down on [Date].[2008].[H2].[Q3].[July]	1	1	68

Table 6.1: Sample Case for User Workflow

In this table, we compare the previous method (download the inflated sub-cubes directly), our downloading inflated sub-cubes in background method (which is discussed in details in chapter 5) and the way that download the answer sub-cubes for every query. For the previous method, the load time is the time for downloading the inflated sub-cubes in the leaf level for the initial query. And the size is 53KB. For the downloading inflated sub-cubes in background method, the load time is the time to download the answer sub-cubes for the initial query. Users can see the result table in the client side at the very first time as long as the client side receive the result for the first query without waiting for the inflated sub-cubes are ready in the cache. And for the downloading answer sub-cubes method, we just calculate the answer sub-cubes download time for every query.

In this sample, we submit 1 roll-up query and 5 drill-down queries. For both the first two methods, all these subsequent queries are served locally. We extract the aggregated result from the inflated sub-cubes without sending a request to the server side. While for the third method, we measure the answer sub-cubes download time for every request.

From this sample case, we can see that for every roll-up or drill-down query, the script time is much less than the time of downloading the answer sub-cubes. Besides, under our hash based mechanism, our drill-down time is less than the one in the previous framework. Our system provide faster drill-down operations by scanning the touched cells only, which is also discussed in section 4.4.

### 6.3 Evaluation

Table 6.2 summarizes the results of the workflow experiment. The `Load Time` in this table means the time to download the inflated sub-cubes. And the `Script Time` represents the total time for all the drill-down or roll-up operations. While the `Initial Answer Sub-cube Time` is the time for downloading the answer sub-cubes for the initial query in our implementation. The `Total Download Time` means the total time to retrieve the answer sub-cubes from the server. In each session, we give reasonable number of drill-downs depending on the size of the inflated sub-cubes. We compare the original caching mechanism and the one combined with our downloading the inflated sub-cubes in background idea.

	Inflated Sub-cube Size (KB)	# Drill-downs	Inflated Answer Sub-cube(ms)			With Cache & Download Inflated Sub-cube in background(ms)			Answer Sub-cube(ms)
			Load Time	Script Time	Total	Initial Answer Sub-cube Time	Load Time	Script Time	Total Download Time
1	12	3	137	6	143	86	142	5	294
2	21	5	184	23	207	74	197	19	474
3	53.2	6	366	19	385	74	401	12	529
4	134.2	6	562	48	610	70	583	21	593
5	472.4	6	1632	76	1708	69	1747	30	573
6	802.3	14	2162	255	2417	73	2435	171	1210
7	1638.4	16	4329	428	4757	67	4729	283	1418

Table 6.2: Workflow Summary

Obviously, if given reasonable quantum of time, we can have the inflated sub-cubes ready in

background so that the user can benefit a lot for their subsequent requests. And also we can have the result available at the very first time for the user's initial query. This is quite important for the user's experience, because most users tend to see the result as fast as possible. Besides, by downloading the inflated sub-cubes, we can provide all the results for the users' subsequent drill-down or roll-up operations. This idea is even more beneficial if the network is not stable or the users are suffering an offline scenario and cannot get access to the network. In traditional implementation, the users cannot see any result in such cases. Which with our mechanism, they can still see the results if the corresponding cells are cached. In other words, we support the drill-downs under the user control.

Meanwhile, if the size of the inflated sub-cubes is really large that we cannot download it within the user's *thinking time*, it might not be a good idea to download the inflated sub-cubes, since the user needs to wait more time until the inflated sub-cubes are ready before their future drill-down operations. And the time cost may exceed the total time for downloading the answer sub-cubes every time. See case 7 in table 6.2.

In conclusion, if the size of the inflated sub-cubes is really large, we should cautious about downloading the inflated sub-cube. At those scenarios, directly downloading the answer sub-cubes may be a better idea. Otherwise, our mechanism can always provide better user experience compared to the previous method and the one that downloads the result from server every time.

## Chapter 7

# Conclusion and Future Work

### 7.1 Conclusion

In this thesis, we focus on the client-side caching for web OLAP system and we proposed a hash based new mechanism that can handle arbitrary *shapes* of sub-cubes in the client's memory. This new mechanism has great performance in downloading the inflated answer sub-cube compared to the no-cache one and also works efficiently in the subsequent drill-down requests compared to the previous version.

By using the hash based caching mechanism, we formalize the problem to a searching problem for the un-cached cells. We stated that the optimal solution would be coming up with the minimum number of query cells using the least amount of time. And we also study the trade-off among our *Sort-Merge* algorithm, the Brute-Force search algorithm and the No-aggregation way for the client-side processing time and the data download time + split time. Our result indicates that our *Sort-Merge* algorithm has the best performance among the three.

Besides, we categorize the queries issued by the users to 3 different types and propose different strategies for each of them respectively. Additionally, we propose faster methods for the drill-down operations compared to the previous version.

We also come up with the idea of downloading the specified level of data first and downloading the inflated sub-cubes in background. By combining this idea with our hash based caching mechanism, we can not only serve the users at the very first time, but also provide the high performance for the subsequent cross-tab or drill-down operations by utilizing the inflated sub-cubes downloaded in background. The workflow shows that if given enough amount of time to download the inflated sub-cubes in background, our hash based caching mechanism always outperforms the one that downloads the answer sub-cubes from the server every time.

## 7.2 Future Work

There are several potential directions which may further extend our results. First, our mechanism lacks the swapping management. For instance, if the cache size exceeds the limit of the system, the system should be able to empty some cells so that there would be enough space to hold the new ones. We can apply the caching idea from the page replacement in operating system. Some algorithms like the LRU, FIFO or WSclock and etc. And in our mechanism, it is easy to add the timestamp and size for each of the cached cells. Based on these statistics, we can easily apply the existing replacement algorithms.

Another idea is to design a better algorithm than the *Sort-Merge* algorithm. As we stated in previous chapters, though our *Sort-Merge* algorithm runs fast, it is not guaranteed to find the optimal solution. Based on the hash based caching mechanism, one can explore more efficient algorithms that runs faster as well as coming up with better result (less query cells).

Besides, there actually exists another type of query the users may issue: the query that may cover parts of the cells in our hash table. More specifically, the cached cells in our hash table is a subset of the issued query. So the better way to cope with this type of query is to find out the parts already cached and just download the un-cached parts and then perform a cell integration to generate a large cell and remove the pre-cached sub cells.

Finally, more study could be put the user access patterns. For example, what the users prefer to see for their initial request and what directions or how many drill-downs can be issued based on the initial table. The more precise we know about the user's access pattern, the better we can improve our algorithm.

# Bibliography

- [1] Online analytical processing, 2014. [http://en.wikipedia.org/wiki/Online\\_analytical\\_processing](http://en.wikipedia.org/wiki/Online_analytical_processing). 7
- [2] Xml for analysis, 2014. [http://en.wikipedia.org/wiki/XML\\_for\\_Analysis](http://en.wikipedia.org/wiki/XML_for_Analysis). 8
- [3] E.F. Codd, S.B. Codd, and C.T. Salley. *Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate*. Codd & Associates, 1993. 7
- [4] Microsoft Corporation and Hyperion Solutions Corporation. Xml for analysis specification, 2012. <http://msdn.microsoft.com/en-us/library/ms977626.aspx>. 8
- [5] Shaul Dar, Michael J. Franklin, Björn Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases, VLDB '96*, pages 330–341, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. 3
- [6] C. Darie. *AJAX and PHP: building responsive web applications*. 2006. 4
- [7] Björn Jónsson, María Arinbjarnar, Bjarnsteinn, Michael J. Franklin, and Divesh Srivastava. Performance and overhead of semantic cache management. *ACM Trans. Internet Technol.*, 6(3):302–331, Aug 2006. 2, 3
- [8] Panos Kalnis and Dimitris Papadias. Proxy-server architectures for olap. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, SIGMOD '01*, pages 367–378, New York, NY, USA, 2001. ACM. 3
- [9] Elaheh Kamaliha. Web-based olap. 2013. 2, 3, 10, 12, 38, 39, 42
- [10] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(1):035–047, January 1996. 3
- [11] Yannis Kotidis and Nick Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, SIGMOD '99*, pages 371–382, New York, NY, USA, 1999. ACM. 3
- [12] J.A. O'Brien and G.M. Marakas. *Management Information Systems*. McGraw-Hill/Irwin, 2011. 7
- [13] Thomas Phan and Wen-Syan Li. Dynamic materialization of query views for data warehouse workloads. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, ICDE '08*, pages 436–445, Washington, DC, USA, 2008. IEEE Computer Society. 3

- [14] Peter Scheuermann, Junho Shim, and Radek Vingralek. Watchman: A data warehouse intelligent cache manager. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 51–62, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. 3
- [15] P. Larson T. Palpanas and J. Goldstein. Cache management policies for semantic caching. In *Technical Report CSRG-439*, 2001. 3
- [16] YUI Team. Yui 2: Treeview, 2012. <http://developer.yahoo.com/yui/treeview/>. 6, 7, 8
- [17] Wo-Shun Luk Tim Hsiao and Stephen Petchulat. Data visualization on web-based olap. In *ACM 14th International Workshop on Data Warehousing and OLAP (DOLAP)*, Glasgow, October 2011. 1, 10
- [18] Microsoft SQL Server 2008 Analysis Services Unleashed. *Irina Gorbach and Alexander Berger and Edward Melomed*. Sams, 2008. 7



# Appendix A

## Partial XMLA Response

```
<return xmlns="urn:schemas-microsoft-com:xml-analysis">
...
<OlapInfo>
  <CubeInfo>
    <Cube>
      <CubeName>Adventure Works</CubeName>
      <LastDataUpdate xmlns="http://schemas.microsoft.com/analysisisservices/2003/engine">2011-03-25T21:16:58.49</
        LastDataUpdate>
      <LastSchemaUpdate xmlns="http://schemas.microsoft.com/analysisisservices/2003/engine">2011-03-25T20
        :53:05.976667</LastSchemaUpdate>
    </Cube>
  </CubeInfo>
  <AxesInfo>
    <AxisInfo name="Axis0">
      <HierarchyInfo name="[Measures]">
        <UName name="[Measures].[MEMBER_UNIQUE_NAME]" type="xsd:string" />
        <Caption name="[Measures].[MEMBER_CAPTION]" type="xsd:string" />
        <LName name="[Measures].[LEVEL_UNIQUE_NAME]" type="xsd:string" />
        <LNum name="[Measures].[LEVEL_NUMBER]" type="xsd:int" />
        <DisplayInfo name="[Measures].[DISPLAY_INFO]" type="xsd:unsignedInt" />
      </HierarchyInfo>
    </AxisInfo>
    <AxisInfo name="Axis1">
      <HierarchyInfo name="[Customer].[Marital Status]">
        <UName name="[Customer].[Marital Status].[MEMBER_UNIQUE_NAME]" type="xsd:string" />
        <Caption name="[Customer].[Marital Status].[MEMBER_CAPTION]" type="xsd:string" />
        <LName name="[Customer].[Marital Status].[LEVEL_UNIQUE_NAME]" type="xsd:string" />
        <LNum name="[Customer].[Marital Status].[LEVEL_NUMBER]" type="xsd:int" />
        <DisplayInfo name="[Customer].[Marital Status].[DISPLAY_INFO]" type="xsd:unsignedInt" />
      </HierarchyInfo>
      ...
    <Axis name="Axis0">
  <Tuples>
    <Tuple>
      <Member Hierarchy="[Measures]">
```

```

        <UName>[Measures].[Internet Sales Amount]</UName>
        <Caption>Internet Sales Amount</Caption>
        <LName>[Measures].[MeasuresLevel]</LName>
        <LNum>0</LNum>
        <DisplayInfo>0</DisplayInfo>
    </Member>
</Tuple>
</Tuples>
</Axis>
    ...
    <CellData>
    <Cell CellOrdinal="0">
        <Value xsi:type="xsd:decimal">7267018.3655</Value>
        <FmtValue>$7,267,018.37</FmtValue>
    </Cell>
    <Cell CellOrdinal="1">
        <Value xsi:type="xsd:decimal">7546600.3097</Value>
        <FmtValue>$7,546,600.31</FmtValue>
    </Cell>
    <Cell CellOrdinal="2">
        <Value xsi:type="xsd:decimal">7920357.3733</Value>
        <FmtValue>$7,920,357.37</FmtValue>
    </Cell>
    <Cell CellOrdinal="3">
        <Value xsi:type="xsd:decimal">6624701.1722</Value>
        <FmtValue>$6,624,701.17</FmtValue>
    </Cell>
    </CellData>
</root>
</return>

```

# Appendix B

## Tables for Data Split Experiment

Table B.1: Data Split Experiment for 2-Dimensional Data

Dimension 1	Dimension 2	#Cells	#Children	#Rows	Size(KB)	Split Time(ms)
[Geography].[Australia]	[Products].[Categories].[Accessories]	1	1400	880	10.7	4
[Geography].[Canada]	[Products].[Categories].[Bikes]	1	9125	8468	114.1	9
[Geography].[US]	[Products].[Categories].[Bikes]	1	47000	43616	627	36
[Geography].[Australia]&[France]	[Products].[Categories].[Accessories]	2	3080	1936	24.1	6
[Geography].[Australia]&[France]	[Products].[Categories].[Bikes]	2	11000	10208	138.8	12
[Geography].[Canada]&[US]	[Products].[Categories].[Bikes]	2	56125	52084	750.2	59
[Geography].[Australia]&[Canada]	[Products].[Categories].[Accessories] & [Clothing]	4	9379	4746	60.2	11
[Geography].[Australia]&[Canada]	[Products].[Categories].[Accessories] & [Bikes]	4	18080	15594	216.8	20
[Geography].[US]&[Canada]	[Products].[Categories].[Accessories] & [Bikes]	4	71840	61962	890.7	74
[Geography].[Australia]&[Canada] & [France] & [Germany]	[Products].[Categories].[Accessories] & [Clothing]	8	18758	9492	125.3	26
[Geography].[Canada]&[France] & [Germany] & [US]	[Products].[Categories].[Accessories] & [Clothing]	8	46646	23604	319.9	63
[Geography].[Canada]&[France] & [Germany] & [US]	[Products].[Categories].[Accessories] & [Bikes]	8	89920	77556	1115.1	138

Table B.2: Data Split Experiment for 3-Dimensional Data

Dimension 1	Dimension 2	Dimension 3	#Cells	#Children	#Rows	Size(KB)	Split Time(ms)
[Geography].[Australia]	[Products].[Categories].[Accessories]	[Customer].[Education].[All Customers]	1	1400	880	10.7	4
[Geography].[Australia]	[Products].[Categories].[Bikes]	[Customer].[Education].[All Customers]	1	25000	22840	304.3	19
[Geography].[Canada]	[Products].[Categories].[Bikes]	[Customer].[Education].[All Customers]	1	45625	41683	563.6	31
[Geography].[US]	[Products].[Categories].[Bikes]	[Customer].[Education].[All Customers]	1	235000	214696	3068.3	159
[Geography].[Australia]&[France]	[Products].[Categories].[Accessories]	[Customer].[Education].[All Customers]	2	15400	9680	120.3	18
[Geography].[Canada]&[US]	[Products].[Categories].[Accessories]	[Customer].[Education].[All Customers]	2	78575	49390	644.2	73
[Geography].[Canada]&[US]	[Products].[Categories].[Bikes]	[Customer].[Education].[All Customers]	2	280625	256379	3684.4	292
[Geography].[Australia]&[Canada]	[Products].[Categories].[Accessories]&[Clothing]	[Customer].[Education].[All Customers]	4	46895	23730	302.8	39
[Geography].[Australia]&[Canada]	[Products].[Categories].[Accessories]&[Bikes]	[Customer].[Education].[All Customers]	4	90400	76953	1042.8	84
[Geography].[US]&[Canada]	[Products].[Categories].[Accessories]&[Bikes]	[Customer].[Education].[All Customers]	4	359200	305769	4396.1	352
[Geography].[Australia]&[Canada]&[France]&[Germany]	[Products].[Categories].[Accessories]&[Clothing]	[Customer].[Education].[All Customers]	8	93790	47460	611.7	112
[Geography].[Canada]&[France]&[Germany]&[US]	[Products].[Categories].[Accessories]&[Clothing]	[Customer].[Education].[All Customers]	8	233230	118020	1595.6	280
[Geography].[Canada]&[France]&[Germany]&[US]	[Products].[Categories].[Accessories]&[Bikes]	[Customer].[Education].[All Customers]	8	449600	382722	5522.3	623

Table B.3: Data Split Experiment for 4-Dimensional Data

Dimension 1	Dimension 2	Dimension 3	Dimension 4	# Cells	#Children	#Rows	Size(KB)	Split Time(ms)
[Geography],[Australia]	[Products],[Categories],[Accessories]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	1	1400	880	10.7	4
[Geography],[Australia]	[Products],[Categories],[Bikes]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	1	150000	137040	1923.7	113
[Geography],[Canada]	[Products],[Categories],[Bikes]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	1	273750	250098	3590.5	193
[Geography],[US]	[Products],[Categories],[Bikes]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	1	1410000	1288176	19291.2	1119
[Geography],[Australia]&[France]	[Products],[Categories],[Accessories]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	2	92400	58080	757.6	61
[Geography],[Canada]&[US]	[Products],[Categories],[Accessories]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	2	471450	296340	4091.9	447
[Geography],[Canada]&[US]	[Products],[Categories],[Bikes]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	2	1683750	1538274	23258.9	1872
[Geography],[Australia]&[Canada]	[Products],[Categories],[Accessories]&[Clothing]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	4	281370	142380	1929.0	230
[Geography],[Australia]&[Canada]	[Products],[Categories],[Accessories]&[Bikes]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	4	542400	461718	6678.3	550
[Geography],[US]&[Canada]	[Products],[Categories],[Accessories]&[Bikes]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	4	2155200	1834614	27802.0	2316
[Geography],[Australia]&[Canada]&[France]&[Germany]	[Products],[Categories],[Accessories]&[Clothing]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	8	562740	284760	3935.1	801
[Geography],[Canada]&[France]&[Germany]&[US]	[Products],[Categories],[Accessories]&[Clothing]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	8	1399380	708120	10068.3	2098
[Geography],[Canada]&[France]&[Germany]&[US]	[Products],[Categories],[Accessories]&[Bikes]	[Customer],[Education],[All Customers]	[Employee],[Departments],[Engineering]	8	2697600	2296332	35035.7	4328

## Appendix C

# User Workflow for Drill-downs

Below are the test cases we used for the drill-down tests.

Table C.1: User Workflow for Drill-downs

	Previous Method	Back-ground Download	Answer sub-cube
<b>Workflow</b>	<b>Script time</b>	<b>Script time</b>	<b>Load time</b>
Download [Customer].[Education].[All customers], [Date].[Calendar].[All Periods].[CY 2008]	Load time (size)	Load time (size)	
	137 (12KB)	86 (12KB)	85
Drill down on [Date].CY 2008].[H2]	2	2	68
Drill down on [Date].CY 2008].[H2].[Q3]	2	2	77
Drill down on [Date].CY 2008].[H2].[Q3].[July]	2	1	64
Download [Geography].[Australia].[New South Wales], [Date].[2006].[Q1]	Load time (size)	Load time (size)	
	184 (21KB)	74 (12KB)	71
Drill down on [Geography].[Australia].[New South Wales].[Alexandria]	8	7	70
Drill down on [Geography].[Australia].[New South Wales].[Alexandria].[2015]	3	2	81
Roll up on [Geography].[Australia].[New South Wales]	1	1	76
Drill down on [Date].[CY 2006].[Q1].[January]	6	5	87

*Continued on next page*

Table C.1 – Continued from previous page

Workflow	Script time	Script time	Load time
Drill down on [Date].[CY 2006].[Q1].[February]	5	4	89
Download [Customer].[Education], [Date].[2005]&[Date].[2006]&[Date].[2007]&[Date].[2008]	Load time (size)	Load time (size)	
	366 (53KB)	74 (53KB)	78
Drill down on [Customer].[Education].[Bachelors]	7	4	88
Roll up on [Customer].[Education]	1	1	84
Drill down on [Date].[2005].[H2]	3	2	71
Drill down on [Date].[2008].[H2]	4	2	65
Drill down on [Date].[2008].[H2].[Q3]	3	2	75
Drill down on [Date].[2008].[H2].[Q3].[July]	1	1	68
Download [Geography]. [Canada].[Alberta]&[Geography]. [Canada].[British Columbia]&[Geography]. [Canada].[Ontario], [Date].[2008].[H1]&[Date].[2008].[H2]	Load time (size)	Load time (size)	
	562 (134KB)	70 (134KB)	77
Drill down on [Date].[2008].[H1].[Q1]	12	5	79
Drill down on [Date].[2008].[H1].[Q1].[January]	7	4	137
Drill down on [Date].[2008].[H2].[Q3]	6	3	69
Drill down on [Date].[2008].[H2].[Q3].[July]	2	2	66
Drill down on [Geography].[Canada].[British Columbia].[Burnaby]	10	4	82
Drill down on [Geography].[Canada].[British Columbia].[Burnaby].[V5A 3A6]	11	3	83
Download [Employee].[Department].[Document Control]&[Engineering]&[Executive]&[Facilities and Maintenance]&[Finance], [Date].[2005]&[2006]&2007]&[2008]	Load time (size)	Load time (size)	
	1632 (472KB)	69 (472KB)	79
Drill down on [Date].[2005].[H2]	10	6	113
Drill down on [Date].[2005].[H2].[Q3]	10	5	83

Continued on next page

Table C.1 – Continued from previous page

<b>Workflow</b>	<b>Script time</b>	<b>Script time</b>	<b>Load time</b>
Drill down on [Date].[2007].[H2]	15	8	68
Drill down on [Date].[2007].[H2].[Q4]	11	4	71
Drill down on [Employee].[Department].[Engineering].[Design Engineer]	15	3	87
Drill down on [Employee].[Department].[Engineering].[Finance].[Finance Manager]	15	4	72
Download [Geography].[Australia]&[Canada]&[France], [Date].[2006]	Load time (size)	Load time (size)	
	2162 (802.3KB)	73 (802.3KB)	69
Drill down on [Geography].[Canada].[British Columbia]	32	20	98
Drill down on [Geography].[Canada].[British Columbia].[Burnaby]	25	29	67
Roll up on [Geography].[Canada]	1	1	77
Drill down on [Geography].[France].[Essonne]	30	25	107
Drill down on [Geography].[France].[Essonne].[Morangis]	22	7	75
Drill down on [Geography].[France].[Essonne].[Morangis].[91420]	20	3	61
Roll up on [Geography].[France]	1	1	79
Drill down on [Geography].[France].[Yveline]	29	22	106
Roll up on [Geography].[France]	1	1	87
Drill down on [Geography].[Australia].[Victoria]	26	8	57
Drill down on [Geography].[Australia].[Victoria].[Melbourne]	20	13	59
Roll up on [Geography].[Australia]	1	1	63
Drill Down on [Date].[2006].[H1]	26	21	65
Drill Down on [Date].[2006].[H1].[Q1]	21	19	135
Download [Geography].[Australia]&[Canada]&[France], [Date].[2006]&[2007]	Load time (size)	Load time (size)	

Continued on next page



Table C.1 – Continued from previous page

<b>Workflow</b>	<b>Script time</b>	<b>Script time</b>	<b>Load time</b>
	4329 (1638.4KB)	67 (1638.4KB)	70
Drill down on [Geography].[Canada].[British Columbia]	36	35	93
Drill down on [Geography].[Canada].[British Columbia].[Burnaby]	38	54	69
Roll up on [Geography].[Canada]	1	1	81
Drill down on [Geography].[France].[Essonne]	41	34	97
Drill down on [Geography].[France].[Essonne].[Morangis]	14	6	79
Drill down on [Geography].[France].[Essonne].[Morangis].[91420]	8	4	63
Roll up on [Geography].[France]	1	1	75
Drill down on [Geography].[France].[Yveline]	47	39	103
Roll up on [Geography].[France]	0	1	82
Drill down on [Geography].[Australia].[Victoria]	16	15	64
Drill down on [Geography].[Australia].[Victoria].[Melbourne]	13	14	58
Roll up on [Geography].[Australia]	1	1	61
Drill Down on [Date].[2006].[H1]	81	27	63
Drill Down on [Date].[2006].[H1].[Q1]	39	12	137
Drill Down on [Date].[2007].[H1]	67	21	63
Drill Down on [Date].[2007].[H1].[Q1]	25	18	123